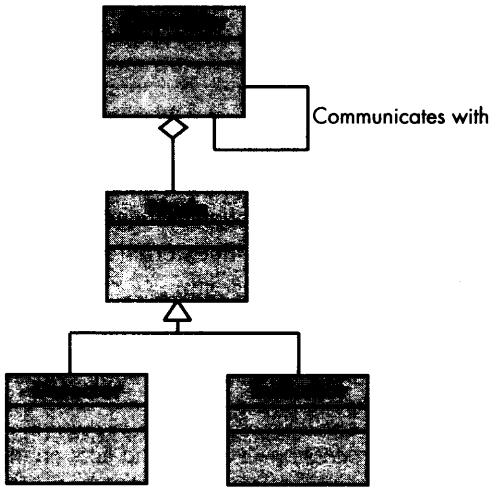
FIGURE 10.7

UML relationships for *SafeHome* security function archetypes (adapted from [BOS00])



Recall that the archetypes form the basis for the architecture but are abstractions that must be further refined as architectural design proceeds. For example, **Detector** might be refined into a class hierarchy of sensors.

10.4.3 Refining the Architecture into Components

ADVICE

Components of the software architecture are derived from three sources—the application domain, the infrastructure domain, and the interface domain. Because analysis modeling does not address infrastructure, allocate sufficient design time to consider it carefully.

As the software architecture is refined into components, the structure of the system begins to emerge. But how are these components chosen? In order to answer this question, the architectural designer begins with the classes that were described as part of the analysis model.[6] These analysis classes represent entities within the application (business) domain that must be addressed within the software architecture. Hence, the application domain is one source for the derivation and refinement of components. Another source is the infrastructure domain. The architecture must accommodate many infrastructure components that enable application components but have no business connection to the application domain. For example, memory management components, communication components, database components, and task management components are often integrated into the software architecture.

The interfaces depicted in the architecture context diagram (Section 10.4.1) imply one or more specialized components that process the data that flow across the interface. In some cases (e.g. a graphical user interface), a complete subsystem architecture with many components must be designed.

6 If a conventional (non-object-oriented) approach is chosen, components can be derived from the data flow model. We discuss this approach in Section 10.6.

"The structure of a software system provides the ecology in which code is born, matures, and dies. A well-designed habitat allows for the successful evolution of all the components needed in a software system."

R. Pattis

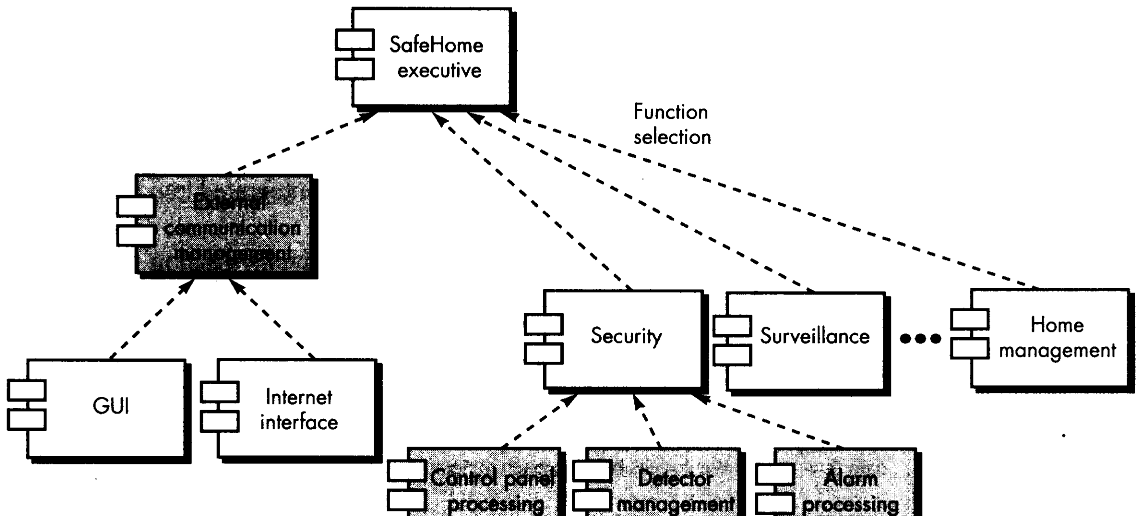
Continuing the *SafeHome* home security function example, we might define the set of top-level components that address the following functionality:

- *External communication management*—coordinates communication of the security function with external entities, for example, Internet-based systems, external alarm notification.
- *Control panel processing*—manages all control panel functionality.
- *Detector management*—coordinates access to all detectors attached to the system.
- *Alarm processing*—verifies and acts on all alarm conditions.

Each of these top-level components would have to be elaborated iteratively and then positioned within the overall *SafeHome* architecture. Design classes (with appropriate attributes and operations) would be defined for each. It is important to note, however, that the design details of all attributes and operations would not be specified until component-level design (Chapter 11).

The overall architectural structure (represented as a UML component diagram) is illustrated in Figure 10.8. Transactions are acquired by *External communication management* as they move in from components that process the *SafeHome GUI* and the *Internet interface*. This information is managed by a *SafeHome executive* component that

FIGURE 10.8 **Overall architectural structure for *SafeHome* with top-level components**



selects the appropriate product function (in this case, security). The *control panel processing* component interacts with the homeowner to arm/disarm the security function. The *detector management* component polls sensors to detect an alarm condition, and the *alarm processing* component produces output when an alarm is detected.

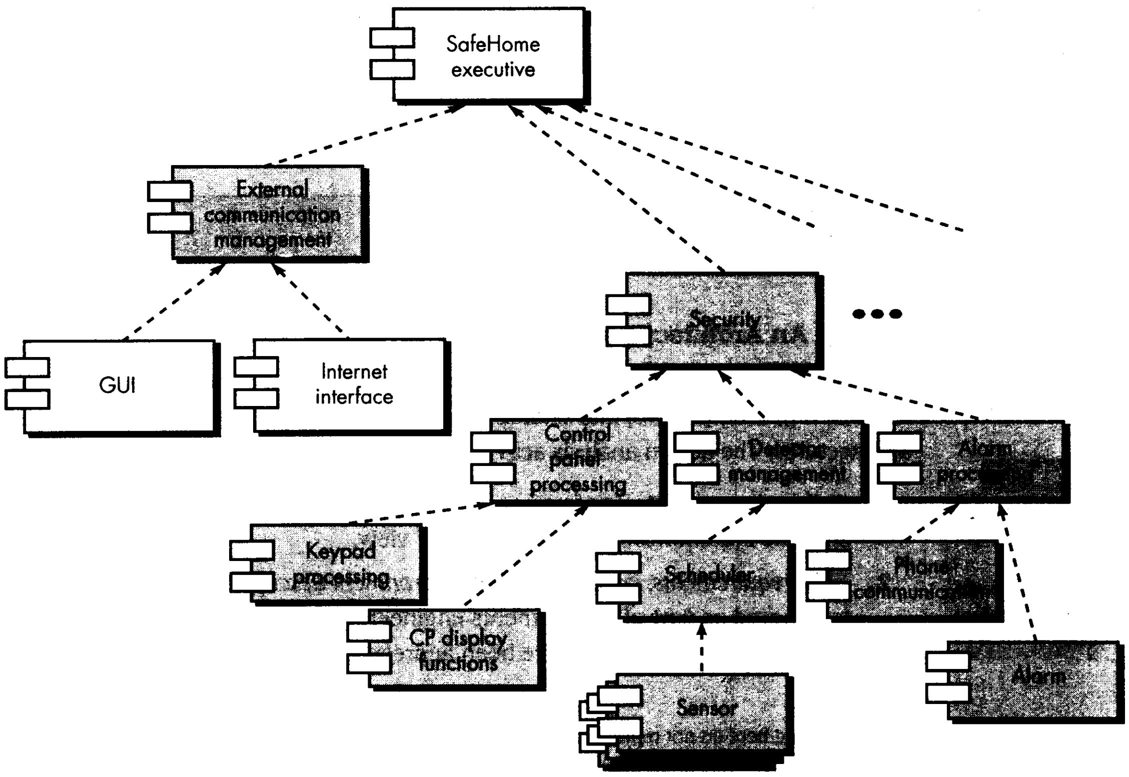
10.4.4 Describing Instantiations of the System

The architectural design that has been modeled to this point is still relatively high level. The context of the system has been represented; archetypes that indicate the important abstractions within the problem domain have been defined; the overall structure of the system is apparent; and the major software components have been identified. However, further refinement (recall that all design is iterative) is still necessary.

To accomplish this, an actual *instantiation* of the architecture is developed. By this we mean that the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.

Figure 10.9 illustrates an instantiation of the *SafeHome* architecture for the security system. Components shown in Figure 10.8 are refined further to show additional detail. For example, the *detector management* component interacts with a *scheduler*

FIGURE 10.9 **An instantiation of the security function with component elaboration**



infrastructure component that implements "concurrent" polling of each *sensor* object used by the security system. Similar elaboration is performed for each of the components represented in Figure 10.8.

SOFTWARE TOOLS

Architectural Design

Objective: Architectural design tools model the overall software structure by representing component interfaces, dependencies and relationships, and interactions.

Mechanics: Tool mechanics vary. In most cases, architectural design capability is part of the functionality provided by automated tools for analysis and design modeling.

Representative Tools[7]

Adalon, developed by Synthis Corp. (www.synthis.com), is a specialized design tool for the design and construction of specific Web-based component architectures.

ObjectiF, developed by microTOOL GmbH (www.microtool.com), is a UML-based design tool that leads to architectures (e.g., Coldfusion, J2EE, Fusebox) amenable to component-based software engineering (Chapter 30).

Rational Rose, developed by Rational (www.rational.com), is a UML-based design tool that supports all aspects of architectural design.

10.5 Assessing Alternative Architectural Designs

At its best, design results in a number of architectural alternatives that are each assessed to determine which is the most appropriate for the problem to be solved. In the sections that follow, we consider the assessment of alternative architectural designs.

[illegible] Let me go upstairs and check."

10.5.1 An Architecture Trade-Off Analysis Method

The Software Engineering Institute (SEI) has developed an *architecture trade-off analysis method* (ATAM) [KAZ98] that establishes an iterative evaluation process for software architectures. The design analysis activities that follow are performed iteratively:

WebRef
In-depth information on ATAM can be obtained at www.sei.cmu.edu/ata/ata_method.html.

1. *Collect scenarios.* A set of use-cases (Chapters 7 and 8) is developed to represent the system from the user's point of view.
2. *Elicit requirements, constraints, and environment description.* This information is required as part of requirements engineering and is used to be certain that all stakeholder concerns have been addressed.

7 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

3. *Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements.*
4. *Evaluate quality attributes by considering each attribute in isolation.* Quality attributes for architectural design assessment include reliability, performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability.
5. *Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.* This can be accomplished by making small changes in the architecture and determining how sensitive a quality attribute, say performance, is to the change. Any attributes that are significantly affected by variation in the architecture are termed *sensitivity points.*
6. *Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5.* The SEI describes this approach in the following manner [KAZ98]:

> Once the architectural sensitivity points have been determined, finding trade-off points is simply the identification of architectural elements to which multiple attributes are sensitive. For example, the performance of a client-server architecture might be highly sensitive to the number of servers (performance increases, within some range, by increasing the number of servers). . . . The number of servers, then, is a trade-off point with respect to this architecture.

These six steps represent the first ATAM iteration. Based on the results of steps 5 and 6, some architecture alternatives may be eliminated, one or more of the remaining architectures may be modified and represented in more detail, and then the ATAM steps are reapplied.[8]

SAFEHOME

[illegible] Assessment

[illegible] Doug Miller's office as [illegible] proceeds.

[illegible] Shakira, and Ed—members [illegible] team. Also, Doug [illegible] engineering group.

[illegible] a couple of [illegible] product, and that's a good thing. I [illegible] going to choose the [illegible]

Ed: I'm working [illegible] either Jamie or I are [illegible]

Doug: Okay, and how [illegible]

Shakira: I took [illegible] and I remember [illegible] do it.

8 The *Software Architecture Analysis Method* (SAAM) is an alternative to ATAM and is well-worth examining by those readers interested in architectural analysis. A paper on SAAM can be downloaded from: http://www.sei.cmu.edu/publications/articles/saam-metho-propert-sas.html.

Vinod: There are, but they're a bit academic. Look, I think we can do our assessment and choose the right one using use-cases and scenarios.

Doug: Isn't that the same thing?

Vinod: Not when you're talking about architectural assessment. We already have a complete set of use-cases. Simply apply each to both architectures and see how the system reacts—how components and connectors work in the use-case context.

Ed: That's a good idea. Makes sure we didn't leave anything out.

Vinod: True, but it also tells us whether the architectural design is convoluted, whether the system has to twist itself into a pretzel to get the job done.

Jamie: Scenarios aren't just another name for use-cases?

Vinod: No, in this case a scenario implies something different.

Doug: You're talking about a quality scenario or a change scenario, right?

Vinod: Yes. What we do is go back to the stakeholders and ask them how *SafeHome* is likely to change over the next, say, three years. You know, new versions, features, that sort of thing. We build a set of change scenarios. We also develop a set of quality scenarios that define the attributes we'd like to see in the software architecture.

Jamie: And we apply them to the alternatives.

Vinod: Exactly. The style that handles the use-cases and scenarios best is the one we choose.

10.5.2 Architectural Complexity

A useful technique for assessing the overall complexity of a proposed architecture is to consider dependencies between components within the architecture. These dependencies are driven by information/control flow within the system. Zhao [ZHA98] suggests three types of dependencies:

> *Sharing dependencies* represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers. For example, for two components *u* and *v*, if *u* and *v* refer to the same global data, then there exists a shared dependence relationship between *u* and *v*.
>
> *Flow dependencies* represent dependence relationships between producers and consumers of resources. For example, for two components *u* and *v*, if *u* must complete before control flows into *v* (prerequisite), or if *u* communicates with *v* by parameters, then there exists a flow dependence relationship between *u* and *v*.
>
> *Constrained dependencies* represent constraints on the relative flow of control among a set of activities. For example, for two components *u* and *v*, if *u* and *v* cannot execute at the same time (mutual exclusion), then there exists a constrained dependence relationship between *u* and *v*.

The sharing and flow dependencies noted by Zhao are similar to the concept of coupling discussed in Chapter 9. Coupling is an important design concept that is applicable at the architectural level and at the component level. Simple metrics for evaluating coupling are discussed in Chapter 15.

10.5.3 Architectural Description Languages

The architect of a house has a set of standardized tools and notation that allow the design to be represented in an unambiguous, understandable fashion. Although the software architect can draw on UML notation, other diagrammatic forms, and a few related tools, there is a need for a more formal approach to the specification of an architectural design.

Architectural description language (ADL) provides a semantics and syntax for describing a software architecture. Hofmann and his colleagues [HOF01] suggest that an ADL should provide the designer with the ability to decompose architectural components, compose individual components into larger architectural blocks, and represent interfaces (connection mechanisms) between components. Once descriptive, language-based techniques for architectural design have been established, it is more likely that effective assessment methods for architectures will be established as the design evolves.

SOFTWARE TOOLS

Architectural Description Languages

The following summary of a number of important ADLs was prepared by Rickard Land [LAN02] and is reprinted with the author's permission. It should be noted that the first five ADLs listed have been developed for research purposes and are not commercial products.

Rapide (poset.stanford.edu/rapide/) [LUC95] builds on the notion of partial ordered sets.

UniCon (www.cs.cmu.edu/~UniCon) [SHA96] defines software architectures in terms of abstractions that designers find useful.

Aesop (www.cs.cmu.edu/~able/aesop/) [GAR94] addresses the problem of style reuse.

Wright (www.cs.cmu.edu/~able/wright/) [ALL97] formalizes architectural styles using predicates, thus allowing for static checks to determine the consistency and completeness of an architecture.

Acme (www.cs.cmu.edu/~acme/) [GAR00] is a second-generation ADL.

UML (www.uml.org/) includes many of the artifacts needed for architectural descriptions, but is not as complete as other ADLs.

