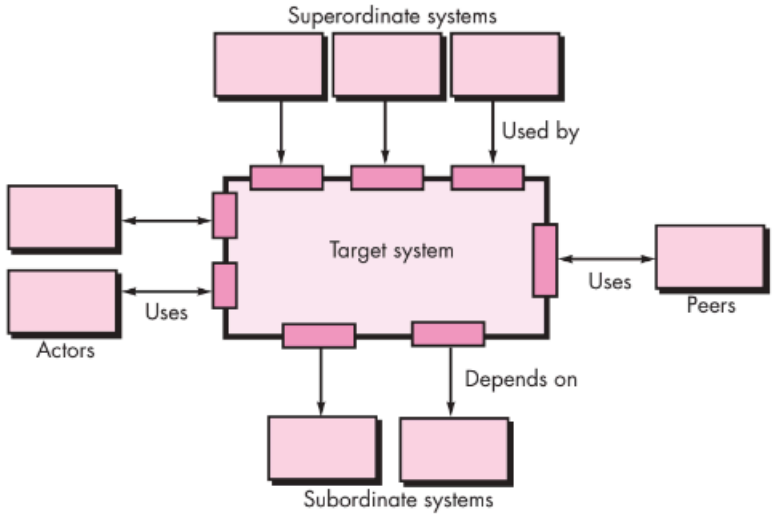
❖ Architectural Design :Design Decisions

As architectural design begins, the software to be developed must be put into context—that is, the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction. This information can generally be acquired from the requirements model and all other information gathered during requirements engineering. Once context is modeled and all external software interfaces have been described, you can identify a set of architectural archetypes. 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. In the sections that follow we examine each of these architectural design tasks in a bit more detail.

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 9.5.

FIGURE 9.5
Architectural context diagram
Source: Adapted from [Bos00].



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 superior to the security function and are shown above the function in Figure 9.6. 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 are both producers and consumers of information used/produced by the security software. Finally, sensors are used by the security software and are shown as subordinate to it.

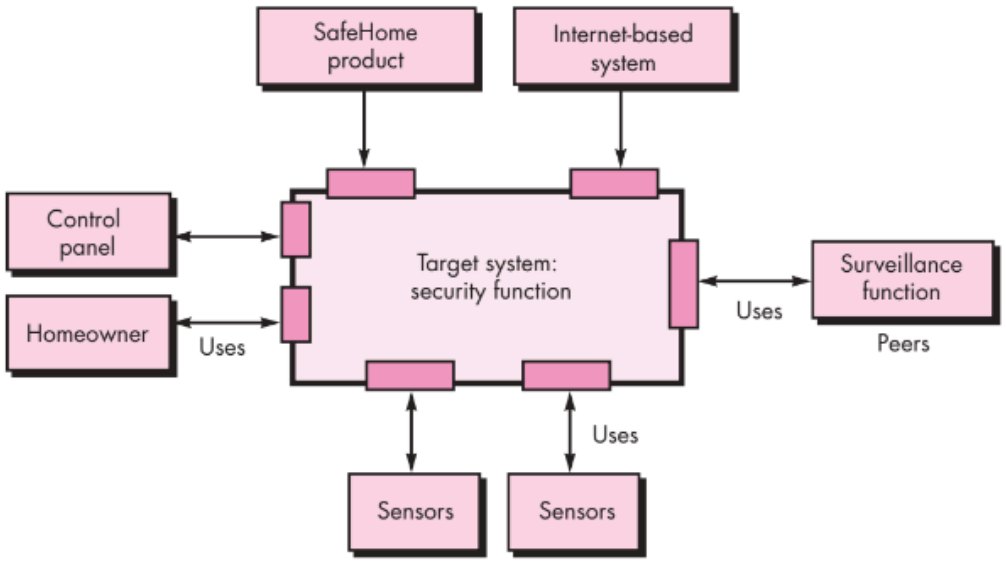


FIGURE 9.6 Architectural context diagram for the *SafeHome* security function

As part of the architectural design, the details of each interface shown in Figure 9.6 would have to be specified. All data that flow into and out of the target system must be identified at this stage.

Defining Archetypes

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 sys- tem 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.

