Part III

A Viewpoint Catalog

This page intentionally left blank

15

INTRODUCTION TO THE VIEWPOINT CATALOG

Part III is a catalog of our seven core viewpoints: Context, Functional, Information, Concurrency, Development, Deployment, and Operational. There are many options for structuring an architectural description, but we believe that this set of viewpoints does a good job of partitioning the AD into a manageable number of sections, while ensuring widespread coverage of concerns.

Figure 15–1 shows the relationships between views created using these viewpoints.

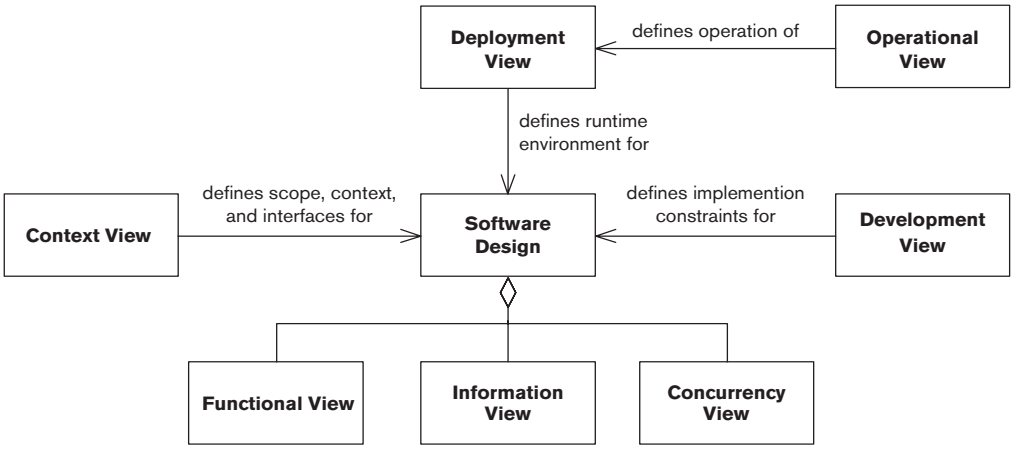

FIGURE 15–1 VIEW RELATIONSHIPS

For convenience, in Table 15–1 we reiterate the viewpoint taxonomy we presented originally in Part I.

For each viewpoint, we present the following details:

- The most important *concerns* addressed by the viewpoint, with an identification of the stakeholders who are most likely to be interested in its views
- The most important *models* you might build to present the views, together with the notations used and the activities for building them
- Some *problems and pitfalls* to be aware of and risk-reduction techniques for mitigating these
- A *checklist* of things to consider when developing the viewpoint and when reviewing it to help ensure correctness, completeness, and accuracy

Because of space limitations, we can present only an overview of some complex and detailed topics. Most of the chapters in Part III could easily expand into entire books in their own right. Our objective is to get you started, and to that end, each viewpoint chapter includes a number of references to sources of further information.

TABLE 15–1 VIEWPOINT CATALOG

Viewpoint	Definition
Context	Describes the relationships, dependencies, and interactions between the system and its environment (the people, systems, and external entities with which it interacts). The Context view will be of interest to many of the system's stakeholders and plays an important role in helping them to understand its responsibilities and how it relates to their organization.
Functional	Describes the system's runtime functional elements, their responsibilities, interfaces, and primary interactions. A Functional view is the cornerstone of most ADs and is often the first part of the description that stakeholders try to read. It drives the shape of other system structures such as the information structure, concurrency structure, deployment structure, and so on. It also has a significant impact on the system's quality properties such as its ability to change, its ability to be secured, and its runtime performance.
Information	Describes the way that the architecture stores, manipulates, manages, and distributes information. The ultimate purpose of virtually any computer system is to manipulate information in some form, and this viewpoint develops a complete but high-level view of static data structure and information flow. The objective of this analysis is to answer the big questions around content, structure, ownership, latency, references, and data migration.

TABLE 15–1 VIEWPOINT CATALOG *(CONTINUED)*

Viewpoint	Definition
Concurrency	Describes the concurrency structure of the system and maps functional elements to concurrency units to clearly identify the parts of the system that can execute concurrently and how this is coordinated and controlled. This entails the creation of models that show the process and thread structures that the system will use and the interprocess communication mechanisms used to coordinate their operation.
Development	Describes the architecture that supports the software development process. Development views communicate the aspects of the architecture of interest to those stakeholders involved in building, testing, maintaining, and enhancing the system.
Deployment	Describes the environment into which the system will be deployed and the dependencies that the system has on elements of it. This view captures the hardware environment that your system needs (primarily the processing nodes, network interconnections, and disk storage facilities required), the technical environment requirements for each element, and the mapping of the software elements to the runtime environment that will execute them.
Operational	Describes how the system will be operated, administered, and supported when it is running in its production environment. For all but the simplest systems, installing, managing, and operating the system is a significant task that must be considered and planned at design time. The aim of the Operational viewpoint is to identify system-wide strategies for addressing the operational concerns of the system's stakeholders and to identify solutions that address these.

This page intentionally left blank

16

THE CONTEXT VIEWPOINT

Definition	Describes the relationships, dependencies, and interactions between the system and its environment (the people, systems, and external entities with which it interacts)
Concerns	System scope and responsibilities, identity of external entities and services and data used, nature and characteristics of external entities, identity and responsibilities of external interfaces, nature and characteristics of external interfaces, other external interdependencies, impact of the system on its environment, and overall completeness, consistency, and coherence
Models	Context model, interaction scenarios
Problems and Pitfalls	Missing or incorrect external entities, missing implicit dependencies, loose or inaccurate interface descriptions, inappropriate level of detail, scope creep, implicit or assumed context or scope, overcomplicated interactions, overuse of jargon
Stakeholders	All stakeholders, but especially acquirers, users, and developers
Applicability	All systems

Many architecture descriptions we've seen focus on views that model the system's internal structures, data elements, interactions, and operation. Architects tend to assume that the "outward-facing" information—the system's runtime context, its scope and requirements, and so forth—is clearly and unambiguously defined elsewhere. In fact, in the first edition of this book, we didn't have a viewpoint for the system's context for this very reason. However, we have decided that we were wrong! In practice, it often isn't realistic to delegate all of these concerns elsewhere, and you frequently need to include a definition of the system's context as part of your architectural description. This can be for a number of reasons, including the following.

- The system context is often implicit rather than being explicitly defined as part of project initiation or requirements capture.
- The system context may be loosely defined during requirements analysis, but at a level of detail that means that you need to add significantly to it.
- You need to refer to elements of the system context elsewhere in your architectural description, which makes it desirable for this information to be part of the architectural description and so under your control.

In practice, most of the architectural descriptions we have created have included a "context diagram," which is essentially a view but without an associated viewpoint definition to guide its structure and content. We therefore decided that we should formalize the definition of context as we have for the other views.

The Context view of a system defines the relationships, dependencies, and interactions between the system and its environment—the people, systems, and external entities with which it interacts. It defines what the system does and does not do; where the boundaries are between it and the outside world; and how the system interacts with other systems, organizations, and people across these boundaries.

The Context view focuses on the outside world and usually represents the system itself as a "black box," hiding all details of its functional elements, data, implementation, and so forth, since these are documented in one of the other views.

CONCERNS

System Scope and Responsibilities

This concern considers the main responsibilities of the system, that is, what, in broad terms, it is required to do. For clarity, it may also identify some specific exclusions, although by definition, anything not listed here is excluded.

Note that this concern does not extend to a complete definition of the system's requirements, which is the responsibility of requirements analysis. Scope definition should be brief, succinct, and easily understood by all stakeholders without going into a lot of detail. It is usually defined in the form of a high-level list of the system's key capabilities or requirements, and it may also be useful to highlight some functional exclusions explicitly for the avoidance of doubt.

Clear definition and agreement of scope are vital early milestones of any system development project. Ideally the scope has already been defined for you, in which case you may limit yourself to summarizing it in the Context view and ratifying it with stakeholders as the AD develops. If the scope is not defined, you may need to do this yourself, again based on input from your stakeholders.

EXAMPLE The scope definition for a simple online retailer might include the following capabilities.

- Present the retailer's catalog to the user, including pictures and product specifications
- Provide a flexible search facility (search on product name, type, keyword, size, and so on)
- Accept orders for goods
- Accept payment by credit card (with asynchronous approval and notification to the customer)
- Provide automated interfaces into back-end systems for fulfillment

The exclusions for the first version of such a system might be:

- The ability to amend or cancel orders (this will need to be done over the telephone but is planned to be automated in a subsequent release)
- The ability to make payments by means other than credit card
- Display of live stock levels and the ability to reserve out-of-stock items

Identity of External Entities and Services and Data Used

An *external entity* is any system, organization, or person with which this system interacts in some way, for example:

- Another system that runs in the same organization as the system being modeled (we refer to these as "internal systems")
- Another system that runs in another organization (we refer to these as "external systems")
- A gateway or other implementation component that has the effect of hiding other systems (which may themselves be internal or external)
- A data store that is external to the system (for example, a shared database or data warehouse)
- A peripheral or other physical device that is external to the system (such as a shared messaging appliance or enterprise search engine)
- A user, a class of user, or some other person or role, such as operational or support staff

Each external entity will implement and offer some services, and manage and provide some data, that are used by this system. Similarly, each external entity will use some services and/or data offered by this one. External entities that do none of these things are not normally of interest.

Note that in this chapter we use the term *services* to refer to functionality that systems provide for each other. This important concept is relevant whether it is implemented by a formal service-oriented architecture (SOA), or some other, more traditional means such as messaging or file transfer.

Nature and Characteristics of External Entities

The quality properties of external entities, such as system stability and availability, performance and throughput capabilities, physical location, or data quality, may significantly affect the architecture of the system.

EXAMPLE A travel booking system exchanges information with many other systems located around the world. Some of these systems in more exotic locations may be only intermittently available, because of time zone differences or because they are more liable to failure. However, a failed communication with such a system might result in a customer's booking being lost, which is highly undesirable.

The travel system's interfaces with external systems will therefore need to be carefully designed. All failed interactions should be automatically retried a configurable number of times, and these retry attempts should be logged to a database so that operational staff can monitor trends. Interactions will need to be designed so that they can potentially be submitted multiple times without error (this is known as "idempotence"). It should be possible to restart very large transfers that fail partway through from the point of failure rather than having to retransmit the whole file.

The quality properties to be considered are exactly those properties that are defined in Part IV of the book, The Perspective Catalog.

However, it is only the "externally visible" properties that need to be considered—it is not normally necessary to consider the internal properties of external systems. For example, an external system may have some unreliable internal components but mask this to the outside world using load distribution techniques to give a high level of external availability. Similarly, you need only consider those interfaces that you will need to use—it is not necessary to understand or document every interface of each of your external entities.

It may be necessary to consider the "nature" of external entities that are not systems. For example, a user may not speak the primary language of the system, or a peripheral such as a shared fax gateway may have performance characteristics that need to be taken into consideration.

Identity and Responsibilities of External Interfaces

For each external entity, the nature of all interfaces between it and this system should be identified. Such an interface may serve one of the following purposes.

- *Data provider or consumer*: The external system supplies data directly to this system or receives data directly from it.
- *Service provider or consumer*: The external system is requested to perform some action by this system or requests some action of this system (e.g., a service call), and the service may return data and/or status information in response to the request.
- *Event provider or consumer*: The external system publishes events that this system wishes to be notified of, or this system publishes events that the external system wishes to be notified of.

For data provider and consumer interfaces, the concern identifies the content, scope, and meaning of the data to be transferred.

For service interactions, the concern identifies the semantics of the request (the nature of what is being requested and any parameters); the actions to be taken by the system fulfilling the request; any data to be returned; any acknowledgment, status, or error information that may be returned; and any exception actions to be taken by either side.

For event provider and consumer interfaces, the concern identifies the events of interest, their meaning and content, and the volume and likely timing of their occurrence.

It may be appropriate to go into more detail for more complex interactions between this system and external entities, such as a payment authorization which must be followed by a payment request.

Nature and Characteristics of External Interfaces

The quality properties of external interfaces may differ significantly from the quality properties of the systems at the other end. For example, there may be a low-bandwidth, relatively unreliable data link to a highly resilient system in another country. The interface is the constraining factor in this case and again will have a significant effect on the architecture of the system.

System characteristics include the following:

- The expected volumes—number of requests or transfers, size of data, seasonal fluctuations, and expected growth over time
- Whether interactions are scheduled (occurring at predefined times), occur in response to events, or are ad hoc

- Whether interactions are completely automated, completely manual (e.g., a user saves a file or sends an e-mail), or somewhere in between
- Whether interactions are transactional—that is, they are required to complete fully or not at all
- Criticality and timeliness—for example, a particular interaction that may be required to complete before the end of the business day in order to be captured by an auditing or accounting system
- Whether interactions are batch (large data sets transferred as a "unit"), message-based, or streaming in nature
- What level of security is required (authentication, authorization, confidentiality, and so on)
- The service level that can be expected of the interface (in terms of response time, latency, scalability, availability, and so on)
- The technical nature of the interface and what protocols are used (open standards or proprietary)
- Data and file formats

Again, you can use the material in Part IV (The Perspective Catalog) to frame your analysis.

Other External Interdependencies

There may be interdependencies between this system and external entities other than data flows or function invocations. These interdependencies may act in either direction—the system may be dependent on an external entity or vice versa. Such dependencies can be subtle and are sometimes hard to find.

This concern identifies the nature of the dependency and may also articulate its architectural impact—that is, what capabilities or features need to be built into the architecture in order for the dependency to be observed.

EXAMPLE An online retailer accepts orders for goods over the Internet through its main e-commerce system. However, to fulfill an order, this system has to interact with a separate payment system to collect payments, a customer account details system to make any updates to the customer's account (such as shipping addresses), and a fulfillment system that dispatches the goods.

From the perspective of the e-Commerce System, it is dealing with three separate independent systems and can treat them as such. However, as can be seen in Figure 16–1 there is a data dependency between two of them that in certain situations must be taken into account. The Fulfillment system

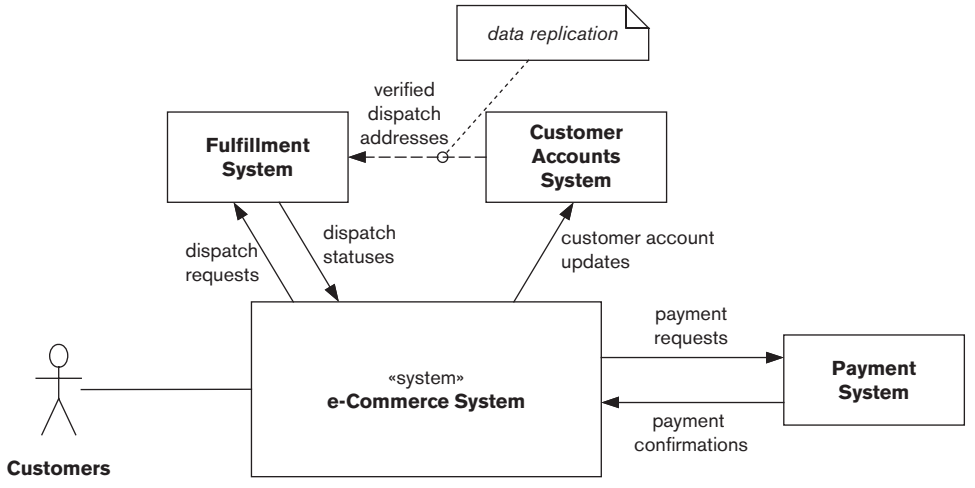

FIGURE 16–1 ONLINE RETAIL SYSTEM DEPENDENCIES

in this organization contains its own list of verified dispatch addresses for each customer, and it will reject orders that are not being sent to these addresses. However, this list is maintained by data replication from the Customer Accounts System. When the workflow for a customer order involves updating dispatch addresses, the e-Commerce System must take this dependency, and the latency of the replication, into account. Otherwise, orders may be rejected by the Fulfillment System because their dispatch addresses are not listed in its database.

The architectural impact in the case of this system might be to allow for resubmission to the Fulfillment System after a delay if a fulfillment request is rejected or to delay orders that have associated address updates to allow data replication to occur. (Interestingly, having talked about the need to understand the details of external interfaces, this is an example that bears this out: The tactic of resubmitting failed orders can be made much more efficient if the interface to the Fulfillment System allows the reason for failure to be reliably discerned from the dispatch status returned by that system.)

Impact of the System on Its Environment

This concern addresses the impact of the system's deployment on its environment, both within the organization in which it is deployed and externally. This includes the following:

- Any systems that are dependencies and so may require functional changes, interface changes, or performance or security improvements
- Any systems that will be decommissioned (switched off) as a result of this system's deployment
- Any data that will be migrated into this system

Although these changes may be someone else's responsibility, they should still be itemized to ensure that they are being addressed by someone and their progress tracked. (We return to this issue in our discussion of functional migration and data migration in Chapter 21.)

Overall Completeness, Consistency, and Coherence

In most cases this system will be part of something much larger: the overall "application landscape." This may even extend to systems distributed across multiple organizations and linked together over private or public networks. Such application landscapes can be very complex and are often poorly understood.

A key concern of your stakeholders (particular users) is that the overall end-to-end solution provides them with the functionality that they need in a sensible way, irrespective of which system provides a specific piece of functionality or manages a specific piece of data.

EXAMPLE In the early days of Internet shopping, retailers worked hard to get their catalogs onto the Internet in a pleasing and visually compelling way. The overriding concern was to get shoppers to visit their site rather than a competitor's. However, many of these retailers did not put nearly as much effort into the behind-the-scenes processes for accepting payment, fulfilling orders, or dealing with exceptions. As a result, they lost customer goodwill and gained a reputation for poor customer service, and in the most extreme cases they went out of business.

TABLE 16–1 STAKEHOLDER CONCERNS FOR THE CONTEXT VIEWPOINT

Stakeholder Class	Concerns
Acquirers	System scope and responsibilities, identity of external entities and services and data used, impact of the system on its environment
Assessors	All concerns
Communicators	System scope and responsibilities, identity and responsibilities of external entities, identity and responsibilities of external interfaces

TABLE 16–1 STAKEHOLDER CONCERNS FOR THE CONTEXT VIEWPOINT *(CONTINUED)*

Stakeholder Class	Concerns
Developers	All concerns
Production engineers	Nature and characteristics of external interfaces, impact of the system on its environment
System administrators	All concerns
Testers	All concerns
Users	System scope and responsibilities; identity of external entities and services and data used; overall completeness, consistency, and coherence

While this concern is more the responsibility of the enterprise architect than of the application architect (see Chapter 5), giving it some consideration will improve your likelihood of success, possibly significantly. An overall solution that hangs together in a consistent and coherent way is much more likely to delight your users than one that is fragmented and misaligned.

At a minimum you should ensure that the main business processes appear to have adequate coverage, with either systems or defined manual processes. Similarly, all of the data required for these processes should be stored somewhere (in this system or externally) and be accessible by those systems that need it.

Stakeholder Concerns

Typical stakeholder concerns for the Context viewpoint include those listed in Table 16–1.

MODELS

Context Model

The context model is the main architectural model within the Context view and often the only one produced. It places the system clearly in its environment and relates it to the external entities with which it interacts, via explicit relationships that represent the interfaces to and from it.

The purpose of the context model is to explain what the system does and does not do, to present an overall picture of the system's interactions with the outside world, and to summarize the roles and responsibilities of the participants in these interactions. This understanding is essential in order to make sure that all who are involved in the development of the system (and in making any necessary changes outside of it) know what they are responsible for and exactly where the boundaries are. This avoids potential duplication of development effort or, even worse, gaps or inconsistencies in the solution.

The context model has a wide audience, being of significant interest to all of the system's stakeholders. For this reason it should use simple, familiar terms, avoid business or technology jargon, and aim for simplicity without abstracting away so much information as to be worthless. It often uses business language to name and describe the elements within it and typically focuses on overall functionality and information flow, rather than the technologies used to implement them.

The context model is usually fairly high-level and abstract, answering the important "why" and "what" questions about the architecture. It does not specify in any detail how the system or its interfaces will be built; these questions are answered in the other architectural views.

The context model presents an overall picture of the system in its environment and typically includes the following types of elements:

- The *system* itself, represented as a black box, with its internal structure hidden, since the Context view is not concerned with how the system is built.
- The *external entities*, represented as black boxes for the same reason. (Indeed, it is likely that the internal details of external entities are not visible or known.) For each external entity, it is important to capture some key information, namely, the *name* of the entity, the *nature* of the entity (e.g., system, data store, person, group), the *owner* of the entity, and the *responsibilities* of the entity from the perspective of this system (the services, functions, and data upon which this system relies).
- The *interfaces* between the system and the external entities, presented at a summary level, highlighting the key data items or function invocations across the interface. Often all of the individual interfaces between the system and each external entity are "rolled up" into a single interface, to make the diagram easier to follow. For each external interface it is important to capture an overview of the *interactions* expected over the interface, the *semantics* of the interface (i.e., the data exchanged and its meaning), the *exception processing* approach that will be used when unexpected things happen, and the key *quality properties* of the interface upon which this system is relying. In many cases, you will just capture a short summary of this information in the context model and reference external sources of information for fuller descriptions.

The context model is a vital communication tool with a wide range of stakeholders from business and technology. It is often used to summarize "what the project is about," identify who the external partners are, and explain the interactions with them. Since it has a wide audience of varying degrees of business and technical expertise, it should be kept relatively simple, and the context diagram should fit on a single page if possible.

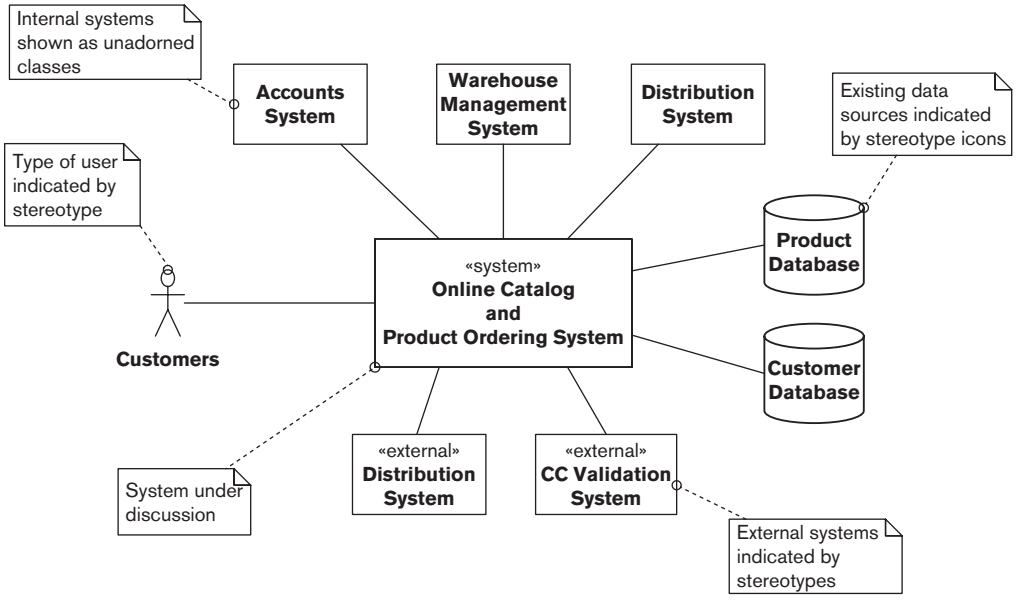

FIGURE 16–2 UML CONTEXT DIAGRAM

NOTATION The two notations that we commonly see used for context models are UML and "boxes–and–lines."

Unfortunately, the UML standard doesn't define a context diagram. The assumption seems to be that the context of the system will be captured using a "use case" diagram, with the boundary of the system being represented by a classifier (class, component, or package) that contains the use cases, or simply by a diagrammatic annotation such as a rectangle drawn around the use cases. However, there are a number of practical difficulties with this approach, including the complexity of the resulting diagram, the fact that the use case list may not be available when the context diagram is created, and the convention that the external interfaces are made to specific use cases. In the context diagram, we really want to abstract this detail away and treat the system as a black box.

The solution to these difficulties is to create a UML diagram of the form shown in Figure 16–2.

This sort of UML diagram can be created using the "use case" or "class diagram" diagram editors of many mainstream UML modeling tools, although in fact it doesn't share a lot of similarity with either standard diagram. The key points about it are as follows:

- The system is represented as a UML component, stereotyped as a subsystem, a stereotype found in the UML standard profile, or with a more specific stereotype that you create yourself.
- External entities that cause human interactions with the system are represented as UML actors.
- External entities that are systems are represented as either further subsystem components or actors, possibly with their icons changed via stereotyping to be more representative of the entities that they represent (as suggested by the UML standard).
- Interfaces between the external entities and the system being designed can be represented as UML information flows, UML dependencies, or UML associations, optionally augmented with UML "conveyed information" icons that define the information flowing over the interface (which we don't show in the example but would be represented as small black arrowheads on the associations).[1]

While UML can be used to create a context diagram, it would be fair to say that the language does not provide particularly strong support for this type of model. For this reason, we often use informal boxes–and–lines notation instead, drawing something more akin to a "rich picture" of the system's context using a simple, ad hoc notation (and it's obviously important to define the notation clearly). Figure 16–3 shows the same system represented in boxes-and-lines notation.

The advantage of this style of diagram is that it can be much more expressive than plain UML, and it's probably easier for most people to create and understand than one created in strict accordance with UML. One of the major disadvantages, apart from your having to design and explain the notation, is that this model (or picture) is separate from the rest of your architectural models, assuming they're in UML. However, a number of UML modeling tools can now draw this sort of informal picture, which largely addresses this concern.

ACTIVITIES Definition of context takes place very early in the project lifecycle and is often rather ad hoc and unstructured as a result. It is also rarely under the control of the architect—you will be a participant and will provide input and feedback, but key decisions will probably be made by the senior stakeholders (typically the acquirer and some senior users).

It is possible to put some level of formality in place, however. At a minimum, a single document should be maintained and lodged in a place where everyone who needs access has it. It may be necessary to restrict access to the

1. Information flows and conveyed information annotations were introduced as part of UML 2. They are supported at differing levels of fidelity by different modeling tools, but they are a valid part of the metamodel, defined in the "Superstructure" specification [OMG10b].

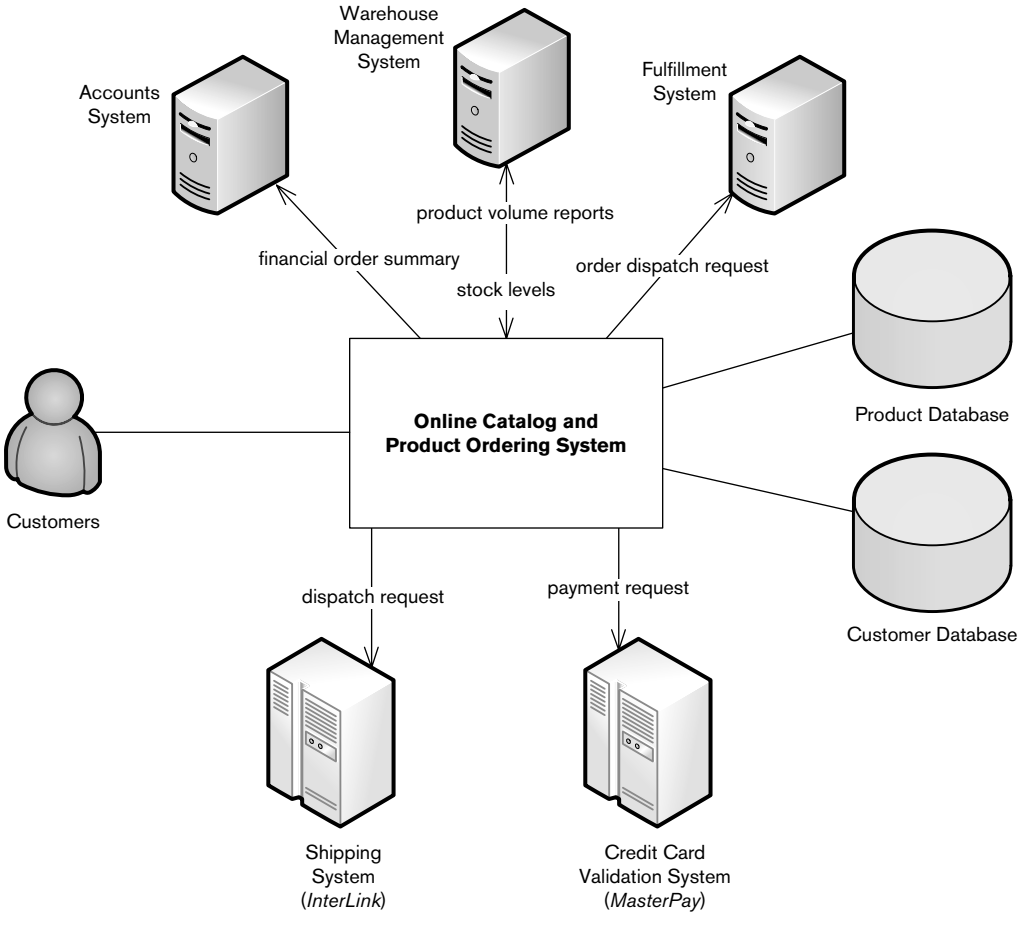

FIGURE 16–3 INFORMAL CONTEXT DIAGRAM

document to key personnel if the project is sensitive (for example, if it will lead to the retirement of existing systems or has contractual implications with suppliers). If possible, historical versions of the document should be retained along with a log of who changed what.

You will typically go through the following steps when preparing a context model.

- *Review the goals of the system*: Briefly review and capture the business and technology goals of the system—for example, "Reduce cost per transaction by 15%," "Streamline the ordering and fulfillment process, enabling better customer service," "Replace the current architecture with one that is more performant, resilient, and amenable to change," and so

on. The goals should make the motivation for the project clear, illustrating how its implementation will improve the current situation, in terms that the acquirer and other key stakeholders can understand.

- *Review the key functional requirements*: Briefly review and summarize the key requirements that characterize what the system must do, grouped by subject area. Use the scope definition for this.
- *Identify the external entities*: Itemize all internal and external systems, gateways, services, external data stores, devices, appliances, and users and roles that may interact with the system. You will need to use your own and others' knowledge of the business area, and any existing documentation such as system diagrams or organizational charts. At this stage, if there is any doubt as to whether an entity should be included, include it—you can always take it out later.
- *Define responsibilities of external entities*: Use your and your stakeholders' knowledge of the entities to map out their expected responsibilities. If there are any responsibilities that you find you can't assign to the system or an external entity, you have missed something in the system's context.
- *Identify the interfaces between this system and each external entity*: Use your and your stakeholders' knowledge of the processes the system will implement to identify the data flows and service invocations (in either direction) that these will require. Again, the scope definition will help make sure you don't miss anything.
- *Identify and validate the interface definitions*: Make sure that each interface is defined (perhaps in the AD but probably elsewhere) and that it is compatible with the use to which it will be put. If the interface is documented elsewhere, make sure you reference it in the AD.
- *Walk through key requirements*: Follow the flow of control and flow of information between the system and the external entities. As you do this, add all the external interfaces that are needed to implement these flows.
- *Walk through scenarios or use cases*: If you have more detailed scenario definitions or use cases, walk through these to validate the model. Add or update any external entities or interfaces required.

Interaction Scenarios

It is often useful to model some of the expected interactions between your system and the external entities in more detail than is provided in a context diagram. This sort of model helps to uncover implicit requirements and constraints (such as ordering, volume, or timing constraints) and helps to provide a further, more detailed level of validation. While you are unlikely to have time to model all the scenarios in which your system will participate, it can be useful to model some of the more complicated, contentious, or less well-understood ones, especially when system usage is unclear or there is disagreement among your stakeholders.

An interaction scenario represents two or more *participants* (usually the system and one or more external entities), and a sequence of *interactions* between them, where an interaction is a flow of information and/or a request to perform an action. The interactions should collectively serve a specific purpose or implement a specific function. Refer to Chapter 10 (Identifying and Using Scenarios) for more detail.

NOTATION Interaction scenarios are usually captured using simple textual interaction lists (rather like those used for use case definitions) or UML sequence diagrams that illustrate the interactions via a graphical notation. More detail is given in Chapter 10.

ACTIVITIES Refer to the discussion of scenarios in Chapter 10.

PROBLEMS AND PITFALLS

Missing or Incorrect External Entities

Most systems development projects tend to be relatively chaotic in their early stages (their teams are in their "forming" or "storming" stages in Tuckman's model of group development). Roles, even senior roles, may not be formally defined, and as a result context is often unclear and subject to frequent change. It is therefore easier than you might think to accidentally leave something out of the context model, include something that is not needed, or put the system boundaries in the wrong place.

Getting the context wrong can have a huge impact later on: Either the project will have to undergo significant change at a late point in the lifecycle, which adds considerably to its cost, duration, and complexity; or the delivered system will be incomplete or provide unnecessary functionality.

RISK REDUCTION

- Work with a wide range of stakeholders to ensure that their concerns are adequately reflected in the context model and interaction scenarios. For example, you should ensure that any functionality they require either is part of the system scope, is provided by an external entity, or is excluded entirely with the agreement of the people who need it.
- Involve a domain expert in this analysis as early as you can, and make sure that person is involved in review and sign-off of this part of the AD.
- Ensure that once the context model has stabilized, it is change-managed and subsequent changes to it are reviewed and agreed upon.

Missing Implicit Dependencies

It is easy to miss the subtle dependencies between external entities. For example, you may assume that a particular business entity or data item is instantaneously available in two external systems, when there is actually a significant latency due to the mechanics of data transfer. Or you could assume that the availability of an external system will affect only one part of your system, whereas in fact other systems you rely on are also dependent on it, making its nonavailability much more important to you. Such implicit dependencies can be hard to understand yet may have significant implications for the architecture. They should therefore be captured early and documented clearly.

RISK REDUCTION

- Assume nothing, work with your stakeholders to uncover and understand implicit dependencies, and ensure that they are documented in the Context view.

Loose or Inaccurate Interface Descriptions

It's tempting to get the basic idea of an external interface and leave it at that, hoping that the design process will elicit the details. In fact, you always have to do this to some extent as you can't understand every detail of every interface. However, it is important that you capture enough detail so that the architectural implications can be understood.

RISK REDUCTION

- Ensure that you understand your external interfaces in sufficient detail to use them confidently, and capture enough information about them in the Context view to characterize the effect they have on your architecture.
- Avoid the temptation to gloss over things that are complex in the expectation that problems will be resolved later.

Inappropriate Level of Detail

Getting the level of detail right is a challenge everywhere in the AD but is especially important in the Context view. If you provide too much detail, stakeholders, especially senior stakeholders like the acquirer, may become overwhelmed and fail to understand the big picture. Conversely, if you gloss over some aspects of the context or scope, expecting them to be fleshed out later, you may miss something important, mislead your stakeholders, or allow incorrect assumptions to be made.

RISK REDUCTION

- Look out for scope or requirements that appear vague because nobody understands what they mean (or people assume different meanings) and explore them further in order to ensure that they are understood.
- If the Context view becomes too detailed, move some of the information either into appendices in the document or into another view in the AD (typically the Functional or Information view).
- Consider applying some rules of thumb to determine whether your Context view is becoming too detailed. Although every situation varies, we have found the following rules to be useful in practice:
  - A context diagram should usually fit on a single sheet of paper.
  - A scope definition should not usually be more than 2 to 3 pages.
  - If there are a lot of requirements, they should be grouped by functional area, organizational responsibility, or some other logical category.
  - If there are more than, say, 10 to 20 external entities, consider whether they can be grouped by type (for example, a large number of suppliers of the same type of goods), or whether you really have a single system at all, rather than a collection of systems.

Scope Creep

Scope creep is the phenomenon of uncontrolled changes to system scope, which often occur gradually without being particularly visible to stakeholders. These changes usually have the effect of increasing what the system is expected to do, often without due consideration of whether this is sensible or achievable. For example, when interviewing users about required functionality, it is easy for each user to add a few more requirements to the mix that really are "nice-to-haves" rather than truly essential. By the time this process is completed, the system is significantly larger and more ambitious, possibly fatally so.

Scope creep can also occur once the scope has stabilized if it is not subject to well-managed change control.

RISK REDUCTION

- Challenge additions or changes to scope to confirm that they really are necessary and make sure their implications are understood.
- Work to help stakeholders understand the consequences of adding requirements, such as increased time to market, development and operational cost, or system complexity and stability.
- Ensure that scope changes are change-managed once the scope has stabilized.

Implicit or Assumed Context or Scope

More than in any other part of the AD, the scope definition is where you should state the obvious when there is any chance of misunderstanding. Don't be tempted to leave things out because "everybody knows that"—the odds are that there are some stakeholders who don't, or that some of these nuggets of information will get lost along the way.

RISK REDUCTION

- Don't be afraid to state the obvious in the Context view. You will be glad you did later on!

Overcomplicated Interactions

Interactions with some external entities (particularly older systems) can be a lot more complicated than expected, so it's easy to end up with unexpected problems when you come to build the interfaces. For example, some of the problems we have encountered when dealing with interfaces to long-established systems have included the need for unusual data encodings, poorly understood (yet complicated) conversational protocols, and complex and proprietary interface technologies that can cause difficulties for development, testing, and operational activities.

RISK REDUCTION

- Take the time to understand interfaces to external systems early in the architectural design process, and don't assume that they're necessarily the same as the interfaces you've met before.
- Find expertise in the interfaces that you need to use, prototype interactions with them, and test them thoroughly in order to understand how they behave in different situations.

Overuse of Jargon

Inputs to the Context view come from a wide variety of sources. It is easy, therefore, to make careless use of business and technology terminology that may not be well understood by the majority of your stakeholders. Since people are often reluctant to question things they do not understand, you risk confusion and misunderstanding.

RISK REDUCTION

- Try to avoid any terminology that is not widely understood. If you need to use jargon and there is any risk of confusion, provide a glossary.

CHECKLIST

- Have you consulted with all of the stakeholders who are interested in the Context view (which is probably all of them)?
- Have you identified all of the external entities with which the system needs to interact and their relevant responsibilities?
- Do you have a good understanding of the nature of every interface with each external entity, and is this documented to an appropriate level of detail?
- Have you considered possible dependencies between the external entities with which you have to interact? Are these implicit dependencies documented in the AD?
- Does the context diagram adequately illustrate all of the interfaces from the system to its environment, with sufficient definition underpinning the diagram?
- Have all key stakeholders formally agreed to the content of the context model? Is this documented somewhere?
- Has the context model been placed under formal change control?
- Is the change control process being followed? Are stakeholders being consulted on changes and their consent obtained?
- Is the context model placed somewhere where everyone can easily find it, such as a public shared folder or wiki page?
- Have you identified all of the key capabilities or requirements of the system, and are they documented to an appropriate level of detail?
- Is the scope definition internally consistent?
- Does the scope identify any important technology constraints, such as mandated platforms?
- Is the scope specified at an appropriate level of detail, balancing brevity with clarity and completeness?
- Have you explored a set of realistic scenarios for external interactions between your system and external actors?
- Are other teams with which you interact clear on the context and scope and any implications for them?
- Have you checked the context model to see if there are any "obvious" statements that should be explicitly stated but have been omitted?
- Do the main business processes appear to have adequate coverage, by either systems or defined manual processes?
- Does all the data needed to support the main business processes appear to be stored somewhere, on-site or externally?
- Does the overall solution hang together in a coherent way?

FURTHER READING

Many software architecture books discuss the process of setting the context of the system; examples include Garland and Anthony [GARL03], which describes a Context viewpoint, and Bosch [BOSC00], which describes how to define the system context at the start of the architectural design process.

A number of requirements engineering books also discuss scoping systems. A particularly good example is Sommerville and Sawyer [SOMM97], which presents a clear set of guidelines around requirements capture, presentation, and ratification. Each guideline is accompanied by a cost/benefit analysis and practical suggestions for how it can be implemented.

Information on Tuckman's model of group development can be found in [TUCK65] and elsewhere.