10.6 Mapping Data Flow into a Software Architecture

The styles discussed in Section 10.3.1 represent radically different architectures, so it should come as no surprise that a comprehensive mapping that accomplishes the transition from the analysis model to a variety of architectural styles does not exist. In fact, there is no practical mapping for some architectural styles. The designer must approach the translation of requirements to design for these styles by using the techniques discussed in Section 10.4.

To illustrate one approach to architectural mapping, we consider a mapping technique for the *call and return* architecture—an extremely common structure for many types of systems. This mapping technique enables a designer to derive reasonably complex call and return architectures from data flow diagrams within the analysis model. The technique, sometimes called *structured design,* is presented in books by Myers [MYE78] and Yourdon and Constantine [YOU79].

Structured design is often characterized as a data flow-oriented design method because it provides a convenient transition from a data flow diagram (Chapter 8) to software architecture. The type of information flow is the driver for the mapping approach.

10.6.1 Transform Flow

Information must enter and exit software in an "external world" form. For example, data typed on a keyboard, tones on a telephone line, and video images in a multimedia application are all forms of external world information. Such externalized data must be converted into an internal form for processing. Information enters the system along paths that transform external data into an internal form. These paths are identified as *incoming flow.* At the kernel of the software, a transition occurs. Incoming data are passed through a *transform center* and begin to move along paths that now lead "out" of the software. Data moving along these paths are called *outgoing flow.* The overall flow of data occurs in a sequential manner and follows one, or only a few, "straight line" paths.[9] When a segment of a data flow diagram exhibits these characteristics, *transform* flow is present.

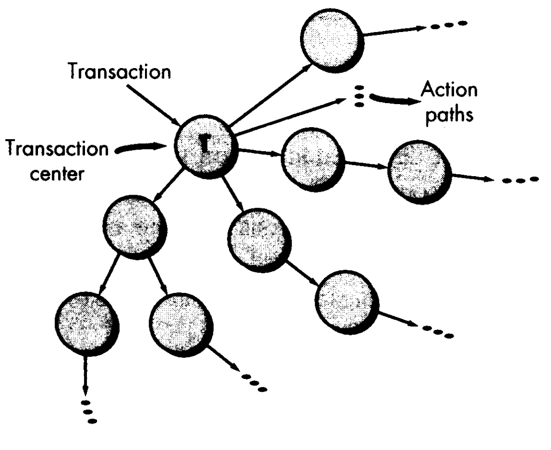
10.6.2 Transaction Flow

Information flow is often characterized by a single data item, called a *transaction,* that triggers other data flow along one of many paths. When a DFD takes the form shown in Figure 10.10, transaction flow is present.

Transaction flow is characterized by data moving along an incoming path that converts external world information into a transaction. The transaction is evalu-

FIGURE 10.10

Transaction flow



9 An obvious mapping for this type of information flow is the data flow architecture described in Section 10.3.1. There are many cases, however, where the data flow architecture may not be the best choice for a complex system. Examples include systems that will undergo substantial change over time or systems in which the processing associated with the data flow is not necessarily sequential.

ated and, based on its value, flow along one of many *action paths* is initiated. The hub of information flow from which many action paths emanate is called a *transaction center.*

It should be noted that, within a DFD for a large system, both transform and transaction flow may be present. For example, in a transaction-oriented flow, information flow along an action path may have transform flow characteristics.

10.6.3 Transform Mapping

Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style. To illustrate this approach, we again consider the *SafeHome* security function.[10] One element of the analysis model is a set of data flow diagrams that describe information flow within the security function. To map these data flow diagrams into an architecture, the following design steps are initiated:

Step 1. Review the fundamental system model. The fundamental system model or context diagram depicts the security function as a single transformation, representing the external producers and consumers of data that flow into and out of the function. Figure 10.11 depicts a level 0 model, and Figure 10.12 depicts refined data flow for the security function.

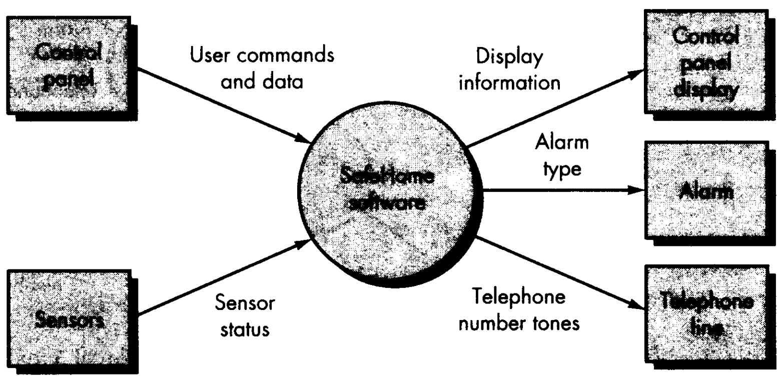
ADVICE

If the DFD is refined further at this time, strive to derive bubbles that exhibit high cohesion.

Step 2. Review and refine data flow diagrams for the software. Information obtained from the analysis models is refined to produce greater detail. For example, the level 2 DFD for *monitor sensors* (Figure 10.13) is examined, and a level 3 data flow

FIGURE 10.11

Context level DFD for the *SafeHome* security function



10 We consider only the portion of the *SafeHome* security function that uses the control panel. Other features, discussed earlier in this book and this chapter, will not be considered here.

FIGURE 10.12

Level 1 DFD for the *SafeHome* security function

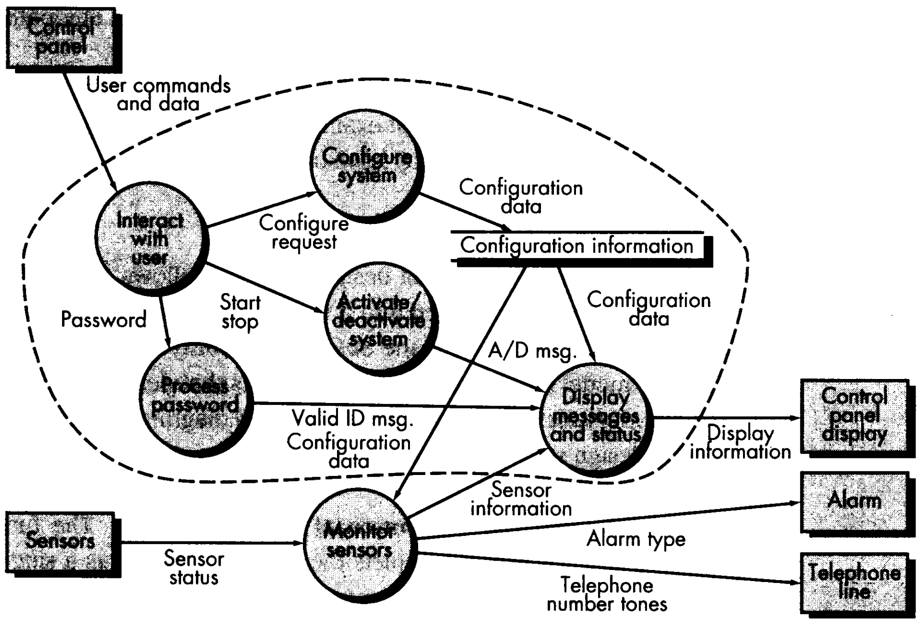


FIGURE 10.13

Level 2 DFD that refines the *monitor sensors* transform

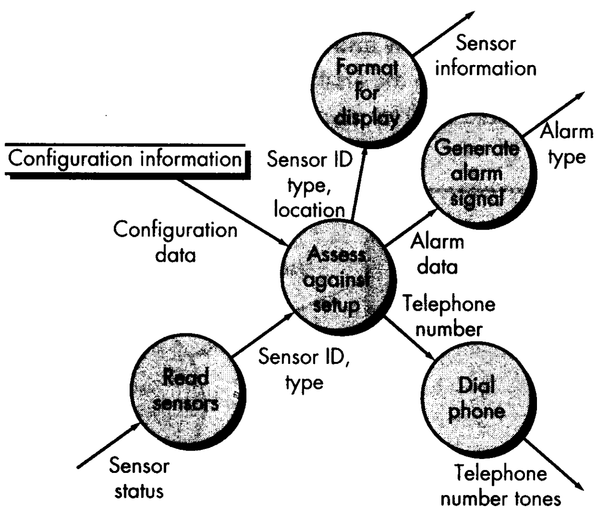
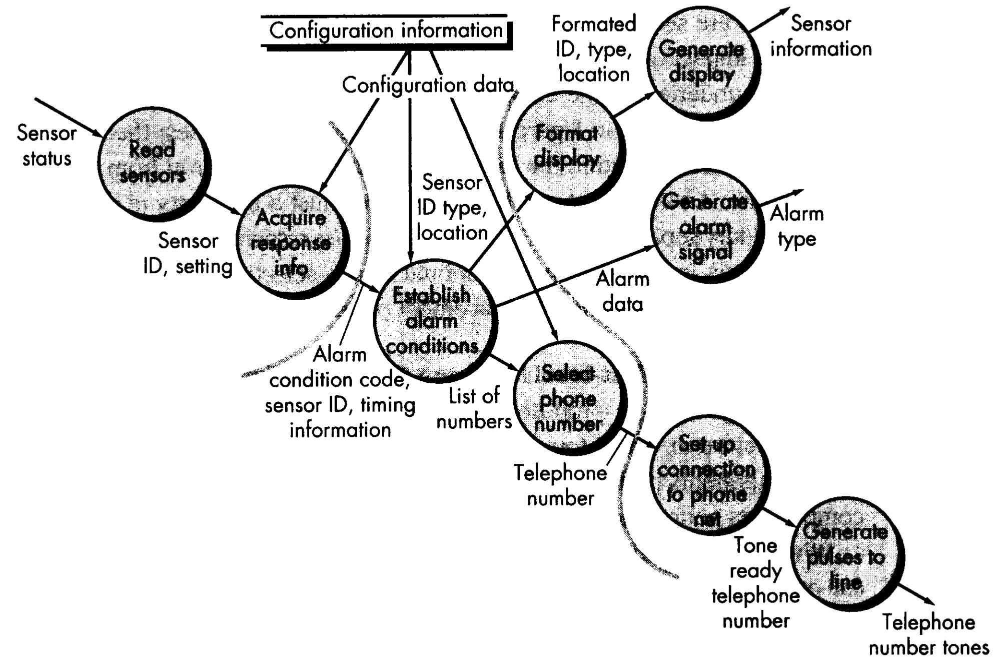


diagram is derived as shown in Figure 10.14. At level 3, each transform in the data flow diagram exhibits relatively high cohesion (Chapter 9). That is, the process implied by a transform performs a single, distinct function that can be implemented as a component in the *SafeHome* software. Therefore, the DFD in Figure 10.14 contains sufficient detail for a "first cut" at the design of architecture for the *monitor sensors* subsystem, and we proceed without further refinement.

FIGURE 10.14 **Level 3 DFD for *monitor sensors* with flow boundaries**



KEY POINT

You will often encounter both types of data flow within the same flow-oriented model. The flows are partitioned and program structure is derived using the appropriate mapping.

Step 3. Determine whether the DFD has transform or transaction flow characteristics. In general, information flow within a system can always be represented as transform. However, when an obvious transaction characteristic (Figure 10.10) is encountered, a different design mapping is recommended. In this step, the designer selects global (software-wide) flow characteristics based on the prevailing nature of the DFD. In addition, local regions of transform or transaction flow are isolated. These *subflows* can be used to refine program architecture derived from a global characteristic described previously. For now, we focus our attention only on the *monitor sensors* subsystem data flow depicted in Figure 10.14.

Evaluating the DFD (Figure 10.14), we see data entering the software along one incoming path and exiting along three outgoing paths. No distinct transaction center is implied (although the transform establishes alarm conditions that could be perceived as such). Therefore, an overall transform characteristic will be assumed for information flow.

ADVICE

Vary the location of flow boundaries in an effort to explore alternative program structures. This takes very little time and provides important insight.

Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries. In the preceding section incoming flow was described as a path that converts information from external to internal form; outgoing flow converts from internal to external form. Incoming and outgoing flow boundaries are open to interpretation. That is, different designers may select slightly different points in the flow as boundary locations. In fact, alternative design solutions can be derived by varying the placement of flow boundaries. Although care should be taken when boundaries are selected, a variance of one bubble along a flow path will generally have little impact on the final program structure.

Flow boundaries for the example are illustrated as shaded curves running vertically through the flow in Figure 10.14. The transforms (bubbles) that constitute the transform center lie within the two shaded boundaries that run from top to bottom in the figure. An argument can be made to readjust a boundary (e.g., an incoming flow boundary separating *read sensors* and *acquire response info* could be proposed). The emphasis in this design step should be on selecting reasonable boundaries, rather than lengthy iteration on placement of boundaries.

ADVICE

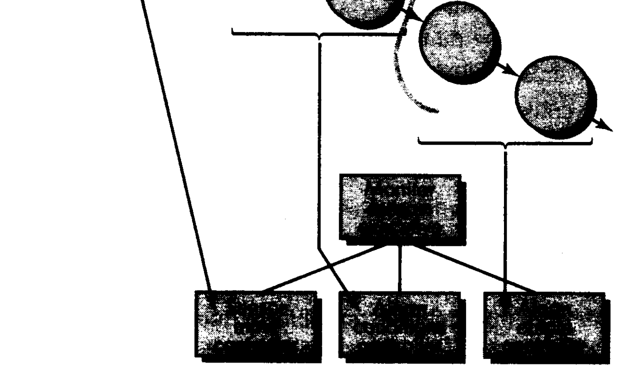
Don't become dogmatic at this stage. It may be necessary to establish two or more controllers for input processing or computation, based on the complexity of the system to be built. If common sense dictates this approach, do it!

Step 5. Perform "first-level factoring." The program architecture derived using this mapping results in a top-down distribution of control. *Factoring* results in a program structure in which top-level components perform decision-making and low-level components perform most input, computation, and output work. Middle-level components perform some control and do moderate amounts of work.

When transform flow is encountered, a DFD is mapped to a specific structure (a call and return architecture) that provides control for incoming, transform, and outgoing information processing. This first-level factoring for the *monitor sensors* subsystem is illustrated in Figure 10.15. A main controller (called *monitor sensors*

FIGURE 10.15

First-level factoring for *monitor sensors*



executive) resides at the top of the program structure and coordinates the following subordinate control functions:

- An incoming information processing controller, called *sensor input controller,* coordinates receipt of all incoming data.
- A transform flow controller, called *alarm conditions controller,* supervises all operations on data in internalized form (e.g., a module that invokes various data transformation procedures).
- An outgoing information processing controller, called *alarm output controller,* coordinates production of output information.

Although a three-pronged structure is implied by Figure 10.15, complex flows in large systems may dictate two or more control modules for each of the generic control functions described previously. The number of modules at the first level should be limited to the minimum that can accomplish control functions and still maintain good functional independence characteristics.

ADVICE

Keep "worker" modules low in the program structure. This will lead to an architecture that is easier to maintain.

Step 6. Perform "second-level factoring." Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture. Beginning at the transform center boundary and moving outward along incoming and then outgoing paths, transforms are mapped into subordinate levels of the software structure. The general approach to second-level factoring is illustrated in Figure 10.16

Although Figure 10.16 illustrates a one-to-one mapping between DFD transforms and software modules, different mappings frequently occur. Two or even three bubbles can be combined and represented as one component, or a single bubble may be expanded to two or more components. Practical considerations and measures of design quality dictate the outcome of second-level factoring. Review and refinement may lead to changes in this structure, but it can serve as a "first-iteration" design.

ADVICE

Eliminate redundant control modules. That is, if a control module does nothing except control one other module, its control function should be imploded to a higher level module.

Second-level factoring for incoming flow follows in the same manner. Factoring is again accomplished by moving outward from the transform center boundary on the incoming flow side. The transform center of *monitor sensors* subsystem software is mapped somewhat differently. Each of the data conversion or calculation transforms of the transform portion of the DFD is mapped into a module subordinate to the transform controller. A completed first-iteration architecture is shown in Figure 10.17.

