10.4 CLASS-RESPONSIBILITY-COLLABORATOR MODELING

Quote:

"One purpose of CRC cards is to fail early, to fail often, and to fail inexpensively. It is a lot cheaper to tear up a bunch of cards than it would be to reorganize a large amount of source code."

C. Horstmann

Class-responsibility-collaborator (CRC) modeling [Wir90] provides a simple means for identifying and organizing the classes that are relevant to system or product requirements. Ambler [Amb95] describes CRC modeling in the following way:

> A CRC model is really a collection of standard index cards that represent classes. The cards are divided into three sections. Along the top of the card you write the name of the class. In the body of the card you list the class responsibilities on the left and the collaborators on the right.

In reality, the CRC model may make use of actual or virtual index cards. The intent is to develop an organized representation of classes. *Responsibilities* are the attributes and operations that are relevant for the class. Stated simply, a responsibility is "anything the class knows or does" [Amb95]. *Collaborators* are those classes that are required to provide a class with the information needed to complete a responsibility. In general, a *collaboration* implies either a request for information or a request for some action.

WebRef

An excellent discussion of these class types can be found at **http://www.oracle.com/technetwork/developer-tools/jdev/gettingstartedwithumlclassmodeling-130316.pdf.**

A simple CRC index card for the **FloorPlan** class is illustrated in Figure 10.3. The list of responsibilities shown on the CRC card is preliminary and subject to additions or modification. The classes **Wall** and **Camera** are noted next to the responsibility that will require their collaboration.

Classes. Basic guidelines for identifying classes and objects were presented earlier in this chapter. The taxonomy of class types presented in Section 10.1 can be extended by considering the following categories:

- *Entity classes*, also called *model* or *business* classes, are extracted directly from the statement of the problem (e.g., **FloorPlan** and **Sensor**).

FIGURE 10.3
A CRC model index card

Class: FloorPlan	
Description	
Responsibility:	**Collaborator:**
Defines floor plan name/type	
Manages floor plan positioning	
Scales floor plan for display	
Scales floor plan for display	
Incorporates walls, doors, and windows	**Wall**
Shows position of video cameras	**Camera**

These classes typically represent things that are to be stored in a database and persist throughout the duration of the application (unless they are specifically deleted).

- *Boundary classes* are used to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used. Entity objects contain information that is important to users, but they do not display themselves. Boundary classes are designed with the responsibility of managing the way entity objects are represented to users. For example, a boundary class called **CameraWindow** would have the responsibility of displaying surveillance camera output for the *SafeHome* system.
- *Controller classes* manage a "unit of work" from start to finish. That is, controller classes can be designed to manage (1) the creation or update of entity objects, (2) the instantiation of boundary objects as they obtain information from entity objects, (3) complex communication between sets of objects, (4) validation of data communicated between objects or between the user and the application. In general, controller classes are not considered until the design activity has begun.

Quote:

"Objects can be classified scientifically into three major categories: those that don't work, those that break down, and those that get lost."

Russell Baker

What guidelines can be applied for allocating responsibilities to classes?

Responsibilities. Basic guidelines for identifying responsibilities (attributes and operations) have been presented in Sections 10.2 and 10.3. Wirfs-Brock and her colleagues [Wir90] suggest five guidelines for allocating responsibilities to classes:

1. **System intelligence should be distributed across classes to best address the needs of the problem.** Every application encompasses a certain degree of intelligence; that is, what the system knows and what it can do. This intelligence can be distributed across classes in a number of different ways. "Dumb" classes (those that have few responsibilities) can be modeled to act as servants to a few "smart" classes (those having many responsibilities). Although this approach makes the flow of control in a system straightforward, it has a few disadvantages: it concentrates all intelligence within a few classes, making changes more difficult, and it tends to require more classes, hence more development effort.

   If system intelligence is more evenly distributed across the classes in an application, each object knows about and does only a few things (that are generally well focused), the cohesiveness of the system is improved.[3] This enhances the maintainability of the software and reduces the impact of side effects due to change.

3 Cohesiveness is a design concept that is discussed in Chapter 12.

To determine whether system intelligence is properly distributed, the responsibilities noted on each CRC model index card should be evaluated to determine if any class has an extraordinarily long list of responsibilities. This indicates a concentration of intelligence.[4] In addition, the responsibilities for each class should exhibit the same level of abstraction. For example, among the operations listed for an aggregate class called **CheckingAccount** a reviewer notes two responsibilities: *balance-the-account* and *check-off-cleared-checks*. The first operation (responsibility) implies a reasonably complex mathematical and logical procedure. The second is a simple clerical activity. Since these two operations are not at the same level of abstraction, *check-off-cleared-checks* should be placed within the responsibilities of **CheckEntry**, a class that is encompassed by the aggregate class **CheckingAccount**.