17

THE FUNCTIONAL VIEWPOINT

Definition	Describes the system's runtime functional elements and their responsibilities, interfaces, and primary interactions
Concerns	Functional capabilities, external interfaces, internal structure, and functional design philosophy
Models	Functional structure model
Problems and Pitfalls	Poorly defined interfaces, poorly understood responsibilities, infrastructure modeled as functional elements, overloaded view, diagrams without element definitions, difficulty in reconciling the needs of multiple stakeholders, wrong level of detail, "God elements," and too many dependencies
Stakeholders	All stakeholders
Applicability	All systems

The Functional view of a system defines the architectural elements that deliver the functions of the system being described. This view documents the system's functional structure—including the key functional elements, their responsibilities, the interfaces they expose, and the interactions between them. Taken together, this demonstrates how the system will perform the functions required of it.

The Functional view is the cornerstone of most ADs and is often the first part of the description that stakeholders try to read. (Too often, it is also the only view of the architecture produced.) It is probably the easiest view for stakeholders to understand. The Functional view usually drives the definition of many of the other architectural views (particularly Information, Concurrency, Development, and Deployment). You will almost always create a Functional

view and will often spend a lot of time refining the functional structure that it defines.

A major challenge when defining the Functional view is to include an appropriate level of detail. Focus on what is architecturally significant—in other words, what has a visible impact on stakeholders—and leave the rest to your designers. Avoid documenting physical implementation details such as servers or infrastructure in your Functional view, as this will overcomplicate your models and confuse your stakeholders. (You will document these elements in your Deployment view.)

CONCERNS

Functional Capabilities

Functional capabilities define what the system is required to do—and, explicitly or implicitly, what it is not required to do (either because this functionality is outside the scope of consideration or because it is provided elsewhere).

On some projects, you will have an agreed-upon set of requirements at the start of architecture definition, and you can focus in the Functional view on showing how your architectural elements work together to provide this functionality. However, in many projects this isn't the case, and as we discussed in Chapter 8 and Chapter 16, the onus will be on you in this case to ensure that there is a clear definition of what the system will (and won't) be required to do.

External Interfaces

External interfaces are the data, event, and control flows between your system and others.

Data can flow inward (usually resulting in an internal change of system state) and/or outward (usually as a result of internal changes of system state). Events can be consumed by your system (notifying your system that something has occurred) or may be emitted by your system (acting as notifications for other systems). A control flow may be inbound (a request by an external system to yours to perform a task) or outbound (a request by your system to another to perform a task).

Interface definitions need to consider both the interface syntax (the structure of the data or request) and semantics (its meaning or effect).

Internal Structure

In most cases, you can design a system in a number of different ways to meet its requirements. It can be built as a single monolithic entity or a collection of

loosely coupled components; it can be constructed from a number of standard packages, linked together using commodity middleware, or written from scratch; or its functional needs can be met by using network-accessible services provided by systems external to this one or even to the organization. Your challenge is to choose among these many options in order to create an architecture that meets the requirements, exhibits the required quality properties, and is fit for purpose.

The internal structure of the system is defined by its internal elements, what they do (i.e., how they map onto the requirements), and how they interact with each other. This internal organization can have a big impact on the system's quality properties, such as its availability, resilience, ability to scale, and security (e.g., a complex system that crosses organizational boundaries is generally harder to secure than a simple one running on a couple of collocated machines).

Functional Design Philosophy

Many of your stakeholders will be interested only in what the system does and the interfaces it presents to users and to other systems. However, some stakeholders will be interested in how well the architecture adheres to established principles of sound design. Technical stakeholders, in particular the development and test teams, want a sound architecture, because a well-designed system is easier to build, test, operate, and enhance. Other stakeholders—particularly acquirers—implicitly want a well-designed system because it is faster, cheaper, and easier to get such a system into production.

The design philosophy will be underpinned by a number of design characteristics such as the examples listed in Table 17–1.

TABLE 17–1 DESIGN CHARACTERISTICS

Design Characteristic	Description	Significance
Coherence	Does the architecture have a logical structure, with the elements working together to form a whole?	If the architecture doesn't look coherent, this may indicate that the element decomposition is wrong, and it may make it hard for stakeholders to understand.
Cohesion	To what extent are the functions provided by an element strongly related to each other?	In a highly cohesive system, related functions are grouped together, resulting in simpler, less error-prone designs.
Consistency	Are mechanisms and design decisions applied consistently throughout the architecture?	A consistently designed and implemented system is much easier to build, test, operate, and evolve than one with a lot of accidental inconsistency.

Continued on next page

TABLE 17–1 DESIGN CHARACTERISTICS *(CONTINUED)*

Design Characteristic	Description	Significance
Coupling	How strong are the element interrelationships To what extent do changes in one module affect others?	Loosely coupled systems are often easier to build, support, and enhance but may suffer from poor efficiency compared with a monolithic approach.
Extensibility	Will the architecture be easy to extend to allow the system to perform new functions in the future?	Extensibility is often the result of other properties such as coherence, low coupling, simplicity, and consistency, but it is worth bearing in mind explicitly when considering your designs.
Functional flexibility	How amenable is the system to supporting changes to the functions already provided?	Systems that are designed to be easy to change are usually harder to build and typically are less efficient than systems that are less adaptable.
Generality	Are the mechanisms and decisions in the architecture as general as is practicable?	If the solutions embodied in the architecture are generic, the architecture will be amenable to extension and change. However, this must be balanced against any resulting increase in cost and complexity.
Interdependency	What proportion of processing steps involves interactions between elements as opposed to within an element?	Communicating between certain types of elements can be an order of magnitude more expensive (in terms of processing time and elapsed time), and significantly less reliable, than performing an operation within a functional element.
Separation of concerns	To what extent is each internal element responsible for a distinct part of the system's operation? To what extent is common processing performed in only one place?	High separation results in a system that is easier to build, support, and enhance but may adversely impact performance and scalability compared with a monolithic approach.
Simplicity	Are the design solutions used within the system the simplest ones that would be suitable?	Complexity makes systems difficult and expensive to build, comprehend, operate, and evolve, but a *simplistic* approach may well not meet the requirements of a sophisticated system.

In general, these design characteristics have a positive effect on a number of system qualities, particularly those relating to evolution, such as flexibility and maintainability. They also usually have a positive effect on other system qualities such as performance and security (e.g., separation of concerns and simplicity can make security easier to achieve, while consistency is likely to make performance

TABLE 17–2 STAKEHOLDER CONCERNS FOR THE FUNCTIONAL VIEWPOINT

Stakeholder Class	Concerns
Acquirers	Primarily functional capabilities and external interfaces
Assessors	All concerns
Communicators	Potentially all concerns, to some extent, depending on context
Developers	Primarily design quality and internal structure, but also functional capabilities and external interfaces
System administrators	Primarily functional design philosophy, external interfaces, and possibly internal structure
Testers	Primarily design quality and internal structure, but also functional capabilities and external interfaces
Users	Primarily functional capabilities and external interfaces

and scalability easier to achieve). In some cases, though, you need to consider the possibility of a negative relationship between "good" design and other system qualities (e.g., very loosely coupled systems can be less performant than more tightly coupled ones); in some cases this can mean the need to compromise over the design characteristics that can be achieved (we note the need for occasional design compromises in some of the perspectives in Part IV).

Principles and patterns are good techniques for defining how you want the design of the system to embody these design characteristics, as they can guide the system's designers to make design decisions that support the characteristics that you are most interested in achieving. We discuss this further in Chapter 8.

Stakeholder Concerns

Typical stakeholder concerns for the Functional viewpoint include those listed in Table 17–2.

MODELS

Functional Structure Model

The functional structure model typically contains the following elements.

- *Functional elements*: A functional element is a well-defined runtime (as opposed to design-time) part of the system that has particular responsibilities and exposes well-defined interfaces that allow it to be connected to other elements. At its simplest level, an element is a

software code module, but in other contexts it could be an application package, a data store, or even a complete system.

- *Interfaces*: An interface is a well-defined mechanism by which the functions of an element can be accessed by other elements. An interface is defined by the inputs, outputs, and semantics of each operation offered and the nature of the interaction needed to invoke the operation. Common types of interfaces found in information systems are remote procedure calls (RPCs) of various types, messaging, events, and in some cases interrupts.
- *Connectors*: Connectors are the pieces of your architecture that link the elements together to allow them to interact. A connector defines the interaction between the elements that use it and allows the nature of the interaction to be considered separately from the semantics of the operation being invoked. The nature of the interactions between elements can be intimately bound up in how they are connected.

  The amount of consideration you need to give connectors depends on your circumstances. At one extreme—for example, when one element calls another via a simple procedure call—you can just note that one element connects to another. At the other extreme, such as a message-based interface, a connector can be defined as a type of element in its own right as it provides capabilities to the interactions that occur across it. As always, the focus needs to be on what is architecturally significant in the context in which you are working.
- *External entities*: As we defined in Chapter 16, external entities are other systems, software programs, hardware devices, or any other entity with which your system interacts. They are obtained from your system's Context view, and each appears in the functional model at the far end of an interface, external to your system.

The functional structure model does not define how code is packaged and executed in processes and on threads, so this view doesn't constrain element packaging or deployment—this is the domain of the Concurrency and Deployment views.

Similarly, it is generally not a good idea to model underlying infrastructure as functional elements, unless that infrastructure performs a functionally significant task, independent of the other functional elements, without which the view doesn't make sense. Infrastructure that simply supports the operation of the functional elements should normally not be shown in the Functional view; it is best considered in the Deployment view.

For example, you might well want to show message queues, as they are important interelement connectors and so the view doesn't make sense without them, but you probably don't need to show the message broker that provides the queues, which doesn't add anything in this context. The message broker would be shown in the Deployment view.

NOTATION You can use a number of techniques to represent the Functional view in a model.

- *UML component diagrams*: Using UML for a Functional view has a number of advantages, including its widespread comprehension and its flexibility. The main UML diagram you will use for the Functional view is a component diagram, which shows a system's elements, interfaces, and interelement connections.

EXAMPLE Figure 17–1 shows the typical elements in a UML component diagram. The system consists of two internal elements, Variable Capture and Alarm Initiator, interacting with one external element, Temperature Monitor. Variable Capture exposes one interface, VariableReporting, which is invoked by Temperature Monitor, and Alarm Initiator exposes one interface, LimitCondition, which is invoked by Variable Capture. VariableReporting is tagged with information that tells us it is an XML remote procedure call, over the HTTP protocol, and that, at most, 10 concurrent invocations can exist at one time.

You represent each of the system's elements and external entities with a UML component icon, annotated with its name and any stereotype needed to make the nature of the element clear. (Stereotypes allow you to extend the semantics of standard UML in a logical and consistent way to meet your individual circumstances.) One particularly useful stereotype is <<external>>, which indicates that the icon refers to an external entity, rather than a system element. Another is <<infrastructure>>,

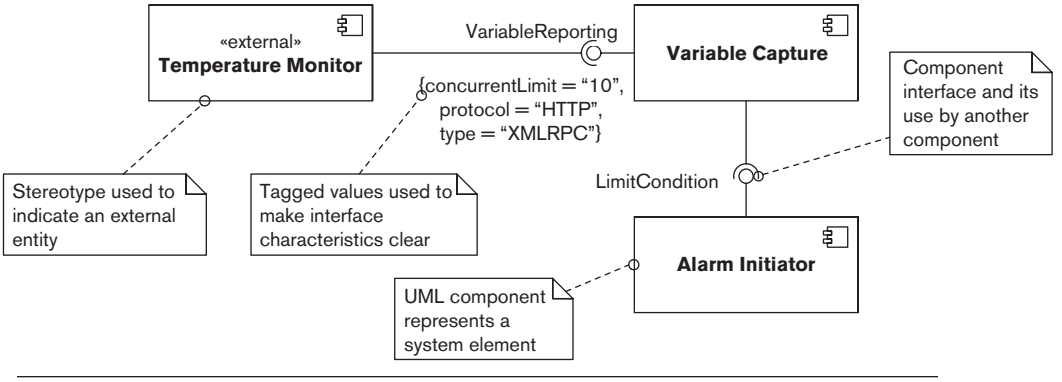

FIGURE 17–1 EXAMPLE OF A FUNCTIONAL STRUCTURE IN UML

which indicates an infrastructure element of the system that has a distinct functional role.

UML interface icons attached to a system element represent the interfaces it exposes. We have found that the small "lollipop" interface icon is more effective in the Functional view than the larger stereotyped class icon. In order to differentiate between types of interfaces, stereotypes may be defined with associated sets of tagged values that allow the characteristics of particular interfaces to be captured (such as "transport"). Using tagged values to capture the type of interface, the protocol used to access it (if any), and the number of concurrent users or connections allowed provides a good basis for interface classification.

Once you have identified elements and interfaces, you can show the connectors between the interfaces with UML dependencies and information flows, as described in the following example.

EXAMPLE The UML component diagram shown in Figure 17–2 is an example of using UML to document the functional structure of a simple system. The system under consideration provides a Web storefront (called the Web Shop) for customers to use when purchasing items from an online catalog that fits into an existing enterprise software environment. (To save space, we have omitted the detailed descriptions of the system components and their interfaces, but obviously these would be crucial information for a real model.)

The model shows that the system communicates with four external entities: the Web browsers of the three main user types (customers, customer care representatives, and catalog administrators) and an external system (the order fulfillment system). Our system is composed of five main functional components linked via a number of connector types (including HTML over HTTP and publish/subscribe messaging, with an LU 6.2 external interface).

Customers order from the Web Shop, which interacts with the Product Catalog, the Order Processor, and the Customer Information System. The catalog administrators maintain the product catalog via their Web-based interface, and the customer care representatives maintain the customer information via a dedicated interface client program (the Customer Care Interface). When the stock level of a particular item in the catalog is needed, the Product Catalog accesses this information from the Stock Inventory (which already exists).

We also have some insights into the nature of the intercomponent interactions. We know that up to 1,000 customers, 80 customer care representatives, and 15 catalog administrators may access the system

simultaneously. We also note that the interaction between the Product Catalog and the Stock Inventory components takes place using a specific protocol (presumably due to preexisting technology). We can assume for this example that the unadorned intercomponent communication takes place via some form of standard remote procedure call (which we will assume has been clearly defined elsewhere).

Having said this, one of the interesting points to note about this model is how much is *not* obvious from the diagram. The responsibilities of the components aren't clear, the details of their interfaces aren't clear, and the details of how the components interact aren't clear. This impresses on us the need to complete the textual descriptions that underpin the diagram and the need to understand the system via a number of models rather than just one (e.g., intercomponent interactions can be shown via system scenario modeling, as we described in Chapter 10).

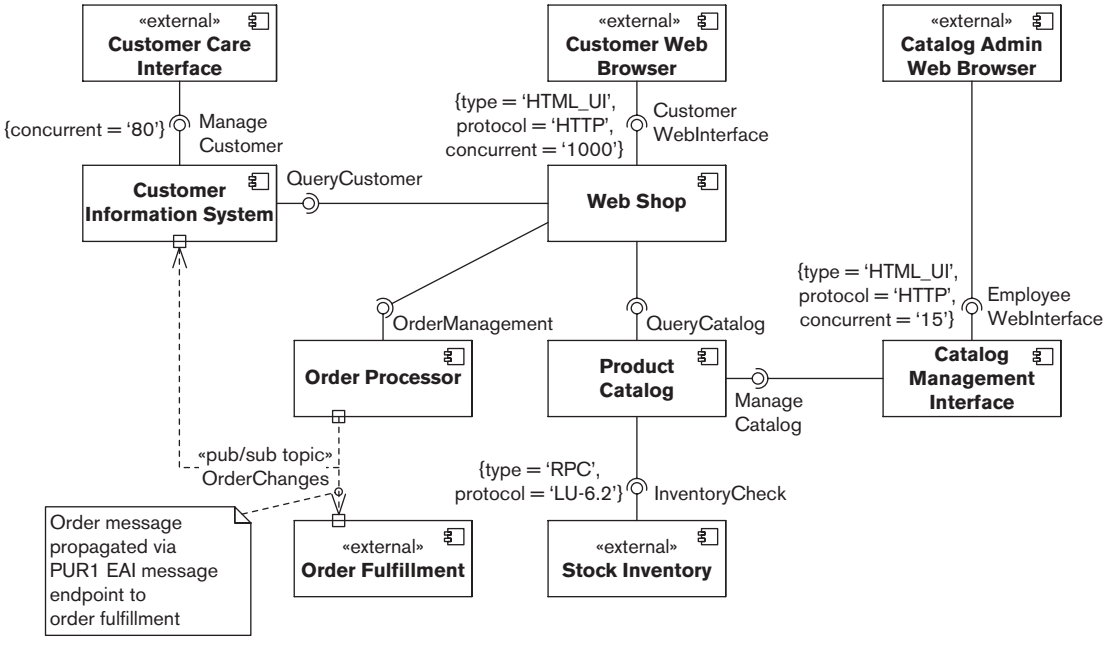

FIGURE 17–2 EXAMPLE OF A UML COMPONENT DIAGRAM

- *Other formal design notations*: UML is not the only well-defined design notation suitable for software development. A number of older structured notations (such as Yourdon, Jackson System Development, and the Object Modeling Technique of James Rumbaugh) have been successfully applied to software development problems for many years. The problem with using any of the notations developed for software design is that they tend to be fairly weak at describing the concepts (such as large-scale elements, interfaces, deployment options, and so on) that are important to architects. The older methods also aren't widely taught or used today, and so tool support may be difficult to come by, and they lack the general familiarity of UML for most people.
- *Architecture description languages (ADLs)*: Languages that do directly support the concepts that software architects are concerned with are generally known as ADLs. A large number of ADLs have been created (including Unicon, Wright, xADL, Darwin, C2, and AADL). The great attraction of ADLs is that they provide native support for some of the things that we need to capture and reason about in our architectural designs (such as components and connectors). However, nearly all ADLs have been developed in the research environment and tend to suffer from a number of practical drawbacks, including lack of stakeholder familiarity with them, relatively narrow scope (often only allowing "components" and "connectors" to be represented), and an inevitable lack of mature tool support. For these reasons, despite a number of years of searching, we still haven't found an ADL that we've been happy to adopt on a day-to-day basis.
- *Boxes-and-lines diagrams*: Many architects use a functional structure diagram drawn by using a custom boxes-and-lines notation. Such a diagram should show just the functional elements and their interfaces and should link the elements to the interfaces they use with a clear graphical device (typically an arrow, possibly with some annotation) that indicates the use of a connector. As with any custom notation, be sure to define the meaning of the notation clearly to avoid confusion.

EXAMPLE The boxes-and-lines diagram shown in Figure 17–3 gives an alternative, less formal, and possibly more user-friendly representation of the system described in the previous example.

In this model, we have defined our own notation. Functional elements are represented by rectangles and the links between them by lines, with arrows indicating the direction(s) of information flow. External user-facing interfaces are represented by an icon meant to look like a computer monitor, and external back-end systems are represented by rectangles with rounded corners. Data stores are represented by an icon that looks

like a disk drum, and functional interfaces (the Internet, the message bus) are represented by a cloud icon. The scope of the system is those elements within the dotted rectangle.

The benefit of the boxes-and-lines diagram is that nontechnical stakeholders, particularly business users and sponsors, may find it easier to understand. Such a model can be an invaluable tool in selling the features and benefits of the system to these stakeholders without getting bogged down in technical detail. Often you may use the boxes-and-lines diagram as a front for more detailed, rigorous UML models.

Although the boxes-and-lines diagram can be used less formally than a UML model, you shouldn't use this as an excuse for being less rigorous. In particular, early in architecture definition, you should define a standard notation for your diagrams—and make sure you stick to it. Try to develop icons that give an indication of the underlying purpose of the elements modeled (e.g., the disk-drum icon shown in Figure 17–3 is often used to model data stores).

You should always support any such model with a definition of its elements and the interfaces between them, presented in a standardized way.

- *Sketches*: You can create a less formal feel for the view by using a sketch, that is, by introducing an ad hoc notation as required to represent each of the aspects of the view that are significant for your system. The use of a sketch is often required to effectively communicate essential aspects of the view to nontechnical stakeholders. The problem with this approach is that it can lead to a poorly defined view and confusion among stakeholders. As with the boxes-and-lines diagram, you can get around this by using a sketch to augment a more formal view notation (such as UML) and using different notations for different stakeholder groups.

Representing procedure-oriented element interactions is relatively straightforward, but modeling message-oriented interactions (such as those found where elements are connected via publish/subscribe messaging systems) can be significantly harder.

We used to model message-oriented interfaces by showing the message distribution mechanism (typically a piece of message-oriented middleware) as a functional element and connecting the various message source and destination elements to it. This does get the point across, but it's difficult to discern the overall message flow in the system. A better approach, originally suggested by Garland and Anthony [GARL03], is to use ports and information flows to model message-oriented interactions between system elements.

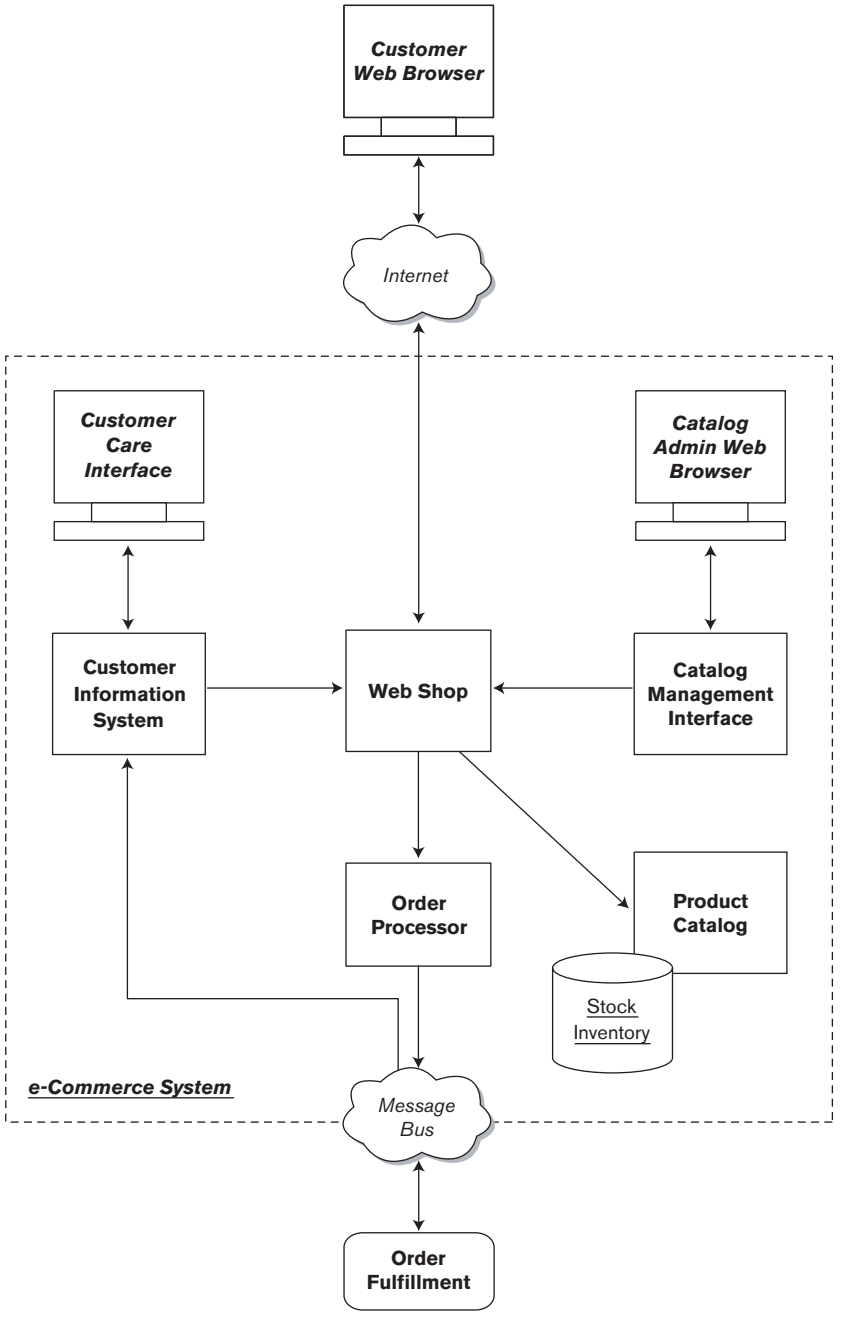

FIGURE 17–3 EXAMPLE OF A BOXES-AND-LINES DIAGRAM

The notion of ports comes originally from the real-time systems community, where a port is an abstract representation of the source or destination of messages. A more general notion of ports was integrated into version 2 of UML, and one of their uses can be to clearly show the messaging within a system.

EXAMPLE An example of using ports and information flows for messaging is shown in the UML model in Figure 17–4.

This diagram illustrates part of a notional system in a financial institution where prices are calculated by one system element (the Price Calculator) and distributed to the other system elements via asynchronous messages. The small boxes attached to the system elements represent ports. The one attached to the Price Calculator is an output port (it creates messages), and the ones attached to the other elements are input ports (they receive messages). A UML 2 information flow connector is used to indicate the message flow between elements, with a stereotype to indicate the type of messaging in use and the "information conveyed" annotation capturing the message type (publish/subscribe messaging and "Prices" in the example).

When the message-oriented interactions are illustrated by using a separate notation, they can be combined with procedure-oriented element interactions on a single diagram without fear of confusion. You can also use such a technique to model higher-level messaging systems, such as those that implement EAI architectures.

Remember that as we said earlier, a Functional view should describe only the system's functional elements. If you need notational items to represent deployment, concurrency, or other aspects of the system, your Functional view has become overloaded.

Note: When talking about system design notations, it's also worth mentioning the existence of SysML, a design language for systems engineering, which is based on UML 2 (SysML is actually defined as a UML 2 profile). We've been following the development of SysML over a number of years, and while it's undoubtedly a useful tool for people working in systems engineering, we haven't found it to be a better alternative to UML 2 for information systems design. SysML is aimed at situations where systems engineers need to integrate hardware, software, personnel, facilities, and other varied aspects of very large systems, rather than the more focused problem of the design of an information system. The sysml.org, omgsysml.org, and sysmlforum.com Web sites are good places to find out more about SysML and to track its evolution.

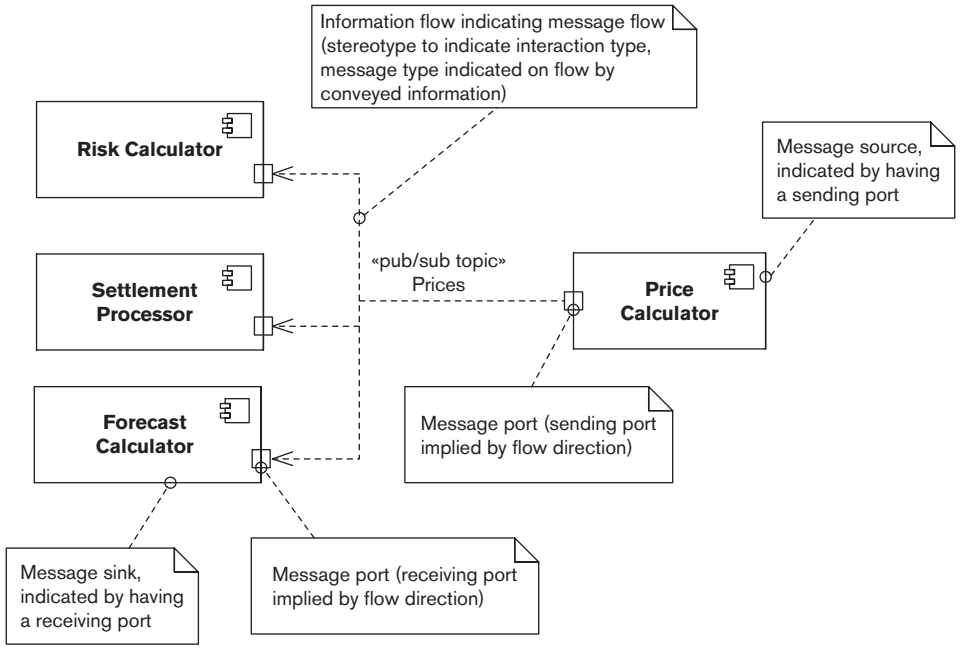

FIGURE 17–4 EXAMPLE OF A MODEL FOR MESSAGING INTERACTIONS

ACTIVITIES

Identify the Elements. You can identify the functional elements by following these steps.

1. Work through the functional requirements, deriving key system-level responsibilities.
2. Identify the functional elements that will perform those responsibilities.
3. Assess the identified set against the desirable design criteria.
4. Iterate back to refine the functional structure until you judge it to be sound.

Of course, some elements may be defined for you already (e.g., software libraries, software packages, preexisting systems or subsystems), in which case the process for these elements is one of understanding rather than identifying and designing.

Refining the set of functional elements involves applying one or more refinements to the functional structure.

- *Generalization*: identifying some common responsibilities across a number of elements and introducing a number of more general elements that can be reused across the system to perform these tasks. Generalization is particularly important as part of a larger enterprise or product-line architecture to allow reuse of software assets across a number of similar products or systems.
- *Decomposition*: breaking a large, complex element into a number of smaller subelements. For large systems, you will often need to break the top-level functional elements into more manageable subsystem-level elements to allow them to be designed and built.
- *Amalgamation*: replacing a number of small functional elements with a larger element that includes all of the functions of the smaller ones. Amalgamation is typically used when a large number of small but similar functional elements have been identified. In such cases, it often makes sense from an architectural perspective to replace the smaller elements with a single large element that can factor out the commonality between the smaller ones and reduce the amount of interactions the system requires.
- *Replication*: replicating either a system element or a piece of processing. An example is data validation, where you identify a validation element for incoming data and then replicate it across a number of the system's external interfaces. Replication can bring performance benefits, but care must be taken to keep the replicated components consistent.

If you are using an architectural style to guide your design process, the process is slightly different because it will involve creating an instantiation of the style such that the system-level responsibilities are assigned to elements of the style. This activity is closely related to the next step—assigning responsibilities to the elements.

We don't talk about the element identification process in a lot of detail in this book because there are many ways to do it, and the correct method to use depends on the type of system and the software development approach you are using. (Procedural, object-oriented, and component-based approaches all influence component identification in different ways.) See the Further Reading section at the end of this chapter for some sources that discuss element identification.

Assign Responsibilities to the Elements. Once you have identified candidate elements, your next activity is to assign clear responsibilities to them—that is, the information managed by the element, the services it offers to other parts of the system, and the activities it initiates. You may have done this in the previous step; if not, complete it here.

EXAMPLE Table 17–3 shows the responsibilities assigned to two of the elements for the e-commerce system described in earlier examples.

TABLE 17–3 EXAMPLES OF ELEMENT RESPONSIBILITIES

Element Class	Responsibilities
Web Shop	• Present customers with an HTML-based user interface they can access with a Web browser. • Manage all state related to the customer interface session. • Interact with other parts of the system to allow customers to view the catalog and stock levels, buy goods, and view their customer information.
Customer Information System	• Manage all persistent information about customers of the system. • Provide a query-only interface that can be used to retrieve information held on a particular customer that should be visible to that customer. • Provide an information management programmatic interface that can be used to create customer information management applications. • Provide an event-driven message-handling interface to accept details of orders placed by customers and the state changes of those orders.

Design the Interfaces. The services offered by your elements need to be accessed via well-defined interfaces. The definition of an interface must include the operations that the interface offers; the input, outputs, preconditions, and effects of each operation; and the nature of the interface (messaging, remote procedure call, Web service, and so on).

A good approach to consider when developing element interfaces is Design by Contract, an interface design method originally created by Bertrand Meyer for developing interfaces in object-oriented systems. This approach involves defining interfaces via "contracts" that use preconditions, postconditions, and invariants to precisely define operation behavior and relationships.

The appropriate notation for interface definition depends on the type of interface and who needs to understand this information (considering factors such as the likely implementation technology, the background of the development team, and the kinds of interfaces that need to be described). The following are some common interface definition notations.

- *Programming languages*: Interfaces can be defined directly by using a programming language to define the operation signatures along with text and/or language assertions to define the operation semantics. This approach is simple but ties you to the style, assumptions, and limitations of the particular programming language. This may not be ideal, particularly if you're using multiple technologies. This approach works particularly well for programming libraries or in other situations where the system is really

a single, large programming artifact or where a single programming language is used to implement the entire system.

- *Interface definition languages (IDLs)*: Specialist IDLs have been developed to support mixed-language distributed systems technology (so there is an IDL for CORBA, an IDL for .NET, WSDL for Web services, and so on). These languages are independent of implementation technology and tend to offer simpler facilities than programming languages do, more suitable for defining architectural interfaces. Provided that your interested stakeholders can read (or be taught to read) them, these languages offer a good option for defining operation signatures.
- *Data-oriented approaches*: Interfaces can also be described purely in terms of messages that are exchanged. Examples of this type of interface definition include interfaces accessed via messaging systems and interfaces defined in terms of structured document exchange (e.g., document-oriented, Web-service-based interfaces with messages defined using XML Schema). This approach works particularly well for event-based interfaces that are defined in terms of the exchange of business events rather than the invocation of operations.

Whatever notation you use to describe interfaces, remember that an interface is significantly more than just a simple definition of how you call the operations. Unfortunately, none of the approaches we have described offer facilities for defining interface semantics, and so a clear definition of an interface will involve the use of natural language or specialist languages like Object Constraint Language (OCL) to achieve this. An interface definition must accurately communicate the pre- and postconditions of each operation and how the operations should be combined in order to perform a useful function (preferably with examples). Anything less than this is likely to cause significant problems when the interfaces come to be used.

Design the Connectors. The elements of your system need to communicate in order to achieve the system's goals, and as you identified your element responsibilities, you probably noted the need for elements to interact in order to implement their responsibilities. The interactions take place across *connectors* of some sort that link delegating elements to the interfaces offered by the elements to which they wish to delegate. Sometimes the type of connector required is self-evident (such as a simple procedure call), whereas in other cases you'll need to think carefully about whether you need synchronous or asynchronous communication, the resiliency required of the connector, the acceptable latency of interactions across it, and so on. For each required interelement communication path in your architecture, add a connector to the model to support it (be that RPC, messaging, file transfer, or other mechanisms).

Check the Functional Traceability. The requirements documentation for your system will have defined a number of functions that the system has to offer. You should carry out a traceability check to ensure that all functional requirements have been met by the proposed functional structure. Such an analysis often reveals missing or incomplete functions in the functional structure model. If it needs to be captured formally, the traceability analysis is usually presented as a table of functional requirements cross-referenced against the functional model elements with responsibilities relating to those requirements.

Walk through Common Scenarios. It can be extremely valuable and illuminating to walk through common system usage scenarios with your stakeholders, using the Functional view to illustrate how the system will behave in each case; doing this with the testers, the development team, and the system administrators can be particularly useful. In such a walkthrough, you should explain how the system's elements would interact in order to implement the scenario. Often, architectural weaknesses or misunderstandings as well as missing elements are identified as part of such a process. Such a walkthrough can form part of a larger architectural assessment exercise such as that introduced in Chapter 14.

Analyze the Interactions. Given the impact that excessive interelement interactions can have, it is useful to analyze the chosen structure from the point of view of the number of interelement interactions taken during common processing scenarios. Refining the functional structure to reduce interelement interactions to a minimum set without distorting the coherence of the functional components usually results in a well-structured system with cohesive, loosely coupled elements. It is typically an important step toward an efficient and reliable system. When performing interaction analysis, you need to make tradeoffs to ensure that reducing interelement interactions does not result in a distorted system structure with undesirable redundancy or inappropriate element partitioning.

Analyze for Flexibility. Successful systems are always under pressure to change. Given this reality, you should consider how flexible your architecture is in the face of change, as early in the project as you can. The functional structure of a system is often one of the primary factors affecting the flexibility of information systems. It's useful to work through some "what if" scenarios that reveal the impact of possible future changes on your system. A common problem at this point is that the changes implied by the change analysis conflict with those suggested by the interaction analysis. Therefore, it is important that you trade off these two factors during architectural evaluation in order to find the right balance for your system, and that you avoid burdening your design with complexity that will never be used. Again, assessing this can be part of your architectural evaluation activities; we talk more about this aspect of design in Chapter 28.

PROBLEMS AND PITFALLS

Poorly Defined Interfaces

Many architects define their elements, responsibilities, and interelement relationships well, yet totally neglect their connectors and interface definitions. Defining interelement interfaces clearly can often be something of a chore. However, it is one of the most important tasks you can perform for the system. Without good interface definitions, major misunderstandings will occur between subsystem development teams, leading to a range of problems from build errors to obviously incorrect behavior to subtle, occasional system unreliability.

RISK REDUCTION

- Define your interfaces and interelement connectors clearly and as early as possible.
- Review interfaces and connectors frequently to ensure that they are clearly understood.
- Do not consider element definition complete until interfaces have been designed.
- Make sure that interface definitions include the operations, their semantics, and examples where possible.

Poorly Understood Responsibilities

It is easy to become very focused on a couple of key scenarios and to consider the functional elements only in this context. If you don't define all of the responsibilities of the elements (and don't perform traceability analysis), a lot of confusion can remain over *exactly* what each functional element is meant to do. This often leads to problems later: Either functionality is missing because it fell between the gaps, or functionality is duplicated because two subsystem development teams both thought that a piece of functionality was their responsibility.

RISK REDUCTION

- Ensure that element responsibilities are formally defined as early as possible.
- Do not allow the development process to drift into element design without element responsibilities being formally defined and agreed upon.
- Make sure that all implementers understand where their boundaries are (and why they are there).
- Make sure that all requirements have been mapped to the elements that implement them.

Infrastructure Modeled as Functional Elements

In general, you should not model underlying infrastructure as functional elements. Adding infrastructure elements to the Functional view simply makes it more confusing without adding useful information. Infrastructure can normally be hidden inside the functional elements; the Deployment view defines the infrastructure in more detail. Include infrastructure elements only if their role is important to understanding how the Functional view works (e.g., you might want to include a messaging gateway that performs some functional processing for you, but it's very rarely the case that including the application server you are using adds anything).

RISK REDUCTION

- Avoid modeling underlying infrastructure elements as you develop your initial element model. Focus on functional elements that solve part of the problem the system is going to address.
- Question the need for any elements that do not have names related to the domain of the problem being addressed.
- Address specific infrastructure concerns in another view (typically, a Deployment view).

Overloaded View

The Functional view is the cornerstone of the AD and is often the primary structuring device. However, beware of letting it become *all of* the views rather than just the central view. It is often tempting to overload the Functional view with the intent to make things clearer by adding deployment or concurrency information or other aspects of the architecture to this view. If you decide to use a compound view, make this an explicit decision. Don't allow the Functional view to simply creep into being an overloaded description of many aspects of the system. Such a description is very unlikely to be easy to understand and therefore is of limited use.

EXAMPLE Figure 17–5 shows an example of what we mean by view overloading.

This model has a number of problems (even assuming that good textual descriptions are used to back up the diagram to form a complete model). It's obviously related to UML 2, but various bits and pieces of ad hoc notation have been added: the dashed line from the Socket Library box to the Web Server box, the dashed lines within the Server Node(s) box, and so on. This means that we don't really know what the diagram

means and will have to ask the architect who drew it. We can probably discern enough for ourselves to continue, although some problems remain.

- The system provides a salesperson with an interface to allow something (perhaps a holiday or flight) to be booked.
- A number of server-side components (presumably Enterprise Java Beans, given the name used) implement something on a server computer. However, we don't know what components exist, just that (presumably) there is a group of them.
- The server components appear to be implemented by using a utility library that in turn uses a calendar library (presumably for specialist calendar processing for dates). This implies that a layered model is planned for the component design.
- A number of processes run on the server computers: one for the Web server, one for the application server, and one for the Oracle database management server. (We're interpreting the dashed lines as operating system processes.)

We can discern this sort of information from the model (and presumably could untangle the notation if we could talk with the architect); the real problem is the overloading of the diagram. Even in our initial understanding of it, we need to consider functional structure, deployment across machines, concurrency, software design constraints, and so on. These are separate concerns, at different abstraction levels, of interest to different stakeholders. The result is that none of the concerns are addressed very clearly, and this model probably can't be used with any of our stakeholders apart from developers and testers (and even they will probably need more detail about each of their concerns).

The overloading of the model is probably also one of the reasons that the notation is confusing. It is very hard to overload a diagram's function and not end up with notational confusion because of the need to represent a number of unrelated concepts together on one diagram.

RISK REDUCTION

- Remove everything from your Functional view except for items related to the functional elements and their interfaces and connectors.
- Create other views, based on the other viewpoints we define in this book, to describe the other aspects of your architecture.
- Develop the other views in parallel and cross-reference between views to illustrate other aspects of the architecture. (We talk about this in Chapter 23.)