The components mapped in the preceding manner and shown in Figure 10.17 represent an initial design of software architecture. Although components are named in a way that implies function, a brief processing narrative (adapted from the PSPEC created during analysis modeling) should be written for each.

ADVICE

Focus on the functional independence of the modules you've derived. High cohesion and low coupling should be your goal.

Step 7. Refine the first-iteration architecture using design heuristics for improved software quality. A first-iteration architecture can always be refined by applying concepts of functional independence (Chapter 9). Components are exploded or imploded to produce sensible factoring, good cohesion, minimal coupling,

FIGURE 10.16

Second-level factoring for *monitor sensors*

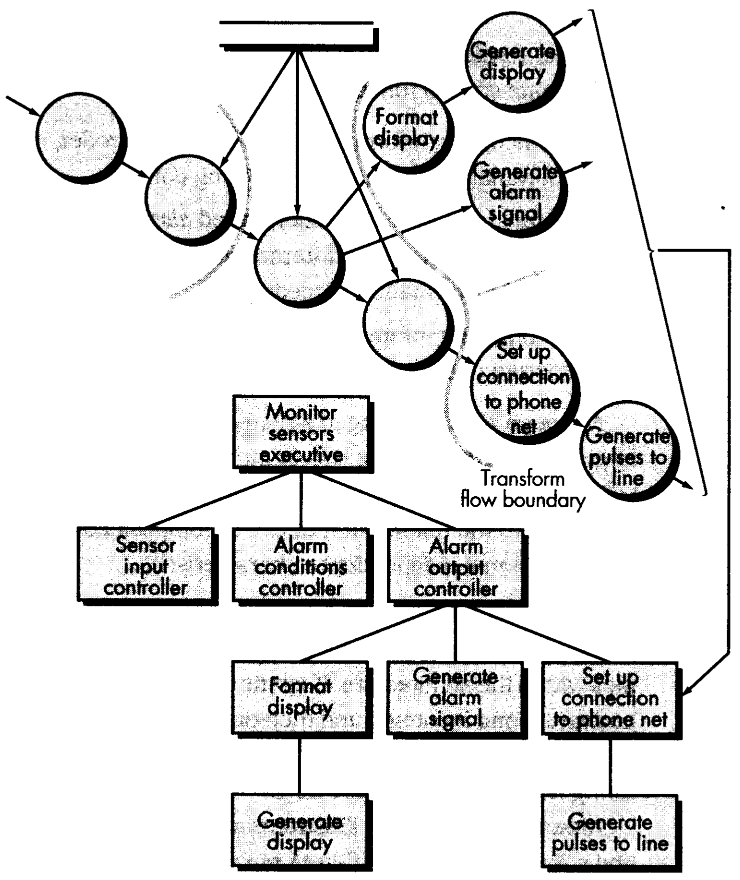
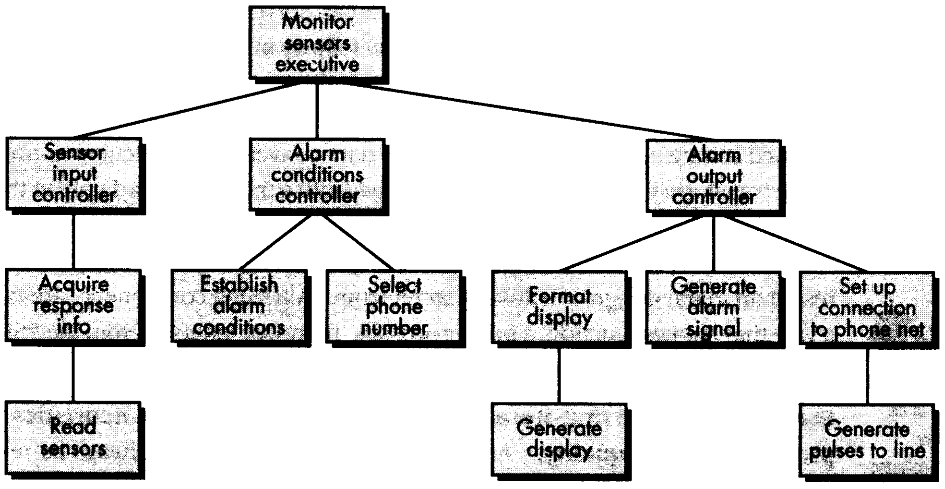


FIGURE 10.17 First iteration structure for *monitor sensors*



and most importantly, a structure that can be implemented without difficulty, tested without confusion, and maintained without grief.

Refinements are dictated by the analysis and assessment methods described briefly in Section 10.5, as well as practical considerations and common sense. There are times, for example, when the controller for incoming data flow is totally unnecessary, when some input processing is required in a component that is subordinate to the transform controller, when high coupling due to global data cannot be avoided, or when optimal structural characteristics cannot be achieved. Software requirements coupled with human judgment is the final arbiter.

The objective of the preceding seven steps is to develop an architectural representation of software. That is, once structure is defined, we can evaluate and refine software architecture by viewing it as a whole. Modifications made at this time require little additional work, yet can have a profound impact on software quality.

The reader should pause for a moment and consider the difference between the design approach described and the process of "writing programs." If code is the only representation of software, the developer will have great difficulty evaluating or refining at a global or holistic level and will, in fact, have difficulty "seeing the forest for the trees."

SAFEHOME

Refining a First-Cut Architecture

The scene: Jamie's cubicle, as design modeling continues.

The players: Jamie and Ed—members of the *SafeHome* software engineering team.

The conversation:

(Ed has just completed a first-cut design of the *monitor sensors subsystem*. He stops in to ask Jamie her opinion.)

Ed: So here's the architecture that I derived.

(Ed shows Jamie Figure 10.17, which she studies for a few moments.)

Jamie: That's cool, but I think we can do a few things to make it simpler . . . and better.

Ed: Such as?

Jamie: Well, why did you use the *sensor input controller* component?

Ed: Because you need a controller for the mapping.

Jamie: Not really. The controller doesn't do much, since we're managing a single flow path for incoming data. We can eliminate the controller with no ill effects.

Ed: I can live with that, I'll make the change and . . .

Jamie (smiling): Hold up! We can also implode the components *establish alarm conditions* and *select phone number*. The transform controller you show isn't really necessary, and the small decrease in cohesion is tolerable.

Ed: Simplification, huh?

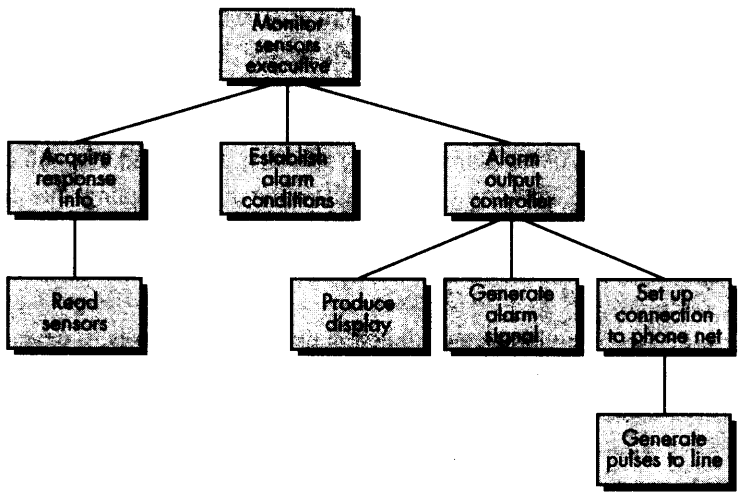
Jamie: Yep. And while we're making refinements, it would be a good idea to implode the components *format display* and *generate display*. Display formatting for the control panel is simple. We can define a new module called *produce display*.

Ed (sketching): So this is what you think we should do?

(He shows Jamie Figure 10.18.)

Jamie: It's a start.

FIGURE 10.18



Refined program structure for *monitor sensors*

10.6.4 Transaction Mapping

In many software applications, a single data item triggers one of a number of information flows that effect a function implied by the triggering data item. The data item, called a *transaction,* and its corresponding flow characteristics are discussed in Section 10.6.2. In this section we consider design steps used to map transaction flow into a software architecture.

Transaction mapping will be illustrated by considering the user interaction subsystem of the *SafeHome* security function. Level 1 data flow for this subsystem is shown as part of Figure 10.12. Refining the flow, a level 2 data flow diagram is developed and shown in Figure 10.19. The data object **user commands** flows into the system and results in additional information flow along one of three action paths. A single data item, **command type,** causes the data flow to fan outward from a hub. Therefore, the overall data flow characteristic is transaction-oriented.

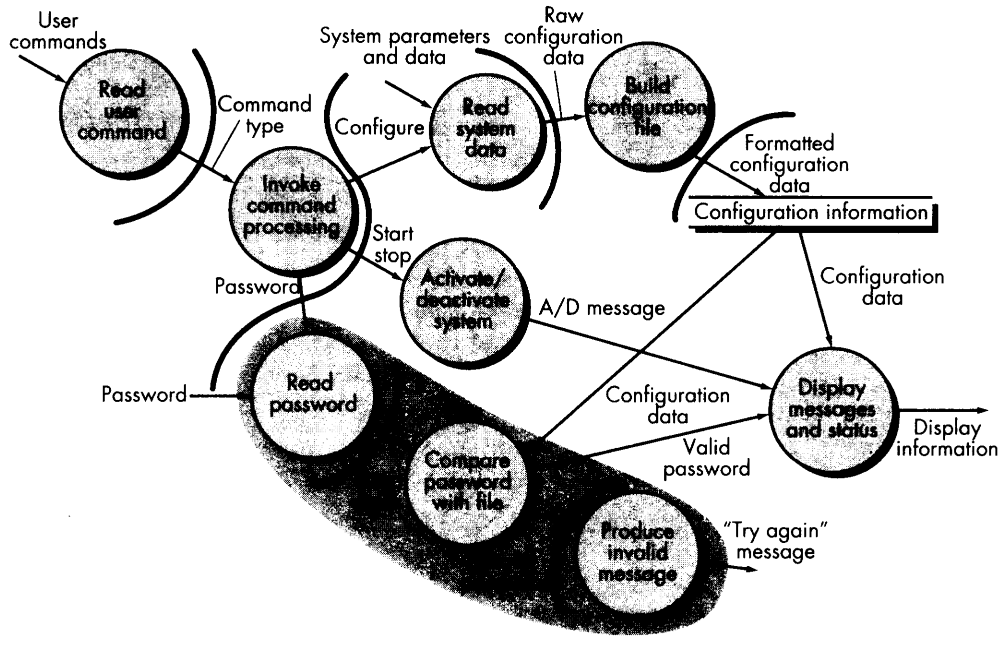
It should be noted that information flow along two of the three action paths accommodates additional incoming flow (e.g., **system parameters and data** are input on the "configure" action path). Each action path flows into a single transform, *display messages and status.*

The design steps for transaction mapping are similar and in some cases identical to steps for transform mapping (Section 10.6.3). A major difference lies in the mapping of DFD to software structure.

Step 1. Review the fundamental system model.

Step 2. Review and refine data flow diagrams for the software.

FIGURE 10.19 **Level 2 DFD for user interaction subsystem**



Step 3. Determine whether the DFD has transform or transaction flow characteristics.

Steps 1, 2, and 3 are identical to corresponding steps in transform mapping. The DFD shown in Figure 10.19 has a classic transaction flow characteristic. However, flow along two of the action paths emanating from the *invoke command processing* bubble appears to have transform flow characteristics. Therefore, flow boundaries must be established for both flow types.

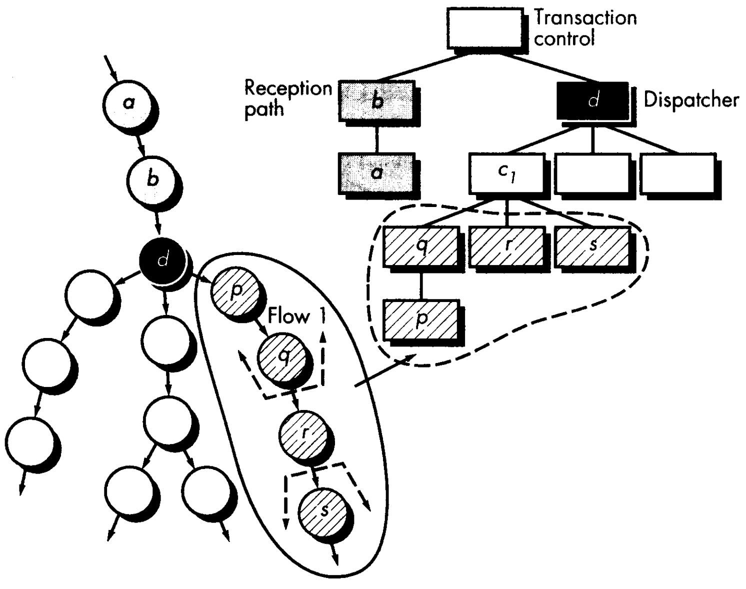
Step 4. Identify the transaction center and the flow characteristics along each of the action paths. The location of the transaction center can be immediately discerned from the DFD. The transaction center lies at the origin of a number of actions paths that flow radially from it. For the flow shown in Figure 10.19, the *invoke command processing* bubble is the transaction center.

KEY POINT
First-level factoring results in the derivation of a control hierarchy for the software. Second-level factoring distributes "worker" modules under the appropriate controller.

The incoming path (i.e., the flow path along which a transaction is received) and all action paths must also be isolated. Each action path must be evaluated for its individual flow characteristic. For example, the "password" path (shown enclosed by a shaded area in Figure 10.19) has transform characteristics. Incoming, transform, and outgoing flow are indicated with boundaries.

Step 5. Map the DFD in a program structure amenable to transaction processing. Transaction flow is mapped into an architecture that contains an incoming branch and a dispatch branch. The structure of the incoming branch is

FIGURE 10.20
Transaction mapping



developed in much the same way as transform mapping. Starting at the transaction center, bubbles along the incoming path are mapped into modules. The structure of the dispatch branch contains a dispatcher module that controls all subordinate action modules. Each action flow path of the DFD is mapped to a structure that corresponds to its specific flow characteristics. This process is illustrated schematically in Figure 10.20.

Considering the user interaction subsystem data flow, first-level factoring for step 5 is shown in Figure 10.21. The bubbles *read user command* and *activate/deactivate system* map directly into the architecture without the need for intermediate control modules. The transaction center, *invoke command processing,* maps directly into a dispatcher module of the same name. Controllers for system configuration and password processing are created as illustrated in Figure 10.21.

Step 6. Factor and refine the transaction structure and the structure of each action path. Each action path of the data flow diagram has its own information flow characteristics. We have already noted that transform or transaction flow may be encountered. The action path-related "substructure" is developed using the design steps discussed in this and the preceding section.