2. **Each responsibility should be stated as generally as possible.** This guideline implies that general responsibilities (both attributes and operations) should reside high in the class hierarchy (because they are generic, they will apply to all subclasses).
3. **Information and the behavior related to it should reside within the same class.** This achieves the object-oriented principle called *encapsulation*. Data and the processes that manipulate the data should be packaged as a cohesive unit.
4. **Information about one thing should be localized with a single class, not distributed across multiple classes.** A single class should take on the responsibility for storing and manipulating a specific type of information. This responsibility should not, in general, be shared across a number of classes. If information is distributed, software becomes more difficult to maintain and more challenging to test.
5. **Responsibilities should be shared among related classes, when appropriate.** There are many cases in which a variety of related objects must all exhibit the same behavior at the same time. As an example, consider a video game that must display the following classes: **Player, PlayerBody, PlayerArms, PlayerLegs, PlayerHead.** Each of these classes has its own attributes (e.g., position, orientation, color, speed) and all must be updated and displayed as the user manipulates a joystick. The responsibilities *update* and *display* must therefore be shared by each of the objects noted. **Player** knows when something has changed and *update* is required. It collaborates with the other objects to achieve a new position or orientation, but each object controls its own display.

4 In such cases, it may be necessary to split the class into multiple classes or complete subsystems in order to distribute intelligence more effectively.

Collaborations. Classes fulfill their responsibilities in one of two ways: (1) A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or (2) a class can collaborate with other classes. Wirfs-Brock and her colleagues [Wir90] define collaborations in the following way:

> Collaborations represent requests from a client to a server in fulfillment of a client responsibility. A collaboration is the embodiment of the contract between the client and the server. . . . We say that an object collaborates with another object if, to fulfill a responsibility, it needs to send the other object any messages. A single collaboration flows in one direction—representing a request from the client to the server. From the client's point of view, each of its collaborations is associated with a particular responsibility implemented by the server.

Collaborations are identified by determining whether a class can fulfill each responsibility itself. If it cannot, then it needs to interact with another class. Hence, a collaboration.

As an example, consider the *SafeHome* security function. As part of the activation procedure, the **ControlPanel** object must determine whether any sensors are open. A responsibility named *determine-sensor-status()* is defined. If sensors are open, **ControlPanel** must set a status attribute to "not ready." Sensor information can be acquired from each **Sensor** object. Therefore, the responsibility *determine-sensor-status()* can be fulfilled only if **ControlPanel** works in collaboration with **Sensor**.

To help in the identification of collaborators, you can examine three different generic relationships between classes [Wir90]: (1) the *is-part-of* relationship, (2) the *has-knowledge-of* relationship, and (3) the *depends-upon* relationship. Each of the three generic relationships is considered briefly in the paragraphs that follow.

All classes that are part of an aggregate class are connected to the aggregate class via an *is-part-of* relationship. Consider the classes defined for the video game noted earlier, the class **PlayerBody** *is-part-of* **Player**, as are **PlayerArms**, **PlayerLegs**, and **PlayerHead**. In UML, these relationships are represented as the aggregation shown in Figure 10.4.

When one class must acquire information from another class, the *has-knowledge-of* relationship is established. The *determine-sensor-status()* responsibility noted earlier is an example of a *has-knowledge-of* relationship.

The *depends-upon* relationship implies that two classes have a dependency that is not achieved by *has-knowledge-of* or *is-part-of*. For example, **PlayerHead** must always be connected to **PlayerBody** (unless the video game is particularly violent), yet each object could exist without direct knowledge of the other. An attribute of the **PlayerHead** object called center-position is determined from the center position of **PlayerBody**. This information is obtained via a third object, **Player**, that acquires it from **PlayerBody**. Hence, **PlayerHead** *depends-upon* **PlayerBody**.

FIGURE 1

A compos
aggregat
class

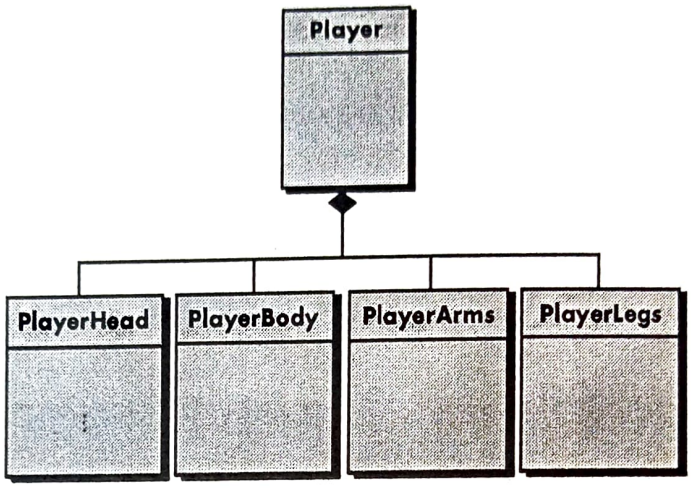


FIGURE 10.4 A composite aggregate class

In all cases, the collaborator class name is recorded on the CRC model index card next to the responsibility that has spawned the collaboration. Therefore, the index card contains a list of responsibilities and the corresponding collaborations that enable the responsibilities to be fulfilled (Figure 10.3).

