CHAPTER

10

ARCHITECTURAL DESIGN—A RECOMMENDED APPROACH

Design has been described as a multistep process in which representations of data and program structure, interface characteristics, and procedural detail are synthesized from information requirements. As we noted in Chapter 9, design is information driven. Software design methods are derived from consideration of each of the three domains of the analysis model. The decisions made while considering the data, functional, and behavioral domains serve as guides for the creation of the software architectural design.

KEY CONCEPTS

QUICK LOOK

What is it? Architectural design represents the structure of data and program components that are required to build a computer-based system. It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system.

Who does it? Although a software engineer can design both data and architecture, the job is often allocated to specialists when large, complex systems are to be built. A database or data warehouse designer creates the data architecture for a system. The "system architect" selects an appropriate architectural style from the requirements derived during software requirements analysis.

Why is it important? You wouldn't attempt to build a house without a blueprint, would you? You also wouldn't begin drawing blueprints by sketching the plumbing layout for the house. You'd need to look at the big picture—the house itself—before you worry about details. That's what architectural design does—it provides you with the big picture and ensures that you've got it right.

What are the steps? Architectural design begins with data design and then proceeds to the derivation of one or more representations of the architectural structure of the system. Alternative architectural styles or patterns are analyzed to derive the structure that is best suited to customer requirements and quality attributes. Once an alternative has been selected, the architecture is elaborated, using an architectural design method.

What is the work product? An architecture model encompassing data architecture and program structure is created during architectural design. In addition, component properties and relationships (interactions) are described.

How do I ensure that I've done it right? At each stage, software design work products are reviewed for clarity, correctness, completeness, and consistency with requirements and with one another.

Philippe Kruchten, Grady Booch, Kurt Bittner, and Rich Reitman [Mic09] suggest that software architecture identifies a system's "structural elements and their interfaces," along with the "behavior" of individual components and subsystems. They write that the job of architectural design is to create "coherent, well-planned representations" of the system and software.

Methods to create such representations of the data and architectural layers of the design model are presented in this chapter. The objective is to provide a systematic approach for the derivation of the architectural design—the preliminary blueprint from which software is constructed.

10.1 Software Architecture

In their landmark book on the subject, Shaw and Garlan [Sha15] argue that since the earliest days of computer programming, "software systems have had architectures, and programmers have been responsible for the interactions among the modules and the global properties of the assemblage." Today, effective software architecture and its explicit representation and design have become dominant themes in software engineering.

10.1.1 What Is Architecture?

When you consider the architecture of a building, many different attributes come to mind. At the most simplistic level, you think about the overall shape of the physical structure. But in reality, architecture is much more. It is the manner in which the various components of the building are integrated to form a cohesive whole. It is the way in which the building fits into its environment and meshes with other buildings in its vicinity. It is the degree to which the building meets its stated purpose and satisfies the needs of its owner. It is the aesthetic feel of the structure—the visual impact of the building—and the way textures, colors, and materials are combined to create the external facade and the internal "living environment." It is small details—the design of lighting fixtures, the type of flooring, the placement of wall hangings; the list is almost endless. And finally, it is art.

Architecture is also something else. It is "thousands of decisions, both big and small" [Tyr05]. Some of these decisions are made early in design and can have a profound impact on all other design actions. Others are delayed until later, thereby eliminating overly restrictive constraints that would lead to a poor implementation of the architectural style.

Just like the plans for a house are merely a representation of the building, the software architecture representation is not an operational product. Rather, it is a representation that enables you to (1) analyze the effectiveness of the design in meeting its stated requirements, (2) consider architectural alternatives at a stage when making design changes is still relatively easy, and (3) reduce the risks associated with the construction of the software.

This definition emphasizes the role of "software components" in any architectural representation. In the context of architectural design, a software component can be something as simple as a program module or an object-oriented class, but it can also be extended to include databases and "middleware" that enable the configuration of

a network of clients and servers. The properties of components are those characteristics that are necessary to an understanding of how the components interact with other components. At the architectural level, internal properties (e.g., details of an algorithm) are not specified. The relationships between components can be as simple as a procedure call from one module to another or as complex as a database access protocol.

We believe that a software design can be thought of as an instance of some software architecture. However, the elements and structures that are defined as parts of particular software architectures are the root of every design. It is our recommendation that design should begin with a consideration of the software architecture.

10.1.2 Why Is Architecture Important?

In a book dedicated to software architecture, Bass and his colleagues [Bas12] identify three key reasons that software architecture is important:

- Software architecture provides a representation that facilitates communication among all stakeholders.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows.
- The architecture constitutes a relatively small model of how the system components are structured and work together.