As an example, consider the password processing information flow shown (inside shaded area) in Figure 10.19. The flow exhibits classic transform characteristics. A **password** is input (incoming flow) and transmitted to a transform center where it is compared against stored passwords. An alarm and warning message (outgoing flow) are produced (if a match is not obtained). The "configure" path is drawn simi-

FIGURE 10.21

First-level factoring for user interaction subsystem

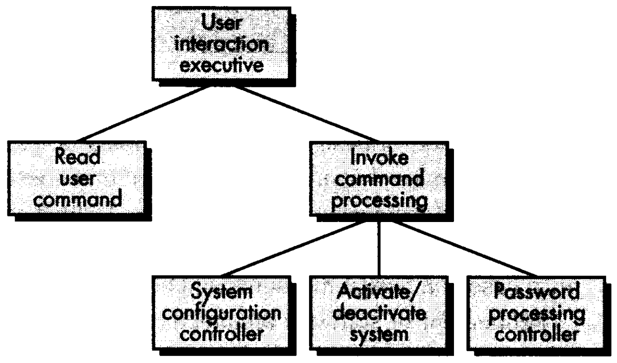
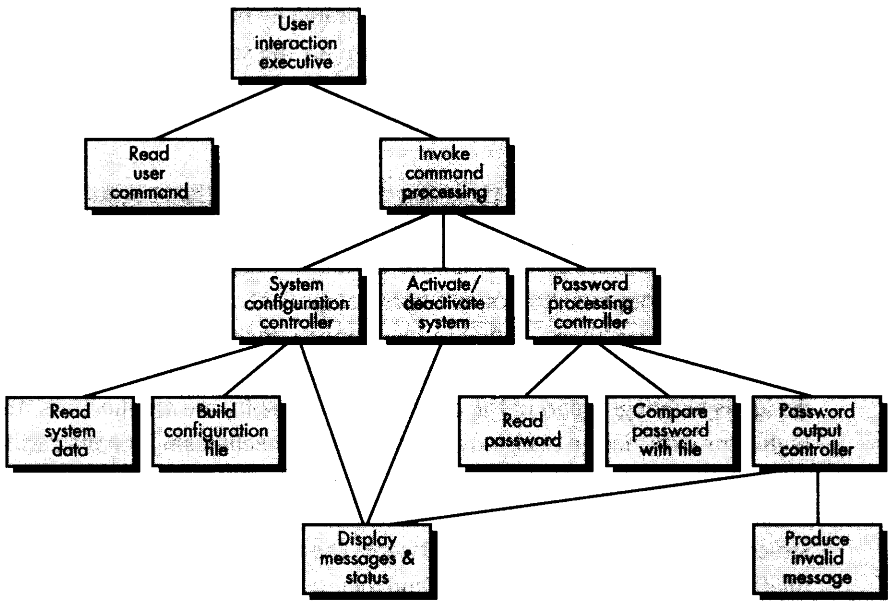

User interaction executive
Read user command
Invoke command processing
System configuration controller
Activate/ deactivate system
Password processing controller

FIGURE 10.22 First iteration architecture for user interaction subsystem


User interaction executive
Read user command
Invoke command processing
System configuration controller
Activate/ deactivate system
Password processing controller
Read system data
Build configuration file
Read password
Compare password with file
Password output controller
Display messages & status
Produce invalid message

larly using the transform mapping. The resultant software architecture is shown in Figure 10.22.

Step 7. Refine the first-iteration architecture using design heuristics for improved software quality. This step for transaction mapping is identical to the corresponding step for transform mapping. In both design approaches, criteria such as module independence, practicality (efficacy of implementation and test), and maintainability must be carefully considered as structural modifications are proposed.

"Make it as simple as possible. But no simpler."

Albert Einstein

10.6.5 Refining the Architectural Design

Any discussion of design refinement should be prefaced with the following comment: Remember that an "optimal design" that doesn't work has questionable merit. The software designer should be concerned with developing a representation of software that will meet all functional and performance requirements and merit acceptance based on design measures and heuristics.

Refinement of software architecture during early stages of design is to be encouraged. As we discussed earlier in this chapter, alternative architectural styles may be derived, refined, and evaluated for the "best" approach. This approach to optimization is one of the true benefits derived by developing a representation of software architecture.

It is important to note that structural simplicity often reflects both elegance and efficiency. Design refinement should strive for the smallest number of components that is consistent with effective modularity and the least complex data structure that adequately serves information requirements.

10.7 Summary

Software architecture provides a holistic view of the system to be built. It depicts the structure and organization of software components, their properties, and the connections between them. Software components include program modules and the various data representations that are manipulated by the program. Therefore, data design is an integral part of the derivation of the software architecture. Architecture highlights early design decisions and provides a mechanism for considering the benefits of alternative system structures.

Data design translates the data objects defined in the analysis model into data structures that reside within the software. The attributes that describe the object, the relationships between data objects and their use within the program all influence the choice of data structures. At a higher level of abstraction, data design may lead to the definition of an architecture for a database or a data warehouse.

A number of different architectural styles and patterns are available to the software engineer. Each style describes a system category that (1) encompasses a set of components that perform a function required by a system, (2) a set of connectors that enable communication, coordination and cooperation among components, (3) constraints that define how components can be integrated to form the system, and (4) semantic models that enable a designer to understand the overall properties of a system.

In a general sense, architectural design is accomplished using four distinct steps. First, the system must be represented in context. That is, the designer should define the external entities that the software interacts with and the nature of the interaction. Once context has been specified, the designer should identify a set of top-level abstractions, called archetypes, that represent pivotal elements of the system's behavior or function. After abstractions have been defined, the design begins to move closer to the implementation domain. Components are identified and represented within the context of an architecture that supports them. Finally, specific instantiations of the architecture are developed to "prove" the design in a real world context.

As a simple example of architectural design, the mapping method presented in this chapter uses data flow characteristics to derive a commonly used architectural style. A data flow diagram is mapped into program structure using one of two mapping approaches—transform mapping or transaction mapping. Once an architecture has been derived, it is elaborated and then analyzed against quality criteria.

REFERENCES

[AHO83] Aho, A. V., J. Hopcroft, and J. Ullmann, *Data Structures and Algorithms,* Addison-Wesley, 1983.

[ALL97] Allen R., "A Formal Approach to Software Architecture," Ph.D. Thesis, Carnegie Mellon University, Technical Report Number: CMU-CS-97-144 1997.

[BAR00] Barroca, L. and P. Hall (eds.), *Software Architecture: Advances and Applications,* Springer-Verlag, 2000.

[BAS03] Bass, L., P. Clements, and R. Kazman, *Software Architecture in Practice,* 2nd ed. Addison-Wesley, 2003.

[BOS00] Bosch, J., *Design & Use of Software Architectures,* Addison-Wesley, 2000.

[BUS96] Buschmann, F., *Pattern-Oriented Software Architecture,* Wiley, 1996.

[DAT00] Date, C. J., *An Introduction to Database Systems,* 7th ed., Addison-Wesley, 2000.

[DIK00] Dikel, D., D. Kane, and J. Wilson, *Software Architecture: Organizational Principles and Patterns,* Prentice-Hall, 2000.

[FRE80] Freeman, P., "The Context of Design," in *Software Design Techniques,* 3rd ed. (P. Freeman and A. Wasserman, eds.), IEEE Computer Society Press, 1980, pp. 2–4.

[GAR94] Garlan D., R. Allen, and J. Ockerbloom, "Exploiting Style in Architectural Design Environments," in *Proceedings of SIGSOFT '94 Symposium on the Foundations of Software Engineering,* 1994.

[GAR00] Garlan D., R. T. Monroe, and D. Wile, "Acme: Architectural Description of Component-Based Systems," in *Foundations of Component-Based Systems,* G. T. Leavens and M. Sitarman, eds. Cambridge University Press, 2000.

[HOF00] Hofmeister, C., R. Nord, and D. Soni, *Applied Software Architecture,* Addison-Wesley, 2000.

[HOF01] Hofmann, C., et al., "Approaches to Software Architecture," downloadable from: http://citeseer.nj.nec.com/84015.html.

[KAZ98] Kazman, R., et al., *The Architectural Tradeoff Analysis Method,* Software Engineering Institute, CMU/SEI-98-TR-008, July 1998.

[KIM98] Kimball, R., L. Reeves, M. Ross, and W. Thornthwaite, *The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing, and Deploying Data Warehouses,* Wiley, 1998.

[LAN02] Land R., A Brief Survey of Software Architecture, Technical Report, Dept. of Computer Engineering, Mälardalen University, Sweden, February, 2002.

[LUC95] Luckham D. C., et al., "Specification and Analysis of System Architecture Using Rapide," *IEEE Transactions on Software Engineering,* issue "Special Issue on Software Architecture," 1995.

[MAT96] Mattison, R., *Data Warehousing: Strategies, Technologies and Techniques,* McGraw-Hill, 1996.

[MYE78] Myers, G., *Composite Structured Design,* Van Nostrand, 1978.

[PRE98] Preiss, B. R., *Data Structures and Algorithms: With Object-Oriented Design Patterns in C++,* Wiley, 1998.

[SHA96] Shaw, M., and D. Garlan, *Software Architecture,* Prentice-Hall, 1996.

[SHA97] Shaw, M., and P. Clements, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems," *Proc. COMPSAC,* Washington, DC, August 1997.

[WAS80] Wasserman, A., "Principles of Systematic Data Design and Implementation," in *Software Design Techniques* (P. Freeman and A. Wasserman, eds.), 3rd ed., IEEE Computer Society Press, 1980, pp. 287–293.

[YOU79] Yourdon, E., and L. Constantine, *Structured Design,* Prentice-Hall, 1979.

[ZHA98] Zhao, J., "On Assessing the Complexity of Software Architectures," *Proc. Intl. Software Architecture Workshop,* ACM, Orlando, FL, 1998, pp. 163–167.

Problems and Points to Ponder

10.1. Using a data flow diagram and a processing narrative, describe a computer-based system that has distinct transform flow characteristics. Define flow boundaries and map the DFD into a software architecture using the technique described in Section 10.6.3.

10.2. Write a three- to five-page paper that presents guidelines for selecting data structures based on the nature of the problem. Begin by delineating the classical data structures encountered in software work and then describe criteria for selecting from these for particular types of problems.

10.3. Explain the difference between a database that services one or more conventional business applications and a data warehouse.

10.4. Using a data flow diagram and a processing narrative, describe a computer-based system that has distinct transaction flow characteristics. Define flow boundaries and map the DFD into a software structure using the technique described in Section 10.6.4.

10.5. Some of the architectural styles noted in Section 10.3.1 are hierarchical in nature and others are not. Make a list of each type. How would the architectural styles that are not hierarchical be implemented?

10.6. If you haven't done so, complete Problem 8.10. Use the design methods described in this chapter to develop a software architecture for the PHTRS.

10.7. Research the ATAM (use the SEI Web site) and present a detailed discussion of the six steps presented in Section 10.5.1.

10.8. Select an application with which you are familiar. Answer each of the questions posed for control and data in Section 10.3.3.

10.9. Some designers contend that all data flow may be treated as transform-oriented. Discuss how this contention will affect the software architecture that is derived when a transaction-oriented flow is treated as transform. Use an example flow to illustrate important points.

10.10. The terms *architectural style, architectural pattern,* and *framework* are often encountered in discussions of software architecture. Do some research (use the Web) and describe how each of these terms differs from its counterparts.

10.11. Present two or three examples of applications for each of the architectural styles noted in Section 10.3.1.

10.12. Using the architecture of a house or building as a metaphor, draw comparisons with software architecture. How are the disciplines of classical architecture and software architecture similar? How do they differ?

Further Readings and Information Sources

The literature on software architecture has exploded over the past decade. Books by Fowler (*Patterns of Enterprise Application Architecture,* Addison-Wesley, 2003), Clements and his colleagues (*Documenting Software Architecture: View and Beyond,* Addison-Wesley, 2002), Schmidt and his colleagues (*Pattern-Oriented Software Architectures,* two volumes, Wiley, 2000), Bosch [BOS00], Dikel and his colleagues [DIK00], Hofmeister and his colleagues [HOF00], Bass, Clements, and Kazman [BAS03], Shaw and Garlan [SHA96], and Buschmann et al. [BUS96] provide in-depth treatment of the subject. Earlier work by Garlan (*An Introduction to Software Architecture,* Software Engineering Institute, CMU/SEI-94-TR-021, 1994) provides an excellent introduction. Clements and Northrop (*Software Product Lines: Practices and Patterns,* Addison-Wesley, 2001)) address the design of architectures that support software product lines. Clements and his colleagues (*Evaluating Software Architectures,* Addison-Wesley, 2002) consider the issues associated with the assessment of architectural alternatives and the selection of the best architecture for a given problem domain.

Implementation-specific books on architecture address architectural design within a specific development environment or technology. Wallnau and his colleagues (*Building Systems from Commercial Components,* Addison-Wesley, 2001) present methods for constructing component-based architectures. Pritchard (*COM and CORBA Side-by-Side,* Addison-Wesley, 1999), Mowbray (CORBA Design Patterns, Wiley, 1997) and Mark et al. (*Object Management Architecture Guide,* Wiley, 1996) provide detailed design guidelines for the CORBA distributed application support framework. Shanley (*Protected Mode Software Architecture,* Addison-Wesley, 1996) provides architectural design guidance for anyone designing PC-based real-time operating systems, multi-task operating systems, or device drivers.

Current software architecture research is documented yearly in the *Proceedings of the International Workshop on Software Architecture,* sponsored by the ACM and other computing organizations, and the *Proceedings of the International Conference on Software Engineering.* Barroca and Hall [BAR00] present a useful survey of recent research.

Data modeling is a prerequisite to good data design. Books by Teory (*Database Modeling and Design,* Academic Press, 1998); Schmidt (*Data Modeling for Information Professionals,* Prentice-Hall, 1998); Bobak (*Data Modeling and Design for Today's Architectures,* Artech House, 1997); Silverston, Graziano, and Inmon (*The Data Model Resource Book,* Wiley, 1997); Date [DAT00], and Reingruber and Gregory (*The Data Modeling Handbook: A Best-Practice Approach to Building Quality Data Models,* Wiley, 1994) contain detailed presentations of data modeling notation, heuristics, and database design approaches. The design of data warehouses has become increasingly important in recent years. Books by Humphreys, Hawkins, and Dy (*Data Warehousing: Architecture and Implementation,* Prentice-Hall, 1999); Kimball et al. [KIM98]; and Inmon [INM95] cover the topic in considerable detail.

General treatment of software design with discussion of architectural and data design issues can be found in most books dedicated to software engineering. More rigorous treatments of the subject can be found in Feijs (*A Formalization of Design Methods,* Prentice-Hall, 1993), Witt et al. (*Software Architecture and Design Principles,* Thomson Publishing, 1994), and Budgen (*Software Design,* Addison-Wesley, 1994).

Complete presentations of data flow-oriented design may be found in Myers [MYE78], Yourdon and Constantine [YOU79], and Page-Jones (*The Practical Guide to Structured Systems Design,* 2nd ed., Prentice-Hall, 1988). These books are dedicated to design alone and provide comprehensive tutorials in the data flow approach.

A wide variety of information sources on architectural design are available on the Internet. An up-to-date list of World Wide Web references that are relevant to architectural design can be found at the SEPA Web site:
http://www.mhhe.com/pressman.

CHAPTER

11 MODELING COMPONENT-LEVEL DESIGN

KEY CONCEPTS
cohesion
components
conventional
object-oriented
process-related
coupling
design
guidelines
graphical notation
principles
tasks
middleware
OCL
packaging principles
PDL
structured programming

Component-level design occurs after the first iteration of architectural design has been completed. At this stage, the overall data and program structure of the software has been established. The intent is to translate the design model into operational software. But the level of abstraction of the existing design model is relatively high, and the abstraction level of the operational program is low. The translation can be challenging, opening the door to the introduction of subtle errors that are difficult to find and correct in later stages of the software process. In a famous lecture, Edsgar Dijkstra, a major contributor to our understanding of software design, stated [DIJ72]:

Software seems to be different from many other products, where as a rule higher quality implies a higher price. Those who want really reliable software will discover that they must find a means of avoiding the majority of bugs to start with, and as a result, the programming process will become cheaper . . . effective programmers . . . should not waste their time debugging—they should not introduce bugs to start with.

Although these words were spoken many years ago, they remain true today. When the design model is translated into source code, we must follow a set of design principles that not only perform the translation but also do not "introduce bugs to start with."

It is possible to represent the component-level design using a programming language. In essence, the program is created using the architectural design model as a guide. An alternative approach is to represent the component-level design

QUICK LOOK

What is it? A complete set of software components is defined during architectural design. But the internal data structures and processing details of each component are not represented at a level of abstraction that is close to code. Component-level design defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each software component.

Who does it? A software engineer performs component-level design.

Why is it important? You have to be able to determine whether the software will work before you build it. The component-level design represents the software in a way that allows you to review the details of the design for correctness and consistency with earlier design representations (i.e., the data, architectural, and interface designs). It provides a means for assessing whether data structures, interfaces, and algorithms will work.

What are the steps? Design representations of data, architecture, and interfaces form the foundation for component-level design. The class definition or processing narrative for each component is translated into a detailed design that makes use of diagrammatic or text-based forms that specify internal data structures, local interface detail, and processing logic. Design notation encompasses UML diagrams and supplementary representations. Procedural design is specified using a set of structured programming constructs.