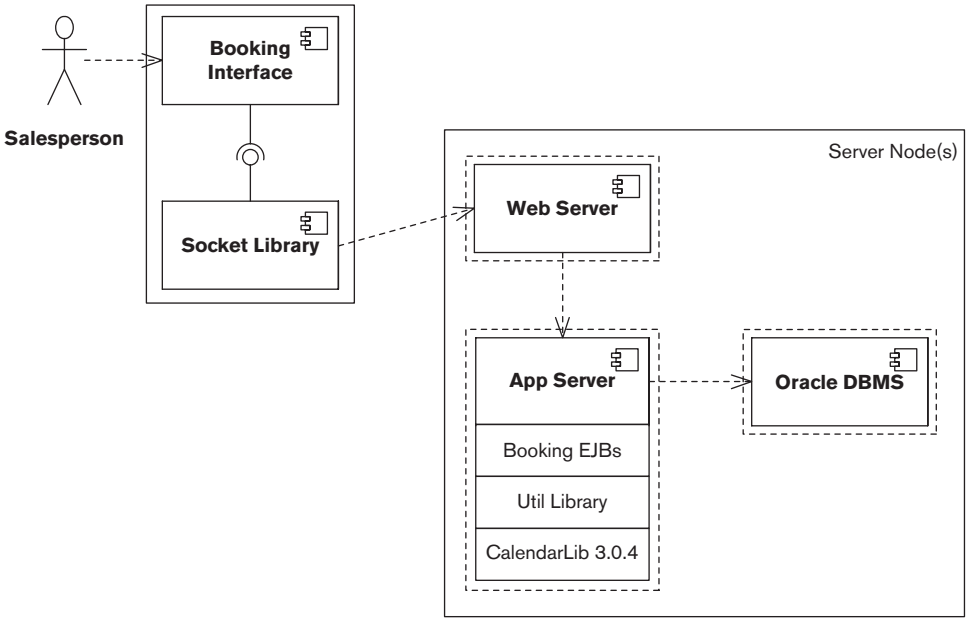

FIGURE 17–5 VIEW OVERLOADING

Diagrams without Element Definitions

When developing models that are inherently structural in nature (such as the functional structure model), there is a tendency to draw the diagram representing the model's structure and then to move on to something else without really defining the entities shown in the model. Defining each of the model elements carefully can be a tedious process, but unless this is done well, the model is meaningless.

RISK REDUCTION

- Define each element as it is added to the model, and review the definitions with your stakeholders to check that the definitions are clear and accurate.
- Do not consider the model complete until every element has a good definition.

Difficulty in Reconciling the Needs of Multiple Stakeholders

The central role of the Functional view means that most stakeholders are interested in it. This can cause you significant problems when formulating

the view—how do you create a view description that means something to all of these different types of stakeholders? End users, developers, system administrators, and all of the other groups have specific interests and needs, and you often need to communicate with each in a different way. It is often difficult to identify a single model or notation suitable for use with all of these parties.

RISK REDUCTION

- Use different modeling languages with different stakeholders. In general, stakeholders break into two major groups—technical stakeholders and nontechnical (business) stakeholders.
- You can communicate effectively with the technical stakeholders by using your primary architectural models (such as the functional structure model). Some explanation of notation may be required, but on the whole a technical stakeholder will understand these models.
- The nontechnical stakeholders are unlikely to understand your primary architectural models, so you'll need to create simplified models for them, derived from the primary models. We have found that a less technical notation (such as the sketches we described in Chapter 12) with brief textual annotation is often a more effective communication medium here.

Wrong Level of Detail

A common question when creating the Functional view is when to stop. If the process of functional analysis becomes too detailed and ends up defining too many layers of elements, you are starting to design all of the software, rather than just the architecturally significant parts. This can cause real problems, not least of which is the lack of input from the development team. Conversely, if you don't include enough detail, there is a risk that people will misinterpret your ideas and the system won't be able to deliver the qualities that you need it to. Obviously, there is no simple solution to this problem—it depends on the context.

RISK REDUCTION

- Our experience suggests that if you have to define more than two or three levels of elements, assuming a limit of about eight to ten functional elements at the top level, you may have a problem. So, if possible, keep your level of detail below this limit.
- Another danger sign can be the inclusion in the Functional view's models of details about the workings or internal structure of functional elements. If your system is very large, modeling it as a group of systems rather than working down into the elements would make the problem tractable.

"God Elements"

Software designers often see object-oriented designs that have a single huge object in the center, with lots of small objects attached to it. This situation is often dubbed the "God object" problem. The underlying problem in such cases is usually an inappropriate partitioning of responsibilities among design elements—the large object (often called "Manager") is really the entire program, and the small objects are often just data structures that this object uses. A very similar problem can exist in ADs, particularly if you consolidate too zealously (perhaps as a result of interaction analysis).

This problem leads to a situation where the system is hard to maintain because the God element is terribly complex and difficult to understand. It also results in this one component's characteristics dominating the quality properties that the system exhibits. It becomes difficult to solve related problems like performance, reliability, or scalability because they all involve changing this one system element.

EXAMPLE The UML element diagram in Figure 17–6 illustrates the sort of structure that often suggests the presence of a God element in your system.

In this situation, the Customer Management system element appears to exhibit the major characteristic of a God element, namely, nearly all interelement interactions involve it. From this structure, it is likely that the Customer Management element contains too much of the system's functionality and has dependencies with too many of the system's elements. Repartitioning the system into a set of elements with more evenly distributed functionality would make sense.

RISK REDUCTION

- Aim for a broadly even distribution of system-level responsibilities among your major elements. As a guideline, if you find more than 50% of your system's responsibilities concentrated in less than 25% of your functional elements, you may be heading toward a number of large elements and your system will lack cohesion, be difficult to develop, and be resistant to change.

Too Many Dependencies

The converse to the God object problem is static object diagrams that look like a number of spiders fighting for control. Complex interactions between elements

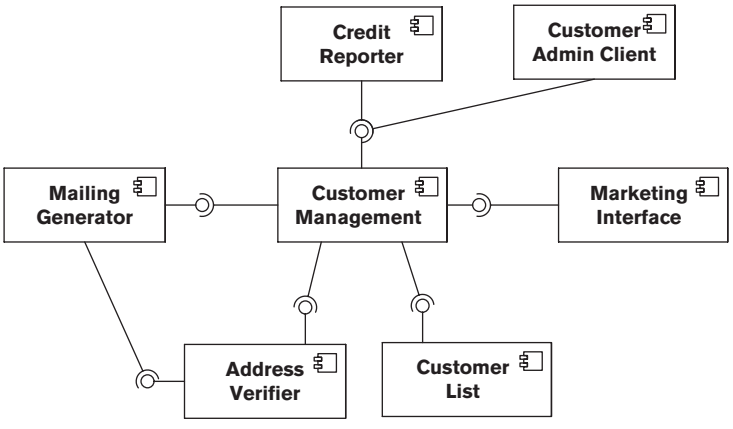

FIGURE 17–6 A GOD ELEMENT

make the system harder to design and build and may lead to a solution that is hard to change and performs poorly.

RISK REDUCTION

- This problem can often be the symptom of too many small elements in the system; practicing some judicious compression may help you resolve it.
- In general, a system element should need to be aware of the existence of only a couple of other elements in order to perform its functions. If any of your elements need to use services from more than 50% of the other elements in the system, consider revising your functional structure.

CHECKLIST

- Do you have fewer than 15 to 20 top-level elements?
- Do all elements have a name, clear responsibilities, and clearly defined interfaces?
- Do all element interactions take place via well-defined interfaces and connectors that link the interfaces?
- Do your elements exhibit an appropriate level of cohesion?
- Do your elements exhibit an appropriate level of coupling?
- Have you identified the important usage scenarios and used these to validate the system's functional structure?

- Have you checked the functional coverage of your architecture to ensure that it meets its functional requirements?
- Have you defined and documented an appropriate set of architectural design principles, and does your architecture comply with these principles?
- Have you considered how the architecture is likely to cope with possible change scenarios in the future?
- Does the presentation of the view take into account the concerns and capabilities of all interested stakeholder groups? Will the view act as an effective communication vehicle for all of these groups?

FURTHER READING

Many software architecture books focus on the functional aspects of architecture, and the subject is (rightly) central to those that take a broader view. In addition to the many books we mentioned in Parts I and II, the following are relevant to the concepts we introduced in this chapter.

Clements et al. [CLEM03] is a detailed, thorough, and practical guide to documenting various architectural styles. In the context of this chapter, the discussions of overloading views and documenting the various styles of interfaces are particularly pertinent. Garland and Anthony [GARL03] describes how to go about designing the software architecture for large-scale information systems; the approach we suggest for modeling message-oriented element interactions comes from this book. The techniques we outline for element identification are based on the architectural "unit operations" described in Bass et al. [BASS03], where they are described more fully.

Many good books explain UML in a tutorial style [FOWL03a, MILE06], and there are a number that focus on how to use it to produce rigorous architectural descriptions [CHEE01, DSOU99]. Another timeless book that explains how to produce rigorous models is [COOK94], now out of print but freely available in PDF form (www.syntropy.co.uk/syntropy). Checkland [CHEC99] presents an approach to understanding real user requirements, using an informal diagrammatic approach called the "rich picture" (analogous to our description of sketches) to help communicate with end users.

Meyer [MEYE00] is the definitive reference on Design by Contract (and much more related to object orientation), and Mitchell and McKim [MITC02] provides a nice, concise, practitioner-oriented introduction to the approach. Wirfs-Brock et al. [WIRF90] is one of the original books on responsibility-driven design, and a refinement to the approach by the same lead author can be found in [WIRF02]. Finally, Shaw [SHAW94] is one of the first written attempts to explain why connectors between elements are just as important to models as the elements themselves.

18

The Information Viewpoint

Definition	Describes the way that the system stores, manipulates, manages, and distributes information
Concerns	Information structure and content; information purpose and usage; information ownership; enterprise-owned information; identifiers and mappings; volatility of information semantics; information storage models; information flow; information consistency; information quality; timeliness, latency, and age; and archiving and information retention
Models	Static information structure models, information flow models, information lifecycle models, information ownership models, information quality analysis, metadata models, and volumetric models
Problems and Pitfalls	Representation incompatibilities, unavoidable multiple updaters, key-matching deficiencies, interface complexity, overloaded central database, inconsistent distributed databases, poor information quality, excessive information latency, and inadequate volumetrics
Stakeholders	Primarily users, acquirers, developers, testers, and maintainers, but most stakeholders have some level of interest
Applicability	Any system that has more than trivial information management needs

The ultimate purpose of any information system is to manipulate data in some form. This data may be stored persistently in a database management system, in ordinary files, or in some other storage medium such as flash memory; or it may be transiently manipulated in memory while a program executes.

Nowadays, many organizations possess massive amounts of information on their customers, their products or services, their own internal processes, and their competitors. Although some of this information may be hard to access, inconsistent, and inaccurate, it still represents a substantial asset—one that, if correctly used, can bring substantial benefits. We see this often in large systems integration projects that attempt to bring together information from a variety of sources to produce a consolidated customer view, an integrated view of the supply chain, or an accurate financial picture.

Formal data modeling and design can be a long and complex process. As an architect, you can do data modeling only at an architecturally significant level of detail. You need to focus on those aspects of the data model where getting it wrong would affect the system as a whole rather than just a part of it. Your task is to develop a summary view of static information structure and dynamic information flow, with the objective of answering the architecturally significant questions around ownership, latency, relationships and identifiers, and so forth.

You use the Information view to answer, at an architectural level, questions about how your system will store, manipulate, manage, and distribute information.

CONCERNS

Information Structure and Content

The structure and content of the information that your system manages are clearly significant concerns. Your challenge as an architect is to focus on the most important aspects of information structure, those that have system-wide impact, and to leave most of the modeling and decision making to the data modelers and data designers.

You should focus on a relatively small number of data items (entities, classes, and so on) and the relationships among them. Deciding which data items are important depends on the problems you are trying to solve and the concerns of your stakeholders. However, you should bear the following in mind when selecting the data items of interest.

- Focus on a small number of data items that are core to the primary responsibilities of your system or that your stakeholders view as particularly significant or meaningful. When considering the interests of the stakeholders, primarily consider your users, but also take into account the concerns of other stakeholder types such as maintainers.
- Focus on information-rich data items, rather than ones that have few attributes (e.g., type entities are typically less important in architectural information models). Choose data items that:
  - Are fundamental to the nature of the concerns being addressed
  - Are significant to the users or other stakeholders

  - Have a complex or poorly understood internal structure
  - Can have a significant impact on the system's quality properties, depending on how they are represented
  - Are heavily used or volatile (the contents are expected to change frequently)
- In the early stages of developing your models, try to focus on abstract rather than physical information, and keep the models simple. Don't worry too much about formal modeling techniques such as relational normalization at this point.
- Your early models should typically align with and be driven by your system's functionality, and you should be concerned less with physical considerations such as location or ownership (although we address these issues and others in this chapter).

Information Purpose and Usage

Information can be used in different ways—to support operational processes, such as taking an order or making a payment; to present current operational status, such as stock levels or production rates; or to analyze historical information and uncover trends and patterns. While it is the same information in each case, the distinction is important in the design of information systems, since the different usage patterns often have significantly different information ownership rules and may require significantly different architectural solutions.

- Most information systems have at their heart a *transaction store* or online transactional processing (OLTP) database. The transaction store manages the information required to support day-to-day operational business processes. This information is highly volatile, and the system needs to be able to process a large number of concurrent read and write operations with short latency and high reliability.
- If the system has significant reporting requirements, this can put a severe strain on the transaction store. A long-running or complex query can disrupt access by operational users, leading to increased response times and lower throughput. For this reason, some systems implement a separate *reporting database* to service these large queries, which is fed in batch or real time from the transaction store. The reporting database is essentially read-only (apart from the incoming information feeds) and is optimized for complex ad hoc queries rather than updates, with many indexes and significant denormalization.
- The transaction store and reporting database usually store only information related to current activity, such as open orders, current stock levels, or today's prices. Some users require access to historical information, to look at individual transactions or to analyze and summarize the information in different

ways. Historical information is usually managed in a separate *data warehouse*, sometimes called an online analytical processing (OLAP) data store. The data warehouse may in turn feed into more specialized *data marts*, which manage information from a specific domain or time period. The data warehouse holds a record of all activity going back many years and can be used to retrieve specific historical information or to analyze trends over time.

- Most systems rely heavily on *reference data* (sometimes known as *static*, *master*, or *lookup data*), which is the information on people, places, and things that categorizes or classifies the system's transactional information. It includes a wide range of business entities, such as calendars, customers, products, parts and supplies, prices, locations, employees, and external organizations. It also includes the "type" information (such as product type or employee role) that characterizes other information. Every organization has its own definition of what it classes as reference data, but it is almost always fairly static, changing relatively infrequently, and there is usually much less of it compared with transactional and operational information. As we will see shortly, reference data may not be owned by your system, which can be a significant architectural challenge.

While the distinction here may not be important in the early days of an information system, over time the system will amass larger and larger volumes of data. It will be much easier to hive off a separate reporting database, data warehouse, or enterprise data store in the future if the initial architectural design has taken this possibility into account and allowed for the impact of partitioning, speeds of different stores, data duplication between stores, and so on.

Information Ownership

In many architectures, particularly those that involve the integration of new and/or existing systems, information is physically distributed across multiple data stores and accessed in different ways. This situation, while often unavoidable, creates all sorts of problems.

- Which copy of a particular data item is the most up-to-date one?
- How do you keep synchronized any information held in multiple places?
- How do you deal with information that is derived from information managed and owned elsewhere, such as account balances derived from account activity?
- What validation and business logic should be applied to the modification of data items, and what assumptions can be made about data items that have been validated elsewhere?
- If the same data item can be modified in several places, how are conflicts reconciled?

EXAMPLE An insurance company employs a large number of workers who visit customers at home to sell them financial products. The company maintains a central database of customers and prospects, an extract of which is downloaded to each salesperson's laptop when visiting the office. Whenever a sale is closed at a customer's home, the information is stored in a holding area on that laptop until it can be uploaded to the central database later.

The company opens a call center that allows customers to update their details and also offers limited capabilities to sell products. This leads to an increase in the number of complaints for various reasons. Sometimes, details stored on laptops overwrite more recent data on the central database, and vice versa. In other cases, updates to the central database are rejected because they fail the central system's more stringent validation.

In order to address these problems, the architect first has to agree with the business stakeholders on some general rules about how to deal with update conflicts and failures (e.g., recent updates always override older ones). These rules are then coded into the central system and laptop applications.

A useful way to analyze these problems and develop architectural strategies to handle them is to develop a model of *information ownership*. The information owner (or master) of a data item is the system or data store that contains the definitive, up-to-date, validated value of that data item. The information owner always has the correct value for that information and can act as the final arbiter when any disputes over accuracy occur.

By defining the owner of each data item, you can ensure that your information consumers are always working with the right information and that your information producers write it only to the correct place. When this is not possible in practice, you can analyze potential conflicts and inconsistencies and then develop strategies to deal with them.

EXAMPLE A national system for registering motor vehicles operates from a number of semiautonomous regional centers. Each center is responsible for registering vehicles purchased in that region. Each vehicle must be allocated a unique number, but conflicts could arise because there is no real-time communication between the regional centers. (In information ownership terms, each center is a creator of the vehicle registration number data item.)

> The problem is resolved by partitioning the information ownership, that is, by allocating to each center a separate, distinct range of numbers to assign to vehicles purchased in its area. Care must be taken to ensure that the ranges will never overlap. This is done by making each range far larger than the anticipated number of cars to be registered: The North center is given the range 1 to 100 million, the West center 101 million to 200 million, and so on.

A by-product, incidentally, of your information ownership analysis will be a high-level definition of some of your system's interfaces. Where one system is an information owner and another is an information consumer (or maintains a copy of that information), some sort of interface is required between them. You can use the interface definitions to cross-check the models in your Information view against the models in your Functional view. Any interface derived from information ownership rules should also exist as a process flow between the two participants.

Enterprise-Owned Information

Nowadays many large organizations maintain "enterprise" sources of important information, and you are usually required to use them rather than owning and managing such information yourself. Enterprise information is usually highly valuable to the organization, and the consequences (to you, and to the organization as a whole) of it being incorrect or out-of-date are severe.

The most common form of enterprise information is enterprise reference data. (As we described earlier, reference data is the information on people, places, and things that categorizes or classifies your system's transactional information.) This may be general-purpose information, such as country codes or currencies, or it may be specific to your organization, such as products, suppliers, or customers. You may also need to make use of more volatile enterprise information, such as end-of-day stock levels or account balances.

Your system may be expected to access enterprise information directly from the source system when it needs it, or it may be required to maintain its own copy that is refreshed regularly in real time or batch. In some cases your system may also need to update the enterprise information itself, using standard mechanisms and business processes defined by the information owner.

In any case, the enterprise information your system uses must be accurate, up-to-date, consistent, and complete. There are several ways this can be achieved, each of which has implications for users as well as for the architecture.

EXAMPLE A travel agency has branches across the country and also sells directly to customers over the Internet and from a call center. The travel agency has started a customer affinity program and wants to build a system to make holiday recommendations to select customers based on their preferences, budgets, and travel history. The system will make use of various types of enterprise reference information, including details of holiday destinations, tour operators, airlines, and hotels. In addition, it will use more volatile enterprise information on standard pricing plans and special offers.

All of this enterprise reference information is held in central data repositories but needs to be managed in different ways. Information on holiday destinations, airlines, and tour operators changes rarely, and a copy can be downloaded to the system's own database weekly. Hotel information and list prices are more volatile, and an overnight extract is required. Special offers arise at short notice, and a "semi-real-time" feed of these is needed (in reality, a small batch extract that runs at regular intervals during the day).

Affinity customers sometimes like to suggest hotels they have used in the past but are not on the travel agency's database. In this case the system needs to be able to upload the hotel details to the enterprise store, and after some validation these should be added so that they are available for other systems to use.

As discussed elsewhere in this chapter, each of these different access models has its advantages but also may lead to problems. Data that is refreshed on an overnight batch schedule may be out-of-date when it is used. Obtaining data in real time mitigates this problem but is more complicated to implement and manage. Accessing a single central repository ensures that data is always up-to-date, but the repository becomes a bottleneck and a single point of failure, and it may not be feasible to do this for systems that are geographically dispersed.

We address some of these concerns further in our discussion of the Location perspective in Chapter 29.

Identifiers and Mappings

Whether information is managed by using relational entities or objects and classes, each data item needs a unique identifier or key that distinguishes it from others of similar type (e.g., customer number, machine serial number, or ISBN). In relational database terminology, this is called a *primary key*; in object-oriented programming, the term *object ID* is often used; a more useful general term (which does not assume any underlying information model) is *identifier*.

When information is spread over multiple repositories, identifiers often become an issue. Different systems may use different mechanisms to identify the same data item, and these mechanisms will need to be reconciled at points where data exchanges occur. Because key assignment can be a volatile activity (consider a sales system where many new orders are created per second), you will need to keep this reconciliation process up-to-date with new information as it arrives.

EXAMPLE A newspaper captures sports information submitted by journalists along with results and scores that arrive electronically. The paper collates the information and publishes daily league tables for individual competitors and teams. Although the paper's own central database allocates identifiers to each competitor and team, most of the information sources refer to them only by name—and in the case of foreign competitors, these names are not always spelled correctly.

The database is suffering some significant information quality issues. Scores and results are sometimes allocated to the wrong player or team, phantom teams with spellings similar to real ones are created regularly, siblings' results are often allocated to the wrong person, and some results fail to be loaded at all.

Problems like these can often be only partially addressed by architectural capabilities and features. Defining standard identifiers for teams and players in this example will help, but business process changes will also be required to ensure that users of the system carefully map names to their correct identifier—perhaps by being required to pick names from a drop-down list rather than type them in directly. However, imposing rules like these can make a system awkward to use, and you should collaborate carefully with your business stakeholders to come up with a solution that is both usable and effective (perhaps using an exception workflow to confirm the correctness of automatically matched identifiers, allowing partial automation with manual input to ensure data quality).

There are many other architectural challenges associated with the use of identifiers. For example, identifiers are normally *invariant*, that is, they never change over the lifetime of the data entity that they identify. However, it is not always possible to enforce this rule. In such cases, the mechanisms (and business processes) for creating and changing identifiers must be very carefully specified and designed.

There can also be some subtleties around the question of whether two data entities actually represent the same thing and should therefore have the same identifier. For example, every book is allocated an ISBN (International Standard Book Number) when it is published. A second edition of the book

EXAMPLE *Derivatives* are financial products whose value is derived from the value of some other underlying asset. For example, a *share option* gives the purchaser the right, but not the obligation, to buy an agreed-upon number of shares at an agreed-upon price at an agreed-upon date in the future. The derivatives market is constantly changing, with new and more complex products being introduced all the time.

When a new derivative product is created, it goes through an approval process to ensure that it is sound, that it is compliant with regulations, and that its financial parameters are clear. This process can take a relatively long time, and in the interim it is common for the product to be allocated a temporary identifier so that a provisional price can be quoted and measures of value and risk can be calculated. Once the product is formally approved, it is given a permanent identifier, which may be different from the temporary one since it is allocated by a different part of the organization.

A link must be established between the two identifiers, so that the provisional quote can be turned into a firm quote and a sale made with a clear audit trail.

may contain only minor revisions and corrections, or may be substantially different, with a new structure and a substantial amount of new content. Should such a major revision be allocated a new ISBN? If so, how can it be linked to the ISBN of the first edition? If not, how are the two editions distinguished from one another? In this example, there are agreed-upon rules about allocating ISBNs, but in many cases it will be down to the architect to decide (or at least capture and agree on the requirements from users).

Another important consideration is whether your identifiers are going to be *user-visible* or not. For example, every debit and credit card has a unique 16-digit card number that the cardholder uses when making a purchase online or over the telephone. On the other hand, although each individual purchase on a credit card statement has its own identifier, this is not usually printed. If a transaction needs to be queried or confirmed, it is identified by the transaction date, the merchant name, and the amount (which is usually unique enough for this purpose).

Volatility of Information Semantics

It is common nowadays for the syntax, semantics, and interrelationships of business information to undergo frequent and unpredictable change. New fields may need to be added to existing entities, new constraints and relationships may arise, or new types of entities may be needed to meet changing business needs.

Although there are mitigation strategies to make such changes less painful (including abstract database access libraries, tools for impact analysis, and

designing interfaces to allow for variation and change), even small changes to an information model can have wide-ranging implications for the systems that use that information. For example, if a new mandatory field is added to a database table, every process that creates or updates rows in that table needs to be changed so that it can provide a value for that field. This process needs some form of control, traditionally managed through a formal process of *data model change control*: The impact of a change on every module in the system is assessed, and only when all parties have implemented the required functional changes is the database change rolled out.

This approach is established and effective, but it drastically slows down the rate at which systems can be changed, and in practice change control often ends up being subverted or bypassed altogether. An alternative approach, which is more flexible while still retaining a level of control, is to decouple the information semantics from the physical structures used to store it. A common way of doing this is to store complex information structures in structured text forms such as XML, JSON, or YAML, either within a database or in external data files. With a disciplined approach, and the possibilities that exist today for automation, you can also take a more dynamic and flexible approach to changing a database schema, as proposed by the Evolutionary Database Design technique (see Further Reading for more details).

The XML family of data management standards includes mature mechanisms for defining the schemas of XML documents and accessing their contents. While changes to the schema still need management and oversight, they can often be implemented more quickly with less effort. The downside of this approach is that XML-based systems tend to be less performant and scalable, due to the XML management overhead and the fact that most database optimizers don't work very well with XML data.

Information Storage Models

The third-normal-form relational database is so dominant in enterprise information systems that it can be easy to forget that there are other approaches available for storing information. The following four major types of information stores are all in wide use today.

- *Relational databases* dominate the enterprise information systems landscape and need little introduction. A typical relational database contains a largely third-normal-form schema and is usually used as some form of transactional or operational data store. Relational databases are usually implemented using a third-party database management system and allow data retrieval and manipulation operations to be expressed in a declarative form using the SQL language. They typically enforce data integrity via an ACID transaction model (meaning that database transactions are used to ensure that updates are Atomic, Consistent, Isolated, and Durable—hence

ACID). Well-designed relational databases avoid data duplication (via normalization), are flexible (due to the ability to write queries in an unconstrained manner across the data model), can provide good performance and scalability characteristics, and are relatively easy to use for small and midsize problems. The limitations of a relational database tend to be the difficulty of scaling them to very large problems and the complexity of the schema and queries that often results when implementing a large enterprise application.

- *Dimensional databases* are another storage model based on the relational storage model and can be implemented using standard relational database engines, although specialized column-based or dimensional stores are often used instead. Rather than using a third-normal-form schema, a dimensional store is based around a multidimensional (or "star") schema model, with large "fact" tables containing the primary data in the database, linked to small "dimension" tables that contain classification data that can be used to group and summarize the fact data. (We describe multidimensional schemas in the Static Information Structure Models section later in this chapter). Dimensional databases are particularly well suited for complicated reporting problems, and so this storage model is often used for reporting databases rather than transactional databases. The major limitation of a dimensional model is the relative difficulty of updating information after it has been added to the database.
- *NoSQL databases* are a relatively recent development and at the time of writing are still fairly rare in mainstream enterprise systems, but they have proved their usefulness in many very large-scale Internet services for e-commerce, Internet search, and social networking.[1] There are many data storage technologies that classify themselves as "NoSQL" products, and each one has its own unique characteristics, strengths, and weaknesses. What is common among the NoSQL products is the fundamental tradeoff they have made, which is to abandon the traditional RDBMS characteristics of strict tabular data storage and SQL-query-based data access (and in some cases ACID transaction semantics) in order to achieve simplicity and very high scalability and performance. Most of these databases are accessed via a simple "map"-based interface that allows records to be stored and retrieved by key, sometimes also offering simple query facilities based on the attributes of the records being

1. In fact, the very first commercial database management systems were network and hierarchical databases, which also didn't use SQL. Since then, object-oriented databases, which also don't use SQL, have come and gone too. Here we're referring to the more recent database technologies aimed at solving very large, distributed data management problems that have been developed primarily to meet the demands of Internet-scale systems.

retrieved. This simpler model of data storage allows the database engine to be distributed across a very large number of servers, a configuration that provides good performance and a high degree of scalability. Of course, if you decide that you need a rich strongly typed database, or a powerful query-processing engine, these technologies are less suitable.

- *File-based stores* shouldn't be forgotten either, and even today, a surprising amount of enterprise data can usually be found stored in flat files. Files have the benefits of simplicity and ubiquity and, for some situations, the best performance too. They are particularly well suited to "write-once" requirements such as logging and auditing. Nearly every technology can read and write files directly, and there are a number of simple query engines that can be used with flat files to simulate a database. Of course, the simplicity of flat files can also make them unsuitable for many demanding tasks, where complicated queries, reliable transactional updates, or complicated data structures make the use of files difficult.

As an architect, you need a good awareness of the different information storage models available to you, and you should carefully consider the needs of your system so that you can match the right sort of storage model to your data storage requirements.

Information Flow

Just as important as the static information structure is the way that information moves around the system and is accessed and modified by its elements. The important questions here include the following.

- Where is data created and destroyed?
- Where is data accessed, modified, and enriched?
- How do individual data items change as they move around the system?

As with information structure, it is usually necessary to consider only the most important information flows as part of architecture definition, that is, those that are crucial to the system's primary responsibilities or those that will have a material impact on its quality properties. In any case, because you will have only a high-level data model to work with, you won't be able to drill down into too much detail here.

Because the main purpose of most systems is to process information, information flow is often analyzed within Functional rather than Information views. This works well as long as you don't end up with a small number of complex, overloaded models that are hard to understand—and as long as you make sure that the data-specific concerns discussed in this chapter are also addressed.

Information Consistency

Information consistency means that information held in different parts of the system, or in different but related data items, should be compatible, congruent, and not in conflict. This may be as simple as a referential integrity constraint (e.g., if a customer is recorded as owning several products of specific types, these products should all exist) or may be more subtle and complex (e.g., a summary financial position should always match the underlying data used to calculate it). Most businesses have sophisticated rules for information consistency, although it is rare in our experience for these to be written down anywhere.

Information consistency is so fundamental to the operation of modern relational databases that its significance in the architectural context can easily be forgotten. A classic example, which we repeat here, illustrates its importance.

EXAMPLE A bank customer uses an automated teller machine to transfer $500 from her checking account to her savings account.

The bank uses two data stores, CHECKING and DEPOSIT, to manage these two different types of accounts. The transfer is implemented as two updates: a withdrawal of $500 from CHECKING, and a corresponding deposit of $500 into DEPOSIT, as shown in Figure 18–1.

It's essential that either both of these updates complete successfully or neither of them do. For example, the transaction might not go ahead if the customer doesn't have sufficient funds in her checking account. If only one of the transactions completes, either the customer or the bank would lose money.

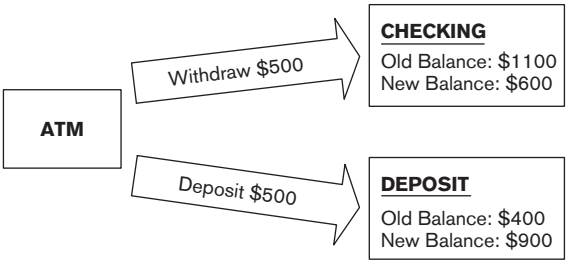

FIGURE 18–1 TRANSACTION MANAGEMENT FOR FUNDS TRANSFER

A *transaction* is a sequence of data updates that occur as an atomic unit—that is, either all updates are accepted and written to permanent storage or none of them are. Transaction management ensures the right outcome by *committing* updates (writing them permanently to disk) only if all updates can be successfully applied. Transaction management will *roll back* (undo) all of the updates if one of them fails.

Transaction management features are provided by all modern relational database systems, and their use is nowadays almost automatic (although care must be taken to avoid pitfalls such as heavy contention or deadlocking). Transaction management across multiple systems is much more complicated to design, build, and operate, requiring complex techniques such as two-phase commit. Such techniques can impose a heavy burden on processing power, leading to increasing latency and response time, and you should use them only when absolutely necessary.

An alternative approach that avoids some of the difficulties with distributed transactions is to use *compensating transactions* to maintain data integrity. In this model, each data update is committed individually, and if a later update fails, each committed update is reversed by a transaction with an equal and opposite effect to the original one. In the preceding example, if the withdrawal was successful but the deposit failed, a compensating deposit of $500 to the checking account could be applied to bring everything back to a consistent state.

Compensating transactions often work better in practice than two-phase commit since they do not require database locks to be held over separate data stores at the same time. However, they have problems of their own, particularly if changes cannot easily be reversed or if a compensating transaction itself fails.

Another approach is known as *eventual consistency*. In this model, distributed applications favor high availability over consistency and are designed to be able to cope with data that is out of synch for a period of time. Such a system guarantees that after an update, all instances of the same data will eventually be updated to this value, without guaranteeing how long this will take.

Eventual consistency is used for infrastructure software such as DNS (the Internet's Domain Name Service) and for some Internet-scale applications such as global search engines, e-commerce sites, and social networking sites, but the principles may also be useful to smaller-scale applications. The model is sometimes referred to as following BASE principles (Basically Available, Soft state, Eventual consistency) in contrast to traditional transaction management, which is referred to as ACID (because, as we noted previously, the transactions are Atomic, Durable, Isolated, and ensure that data is Consistent).

We also discuss the application of this technique as a way of scaling to very large data volumes in Chapter 26 and its relevance to achieving high system availability in Chapter 27.

Information Quality

The quality of a particular data item is the extent to which the current value of that data item agrees with the correct value in the real world. Poor-quality information can have a significant impact on an organization's ability to carry out its operations. If you don't have accurate information about your customers, for example, you risk annoying them, losing them, or even being sued by them. (Given all this, it is still a surprise how many systems manage to survive on information that is incomplete, incorrect, or outdated, proving the old adage that something is often better than nothing, even if it is imperfect.)

EXAMPLE A mail-order furniture company has created a marketing database from customer orders and requests for brochures or quotations. It uses this customer database to phone customers about special offers and to try to persuade them to buy more of the company's products.

Unfortunately, the data in the marketing database has been cobbled together from a number of sources and is therefore outdated and inaccurate. Moreover, a number of customers have asked not to be cold-called, but these requests have not always been transferred from the spreadsheet where they're managed into the marketing database.

As a result, many customers receive cold calls who do not want them, or are offered products they already own, or are offered unsuitable products (e.g., those that are too expensive). This creates a significant amount of dissatisfaction among existing and potential customers, leading to bad publicity and possibly to lost sales.

Information quality becomes an issue for you as an architect in cases where the system makes use of information from a variety of sources, particularly when some of these are external to your sphere of influence. If your information quality is variable, you must consider such issues as the following.

- How will information quality be assessed and monitored (especially when information is frequently updated)?
- What minimum information quality criteria must be met?
- How will these criteria be enforced?
- How will poor-quality information be improved? Will this be done in an automated way, or will it require manual intervention?

- Can good-quality information be corrupted by information of lesser quality? (For example, a customer address is updated, but the postal code is omitted.) If so, should this be prevented or checked?
- Is it possible for information quality to degrade as it flows around the system?

The answers to these questions are likely to have implications for your architecture. For example, it may be necessary to develop or deploy automated tools for monitoring or assessing information quality or for repairing poor-quality data. If repairing data needs some human intervention, you may have to set up a holding area where data can sit until it has been manually repaired.

It is becoming more common to use *workflow* to address information quality problems when repair processes cannot easily be automated. In this model, a list of tasks, such as correcting a customer's name or address or dealing with a suspect transaction, is managed in a central database. Tasks are assigned to users and the system tracks their status to completion. Tasks can either be standardized (defined at design time) or, in the most sophisticated workflow systems, ad hoc (created by someone at runtime). Service levels may be defined that commit the company to fixing problems within a certain time or at a certain rate.

If well designed, this approach can be an effective way of improving information quality and customer satisfaction.

Timeliness, Latency, and Age

If your information is held in a single data store and always accessed synchronously in real time, timeliness, latency, and age may not be significant issues. Unfortunately, many systems do not work this way, and it is inevitable that some scenarios involve information that is old or out-of-date, if only by a few minutes.

EXAMPLE A commodity brokerage accepts a number of feeds from information sources that provide up-to-date pricing and volume information, as well as news stories relevant to the commodities being traded. The feeds are all channeled through a single gateway application that sorts, filters, and distributes the information to appropriate subscribers.

A catastrophic hardware failure renders the gateway unavailable for several days. When it comes back online, the subscribers are flooded with several thousand cached price messages that, because they are several days old, are of no interest to the recipients.

The gateway is modified so that after a failure, it discards cached price messages that are older than a certain configurable age. Another failure occurs (a change of hardware supplier is called for), and recovery is much faster.

In this example we have separate *information providers* (the external systems that provide pricing and volume information) and *information consumers* (the internal users who make use of it). Because the process of information transfer from provider to consumer takes a finite (and possibly long) time, discrepancies can occur. If the time lag cannot be reduced to close to zero, you need to work with stakeholders to develop solutions to the problems that may arise from inconsistent information.

The time lag between the visibility of information to providers and to consumers is expressed by means of *latency*, the length of time between a data item being updated at the data source and the updated value being available to all parts of the system.

You may also need to take into account the age of some data items (the time since the data item was last updated by its data source). A system that disseminates information on volatile stock prices, or the physical location of trucks, for example, may not be interested in information that is hours or even minutes old. You may be able to discard this information because it is no longer needed.

You should identify key points where time-based inconsistencies can arise and, with the help of your stakeholders, develop strategies to handle them, such as the following.

- Tag important data items with a "last updated" date and time.
- Define "currency windows" for significant data items.
- Warn users when information may be outdated.
- Hide or discard information that may be too old.
- Reduce latency by means of faster interfaces or direct access to data sources.

Archiving and Information Retention

In many systems, it is becoming rare for information to be deleted; it may be kept for legal reasons or for historical analysis. Although disk storage is now relatively inexpensive, managing large databases is a complex process and even enterprise disk architectures cannot expand indefinitely, so sooner or later your information will grow to a point where it is not desirable to keep it

all online. Then you will need to archive older, less useful information to some other storage medium such as high-capacity offline storage.

You must define carefully the scope of information to archive. It obviously can't be information that is still needed to support any production activities, nor should it be information that is likely to be useful for regular analysis. Information is usually selected on the basis of age combined with business rules to determine its usefulness.

Your archiving strategy can have a significant impact on your architecture.

- Archiving large volumes of information may make some systems fully or partly unavailable for significant periods of time.
- Your physical disk sizing needs to take into account the length of time that information will be retained.
- You may need to define the processes that move production information to archive media.
- You may need to take special actions to ensure the integrity and consistency of the production and archive storage.
- There may be an impact on the network infrastructure if archive storage is remote.

Don't try to add archival capabilities as an afterthought. Design your architecture from the beginning in such a way that archiving is a natural part of the information lifecycle.

Stakeholder Concerns

Typical stakeholder concerns for the Information viewpoint include those listed in Table 18–1.

TABLE 18–1 STAKEHOLDER CONCERNS FOR THE INFORMATION VIEWPOINT

Stakeholder Class	Concerns
Acquirers	Concerned with preserving and safeguarding the value of the organization's information assets, so the following are key (although not always recognized as such): • Information quality and archiving • Reference data • Information retention
Assessors	Interested in all aspects, with a focus on information structure and flow, identifiers and mappings, and information quality
Communicators	Rarely focus on detail on the information architecture, but may find a background understanding of the key principles and strategies helpful

TABLE 18–1 STAKEHOLDER CONCERNS FOR THE INFORMATION VIEWPOINT *(CONTINUED)*

Stakeholder Class	Concerns
Developers and maintainers	Interested in how the architect's models will translate into real databases and (real-time, batch) information interfaces, and implementation details such as how the data structures will support the required processing and how consistency will be guaranteed
System administrators	Interested in how these real-world system components will be managed and supported
Testers	Interested in the main database structures, how they are affected by the operation of the system, the data flow through the system, and how to create realistic test data sets
Users	Concerned with functional aspects of the information architecture (e.g., information ownership and regulation) and user-visible qualities such as timeliness, latency, and age; and information quality

MODELS

Data modeling is probably the best-served area of information systems in terms of established, rigorous, and generally understood analysis and modeling techniques. The three most important types of models are the following:

1. *Static information structure models*, which analyze the static structure of the information
2. *Information flow models*, which analyze the dynamic movement of information between elements of the system and the outside world
3. *Information lifecycle models*, which analyze the way information changes over time

We discuss these models in this section—particularly how they are used in the architectural context—and briefly describe some other types of models you may find useful, such as information ownership models, information quality analyses, metadata models, and volumetrics models.