When a complete CRC model has been developed, the representatives from the stakeholders can review the model using the following approach [Amb95]:

1. All participants in the review (of the CRC model) are given a subset of the CRC model index cards. Cards that collaborate should be separated (i.e., no reviewer should have two cards that collaborate).
2. All use-case scenarios (and corresponding use-case diagrams) should be organized into categories.
3. The review leader reads the use case deliberately. As the review leader comes to a named object, she passes a token to the person holding the corresponding class index card. For example, a use case for *SafeHome* contains the following narrative:

> The homeowner observes the *SafeHome* control panel to determine if the system is ready for input. If the system is not ready, the homeowner must physically close windows/doors so that the ready indicator is present. [A not-ready indicator implies that a sensor is open, i.e., that a door or window is open.]

When the review leader comes to "control panel," in the use case narrative, the token is passed to the person. holding the **ControlPanel** index card. The phrase "implies that a sensor is open" requires that the index card contains a responsibility that will validate this implication (the responsibility *determine-sensor-status()* accomplishes this). Next to the responsibility on the index card is the collaborator **Sensor.** The token is then passed to the **Sensor** object.

4. When the token is passed, the holder of the class card is asked to describe the responsibilities noted on the card. The group determines whether one (or more) of the responsibilities satisfies the use-case requirement.
5. If the responsibilities and collaborations noted on the index cards cannot accommodate the use case, modifications are made to the cards. This may include the definition of new classes (and corresponding CRC index cards) or the specification of new or revised responsibilities or collaborations on existing cards.

This modus operandi continues until the use case is finished. When all use cases (or use case diagrams) have been reviewed, requirements modeling continues.

SafeHome

CRC Models

The scene: Ed's cubicle, as requirements modeling begins.

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

The conversation:

[Vinod has decided to show Ed how to develop CRC cards by showing him an example.]

Vinod: While you've been working on surveillance and Jamie has been tied up with security, I've been working on the home management function.

Ed: What's the status of that? Marketing kept changing its mind.

Vinod: Here's the first-cut use case for the whole function . . . we've refined it a bit, but it should give you an overall view . . .

Use case: *SafeHome* home management function.

Narrative: We want to use the home management interface on a PC or an Internet connection to control electronic devices that have wireless interface controllers. The system should allow me to turn specific lights on and off, to control appliances that are connected to a wireless interface, to set my heating and air-conditioning system to temperatures that I define. To do this, I want to select the devices from a floor plan of the house. Each device must be identified on the floor plan. As an optional feature, I want to control all audiovisual devices—audio, television, DVD, digital recorders, and so forth.

With a single selection, I want to be able to set the entire house for various situations. One is home, another is away, a third is overnight travel, and a fourth is extended travel. All of these situations will have settings that will be applied to all devices. In the overnight travel and extended travel states, the system should turn lights on and off at random intervals (to make it look like someone is home) and control the heating and air-conditioning system. I should be able to override these setting via the Internet with appropriate password protection . . .

Ed: The hardware guys have got all the wireless interfacing figured out?

Vinod (smiling): They're working on it; say it's no problem. Anyway, I extracted a bunch of classes for home management and we can use one as an example. Let's use the **HomeManagementInterface** class.

Ed: Okay . . . so the responsibilities are what . . . the attributes and operations for the class and the collaborations are the classes that the responsibilities point to.

Vinod: I thought you didn't understand CRC.

Ed: Maybe a little, but go ahead.

Vinod: So here's my class definition for **HomeManagementInterface.**

Attributes:

optionsPanel—contains info on buttons that enable user to select functionality.

situationPanel—contains info on buttons that enable user to select situation.

floorplan—same as surveillance object but this one displays devices.

deviceIcons—info on icons representing lights, appliances, HVAC, etc.

devicePanels—simulation of appliance or device control panel; allows control.

13.3 Architectural Styles

Quote:

"There is at the back of every artist's mind, a pattern or type of architecture."

G. K. Chesterton

When a builder uses the phrase "center hall colonial" to describe a house, most people familiar with houses in the United States will be able to conjure a general image of what the house will look like and what the floor plan is likely to be. The builder has used an *architectural style* as a descriptive mechanism to differentiate the house from other styles (e.g., A-frame, raised ranch, Cape Cod). But more important, the architectural style is also a template for construction. Further details of the house must be defined, its final dimensions must be specified, customized features may be added, building materials are to be determined, but the style—a "center hall colonial"—guides the builder in his work.