What is the work product? The design for each component, represented in graphical, tabular, or text-based notation, is the primary work product produced during component-level design.

How do I ensure that I've done it right? A design walkthrough or inspection is conducted. The design is examined to determine whether data structures, interfaces, processing sequences, and logical conditions are correct and will produce the appropriate data or control transformation allocated to the component during earlier design steps.

using some intermediate (e.g., graphical, tabular, or text-based) representation that can be translated easily into source code. Regardless of the mechanism that is used to represent the component-level design, the data structures, interfaces, and algorithms defined should conform to a variety of well-established design guidelines that help us to avoid errors as the procedural design evolves. In this chapter, we examine these design guidelines and the methods available for achieving them.

11.1 What Is a Component?

Stated in a general fashion, a *component* is a modular building block for computer software. More formally, the *OMG Unified Modeling Language Specification* [OMG01] defines a component as "a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."

As we discussed in Chapter 10, components populate the software architecture, and, as a consequence, play a role in achieving the objectives and requirements of the system to be built. Because components reside within the software architecture, they must communicate and collaborate with other components and with entities (e.g., other systems, devices, people) that exist outside the boundaries of the software.

"The details are not the details. They make the design."

Charles Eames

The true meaning of the term "component" will differ depending on the point of view of the software engineer who uses it. In the sections that follow, we examine three important views of what a component is and how it is used as design modeling proceeds.

11.1.1 An Object-Oriented View

KEY POINT

From an OO viewpoint, a component is a set of collaborating classes.

In the context of object-oriented software engineering, a component contains a set of collaborating classes.[1] Each class within a component has been fully elaborated to include all attributes and operations that are relevant to its implementation. As part of the design elaboration, all interfaces (messages) that enable the classes to communicate and collaborate with other design classes must also be defined. To accomplish this, the designer begins with the analysis model and elaborates analysis classes (for components that relate to the problem domain) and infrastructure classes (for components that provide support services for the problem domain).

To illustrate this process of design elaboration, consider software to be built for a sophisticated print shop. The overall intent of the software is to collect the customer's requirements at the front counter, cost a print job, and then pass the job on to an automated production facility. During requirements engineering, an analysis class called **PrintJob** was derived. The attributes and operations defined during analysis are noted at the top left of Figure 11.1. During architectural design, **PrintJob** is defined as a component within the software architecture and is represented using the shorthand UML notation shown in the middle right of the figure. Note that **PrintJob** has two interfaces, *computeJob,* that provides job costing capability, and *initiateJob,* that passes the job along to the production facility. These are represented using the "lollipop" symbols shown to the left of the component box.

ADVICE

Recall that analysis modeling and design modeling are both iterative actions. Elaborating the original analysis class may require additional analysis steps, which are then followed with design modeling steps to represent the elaborated design class (the details of the component).

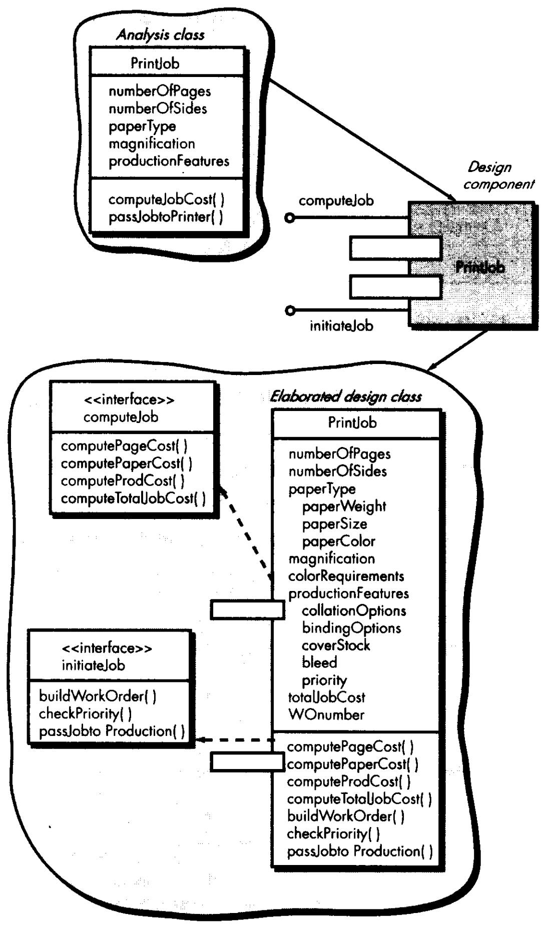
Component-level design begins at this point. The details of the component **PrintJob** must be elaborated to provide sufficient information to guide implementation. The original analysis class is elaborated to flesh out all attributes and operations required to implement the class as the component **PrintJob.** Referring to the lower right portion of Figure 11.1, the elaborated design class **PrintJob** contains more detailed attribute information as well as an expanded description of operations required to implement the component. The interfaces *computeJob* and *initiateJob* imply communication and collaboration with other components (not shown here). For example, the operation *computePageCost()* (part of the *computeJob* interface) might collaborate with a **PricingTable** component that contains job pricing information. The *checkPriority()* operation (part of the *initiateJob* interface) might collaborate with a **JobQueue** component to determine the types and priorities of jobs currently awaiting production.

This elaboration activity is applied to every component defined as part of the architectural design. Once it is completed, further elaboration is applied to each attribute, operation, and interface. The data structures appropriate for each attribute must be specified. In addition, the algorithmic detail required to implement the processing logic associated with each operation is designed. This procedural design activity is discussed later in this chapter. Finally, the mechanisms required to implement the interface are designed. For OO software, this may encompass the description of all messaging that is required to effect communication between objects within the system.

1 In some cases, a component may contain a single class.

FIGURE 11.1

Elaboration of a design component



11.1.2 The Conventional View

In the context of conventional software engineering, a component is a functional element of a program that incorporates processing logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it. A conventional component, also called a *module,* resides within the software architecture and serves one of three important roles as: (1) a *control component* that coordinates the invocation of all other problem domain components, (2) a *problem domain component* that implements a complete or partial function that is required by the customer, or (3) an *infrastructure component* that is responsible for functions that support the processing required in the problem domain.

Like object-oriented components, conventional software components are derived from the analysis model. In this case, however, the data flow-oriented element of the analysis model serves as the basis for the derivation. Each transform (bubble) represented at the lowest levels of the data flow diagram (Chapter 8) is mapped (Section 10.6) into a module hierarchy. Control components (modules) reside near the top of the hierarchy (architecture), and problem domain components tend to reside toward the bottom of the hierarchy. To achieve effective modularity, design concepts like functional independence (Chapter 9) are applied as components are elaborated.

"A complex system that works is invariably found to have evolved from a simple system that worked."

John Gall

To illustrate this process of design elaboration for conventional components, we again consider software to be built for a sophisticated photocopying center. A set of data flow diagrams would be derived during analysis modeling. We'll assume that these are mapped (Section 10.6) into an architecture shown in Figure 11.2. Each box represents a software component. Note that the shaded boxes are equivalent in function to the operations defined for the **PrintJob** class discussed in Section 11.1.1. In this case, however, each operation is represented as a separate module that is invoked as shown in the figure. Other modules are used to control processing and are therefore control components.

During component-level design, each module in Figure 11.2 is elaborated. The module interface is defined explicitly. That is, each data or control object that flows

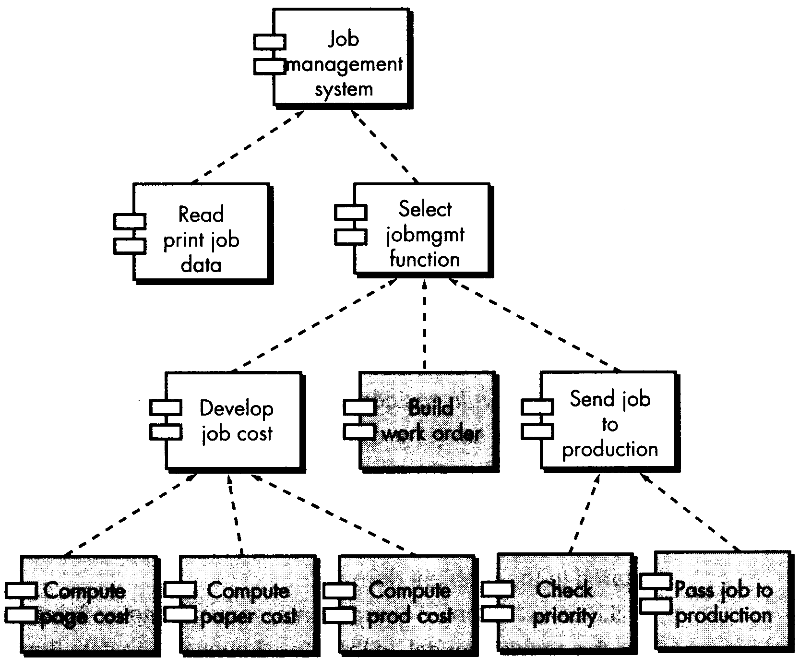


FIGURE 11.2

Structure chart for a conventional system

across the interface is represented. The data structures that are used internal to the module are defined. The algorithm that allows the module to accomplish its intended function is designed using the stepwise refinement approach discussed in Chapter 9. The behavior of the module is sometimes represented using a state diagram.

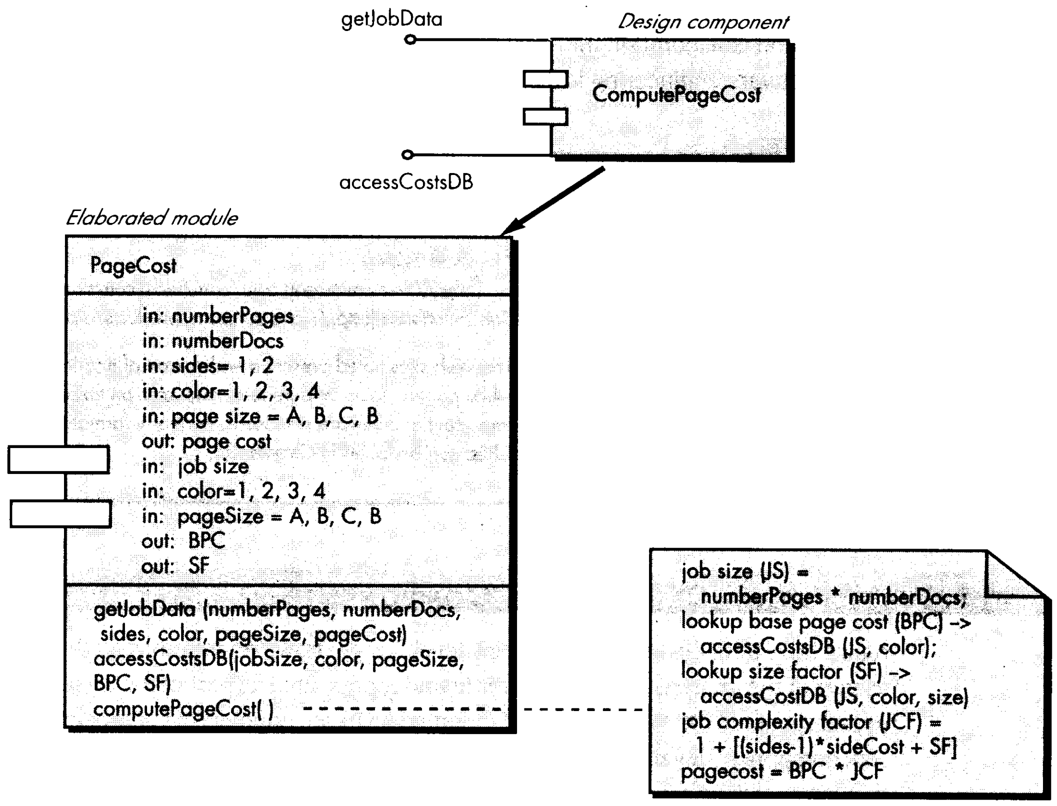
ADVICE

As the design for each software component is elaborated, the focus shifts to the design of specific data structures and procedural design to manipulate the data structures. However, don't forget the architecture that must house the components or the global data structures that may serve many components.

To illustrate this process, consider the module *ComputePageCost.* The intent of this module is to compute the printing cost per page based on specifications provided by the customer. Data required to perform this function are: **number of pages in the document, total number of documents to be produced, one- or two-side printing, color requirements, size requirements.** These data are passed to *ComputePageCost* via the module's interface. *ComputePageCost* uses these data to determine a page cost that is based on the size and complexity of the job—a function of all data passed to the module via the interface. Page cost is inversely proportional to the size of the job and directly proportional to the complexity of the job.

Figure 11.3 represents the component-level design using a modified UML notation. The *ComputePageCost* module accesses data by invoking the modules *getJob-Data,* which allows all relevant data to be passed to the component, and a database

FIGURE 11.3 **Component-level design for *ComputePageCost***


getJobData
Design component
ComputePageCost
accessCostsDB
Elaborated module
PageCost
in: numberPages
in: numberDocs
in: sides= 1, 2
in: color=1, 2, 3, 4
in: page size = A, B, C, B
out: page cost
in: job size
in: color=1, 2, 3, 4
in: pageSize = A, B, C, B
out: BPC
out: SF
getJobData (numberPages, numberDocs, sides, color, pageSize, pageCost)
accessCostsDB(jobSize, color, pageSize, BPC, SF)
computePageCost( )
job size (JS) = numberPages * numberDocs;
lookup base page cost (BPC) -> accessCostsDB (JS, color);
lookup size factor (SF) -> accessCostDB (JS, color, size)
job complexity factor (JCF) = 1 + [(sides-1)*sideCost + SF]
pagecost = BPC * JCF

interface, *accessCostsDB,* which enables the module to access a database that contains all printing costs. As design continues, the *ComputePageCost* module is elaborated to provide algorithm and interface detail (Figure 11.3). Algorithm detail can be represented using the pseudocode text shown in the figure or with a UML activity diagram. The interfaces are represented as a collection of input and output data objects or items. Design elaboration continues until sufficient detail is provided to guide construction of the component.

11.1.3 A Process-Related View

The object-oriented and conventional views of component-level design presented in the preceding sections assume that the component is being designed from scratch. That is, the designer must create a new component based on specifications derived from the analysis model. There is, of course, another approach.

Over the past decade, the software engineering community has emphasized the need to build systems that make use of existing software components. In essence, a catalog of proven design or code-level components is made available to the software engineer as design work proceeds. As the software architecture is developed, components or design patterns are chosen from the catalog and used to populate the architecture. Because these components have been created with reusability in mind, a complete description of their interface, the function(s) they perform, and the communication and collaboration they require are all available to the designer. Component-based software engineering is discussed in considerable detail in Chapter 30.

SOFTWARE TOOLS

Middleware and Component-Based Software Engineering

One of the key elements that leads to the success or failure of CBSE is the availability of middleware. *Middleware* is a collection of infrastructure components that enable problem domain components to communicate with one another across a network or within a complex system. Three competing standards are available to software engineers who want to use component-based software engineering as their software process:

OMG CORBA (http://www.corba.org/).
Microsoft COM
(http://www.microsoft.com/com/tech/complus.asp).
Sun JavaBeans (http://java.sun.com/products/ejb/).

The Web sites noted present a wide array of tutorials, white papers, tools, and general resources on these important middleware standards. Further information on CBSE can be found in Chapter 30.

11.2 Designing Class-Based Components

As we have already noted, component-level design draws on information developed as part of the analysis model (Chapter 8) and represented as part of the architectural model (Chapter 10). When an object-oriented software engineering approach is chosen, component-level design focuses on the elaboration of analysis classes (problem domain specific classes), and the definition and refinement of infrastructure classes.

The detailed description of the attributes, operations, and interfaces used by these classes is the design detail required as a precursor to the construction activity.

11.2.1 Basic Design Principles

Four basic design principles are applicable to component-level design and have been widely adopted when object-oriented software engineering is applied. The underlying motivation for the application of these principles is to create designs that are more amenable to change and to reduce the propagation of side effects when changes do occur. These principles can be used to guide the designer as each software component is developed.