Static Information Structure Models

Static information structure models analyze the static structure of the information: the important data elements and the relationships among them.

Entity-relationship modeling is an established technique of data analysis that is based on a solid underlying mathematical model. Data items of interest

are referred to as *entities*, and their constituent parts are called *attributes*. The information semantics defines the static *relationships* among entities. Each relationship has a *cardinality*, which defines how many instances of one of the entities can be related to an instance of the other.

EXAMPLE A library stores a number of *books* for its *members*. Members *check out* books for a period of time, after which they are renewed or returned. Each book has one or more *authors*, who receive a fee each time a book is checked out. The fee is paid to the author via the book's *publisher*.

Each of the italicized terms in this description is represented as an entity in the entity-relationship model. Attributes of the model include book title, author name, ISBN number, and publisher name and address.

Class models perform a role similar to that of entity-relationship models but for the object-oriented world. They model data items (*classes*), their constituent data parts (*attributes*), and the static relationships among them (*associations*). It is possible to use class model notation to model relational entities by omitting the behavioral aspects from the model and limiting the association types (e.g., no generalization or composition).

Class models can also document the behavioral aspects of a system, such as interfaces and methods, and features specific to object-oriented analysis, such as inheritance.

EXAMPLE In the previous example, classes would be modeled for *books*, *members*, *authors*, and *publishers*. Methods would provide the necessary functionality for *checking out* books.

NOTATION There are a number of similar notation styles for documenting entity-relationship models. Figure 18–2 shows an entity-relationship diagram in the crow's foot style for the library example.

A UML class model for the same example would look something like Figure 18–3.

Data warehouses and data marts are usually modeled using more specialized semantics called a *star schema* (also known as a *multidimensional schema* or *cube*). A star schema consists of *fact tables*, which contain numerical data or other "facts" aggregated at many different levels and have large compound keys. Clustered around each fact table are a number of *dimension tables*, which model the different levels at which information can be aggregated. The chief advantage of using a star schema is that an aggre-

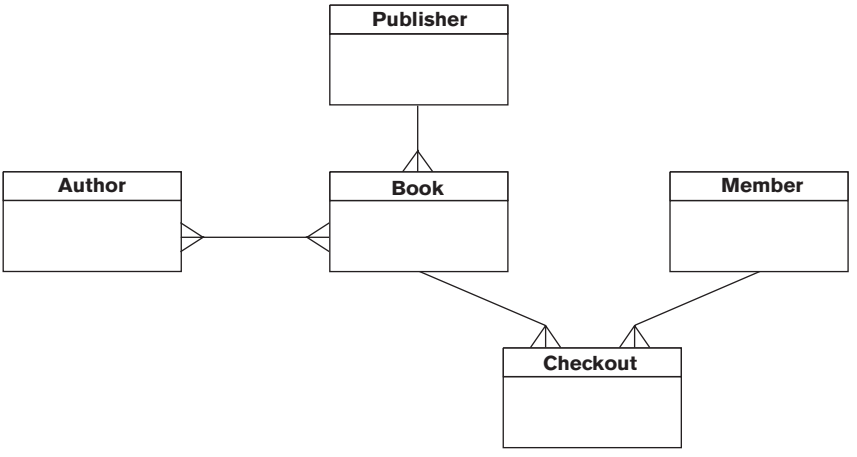

FIGURE 18–2 ENTITY-RELATIONSHIP DIAGRAM FOR THE LIBRARY EXAMPLE

gated value can be retrieved in a single database read, rather than querying and summing all the underlying transactions. A snowflake schema extends this model by normalizing the dimension tables into a hierarchical structure.

An example star schema for the library system is given in Figure 18–4 (although in practice a library management system is unlikely to need to manage the sort of volumes that would necessitate a data warehouse).

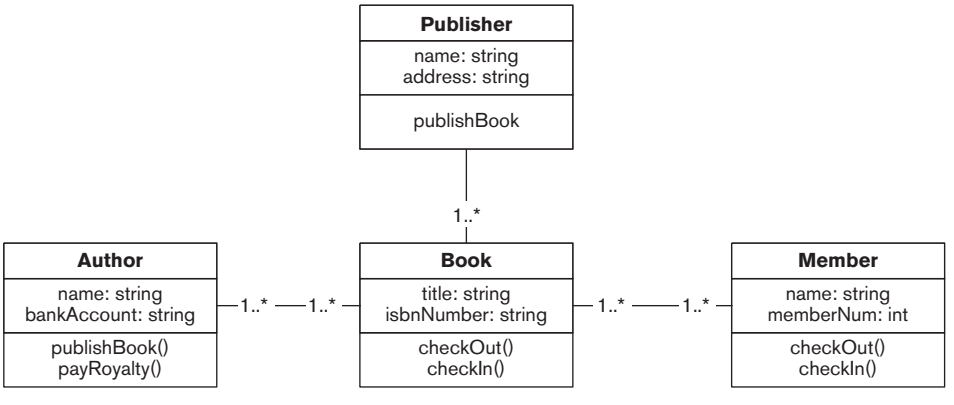

FIGURE 18–3 UML CLASS MODEL FOR THE LIBRARY EXAMPLE

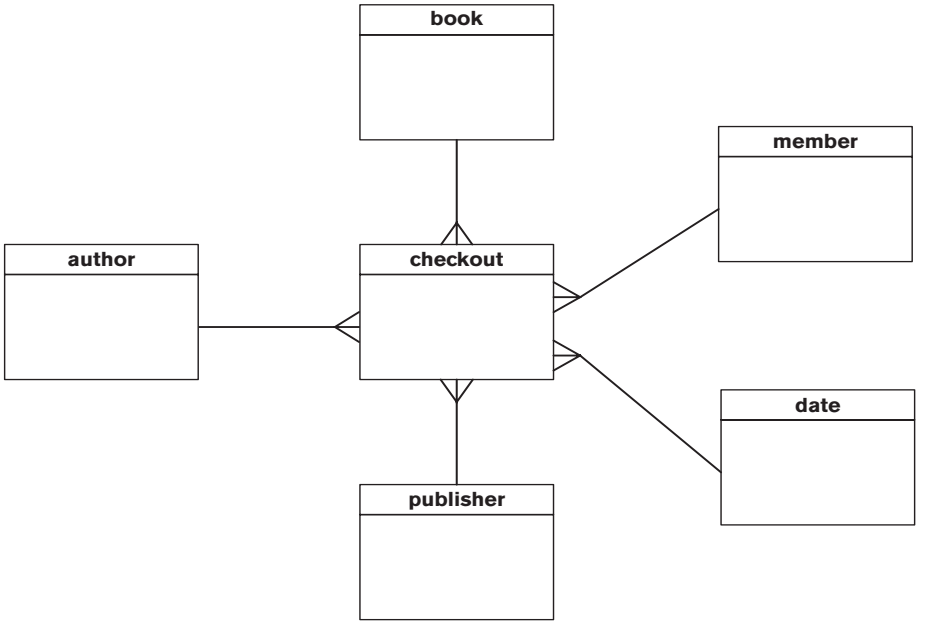

FIGURE 18–4 STAR SCHEMA ENTITY-RELATIONSHIP DIAGRAM FOR THE LIBRARY EXAMPLE

ACTIVITIES Formal information modeling includes a wide range of activities.

- The first step is to identify the important *data entities*. This is usually done by inspecting the business processes and use cases for nouns such as customer, product, payment, or event. In an architectural description, you should focus on a small number of important entities (for example, anything with a "type" in its name can usually be ignored).
- A process called *normalization* reduces the model to its purest form, in which there is no repeated, redundant, or duplicated information. It is rare for relational models to be taken beyond third-normal form, and from the architect's perspective it is often more useful (although less rigorous) to model some information unnormalized.
- *Domain analysis* looks at attributes (fields) of data items and the rules that define their permissible values. For example, a customer number may always be a ten-digit integer with the last digit being a check digit, or a telephone number is always a country code followed by a dialing code and a number. Domain analysis is important in schema design but is usually too detailed for an AD.
- Techniques such as *structural decomposition* or *aggregation* are used to derive class models. Structural decomposition involves breaking an

element into smaller coherent pieces, while aggregation is the reverse process—creating a new element by combining other, similar elements.

Unfortunately, static information structure models are not easily decomposed into levels of detail—for entity-relationship diagrams in particular, it is, in theory, "all or nothing." In practice, you do not have time to produce a hundred- or maybe thousand-entity information model as part of your architecture. The way to approach this is to focus on a small number of the most important entities/classes and the relationships among them.

You can usually omit from your model detail such as intersection entities (replace these with nonnormalized, many-to-many relationships, as we did in the entity-relationship diagram shown in Figure 18–3 between author and book) and type entities (such as product type).

As a very general guideline, if you have more than about 20 to 30 entities, or if your entity-relationship diagram won't easily fit on a single page, you have probably presented too much detail. In this case, you need to either remove some less important entities from the model or use partitioning and/or decomposition to simplify the overall picture.

Information Flow Models

Information flow models analyze dynamic movement of information between elements of the system and the outside world.

These models identify the main architectural elements and the information flows between them. Each flow represents some information transferred from one component to another—in other words, an information interface. Associated with each flow is a direction, the scope of the information transferred, volumetric information, and (in a physical model) the means whereby information is exchanged, whether it is a transfer of flat files or a real-time exchange of XML messages.

EXAMPLE A publisher supplies *lists of newly published books* to libraries in a PDF document that is mailed to librarians monthly. When a library receives a book, it is accompanied by an *electronic delivery note* in the form of an XML file, which is imported directly into the library's book management system. When books are checked out and back in, the *new state* is recorded by means of bar-code readers. When a book is disposed of, it is manually *marked as deleted* in the system by a PC application that accesses the database directly.

Each italicized term represents an information flow into, out of, or around the system.

As with static information modeling, you should aim to keep your information flow models high-level and simple. It is not necessary to provide much detail at the architectural stage. Fortunately, most notations support this naturally through decomposition.

Information flow modeling is most useful for data-intensive systems, and it complements the modeling of interfaces and function invocations in the Functional view (see Chapter 17), which is often more appropriate to processing-intensive systems. In practice, you usually do only one or the other, depending on the nature of the system, the skills of the architect, and the interests of the key stakeholders.

NOTATION There are a number of information flow notations from classic systems analysis, such as Gane and Sarson or SSADM data flow diagrams, although these are as much about process as about information flow. Figure 18–5 shows an example of a data flow diagram.

The following notation is used in the diagram.

- Large rectangles represent processes that manipulate information.
- Narrow open rectangles represent data stores (logical or physical collections of information).
- Arrows represent information flows.

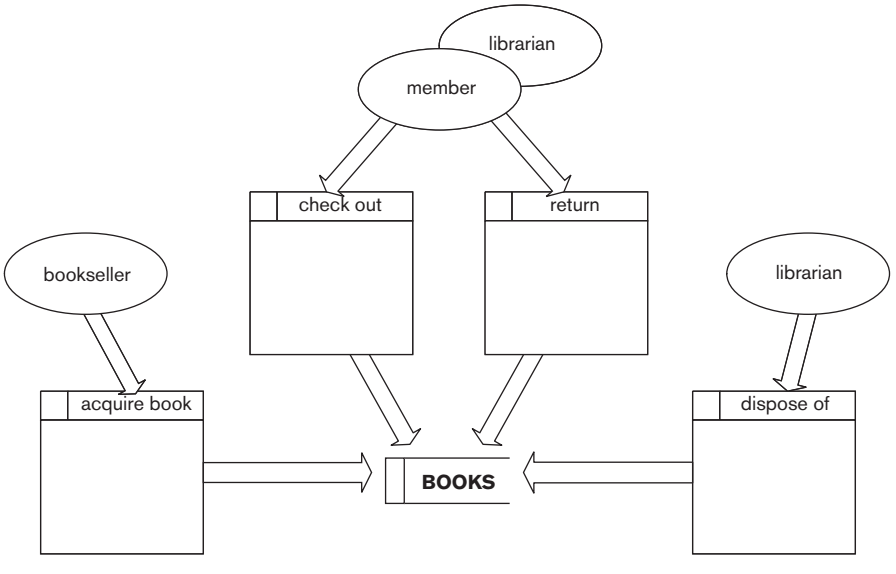

FIGURE 18–5 DATA FLOW DIAGRAM FOR THE LIBRARY EXAMPLE

- Ellipses represent external entities (people or other systems that interact with this system).

The diagram conveys several pieces of information.

- Members and the librarian provide information to the checkout and return processes.
- A bookseller provides information to the acquire book process.
- The librarian provides information to the dispose of process.
- All this information is written to the BOOKS data store.

Information flow is usually represented in UML using activity diagrams, which include the same sort of elements as shown in Figure 18–5.

ACTIVITIES Information flow models are typically created through a process of stepwise refinement, with the most important flows being considered first and then broken into further detail where necessary.

You can use your information ownership model, if you have one, to cross-check against the information flows required to maintain information integrity where ownership is distributed (as discussed earlier).

Information Lifecycle Models

Lifecycle models analyze the way information values change over time.

Entity life histories model the transitions that data items undergo in response to external events, from creation through one or more updates to final deletion. A life history can be a useful cross-check to ensure that there is processing to deal with all of the life events associated with an entity. In particular, it can help you ensure that entities are created in a controlled manner and that all entities have a means of deletion.

EXAMPLE A book is created when it is *published* (as far as the library system is concerned, anyway). The book is then *acquired* by the library and repeatedly *checked out* and *returned* until it is finally *disposed of*.

Each italicized verb in this description is an event in an entity life history for a book.

State transition models (or *statecharts* in UML terminology) model the overall changes in a system element's state in response to external stimuli. This is a useful way to model systems whose interactions with the outside

world cause their internal state to go through many transitions in seemingly unpredictable ways. A statechart models a system element as a finite state machine (FSM). An FSM always has a current state, which is the sum total of the information it holds. When an external event occurs, the FSM changes deterministically to another state and may also instigate some special processing as a result of the change.

EXAMPLE A book is initially *published*; it is then *acquired* by the library, and once on the shelves it alternates between being *available for loan* and *checked out*, until it is *disposed of*.

Each italicized term represents a state of a book.

NOTATION An entity life history is usually represented by using some sort of tree structure, with nodes for each event and branches to represent iteration, selection, and so forth, as shown in Figure 18–6.

A UML state diagram uses railroad tracks to represent the possible state transitions of a book, as shown in Figure 18–7.

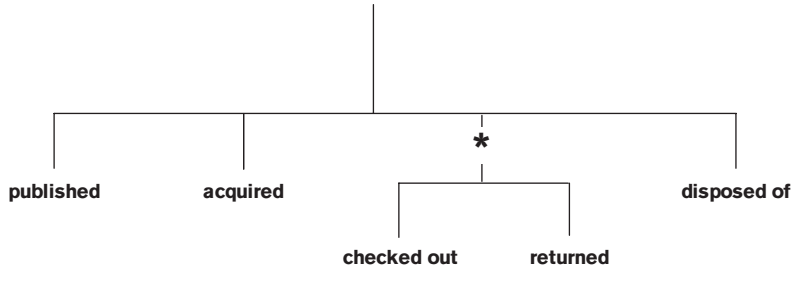

FIGURE 18–6 ENTITY LIFE HISTORY FOR THE LIBRARY EXAMPLE

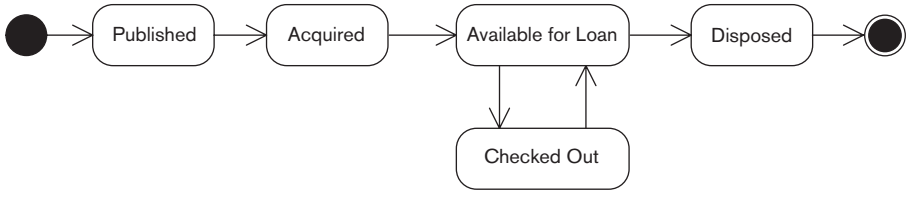

FIGURE 18–7 UML STATE DIAGRAM FOR A BOOK IN THE LIBRARY EXAMPLE

ACTIVITIES Lifecycle models are derived through an understanding of the system's functional requirements, by identifying all of the significant events and understanding the information impact of each.

Other Types of Information Models

INFORMATION OWNERSHIP MODELS Information ownership models define the owner for each data item in the architecture. In this context, "data item" typically means entity (table) or, occasionally, attribute (field), although more complex partitions can be modeled. Of course, in practice, life is never this simple, and you may have to model a number of different classes of information ownership, such as:

- Owner or master, which holds the definitive value for that data item
- Creator, which creates new instances of that data item
- Updater, which modifies existing instances of that data item
- Deleter, which deletes existing instances of that data item
- Reader, which can read but not change instances of that data item
- Copy, which holds a read-only copy of that data item
- Validater, which performs validation on the data item to ensure that it meets business rules
- A combination of these

At its simplest level, information ownership can be modeled by using a grid, with systems and data stores along one axis and data items along the other. Each cell in the grid defines the ownership class of that data item, as shown in Table 18–2.

It may be useful to develop a *trust and permissions model* to define which systems, under which circumstances, are allowed to modify which data items. For example, an external system that provides data updates in a weekly batch might be trusted less than one managed and monitored internally, might require further validation before updates are accepted, or might be constrained to updat-

TABLE 18–2 EXAMPLE OF AN INFORMATION OWNERSHIP GRID

System	Customer	Product	Order	Fulfillment
Catalog	None	Owner	None	None
Purchasing	Reader	Updater	Owner	Creator
Delivery	Copy	Reader	Reader	Updater
Customer	Owner	Reader	Reader	Reader

ing only noncritical data values. As well as being useful here, the definition of information ownership will be an important input to the process of securing the system, as explained with regard to the Security perspective in Chapter 25.

In practice, you may not be able to avoid having more than one creator/updater/deleter for a data item (although it is useful to try to define a single information owner). This particularly occurs when valuable information is held in legacy systems. When two systems can modify the same piece of data, you need to develop *conflict resolution strategies*, such as the following, to ensure that business rules are followed and that information is left in a consistent state.

- Always accept the latest update.
- Maintain multiple copies of the same data item, tagged with their sources.
- Maintain a history of data changes rather than just the latest version of the data.
- Trust one system more than another, so that system's updates take priority.
- Create more complex rules depending on the data changed and the nature of the change.
- Record multiple values and require manual intervention to fix the conflict.
- Reject conflicting updates altogether.
- Use a combination of these strategies.

With multiple updaters a particular problem is detecting that a conflict has occurred. This can be addressed by stamping each record with an incrementing version number and the date and time that the record was last updated.

Although you are unlikely to define detailed rules as part of your AD, it is important to provide sufficient advice and guidance for your designers.

INFORMATION QUALITY ANALYSIS From the architectural perspective, your information quality analysis will focus on defining *sources* of poor-quality information and *principles and strategies* for dealing with this information. Possible strategies include the following.

- *Accept poor-quality information*: This approach is suitable when poor-quality information is not an issue or when the cost of repairing information far outweighs the benefit of improving it.

EXAMPLE An Internet search engine manages a database of many hundreds of millions of URLs. At any one time, a small proportion of these will no longer be valid because pages have been renamed or Web sites removed. However, it is not cost-effective for the search engine to regularly clean up its database to remove these links.

- *Automatically fix poor-quality information*: There are a number of tools available to do this, depending on the type of information.

EXAMPLE You can use tools that will repair or complete addresses or telephone numbers, based on databases of postal codes or telephone dialing rules.

- *Discard poor-quality information*: This may be the best approach when the cost of bad information far outweighs the cost of not having the information at all.

EXAMPLE A company receives bulk mailing lists of variable quality from an external supplier, which it uses to send out marketing material to potential customers. For about 10% of the data, postal codes are missing, invalid, or do not correspond to the mailing address. Such records are discarded because the company is penalized by the postal service if too much of its outgoing mail is incorrectly or incompletely addressed, and material sent to these addresses is unlikely to arrive anyway.

- *Repair poor-quality information manually* (in other words, get users to fix it): This is a very costly approach, however, and you must consider how poor-quality information will be identified and how it will be forwarded to users for correction.

Be aware that there may be legislative requirements for information quality (e.g., some countries charge penalties for maintaining or using incorrect information on members of the public). We consider this point further in our discussion of the Regulation perspective in Chapter 29.

METADATA MODELS*Metadata* is "data about data." Metadata consists of rules that describe and prescribe data items of interest—entities, attributes, relationships, and so forth. Metadata originated in the study of geospatial data and has had an increased profile in recent years following the growth of the World Wide Web and various initiatives around business-to-business communication.

ISO Standard 11197-3 defines metadata as "the information and documentation which makes data sets understandable and sharable for users."[2] Metadata may address a number of aspects of the information it describes, such as:

2. [ISO96], p. vii.

- Data format (syntax)
- Data meaning (semantics)
- Data structure
- Data context (the relationships among data items)
- Data quality

Many organizations are beginning to develop enterprise-wide metadata models; if these are available to you, they can form an extremely valuable input to your Information view. In addition, a number of cross-industry metadata models are being developed under the auspices of groups like the Dublin Core Metadata Initiative.

Metadata models are closely allied to the other types of information models we have described, particularly information structure models that include some elements of metadata (field attributes, relationships, and so on). Most metadata models take the form of structured (or unstructured) text, but some more formal notations are available, in particular those based around XML.

Some automated tools can extract metadata from large databases. Although these are to some extent in their infancy, they can be extremely useful, especially when dealing with legacy systems whose data internals may not be well understood.

There are some industry standard data models that may be of use in your metadata analysis, such as the ARTS Standard Relational Data Model for retail, or the ISO 20022 standard for financial services messaging.

VOLUMETRIC MODELS Volumetric models look at current and predicted data volumes. These can range from a few simple calculations on a scrap of paper to sophisticated statistical models to complete online simulations of systems. At the architectural level, they are usually kept fairly simple because the execution details of the system aren't yet known to any degree of accuracy.

PROBLEMS AND PITFALLS

Representation Incompatibilities

At their simplest, data incompatibilities arise because different systems represent field-level information in different ways, either by using different models for the information (e.g., polar versus Cartesian coordinates) or simply different encoding schemes (e.g., metric or imperial lengths). For example:

- One system may use Y and N for Boolean values, while another uses 1 and 0, or hex FF and 00.

- One system may use standard ISO abbreviations such as FR or DE for countries, while another has its own numeric encoding.
- One system may record monetary amounts in euros, while another uses the local currency in which the transaction took place.
- One system may record amounts by volume, another by weight.
- One system might keep running totals, and another system might just deal in deltas.

These sorts of problems are usually fairly easy to resolve. Much more problematic, however, are incompatibilities between business models.

EXAMPLE An architecture is required to integrate a telephone billing system with another system used to manage prospects, sales, and marketing promotions. A telephone customer may have several phone lines or may charge calls on a single line to different charge codes; for this reason, the billing system is based on the concept of a telephone account. Even worse, some accounts may be held jointly by several customers (especially business accounts), and some others (such as public emergency phone lines) have no real customer at all.

The sales system is concerned solely with customers (and, more important, prospective customers). However, the system needs to know about these customers' existing accounts, as well as other details such as payment history and usage, in order to avoid trying to sell customers something they already have.

The business models for these systems are fundamentally incompatible, and a lot of work is going to be needed to develop an architecture that successfully brings them together.

Incompatible business models can usually be reconciled only by using what may turn out to be fairly complex processing. In the example, you would probably have to develop a subsystem or service that was responsible for maintaining the links between customers and their accounts. This service would have to be updated (possibly in real time) when customers or accounts were created, deleted, or updated, or when the links between them were changed. It would own and manage the information itself and provide that data on demand to any other architectural element that required it.

Such a service would sit at the core of the architecture, being accessed by many other architectural elements, with ambitious targets for performance, scalability, and availability. This service would need to be very carefully designed, built, and tested.

RISK REDUCTION

- Develop a common, high-level model of the data structure, the key data attributes, and their domains, and validate it against all parts of the system (internal and external).
- Review your model with the business to ensure that it reflects reality.
- Focus on a small number of critically important attributes, rather than trying to model everything.
- Don't forget to include external entities in your model (e.g., if you exchange data with other organizations).
- Consider developing a data abstraction layer on top of data sources to hide the incompatibilities from other parts of the architecture.

Unavoidable Multiple Updaters

When creating distributed architectures, we all strive to achieve models whereby each data item is updated in one place and one place only. Unfortunately, in the real world this ambition cannot always be realized, for a number of reasons: Legacy systems cannot easily be changed, information may be sourced from outside the organization, or there may be limitations imposed by geography or politics.

As we have seen, multiple creators or updaters can have a significant impact on the architecture, and resolving such problems is not always easy. From the architectural perspective, you need to be aware of where this can happen so that you can take suitable measures to mitigate the risks.

RISK REDUCTION

- Ensure that your information ownership model is complete and accurate and that all data items with multiple updaters are identified.
- Determine with your stakeholders (primarily your users) which of these multiple updaters are important, and focus on these.
- Understand where inconsistencies through multiple updaters can arise and locate the crunch points where incompatible data items meet.
- Develop strategies for resolving these, such as always overriding old updates with newer ones, or maintaining two copies of data and resolving problems manually.

Key-Matching Deficiencies

When you are bringing together information from multiple systems, key-matching problems almost inevitably arise, as we saw earlier. These may not become apparent until you get into detailed design—by which time it is very expensive to change the architecture—or, even worse, once the system is running.

RISK REDUCTION

- Make sure that you have identified keys for all entities, and satisfy yourself that these keys are compatible across the architecture.
- At all points where information from different systems comes together, ensure that you have the means to map keys from one system to the other.
- Sample real data and run consistency checks on it.
- Whenever possible, go for common keys and standardized ways of modeling information.

Interface Complexity

If two systems need to transfer information between themselves, one bidirectional interface needs to be built. For three systems, three interfaces are needed; for four systems, six. In the worst-case situation, if your architecture comprises n systems, each of which needs to exchange information with every other, you need to build $n(n-1)/2$ interfaces, as shown in Figure 18–8.

Even though it is unlikely that every system in your architecture needs to exchange information with every other, once you have more than a handful of systems, the number of interfaces required becomes unmanageable. Change the interface definition for any one of your n systems, and $n-1$ interfaces need to be redesigned, recoded, tested, and deployed. This represents a significant burden for developers and often acts as a barrier to change.

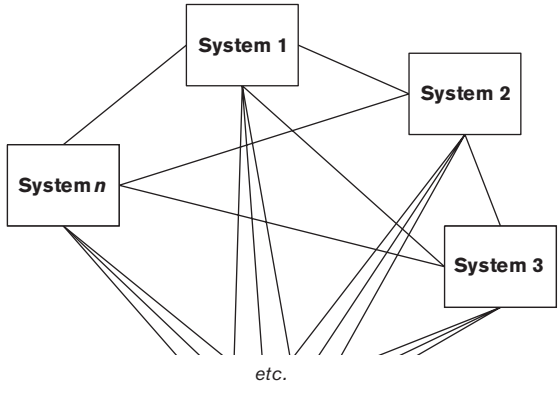

FIGURE 18–8 INTERFACE COMPLEXITY

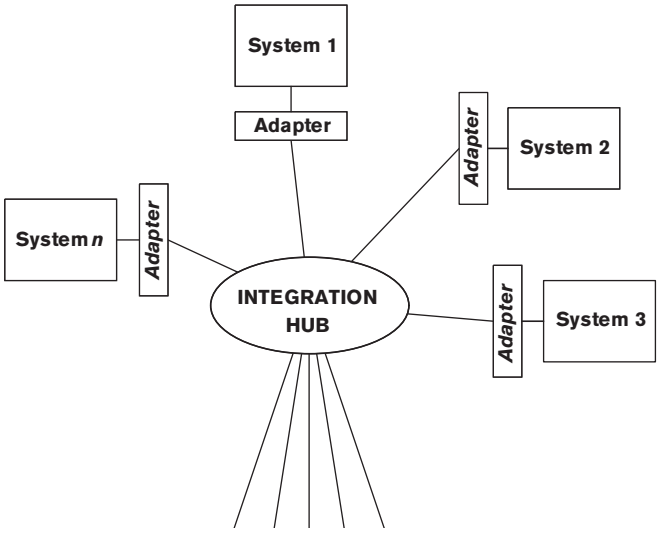

FIGURE 18–9 EXAMPLE OF AN INTEGRATION HUB

RISK REDUCTION

- When interface requirements are complex, consider applying an architectural style called the *integration hub*. In this model, all systems are linked once via a specialized adapter to one central integration hub. The adapter performs system-specific translation, and the hub handles message routing, resilience, and more specialized functions such as publish and subscribe, acknowledgment, and guaranteed delivery. An example is shown in Figure 18–9.

  The advantage of this approach is that if a system changes, often only the adapter for that system needs to be modified. Furthermore, specialized code for routing, resilience, and so forth has to be implemented only once, in the central hub. (Of course, central hubs also have disadvantages in that they are often a single point of failure, can be a scalability bottleneck, and ironically can slow down change due to the difficulty of scheduling and prioritizing changes to such a critical shared component.) A third-party product is typically used to implement an integration hub. A large number of such off-the-shelf, highly configurable integration hubs are available as both commercial and open source products.

  Integration hubs and similar architectures (such as a message bus) form part of the wider topic of Enterprise Application Integration, a full consideration of which is outside the scope of this book. We provide some references on this subject in the Further Reading section at the end of this chapter.

Overloaded Central Database

Many of the problems described in this chapter can be eliminated by storing all information in a single central database. This approach is much simpler and cleaner, since there is no need for key mappings, update reconciliation, or complex interfaces, and all data is immediately available.

However, a single central database is a single point of failure and will eventually become a performance bottleneck. For geographically distributed systems, a central database will give poor latency for remote users, and they may find that system availability is constrained due to limitations of the global network. Managing all data in a single central database can cause the data model to become overloaded or unworkable and can cause design-time and runtime contention. For these reasons, care must be taken when designing a system based on a single central database.

RISK REDUCTION

- Carefully consider the likely growth of your system in terms of data volumes, numbers of users, and their locations. (We discuss this issue in Chapter 28 on the Evolution perspective.)
- Consider the deployment (now or in the future) of a reporting database, separate from the main operational data store, and design your architecture with this possibility in mind.
- Be aware of the need to partition data in the future and design a strategy for it now (even if it is not yet implemented).
- If you do opt for a single central database, make sure that there are some scalability options available in case the system is more successful than expected.
- Look into the use of database clustering technologies and other mechanisms for improving availability and performance.

Inconsistent Distributed Databases

Conversely, some of the problems described in this chapter can be eliminated by replicating information between multiple databases in different locations or even geographical regions. This approach brings data near to where it is needed, with a consequent reduction in latency and improvement in availability.

However, distributed information architectures are harder to design and build and often lead to information inconsistency due to the replication delay. Furthermore, updates are harder to manage in cases where replicate copies are not read-only. While these problems are not insurmountable, they require careful design and a solid implementation.

RISK REDUCTION

- Carefully consider the need for a distributed information architecture, balancing the benefits this brings against the cost in complexity and data inconsistency.
- If you adopt a distributed model, ensure that you have effective strategies in place for dealing with inconsistency and that these are agreed upon with your key stakeholders, especially users.
- Ensure that there are effective operational tools and processes in place for detecting and dealing with problems that can't be dealt with automatically.

Poor Information Quality

If the actual data is inconsistent, inaccurate, or incomplete, it doesn't matter how good your information model is—you will face big problems when your system goes into operation.

In fact, the real problem is not necessarily poor information quality but *unexpectedly* poor information quality. If you know that some information will be inadequate, you can develop strategies early to deal with it and successfully manage the expectations of your stakeholders in this area.

RISK REDUCTION

- Validate your key assumptions about information quality early (e.g., "All products can be uniquely identified globally by using an immutable common key").
- Make sure that you understand what information is important and what is less important (your stakeholders, primarily users, can tell you this), then focus on the important information.
- Make use of commercially available information quality tools to analyze the quality of existing information.
- Identify the places where poor-quality information can appear, and develop strategies for dealing with it, such as rejecting poor-quality information, marking it as suspect, or attempting to fix it.

Excessive Information Latency

Excessive latency typically arises from overly complex architectures or architectures that are not designed to handle the volumes of information they are presented with. You may also have latency issues that are outside your control. For example, information may arrive from an external source only once a week, or updates may need to be applied in batches overnight because of the limitations of a legacy system.

As with information quality, poor latency becomes an issue only if it is *unexpectedly* poor. By identifying expected latency early, you can identify problem areas and develop strategies to deal with them.

RISK REDUCTION

- When there is distance or complexity between information providers and information consumers, ensure that you predict, as best you can, what the information latency will be.
- When latency is significant, review this with your stakeholders to determine whether it is a concern.
- Better still, obtain agreement on realistic latency requirements for all data items up front, and validate your model against these.

Inadequate Volumetrics

A system designed to handle a thousand updates per day is unlikely to cope well when faced with a million updates per day. Unless you are clear about the volumes of information the system is expected to handle, you have little chance of designing an appropriate architecture. (We address the issue of volumetrics in more detail in Chapter 26.)

RISK REDUCTION

- Make sure that data volumes are captured, reviewed, and approved by your stakeholders. You may want to separately capture "business" volumes (such as numbers of orders) from acquirers and users, and "technical" volumes (such as numbers of database updates) from technical stakeholders.
- Make sure that volumes are realistic. If the stakeholders convey doubt or vagueness about this, pursue the issue, and if in doubt, increase them to allow for the margin of error.
- Make sure that your data volumes cover all scenarios—not just the online day, for example, but also the overnight processing and peak periods such as the end of the year or holiday processing.
- Make sure that there is an effective translation of business volumes into physical ones. For example, a single business transaction, such as placing an order, may result in several physical transactions, such as decrementing stock levels, posting account records, assigning compensation to sales staff, and arranging delivery of the ordered item.
- Make sure that your volumes take future expansion into account.
- Prototype your data stores and the access to them for the expected volumes you do have.

CHECKLIST

- Do you have an appropriate level of detail in your data models (e.g., no more than about 20–30 entities)?
- Does the data model support the processing requirements now and those likely in the future?
- Are keys clearly identified for all important entities?
- When an entity is distributed across multiple systems or locations with different keys, are the mappings between these keys defined? Do you have processes for maintaining these mappings when data items are created?
- Have you taken account of data in one place that is derived from data managed and owned elsewhere, such as account balances derived from account activity?
- Have you defined strategies for resolving data ownership conflicts, particularly when there are multiple creators or updaters?
- Are latency requirements clearly identified, and are mechanisms in place to ensure that these are achieved?
- Do you have clear strategies for transactional consistency across distributed data stores, and do these balance this need with the cost in terms of performance and complexity?
- Have you considered which data storage models to use for the various data stores in your system, taking into account the strengths and weaknesses of each?
- Do you have mechanisms in place for validating migrated data and dealing appropriately with errors?
- Do you have the right sort of data stores (operational data store, reporting databases, data warehouses, and data marts) for the expected volumes and performance requirements?
- Have you defined sufficient storage and processing capacity for archiving? For restoring archived data?
- Has a data quality assessment been done? Have you created strategies for dealing with poor-quality data?
- Have you confirmed which entities in your information model should be obtained from shared enterprise sources, and if so, does your architecture make use of these appropriately?

FURTHER READING

The literature on information architecture per se (as opposed to data design techniques or specific data management technologies) is sparse.

Fortunately, data modeling, and particularly relational modeling, which underpins much that we do, has a strong theoretical grounding, so there is a plethora of books on the subject. The classic of the genre, which is still being updated, is probably Date [DATE03]. Other good general books include Elmasri and Navathe [ELMA99] and Kroenke [KROE02].

Kim [KIMW99] looks at some of the newer techniques such as object-oriented databases. Redman [REDM97] provides a detailed discussion of the issues around data quality and how to develop strategies for data quality analysis and improvement.

Enterprise Application Integration architectures are covered in a large number of books, such as Linthicum [LINT03] and Ruh et al. [RUHW00].

You can find further information on metadata modeling in ISO Standard 11197-3 [ISO96] and books such as [MARC00]. Information on specific metadata models such as the ARTS Standard Relational Data Model or the ISO 20022 standard for financial services messaging can be found on the Web sites for those organizations.

If you are interested in ideas on how to flexibly evolve a database schema as part of the software development process, Scott Ambler and Pramod Sadalage's book on database refactoring [AMBL06], which introduces the Evolutionary Database Design technique, will be of interest.

There are many books on data warehousing, from the two pioneers of this approach, William Inmon (e.g., [INMO05]) and Ralph Kimball (e.g., [KIMB02]), and many others.

Alec Sharp and Patrick McDermott's book [SHAR08] provides a good description of the subject and the techniques used. A vast number of books (too numerous to mention here) cover specific relational database products (e.g., Oracle, SQL Server, DB2, Sybase, MySQL) and tools and technologies for application development, systems management, and integration.

The best place to obtain information on nonrelational database technologies, such as the NoSQL movement, is the Internet.

The Data Management Association (DAMA) has much useful information, runs conferences and seminars, provides training and certification, and has chapters worldwide. They can be found at www.dama.org.

This page intentionally left blank

19

The Concurrency Viewpoint

Definition	Describes the concurrency structure of the system and maps functional elements to concurrency units to clearly identify the parts of the system that can execute concurrently and how this is coordinated and controlled
Concerns	Task structure, mapping of functional elements to tasks, interprocess communication, state management, synchronization and integrity, supporting scalability, startup and shutdown, task failure, and reentrancy
Models	System-level concurrency models and state models
Problems and Pitfalls	Modeling the wrong concurrency, modeling the concurrency wrongly, excessive complexity, resource contention, deadlock, and race conditions
Stakeholders	Communicators, developers, testers, and some administrators
Applicability	All information systems with a number of concurrent threads of execution

Historically, information systems were designed to operate with little or no concurrency, running via batch mode on large central computers. However, a number of factors (including distributed systems, increasing workloads, and cheap multiprocessor hardware) have combined so that today's information systems often have little or no batch processing and are inherently concurrent.

In contrast, control systems have always been inherently concurrent and event-driven, given their need to react to external events in order to perform

control operations. It is natural, then, that as information systems become more concurrent and event-driven, they start to take on a number of characteristics traditionally associated with control systems. In order to deal with this concurrency, the information systems community has naturally adopted and adapted proven techniques from the control systems community. Many of these techniques form the basis of the Concurrency viewpoint.

The Concurrency view is used to describe the system's concurrency and state-related structure and constraints. This involves defining the parts of the system that execute at the same time and how this is controlled (e.g., defining how the system's functional elements are packaged into operating system processes and how the processes coordinate their execution). To do this, you need to create a process model and a state model: The process model shows the planned process, thread, and interprocess communication structure; the state model describes the set of states that runtime elements can be in and the valid transitions between those states.

Once you have created process and state models, you can use a number of analysis techniques to ensure that the planned concurrency scheme is sound. The use of such techniques is typically part of creating a Concurrency view, too.

It's worth noting that not all information-based systems really benefit from a Concurrency view. Some information systems have little concurrency. Others, while exhibiting concurrent behavior, use the facilities of underlying frameworks and containers (e.g., application servers and databases) to hide the concurrency model in use.

EXAMPLE Data warehouse systems tend to be batch-loaded overnight and accessed from a number of desktop machines. These systems do exhibit concurrent behavior—multiple clients can request information from the data warehouse concurrently. However, such a system will typically rely on the underlying database management system to handle all of the concurrency for it (in any way it chooses). Therefore, the process model used is of little architectural significance, and you have little or no control over it. The interesting aspects of concurrency relate much more to the design of the physical data model and should be handled there.

In contrast, however, many of today's information systems are inherently event-driven, reactive, concurrent systems. This is particularly the case when considering infrastructure such as middleware products. Systems of this type typically sit idle until an external event occurs and then process the event. Given that many external events can occur simultaneously and

that the interarrival time of such events may be lower than the time taken to process them, this kind of information-based system is inherently concurrent, with many operations being executed at once.