The software that is built for computer-based systems also exhibits one of many architectural styles. Each style describes a system category that encompasses (1) a set of components (e.g., a database, computational modules) 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 by analyzing the known properties of its constituent parts [Bas03].

? What is an architectural style?

An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system. In the case where an existing architecture is to be reengineered (Chapter 36), the imposition of an architectural style will result in fundamental changes to the structure of the software including a reassignment of the functionality of components [Bos00].

WebRef

Attribute-based architectural styles (ABAS) can be used as building blocks for software architectures. Information can be obtained at **www.sei.cmu.edu/architecture/abas.html.**

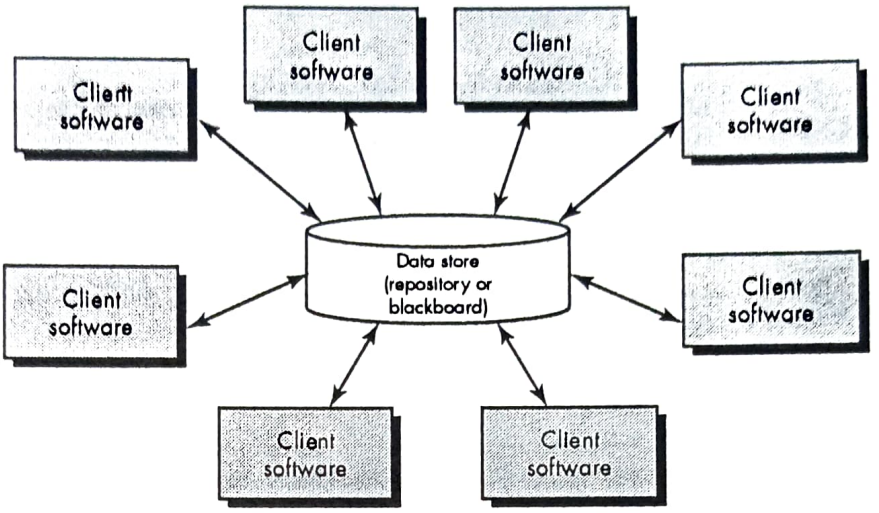
An architectural pattern, like an architectural style, imposes a transformation on the design of an architecture. However, a pattern differs from a style in a number of fundamental ways: (1) the scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety, (2) a pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level (e.g., concurrency) [Bos00], (3) architectural patterns (Section 13.3.2) tend to address specific behavioral issues within the context of the architecture (e.g., how real-time applications handle synchronization or interrupts). Patterns can be used in conjunction with an architectural style to shape the overall structure of a system.

13.3.1 A Brief Taxonomy of Architectural Styles

Although millions of computer-based systems have been created over the past 60 years, the vast majority can be categorized into one of a relatively small number of architectural styles:

Data-Centered Architectures: A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components

FIGURE 13.1
Data-centered architecture



Quote:

"The use of patterns and styles of design is pervasive in engineering disciplines."

Mary Shaw and David Garlan

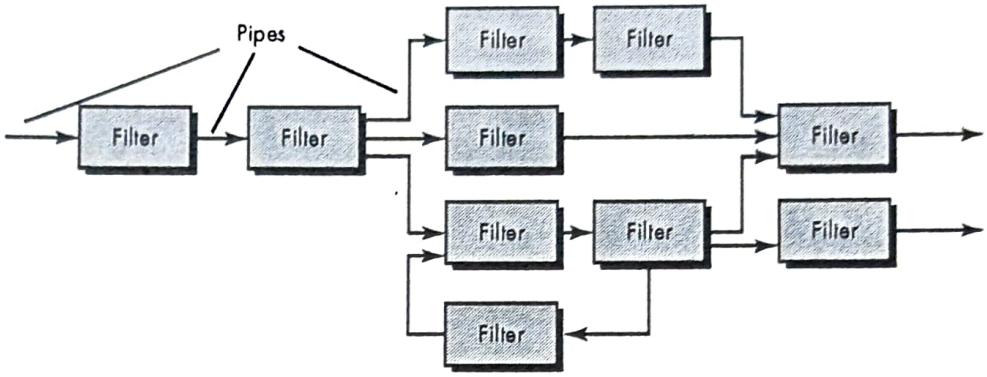
that update, add, delete, or otherwise modify data within the store. Figure 13.1 illustrates a typical data-centered style. Client software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a "blackboard" that sends notifications to client software when data of interest to the client changes.

Data-centered architectures promote *integrability* [Bas03]. That is, existing components can be changed and new client components added to the architecture without concern about other clients (because the client components operate independently). In addition, data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients). Client components independently execute processes.

Data-Flow Architectures This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern (Figure 13.2) has a set of components, called *filters*, connected by *pipes* that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the workings of its neighboring filters.