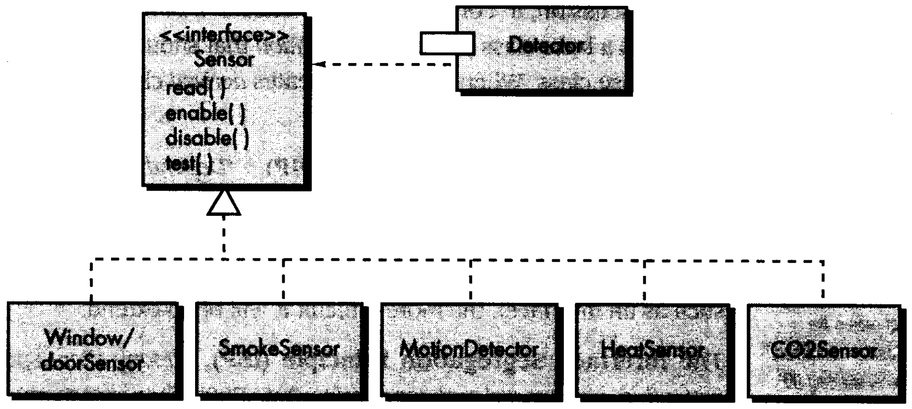
The Open-Closed Principle (OCP). *"A module [component] should be open for extension but closed for modification"* [MAR00]. This statement seems to be a contradiction, but it represents one of the most important characteristics of a good component-level design. Stated simply, the designer should specify the component in a way that allows it to be extended (within the functional domain that it addresses) without the need to make internal (code or logic-level) modifications to the component itself. To accomplish this, the designer creates abstractions that serve as a buffer between the functionality that is likely to be extended and the design class itself.

For example, assume that the *SafeHome* security function makes use of a **Detector** class that must check the status of each type of security sensor. It is likely that as time passes, the number and types of security sensors will grow. If internal processing logic is implemented as a sequence of if-then-else constructs, each addressing a different sensor type, the addition of a new sensor type will require additional internal processing logic (still another if-then-else). This is a violation of OCP.

One way to accomplish OCP for the **Detector** class is illustrated in Figure 11.4. The *sensor* interface presents a consistent view of sensors to the **Detector** component. If a new type of sensor is added no change is required for the **Detector** class (component). The OCP is preserved.

FIGURE 11.4

Following the OCP



SafeHome

The OCP in Action

The scene: Vinod's cubicle.

The players: Vinod and Shakira—members of the SafeHome software engineering team.

The conversation:

Vinod: I just got a call from Doug [the team manager]. He says marketing wants to add a new sensor.

Shakira (smirking): Not again, jeez!

Vinod: Yeah... and you're not going to believe what these guys have come up with.

Shakira: Amaze me.

Vinod (laughing): They call it a doggie angst sensor.

Shakira: Say what?

Vinod: It's for people who leave their pets home in apartments or condos or houses that are close to one another. The dog starts to bark. The neighbor gets angry and complains. With this sensor, if the dog barks for more than, say, a minute, the sensor sets a special alarm mode that calls the owner on his or her cell phone.

Shakira: You're kidding me, right?

Vinod: Nope. Doug wants to know how much time it's going to take to add it to the security function.

Shakira (thinking a moment): Not much ... look. [She shows Vinod Figure 11.4] We've isolated the actual sensor classes behind the **sensor** interface. As long as we have specs for the doggie sensor, adding it should be a piece of cake. Only thing I'll have to do is create an appropriate component . . . uh, class, for it. No change to the **Detector** component at all.

Vinod: So I'll tell Doug it's no big deal.

Shakira: Knowing Doug, he'll keep us focused and not deliver the doggie thing until the next release.

Vinod: That's not a bad thing, but can you implement now if he wants you to?

Shakira: Yeah, the way we designed the interface lets me do it with no hassle.

Vinod (thinking a moment): Have you ever heard of the "Open-Closed Principle"?

Shakira (shrugging): Never heard of it.

Vinod (smiling): Not a problem.

The Liskov Substitution Principle (LSP). *"Subclasses should be substitutable for their base classes"* [MAR00]. This design principle, originally proposed by Barbara Liskov [LIS88] suggests that a component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead. LSP demands that any class derived from a base class must honor any implied contract between the base class and the components that use it. In the context of this discussion, a "contract" is a *precondition* that must be true before the component uses a base class and a *post-condition* that should be true after the component uses a base class. When a designer creates *derived* classes, they must also conform to the pre- and post-conditions.

ADVICE

If you dispense with design and hack out code, just remember that code is the ultimate "concretion." You're violating DIP.

Dependency Inversion Principle (DIP). *"Depend on abstractions. Do not depend on concretions"* [MAR00]. As we have seen in the discussion of the OCP, abstractions are the place where a design can be extended without great complication. The more a component depends on other concrete components (rather than on abstractions such as an interface), the more difficult it will be to extend.

The Interface Segregation Principle (ISP). *"Many client-specific interfaces are better than one general purpose interface"*[MAR00]. There are many instances in which multiple client components use the operations provided by a server class. ISP suggests that the designer should create a specialized interface to serve each major category of clients. Only those operations that are relevant to a particular category of clients should be specified in the interface for that client. If multiple clients require the same operations, they should be specified in each of the specialized interfaces.

As an example, consider the **FloorPlan** class that is used for the *SafeHome* security and surveillance functions. For the security functions, **FloorPlan** is used only during configuration activities and uses the operations *placeDevice(), showDevice(), groupDevice(),* and *removeDevice()* to place, show, group, and remove sensors from the floor plan. The *SafeHome* surveillance function uses the four operations noted for security, but also requires special operations to manage cameras: *showFOV()* and *showDeviceID().* Hence, ISP suggests that client components from the two *SafeHome* functions have specialized interfaces defined for them. The interface for security would encompass only the operations *placeDevice(), showDevice(), groupDevice(),* and *removeDevice().* The interface for surveillance would incorporate the operations *placeDevice(), showDevice(), groupDevice(),* and *removeDevice(), showFOV(),* and *showDeviceID().*

Although component-level design principles provide useful guidance, components themselves do not exist in a vacuum. In many cases, individual components or classes are organized into subsystems or packages. It is reasonable to ask how this packaging activity should occur. Exactly how should components be organized as the design proceeds? Martin [MAR00] suggests additional packaging principles that are applicable to component-level design:

KEY POINT

Designing components for reuse requires more than good technical design. It also requires effective configuration control mechanisms (Chapter 27).

The Release Reuse Equivalency Principle (REP). *"The granule of reuse is the granule of release"* [MAR00]. When classes or components are designed for reuse, there is an implicit contract that is established between the developer of the reusable entity and the people who will use it. The developer commits to establish a release control system that supports and maintains older versions of the entity while the users slowly upgrade to the most current version. Rather than addressing each class individually, it is often advisable to group reusable classes into packages that can be managed and controlled as newer versions evolve.

The Common Closure Principle (CCP). *"Classes that change together belong together"* [MAR00]. Classes should be packaged cohesively. That is, when classes are packaged as part of a design, they should address the same functional or behavioral area. When some characteristic of that area must change, it is likely that only those classes within the package will require modification. This leads to more effective change control and release management.

The Common Reuse Principle (CRP). *"Classes that aren't reused together should not be grouped together"* [MAR00]. When one or more classes with a package changes, the release number of the package changes. All other classes or packages that rely on the package that has been changed must now update to the most recent release of the package and be tested to ensure that the new release operates without incident. If classes are not grouped cohesively, it is possible that a class with no relationship to other classes within a package is changed. This will precipitate unnecessary integration and testing. For this reason, only classes that are reused together should be included within a package.

11.2.2 Component-Level Design Guidelines

In addition to the principles discussed in Section 11.2.1, a set of pragmatic design guidelines can be applied as component-level design proceeds. These guidelines apply to components, their interfaces, and the dependencies and inheritance characteristics that have an impact on the resultant design. Ambler [AMB02] suggests the following guidelines:

What should we consider when we name components?

Components. Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model. Architectural component names should be drawn from the problem domain and should have meaning to all stakeholders who view the architectural model. For example, the class name **FloorPlan** is meaningful to everyone reading it regardless of technical background. On the other-hand, infrastructure components or elaborated component-level classes should be named to reflect implementation-specific meaning. If a linked list is to be managed as part of the **FloorPlan** implementation, the operation *manageList()* is appropriate, even if a nontechnical person might misinterpret it.[2]

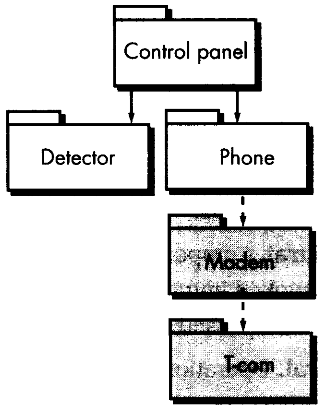
It is also worthwhile to use stereotypes to help identify the nature of components at the detailed design level. For example, <<infrastructure>> might be used to identify an infrastructure component; <<database>> could be used to identify a database that services one or more design classes or the entire system; <<table>> could be used to identify a table within a database.

Interfaces. Interfaces provide important information about communication and collaboration (as well as helping us to achieve the OCP). However, unfettered representation of interfaces tends to complicate component diagrams. Ambler [AMB02] recommends that (1) lollipop representation of an interface should be used in lieu of the more formal UML box and dashed arrow approach, when diagrams grow complex; (2) for consistency, interfaces should flow from the left-hand side of the component box; (3) only those interfaces that are relevant to the component under consideration should be shown, even if other interfaces are available. These recommendations are intended to simplify the visual nature of UML component diagrams.

Dependencies and inheritance. For improved readability, it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes)

2 It is unlikely that someone from marketing or the customer organization (a nontechnical type) would examine detailed design information.

FIGURE 11.5
Layer cohesion



to top (base classes). In addition, component interdependencies should be represented via interfaces, rather than by representation of a component-to-component dependency. Following the philosophy of the OCP, this will help to make the system more maintainable.

11.2.3 Cohesion

In Chapter 9, we described cohesion as the "single-mindedness" of a component. Within the context of component-level design for object-oriented systems, *cohesion* implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself. Lethbridge and Laganiére [LET01] define a number of different types of cohesion (listed in order of the level of the cohesion[3]):

Functional. Exhibited primarily by operations, this level of cohesion occurs when a module performs one and only one computation and then returns a result.

ADVICE

Although an understanding of the various levels of cohesion is instructive, it is more important to be aware of the general concept as you design components. Keep cohesion as high as is possible.

Layer. Exhibited by packages, components, and classes, this type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers. Consider for example, the *SafeHome* security function requirement to make an outgoing phone call if an alarm is sensed. It might be possible to define a set of layered packages as shown in Figure 11.5. The shaded packages contain infrastructure components. Access is from the control panel package downward.

Communicational. All operations that access the same data are defined within one class. In general, such classes focus solely on the data in question, accessing and storing it.

3 In general, the higher the level of cohesion, the easier the component is to implement, test, and maintain.

Classes and components that exhibit functional, layer, and communicational cohesion are relatively easy to implement, test, and maintain. The designer should strive to achieve these levels of cohesion. However, there are many instances when the following lower levels of cohesion are encountered:

Sequential. Components or operations are grouped in a manner that allows the first to provide input to the next and so on. The intent is to implement a sequence of operations.

Procedural. Components or operations are grouped in a manner that allows one to be invoked immediately after the preceding one was invoked, even when there is no data passed between them.

Temporal. Operations that are performed to reflect a specific behavior or state, e.g., an operation performed at start-up or all operations performed when an error is detected.

Utility. Components, classes, or operations that exist within the same category but are otherwise unrelated are grouped together. For example, a class called **Statistics** exhibits utility cohesion if it contains all attributes and operations required to compute six simple statistical measures.

These levels of cohesion are less desirable and should be avoided when design alternatives exist. It is important to note, however, that pragmatic design and implementation issues sometimes force a designer to opt for lower levels of cohesion.

SAFEHOME

Cohesion in Action

The scene: Jamie's cubicle.

The players: Jamie and Ed—members of the *SafeHome* software engineering team who are working on the surveillance function.

The conversation:

Ed: I have a first-cut design of the camera component.

Jamie: Wanna do a quick review?

Ed: I guess . . . but really, I'd like your input on something.

(Jamie gestures for him to continue.)

Ed: We originally defined five operations for **camera.** Look . . . [shows Jamie the list]

determineType() tells me the type of camera.

translateLocation() allows me to move the camera around the floor plan.

displayID() gets the camera ID and displays it near the camera icon.

displayView() shows me the field of view of the camera graphically.

displayZoom() shows me the magnification of the camera graphically.

Ed: I've designed each separately, and they're pretty simple operations. So I thought it might be a good idea to combine all of the display operations into just one that's called *displayCamera()*—it'll show the ID, the view, and the zoom. Whaddaya think?

Jamie (grimacing): Not sure that's such a good idea.

Ed (frowning): Why? All of these little ops can cause headaches.

Jamie: The problem with combining them is we lose cohesion. You know, the *displayCamera()* op won't be single-minded.

Ed (mildly exasperated): So what? The whole thing will be less than 100 source lines, max. It'll be easier to implement, I think.

Jamie: And what if marketing decides to change the way that we represent the view field?

Ed: I'll just jump into the *displayCamera()* op and make the mod.

Jamie: What about side effects?

Ed: Whaddaya mean?

Jamie: Well, say you make the change but inadvertently create a problem with the ID display.

Ed: I wouldn't be that sloppy.

Jamie: Maybe not, but what if some support person two years from now has to make the mod. He might not understand the op as well as you do and, who knows, he might be sloppy.

Ed: So you're against it?

Jamie: You're the designer . . . it's your decision . . . just be sure you understand the consequences of low cohesion.

Ed (thinking a moment): Maybe we'll go with separate display ops.

Jamie: Good decision.

11.2.4 Coupling

In earlier discussions of analysis and design, we noted that communication and collaboration are essential elements of any object-oriented system. There is, however, a darker side to this important (and necessary) characteristic. As the amount of communication and collaboration increases (i.e., as the degree of "connectedness" between classes grows), the complexity of the system also increases. And as complexity rises, the difficulty of implementing, testing, and maintaining software also increases.

Coupling is a qualitative measure of the degree to which classes are connected to one another. As classes (and components) become more interdependent, coupling increases. An important objective in component-level design is to keep coupling as low as is possible.

Class coupling can manifest itself in a variety of ways. Lethbridge and Laganiére [LET01] define the following coupling categories:

ADVICE

As the design for each software component is elaborated, the focus shifts to the design of specific data structures and procedural designs to manipulate the data structures. However, don't forget the architecture that must house the components or the global data structures that may serve many components.

Content coupling. Occurs when one component "surreptitiously modifies data that is internal to another component" [LET01]. This violates information hiding—a basic design concept.

Common coupling. Occurs when a number of components all make use of a global variable. Although this is sometimes necessary (e.g., for establishing default values that are applicable throughout an application), common coupling can lead to uncontrolled error propagation and unforeseen side effects when changes are made.

Control coupling. Occurs when *operation A()* invokes *operation B()* and passes a control flag to *B*. The control flag then "directs" logical flow within *B*. The problem with this form of coupling is that an unrelated change in *B* can result in the necessity to change the meaning of the control flag that *A* passes. If this is overlooked, an error will result.

Stamp coupling. Occurs when **ClassB** is declared as a type for an argument of an operation of **ClassA.** Because **ClassB** is now a part of the definition of **ClassA,** modifying the system becomes more complex.

Data coupling. Occurs when operations pass long strings of data arguments. The "bandwidth" of communication between classes and components grows and the complexity of the interface increases. Testing and maintenance are more difficult.

Routine call coupling. Occurs when one operation invokes another. This level of coupling is common and is often quite necessary. However, it does increase the connectedness of a system.

Type use coupling. Occurs when component **A** uses a data type defined in component **B** (e.g., this occurs whenever "a class declares an instance variable or a local variable as having another class for its type" [LET01]). If the type definition changes, every component that uses the definition must also change.