EXAMPLE Consider an e-commerce system that uses a message-based approach to processing transaction requests. In such a system, when a request arrives, it is translated into a message that is queued for the appropriate functional element that can process it. In order to prevent message queues from growing too long and to make efficient use of processing resources, the processing element will need to process a number of messages concurrently. In this case, there may be a large number of concurrent operations within the functional element, each one needing access to shared resources.

The Concurrency viewpoint is extremely relevant to systems that exhibit this kind of behavior. Creating a Concurrency view allows the concurrency design of such systems to be made explicit and helps interested stakeholders understand concurrency constraints and requirements. It also allows you to analyze the system to avoid common concurrency problems such as deadlocks or bottlenecks.

CONCERNS

Task Structure

The most important aspect of creating a Concurrency view is establishing the system's process structure, which identifies the overall strategy for using concurrency in the system. It defines the set of processes across which the system's workload is partitioned and how the functions of the system are distributed across them. It is also usually necessary to consider the use of operating system threads within processes or to abstract away from individual processes and consider groups of similar processes instead.

Note: Throughout this chapter, we use the word *task* as a generic term to describe a processing thread—whether it is a single operating system process, one thread within a multithreaded process, or some other software execution unit. When the difference is significant, we specifically use the terms *process* or *thread* as appropriate.

The aspects of the system's task structure that this view needs to address depend very much on the kind of system you are dealing with.

EXAMPLE A complex, small-footprint system may have only one or two operating system tasks but may need to use a very complex thread model to meet its efficiency and responsiveness goals. In this case, the focus of the task structure activity needs to be at the thread level.

A large enterprise system may be composed of literally hundreds of concurrent processes, many containing dozens of threads. In this sort of system, the task structure activity needs to be at the level of groups of similar processes in order to focus on the architecturally significant aspects of the concurrency.

Mapping of Functional Elements to Tasks

The mapping of functional elements to tasks can have a significant effect on the performance, efficiency, resilience, reliability, and flexibility of your architecture, so this needs careful consideration. The key question to address is which functional elements need to be isolated from each other (and so placed in separate processes) and which need to cooperate closely (and so need to run within the same process).

Interprocess Communication

When functional elements reside within a single operating system process, communication among them is relatively simple because of their shared address space. While some coordination may be required (see the Synchronization and Integrity concern), you can use any number of data structures to pass information among them. Similarly, a number of easily used control mechanisms (such as the procedure call and variants of it) can transfer control among elements as needed.

In contrast, when elements reside in different operating system processes, communication among them becomes more complex. This complexity increases if the processes also reside on different physical machines.

A number of interprocess communication mechanisms can be used to link elements in different processes, including remote procedure calls, messaging, shared memory, pipes, queues, and so on. Each has its own strengths, weaknesses, and constraints, and inappropriate use of these mechanisms can cause problems at the system level (e.g., message queue latency between processes causing scalability or throughput problems). In order to deliver a system with an acceptable set of quality

properties, the Concurrency view needs to consider and identify the set of interprocess communication mechanisms that will be used to provide the interelement communication required by the system's functional structure.

State Management

In many systems, the runtime state of system elements is important to the correct operation of the system. This is particularly the case for event-driven systems that exhibit a high degree of concurrency, where business operations tend to be processed via state machine implementations.

In such systems, a concern of the Concurrency view is to clearly define the set of states that each functional element of the system can be in at runtime, the set of valid transitions between those states, and the causes and effects of the interstate transitions. Such careful state management is a major factor in ensuring reliability and correct behavior for most concurrent systems. Again, if you are using a formal architectural style, it may define how the system's runtime state should be handled.

Note that this concern refers to the state of the runtime elements of the system (which could be termed the *technical state* of the system). Another type of state management important to many information systems is the set of valid states and transitions for their core persistent information (business objects—the *business state* of the system). However, this is a distinct concept of state, and we refer to persistent object state models as *lifecycles* to avoid any confusion between the two. Object lifecycles are discussed in Chapter 18 as part of the Information viewpoint.

Having said this, it is also quite reasonable to consider state management in the Functional view—after all, the state of functional elements is what we're considering. However, our experience is that the design of the system's state management usually fits better in the Concurrency view. Those systems where state is important are usually those where concurrency is important too, and considering system-level state usually involves the consideration of the concurrency around it as well.

Synchronization and Integrity

As soon as more than one thread of control exists in the system, it is important to ensure that concurrent execution cannot result in corruption of information within the system. This concern applies at a number of levels in the system, from a shared variable within a multithreaded module at one end of the scale to critical corporate transaction data in shared data stores at the other.

An important concern for the Concurrency view to address is how concurrent activity will be coordinated so that the system operates correctly and maintains the integrity of the data within it.

Supporting Scalability

In any highly concurrent system, the approach taken to concurrency, synchronization, and state management can have a profound effect on the scalability that the system can achieve. Too much or too little concurrency can slow a system down and prevent it from handling heavy workloads efficiently, while excessive or simply naïve synchronization can result in a system that performs very well for light workloads but grinds to a halt when heavy workloads are applied. The problem is that designing high-performance, highly concurrent systems is difficult and can be an error-prone process. An important concern for this view to address is how the concurrency approach used will support the performance and scalability required while being simple enough to implement cost-effectively and reliably. We discuss an approach to achieving this in Chapter 26 on the Performance and Scalability perspective.

Startup and Shutdown

When you have more than one operating system process in your system, startup and shutdown of the system can become more complicated to manage. Intertask dependencies may mean that tasks need to be started and stopped in very specific orders so that if some tasks fail to start, others will not be started. The system startup and shutdown dependencies are an important part of your concurrency design and need to be clearly understood by developers, testers, and administrators.

Task Failure

When functional elements reside in different processes or run on different threads, dealing with element failure can be complex. This is because an element in one task cannot rely on another task being available when it needs to communicate with it, whereas when an element calls another one in the same task, it knows it will be there. Your concurrency design needs to take into account this added possibility of failure and ensure that the failure of one task doesn't bring the entire system to a halt. In order to address this concern, you need a system-wide strategy for recognizing and recovering from task failure.

Reentrancy

Reentrancy refers to the ability of a software element to operate correctly when used concurrently by more than one processing thread. This is primarily a concern for software developers when designing their software elements. From an architectural perspective, reentrancy is an important constraint for

certain elements, so the architecture must clearly define which modules need to be reentrant and which do not.

EXAMPLE If you are developing an e-mail server, the ability to support a great deal of concurrency is likely to be a key concern. Without this, it will be hard to use the e-mail server for large user populations who will want to send and receive e-mail simultaneously. You can take a number of approaches to achieve such concurrency, but for the sake of argument, let's assume that you have decided to implement the server by using a single operating system process and many (perhaps hundreds of) concurrent operating system threads running within it: some sending e-mail, some receiving e-mail, and some managing the server's internal state.

In this sort of environment, it is crucial to decide which of the elements of your system have to be reentrant and which do not. Any element involved in sending and receiving e-mail (e.g., a name resolution library that translates e-mail domains to network addresses) will need to be reentrant to ensure that it can be used simultaneously by many sending and receiving threads. Without such a guarantee, the name resolution library could be the source of many subtle problems later if its internal state could be corrupted by concurrent access.

The reentrancy needs of your architecture can also affect which third-party software elements you can use within the system and where you can use them.

Stakeholder Concerns

Typical stakeholder concerns for the Concurrency viewpoint include those shown in Table 19–1.

TABLE 19–1 STAKEHOLDER CONCERNS FOR THE CONCURRENCY VIEWPOINT

Stakeholder Class	Concerns
Administrators	Task structure, startup and shutdown, and task failure
Communicators	Task structure, startup and shutdown, and task failure
Developers	All concerns
Testers	Task structure, mapping of functional elements to tasks, startup and shutdown, task failure, and reentrancy

MODELS

System-Level Concurrency Models

The Concurrency view maps the functional elements onto runtime execution entities via a concurrency model. The concurrency model typically contains the following items.

- *Processes*: In this context, the term *process* refers to an operating system process, that is, an address space that provides an execution environment for one or more independent threads of execution. The process is the basic unit of concurrency in the design of the system. At the architecture level, the processes are normally assumed to be isolated from each other so that if one process wants to affect the execution of another, it must use an interprocess communication mechanism.
- *Process groups*: At the architecture level, it can often be useful to group individual processes so that a collection of closely related processes can be considered as a single entity at the system level. This can provide a useful abstraction that allows less important concurrency concerns to be deferred until subsystem design. An example is a database management system (DBMS). The important point from the system level is that the DBMS is a functional unit, accessed via well-defined interfaces, that runs in its own process or group of processes. However, the details of the exact number of processes it uses (e.g., how many logging processes run within the DBMS) and the function of each are almost certainly irrelevant to the architecture—indeed, this will probably be decided by a technical specialist later in the design process. Using a process group in this situation makes it clear that a group of related processes will be used but defers the details of the set until later. The other common use for process groups is simply as a hierarchical structuring technique for large or complex systems that contain many processes. All of the processes may need description, but the use of process groups can make the process model easier to comprehend.
- *Threads*: In this context, the term *thread* refers to an operating system thread, that is, a thread of execution that can be independently scheduled within an operating system process. Threads are known as *lightweight processes* by some operating systems. At the level of system architecture, threads can often be ignored, with the details of their use being the responsibility of subsystem designers (perhaps with you guiding their use via design patterns in the Development view). However, for some systems you do want to model the use of threads in at least some parts of the system. Threads are normally represented in process models via a decomposition of a process.

- *Interprocess communication*: When processes are running, they are assumed to be isolated from each other so that one process cannot change anything in another process. However, in most concurrent systems, processes do need to interact in order to coordinate their execution, request services from each other, and pass information among themselves. They achieve these interactions via a number of interprocess communication mechanisms ("IPC mechanisms"), which are the connectors in the system's runtime architecture.

  The mechanisms available vary depending on the underlying technology platforms in use. However, interprocess communication mechanisms generally fall into one of these groups.

  - *Procedure call mechanisms* are all some variation on an interprocess function call and are usually based on some form of remote procedure call or some sort of message-passing operation.
  - *Execution coordination mechanisms* allow two or more processes (or threads) to signal to each other when certain events occur. Coordination mechanisms include semaphores and mutexes and are typically limited to coordination between processes or threads running on the same physical machine.
  - *Data-sharing mechanisms* allow a number of processes to share one or more data structures and access them concurrently (possibly coordinating this access via coordination mechanisms). Data-sharing mechanisms include shared memory, distributed tuple spaces (such as Linda and more recent implementations such as GigaSpaces), and simple, traditional mechanisms such as client/server databases and shared file storage.
  - *Messaging mechanisms* are related to data-sharing mechanisms, but rather than placing data structures in a shared space for concurrent access, they transmit data structures from one task to another. Messaging systems normally implement one or both of two well-defined messaging models: queuing and publish/subscribe. Queuing introduces a "first in, first out" queue structure between producers and consumers where consumers destructively read messages from the queue (i.e., a message is delivered to only one consumer). Publish/subscribe introduces a "topic" or "bus" between producers and consumers where the consumers indicate the types of messages that they are interested in and a message is consumed by all consumers interested in it.

As the architect, you need to choose the interprocess communication mechanisms carefully because of the impact that they can have on the quality properties that the system exhibits (such as its performance, scalability, and reliability). The IPC mechanisms also impose significant constraints on the functional elements that use them, and so it is important to choose them early so that these constraints can be taken into account.

NOTATION You can represent the Concurrency view in a number of ways. Some of the more common notational approaches include UML and other formal notations, along with less formal notations, which we describe briefly here.

- *UML*: UML's concurrency modeling facilities are rather simple but do include the notion of an active object (i.e., an object with a thread of control). There are a number of ways that concurrency structures can be represented in UML, including stereotyped packages, components, and classes, but unfortunately no approach has become a standard. We have used a number of conventions in our models and have found that stereotyped active components are a good place to start when modeling processes and threads. We sometimes also add a process group stereotype to represent a group of related processes (such as a database engine) if this is a useful abstraction for the system we are working on.

  Simple examples of interprocess communication, like remote procedure calls, can be represented by using standard UML intercomponent associations, with arrowheads indicating the direction of communication (and possibly using tagged values on the association to make the communication mechanism clear). More complex forms of interprocess communication (shared memory, semaphores, and so on) can be represented quite effectively by introducing further stereotypes and showing associations between the components in the tasks and the interprocess communication mechanisms they use.

  Figure 19–1 shows an example of UML being used for a concurrency model.

  This model shows how the system is implemented by using three processes (a client, a statistics service, and a statistics calculator) along with a process group to implement the Oracle DBMS instance. The concurrent activity between the Statistics Accessor and Statistics Calculator components needs to be coordinated because they are in different processes; a mutex is used to achieve this. The illustrated scenario is very simple, and there is little or no architecturally significant thread design in this model. Figure 19–2 shows a more involved model with more architecturally significant threading.

  The concurrency model shown in Figure 19–2 illustrates a case where the process structure is very simple, namely, two processes that communicate via a socket stream. However, the thread structure in the DBMS Process instance is architecturally significant, and its structure and interthread coordination strategy need to be documented and explained. The model shows that there is a single thread containing the Network Listener component, which communicates with between 1 and 40 threads that contain the four main query-processing components via an interprocess communication queue. The Disk I/O Manager component

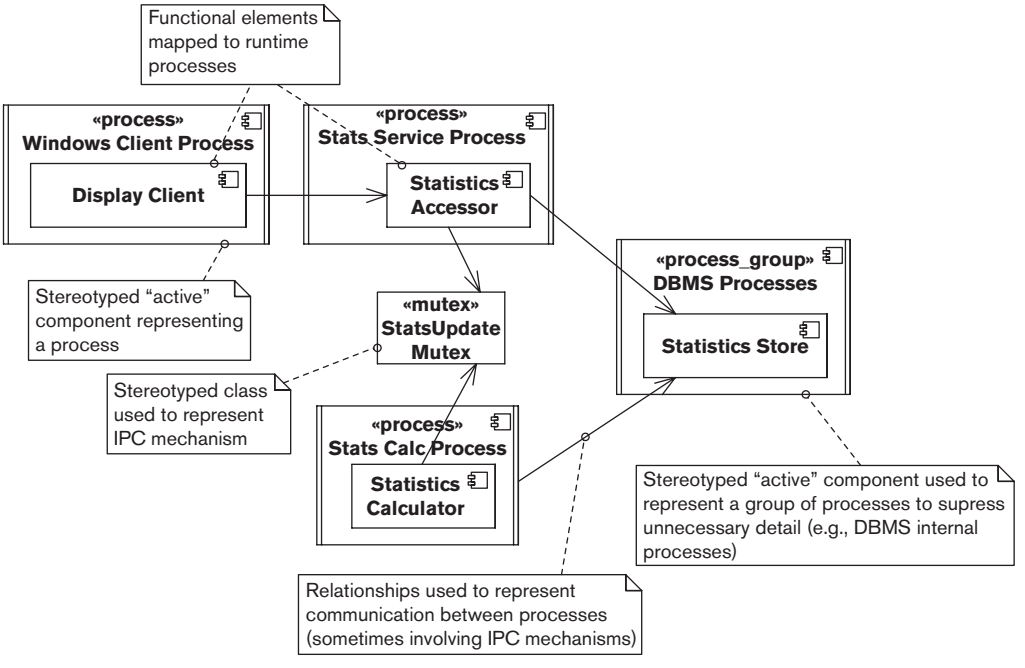

FIGURE 19–1 CONCURRENCY MODEL DOCUMENTED BY USING UML

is hosted on its own thread, and there may be up to 10 instances of this running. The Data Access Engine component communicates with the Disk I/O Manager instances via the shared memory mechanism.

- *Formal notations*: The real-time and control systems research community has created a number of concurrency modeling languages that allow the creation and analysis of process models. A number of these languages, such as LOTOS, Communicating Sequential Processes (CSP), and the Calculus of Communicating Systems (CCS), are formal and represented textually. Most of these languages are mathematical and fairly abstract, and they aren't widely used in information systems development. While this doesn't mean they can't be useful, we have yet to come across a large-scale industrial application of them to information systems. The problem with using these languages in practice is often the need to teach them to the interested stakeholders, and there is always a need to ensure that the representation and analysis that they allow will be useful for the specific situation to which you are trying to apply them.
- *Informal notations*: In our experience, by far the most common notation used to represent process models is an informal one, created by the author of the model. Given the relatively small number of object types in a process model, an informal notation invented for the problem at hand

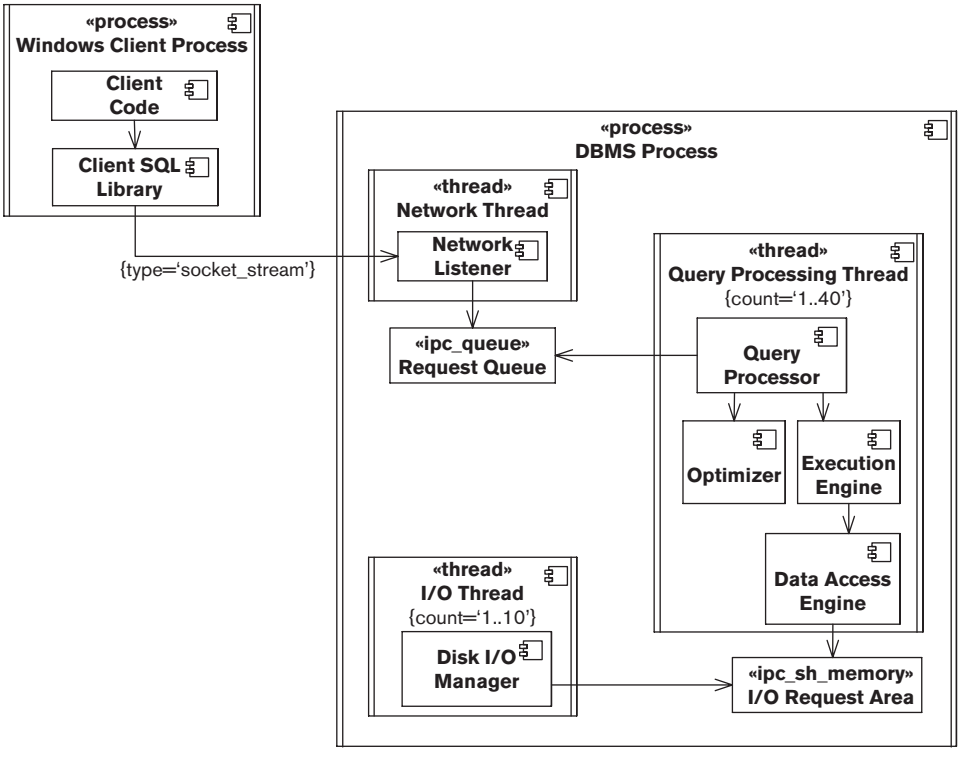

FIGURE 19–2 THREAD-BASED CONCURRENCY MODEL

often works very effectively as a communication tool as long as it is explained clearly. The notation needs to capture processes, process groups, threads, and the interprocess communication mechanisms in use. As long as the notation to represent each of these concepts is well defined, an informal notation often has much to recommend it. In particular, the notation can be kept simple and avoids the potentially awkward process of bending a general-purpose notation like UML to represent the model being described. The risk with informal notations is that they are never clearly defined and so lead to ambiguous descriptions.

ACTIVITIES

Map the Elements to the Tasks. The first step when creating your process model is to work out how many processes you need and to decide which functional elements will run in which processes. In some cases, this is a straightforward job—each functional element ends up being a process (or perhaps a process group), or all of the elements end up in a single process. In other

cases, there is a complex N:M mapping between functional elements and processes, with some elements partitioned between processes and other elements running in shared processes. The important point about this mapping is that you should introduce concurrency only where it is actually required. Concurrency adds complexity to the system and adds significant overhead to inter-element communication when it must cross process boundaries. Therefore, add more processes to your system only if you need them for distribution, scalability, isolation, or other reasons guided by the requirements for your system.

Determine the Threading Design. The term *threading design* refers to the process of deciding on the number of threads to include in each system process and how those threads are to be allocated and used. In most cases, threading design is not something that the architect needs to get directly involved in—this is usually the job of the subsystem designers. However, you may get involved in designing and specifying general threading approaches or patterns that should be used at various points in the system in order to meet the system's required quality properties or to ensure consistency across the implementation.

Define Mechanisms for Resource Sharing. As soon as concurrency is introduced into the system, you must carefully consider how to share resources between concurrent threads of execution. Resource sharing is considered in some other parts of the architecture, too (notably, the Information view), and the two activities might be best tackled as a single task. This isn't a book on concurrent computing, so we don't have space to discuss all of the options and potential pitfalls to consider when sharing resources. The simplistic advice is simply that whenever a resource (such as a piece of data in memory, a file, a database object, or a piece of shared memory) is shared among two or more concurrent threads of execution, it must be protected from corruption. This is usually achieved with some form of locking protocol. As with threading design, the details of resource sharing are rarely architecturally significant. Your role in relation to this is to ensure that suitable resource-sharing approaches are used where necessary and that the approach used is suitable in the overall context of the system and does not produce unacceptable side effects for the system as a whole.

Define the IPC Mechanisms to Use. In most concurrent systems the tasks need to communicate frequently, and so, along with deciding how to share resources between tasks, you will need to consider what communication is needed between them and which interprocess communication mechanisms you will use to enable it. Again, we don't have space here to discuss all of the options and the tradeoffs that they imply, but as is usually the case, a simple and regular scheme that minimizes the amount of intertask communication is likely to be the best choice. Better still is using a library or framework (such

as an implementation of the Actor pattern) to avoid having to deal with a lot of this complexity yourself. While this sounds like simplistic advice, implementing complex intertask communication correctly is a very difficult job best left to specialists. As with other architectural concerns, your focus needs to be on defining a common system-wide approach and on reducing the risk involved in its implementation.

Assign Priorities to Threads and Processes. Some tasks in your system may be more important than others. If you have tasks of different importance running on one machine, you need to control their execution so that the more important work gets done before the less important work. The normal method for achieving this is to use the facilities of the underlying operating system to assign priority levels to the different threads and processes. All modern operating systems provide this feature in roughly the same way. Tasks are explicitly or implicitly given runtime priorities. When the operating system's thread scheduler is choosing tasks to run, it considers the higher-priority tasks before the lower-priority ones, thus getting the important work done first. If you can avoid assigning explicit priorities to threads, in general do so—processing priorities can add a lot of complexity to your process model and can introduce subtle but serious problems such as priority inversion. However, sometimes you can't avoid it. In these cases, keep the assignment of priorities as simple and as regular as possible, and analyze and prototype your approach to make sure that you aren't introducing problems worse than the one you're trying to solve.

Analyze Deadlocks. Having introduced concurrency into the system, you have also introduced the risk of the entire system grinding to a halt in unexpected ways. Whenever you have concurrency in the presence of shared resources, you always have the possibility of deadlock. You can use a number of modeling and analysis techniques to try to spot potential deadlocks. An example of such a technique is Petri Net Analysis, which allows you to create a model of your processing threads and shared resources and then analyze the model to catch potential deadlock situations. With experience, it is also usually possible to perform effective deadlock analysis through careful, informal consideration of your concurrency model.

Analyze Contention. Whenever you have a number of tasks and shared resources, you almost always find contention. Contention occurs between tasks when more than one task requires a shared resource concurrently. The introduction of coordination mechanisms (such as mutexes) inevitably introduces contention when workloads are high. If contention rises beyond a certain point, the system will slow dramatically, and little useful work will get done. In order to avoid this during normal operation, you need to analyze your shared resources from this point of view. The basis of the technique is to identify each of your possible contention points. Then, for each, estimate the likely number of concurrent tasks contending for the resource and for how

long each will need the resource. This allows you to establish the likely wait times that each task experiences at each point and then to estimate how such contention will affect your processing times and throughput. Repeating the exercise for different workloads allows you to estimate the maximum theoretical workloads your system can possibly support. Once you understand the potential for the system becoming overloaded, you can design mechanisms into your software to handle such conditions gracefully (such as implementing the Circuit Breaker pattern).

State Models

A state model is used to describe the set of states that a system's runtime elements can be in and the valid transitions between the states. The set of states and transitions for one runtime element is known as a *state machine*, and the collection of all of the interesting state machines for your system forms the overall state model.

Usually, you will find that each system task identified in the concurrency model will have one or at most a few functional units mapped to it that are effectively in control of the task. These functional units normally have the system's interesting state models associated with them. If you create a state model, be sure to focus on these system elements so that the state model describes only architecturally significant information. You don't need to capture all of the state machines inside all of the system's elements; the AD needs to describe only state that is visible at the system level, not state that is hidden inside the system's elements.

An important decision to make before you start creating the state model is the set of semantics you want to use in your state machines. Modern state modeling notations (in particular, UML's statechart, discussed later) allow you to introduce a mind-boggling degree of complexity. You need to use such notations carefully if you want to produce a comprehensible model.

A basic state machine in the state model would normally contain the following types of entities.

- *State*: A state is an identifiable, named, stable condition during a runtime functional element's lifetime. States are normally associated with waiting for something (an event) to occur or performing some sort of operation.
- *Transition*: A state transition defines an allowable change in state, from one state to another, following the occurrence of an event. From a modeling point of view, transitions are normally considered to occur in zero time and so cannot be interrupted.
- *Event*: An event is an indication that something of interest has happened in the system (and is normally recognized by an operation being invoked on an element or a time period ending). Events are the triggers that cause transitions between states to occur.

- *Actions*: Actions are atomic (noninterruptible) pieces of processing that can be associated with a transition (so an event causes the transition to occur, and then an action is executed as part of the state transition).

More sophisticated state modeling notations allow additional modeling elements such as *guards* (Boolean conditions governing state transitions), *activities* (long, interruptible items of processing that can be associated with states), and *hierarchical states*.

NOTATION State models are typically represented by a graphical notation derived in some way from the classic state transition diagram. The most popular variant in use today is probably the UML notation for representing state, the *statechart*. At the end of this subsection we briefly discuss other graphical notations as well as some nongraphical ones, but first we focus on UML's statechart.

- *UML*: A statechart is a flexible notation that can be used in a number of different ways at differing levels of sophistication. Deciding which parts of the notation to use is an important step before getting too far into the modeling process. Figure 19–3 shows a UML statechart that represents the state model for a calculation engine.

  This statechart shows much of the important notation for a UML statechart, with a composite primary state, concurrent state management, and the use of start and end pseudo states to indicate how the element's lifecycle begins and ends. The single top-level state (Running) is entered when the element is started and exited when a shutdown event is received (the `reset()` action is performed as part of that transition).

  The Running state has been decomposed into four substates that comprise the business of running this element: Waiting for Data, Calibrating Metrics, Calculating, and Distributing Results. The transition arrows indicate the possible transitions between states (along with the events that cause the transitions and the actions that will be executed).

  The Calculating state is interesting because it is a concurrent state, as you can see from the dashed line that bisects it. This means that while in the Calculating state, the element is actually in two concurrent substates (Calculating Values and Calculating Risk). When the activity associated with these states completes, the transition from the states is taken, and when both are complete, the element can leave the Calculating state.

  There are two architecturally significant aspects to this state machine.

  - If new input data becomes ready when the element is in the Calculation state, in-progress results are discarded (by executing the `reset()` action), and calculation starts again. In contrast, if this occurs while results are being distributed, the distribution process is not interrupted.

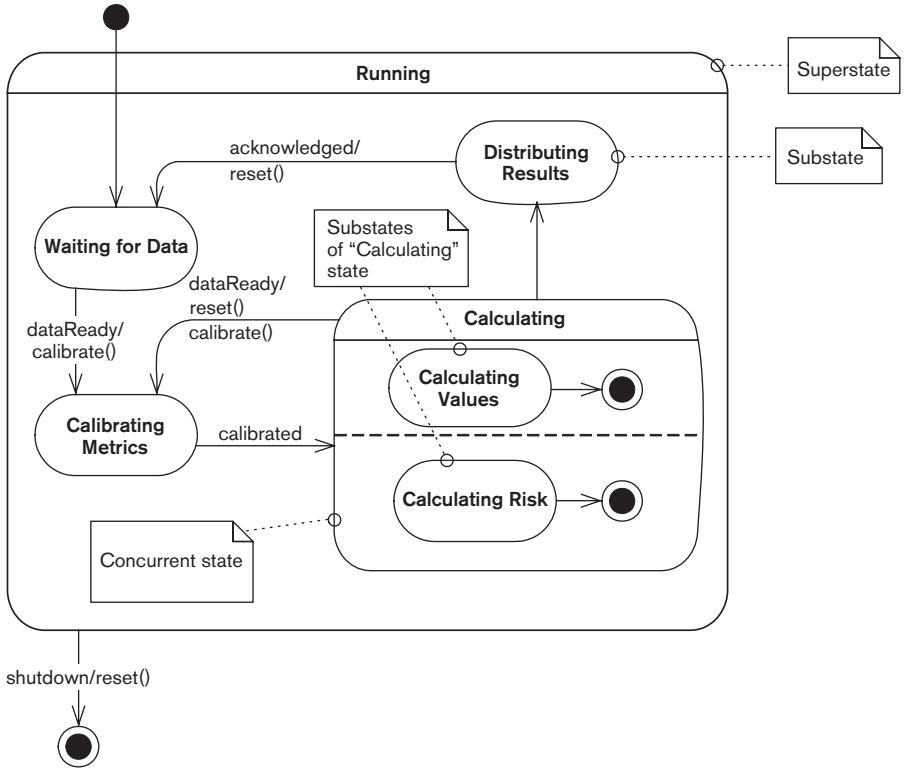

FIGURE 19–3 EXAMPLE STATECHART FOR A CALCULATION ENGINE

- No matter the state of the element, if a shutdown event is received, all processing immediately stops, the state is reset, and the element exits.

Of course, whether or not these facts are architecturally significant depends on the situation. However, we can make a reasonable argument that these facts are visible at the system level and thus can affect or be relied on by other system elements, and therefore, these facts need to be captured as part of the architecture.

An interesting point to note about the UML statechart is that its ability to show hierarchical state composition allows you to express architectural constraints on state models without needing to define the entire model. The statechart in Figure 19–4 illustrates this point.

This statechart distills one of the architecturally significant features from the statechart in Figure 19–3, namely, that a shutdown event must be immediately responded to in any running state and a reset of the element performed as part of shutdown. In effect, this documents an architectural constraint that the designer of the corresponding part of the system

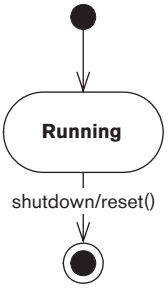

FIGURE 19–4 ARCHITECTURAL CONSTRAINT STATECHART

must respect; but while clearly defining this constraint, the statechart leaves the details of the lower-level states to the subsystem designer.

- *Other graphical notations*: In addition to UML, many other graphical notations exist for modeling state. Some of the better-known ones include simple state transition diagrams, Petri Nets, SDL, and David Harel's original Statecharts. All of these notations have strengths and weaknesses when compared to each other and to UML statecharts, and they are worth considering if UML statecharts cause you problems. However, the standardization of and wide familiarity with UML statecharts means that in general they should probably be your first choice. The Further Reading section at the end of this chapter contains some references to information about these alternative notations.
- *Nongraphical notations*: In principle, all of the graphical notations can be represented in a textual form (and indeed many graphical state modeling notations do define an equivalent textual form). Similarly, a number of primarily textual formalisms for modeling and analyzing state can be represented in a graphical form (for an accessible example, look at the Finite State Processes language). A textual state model can be useful when the model needs to be processed in some way by machine, but for human readers it is almost always better to use a graphical notation where possible.

ACTIVITIES

Define the Notation. Before starting to create your state model, spend some time working out your needs for the modeling notation and defining how you will use it.

Identify the States. The primary activity when creating a state model is to work out what states your system elements can be in and the processing (if any) associated with each state. Beware of accidentally modeling activities as

states; this is a common modeling mistake. If in doubt, try considering your state machine as a UML activity diagram. If you can do this, you have probably modeled activities rather than states. When performing state identification at the architectural level, focus on the states that are visible from outside the element and thus have a system-wide effect.

Design the State Transitions. Once you know what states your elements can be in, design a set of transitions that allows them to move between the states correctly. For each transition, clearly identify how the transition is triggered and any (atomic) actions that must be performed as a side effect of traversing it. Make sure that the events and actions you identify can be supported by the operations and state of the element for which you are designing a state machine.

PROBLEMS AND PITFALLS

Modeling the Wrong Concurrency

When considering the concurrency design of a system, it is easy to get bogged down in the details of the internal concurrency and state design of each element. It's not part of your job as an architect to design detailed thread models that define how individual threads in a server will be allocated, used, and freed, along with all of the coordination between them. Remember that your role is to concentrate on the system as a whole rather than all of the details of each element. The concurrency with which you should be concerned is the architecturally significant concurrency, that is, the overall concurrency structure, the mapping of functional elements to that structure, and the system-level state model. You may also be involved in specifying common approaches such as design patterns that need to be used for the concurrency within elements, but in general you should not need to design all of the details—this will only distract you from the system-level problems (which are often quite enough to worry about).

RISK REDUCTION

- Focus on architecturally significant aspects of concurrency.
- Involve the lead software developers as early as possible so they can work on the more detailed aspects of this problem.

Modeling the Concurrency Wrongly

Meaningful concurrency models can be quite difficult to create, so it is important to spend the time and effort required to create a good model. To be use-

ful, the models you create should use your notation correctly and be a valid representation of the situation you are representing. The following are some of the common modeling mistakes to be aware of.

- Modeling activities from your system as states in your state model and so accidentally creating an activity diagram rather than a state model. This is a common confusion when people are new to state modeling.
- Having a model with nonterminal states that can't be exited because the required events will never occur. This indicates that some conditions haven't been thought through.
- Having transitions that cannot be traversed due to an invalid combination of events and conditions. This can indicate a misunderstanding or that something is missing.
- The existence of a large number of transitions with just trigger events or just actions. Such transitions are sometimes required, but having a large number of them often indicates that there are too many states in the state model.

RISK REDUCTION

- Before you start, take the time to understand your concurrency modeling notation (and the way that your tool implements the notation, if you're using a modeling tool).
- Watch out for states, transitions, or actions that are difficult to name or that seem to need the wrong sort of name (such as a verb for a state). These suggest that there is something wrong with the model.
- Walk through your models, "playing computer" in order to validate them and check for missing or incorrect elements. If your tool offers model animation facilities, these are a more reliable way of achieving this.

Excessive Complexity

Simplicity should always be an aim when designing a system. Simple designs are easier to create, analyze, build, deliver, and support. However, this is particularly important when considering concurrency because it is fundamentally difficult to understand. As we have seen, the price of complex concurrency can be very high at design time, implementation time, and beyond. More software engineering hours have probably been wasted on reworking problematic concurrency than on almost anything else. Simplicity in your concurrency approach will have a major positive impact on the amount of effort required to deliver and support your system.

RISK REDUCTION

- Be sure that all of the concurrency you introduce is justified in terms of stakeholder benefits.
- When designing state models, use the simplest subset of notation possible to capture your state machines in order to encourage a simple state model.

Resource Contention

Resource contention usually manifests itself as long wait times for shared resources or excessive activity in small, specific parts of the system (colloquially known as *hot spots*). Careful and early analysis of the concurrency model for potential contention can help you avoid such problems, but in reality, as soon as one resource contention point is eliminated, the next one will emerge. Therefore, tackling resource contention is normally a process of reducing the contention to an acceptable level.

RISK REDUCTION

- Analyze your system as it is being designed to spot resource contention as early as possible, and design around it. Use your usage scenarios to predict which parts of the system are likely to encounter high levels of concurrency, and focus your attention in these regions.
- Reduce contention by decomposing locks on large resources into a number of finer-grained locks, thus reducing the amount of time locks are held.
- Consider alternative locking techniques such as optimistic locking that reduce the time locks are held.
- Eliminate shared resources where you can, or consider making them immutable to avoid the need to lock them when accessed by multiple tasks.
- If possible, reduce the amount of concurrency you need around problematic contention points.
- Consider whether it is possible to avoid locking by using approximately correct results that may be in the process of being updated or whether it would be possible to allow copies of data to be loosely rather than tightly consistent, to avoid locking them all simultaneously during the update process.

Deadlock

Deadlock occurs when one task, A, requires access to a resource that has already been locked by another task, B, and task B is also waiting for access to a resource that has been locked by task A. We describe these two tasks as

"deadlocked," and neither of them will make progress until the deadlock is broken by one of the tasks releasing one of the locks (possibly by the task being terminated by a supervisor task). As with resource contention, you can often avoid deadlock through early and thorough analysis of the system. Danger points are those parts of the system where different types of processing tasks need access to a number of the same resources. When you find potential deadlock points, you will probably need to redesign the system to avoid the problem.

RISK REDUCTION

- Where possible, ensure that resources are always allocated to tasks and locked in a fixed order.
- Attempt to isolate parallel tasks in such a way that deadlock between them is impossible.
- Reduce the number, scope, and duration of locks held where possible (or even use immutable data structures if possible, to completely avoid locking).
- Certain commercial products that use locks (such as database management systems) provide significant assistance with handling deadlock—in most cases, recognizing it and breaking it by terminating one or more of the problematic transactions. These technologies can be very useful when dealing with deadlock, but their use often needs to be carefully designed into the system so that such deadlock recovery actions are handled correctly.

Race Conditions

A race condition is problematic behavior that results from unexpected dependence on the relative timing of events. It usually occurs when two or more tasks are attempting to perform the same action concurrently. The tasks race for the resource, and the first one to reach the appropriate point in the program code wins and performs the action.

Race conditions are problematic only when they are unplanned because the system has not been designed to cope with more than one task performing the action concurrently. In these cases, information can be corrupted or lost, and the system can behave in unpredictable ways. A classic example is a system-wide data structure in an operating system process that a number of threads can update. If multiple tasks try to update the data structure concurrently (e.g., to increment a counter indicating the number of requests accepted), the resulting value will be undefined and very likely incorrect.

RISK REDUCTION

- Ensure that there are no unprotected, shared system-level resources that can cause race conditions.
- Use immutable data structures where possible to avoid the possibility of race conditions.
- Automatically introduce protection mechanisms for all potentially shared resources.
- Ensure that the definition of each element interface clearly states whether or not the interface is reentrant.

CHECKLIST

- Is there a clear system-level concurrency model?
- Are your models at the right level of abstraction? Have you focused on the architecturally significant aspects?
- Can you simplify your concurrency design?
- Do all interested parties understand the overall concurrency strategy?
- Have you mapped all functional elements to a process (and thread if necessary)?
- Do you have a state model for at least one functional element in each process and thread? If not, are you sure the processes and threads will interact safely?
- Have you defined a suitable set of interprocess communication mechanisms to support the interelement interactions defined in the Functional view?
- Are all shared resources protected from corruption?
- Have you minimized the intertask communication and synchronization required?
- Do you have any resource hot spots in your system? If so, have you estimated the likely throughput, and is it high enough? Do you know how you would reduce contention at these points if forced to later?
- Can the system possibly deadlock? If so, do you have a strategy for recognizing and dealing with this when it occurs?

FURTHER READING

The area of concurrency has been studied and written about widely, although not many books consider it from an architect's perspective.

A good overview of concurrency (albeit with a Java-specific slant) and a good introduction to (fairly formal) modeling and analysis appear in Magee and Kramer [MAGE06]; this book also introduces the Finite State Processes language mentioned earlier. Unfortunately, Cook and Daniels [COOK94] is out of print; however, it has recently made a welcome reappearance as a freely available online book at www.syntropy.co.uk/syntropy, as it contains a lot of good advice on modeling and a particularly good discussion of using statecharts to model object state. You can find a lot of good UML-specific advice about state modeling in Rumbaugh et al. [RUMB99], which is organized as a reference so it's easy to find definitions of the various UML elements involved.

Each of the visual formalisms has its own following and literature. Girauld and Valk [GIRA02] is a relatively academic text that explains how to apply Petri Nets to the analysis of concurrency characteristics, while the SDL Forum Society Web site [SDL02] is a good starting point for finding out more about SDL. A fairly recent reference on CSP is Roscoe [ROSC97]; the definitive book on CCS is still Milner [MILN89]; and the original reference for statecharts was Harel's paper [HARE87].