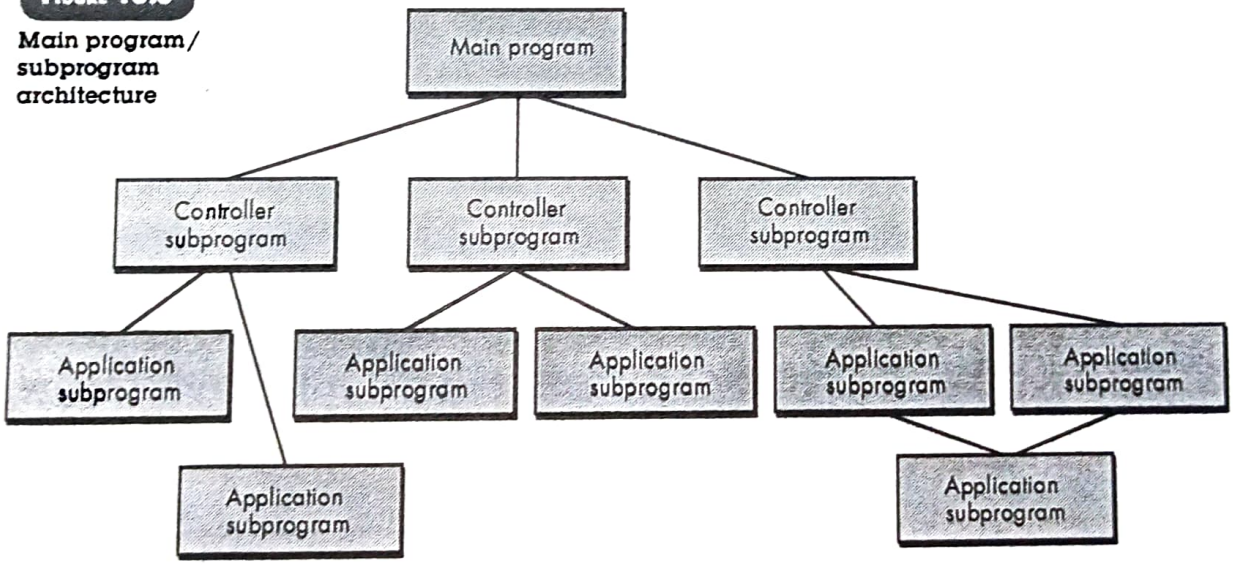
If the data flow degenerates into a single line of transforms, it is termed ***batch sequential.*** This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it.

FIGURE 13.2
Data-flow architecture



Pipes and filters

FIGURE 13.3
Main program/subprogram architecture



Call and Return Architectures This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of substyles [Bas03] exist within this category:

- *Main program/subprogram architectures.* This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components. Figure 13.3 illustrates an architecture of this type.
- *Remote procedure call architectures.* The components of a main program/subprogram architecture are distributed across multiple computers on a network.

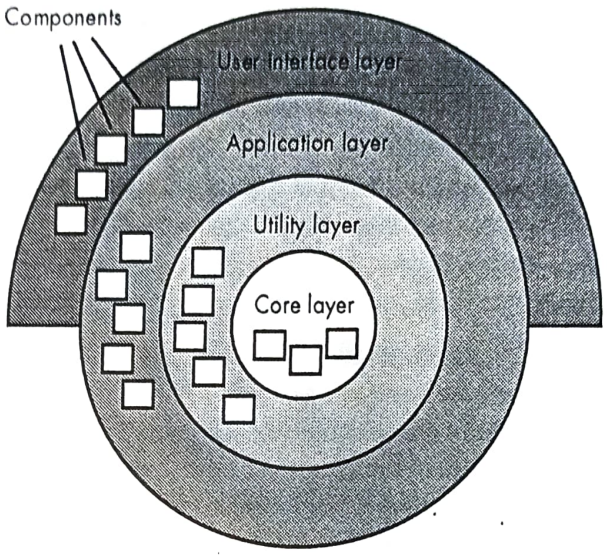


FIGURE 13.4 Layered architecture

Object-Oriented Architectures. The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components are accomplished via message passing.

Layered Architectures. The basic structure of a layered architecture is illustrated in Figure 13.4. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.

These architectural styles are only a small subset of those available.[2] Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style and/or combination of patterns that best fits those characteristics and constraints can be chosen. In many cases, more than one pattern might be appropriate and alternative architectural styles can be designed and evaluated. For example, a layered style (appropriate for most systems) can be combined with a data-centered architecture in many database applications.

Choosing the right architecture style can be tricky. Buschman [Bus10a] suggests two complementary concepts that can provide some guidance. *Problem frames* describe characteristics of recurring problems, without being distracted by references to details of domain knowledge or programming solution implementations. *Domain-driven design* suggests that the software design should

2 See [Roz11], [Tay09], [Bus07], [Gor06], or [Bas03], for a detailed discussion of architectural styles and patterns.

alternatives. Potential alternative solutions (with their pros and cons) from previous software applications are included to assist the architect in making the best decision possible.