Inclusion or import coupling. Occurs when component **A** imports or includes a package or the content of component **B.**

External coupling. Occurs when a component communicates or collaborates with infrastructure components (e.g., operating system functions, database capability, telecommunication functions). Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system.

Software must communicate internally and externally. Therefore, coupling is a fact of life. However, the designer should work to reduce coupling whenever possible and understand the ramifications of high coupling when it cannot be avoided.

SAFEHOME

Coupling in Action

The scene: Shakira's cubicle.

The players: Vinod and Shakira—members of the SafeHome software engineering team who are working on the security function.

The conversation:

Shakira: I had what I thought was a great idea . . . then I thought about it a little, and it seemed like a not-so-great idea. I finally rejected it, but I just thought I'd run it by you.

Vinod: Sure, what's the idea?

Shakira: Well, each of the sensors recognizes an alarm condition of some kind, right?

Vinod (smiling): That's why we call them sensors, Shakira.

Shakira (exasperated): Sarcasm, Vinod. You've got to work on your interpersonal skills.

Vinod: You were saying?

Shakira: Okay, anyway, I figured . . . why not create an operation within each sensor object called makeCall() that would collaborate directly with the **OutgoingCall** component, well, with an interface to the **OutgoingCall** component.

Vinod (pensive): You mean rather than having that collaboration occur out of a component like **ControlPanel** or something?

Shakira: Yeah . . . but then I said to myself, that means that every sensor object will be connected to the **OutgoingCall** component, and that means that it's indirectly coupled to the outside world and . . . well, I just thought it made things complicated.

Vinod: I agree. In this case, it's a better idea to let the sensor interface pass info to the **ControlPanel** and let it initiate the outgoing call. Besides, different sensors might result in different phone numbers. You don't want the sensor to store that information because if it changes . . .

Shakira: It just didn't feel right.

Vinod: Design heuristics for coupling tell us [illegible]

Shakira: Whatever. . .

11.3 Conducting Component-Level Design

Earlier in this chapter we noted that component-level design is elaborative in nature. The designer must transform information from the analysis and architectural models into a design representation that provides sufficient detail to guide the construction (coding and testing) activity. The following steps represent a typical task set for component-level design, when it is applied for an object-oriented system.

Step 1. Identify all design classes that correspond to the problem domain. Using the analysis and architectural models, each analysis class and architectural component is elaborated as described in Section 11.1.1.

ADVICE

If you're working in a non-OO environment, the first three steps focus On refinement of data objects and processing functions (transforms) identified as part of the analysis model.

Step 2. Identify all design classes that correspond to the infrastructure domain. These classes are not described in the analysis model and are often missing from the architecture model, but they must be described at this point. As we have noted earlier, classes and components in this category include GUI components, operating system components, object and data management components, and others.

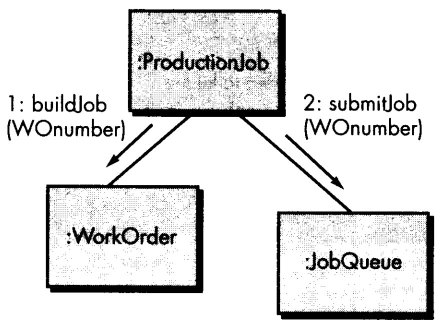
Step 3. Elaborate all design classes that are not acquired as reusable components. Elaboration requires that all interfaces, attributes, and operations necessary to implement the class be described in detail. Design heuristics (e.g., component cohesion and coupling) must be considered as this task is conducted.

Step 3a. Specify message details when classes or components collaborate. The analysis model makes use of a collaboration diagram to show how analysis classes collaborate with one another. As component-level design proceeds, it is sometimes useful to show the details of these collaborations by specifying the structure of messages that are passed between objects within a system. Although this design activity is optional, it can be used as a precursor to the specification of interfaces that show how components within the system communicate and collaborate.

Figure 11.6 illustrates a simple collaboration diagram for the printing system discussed earlier. Three objects, **ProductionJob, WorkOrder,** and **JobQueue,** collaborate to prepare a print job for submission to the production stream. Messages are passed between objects as illustrated by the arrows in the figure. During analysis modeling the messages are specified as shown in the figure. However, as design

FIGURE 11.6

Collaboration diagram with messaging



proceeds, each message is elaborated by expanding its syntax in the following manner [BEN02]:

> **[guard condition] sequence expression (return value) :=**
> **message name (argument list)**

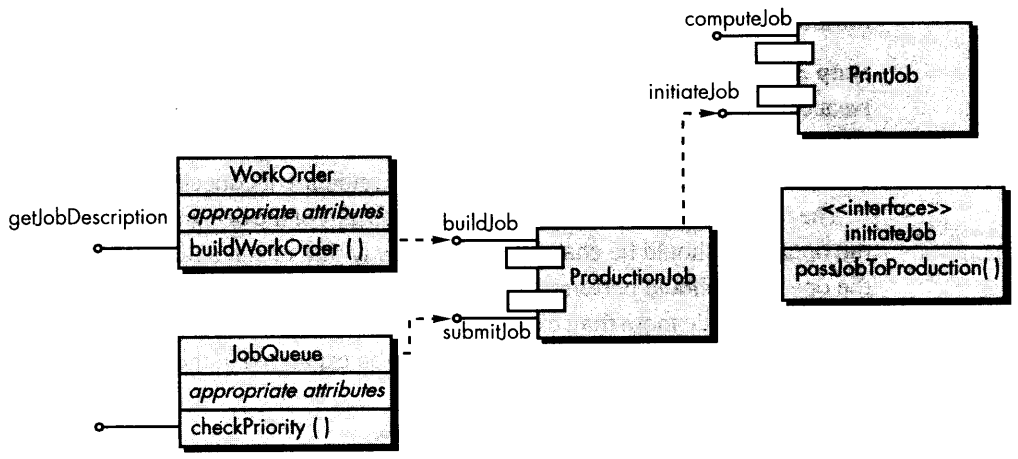
where a **[guard condition]** is written in Object Constraint Language (OCL)[4] and specifies any set of conditions that must be met before the message can be sent; **sequence expression** is an integer value (or other ordering indicator, e.g., 3.1.2) that indicates the sequential order in which a message is sent; **(return value)** is the name of the information that is returned by the operation invoked by the message; **message name** identifies the operation that is to be invoked, and **(argument list)** is the list of attributes that are passed to the operation.

Step 3b. Identify appropriate interfaces for each component. Within the context of component-level design, a UML interface is "a group of externally visible (i.e., public) operations. The interface contains no internal structure, it has no attributes, no associations. . . ." [BEB02]. Stated more formally, an interface is the equivalent of an abstract class that provides a controlled connection between design classes. The elaboration of interfaces is illustrated in Figure 11.1. In essence, operations defined for the design class are categorized into one or more abstract classes. Every operation within the abstract class (the interface) should be cohesive; that is, it should exhibit processing that focuses on one limited function or subfunction.

Referring to Figure 11.1, it can be argued that the interface *initiateJob* does not exhibit sufficient cohesion. In actuality, it performs three different subfunctions: building a work order, checking job priority, and passing a job to production. The interface design should be refactored. One approach might be to reexamine the design classes and define a new class **WorkOrder** that would take care of all activities associated with the assembly of a work order. The operation *buildWorkOrder()* becomes a part

4 OCL is discussed briefly in Section 11.4 and in Chapter 28.

FIGURE 11.7 Refactoring interfaces and class definitions for PrintJob



of that class. Similarly, we might define a class **JobQueue** that would incorporate the operation *checkPriority()*. A class **ProductionJob** would encompass all information associated with a production job to be passed to the production facility. The interface *initiateJob* would then take the form shown in Figure 11.7. The interface *initiateJob* is now cohesive, focusing on one function. The interfaces associated with **ProductionJob, WorkOrder,** and **JobQueue** are similarly single-minded.

Step 3c. Elaborate attributes and define data types and data structures required to implement them. In general, data structures and types used to describe attributes are defined within the context of the programming language that is to be used for implementation. UML defines an attribute's data type using the following syntax:

```
name : type-expression = initial-value {property string}
```

where **name** is the attribute name and **type expression** is the data type; **initial value** is the value that the attribute takes when an object is created; and **property-string** defines a property or characteristic of the attribute.

During the first component-level design iteration, attributes are normally described by name. Referring once again to Figure 11.1, the attribute list for **PrintJob** lists only the names of the attributes. However, as design elaboration proceeds, each attribute is defined using the UML attribute format noted. For example, **paperType-weight** is defined in the following manner:

```
paperType-weight: string = "A" { contains 1 of 4 values - A, B, C, or D}
```

which defines **paperType-weight** as a string variable initialized to the value A that can take on one of four values from the set {A,B,C,D}.

If an attribute appears repeatedly across a number of design classes, and it has a relatively complex structure, it is best to create a separate class to accommodate the attribute.

Step 3d. Describe processing flow within each operation in detail. This may be accomplished using a programming language-based pseudocode (Section 11.5.5) or with a UML activity diagram. Each software component is elaborated through a number of iterations that apply the stepwise refinement concept (Chapter 9).

The first iteration defines each operation as part of the design class. In every case, the operation should be characterized in a way that ensures high cohesion; that is, the operation should perform a single targeted function or subfunction. The next iteration does little more than expand the operation name. For example, the operation *computePaperCost()* noted in Figure 11.1 can be expanded in the following manner:

```
computePaperCost (weight, size, color): numeric
```

This indicates that *computePaperCost()* requires the attributes **weight**, **size** and **color** as input and returns a value that is numeric (actually a dollar value) as output.

"If I had more time, I would have written a shorter letter."
Blaise Pascal

ADVICE

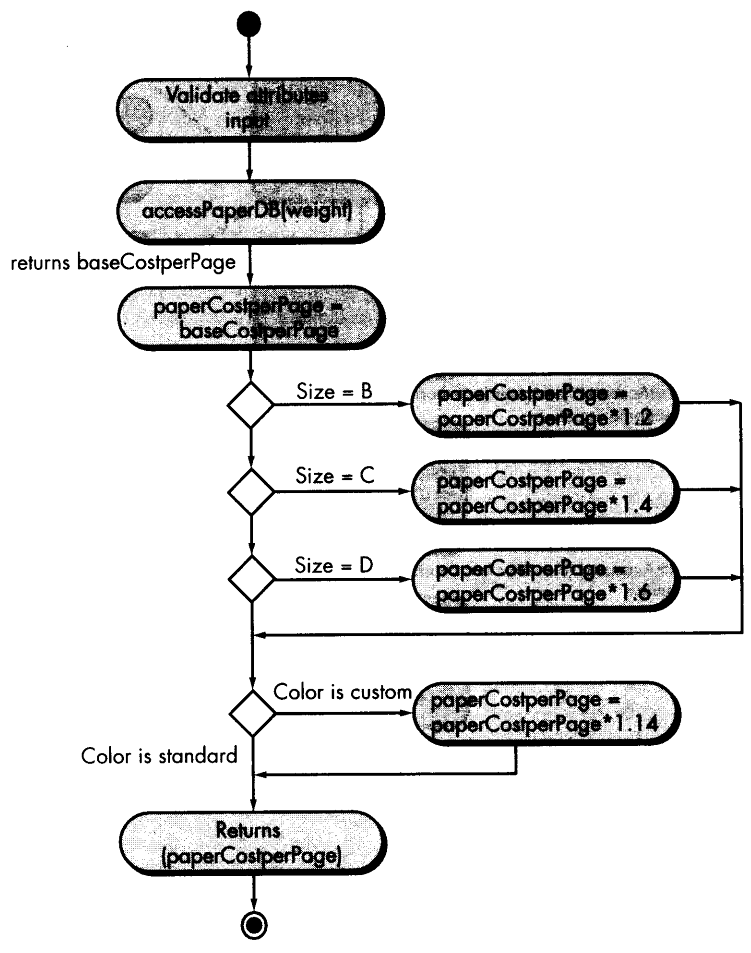
Use stepwise elaboration as you refine the component design. Always ask, "Is there a way this can be simplified and yet still accomplish the same result?"

If the algorithm required to implement *computePaperCost()* is simple and widely understood, no further design elaboration may be necessary. The software engineer who does the coding will provide the detail necessary to implement the operation. However, if the algorithm is more complex or arcane, further design elaboration is required at this stage. Figure 11.8 depicts a UML activity diagram for *computePaperCost()*. When activity diagrams are used for component-level design specification, they are generally represented at a level of abstraction that is somewhat higher than source code. An alternative approach—the use of pseudocode for design specification—is discussed later in this chapter.

Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them. Databases and files normally transcend the design description of an individual component. In most cases, these persistent data stores are initially specified as part of architectural design. However, as design elaboration proceeds, it is often useful to provide additional detail about the structure and organization of these persistent data sources.

Step 5. Develop and elaborate behavioral representations for a class or component. UML state diagrams were used as part of the analysis model to represent the externally observable behavior of the system and the more localized behavior of individual analysis classes. During component-level design, it is sometimes necessary to model the behavior of a design class.

FIGURE 11.8 **UML activity diagram for *computePaperCost()***

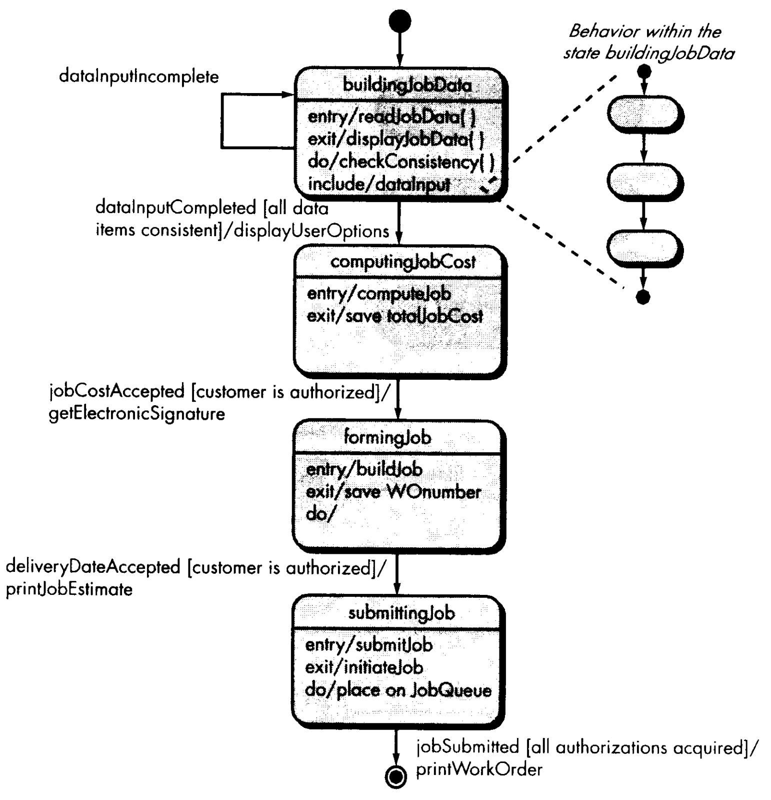

Validate attributes input
accessPaperDB(weight)
returns baseCostperPage
paperCostperPage = baseCostperPage
Size = B
paperCostperPage = paperCostperPage*1.2
Size = C
paperCostperPage = paperCostperPage*1.4
Size = D
paperCostperPage = paperCostperPage*1.6
Color is custom
paperCostperPage = paperCostperPage*1.14
Color is standard
Returns (paperCostperPage)

The dynamic behavior of an object (an instantiation of a design class as the program executes) is affected by events that are external to it and the current state (mode of behavior) of the object. To understand the dynamic behavior of an object, the designer must examine all use-cases that are relevant to the design class throughout its life. These use-cases provide information that helps the designer to delineate the events that affect the object and the states in which the object resides as time passes and events occur. The transitions between states (driven by events) is represented using a UML statechart [BEN02] as illustrated in Figure 11.9.