We reference it elsewhere too, but Michael Nygard's book [NYGA07] has much to say on the topic of safely introducing concurrency into systems. A rich collection of design patterns for creating concurrent systems is documented in [SCHM00], and a thorough programming-level "nuts and bolts" introduction to concurrency practice can be found in [BRES09]

20

The Development Viewpoint

Definition	Describes the architecture that supports the software development process
Concerns	Module organization, common processing, standardization of design, standardization of testing, instrumentation, and codeline organization
Models	Module structure models, common design models, and codeline models
Problems and Pitfalls	Too much detail, overburdened architectural description, uneven focus, lack of developer focus, lack of precision, and problems with the specified environment
Stakeholders	Production engineers, software developers and testers
Applicability	All systems with significant software development involved in their creation

A considerable amount of planning and design of the software development environment is often required to support the design, build, and testing of software for complex systems. Things to think about include code structure and dependencies, build and configuration management of deliverables, system-wide design constraints, and system-wide standards to ensure technical integrity. It is the role of the Development view to address these aspects of the system development process, as it is this view that addresses the specific concerns of the software developers and testers.

This viewpoint is relevant to nearly all large information system projects because almost all of them have some element of software development, whether it is configuring and scripting off-the-shelf software, writing a system from scratch, or something between these extremes. The importance of this view depends on the complexity of the system being built, the expertise of

the software developers, the maturity of the technologies used, and the familiarity that the whole team has with these technologies.

In this view you need to focus on concerns that are architecturally significant. You should view your work as providing a stable environment for the more detailed design work that will be performed as part of the software development activity.

CONCERNS

Module Organization

The large systems you are likely to encounter as an architect may be built from hundreds of thousands of lines of source code spread over thousands of files. Source files are normally organized into larger units called *modules* that contain related code (such as the code to implement a library or a functional element). Arranging code in a logical structure like this helps to manage dependencies and helps developers to understand it and work on it without affecting other modules in unexpected ways.

When working with a complex module structure, you need to identify and thoroughly understand and manage the dependencies between the modules to avoid ending up with a system that is difficult and error-prone to maintain, build, and release.

Common Processing

Any large system will benefit from identifying and isolating common processing into separate code modules. For example, standardizing how the system logs messages and handles configuration parameters can significantly simplify its administration.

The Development view helps ensure that the areas of common processing are identified and clearly specified. You will typically do this only in outline form, adding further refinement and detail as development progresses.

Standardization of Design

Most systems are developed by teams of software developers rather than individuals. Standardizing key aspects of design provides critical benefits to the maintainability, reliability, and technical cohesion of the system (and saves time, too). You can achieve design standardization by using design patterns and off-the-shelf software elements.

Standardization of Testing

Standardization of test approaches, technologies, and conventions helps ensure a consistent approach to testing and speeds up the testing process. Key concerns include test tools and infrastructure, standard test data, standard test approaches, and test automation.

Instrumentation

Instrumentation is the practice of inserting special code for logging information about step execution, system state, resource usage, and so on that is used to aid monitoring and debugging. Because instrumentation can have an adverse impact on performance, it should be possible to switch off this capability, alter the level of detail at which messages are logged, and possibly even use build tools to remove the instrumentation code altogether.

System messages can be logged to a system console, a file, or a message service, and metrics on system usage can be logged to a file or a database for later analysis.

Codeline Organization

The system's source code needs to be stored in a directory structure, managed via a configuration management system, built and tested regularly (ideally every time the software changes—"continuous integration"), and released as tested binaries for further testing and use. The way that all of this is achieved is normally termed the *codeline organization* for a system. The *codeline* is a particular version of a set of source code files with a well-defined organizational structure, usually with an associated automated system to build, test, and release a specified version or variant of the system.

Ensuring that the system's code can be managed, built, tested, and released is crucial to achieving a reliable system—particularly when you're using iterative development and many releases are necessary. As an architect, you may wish to specify, in outline form at least, how this is to be done, or better still, work with the development team to define the approach and design its implementation.

Stakeholder Concerns

Typical stakeholder concerns for the Development viewpoint include those shown in Table 20–1.

TABLE 20–1 STAKEHOLDER CONCERNS FOR THE DEVELOPMENT VIEWPOINT

Stakeholder Class	Concerns
Developers	All concerns
Production engineers	May be involved in or have responsibility for provisioning development and test environments, and mechanisms and controls over the system's transition into production
Testers	Common processing, instrumentation, test standardization, and possibly codeline organization

MODELS

Module Structure Models

The module structure model defines the organization of the system's source code, in terms of the modules into which the individual source files are collected and the dependencies among these modules. It is also common to impose some degree of higher-level organization on the modules themselves to avoid having to enumerate many individual dependencies.

Once you have identified a set of modules into which you can organize the source files, you can use the common architectural approach of grouping modules at similar abstraction levels into layers. You can then organize these layers into a dependency stack from the most abstract or highly functional (conceptually at the top) down to the least (at the bottom). You can then define interlayer dependency rules to avoid unwanted dependencies between modules at very different abstraction levels. Typically, software in a module communicates only with other modules at the same layer or in the layers directly above and below it (although there are often exceptions to this rule for performance or efficiency reasons).

In some situations (e.g., when separate module structures are needed for client and server elements), you may need a number of such models. In other cases (e.g., when developing an extension to a monolithic application package), a module structure model is less useful.

NOTATION A module structure model is often represented as a UML component diagram, using the package icon to represent a code module and dependency arrows to show intermodule dependencies. If you require higher-level module organization, you can show module grouping by enclosing packages annotated with suitable stereotypes.

Another common alternative is a simple boxes-and-lines diagram that shows the layers, their relative ordering, and the components within them.

EXAMPLE Figure 20–1 shows an example of using UML to document a module structure model.

This layer model shows a module organization with three layers, each layer being represented by a stereotyped package. The system's modules are shown as UML packages within the layers.

The model shows that the domain layer depends on the utility layer, which in turn depends on the platform layer (i.e., the domain-layer components can access only the utility-layer components, and so on).

However, you can also see that nonstrict layering has been used in this system because all of the domain-layer components depend on facilities provided by the Java Standard Library component rather than accessing its facilities via intermediate utility components. (In contrast, the domain-level components cannot access the JDBC Driver component.)

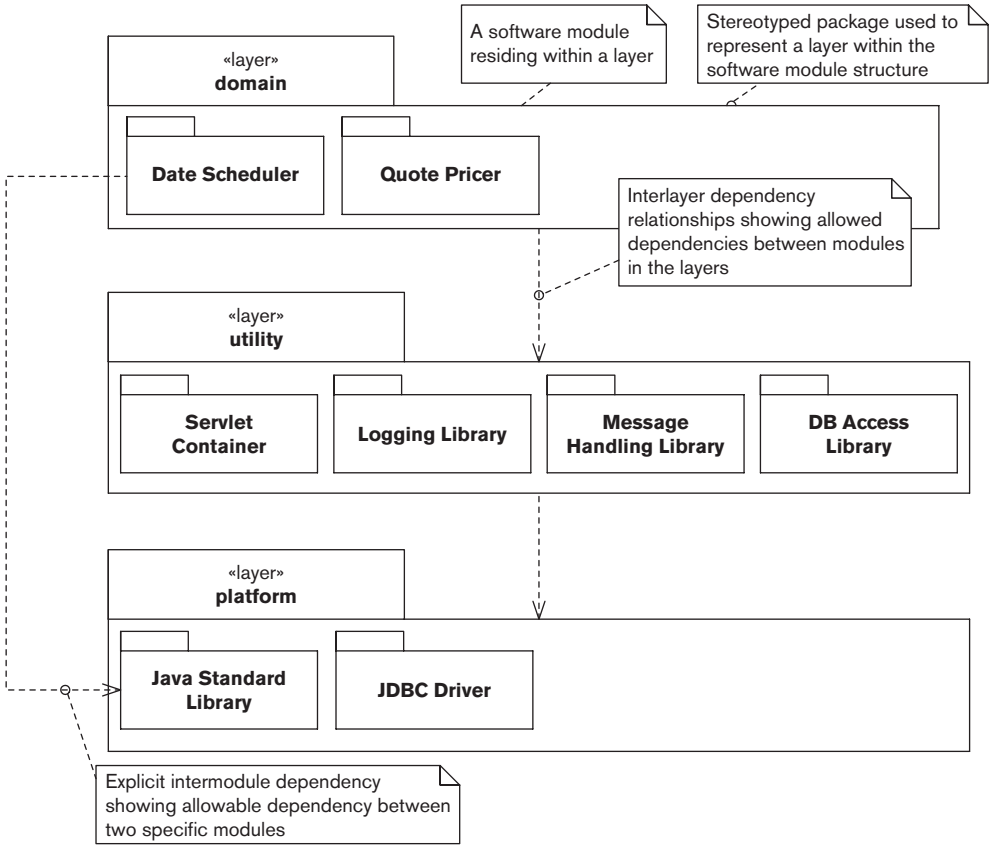

FIGURE 20–1 EXAMPLE OF A UML MODULE STRUCTURE MODEL

ACTIVITIES

Identify and Classify the Modules. Group the source code for the system into a set of modules, and (optionally) classify them—by abstraction or other criteria—into a higher-level organization.

Identify the Module Dependencies. Identify a clear set of dependencies between the modules (or the higher-level groups) so that everyone involved in the design and construction of the system can understand the impact of making changes.

Identify the Layering Rules. If a layered approach is to be used, you need to design a set of rules to be followed with respect to the layers. Can modules call modules only in their own layer and the one above or below, or do you want a less rigid rule in order to meet system quality properties such as performance and flexibility?

Common Design Models

To maximize commonality across element implementations, it is desirable to define a set of design constraints that apply when designing the system's software elements. Such design constraints are valuable for two principal reasons.

- You can reduce risk and duplication of effort by identifying standard approaches to be used when solving certain types of problems.
- Commonality among system elements helps increase the system's overall technical coherence and makes it easier to understand, operate, and maintain.

A common design model has the following three important pieces.

1. A definition of the *common processing* required across elements, such as:
   - Initialization and recovery
   - Termination and restart of operation
   - Message logging and instrumentation
   - Internationalization
   - Use of third-party libraries
   - Processing configuration parameters (at startup or while running)
   - Security (e.g., authentication or encryption)
   - Transaction management
   - Database interaction
   - Internal and external interfacing

   These aspects of your software element designs can benefit greatly from using a standard approach across all system elements. Identifying

and defining common processing is a key architectural task that directly contributes to the overall technical coherence of the system.

2. A definition of *standard design approaches* that should be used when designing the system's elements. These start to emerge when (having defined the functional structure) you think ahead a little about how the subsystems might be implemented. When you see situations where the same sort of processing is performed by different elements, or where you know that the implementation of a certain aspect of an element will have a system-wide impact, you should consider whether you need a standard design approach. When identifying such an approach, you must define what the approach is, where it should be used, and why it should be used. In other words, it is a special sort of design pattern.
3. A definition of what *common software* should be used and how it should be used. This may be the result of making other higher-level decisions (e.g., selecting an access library for your chosen database) or identifying a reusable component (e.g., a third-party message-logging library or a locally developed graphical user interface element) that can save you development time and reduce risk. In either case, your common design model needs to clearly identify what common elements should be used, where they should be used, and how they should be used.

As with the module structure model, you may need to define different design constraints for different parts of the system. In any case, as an architect, you are only starting a task that will continue throughout the design and build.

NOTATION The common design model is a partial design document, and as such, the notations it uses are those of software design—usually a combination of text and more formal notation such as UML.

The following example shows some possible design constraints from a common design model.

EXAMPLE Here is an example of a common design model.

Common Processing Required

1. Message logging

    - All components must log human-readable messages that clearly state what has occurred and any corrective action that is expected in response.

- Messages must be logged at one of the following levels: Fatal, Error, Warning, Information, Debug. Fatal should be used to indicate an unrecoverable error, where the component will stop immediately; Error indicates an unrecoverable error, where the component can reset itself and continue execution; Warning indicates a possible error or unexpected condition that may need operator intervention to review and address; Information is used to report conditions that occur during normal operation and require no operator intervention; Debug should be used to indicate internal details of the component's operation.
- Components should log messages at all five possible logging levels.
- Logging should be achieved via a standard library (as defined later) to standardize destination, format, configuration, and so on.

[. . .]

2. Internationalization

- All user- and administrator-visible strings must be stored in message catalogs so that hard-coded strings are not present in source code.
- Parameters must be inserted into internationalized strings using position-independent placeholders to avoid problems with ordering across languages.
- Locale-sensitive information (dates, times, currency strings, and so on) must be formatted according to the current locale in force, and default formats should not be used.
- Strings logged at Debug level or for other purely internal use should not be internationalized but should be hard-coded in the source code.

[. . .]

Standard Design

1. Internationalization

- For internationalization of locale-sensitive resources (primarily strings), use an external resource catalog to store resources outside the source code files. This means that all strings must be extracted from a message catalog before they can be used in a program (e.g., to write a log message).
- As the server software is being written entirely in Java, the internationalization implementation will use the Java Platform's native internationalization facilities: the resource bundle, the formatting classes in the `java.text` package, and the `Locale` class.

- The relationships between these different elements of the internationalization technology are as follows. [. . .]
- *[You would place a definition of a design pattern for using the Java internationalization facilities here.]*

[. . .]

Standard Software Components

1. Message logging

- All message logging must be performed using the standard `CCJLog` package, which is part of the standard build environment.
- The `CCJLog` package must be used in a standard way, which is documented as a code sample in the `src/server/sample/logging/CCJLog` source directory.

[. . .]

ACTIVITIES

Identify Common Processing. Identify *what* common processing is required, *where* the processing is required (in all elements or just some?), and *how* the common processing should be performed.

Identify the Required Design Constraints. Establish whether any common processing should be standardized and whether critical aspects of subsystem design will have a negative system-wide impact if not designed in a certain way. If you find such situations, consider whether you can impose a design constraint that will resolve the problem, and, if so, add it to the list.

Identify and Define the Design Patterns. Document a set of mini design patterns that clearly define the constraints. The constraints are defined in terms of the software design that needs to be followed, the applicability of the constraint (i.e., where to use it), and the rationale for the constraint (to allow those following it to understand its role).

Define the Role of Standard Elements. Consider whether you have any standard software elements that can be shared among subsystems. You will often identify such standard elements when considering the system's common processing. If you find standard elements, clearly define their roles and how they should be used.

Codeline Models

Although you certainly don't want to be dictating the minutiae of the software developers' lives, you do need to ensure that there is order rather than chaos when it comes to the organization of the system's code.

The key things to define are the overall structure of the codeline; how the code is controlled (usually via configuration management); where different types of source code live in that structure; how it should be maintained and extended over time (in particular, how any concurrent development of different releases should work); and the automated tools that will be used to build, test, release, and deploy the software. A codeline model normally needs to capture the following essential facts:

- How code will be organized into source files
- How the files will be grouped into modules
- What directory structure will be used to hold the files
- How the source will be automatically built and tested to form candidate releasable binaries
- What type and scope of tests will need to be run regularly and when they should be run
- How the binaries will be released into a test or production environment for testing and use, again ideally via an automated process
- How the source will be controlled using configuration management (including any use of branching, change sets, and so on) to coordinate multiple developers working on it concurrently
- What automated tools will be used for the build, test, and release process and how they will work together in order to form a complete continuous integration and delivery system

Defining these aspects of the development environment is an important part of achieving reliable, repeatable build and release processes. The information you provide through your model will help prevent confusion and frustration as developers work together.

In situations where development of the system will be distributed among different teams or among members of teams working at different locations, addressing this concern becomes even more important. You may have to take into account factors such as different time zones or even the different languages spoken by development staff.

Depending on the skill and experience of the developers, you may be comfortable leaving the majority of this work to your design team; at the other extreme, you may want to specify this in some detail.

NOTATION In principle, you can represent the codeline model by using structured notations such as UML. However, our experience of trying this suggests that it often isn't worth the bother. A simple approach based on text and tables with a few clear diagrams to explain the conventions used should suffice.

ACTIVITIES

Design the Source Code Structure. Design the overall structure of the directory hierarchy to be used to store your system's source code. This must be flexible enough to provide easy maintenance but simple enough that developers know where their source files should live.

Define the Build, Integration, and Test Approach. To achieve a reliable system build process, you need to mandate a common approach across the system. A build and release specialist may do this for you, but the approach used for automating the build, integration, and testing does need careful design. Whatever approach you use, it must make it possible to easily build the system automatically and also allow developers to use central or local copies of the latest build.

Define the Release Process. Having completed a clean build of the system, you need to release the resulting work products (binaries, libraries, generated documentation, and so on) for testing and use. To ensure that this process is reliable and repeatable, you must design a clear process, again preferably automated. As before, specialists may do the design for you, or you may need to do it yourself. It is particularly important to be clear about the build validation (such as automated test suite execution) that needs completion before release. This process will need to use any deployment tools that are required in your environment, whether internal to your organization or supplied by a third party if you are deploying software to an external hosting environment, such as externally hosted servers or a public cloud computing service.

Define the Configuration Management. To ensure repeatability and technical integrity, you must use a common approach to configuration management. Its definition should encompass the tools to be used, the configuration structures (such as variants, branches, and labels) to be used, and the process for managing the deliverables under configuration control.

PROBLEMS AND PITFALLS

Too Much Detail

Most software architects are experienced software designers, which means that you probably have a lot of background knowledge related to the process of software design and implementation. The danger that stems from this is the temptation to use the Development view to define low-level details about the system's implementation that are really the concern of the designers and implementers.

RISK REDUCTION

- Minimize the number of design constraints you identify. Identifying too many is often counterproductive and causes problems as developers try to shoehorn their elements into the space left by a number of different constraints (or simply ignore them).
- Carefully review everything you describe in the Development view, and question whether it is architecturally significant. If not, eliminate that detail from the Development view.

Overburdened Architectural Description

A problem related to having too much detail is the question of where to put the contents of the Development view (particularly in the common design model). For a complex system, the common design model can require a significant amount of text, and given that it is aimed at a specialized group of stakeholders, it can seem out of place in the main AD document.

RISK REDUCTION

- Capture the details of the system-wide design constraints in a separate document specifically aimed at the software developers, and then summarize the constraints required and their rationale in a short section of the AD. This allows interested stakeholders to satisfy themselves that the design constraints have been considered, without needing to understand the details of these constraints.

Uneven Focus

We all have a tendency to focus on things that we understand and find interesting. This can lead to a situation where, for example, the design patterns to be used for network request handling are discussed in minute detail, but the initialization processing required of each element is hardly considered at all.

RISK REDUCTION

- Try to step back from the system and consider all of the aspects of software development that need to be defined at an architectural level.
- Find specialist expertise to advise you in areas you aren't familiar with.

Lack of Developer Focus

Always remember that the primary (and often only) customers of the Development view are the software developers and testers working on your project.

The Development view must answer their questions and be relevant to their concerns. If it isn't, it will almost certainly be ignored.

RISK REDUCTION

- Involve the developers and testers in defining the Development view.
- Delegate aspects of the view's development to senior software developers when possible, to give the software development team ownership of the aspects of the architecture that affect them.

Lack of Precision

Because the Development view has to cover many aspects of the software development, and because you are unlikely to have expertise in all of them, lack of precision is a risk. Developers might misinterpret imprecise descriptions or, if they cannot understand the descriptions, might ignore them altogether.

RISK REDUCTION

- This problem often occurs when an architect knows that it is important to define some aspect of the system but knows little about it and thus simply states that it needs to be performed. When defining the Development view, make sure to review its contents early with the software developers and testers to check that the view's definitions are precise enough.
- Do not be afraid to make use of the knowledge of subject matter experts where your experience is limited—you are not expected to be an expert in everything!

Problems with the Specified Environment

Keeping up-to-date with new and emerging technologies takes a lot of time. It is particularly hard to get reliable information on how mature those technologies are and how appropriate they might be for your architecture.

This imposes the risk of specifying aspects of the Development view based on out-of-date (or perhaps just incorrect) knowledge and assumptions, which can lead to later problems in development or live operation and damage your credibility with developers.

The other related mistake that is easy to make is to try to impose approaches that have worked well before but that are simply wrong for the project teams in the environment in which you're currently working. Short-lived stand-alone systems require a different development environment from huge long-lived product lines that will be used for many years by external customers. Make sure you understand the needs and constraints of the project environment before trying to define the Development view.

RISK REDUCTION

- Make sure you specify technology and techniques you really know about, or get trusted, expert advice from subject matter experts to help make the relevant decisions.
- Understand what is needed in the current project environment and make sure that your Development view reflects these needs and doesn't overcomplicate or oversimplify the development environment.
- Delegating the research and design of aspects of the Development view to members of the software development team can help alleviate this problem while having other positive side effects, such as giving the software developers a heightened sense of ownership of the system.

CHECKLIST

- Have you defined a clear strategy for organizing the source code modules in your system?
- Have you defined a general set of rules governing the dependencies that can exist between code modules at different abstraction levels?
- Have you identified all of the aspects of element implementation that need to be standardized across the system?
- Have you clearly defined how any standard processing should be performed?
- Have you identified any standard approaches to design that you need all element designers and implementers to follow? If so, do your software developers accept and understand these approaches?
- Will a clear set of standard third-party software elements be used across all element implementations? Have you defined the way they should be used?
- Will the development and test environments that have been defined work reliably and be usable and efficient for developers and testers to work in?
- Have you or someone else defined a suitable set of tools to reliably automate the end-to-end build, integration, test, and release processes? Does the set of tools include any internal or third-party tools that you require to deploy to the internal or external test and production environments that you are using?
- Is this view as minimal as possible?
- Is the presentation of this view in the AD appropriate?

FURTHER READING

Many books discuss the use of design patterns in software development, the original book being, of course, Gamma et al. [GAMM95]. This topic is explored further in Coplien et al. [PLOP05–99, PLOP06].

There are a number of good books covering relevant topics such as configuration management, continuous integration, automated testing, release processes, and so forth. [AIEL10] is a fairly high-level overview of the entire area, focusing on configuration management and release control, and [BERC03] is a very thorough guide to software configuration management, illustrated using a set of patterns. [DUVA07] is a thorough and practical guide to continuous integration, and [HUMB10] is a detailed guide to automating the processes involved in building, testing, and releasing software. Finally, there are a large number of books on automated testing, but we particularly like [FREE09], which provides lots of practical advice on automation but also gets behind the mechanisms to show how and why to put automated testing at the heart of the development process.

This page intentionally left blank

21

The Deployment Viewpoint

Definition	Describes the environment into which the system will be deployed and the dependencies that the system has on elements of it
Concerns	Runtime platform required, specification and quantity of hardware or hosting required, third-party software requirements, technology compatibility, network requirements, network capacity required, and physical constraints
Models	Runtime platform models, network models, technology dependency models, and intermodel relationships.
Problems and Pitfalls	Unclear or inaccurate dependencies, unproven technology, unsuitable or missing service-level agreements, lack of specialist technical knowledge, late consideration of the deployment environment, ignoring intersite complexities, inappropriate headroom provision, and not specifying a disaster recovery environment
Stakeholders	System administrators, developers, testers, communicators, and assessors
Applicability	Systems with complex or unfamiliar deployment environments

The Deployment view focuses on aspects of the system that are important after the system has been built and needs to be validation tested and transitioned to live operation. This view defines the physical environment in which the system is intended to run, including the hardware or hosting environment (e.g., processing nodes, network interconnections, and disk storage facilities), the technical environment requirements for each type of processing node in the system, and the mapping of your software elements to the runtime environment that will execute them.

A Deployment view is useful for any information system with a required deployment environment that is not immediately obvious to all of the interested stakeholders. This includes the following situations:

- Systems with complex runtime dependencies (e.g., specific third-party software packages or particular network services are needed to support the system)
- Systems with complex runtime environments (e.g., elements are distributed over a large number of machines)
- Systems hosted in third-party environments, such as hosting services or public clouds, in order to allow a clear definition of the environment required and how the system will deploy into it
- Situations where the system may be deployed into a number of different environments and the essential characteristics of the required environments need to be clearly illustrated (which is typically the case with packaged software products)
- Systems that need specialist or unfamiliar hardware or software in order to run

In our experience, most large information systems fall into one of these groups, so you will almost always need to create a Deployment view.

CONCERNS

Runtime Platform Required

The Deployment view must clearly identify the type of runtime platform that the system needs and the role that each part of it plays. This includes general-purpose compute nodes to host servers and computational logic, special-purpose compute nodes to host database engines, storage for databases and file systems, devices that allow users to access the system or print information, network services required to meet certain quality properties (such as firewalls for security), specialist hardware (such as cryptographic accelerators), and so on. The manner in which the platform is provided, whether it be physical hardware commissioned in-house, virtual servers and storage provided by a third-party hosting company, the use of a public cloud computing environment, or some other option, needs to be clearly defined too, as does the location of each part of the platform.

Defining the runtime platform involves identifying the general types of processing elements required (such as compute server node, application server node, storage array, and so on), defining the dependencies between them, and mapping each of your functional elements to one of these types. In

effect, this is a logical model of the runtime platform that your system requires. Then, when you have defined what each piece of the platform is used for, you can think about the details of exactly what hardware elements you need to provide it.

Specification and Quantity of Hardware or Hosting Required

This concern, which follows from the previous one, addresses the specific details of the hardware that will need to be procured and commissioned in order to deploy the system—in effect, a physical model of the hardware your system needs. This hardware may need to be ordered and commissioned in-house or via a third party or may be specifications for a virtual computing environment, such as ordering capacity from a cloud computing supplier.

This is a separate concern from the previous one because it is much more specific and of interest to different stakeholders. For example, developers are interested in whether the deployment platform will use Intel or Sun SPARC servers; whether the servers will run Linux, HP-UX, or Windows; and what general processing resources will be available to them. However, system administrators are interested in the detailed specification and quantity of the hardware elements or specification of the hosting environment that needs to be acquired to create your runtime environment. The service-level agreements (SLAs) for each part of the runtime environment will also need to be agreed to and validated as acceptable for the level of service your system needs to provide.

Be specific when considering the specification, quantity, and service level of the hardware and services that you need. If specific models of equipment or specifications of hosted environment services are required, you need to clearly identify and record them for easy reference. If specific models or services aren't required, you should still be precise where needed.

Third-Party Software Requirements

All information systems make use of third-party software as part of their deployment environment—even if only an operating system. Many information systems make use of dozens of third-party software products, including operating systems, programming libraries, messaging systems, application servers, databases, data movement products, Web servers, and so on. If you are deploying your system to a platform-as-a-service environment, there is probably a specific set of platform services and options that you need in order for your system to run successfully.

Your Deployment view should make clear all of the dependencies between your system and any third-party software products. This ensures that the

developers know what software will be available for them to use and that the system administrators know exactly what needs to be installed and managed on each piece of hardware. It also helps you to spot any gaps in your analysis as early as possible.

Technology Compatibility

Each software and hardware element in your system may impose requirements on other technology elements. For example, a database interface library may require a particular operating system network library in order to function correctly, or a disk array may require a particular type of interface in the machines that will access it.

Furthermore, if you use a number of pieces of third-party technology together, there is always the danger of uncovering incompatible requirements. For example, your database interface library may require a certain version of the operating system, but a graphics library you want to use isn't supported on that version. Such incompatibilities have a habit of emerging late in the testing cycle and causing a lot of disruption—so if you consider them early, you will avoid problems later.

Network Requirements

Your Functional and Concurrency views define the functional structure of your architecture and make it clear how its elements interact. Part of the process of creating the Deployment view is to decide which hardware elements host each of these functional elements. Because elements that need to communicate often end up on different machines, some of the interelement interactions can be identified as network interactions.

One of the concerns the Deployment view addresses is the set of services that the system requires of its underlying network as a result of these network interactions. This view needs to clearly identify the required links between machines; the required capacity, latency, and reliability of the links; the communications protocols used; and any special network functions the system requires (load balancing, firewalls, encryption, and so on).

Network Capacity Required

In our experience, software architects need to get less involved in specifying network configuration than in identifying the processing and storage hardware because the network is normally provided by a group of specialists who design, implement, and operate the network for an entire organization.

However, this group needs to know how much network capacity your system requires and the type of traffic you need to carry over the network. In order to provide this information, you must estimate and record the amount and type of network traffic that needs to be carried over each intermachine link in the proposed network topology.

Physical Constraints

As software engineers we are lucky when compared to our colleagues working in other engineering disciplines. Normally, we don't have to worry that much about physical constraints because software has no weight, has no physical size, and occupies no physical space. However, when taking a system-level view, physical constraints suddenly become important again.

Considerations such as desk space for client workstations, floor space for servers, power, temperature control, cabling distances, and so on may seem relatively mundane. However, if someone doesn't consider them, your system simply won't be deployed. There is no point in specifying four monitors for each workstation if your users have desk space for only two. Similarly, if there isn't enough floor space in your data center for your servers, they won't be installed.

Stakeholder Concerns

Typical stakeholder concerns for the Deployment viewpoint include those shown in Table 21–1.

TABLE 21–1 STAKEHOLDER CONCERNS FOR THE DEPLOYMENT VIEWPOINT

Stakeholder Class	Concerns
Assessors	Types of hardware or hosting required, technology compatibility, and network requirements
Communicators	Types and specification of hardware or hosting required, third-party software requirements, and network requirements (particularly topology)
Developers	Types and (general) specification of hardware or hosting required, third-party software requirements, technology compatibility, and network requirements (particularly topology)
System administrators	Types, specification, and quantity of hardware or hosting required; third-party software requirements; technology compatibility; network requirements; network capacity required; and physical constraints
Testers	Types, specification, and quantity of hardware or hosting required; third-party software requirements; and network requirements

MODELS

Runtime Platform Models

The runtime platform model is the core of this view. This description defines the set of hardware nodes that are required, which nodes need to be connected to which other nodes via network (or other) interfaces, and which software elements are hosted on which hardware nodes.

A runtime platform model has the following main elements.

- *Processing nodes*: Each computer in your system is represented by one processing node in the runtime platform model. This allows you and other stakeholders to see what processing resources are required for the system. For situations where many similar machines are required (e.g., Web server farms), you can use a summary notation (such as UML's shadow notation) to simplify the diagram, but make sure that the number of nodes required is still clear.
- *Client nodes*: You also need to represent client hardware, but probably in less detail than the main processing hardware. You may have less control over client hardware than server hardware, and if this is the case, you need only represent the types and quantities of client machines required rather than the precise details of each. If you have special needs for presentation or user interaction hardware (e.g., touch screens, printers), this is specified as part of the client hardware.
- *Runtime containers*: Client and server nodes may need to provide a runtime container (such as a software application server or a client virtual machine) to provide a suitable runtime environment for the functional elements deployed onto them.
- *Online storage hardware*: This defines how much storage is needed, of what type, how it is partitioned, what it is used for, the assumptions you are making about its reliability and speed, and whether or not processing takes place close to its associated stored data. The storage hardware could be disk devices within a processing node or dedicated storage nodes such as disk arrays. Make the distinction between the two types clear so that the physical impact of separate storage nodes on the deployment environment is understood. You need to include the capacity (and possibly speed) of each type of storage hardware in the model.
- *Offline storage hardware*: Despite the ever-growing capacity of online storage hardware, many systems that deal with a lot of information still require offline storage (archives) as well. Somehow the problems always grow faster than the hardware capacity. Offline storage will also probably be required to allow backup of information held online. You need to ensure that there is sufficient capacity, that the hardware is fast enough to complete archive and retrieval in an acceptable time, and that there is

sufficient network bandwidth between it and the online storage. The requirements for the type, capacity, speed, and location of your offline storage hardware all need to be defined here.

- *Network links*: Your model needs to capture the essential connections required by your system (rather than your ideas on how the network will be built from specific network elements). It is sufficient at this point to show the links between your hardware nodes; you'll capture more details about the network, such as internode bandwidth requirements, in the network model (described next in this chapter).
- *Other hardware components*: You may need to consider specialist hardware for network security, user authentication, special interfacing to other systems, or specialist processing (e.g., for automated teller machines).
- *Runtime element-to-node mapping*: The final element of this model is a mapping of the system's functional elements to the processing nodes where they execute. How to go about defining this mapping depends on how complex your concurrency structure is. If you have a Concurrency view, you can map the operating system processes identified in that view to the processing nodes. If you don't have a Concurrency view, you can map functional elements from the Functional view directly to processing nodes (and in this case, presumably the details of the operating system processes in use aren't architecturally significant).

This runtime platform model is typically captured as a network node diagram that shows nodes, storage, the interconnections required between the nodes, and the allocation of the software elements between the nodes.

NOTATION Common notations used for capturing the runtime platform model include the use of UML, traditional boxes-and-lines diagrams, and textual notations. Each of these options is outlined in this subsection.

- *UML deployment diagram*: You can use a UML deployment diagram to document a runtime platform model. This diagram shows computing "nodes," and optionally "execution environments" (such as runtime containers), with "artifacts" representing the software elements deployed to them and the "communication paths" between the nodes (the communication path being a specialization of a UML association). Interelement dependencies can also be indicated on the diagram using regular or stereotyped UML dependencies. Figure 21–1 shows an example of using a UML deployment diagram as a simple runtime platform model that maps functional elements to processing nodes, in some cases with execution environments.

  When using the UML "artifact" to represent the software being deployed, it may be useful to show the actual binary files that are deployed. Artifacts can

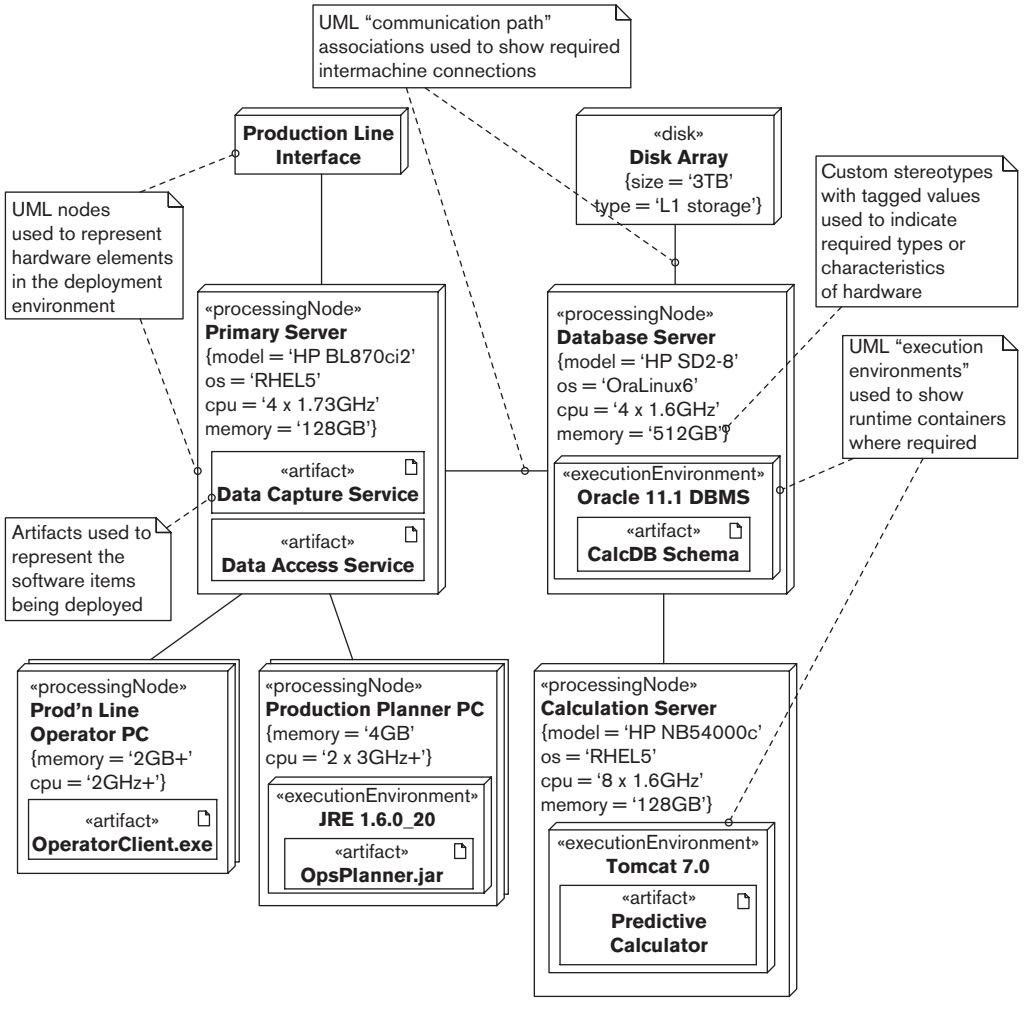

FIGURE 21–1 EXAMPLE OF A RUNTIME PLATFORM MODEL

also be used to represent entire system elements from the Functional view, which can be clearer and simpler. We show both styles of artifact in the diagram (e.g., the "OpsPlanner.jar" artifact is a deployed binary file, whereas the "Data Capture Service" is a system element, which is probably composed of a number of files). If you are using a UML tool to create your models, and the relationships between system elements and deployed artifacts aren't obvious, you may wish to use the «deploy» dependency to record these relationships.

UML does not provide very specific semantics for the nodes and communication paths and does not provide a library of predefined types

to choose from. Therefore, effective use of this diagram type usually relies on the use of stereotypes, tagged values, and comments in order to distinguish between different types of nodes and links. A runtime platform model also needs to be augmented by plain-text descriptions of the major elements, clearly defining the role and important characteristics of each.

- *Boxes-and-lines diagram*: Given the basic nature of the UML deployment diagram, many architects choose a simple boxes-and-lines notation for Deployment views. Boxes are used to represent nodes and elements, with arrows for interconnection; the diagram is annotated as required in order to make the meaning of each diagram element clear. With such an approach, you need to carefully define the diagrammatic elements used to avoid causing any confusion for the reader. This notation is easier to draw with drawing tools that don't support UML, and it may be more comprehensible to nontechnical stakeholders.
- *Text and tables*: Reference information such as required hardware specifications is best represented by text that is organized into tables for easy, unambiguous reference.

ACTIVITIES

Design the Deployment Environment. You typically start by identifying the key servers in the system, any important client hardware requirements, and the network links necessary between the nodes. With this done, you have the backbone of your deployment environment. The rest of the process is normally elaboration, adding any special-purpose hardware required (e.g., cryptographic accelerators, or nodes for redundant capacity) and specifying the hardware and software configurations for each node along with any interconnections.

Map the Elements to the Hardware. Once you have a proposed deployment environment, you need to find a home in it for each of your functional (software) elements. In reality, this is an iterative process where mapping the software elements to hardware resources may suggest changes in the deployment environment design (or newly identified deployment environment options may suggest new alternatives for software element locations). The main challenges here relate to managing dependencies, ensuring that enough machine capacity is available, and trading off the advantages of separated versus colocated elements (e.g., security versus performance). Refer to Chapters 25 and 26 for more depth on these topics.

Estimate the Hardware Requirements. This activity normally starts with some initial estimation before initial deployment environment design, followed by an iterative process of refinement as architecture and design progress. The resources you need to estimate include processing power, memory, disk space, and I/O bandwidth for each processing node.

Conduct a Technical Evaluation. In order to design and estimate the deployment environment, you may need to perform a number of technical evaluation exercises such as prototype element development, benchmarks, and compatibility tests. For example, you may wish to create a representative prototype system to ensure that your application server, object persistence library, and database all work smoothly together and to check the transaction throughput you can achieve.

To ensure a representative test, identify the key attributes of your application (size, type of processing, and so on) and make sure you include all of this in your technical evaluation. Involve experts in the test to gain the benefit of their experience and ensure that you do not overlook anything important.

Obtaining time and resources for technical evaluation is often a problem. We have found that arguing for evaluation resources in terms of risk management is often the most effective way to deal with this.

Assess the Constraints. It is rare for architects to be left to define a Deployment view without any external constraints. The constraints you encounter may be formal standards, informal guidelines, or simply implicit constraints that you know exist. However the constraints are expressed, you need to review your proposed deployment environment design to ensure that they are met.

Network Models

In the interests of simplicity, the runtime platform description does not usually define the network in any detail. If the underlying network is complex, it is usually described in a separate network model.

In our experience, the network is usually designed and implemented by networking specialists rather than the software architect. However, it is important that you provide the networking specialists with a clear specification of the capabilities of the network you are expecting. This description must indicate which nodes need to be connected, any specific network services that you require (such as firewalls or compression), and the bandwidth requirements and quality properties required from each part of the network. This model is normally a logical or service-based view of what you require of the network, rather than a physical view that specifies its individual elements. In the case of software product development, such a model is a valuable specification for customers planning the deployment of your software.

The primary elements of a network model are as follows.

- *Processing nodes*: The processing nodes represent your system elements that use the network to transport data. This set of nodes should match the set from the runtime platform model, but here they are abstracted to simple elements with network interfaces.