The *decision model* documents both the architectural decisions required and records the decisions actually made on previous projects with their justifications. The guidance model feeds the architectural decision model in a *tailoring* step that allows the architect to delete irrelevant issues, enhance important issues, or add new issues. A decision model can make use of more than one guidance model and provides feedback to the guidance model after the project is completed. This feedback may be accomplished by *harvesting* lessons learned from project postmortem reviews.

13.6 Architectural Design

As architectural design begins, context must be established. To accomplish this, the external entities (e.g., other systems, devices, people) that interact with the software and the nature of their interaction are described. This information can generally be acquired from the requirements model. Once context is modeled and all external software interfaces have been described, you can identify a set of architectural archetypes.

? What is an archetype?

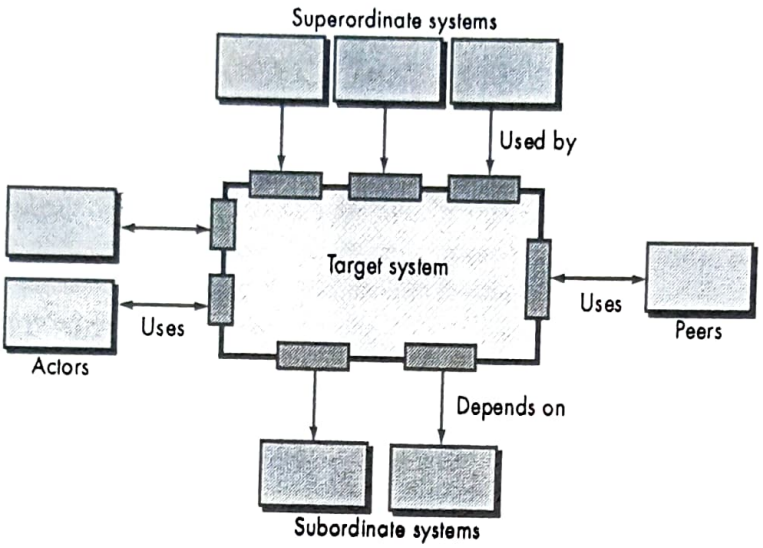
An *archetype* is an abstraction (similar to a class) that represents one element of system behavior. The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail. Therefore, the designer specifies the structure of the system by defining and refining software components that implement each archetype. This process continues iteratively until a complete architectural structure has been derived.

A number of questions [Boo11b] must be asked and answered as a software engineer creates meaningful architectural diagrams. Does the diagram show how the system responds to inputs or events? What visualizations might there be to help emphasize areas of risk? How can hidden system design patterns be made more obvious to other developers? Can multiple viewpoints show the best way to refactor specific parts of the system? Can design trade-offs be represented in a meaningful way? If a diagrammatic representation of software architecture answers these questions, it will have value to software engineers that use it.

13.6.1 Representing the System in Context

At the architectural design level, a software architect uses an *architectural context diagram* (ACD) to model the manner in which software interacts with entities external to its boundaries. The generic structure of the architectural context diagram is illustrated in Figure 13.5.

FIGURE 13.5
Architectural context diagram
Source: Adapted from [Bos00].



? How do systems interoperate with one another?

Referring to the figure, systems that interoperate with the *target system* (the system for which an architectural design is to be developed) are represented as:

- *Superordinate systems*—those systems that use the target system as part of some higher-level processing scheme.
- *Subordinate systems*—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
- *Peer-level systems*—those systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system.
- *Actors*—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.

Each of these external entities communicates with the target system through an interface (the small shaded rectangles).

To illustrate the use of the ACD, consider the home security function of the *SafeHome* product. The overall *SafeHome* product controller and the Internet-based system are both superordinate to the security function and are shown above the function in Figure 13.6. The surveillance function is a *peer system* and uses (is used by) the home security function in later versions of the product. The homeowner and control panels are actors that producer and consume information used/produced by the security software. Finally, sensors are used by the security software and are shown as subordinate to it.