The transition from one state (represented by a rectangle with rounded corners) to another occurs as a consequence of an event that takes the form:

Event-name (parameter-list) [guard-condition] / action expression

FIGURE 11.9 Statechart fragment for the PrintJob class


dataInputIncomplete
buildingJobData
entry/readJobData( )
exit/displayJobData( )
do/checkConsistency( )
include/dataInput
dataInputCompleted [all data items consistent]/displayUserOptions
computingJobCost
entry/computeJob
exit/save totalJobCost
jobCostAccepted [customer is authorized]/ getElectronicSignature
formingJob
entry/buildJob
exit/save WOnumber
do/
deliveryDateAccepted [customer is authorized]/ printJobEstimate
submittingJob
entry/submitJob
exit/initiateJob
do/place on JobQueue
jobSubmitted [all authorizations acquired]/ printWorkOrder
Behavior within the state buildingJobData

where **event-name** identifies the event; **parameter-list** incorporates data that are associated with the event; **guard-condition** is written in Object Constraint Language (OCL) and specifies a condition that must be met before the event can occur, and **action expression** defines an action that occurs as the transition takes place.

Referring to Figure 11.9, each state may define *entry/* and *exit/* actions that occur as transitions into and out of the state occur. In most cases, these actions correspond to operations that are relevant to the class that is being modeled. The *do/* indicator provides a mechanism for indicating activities that occur while in the state and the *include/* indicator provides a means for elaborating the behavior by embedding more statechart detail within the definition of a state.

It is important to note that the behavioral model often contains information that is not immediately obvious in other design models. For example, careful examination of the statechart in Figure 11.9 indicates that the dynamic behavior of the **PrintJob** class is contingent upon two customer approvals as costs and schedule data for the print job are derived. Without approvals (the guard condition ensures that the customer is authorized to approve) the print job cannot be submitted because there is no way to reach the *submittingJob* state.

Step 6. Elaborate deployment diagrams to provide additional implementation detail. Deployment diagrams (Chapter 9) are used as part of architectural design and are represented in descriptor form. In this form, major system functions (often represented as subsystems) are represented within the context of the computing environment that will house them.

During component-level design, deployment diagrams can be elaborated to represent the location of key packages of components. However, components generally are not represented individually within a component diagram. The reason for this is to avoid diagrammatic complexity. In some cases, deployment diagrams are elaborated into instance form at this time. This means that the specific hardware and operating system environment(s) that will be used is (are) specified and the location of component packages within this environment is indicated.

Step 7. Factor every component-level design representation and always consider alternatives. Throughout this book, we have emphasized that design is an iterative process. The first component-level model you create will not be as complete, consistent, or accurate as the *n*th iteration you apply to the model. It is essential to refactor as design work is conducted.

In addition, a designer should not suffer from tunnel vision. There are always alternative design solutions, and the best designers consider all (or most) of them before settling on the final design model. Develop alternatives and consider each carefully, using the design principles and concepts presented in Chapters 5 and 9 and in this chapter.

11.4 Object Constraint Language

KEY POINT
OCL provides a formal grammar and syntax for describing component-level design elements.

The wide variety of diagrams available as part of UML provide a designer with a rich set of representational forms for the design model. However, graphical representations are often not enough. The designer needs a mechanism for explicitly and formally representing information that constrains some element of the design model. It is possible, of course, to describe constraints in a natural language such as English, but this approach invariably leads to inconsistency and ambiguity. For this reason, a more formal language—one that draws on set theory and formal specification languages (Chapter 28) but has the somewhat less mathematical syntax of a programming language—seems appropriate.

The *Object Constraint Language* (OCL) complements UML by allowing a software engineer to use a formal grammar and syntax to construct unambiguous statements about various design model elements (e.g., classes and objects, events, messages, interfaces). The simplest OCL language statements are constructed in four parts: (1) a *context* that defines the limited situation in which the statement is valid; (2) a *property* that represents some characteristics of the context (e.g., if the context is a class, a property might be an attribute); (3) an *operation* (e.g., arithmetic, set-oriented) that manipulates or qualifies a property; and (4) *keywords* (e.g., **if**, **then**, **else**, **and**, **or**, **not**, **implies**) that are used to specify conditional expressions.

As a simple example of an OCL expression, consider the guard condition placed on the *jobCostAccepted* event that causes a transition between the states **computingJobCost** and **formingJob** within the statechart diagram for the **PrintJob** class (Figure 11.9). In the diagram, the guard condition is expressed in natural language and implies that authorization can only occur if the customer is authorized to approve the cost of the job. In OCL, the expression may take the form:

```
customer
   self.authorizationAuthority = 'yes'
```

where a Boolean attribute, **authorizationAuthority**, of the class (actually a specific instance of the class) named **Customer** must be set to **yes** for the guard condition to be satisfied.

WebRef
The complete OCL specification can be downloaded from **www.omg.org.**

As the design model is created, there are often instances (e.g., Section 11.2.1) in which pre- or post-conditions must be satisfied prior to completion of some action specified by the design. OCL provides a powerful tool for specifying pre- and post conditions in a formal manner. As an example, consider an extension to the print shop system (discussed throughout this chapter) in which the customer provides an upper cost bound for the print job and a "drop-dead" delivery date at the same time as other print job characteristics are specified. If cost and delivery estimates exceed these bounds, the job is not submitted and the customer must be notified. In OCL, a set of pre- and post-conditions may be specified in the following manner:

```
context PrintJob::validate(upperCostBound : Integer, custDeliveryReq :
  Integer)
   pre: upperCostBound > 0
      and custDeliveryReq > 0
      and self.jobAuthorization = 'no'
   post: if self.totalJobCost <= upperCostBound
         and self.deliveryDate <= custDeliveryReq
      then
         self.jobAuthorization = 'yes'
      endif
```

This OCL statement defines an *invariant*—conditions that must exist prior to (**pre**) and after (**post**) some behavior. Initially, a precondition establishes that bounding cost and delivery date must be specified by the customer, and authorization must be set to "no." After costs and delivery are determined, the post-condition is applied. It should also be noted that the expression: **self.jobAuthorization = 'yes'** is not assigning the value "yes," but is declaring that the **jobAuthorization** must have been set to "yes" by the time the operation finishes.

A complete description of OCL is beyond the scope of this book.[5] Interested readers should see [WAR98] and [OMG01] for additional detail.

SOFTWARE TOOLS

UML/OCL

Objective: A wide variety of UML tools are available to assist the designer at all levels of design. Some of these tools provide OCL support.

Mechanics: Tools in this category enable a designer to create all UML diagrams that are necessary to build a complete design model. More importantly, many tools provide solid syntax and semantic checking, and version and change control management (Chapter 27). When OCL capability is provided, tools enable the designer to create OCL expressions and, in some cases, "compile" them for various types of evaluation and analysis.

Representative Tools[6]

ArgoUML, distributed at Tigress.org (http://argouml.tigris.org/), supports the complete UML and OCL and includes a variety of design assist tools that go beyond the generation of UML diagrams and OCL expressions.

Dresden OCL toolkit, developed by Frank Finger at the Dresden University of Technology (http://dresden-ocl.sourceforge.net/), is a toolkit based on an OCL compiler encompassing several modules which parse, type check, and normalize OCL constraints.

OCL parser, developed by IBM (http://www-3.ibm.com/software/ad/library/standards/ocl-download.html), is written in Java and is available for free to the object-oriented community to encourage the use of OCL with UML modelers.

11.5 Designing Conventional Components

The foundations of component-level design for conventional software components[7] were formed in the early 1960s and were solidified with the work of Edsgar Dijkstra and his colleagues ([BOH66], [DIJ65], [DIJ76]). In the late 1960s, Dijkstra and others proposed the use of a set of constrained logical constructs from which any program could be formed. The constructs emphasized "maintenance of functional domain." That is, each construct had a predictable logical structure, was entered at the top and exited at the bottom, enabling a reader to follow procedural flow more easily.

5 However, further discussion of OCL (presented in the context of formal methods) is presented in Chapter 28.

6 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

7 A conventional software component implements an element of processing that addresses a function or subfunction in the problem domain or some capability in the infrastructure domain. Often called *modules, procedures,* or *subroutines,* conventional components do not encapsulate data in the way that OO components do.

KEY POINT
Structured programming is a design technique that constrains logic flow to three constructs: sequence, condition, and repetition.

The constructs are sequence, condition, and repetition. *Sequence* implements processing steps that are essential in the specification of any algorithm. *Condition* provides the facility for selected processing based on some logical occurrence, and *repetition* allows for looping. These three constructs are fundamental to *structured programming*—an important component-level design technique.

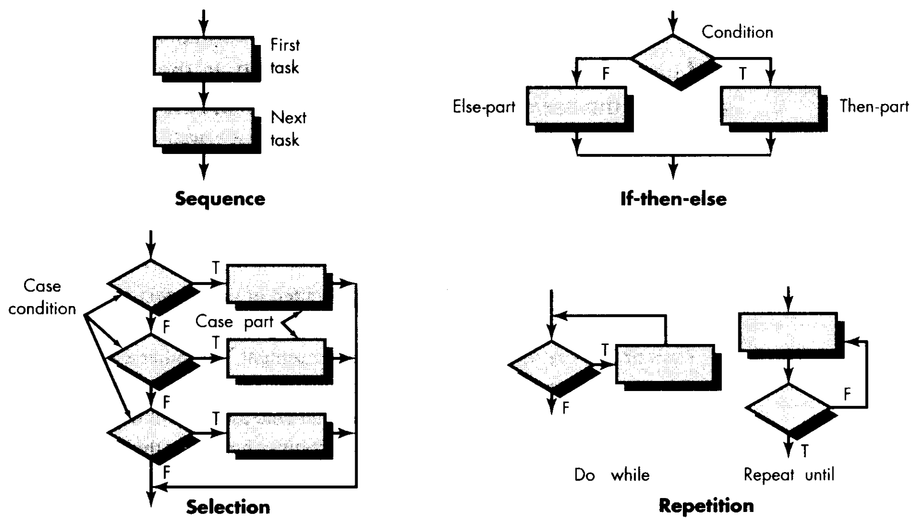
The structured constructs were proposed to limit the procedural design of software to a small number of predictable operations. Complexity metrics (Chapter 15) indicate that the use of the structured constructs reduces program complexity and thereby enhances readability, testability, and maintainability. The use of a limited number of logical constructs also contributes to a human understanding process that psychologists call *chunking*. To understand this process, consider the way in which you are reading this page. You do not read individual letters but rather recognize patterns or chunks of letters that form words or phrases. The structured constructs are logical chunks that allow a reader to recognize procedural elements of a module, rather than reading the design or code line by line. Understanding is enhanced when readily recognizable logical patterns are encountered.

11.5.1 Graphical Design Notation

We have discussed the UML activity diagram earlier in this chapter and in Chapters 7 and 8. The activity diagram allows a designer to represent sequence, condition, and repetition—all elements of structured programming—and is the descendent of an earlier pictorial design representation (still used widely) called a *flowchart*.

A flowchart, like an activity diagram, is quite simple pictorially. A box is used to indicate a processing step. A diamond represents a logical condition, and arrows show the flow of control. Figure 11.10 illustrates three structured constructs. The

FIGURE 11.10
Flowchart constructs



sequence is represented as two processing boxes connected by a line (arrow) of control. *Condition,* also called *if-then-else,* is depicted as a decision diamond that if true, causes *then-part* processing to occur, and if false, invokes *else-part* processing. *Repetition* is represented using two slightly different forms. The *do while* tests a condition and executes a loop task repetitively as long as the condition holds true. A *repeat until* executes the loop task first, then tests a condition and repeats the task until the condition fails. The *selection* (or *select-case*) construct shown in the figure is actually an extension of the *if-then-else.* A parameter is tested by successive decisions until a true condition occurs and a *case part* processing path is executed.

ADVICE

Structured programming constructs should make it easier to understand the design. If using them without "violation" introduces unnecessary complexity, it is okay to violate.

In general, the dogmatic use of only the structured constructs can introduce inefficiency when an escape from a set of nested loops or nested conditions is required. More importantly, additional complication of all logical tests along the path of escape can cloud software control flow, increase the possibility of error, and have a negative impact on readability and maintainability. What can we do?

The designer is left with two options: (1) The procedural representation is redesigned so that the "escape branch" is not required at a nested location in the flow of control or (2) the structured constructs are violated in a controlled manner; that is, a constrained branch out of the nested flow is designed. Option 1 is obviously the ideal approach, but option 2 can be accommodated without violating of the spirit of structured programming.

11.5.2 Tabular Design Notation

ADVICE

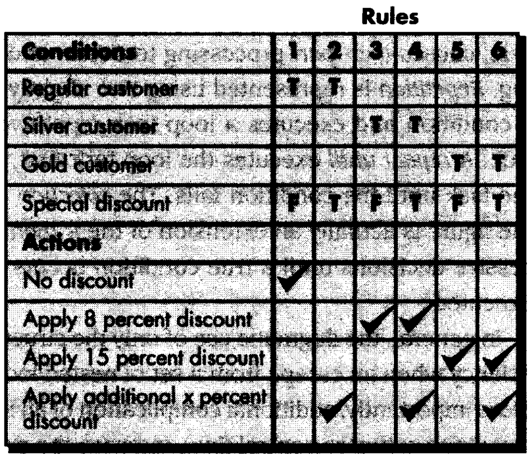
Use a decision table when a complex set of conditions and actions are encountered within a component.

In many software applications, a module may be required to evaluate a complex combination of conditions and select appropriate actions based on these conditions. *Decision tables* [HUR83] provide a notation that translates actions and conditions (described in a processing narrative) into a tabular form. The table is difficult to misinterpret and may even be used as a machine readable input to a table driven algorithm.

A decision table is divided into four quadrants. The upper left-hand quadrant contains a list of all conditions. The lower left-hand quadrant contains a list of all actions that are possible based on combinations of conditions. The right-hand quadrants form a matrix that indicates condition combinations and the corresponding actions that will occur for a specific combination. Therefore, each column of the matrix may be interpreted as a *processing rule.* The following steps are applied to develop a decision table:

? How do I build a decision table?

1. List all actions that can be associated with a specific procedure (or module).
2. List all conditions (or decisions made) during execution of the procedure.
3. Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions.
4. Define rules by indicating what action(s) occurs for a set of conditions.



	Rules					
Conditions	**1**	**2**	**3**	**4**	**5**	**6**
Regular customer	T	T				
Silver customer			T	T		
Gold customer					T	T
Special discount	F	T	F	T	F	T
Actions						
No discount	✓					
Apply 8 percent discount			✓	✓		
Apply 15 percent discount					✓	✓
Apply additional x percent discount		✓		✓		✓

FIGURE 11.11

Resultant decision table

To illustrate the use of a decision table, consider the following excerpt from an informal use-case that has just been proposed for the print shop system:

> Three types of customers are defined: a regular customer, a silver customer, and a gold customer (these types are assigned by the amount of business the customer does with the print shop over a 12-month period). A regular customer receives normal print rates and delivery. A silver customer gets an 8 percent discount on all quotes and is placed ahead of all regular customers in the job queue. A gold customer gets a 15 percent reduction in quoted prices and is placed ahead of both regular and silver customers in the job queue. A special discount of *x* percent in addition to other discounts can be applied to any customer's quote at the discretion of management.

Figure 11.11 illustrates a decision table representation of the preceding informal use-case. Each of the six rules indicates one of six viable conditions. As a general rule, the decision table can be used effectively to supplement other procedural design notation.

11.5.3 Program Design Language

Program design language (PDL), also called *structured English* or *pseudocode,* is "a pidgin language in that it uses the vocabulary of one language (i.e., English) and the overall syntax of another (i.e., a structured programming language)" [CAI75]. In this chapter, PDL is used as a generic reference for a design language.

At first glance PDL may look like a programming language. The difference between PDL and a real programming language lies in the use of narrative text (e.g., English) embedded directly within PDL statements. Given the use of narrative text embedded directly into a syntactical structure, PDL cannot be compiled. However, tools can translate PDL into a programming language "skeleton" and/or a graphical representation (e.g., a flowchart) of design. These tools also produce nesting maps, a design operation index, cross-reference tables, and a variety of other information.