- *Network nodes*: Additional network nodes can be added to represent network services that you expect to be available (such as firewall security, load balancing, or encryption).
- *Network connections*: The network connections are the links between the network and processing nodes. They are elaborated to include the characteristics of the service you expect the link to provide (most typically bandwidth and latency, but perhaps also quality of service, reliability, or other network qualities).

This description is typically represented as an annotated network diagram, which is really a network-oriented specialization of the runtime environment diagram. In cases where your network requirements are very simple, you can describe the network sufficiently by elaborating the runtime platform model, rather than creating a separate network model. However, given the critical dependency that most of today's systems have on the underlying network, a separate network model is a useful tool to focus attention on this aspect of the system.

Figure 21–2 shows a simple example of a network model for the runtime platform we depicted earlier in Figure 21–1. This diagram would be augmented with textual descriptions for each of the major elements.

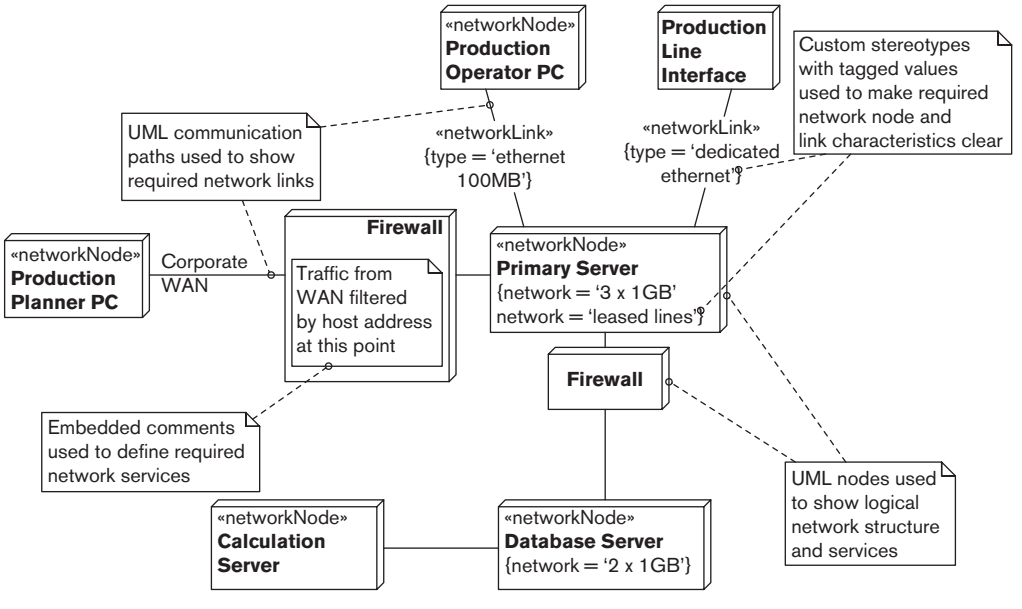

FIGURE 21–2 EXAMPLE OF A NETWORK MODEL

NOTATION Common notations used for capturing the network model include the use of UML and traditional boxes-and-lines diagrams.

- *UML deployment diagram*: UML's deployment diagram is a useful base notation for a network model. However, as with the runtime platform description, you will probably need to annotate it with stereotypes, tagged values, and comments in order to make your intentions clear.
- *Boxes-and-lines diagram*: For reasons similar to those discussed earlier, the network model is often drawn using an informal notation.

ACTIVITIES

Design the Network. The network design is typically handled separately from that of the computer hardware because different specialists are involved. From your point of view, this is a process of sketching what you need from the network (in terms of connections, capacity, quality of service, and security). This results in what is effectively a logical rather than a physical network design, which then becomes a specification for a specialist network designer to take further.

Estimate the Capacity and Latency. Part of designing your logical network is to estimate the capacity and latency that you are expecting between each node. Precision isn't that important at this stage, but a realistic estimation of the magnitude of the traffic to be carried and expected round-trip time is important. You can estimate the capacity figures by combining peak transaction throughput and a rough approximation of the size of messages required to carry the transaction's information. The latency is normally estimated using a combination of standard metrics for the type of network in use (combined with the distance between nodes) and some measurement of the existing network. Both results are normally combined with judicious scaling factors to allow for inevitable overheads and prediction inaccuracies.

Technology Dependency Models

In some cases, you can manage the dependencies within your development or test environment by bundling your software and its dependencies into one deployment unit. However, in many cases this simply won't be possible for reasons such as efficiency, cost, licensing, or flexibility. If this is the case, you need to manage the dependencies in your deployment environment.

Technology dependencies are usually captured on a node-by-node basis in simple tabular form. The software dependencies are typically derived from the Development view, where you define the environment used by the software developers. You can also derive hardware dependencies from test or development environments, but in many cases you have to rely on manufacturer specifications and some judicious testing to confirm them.

TABLE 21–2 SOFTWARE DEPENDENCIES FOR THE PRIMARY SERVER NODE

Component	Requires
Data Access Service	HP-UX 64-bit 11.23 + patch bundle B.11.23.0703 HP aCC C++ runtime A.03.73
Data Capture Service	HP-UX 64-bit 11.23 + patch bundle B.11.23.0703
	HP aCC C++ runtime A.03.73
	Oracle OCI libraries 11.1.0.7
HP aCC C++ Compiler & Runtime	HP patch PHSS_35102
	HP patch PHSS_35103
Oracle OCI 11.1.0.7	HP-UX optional package X11MotifDevKit.MOTIF21
	HP-UX patch PHSS_37958

EXAMPLE Table 21–2 shows an example of software dependencies for the Primary Server node in our example from Figure 21–1.

From this table it is possible to see that this node in the system needs a particular version of HP-UX with a patch bundle, a couple of specific operating system patches, a set of C++ libraries, and one optional module installed, as well as a particular version of an Oracle product.

In simple cases, it may be possible to use the Development view contents rather than list dependencies in this view. However, in more complex cases, it is unlikely that the Development view contains the detail required to fully define the software dependencies for each node type in the system.

NOTATION A technology dependency model is often best captured by using a simple text-based approach, but it can sometimes benefit from the use of some simple graphical notations.

- *Graphical notations*: One way to capture software dependencies is to extend your runtime platform model to add an indication of the software stack required on each machine to support the system elements executing there. In simple cases, this can be a useful elaboration of the runtime platform model. The problem with this is that complete and accurate software dependency stacks on each node can clutter the runtime platform model to the point where it is no longer usable—in this case, you should record this information separately.

- *Text and tables*: Dependencies are almost always captured as simple text tables. It is important to capture the exact requirements for third-party software (e.g., detailed version numbers, option names, and patch levels).

ACTIVITIES

Analyze the Runtime Dependency. This is usually a manual exercise to work through your system elements, identifying the dependencies they have and then repeating this process for each of the third-party elements. You normally derive the runtime dependencies from documentation supplied with each piece of third-party technology you are using and your own build and test environment requirements. With this done, you can clearly define the third-party elements you need for each processing node in the system.

Conduct a Technical Evaluation. In order to correctly document dependencies, you may need to do some prototyping or technical investigation.

Intermodel Relationships

For complex systems, a Deployment view contains two or three closely related models rather than a single model. We have found that the three models described earlier tend to be used by different stakeholders at different times. People in the groups responsible for deployment refer to the runtime platform model early in the project, a specialist networking group consults the network model, and system administrators use the technology dependency model during more detailed installation planning close to deployment. For this reason, we've found it valuable to present each separately.

A good way to think about these models is as a set of informal layers, with the core of the view being the runtime platform model. You can think of the network model as a lower layer supporting the runtime platform by defining details of the network required. The technology dependency model can be thought of as a more detailed layer on top of the runtime platform that defines the software and hardware installation requirements on each machine in the deployment environment.

In an ideal world, a software architecture tool would allow you to create a single model for yourself and then extract different aspects of it automatically as required. However, we aren't aware of the existence of any such tool today, and so you'll probably have to work with separate models for the foreseeable future.

Figure 21–3 illustrates this relationship between the models within the Deployment view. The runtime platform model is the core of the view, with the network model providing more details of the network underpinning the system and the technology dependency model providing more detail about the hardware and software installed on each node to provide the runtime environment.

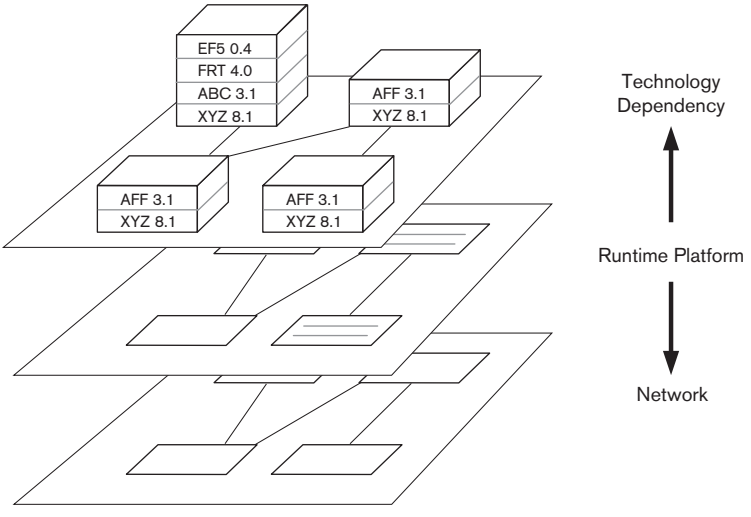

FIGURE 21–3 MODELS IN DEPLOYMENT VIEW

PROBLEMS AND PITFALLS

Unclear or Inaccurate Dependencies

Large-scale computing technology tends to be fairly complex, and it often has many explicit and implicit dependencies on its runtime environment that will cause problems if not satisfied. This difficulty is compounded by the fact that most of these dependencies are invisible and can't be checked easily—you may not discover that you have the wrong version of a utility library until your database server fails to start.

"You need Oracle and Linux" or "It uses Intel hardware" are pretty common dependency statements. For all but the smallest systems, these are too vague to allow safe deployment of the system. You should specify which versions are required, whether any optional parts of the products are needed, whether any patches are required, and so on. With the complexity and flexibility of enterprise software products today, you need to be very clear about what is required and what isn't.

RISK REDUCTION

- Capture clear, accurate, detailed dependencies between your software elements and the runtime environment in the Deployment view.
- Capture dependencies between third-party software and the runtime environment it needs.

- Perform compatibility testing to ensure that the dependencies between the elements are correct.
- Use existing, proven combinations of technologies where the dependencies are well understood.

Unproven Technology

Everyone wants to use the newest and coolest technology—and understandably so, as it often has the potential to bring great benefits. However, because its characteristics are unknown, using technology with which you don't have experience brings significant risks: functional shortcomings, for example, or inadequate performance, availability, or security.

RISK REDUCTION

- As much as possible, use existing software and hardware that you can test before committing to its use.
- When you must use new technology (or technology new to you), get advice from people who have used the technology before, or if this is not possible, test it thoroughly.
- Create realistic, practical prototypes and benchmarks to make sure that technologies work as advertised.
- Perform compatibility testing to ensure that new technologies work well with existing technologies.

Unsuitable or Missing Service-Level Agreements

The runtime environment for your system is usually provided by other people, whether they are a separate part of your organization or are a completely separate organization. When providing services such as hardware, data storage, networking, and so on, it is usual to specify an SLA to define the service that you can expect from the provider. This will cover aspects of the service such as cost, expected performance and reliability, recovery time guarantees in case of failure, data backup service, and so on. You need to check the SLAs carefully to make sure that the guarantees that they provide will allow you to meet the goals of your system.

RISK REDUCTION

- Obtain a reliable SLA for the runtime environment elements that are provided by third parties (and estimate your own SLA if providing elements yourself).

- Attempt to test the guarantees that the SLAs provide.
- Analyze the SLAs to understand how they combine and the implications of their combination.

Lack of Specialist Technical Knowledge

Designing a large information system is a complex undertaking that requires a huge amount of specialist knowledge about many different subjects. No one person can possibly be an expert on all of the technologies you may need to use. This is why we use teams of people to develop systems and why some people specialize in particular technologies, allowing them to advise others.

Given the number of technologies used in many systems, it can be difficult to assemble a project team with expertise in all of the technologies required. This can lead to a situation where you end up relying on vendor claims for products rather than proven knowledge and experience.

RISK REDUCTION

- Bring specialist knowledge into your team so that you have mastery of all of the key technologies you need to use to deliver your system. If you don't need the knowledge full-time, hire trusted and experienced part-time experts.
- Obtain external expert review of your architecture to validate your assumptions and decisions.
- Obtain binding contractual commitments from your technology suppliers when possible.

Late Consideration of the Deployment Environment

The deployment environment is where your system hits reality. We've seen problems in some projects when the system is designed from a purely software-oriented perspective and the deployment environment is considered only when the software is complete. Remember that an inappropriate deployment environment can make an otherwise good system totally unusable.

The deployment environment also often affects how the software is designed and implemented, and this can be expensive to change. For example, if plans change and you need to use a group of small machines rather than a single large machine to host your server elements, this could have a significant impact on the architecture of your server software, a change that would be expensive to make late in a project.

RISK REDUCTION

- Design your deployment environment as part of architecture definition rather than as part of a separate exercise performed after the system has been developed.
- Obtain external expert review of your architecture to get early feedback before you spend too much time or money.

Ignoring Intersite Complexities

Many systems are deployed to an environment involving more than one physical site, and this is becoming ever more prevalent as organizations move to use third-party hosting providers and cloud computing environments to augment their own data centers. Even when the entire environment is hosted in-house, concerns such as resiliency, disaster recovery, geographical location of the business, and data movement restrictions can result in systems being hosted across a number of geographically distant sites.

If you do have a multisite deployment environment, it is important to consider the impact of this quite early in your architectural design work as it can have a major impact on the quality properties of the system, particularly its security, performance, and scalability. Network latency between sites is the most obvious problem (meaning that interelement interactions across these links need to be considered carefully), but the need to keep the system secure across multiple sites and the possible scalability limitations of needing to synchronize information across sites are some of the other areas of concern that need to be addressed.

RISK REDUCTION

- Understand any requirements for multisite deployment as early as possible in your design work, and if it looks likely that multisite deployment is going to be required, consider its impact on all of your system qualities.
- Work with your infrastructure teams to understand the implications of distributing your system to multiple sites and the restrictions that the infrastructure may impose on this.
- Try to test various representative aspects of multisite deployment as soon as you can so that you are confident that you understand its implications.

Inappropriate Headroom Provision

Headroom is additional capacity (CPU power, memory, disk space, network bandwidth, and so on) that you include in your hardware specifications to accommodate spikes in demand or future growth in volumes. You usually add

some headroom to your sizing estimates so that your system can cope with additional demand without incurring hardware upgrade costs.

Specifying headroom involves a delicate balance between optimism about future growth and spending restraint. If you get it wrong, you end up deploying either expensive hardware that is insufficiently used or a system that fails to meet its performance requirements. We discuss this further in Chapter 26.

RISK REDUCTION

- Make sure your hardware specifications include an appropriate amount of headroom. Refer to the Performance and Scalability perspective, discussed in Chapter 26, for a discussion of how to model this effectively.

Not Specifying a Disaster Recovery Environment

Disaster recovery is the means whereby systems can be kept operational in the event of a significant failure, such as loss of electric power, widespread storage failure, or a natural disaster such as fire or flood.

Many disaster recovery strategies require the deployment of a separate operational environment at a different location (for example, a standby or alternate data center). To keep costs down, the standby environment may have a lower specification than the production environment. In any case, as it is usually the responsibility of a development project to specify, implement, and pay for the standby hardware, this must form part of your architectural description.

We discuss this further in Chapter 27.

RISK REDUCTION

- Make sure your Deployment view includes a specification of any disaster recovery hardware required.

CHECKLIST

- Have you mapped all of the system's functional elements to a type of element in your runtime platform? Have you mapped them to specific hardware devices if appropriate?
- Is the role of each piece of your runtime platform fully understood? Is the specified hardware or service suitable for the role?
- Have you established detailed specifications for the system's hardware devices or the hosted services that you require? Do you know exactly how many of each device or how much of each service is required?
- Do you have service-level agreements for the elements of the runtime environment that are supplied by third parties? Are the guarantees in the

agreements suitable for your system? Can you test whether the guarantees are credible or not?

- Have you identified all required third-party software and documented all the dependencies between system elements and third-party software?
- Are the network topology and services required by the system understood and documented?
- Have you estimated and validated the required network capacity? Can the proposed network topology be built to support this capacity?
- Have network specialists validated that the required network can be built?
- Have you performed compatibility testing when evaluating your architectural options to ensure that the elements of the proposed deployment environment can be combined as desired?
- Have you used enough prototypes, benchmarks, and other practical tests when evaluating your architectural options to validate the critical aspects of the proposed deployment environment?
- Can you create a realistic test environment that is representative of the proposed deployment environment?
- Are you confident that the deployment environment will work as designed? Have you obtained external review to validate this opinion?
- Are the assessors satisfied that the deployment environment meets their requirements in terms of standards, risks, and costs?
- Have you checked that the physical constraints (such as floor space, power, cooling, and so on) implied by your required deployment environment can be met?
- Do your hardware and service specifications include an appropriate amount of headroom?
- Does your Deployment view include a specification of a disaster recovery environment, if required?

FURTHER READING

A great deal of literature describes specific deployment technologies; unfortunately, little of it discusses how to design an entire realistic and reliable system deployment environment. Some other software architecture books [CLEM10, GARL03, HOFM00] contain useful explanations of how to document deployment views. Dyson and Longshaw's book on designing large-scale applications [DYSO04] includes a number of patterns relating to the Deployment view. Some of the further reading we recommend in the perspectives in Part IV also contains principles and patterns relevant to the design of a deployment environment.

22

The Operational Viewpoint

Definition	Describes how the system will be operated, administered, and supported when it is running in its production environment
Concerns	Installation and upgrade, functional migration, data migration, operational monitoring and control, alerting, configuration management, performance monitoring, support, backup and restore, and operation in third-party environments
Models	Installation models, migration models, configuration management models, administration models, and support models
Problems and Pitfalls	Lack of engagement with the operational staff, lack of backout planning, lack of migration planning, insufficient migration window, missing management tools, production environment constraints, lack of integration into the production environment, inadequate backup models, and unsuitable alerting
Stakeholders	System administrators, production engineers, developers, testers, communicators, and assessors
Applicability	Any system being deployed into a complex or critical operational environment

Considerable effort is spent defining the architecture and design of today's large systems. However, it is rare in our experience to find a system for which comparable consideration is given to how the system will be controlled, managed, and monitored. The aim of the Operational viewpoint is to identify a system-wide strategy for addressing the operational concerns of the system's stakeholders and to identify solutions that address these.

For a large information system, the Operational view focuses on concerns that help ensure that the system is a reliable and effective part of the commissioning

enterprise's information technology environment, whether it is hosted within the organization or externally by a third-party provider. For a product development project, the Operational view is more generic and illustrates the *types* of operational concerns that customers of the product are likely to encounter, rather than the concerns of a specific site. This view also identifies the solutions to be applied throughout the product implementation to resolve these concerns.

Of all of the views you create for your AD, the Operational view is often the one that is least well defined and needs the most refinement and elaboration during the system's construction. This is simply because many of the details that the Operational view considers are not fully defined until design and construction are well under way. However, considering the issues described in this chapter as early as possible will save you a lot of time and effort later.

CONCERNS

Installation and Upgrade

Installation and upgrade can range from the development team installing and configuring software elements on customer-specific hardware; to the ultimate users of the system obtaining hardware and software from a number of sources and performing installation, integration, and configuration themselves; to allocating resources in a public cloud computing environment and uploading software to it. In many organizations, software installation is performed by a separate team whose members are specially authorized to make changes to the production environment. This team may well expect that the installation process has been carefully preplanned and is largely automated.

The other major area of variability is whether this is a pure installation or whether a previous version of your system is already installed, making the installation of the current version actually an upgrade. Upgrade can be significantly more complex than installation, due to the need to respect existing data, configuration settings, the state of running elements, and so on, and in some cases to keep the system in operation during the upgrade. However, the use of iterative development approaches means that upgrade, rather than installation, is the norm, so you need to master it.

As an architectural concern, installation is less about the design of detailed procedures and plans and more about ensuring that the system can be installed or upgraded in a way that is acceptable to stakeholders. This involves working with technical specialists to understand the installation processes, software developers to ensure that their elements can be easily and reliably installed, and production engineers to assure a practical, low-risk installation approach.

Functional Migration

Functional migration is the process of replacing existing capabilities with the ones provided by your system. This usually means migrating users of an older system to use your new system. Your migration approach may comprise one or more of the following:

- A *big bang* where the migration occurs in a single step at a single point in time (often over a weekend)
- A *parallel run* where new and old versions of a system are used side by side until confidence in the new system is high enough to allow switching off the old one
- A *staged migration* where parts of a process or an organization are moved to a new system, one by one, to manage the risk and cost of the migration activity

Like many architectural concerns, migration is centered on two issues—*risk* and *cost*. The big bang approach, for example, can be the cheapest because it requires no replication of resources, but it can be extremely risky because there is no easy recovery route if the migration goes wrong. Other approaches can be much more expensive (because they require duplication of resources and the implementation of costly processes to ensure that systems run together in lockstep) but reduce risk.

Data Migration

Most if not all system development involves some element of data migration—that is, loading data from existing systems into the new one(s). A goal of a data migration exercise is almost always to automate as much as possible, particularly when large volumes of data are involved. When migrated data is very old, of variable quality, or poorly modeled, data migration may be extremely complex. If you need to migrate data between geographical locations (for example, into a data center in a different region of the world, or into an externally hosted location), additional security and performance concerns can complicate things further.

Data migration software is typically viewed as utility software with a limited life, rather than as a system requiring long-term support. This does not mean that it is of any lesser quality, but it may consist of a collection of automated software, semiautomated procedures, and manual intervention to deal with exceptions (such as missing data or data in unexpected formats). This also adds to the complexity of the process.

Nowadays, systems that manage hundreds of gigabytes or terabytes of data are not uncommon, and this presents its own migration challenges.

Massive data stores are far more likely to include data that does not conform to business rules and therefore requires exceptional processing (possibly manual intervention). It can take days or weeks to extract data from or load data into massive data stores, and it is important that you do not underestimate the time required to reorganize databases, create indexes, and so on.

Fortunately, you can also make use of a wide range of Extraction, Transformation, and Load (ETL) tools that will help you automate this process. Many ETL tools allow you to define transformation rules visually and provide facilities for accessing a wide range of different physical formats, performing standard transformations, and monitoring and analyzing the results. Database replication facilities can also be used for data migration and are useful in situations where you need to keep a number of databases synchronized for a period of time.

Another frequently overlooked problem occurs when you are migrating data from a live system that is continuing to be updated while you migrate from it, as discussed in this example.

EXAMPLE A government tax office has a very large database of taxpayers that it is migrating into a new system. The database is updated through end-user screens in tax offices throughout the country.

The architect predicts that extracting all of the data from the database will take between three and five days. The data must be sorted, which will take another day, and will then be loaded into the new system, which will take ten days. Finally, indexes must be created on the new system, which will take another day. The overall elapsed time to migrate is over two weeks, during which time the original system is estimated to have received 100,000 updates, as shown in Figure 22–1.

It will not be possible to halt the country's tax-collection activities for two weeks while the data is migrated. Special code therefore has to be written to capture these updates as they occur and to apply them to the new system once the bulk of the data has been migrated into it, so that the extract is complete.

In short, data migration may be a significant piece of work in its own right, and you should manage it the same way as any other development project, with requirements, design, build and test, and acceptance—and, of course, architecture. Many of the architectural principles described in this book apply equally well to such a migration subproject, although the success criteria are different. In a migration project, it is the successfully migrated data that has to be accepted, rather than the migration software, which will be discarded once migration is complete.

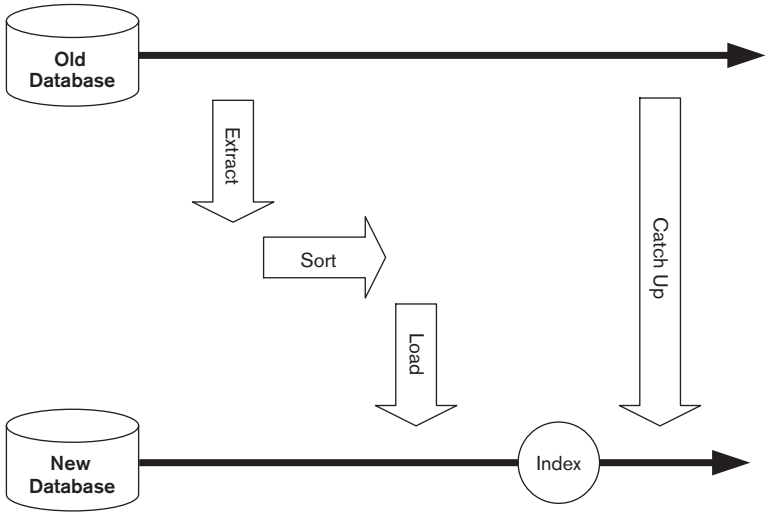

FIGURE 22–1 DATA MIGRATION FROM A LIVE SYSTEM

Operational Monitoring and Control

Once a system is running in its production environment, it will require some amount of routine monitoring, to ensure that it is working correctly, and some routine control operations, to keep it working that way (startup, shutdown, transaction resubmission, and so on).

Some systems need little monitoring or control—for example, a file server that needs only direct operational control when it fails or fills up. Others may need quite a lot—for example, a large financial reconciliation system that accepts data feeds from a variety of sources and may need routine monitoring and control to identify and rectify communication link and data reconciliation failures.

The amount of monitoring and control needed depends on the likely number and variety of unexpected operational conditions the system is likely to encounter in production. However, the development and integration of monitoring and control facilities can be a major effort in itself, so you may have to balance stakeholder needs in this area against cost and time. You also need to consider the system's deployment environment to make sure that the solutions you identify are appropriate.

Alerting

An alert is a notification from the system that an event has occurred, usually that something has gone wrong that requires human intervention in order to

fix. This may be a technical alert, such as the system is unable to connect to a database server, or a business alert, such as bad data has been received on an automated input. Some significant nonfailure events, such as startup or shutdown of a service, also should be alerted (for information rather than action).

Unlike operational monitoring (in which systems are largely passive), alerting is an active system function. The system sends alerts to a central console or alert management tool, where they are displayed to support staff so they can take appropriate action. For example, a server may need to be restarted or reset, a batch job may need to be resubmitted, or in some cases the alert may need to be handed over to development teams for diagnosis and repair.

Many large organizations have corporate standards for alerting that must be followed. These define things such as which events must be alerted, the information that should be included in the alert, and where the alert should go. They also often contain advice and guidance to avoid alert flooding (see the Pitfalls section later in this chapter). If your system is deployed to a third-party hosting environment, the hosting provider will almost certainly provide its own proprietary mechanisms for raising and monitoring alerts, and you will need to consider how to use and integrate into them.

Configuration Management

Many of the elements that make up your deployment environment will have their own configuration parameters. Databases, operating systems, middleware products, and of course your own software elements may all require detailed, specific configuration for the system to operate correctly. You may also need to make coordinated sets of changes to these configurations on a regular basis (the canonical example being a switch from online to batch mode and back again every 24 hours). Managing a number of separate element configurations can rapidly become complex enough to be a major source of operational risk for the system.

The discipline of configuration management aims to address this problem. Configuration management encompasses the processes and technologies to group, modify, and track element configuration parameters in a reliable and predictable manner.

The process of operational configuration management tends to be a fairly specialized job, handled by the system administration and production engineering groups that run the production systems. From the architectural perspective, addressing this concern involves understanding the operational configuration your system requires and ensuring that it is possible to achieve it in a way that will be accepted by the interested stakeholders.

Performance Monitoring

The process of understanding and improving system performance is known as *performance engineering*, which we discuss in Chapter 26. However, the basis of all performance engineering work is *measurement*, so performance monitoring is an important concern for most systems. Your system needs to be able to capture, present, and store accurate quantitative performance information.

Production system administrators are often the first people who need to recognize and respond to a performance problem. You need to involve them in this process as early as possible to make sure they can work with the proposed solution.

In Chapter 26, we discuss in more detail the kinds of metrics needed for performance engineering and how you can capture and report them.

Support

End users, support staff, and maintainers have an interest in the type and level of support needed, who will provide that support, and the channels through which it will be delivered. As well as the system itself, support may be needed for the associated hardware infrastructure (computers, printers, and the network).

Backup and Restore

As we saw in our description of the Information viewpoint in Chapter 18, data is an extremely valuable asset to any organization and should be protected and "insured" the same way that its other assets are. Processes to do this should be carefully designed, built, and executed and should also be regularly tested to ensure that they are still working correctly.

EXAMPLE One of us visited a trade organization many years ago that ran its membership database on a stand-alone UNIX system. The system administrator faithfully backed up the database to tape every night, but unfortunately, because the output of the process was not captured to a log file, nobody realized that the tape drive was broken and no data was being written. Only when the inevitable happened and a disk failed did the soon-to-be-unemployed system administrator realize that he had a shelf full of blank tapes. The organization had to re-create its membership database from paper records, which was a slow, painful, and costly process.

You shouldn't forget the restore side of the equation, either. At a minimum, restoring data should leave it in a transactionally consistent state (i.e., with all

updates entirely committed to the restored database or not recovered at all). You will need to consider the amount of data lost as a result of the restore; at a minimum, this will be any transactions that are active at the time of failure but may include a lot more, particularly if backups can be done only when the system is offline.

If your data is distributed, this problem becomes much harder to solve. Although a failure in any one program will affect only that program's data, the data may then become inconsistent with the rest of the system; you need to develop strategies to deal with this. Typically, the solution involves recovering or re-creating the lost data, either manually or (preferably) automatically. In some cases, it may be more appropriate to revert the rest of the system to the older state.

A significant complication for backup and restore planning is the fact that, in many situations, transactional consistency must extend across a system's entire distributed data set, as shown in the next example.

EXAMPLE A university maintains academic records for all of its students in a number of databases. The main database stores results for each exam taken by each student, and a consolidated database turns these into an overall score for each student based on exam success, as shown in Figure 22–2.

A database corruption means that the Exam Results Database has to be restored from its latest clean backup, which is almost three months old, and the results for the last three months rekeyed into it. Although the Student Scores Database is unaffected by the corruption, special actions must be taken to prevent this manually recovered data from filtering into the Student Scores Database and corrupting its data. As a result, it could take several weeks to repair the damage due to the corruption of one database.

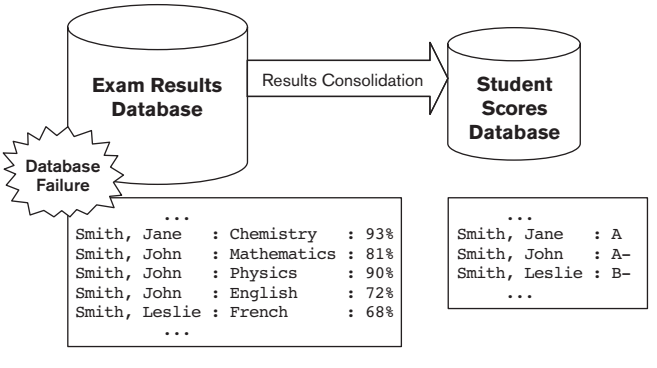

FIGURE 22–2 EXAMPLE OF BACKUP AND RECOVERY

If your system contains data distributed over a number of data stores, you must ensure that your Operational view takes this into account when considering backup and recovery.

Operation in Third-Party Environments

An increasingly popular option for application hosting is to use external hosting facilities such as external server hosting or cloud computing environments. External hosting can be a simpler, more flexible, and more cost-effective option than traditional internal hosting for some types of organizations and applications, and it seems to be a trend that is likely to increase.

If external hosting is a likely option for some or all of your application, it is important to start considering how you would operate your system in such an environment well before you need to do so. Any external hosting option will bring operational complications, so it is important to understand the particular facilities and constraints of the service to which your application may be deployed.

Some of the aspects of the operational environment that can be affected by the use of third-party hosting environments include the need to integrate into whatever monitoring, alerting, and management tools the service provides; the need to migrate data into the environment in order to support processing there; the need to work with the provider's support channels for problem escalation; and the need to understand and test the backup and restore facilities provided. You are unlikely to be able to gain physical access to your servers in any circumstances—indeed, you may not even know where they are located—and therefore need to be able to perform all operational actions remotely.

Stakeholder Concerns

Typical stakeholder concerns for the Operational viewpoint include those listed in Table 22–1.

TABLE 22–1 STAKEHOLDER CONCERNS FOR THE OPERATIONAL VIEWPOINT

Stakeholder Class	Concerns
Assessors	Functional migration, data migration, support, and operation in third-party environments
Communicators	Installation and upgrade, functional migration, operational monitoring and control, and operation in third-party environments
Developers	Operational monitoring and control, performance monitoring, and operation in third-party environments

Continued on next page

TABLE 22–1 STAKEHOLDER CONCERNS FOR THE OPERATIONAL VIEWPOINT *(CONTINUED)*

Stakeholder Class	Concerns
Production engineers	Installation and upgrade, operational monitoring and control, configuration management, performance monitoring, and operation in third-party environments
Support staff	Functional migration, data migration, alerting, support, and operation in third-party environments
System administrators	All concerns
Testers	Installation and upgrade, functional migration, data migration, monitoring and control, performance monitoring, and operation in third-party environments
Users	Support

MODELS

The Operational view consists of models that illustrate how the system will be put into production and kept running effectively once it is there.

Bear in mind that for most enterprise systems, each of these models can be quite large and involved. When this is the case, it is sensible to summarize the model in the AD and reference a fuller model in another document, to avoid making the AD too large and unwieldy.

Installation Models

Moving a system from its development environment to its production environment is a critical part of the system's lifecycle. Your AD needs to demonstrate that it is possible for a system built using this architecture to be installed (and upgraded) in a practical way.

The installation model should discuss installation and/or upgrade as needed for your system. This model needs to help the reader understand:

- What needs to be installed or upgraded to move the system into production
- What dependencies exist between the various groups of items to be installed and upgraded
- What constraints exist on the process to perform the installation and/or upgrade for the system
- What would need to be done to abandon and undo the installation and/or upgrade if something goes seriously wrong

The AD doesn't need to include a complete installation and upgrade plan—that information goes into a different document produced when needed. Instead, the installation model provides your view of the requirements and constraints the architecture imposes on installation and upgrade. The installation model in your initial AD is likely to contain only an overview of your installation strategy (because the details of what needs to be installed aren't fully known at that time), but you will be able to elaborate and refine this model as construction of the system progresses.

NOTATION The best notation to use for an installation model really depends on the situation and what the primary stakeholders (the system administrators) are familiar with. In our experience, an approach using text and tables is often the best way to communicate this information.

Simple lists work well for laying out and defining the elements of the installation problem. In simple cases, cross-reference tables can describe dependencies, while more involved dependencies are usually effectively addressed with the use of dependency diagrams.

ACTIVITIES

Identify the Installation Groups. Start by considering what elements of your architecture need to be installed and/or upgraded, and identify groups of them that can be handled together. For each group, define which elements it contains and the approach that will be used to install or upgrade that group.

Identify Any Dependencies. Technical dependencies often exist between different parts of a complex system during installation, so that the installation process has to proceed in a specific order. Identify the dependencies that exist between your installation groups to reveal these constraints.

Identify Any Constraints. Consider the overall installation process and the different ways it could be achieved. Other than the ordering dependencies you considered in the previous activity, does your architecture or deployment environment impose any further constraints on the process? (For example, do you need to start one element after it is installed so it can generate code or data needed to install the next one? Do you need to follow any particular environment-specific procedure when installing particular elements of the system? Do you need to restart one of the machines during installation?)

Design the Backout Approach. Consider what would need to be done to undo any of the installation tasks you have identified. In particular, identify anything that would be complex or time-consuming to undo.

EXAMPLE This example shows an installation model for a rental-tracking system, based on the results of the activities just described.

Installation Groups

- *Windows Desktop Client*: contains all of the software in the `WIN-CLIENT` component. Installation shall be via InstallShield automatic installer, remotely executed via the management tool.
- *Database Schema*: contains all DBMS schema definitions and data abstraction stored procedures. To be packaged as simple SQL scripts and installed using a custom-written Perl script.
- *Web Interface*: contains the server-resident user interface components (the `WEBINTERFACE` component). Installation will be by manual administrative action, copying files into IIS directories according to written instructions.
- *Rental-Tracking Service*: contains the .NET assemblies that implement the services called by the Web and Windows interfaces (the `RENTALTRACKER` component). Installation will be by manual administrative action, copying files into IIS directories according to written instructions.
- *Reporting Engine*: contains the .NET assemblies that implement the summary reporting engine. Installation will be by manual administrative action, copying files into IIS directories according to written instructions.

Dependencies

- Windows Desktop Client, Web Interface, Rental-Tracking Service, and Reporting Engine depend on Database Schema.
- Windows Desktop Client and Web Interface depend on Rental-Tracking Service.
- Web Interface depends on Reporting Engine.

Constraints

- *Windows Desktop Client*: A restart of the client machine will be required during this installation process.

Backout Strategy

This is the first release of the software, so backout is reasonably straightforward and consists simply of uninstallation. For each installation group, the following action will be required.

- *Windows Desktop Client*: Run the installer with an `uninstall` flag.
- *Database Schema*: A custom Perl script will be supplied to remove all objects created during the installation.
- *Web Interface*: Manual administrative action will be required. The supplied instructions will list the files to be removed.
- *Rental-Tracking Service*: Manual administrative action will be required. The supplied instructions will list the files to be removed.
- *Reporting Engine*: Manual administrative action will be required. The supplied instructions will list the files to be removed.

Migration Models

If a migration process is required, the migration model needs to illustrate the strategy that will be used. Again, a complete plan is not called for in the AD, but rather a succinct definition of the strategies to be employed. This model should allow its reader to understand:

- What overall strategies can (or will) be employed to migrate information and users to the system
- How the new system will be populated with information from the existing environment
- How information in the new and old environments will be kept synchronized (if required)
- How operation could revert to the old system if serious problems emerge with the new one

As with the installation model, the migration model should focus on the requirements and constraints that the current architecture places on the detailed migration process that will be developed later.

NOTATION A migration model is usually documented by using text and tables because no suitable, widely accepted graphical notations are available. Some informal diagrams may help illustrate data migration and synchronization, and particularly complex data migration may require some form of data model to illustrate the transformations involved.

ACTIVITIES

Establish Possible Strategies. Assess your architecture and the existing system(s), and establish which migration strategies (i.e., big bang, parallel run, staged migration) are possible, how each would work, and the tradeoffs among them.

Define the Primary Strategy. In some situations, you will simply define the options and someone else will decide which approach to use (e.g., if you're developing a product and different customers want to migrate in different ways). In other cases, you will be tasked with the responsibility of defining which strategy best meets the needs of your stakeholders and making it happen. If migration is going to take more than a short period of time, you will need to consult with your stakeholders, particularly users, to define an approach that minimizes any business disruption.

Design the Data Migration Approach. Having identified a strategy to use, you need to decide how to populate the system with all of the information in the existing system(s). This doesn't mean you need to spend days mapping fields between databases, but it does mean you have to understand the problem well enough to choose an appropriate approach for the data migration and determine how long it is likely to take and what tasks and resources might be required.

Design the Information Synchronization Approach. Some situations call for information to be synchronized between the old system(s) and the new one. This is particularly the case when using the parallel run migration strategy because information in the old system(s) may continue to be updated after the new system goes live. If synchronization is required, it may be unidirectional (just into the new system) or bidirectional (information changes need to be migrated from new to old as well as old to new). Your task is to identify an overall approach that will allow the required degree of synchronization to be performed within the operational constraints of your environment.

Identify the Backout Strategy. Being able to back out to an existing system (if available) is an attractive risk-reduction option for live operation. The problem is that it isn't always clear how such a backout would work, or if it is even possible (e.g., reverse data migration may not be practical due to the design of the new system). You need to decide whether a backout strategy involving the old system(s) is required and, if so, how it could work.