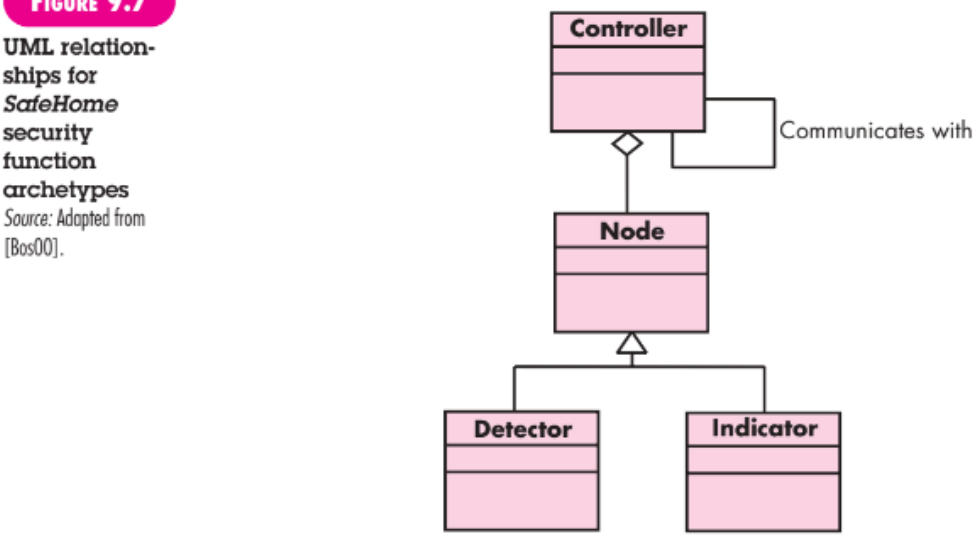


FIGURE 9.7
UML relationships for *SafeHome* security function archetypes
Source: Adapted from [Bos00].

Each of these archetypes is depicted using UML notation as shown in Figure 9.7.

❖ Views and Patterns

Software architecture is the blueprint of building software. It shows the overall structure of the software, the collection of components in it, and how they interact with one another while hiding the implementation.

This helps the software development team to clearly communicate how the software is going to be built as per the requirements of customers.

There are various ways to organize the components in software architecture. And the different predefined organization of components in software architectures are known as software architecture patterns. A lot of patterns were tried and tested. Most of them have successfully solved various problems. In each pattern, the components are organized differently for solving a specific problem in software architectures.

Well, I hope you don't want to bore yourself by reading the endless types of software architecture patterns. That's why among the multiple software architecture patterns, we are going to see a few of the most important and commonly used patterns.

Different Software Architecture Patterns :

- Layered Pattern
- Client-Server Pattern
- Event-Driven Pattern
- Microkernel Pattern
- Microservices Pattern

Let's see one by one in detail.

1. Layered Pattern :

As the name suggests, components(code) in this pattern are separated into layers of subtasks and they are arranged one above another.

Each layer has unique tasks to do and all the layers are independent of one another. Since each layer is independent, one can modify the code inside a layer without affecting others.

It is the most commonly used pattern for designing the majority of software. This layer is also known as 'N-tier architecture'. Basically, this pattern has 4 layers.

Presentation layer (The user interface layer where we see and enter data into an application.)

Business layer (this layer is responsible for executing business logic as per the request.)

Application layer (this layer acts as a medium for communication between the 'presentation layer' and 'data layer'.

Data layer (this layer has a database for managing data.)

Ideal for:

E-commerce web applications development like Amazon.

2. Client-Server Pattern :

The client-server pattern has two major entities. They are a server and multiple clients.

Here the server has resources(data, files or services) and a client requests the server for a particular resource. Then the server processes the request and responds back accordingly.

Examples of software developed in this pattern:

Email.

WWW.

File sharing apps.

Banking, etc…

So this pattern is suitable for developing the kind of software listed in the examples.

3. Event-Driven Pattern :

Event-Driven Architecture is an agile approach in which services (operations) of the software are triggered by events.

Well, what does an event mean?

When a user takes action in the application built using the EDA approach, a state change happens and a reaction is generated that is called an event.

Eg: A new user fills the signup form and clicks the signup button on Facebook and then a FB account is created for him, which is an event.

Ideal for:

Building websites with JavaScript and e-commerce websites in general.

4. Microkernel Pattern :

Microkernel pattern has two major components. They are a core system and plug-in modules.

The core system handles the fundamental and minimal operations of the application.

The plug-in modules handle the extended functionalities (like extra features) and customized processing.

Let's imagine, you have successfully built a chat application. And the basic functionality of the app is that you can text with people across the world without an internet connection. After some time, you would like to add a voice messaging feature to the application, then you are adding the feature successfully. You can add that feature to the already developed application because the microkernel pattern facilitates you to add features as plug-ins.

Microkernel pattern is ideal for:

Product-based applications and scheduling applications. We love new features that keep giving dopamine boost to our brain. Such as Instagram reels, YouTube Shorts and a lot more that feasts us digitally. So this pattern is mostly preferred for app development.

5. Microservices Pattern :

The collection of small services that are combined to form the actual application is the concept of microservices pattern. Instead of building a bigger application, small programs are built for every service (function) of an application independently. And those small programs are bundled together to be a full-fledged application.

So adding new features and modifying existing microservices without affecting other microservices are no longer a challenge when an application is built in a microservices pattern.

Modules in the application of microservices patterns are loosely coupled. So they are easily understandable, modifiable and scalable.

Example Netflix is one of the most popular examples of software built-in microservices architecture. This pattern is most suitable for websites and web apps having small components.

❖ Application Architectures

An application architecture is a structural map of how a software application is assembled and how applications interact with each other to meet business or user requirements. An application architecture helps ensure applications are scalable and reliable, and assists enterprises in identifying gaps in functionality.

In general, an application architecture defines how applications interact with entities such as middleware, databases and other applications. Application architectures usually follow software design principles that are generally accepted among adherents, but might lack formal industry standards.

The application architecture can be thought of like architectural blueprints when constructing a building. The blueprints set out how the building should be laid out and where things such as electric and plumbing service should go. The builders can then use