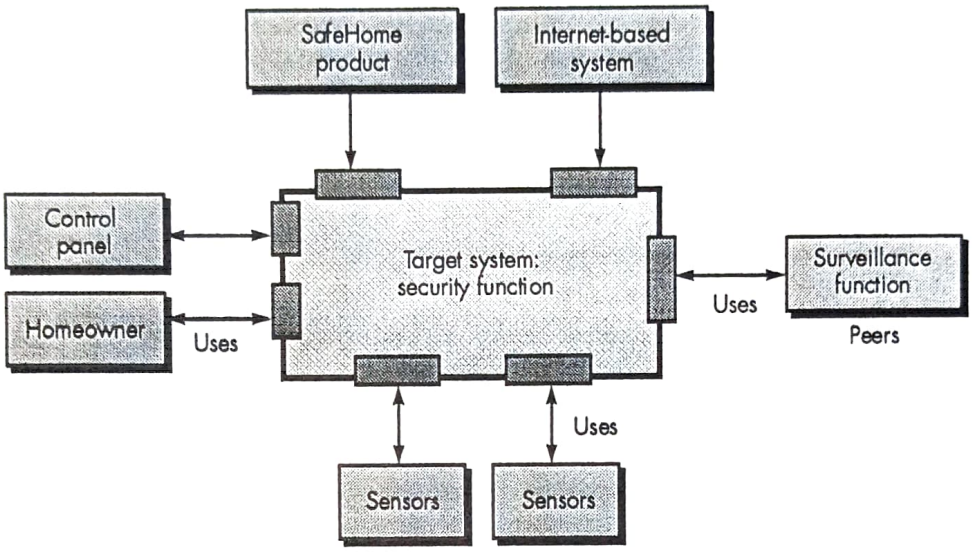


FIGURE 13.6 Architectural context diagram for the *SafeHome* security function

As part of the architectural design, the details of each interface shown in Figure 13.6 would have to be specified. All data that flow into and out of the target system must be identified at this stage.

13.6.2 Defining Archetypes

KEY POINT
Archetypes are the abstract building blocks of an architectural design.

An *archetype* is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. In general, a relatively small set of archetypes is required to design even relatively complex systems. The target system architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.

In many cases, archetypes can be derived by examining the analysis classes defined as part of the requirements model. Continuing the discussion of the *SafeHome* home security function, you might define the following archetypes:

- **Node.** Represents a cohesive collection of input and output elements of the home security function. For example, a node might be composed of (1) various sensors and (2) a variety of alarm (output) indicators.
- **Detector.** An abstraction that encompasses all sensing equipment that feeds information into the target system.
- **Indicator.** An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.
- **Controller.** An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

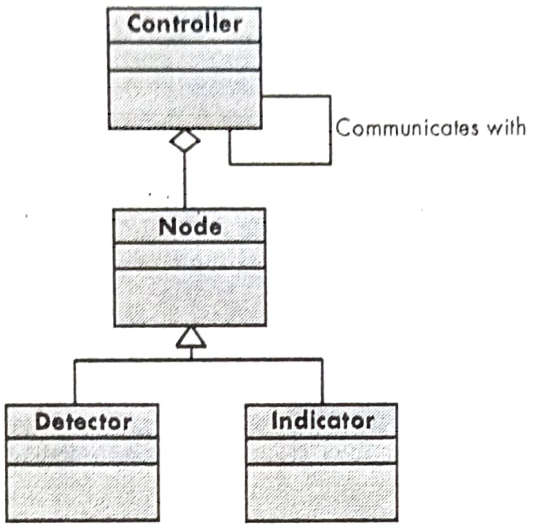


FIGURE 13.7 UML relationships for *SafeHome* security function archetypes
Source: Adapted from [Bos00].

Each of these archetypes is depicted using UML notation as shown in Figure 13.7. 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.

13.6.3 Refining the Architecture into Components

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, you begin with the classes that were described as part of the requirements 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 13.6.1) imply one or more specialized components that process the data that flows across the interface. In some cases (e.g., a graphical user interface), a complete subsystem architecture with many components must be designed.

Quote:
"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

6 If a conventional (non-object-oriented) approach is chosen, components may be derived from the subprogram calling hierarchy (see Figure 13.3).

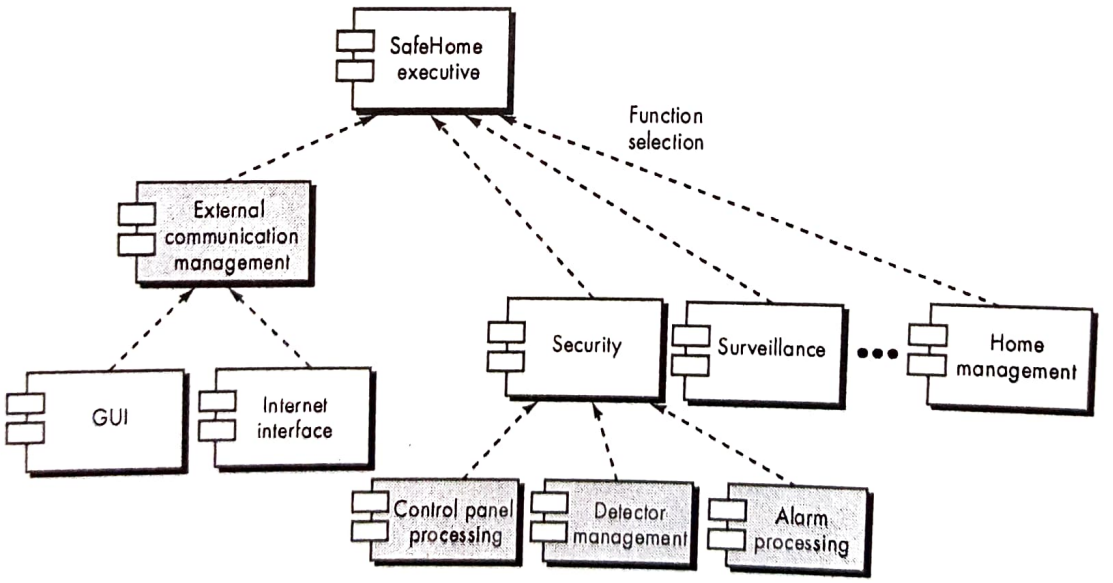
Continuing the *SafeHome* home security function example, you might define the set of top-level components that address the following functionality:

- *External communication management*—coordinates communication of the security function with external entities such as other Internet-based systems and 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 14).

The overall architectural structure (represented as a UML component diagram) is illustrated in Figure 13.8. 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 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*

FIGURE 13.8 Overall architectural structure for *SafeHome* with top-level components



a *scheduler* infrastructure component that implements polling of each *sensor* object used by the security system. Similar elaboration is performed for each of the components represented in Figure 13.8.

SOFTWARE TOOLS

Architectural Design

Objective: Architectural design tools model the overall software structure by representing component interface, 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.de/objectiF/en/**), is a UML-based design tool that leads to architectures (e.g., Coldfusion, J2EE, Fusebox) amenable to component-based software engineering (Chapter 14).

Rational Rose, developed by Rational (**http://www-01.ibm.com/software/rational/**), is a UML-based design tool that supports all aspects of architectural design.

13.6.5 Architectural Design for Web Apps

WebApps[8] are client-server applications typically structured using multilayered architectures, including a user interface or view layer, a controller layer which directs the flow of information to and from the client browser based on a set of business rules, and a content or model layer that may also contain the business rules for the WebApp.

The user interface for a WebApp is designed around the characteristics of the web browser running on the client machine (usually a personal computer or mobile device). Data layers reside on a server. Business rules can be implemented using a server-based scripting language such as PHP or a client-based scripting language such as javascript. An architect will examine requirements for security and usability to determine which features should be allocated to the client or server.

The architectural design of a WebApp is also influenced by the structure (linear or nonlinear) of the content that needs to be accessed by the client. The architectural components (Web pages) of a WebApp are designed to allow control to be passed to other system components, allowing very flexible navigation structures. The physical location of media and other content resources also influences the architectural choices made by software engineers.

---

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.

8 WebApp design is discussed in more detail in Chapter 17.

13.6.6 Architectural Design for Mobile Apps

Mobile apps[9] are typically structured using multilayered architectures, including a user interface layer, a business layer, and a data layer. With mobile apps you have the choice of building a thin Web-based client or a rich client. With a thin client, only the user interface resides on the mobile device, whereas the business and data layers reside on a server. With a rich client all three layers may reside on the mobile device itself.

Mobile devices differ from one another in terms of their physical characteristics (e.g., screen sizes, input devices), software (e.g., operating systems, language support), and hardware (e.g., memory, network connections). Each of these attributes shapes the direction of the architectural alternatives that can be selected. Meier and his colleagues [Mei09] suggest a number of considerations that can influence the architectural design of a mobile app: (1) the type of web client (thin or rich) to be built, (2) the categories of devices (e.g., smartphones, tablets) that are supported, (3) the degree of connectivity (occasional or persistent) required, (4) the bandwidth required, (5) the constraints imposed by the mobile platform, (6) the degree to which reuse and maintainability are important, and (7) device resource constraints (e.g., battery life, memory size, processor speed).

13.7 Assessing Alternative Architectural Designs

In their book on the evaluation of software architectures, Clements and his colleagues [Cle03] state:

> To put it bluntly, an architecture is a bet, a wager on the success of a system. Wouldn't it be nice to know in advance if you've placed your bet on a winner, as opposed to waiting until the system is mostly completed before knowing whether it will meet its requirements or not? If you're buying a system or paying for its development, wouldn't you like to have some assurance that it's started off down the right path? If you're the architect yourself, wouldn't you like to have a good way to validate your intuitions and experience, so that you can sleep at night knowing that the trust placed in your design is well founded?

Indeed, answers to these questions would have value. 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 present two different approaches for the assessment of alternative architectural designs. The first method uses an iterative method to assess design trade-offs. The second approach applies a pseudo-quantitative technique for assessing design quality.

---

9 Mobile app design is discussed in more detail in Chapter 18.