Configuration Management Models

You may need to create a configuration management model if your system requires complex, regular reconfiguration (e.g., reconfiguring parts of the

system to handle different types of workloads according to a calendar-based schedule). This model must explain to its readers:

- The groups of configuration items in the system and how each is managed
- The dependencies that exist among the configuration groups
- The different configuration value sets required for routine system operation (and why each is required)
- How the different sets of configuration values will be applied to the system, taking into account the characteristics of the operational environment(s) that you are using

The aim is to create a model of the system configuration management aproach (rather than identifying lots of individual configuration values). This allows those responsible for system configuration management to understand the problem and plan their solution for it.

Like the installation model, this model is unlikely to be complete in your early AD but can be elaborated and refined as construction of the system progresses and the details of the configuration items required become known.

NOTATION This model is often quite simple and best documented by using text and tables. In more complex cases, it is usually best treated primarily as a data model, and a data-modeling notation, such as entity-relationship diagrams or UML, is a useful addition to the textual description.

ACTIVITIES

Identify the Configuration Groups. Consider all of the configuration values your system requires and break them into cohesive groups with as few intergroup dependencies as possible. This allows you to abstract the problem of managing the individual values to the level of managing large groups as a single unit (think of them as clumps of values). Name each group, explain its purpose, and explain how the configuration group would be managed (how values are defined, collected, applied, and so on).

Identify Any Configuration Group Dependencies. Having identified the groups of configuration information in your system, you can clearly identify and record any dependencies among them. For example, if changing the configuration of your database management system means reconfiguring the operating system, or adding more instances of one of your elements means changing the application server configuration, record these as intergroup dependencies. Identifying these dependencies allows you to start understanding the problem of reconfiguring your system in production.

Identify Configuration Value Sets. Consider your system during its routine operational lifecycle, and establish how many configurations your system will need during it. Define the characteristics of each value set, and identify the configuration groups that change between different configurations. For each set you identify, define its purpose and when it needs to be applied. This allows assessment of the operational impact of your architecture's configuration needs.

Design the Configuration Change Strategy. Once you have identified the configuration your system needs and the changes you need to make to it, design an approach to achieve this in your intended production environment, taking into account any constraints that it imposes. Again, rather than focusing on the minutiae of the administration process, you need to identify a practical overall approach that the production administrators of your system will accept.

EXAMPLE This example shows a configuration management model for the rental-tracking system we described earlier.

Configuration Groups

- *DBMS Parameters*: the SQL Server 2008 parameters that control the initialization, operation, and performance characteristics of the database. These are managed via SQL scripts, applied by database administrators.
- *IIS Parameters*: the IIS parameters that control the initialization, operation, and performance characteristics of the server. These are managed by using a set of PowerShell scripts that will be supplied with the system.
- *Reporting Engine Options*: the Reporting Engine parameters that control what is summarized and when summaries are computed. These are managed as a set of configuration files read by the component.

Configuration Dependencies

- When the IIS parameters are set to allow more connections, the DBMS parameters must be changed to allow for the possible increase in load.
- If the Reporting Engine Options are set for more aggressive summary activity, the DBMS parameters must be set to allow for an increased amount of data cache being required.

Configuration Sets

- *Standard*: normal configuration for planned system workload of up to 1,200 concurrent users with the Reporting Engine producing level 1 summary statistics every 6 hours.
- *High Volume*: configuration to be applied when high client volume is expected. Increases capacity to 2,000 concurrent users and switches off routine operation of the Reporting Engine.
- *Month End*: configuration to be applied during the last two days of the month, limiting concurrent usage to 800 users and allowing the Reporting Engine to run continually to produce complete summary statistics.

Configuration Change Strategy

The configuration sets will be applied as follows.

- The DBMS will be manually reconfigured first, by the database administrator running a single script that sets the parameters for the desired configuration set. (This could involve a DBMS restart.)
- Next, the Reporting Engine Options will be changed by altering the Engine's configuration file parameter and restarting it.
- Finally, the IIS configuration set will be applied by an administrator running the appropriate PowerShell script and restarting the IIS server.

Administration Models

When your system is running in its production environment, it will require some degree of administration to monitor it and keep it running smoothly. The administration model is the section of your AD where you define the operational requirements and constraints of your architecture and the facilities it provides for administrative users.

The administration model must define the following items.

- *Monitoring and control facilities*: In order to support your system's administrators, you may need to provide or use some monitoring and control facilities as part of your architecture. This may involve custom utilities and features and/or integration into one or more existing internal or third-party management environments. It can be as simple as a basic message log or as complex as a full-blown integration with a management or monitoring infrastructure. You need to clearly define the

facilities you are going to provide, use, or integrate into; how these address the problem; and whether any limitations in these facilities could constrain their applicability or usefulness.

- *Required routine procedures*: You should also review the architecture you have designed and identify any administrative work that needs to be performed on a regular basis or that may be required in exceptional circumstances. Depending on the system, this can be as basic as a weekly backup and a monthly health check, or it can be a set of complex procedures performed on a round-the-clock basis to keep a critical high-volume system running at peak efficiency. For each procedure, you need to define its purpose, when it is performed, who performs it, and what is involved in performing it. In most cases, you should cross-refer to the relevant monitoring and control facilities provided.
- *Likely error conditions*: Any complex system can suffer unexpected failures due to internal or external faults. From simple situations such as disks filling up to sudden failures of the underlying network causing a cascade of problems, many error conditions that occur can require administrative intervention to rectify them. Some of these conditions are independent of your architecture and are caused by underlying platform failures. The administrators of your system will probably already be experts at diagnosing and recovering from these failures. However, you cannot expect them to understand the possible error conditions that are unique to your architecture, and you need to explain these carefully to help administrators understand the conditions they may need to recover from. Your description should include when the condition can occur, how to recognize it (referencing relevant monitoring facilities provided), how to rectify it (referencing relevant control facilities provided), and possible further failures the condition could trigger.
- *Performance monitoring facilities*: A specialist subset of system monitoring is the ability to monitor the performance of the system. The difference between operational monitoring and performance monitoring tends to be how the data is used. Operational monitoring usually reports by exception and produces little or no output data when everything is going well. In contrast, performance monitoring facilities are usually designed so that performance information can be extracted and analyzed routinely to allow system performance to be tracked over time. We talk more about performance activities in general in Chapter 26. In an administration model, you need to explain the types of performance measures you will make available and how administrators or developers will extract and analyze the information when required.

An important point to note is the strong degree of cross-reference between the administrative facilities you define in this model and the common design

model in the Development view. In the Operational view, you define the facilities you will provide for the administrative stakeholders. The Development view needs to define the common processing required across all of the system's elements in order to actually achieve those facilities.

NOTATION The primary customers of the administration model are system administrators, who may not be software developers by training. We have found that the right notation for this model is nearly always text and tables augmented with a few informal diagrams where needed. Extensive use of more formal notation such as UML is less appropriate for this model.

ACTIVITIES

Identify the Routine Maintenance Required. Consider your system running in its production environment and create a list of the types of operational tasks that will need to be performed to keep the system running smoothly. For each task type, define who needs to perform it, when it needs to be performed, and how it should be performed.

Identify Your Likely Error Conditions. Analyze your architecture by considering its primary usage scenarios, and work out what is likely to go wrong during the operational lifecycle (elements failing, data stores filling up, systems running out of memory or other runtime resources). Make sure you think about the ones related to administration and maintenance as well as the ones that end users would be aware of—it is often harder to plan for failures during larger-scale administrative scenarios (such as data maintenance). Identify the classes of error conditions that can occur, what causes them, and how they can be rectified to get the system running again. You should also try to estimate the likely availability impact of the failure to ensure that you can recover the system in a time acceptable to your stakeholders. You may want to consider the error conditions that could occur if routine maintenance *isn't* performed, so that the importance of this maintenance is understood.

Specify Any Custom Utilities. Routine and exceptional procedures may require system-specific utilities to allow administrators to perform them efficiently. Such utilities can range from very simple database or operating system scripts to significant pieces of software in their own right. Consider whether any such utilities are required, and specify any you need.

Identify the Key Performance Scenarios. Some of your architectural usage scenarios will be much more important than others from a performance perspective (look for the scenarios that support time-critical business processes, involve a high workload, are executed very frequently, or are required by key stakeholders). Extract these scenarios from the overall system usage scenarios.

Identify the Performance Metrics. Consider the key performance scenarios and identify metrics that will allow you to measure the performance

achieved for each and to analyze where the system spends most of its time and resources. In order to abstract the problem, it may be more useful to identify classes of metrics rather than individual ones. Make sure that you record what each metric or class identified actually means and what it is used for.

Design the Monitoring Facilities. Having established the operational tasks and performance metrics required, you can design monitoring facilities to be used across the system to provide routine system monitoring and error condition recognition and to gather performance metrics from the system's elements. This design will be at the outline level, to be fleshed out later during the development increments of the lifecycle. However, at this stage you should provide enough detail to clarify what needs to be done in each system element to provide the administration facilities required.

EXAMPLE This example shows an administration model for the rental-tracking system.

Monitoring and Control

The monitoring and control facilities are as follows.

- *Server Message Logging*: All server components will write information, warning, and error messages to the Windows Event Log of the machine they are running on.
- *Client Message Logging*: The client software will log messages if an unexpected error is encountered. The log will be written to the hard disk of the client machine for later manual retrieval.
- *Startup and Shutdown*: No system-specific startup and shutdown facilities will be provided because the software will run in the context of the IIS and SQL Server servers, and their facilities are considered to be sufficient.

Operational Procedures

Routine operational procedures are as follows.

- *Backup*: Operational data in the SQL Server database will need to be backed up. This will involve backing up the transaction logs every 15 minutes and backing up the application's databases every day. Details of this procedure will be left to the database administrators.

- *Pruning of Summary Information*: The Reporting Engine does not remove the summary reporting information it creates. This information is left in place and is available to users of the Windows client interface. Database administrators will need to monitor the performance of the Reporting Engine and the management reporting aspects of the Windows client components and manually prune the summary information when its volume starts to impact performance. A written procedure will be supplied to explain how the pruning should be performed.

Error Conditions

The error conditions that administrators should be expected to handle are as follows.

- *Database Out of Log Space*: If transaction volume rises above a certain point, it is possible that the transaction log will fill. This will cause the system to suspend operation. Database administrators will need to recognize log space problems and manually back up the logs to free space. If this happens routinely, the backup interval for the transaction logs should be reduced.
- *Database Out of Data Space*: If the database runs out of data space, the system will stop operating. Again, database administrators will need to recognize this condition and either prune the summary information (see above) or add more data space to the system. A written estimate of the amount of space required for various volumes of workload will be provided.
- *IIS Failure*: If the IIS server fails, the system will completely fail, and Windows clients will lose contact with the server. Administrators need to recognize this condition and restart IIS. The system will recover automatically once IIS is restarted. The Windows clients will automatically reconnect once the server is available again.

Performance Monitoring

No application-specific performance monitoring facilities are planned. System performance monitoring should be achieved by using the following facilities.

- *SQL Server Counters*: The SQL Server 2008 product allows a wide range of performance counters to be collected and viewed via the Windows Server 2008's Reliability & Performance Monitor and the

SSMS Activity Monitor. These performance metrics should be used to assess the volume of workload on the database and the time taken for the application's transactions to complete.

- *IIS/ASP.NET Counters*: IIS Server and ASP.NET produce a wide range of performance counters to be collected via the Windows Server 2008's Reliability & Performance Monitor application. These counters should be used to assess the number of Web requests being serviced and how long it is taking to service them.
- *.NET Counters*: The .NET runtime allows a wide range of performance counters to be collected via the Windows Server 2008's Reliability & Performance Monitor application. These counters should be used to establish the amount of non-Web-request workload that the application is performing and how long it is taking to perform the operations.

Support Models

Once your system is running in production, at least some of the system's stakeholders are likely to need help using or operating it, and other parties will need to provide assistance to them. The support model should provide a clear abstraction of the support that will be provided, who will provide the support, and how problems can be escalated between parties when searching for a resolution. This means defining the following in your support model.

- *Groups needing support*: The model must clearly define the groups of stakeholders who will require support, the nature of the support they need, and the appropriate mechanisms for delivering that support.
- *Classes of incidents*: The model must also define what sorts of support incidents are likely to be encountered and what sort of response is reasonable to expect in each case. The definition of each class of incident should clearly state the characteristics of an incident in that class, typically in terms of operational, organizational, or financial impacts.
- *Support providers and responsibilities*: Each type of support incident needs to be handled by at least one support provider, who must accept responsibility for resolving the incident. The model should capture who the support providers are and their responsibilities for incident resolution.
- *Escalation process*: A serious incident often requires a number of different support providers to resolve the situation because it is too complex or specialized for a single provider to handle. Your model should define how

incidents are escalated between support providers and the responsibilities of each when this happens. This will help ensure that incident resolution does not stall because of confusion over responsibilities or a lack of expertise by a particular provider.

As with the other models for the Operational view, the focus of the support model should be to provide an overview of the support problem and a strategy for its solution rather than the definition of detailed procedures.

NOTATION This model needs to be understood by a number of different technical and nontechnical stakeholder groups. The majority of the model should normally be a text-and-tables definition of the support to be provided, with some flow diagrams (such as UML activity diagrams) where required to make the information flow and decision-making processes clear.

ACTIVITIES

Identify the Supported Groups. Identify the groups of stakeholders who will need support, the type of support they will need, and the possible avenues through which that support could be provided.

Identify the Support Providers. Decide who will be providing support to your stakeholders. For each provider (which will probably be an organization), define the support to be provided and how it is to be provided.

Identify Any Incidents Requiring Support. Consider the types of incidents that could trigger the need for assistance by each of your groups of supported stakeholders, and characterize each incident type by likely frequency and severity.

Map the Providers, Incidents, and Groups. Decide which support providers will resolve which incident types for which stakeholder groups, and ensure that each provider can offer suitable support.

Plan the Escalation. Consider your groups of support providers, and identify which of them may need to escalate problems to other internal or external support providers. Define the escalation paths that should be used between providers and the responsibilities of each provider when this happens.

EXAMPLE This example shows a support model for the rental-tracking system.

Supported Groups

- *Web Users*: People using the Web interface to book or manage their rentals may need support if there are problems with the site or if

they have difficulties using the Web interface. Few assumptions may be made about this group, and the primary support channel should be e-mail, with telephone backup.

- *Windows Users*: Internal users using the Windows client may need help with a range of problems including usage issues, system problems, and PC support. Their primary support channel is assumed to be telephone, although they may be prepared to receive support via e-mail.
- *Windows Administrators*: The administrators of the server machines are technically sophisticated and will require assistance only in unexpected failure scenarios. They will need to receive immediate assistance via telephone as well as query resolution via e-mail.
- *Database Administrators*: The database administrators are technically sophisticated and will require assistance only with unfamiliar database behavior. They will need to receive immediate assistance via telephone as well as query resolution via e-mail.

Support Providers

- *Web Services Help Desk*: This organizational group is responsible for resolving all support incidents raised by users of the Web interface. They provide support via e-mail and telephone, six days per week, 20 hours per day.
- *IT Help Desk*: This organizational group is responsible for resolving all support incidents raised by users of the Windows client interface. They provide support via e-mail and telephone and can often provide direct assistance at the end user's desk as well. Support is provided during normal business hours.
- *DBA Group*: This organizational group is responsible for resolving all support incidents related to database management systems. They provide support via e-mail and telephone. Support is normally provided during normal business hours, with the option of using on-call staff outside this period.
- *Windows Administrators*: This organizational group is responsible for resolving all support incidents related to IIS, .NET, and Windows Server 2008 and underlying hardware. They provide support via e-mail and telephone. Support is provided during normal business hours, with the option of using on-call staff outside this period.

- *Microsoft Support*: This is an external organization (the Microsoft Corporation's Support division) that is responsible for assisting with the resolution of support incidents caused by a fault or usage problem with the SQL Server 2008, Windows Server 2008, or IIS products. They provide support via e-mail, newsgroups, Web sites, fax, and telephone. Support is provided 24 hours per day, every day.
- *Development Team*: This is the organizational group that developed the system originally and maintains it on an ongoing basis. They are responsible for resolving any incident that other support providers cannot resolve. They provide support via e-mail, telephone, and site visits during normal business hours, with the ability to reach on-call staff during other times.

Support Incidents and Resolution

- *Web Usage Difficulties*: This class of support incident covers any situation where a user of the Web interface is having problems using the system that are not caused by failure or malfunction of a system component. These incidents should be resolved in a single interaction with the Web Services Help Desk, either by phone or by e-mail. The impact on the organization should be minimal.
- *Windows Usage Difficulties*: This class of support incident covers any situation where a user of the Windows client interface is having problems using the system that are not caused by failure or malfunction of a system component. These incidents should be resolved in a single interaction with the IT Help Desk, either by phone or by e-mail. The impact on the organization should be minimal.
- *End-User System Errors*: This class of support incident covers any situation where a user of the system encounters a problem caused by failure or malfunction of a system component. These incidents should be resolved within 1 working day. The user should interact entirely with staff of the IT or Web Services Help Desk, who will manage problem resolution and deal with other support providers as required. The impact on the organization should be moderate and should not threaten business operations beyond inconvenience.
- *Slow End-User Performance*: This class of support incident covers any situation where end users complain of unacceptably slow performance. These incidents should be resolved within three working days. The user should interact entirely with members of the IT

or Web Services Help Desk, who will manage problem resolution and deal with other support providers as required. The impact on the organization should be moderate and should not threaten business operations beyond inconvenience.

- *Database Corruption*: This class of support incident covers any situation where the database system reports internal corruption. These incidents should be resolved within 2 hours (although realistically it is recognized that they could return; however, the original incident should be resolved within 2 hours). The DBA Group is responsible for recognizing and resolving these situations. The impact on the organization should be moderate, but business operations will be interrupted while the problem is resolved.
- *Database Failure*: This class of support incident covers any situation where the database system needs to be recovered from backups. These incidents should be resolved within 4 hours. The DBA Group is responsible for recognizing and resolving these situations. The impact on the organization may be severe during this period, with business operations being interrupted for the whole period of the incident, but should not continue beyond the resolution of the incident.
- *IIS or Windows Server Failure*: This class of support incident covers any situation where the IIS Server, underlying operating system, or underlying hardware suffers a failure. These incidents should be resolved within 1 hour. The Windows Administrators are responsible for recognizing and resolving these situations. The impact on the organization may be severe during this period, with business operations being interrupted for the whole period of the incident, but should not continue beyond the resolution of the incident.

Escalation

The escalation process is as follows.

- Users of the Web interface will report problems to the Web Services Help Desk.
- Users of the Windows client interface will report problems to the IT Help Desk.
- The Help Desks will report system problems to the Windows Administrators.
- The Windows Administrators will report database problems to the DBA Group.

- The Windows Administrators will report other problems to the Development Team.
- The Windows Administrators, DBA Group, and Development Team will report problems with Microsoft software to the Microsoft Support organization.

In each case, the organization accepting the incident must provide the reporter with a unique identifier for the incident and record the reporter's description of it. If the problem is not immediately resolved, the organization accepting the incident must provide the reporter with information on resolution status within 75% of the target resolution time.

PROBLEMS AND PITFALLS

Lack of Engagement with the Operational Staff

In many organizations, a gulf exists between the development staff members who build systems and the operational staff members who deploy and administer them. This can be a significant problem if you want to achieve a smooth, incident-free system rollout.

RISK REDUCTION

- The best solution to this problem is to engage early with the operational groups, stressing how valuable their contribution is. Operational staff often have a legitimate grievance with software developers because systems are frequently passed on to them with very little thought given to operational requirements.
- Use an explicit Operational view to help avoid this situation.

Lack of Backout Planning

Many systems we have seen don't have real backout plans. In fact, many commercial software products don't have graceful recovery mechanisms to cope with situations like failed upgrades. Without a good backout plan, you are relying on a perfect rollout for your entire system, which experience suggests is somewhat optimistic.

RISK REDUCTION

- Make sure that your system can be backed out of its production environment by defining a clear procedure and reviewing it.

Lack of Migration Planning

Many information systems replace a manual system, a previous automated system, or an earlier version of themselves, but many systems are developed without a good migration plan. Migration planning isn't glamorous or, in many cases, even interesting, but without it, you are unlikely to achieve a smooth system deployment.

RISK REDUCTION

- Make sure you understand the migration needs of your architecture as early as possible.
- Address migration needs in your AD.

Insufficient Migration Window

In our experience, data migration *always* takes longer than anticipated, typically because the data does not conform to the level of quality and consistency expected of it and because of the problems associated with handling and manipulating large volumes of data. If you are moving data between geographical locations or between different types of data stores (e.g., importing data from a legacy hierarchical database into a relational database), the process will take longer still.

RISK REDUCTION

- Consider how you will deal with data errors and inconsistencies.
- Develop processes for accepting migrated data, and make sure your stakeholders have bought into them.
- In your hardware sizing models, factor in the storage requirements for transitional data.
- Include adequate elapsed-time contingency in your migration plan.
- Factor in the time needed to reorganize databases, create indexes, and so on.
- When you are migrating data from live systems and the migration time is substantial, create strategies for reconciling any data updates made during the migration period.

Missing Management Tools

Most software developers (and, in fact, many architects) are very focused on the business of building new software. However, successful software spends most of its life in production, not development. This mismatch between focus

and lifecycle often manifests itself as a lack of operational facilities, which can result in a system that is difficult to monitor and control. Software developers can monitor or control the system by using primitive tools (operating system commands or simple scripts) because of their detailed knowledge of its internal workings. Operational staff often don't have this knowledge and need more sophisticated tools to automate the required operational procedures. Without such tools, the system is unlikely to be managed well.

RISK REDUCTION

- Understand the needs of your administration stakeholders as early as possible and involve them in the development of the Operational view.
- Ensure that administrators' needs are addressed with standard, system-wide facilities.

Production Environment Constraints

All production environments, whether in-house or external, impose constraints on your application and its operation, so it's important to understand the likely deployment environments you'll be using, and the constraints that each imposes, as soon as you can. If you are using a mixture of environments (for example, hosting in-house but using a cloud environment for short-term "burst" capacity, or using different cloud providers for production and disaster recovery), naturally you need to understand the overall set of constraints across all of the environments.

Production environments may impose constraints on you such as rigid unavailability periods for platform maintenance, particular tools that must be used for deployment or monitoring, limited choices of platform components that are available, procedures that must be adhered to for deployment and operation, and limits to the service-level agreements available. All of these have the ability to affect the system qualities that you may be able to achieve in a particular production environment.

RISK REDUCTION

- Agree on the target production environment(s) for your system as early as possible to allow their constraints to be understood.
- Clearly define what you need and are expecting from a production environment (e.g., reliability, capacity, and availability) as soon as you can so that you can spot possible problems with proposed environments.
- Analyze the planned production environments and what they offer and require so that you understand the opportunities and constraints that they imply and can integrate these into your work.

- Whether in-house or external, obtain definite commitments from the suppliers of your production environments for the service levels that they offer, and test them as much as you can to gain confidence in their realism.

Lack of Integration into the Production Environment

Most information systems are deployed into some sort of existing production environment, even if it is a simple or informal one. Unfortunately, it's common to find that a new system doesn't work with the environment. This can be quite a problem for operational staff who need to learn new interfaces or tools or even totally new ways to manage the system, particularly if you are using a third-party hosting environment for some or all of the system.

RISK REDUCTION

- Make sure that you understand the existing environment and its integration needs early in your system design.
- Involve experts who understand the target production environment as early as possible, and get their advice on how it works and the type and level of integration needed.

Inadequate Backup Models

Backup and restore processes can fail quite spectacularly, and you don't want to find out about problems in your model when you are desperately trying to recover important data.

RISK REDUCTION

- Do not be tempted to skimp on this area or omit it from consideration entirely.
- Incorporate backup and restore as a central part of your architecture rather than trying to add it afterward.
- Make sure that your backup scheme includes all the information you need for data recovery.
- Estimate how long backup and recovery will take, and perform some practical testing under realistic conditions.
- Make sure that your model describes how data will be restored as well as backed up.
- Consider how to maintain data consistency across multiple data stores when you have to restore one of them to an earlier state.
- Consider a "belt-and-braces" approach to backup. For example, many end-user systems—especially older, mainframe-based ones—write copies

of updates received from clients to an audit and recovery area as well as to the main database. This means that if the database is damaged, it is possible to replay these transactions into the database in order to get it in synch again.

Unsuitable Alerting

This pitfall either manifests itself as *alert starvation*, when the system fails to send appropriate alerts when important events occur, or *alert flooding*, when the system sends so many events to the central console that they are ignored or important ones are missed. Either scenario is a significant operational problem since small incidents can soon become big ones and big ones can become catastrophic.

RISK REDUCTION

- Although this pitfall is probably more appropriately addressed during design and build, you can set the scene by defining some suitable architectural principles and approaches in the AD.

CHECKLIST

- Do you know what it takes to install your system?
- Do you have a plan for backing out a failed installation?
- Can you upgrade an existing version of the system (if required)?
- Do you understand the facilities and constraints of the proposed production environment(s) that you plan to use? Can you live with or mitigate these if they are not ideal?
- Do you know how information will be moved from the existing environment into the new system?
- Do you have a clear migration strategy to move workload to the new system? Can you reverse the migration if you need to? How will you deal with data synchronization (if required)?
- Do you know how the system will be backed up? Are you confident that the approach identified will allow reliable system restoration in an acceptable time period?
- Are the administrators confident that they can monitor and control the system in production?
- Do the administrators have a clear understanding of the procedures they need to perform for the system?

- How will performance metrics be captured for the system's elements?
- Can you manage the configuration of all of the system's elements?
- Do you know how support will be provided for the system? Is the support suitable for the stakeholders it is being provided for?
- Have you cross-referenced the requirements of the administration model back to the Development view to ensure that they will be implemented consistently?
- Is the data migration architecture compatible with the amount of time available to perform the data migration? Are there catch-up mechanisms in place where the source data is volatile during the data migration?

FURTHER READING

Little existing literature deals with the operational aspects of a system from the perspective of the application development team. Although there are many books on installing and managing specific pieces of technology, we have found very few books that examine the principles that underpin reliable production systems operation.

Some books that at least partially address this area are [KERN04], [BEHR05], and [JAYA05], and [ALLS10] is an interesting collection of essays written by operations experts that provides a valuable insight into production operations. Also, Dyson and Longshaw [DYSO04] includes a number of patterns useful in the Operational view. It can also be useful to understand how production services are provided, and ITIL has been a very influential model in this area; [BON07] is a concise overview of ITIL v3. A book written from the software development perspective but dealing extensively with the interface to production operations is [NYGA07].

23

Achieving Consistency across Views

The use of views addresses one of the biggest challenges you face as an architect: to represent a large and complex system in a way your stakeholders can understand. A view is a way to portray those aspects or elements of the architecture that are relevant to the concerns the view intends to address—and, by implication, the stakeholders for whom those views are important.

Without views, you end up with a single, all-encompassing model that tries (and usually fails) to illustrate all of the aspects of your system. Such a model is complex, uses a mix of notations, and is too hard for anyone to understand—never mind appreciate the subtleties, nuances, and implications of your architectural choices.

However, the problem with partitioning the representation of your architecture through using views is that it is difficult to ensure consistency between them—in other words, to ensure that the structures, features, and elements that appear in one view are compatible and in alignment with the content of your other views. This consistency is a vital characteristic of your AD—without it, the system will not work properly, will not achieve its design goals, and may even be impossible to build.

Unfortunately, although some design tools can simplify the process of creating your models, we are not aware of any currently available tool that will automate such consistency checking to the extent that you need it to. The use of formal modeling languages such as UML only partially addresses this problem, and the tools that support these languages typically provide only basic features for checking one model for consistency against another. And, of course, if you are using an informal notation or one you have developed for your specific situation, the problem is even worse.

Ensuring consistency between views therefore largely comes down to the skill, thoroughness, and diligence of the architect and (to a lesser extent)

the stakeholders. We have found the following strategies to be helpful in achieving inter-view consistency.

- *Focus on consistency from the outset*: We saw that trying to apply quality properties after the fact doesn't work—good performance, availability, and resilience have to be designed into your solution from the start. Similarly, it is no good waiting until your models are nearly complete to determine whether they are consistent with one another: More likely than not, they won't be, and you will be faced with a significant piece of rework and additional review.
- *Enumerate model elements*: Assigning each significant model element a unique identifier simplifies the process of asking such questions as "Is element 3 from Model B consistent with element 5 from Model D?"
- *Ensure that consistency checks are a formal part of reviews*: Consistency should be one of the criteria you use to review models and other architectural documentation. This means both *internal consistency* (Is this part of the model consistent with other parts of this model?) and *external consistency* (Is this model consistent with other models that make up the AD?). If you perform such a formal consistency check, you should include its results (and the actions taken) in an appendix to your AD.

RELATIONSHIPS BETWEEN VIEWS

Although all of the views are obviously interrelated, in practice there are *strong* dependencies only between *some* of the views. The UML class diagram in Figure 23–1 shows the most important of these dependencies. The relationships illustrate a strong dependency, which implies that if something changes at the end of the arrow, a change will probably be required at the start of the arrow.

Conversely, if there's no dependency between two views, changing something in one is unlikely to itself necessitate a change in the other. (So changing a Development view element, for example, does not in itself imply any changes to the Functional models—unless you are changing it for a reason not to do with development, of course.)

Note that if you don't develop a particular view—for example, if you encapsulate the concurrency aspects of the architecture in the Functional view, rather than in a separate Concurrency view—it is still useful to apply the checklists presented in this chapter for that view, to ensure that you have addressed its most important concerns.

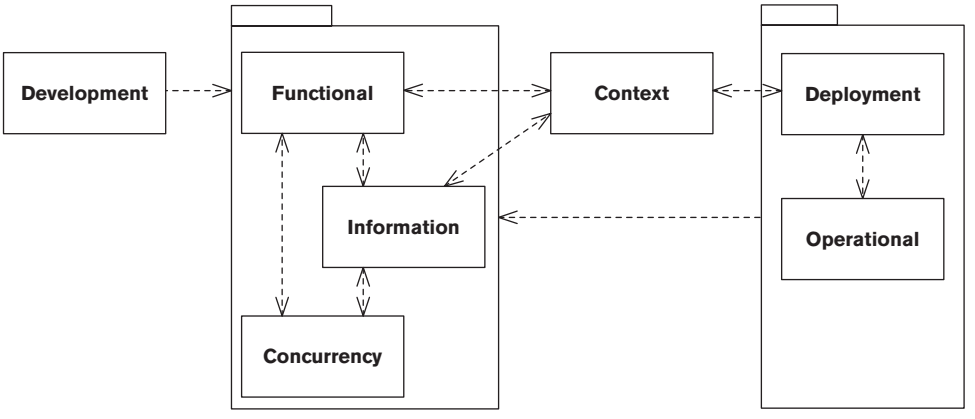

FIGURE 23–1 DEPENDENCIES BETWEEN VIEWS

CONTEXT AND FUNCTIONAL VIEW CONSISTENCY

Goal: To ensure that the system scope and requirements are fully and correctly implemented by the system.

- Does each requirement map to one or more functional elements that implement that requirement?
- Is every functional element necessary (directly or indirectly) in order to implement at least one requirement?
- Has every quality property that affects system functionality been taken into consideration in the system structure defined by the Functional view?
- Is every external entity defined in one view also present in the other view, and do they have the same definition in each view?
- Is every interface defined in one view also present in the other view, and do they have the same definition (responsibilities, nature, and characteristics) in each view?
- Are the interaction scenarios defined in the Context view compatible with the functional structure of the system and the way its elements interact with one another and the outside world?

CONTEXT AND INFORMATION VIEW CONSISTENCY

Goal: To ensure that data flows in and out of the system are compatible with the information management approach defined in the Information view.

- Has consideration been given in the Information view to all of the data items identified in the Context view that flow into the system (ownership, consistency, timeliness, and so on)?
- Has consideration been given in the Information view to all of the data items identified in the Context view that flow out of the system (ownership, consistency, timeliness, and so on)?
- Has every quality property that affects information management been taken into consideration in the Information view?
- Is the data ownership model in the Information view (particularly when data is owned by external entities) compatible with the responsibilities defined for external entities in the Context view?
- Is the high-level data model in the Information view compatible with the data models used by external systems, or if not, have appropriate mechanisms for data transformation been defined?
- If external archiving services are defined in the Information view, are they represented as external entities in the Context view?

CONTEXT AND DEPLOYMENT VIEW CONSISTENCY

Goal: To ensure that external connections between this system and others can be supported in the planned deployment environment.

- Do all external entities that represent systems, interfaces, or other technology-based connections appear consistently in both the Context and the Deployment views?
- Does the Deployment view contain all of the hardware and software required to communicate with the external entities identified in the Context view?
- Is the technology used for each interface in the Deployment view appropriate for its nature and characteristics as defined in the Context view?
- Are system elements that communicate with external entities deployed to parts of the deployment environment where external communication is possible (e.g., to a DMZ in the network)?
- Has every quality objective identified in the Context view that affects the deployment environment been taken into account in the Deployment view?

FUNCTIONAL AND INFORMATION VIEW CONSISTENCY

Goal: To ensure that the functional and information structures are compatible and that nothing is missing in one that is required by the other.

- Does every nontrivial functional element in the Functional view that needs persistent data have corresponding data elements in the Information view?
- Does every nontrivial data element in the Information view have at least one element in the Functional view that is responsible for the maintenance of that data?
- If information flows are described in the Information view, are they consistent with the interelement interactions in the Functional view?
- If the Information view requires specific functional features (e.g., distributed transaction support, redundant logging of updates, and so on), are these features addressed in the Functional view?
- Do the data ownership models in the Information view align with the functional structure in the Functional view?
- If the data ownership characteristics are complex (e.g., multiple creators or updaters), do the functional models reflect the requirements for maintaining distributed data consistency?
- If there are significant issues around the maintenance of distributed identifiers (keys), do the functional models include features to address these problems?
- If the architecture has significant data migration and data quality analysis aspects, are there functional elements for these in the Functional view?
- If the functional structure has loose coupling as an architectural goal, is this reflected (as far as possible) in the static information structure?

FUNCTIONAL AND CONCURRENCY VIEW CONSISTENCY

Goal: To ensure that the functional elements are all mapped to tasks that will allow them to execute and that the interelement interactions are supported by interprocess communication mechanisms if required.

- Is every functional element in the Functional view mapped to a concurrency element (a process or thread) responsible for its execution in the Concurrency view?
- If functional elements are partitioned into separate processes, are suitable interprocess communication mechanisms used to allow all of the interelement interactions shown in the Functional view?
- If multiple functional elements are packaged into a single process, is it clear which element controls the process?

FUNCTIONAL AND DEVELOPMENT VIEW CONSISTENCY

Goal: To ensure that all of the functional elements are mapped to a design-time module and to ensure that the common processing, test approach, and codeline specified are all compatible with and can support the proposed functional structure.

- Does the code module structure include all of the functional elements that need to be developed?
- Does the Development view specify a development environment for each of the technologies used by the Functional view?
- If the Functional view specifies the use of a particular architectural style, does the Development view include sufficient guidelines and constraints to ensure correct implementation of the style?
- Where common processing is specified, can it be implemented in a straightforward manner over all of the elements defined in the Functional view?
- Where reusable functional elements can be identified from the Functional view, are these modeled as libraries or similar features in the Development view?
- If a test environment has been specified, does it meet the functional needs and priorities of the elements defined in the Functional view?
- Can the functional structure described in the Functional view be built, tested, and released reliably using the codeline described in the Development view?

FUNCTIONAL AND DEPLOYMENT VIEW CONSISTENCY

Goal: To ensure that each of the functional elements is correctly mapped to its deployment environment.

- Has each functional element been mapped to a processing node to allow it to be executed?
- Where functional elements are hosted on different nodes, do the network models allow the required element interactions to occur?
- Are functional elements hosted as close as possible to the information they need to process?
- Are functional elements that need to interact extensively hosted as close together as possible?

- Are the specified network connections sufficient for the needs of the interelement interactions that will be carried over them (in terms of capacity, reliability, security, and so on)?
- Is the hardware specified in the Deployment view the most efficient solution for hosting the specified functional elements?

FUNCTIONAL AND OPERATIONAL VIEW CONSISTENCY

Goal: To ensure that each of the specified functional elements can be installed, used, operated, managed, and supported.

- Does the Operational view make it clear how every functional element will be installed (and upgraded if necessary)?
- If migration is required, does the Operational view make it clear how migration will occur to every functional element that needs it?
- Does the Operational view explain how each functional element will be monitored and controlled in the production environment?
- Does the Operational view explain how the configuration of each functional element will be managed in the production environment?
- Does the Operational view explain how the performance of each functional element will be monitored in the production environment?
- Does the Operational view explain how each functional element will be supported in the production environment?
- Are the approaches that the Operational view specifies for installation, migration, monitoring, control, and support the simplest set that will support the needs of the system's functional elements?

INFORMATION AND CONCURRENCY VIEW CONSISTENCY

Goal: To ensure that the concurrency structure of the system does not cause data access problems and that the proposed information structure is compatible with the concurrency structure.

- Does the concurrency design imply concurrent access to any of the system's data elements? If so, have the data elements been protected from concurrent access problems?
- When functional elements are packaged into operating system processes, is the data they require still available to them?
- If functional elements that share data elements are packaged into different operating system processes, has a suitable interprocess data-sharing mechanism been defined?

INFORMATION AND DEVELOPMENT VIEW CONSISTENCY

Goal: To ensure that the proposed development environment can provide the technical resources required to develop the data management aspects of the system.

- Does each data management technology identified in the Information view have development tools and the environment defined for it?
- Does the sizing of the development environments and test data platforms reflect the data volumes created in the Information view?
- If the Information view defines a significant migration data aspect, are there development tools and environments defined to support this?
- If the Information view defines external data components (e.g., for existing systems or external systems under construction), does the Development view take this into account (e.g., the creation of stub environments, realistic test data, and so on)?

INFORMATION AND DEPLOYMENT VIEW CONSISTENCY

Goal: To ensure that the proposed deployment environment provides the resources required to support the defined information structure.

- Does the Deployment view include enough storage (of the appropriate types) to support the information storage approach specified by the Information view?
- If separate storage hardware is used, does the Deployment view specify sufficiently fast and reliable links from the storage to the processing hardware?
- Does the Deployment view reflect the requirements for backup and recovery as addressed by the Information view?
- If large volumes of information need to be moved, is sufficient bandwidth available so that this can be achieved without critically impacting the operation of the system?

INFORMATION AND OPERATIONAL VIEW CONSISTENCY

Goal: To ensure that the system's information structure can be installed, used, operated, managed, and supported.

- Does the Operational view make it clear whether specific installation steps are required for the system's data management technology?
- If migration is required, does the Operational view make it clear how data migration will occur?
- Does the Operational view explain how the data management technology will be monitored and controlled in the production environment?
- Does the Operational view explain how the configuration of the data management technology will be managed in the production environment?
- Does the Operational view explain how the performance of the data management technology will be monitored in the production environment?
- Does the Operational view explain how the data management technology will be supported in the production environment?

CONCURRENCY AND DEVELOPMENT VIEW CONSISTENCY

Goal: To ensure that the concurrency structure specified in the Concurrency view can be built and tested in the development environment specified by the Development view.

- If the concurrency structure is complex, are sufficient design patterns specified in the Development view to guide its implementation?
- Does the codeline defined in the Development view support the packaging of the system's functional elements into the operating system processes specified by the Concurrency view?
- Does the test approach defined in the Development view support testing of the concurrency structure specified in the Concurrency view?
- Does the development environment defined in the Development view allow development and testing of the concurrency structure specified in the Concurrency view?

CONCURRENCY AND DEPLOYMENT VIEW CONSISTENCY

Goal: To ensure that the system's runtime tasks are correctly mapped to execution resources.

- Is every operating system process mapped to a processing node to allow it to run?
- Can the interprocess communication facilities used in the Concurrency view be implemented on and between the processing nodes specified in the Deployment view?

- Are the processing nodes specified in the Deployment view sufficiently powerful to host the processes mapped to them from the Concurrency view?
- Is every processing node in the Deployment view fully used by the processes mapped to it?

DEPLOYMENT AND OPERATIONAL VIEW CONSISTENCY

Goal: To ensure that the deployment environment described in the Deployment view can be installed, used, monitored, managed, and supported.

- Does the Operational view define how each of the elements in the deployment environment will be installed?
- Does the Operational view describe how each of the elements in the deployment environment can be monitored and controlled?
- Does the Operational view make it clear which monitoring and control facilities already exist, which can be bought, and which must be developed?
- Can each of the elements in the deployment environment be supported in the organization?