The architectural design model and the architectural patterns contained within it are transferable. That is, architecture genres, styles, and patterns (Sections 10.3 through 10.6) can be applied to the design of other systems and represent a set of abstractions that enable software engineers to describe architecture in predictable ways.

Making good decisions while defining the software architecture is critical to the success of a software product. The software architecture sets the structure of the system and determines its quality [Das15].

10.1.3 Architectural Descriptions

Each of us has a mental image of what the word *architecture* means. The implication is that different stakeholders will see a given software architecture from different viewpoints that are driven by different sets of concerns. This implies that an architectural description is actually a set of work products that reflect different views of the system.

Smolander, Rossi, and Purao [Smo08] have identified multiple metaphors, representing different views of the same architecture that stakeholders use to understand the term *software architecture*. The *blueprint metaphor* seems to be most familiar to the stakeholders who write programs to implement a system. Developers regard architecture descriptions as a means of transferring explicit information from architects to designers to software engineers charged with producing the system components.

The *language metaphor* views architecture as a facilitator of communication across stakeholder groups. This view is preferred by stakeholders with a high customer focus (e.g., managers or marketing experts). The architectural description needs to be concise and easy to understand because it forms the basis for negotiation, particularly in determining system boundaries.

The *decision metaphor* represents architecture as the product of decisions involving trade-offs among properties such as cost, usability, maintainability, and performance that have resource consequences for the system being designed. Stakeholders such as project managers view architectural decisions as the basis for allocating project resources and tasks. These decisions may affect the sequence of tasks and shape the structure of the software team.

The *literature metaphor* is used to document architectural solutions constructed in the past. This view supports the construction of artifacts and the transfer of knowledge between designers and software maintenance staff. It also supports stakeholders whose concern is reuse of components and designs.

An *architectural description* (AD) represents a system using multiple views, where each *view* is "a representation of a whole system from the perspective of a related set of [stakeholder] concerns." The IEEE Computer Society standard IEEE-Std-42010:2011(E), *Systems and software engineering—Architectural description* [IEE11], describes the use of architecture viewpoints, architecture frameworks, and architecture description languages as a means of codifying the conventions and common practices for architectural description.

10.1.4 Architectural Decisions

Each view developed as part of an architectural description addresses a specific stakeholder concern. To develop each view (and the architectural description as a whole), the system architect considers a variety of alternatives and ultimately decides on the specific architectural features that best meet the concern. Architectural decisions themselves can be considered to be one view of the architecture. The reasons that decisions were made provide insight into the structure of a system and its conformance to stakeholder concerns.

As a system architect, you can use the template suggested in the sidebar to document each major decision. By doing this, you provide a rationale for your work and establish a historical record that can be useful when design modifications must be made. For agile developers, a lightweight *architectural decision record* (ADR) might simply contain a title, a context (assumptions and constraints), the decision (resolution), status (proposed accepted rejected), and consequences (implications) [Nyg11].

Grady Booch [Boo11a] writes that when setting out to build an innovative product, software engineers often feel compelled to plunge right in, build stuff, fix what doesn't work, improve what does work, and then repeat the process. After doing this a few times, they begin to recognize that the architecture should be defined first and decisions associated with architectural choices must be stated explicitly. It may not be possible to predict the right choices before building a new product. However, if innovators find that architectural decisions are worth repeating after testing their prototypes in the field, then a *dominant design*[1] for this type of product may begin to emerge. Without documenting what worked and what did not, it is hard for software engineers to decide when to innovate and when to use previously created architecture.

1 *Dominant design* describes an innovative software architecture or process that becomes an industry standard after a period of successful adaptation and use in the marketplace.

INFO

Architecture Decision Description Template

Each major architectural decision can be documented for later review by stakeholders who want to understand the architecture description that has been proposed. The template presented in this sidebar is an adapted and abbreviated version of a template proposed by Tyree and Ackerman [Tyr05].

Design issue: Describe the architectural design issues that are to be addressed.

Resolution: State the approach you've chosen to address the design issue.

Category: Specify the design category that the issue and resolution address (e.g., data design, content structure, component structure, integration, presentation).

Assumptions: Indicate any assumptions that helped shape the decision.

Constraints: Specify any environmental constraints that helped shape the decision (e.g., technology standards, available patterns, project-related issues).

Alternatives: Briefly describe the architectural design alternatives that were considered and why they were rejected.

Argument: State why you chose the resolution over other alternatives.

Implications: Indicate the design consequences of making the decision. How will the resolution affect other architectural design issues? Will the resolution constrain the design in any way?

Related decisions: What other documented decisions are related to this decision?

Related concerns: What other requirements are related to this decision?

Work products: Indicate where this decision will be reflected in the architecture description.

Notes: Reference any team notes or other documentation that was used to make the decision.

10.2 AGILITY AND ARCHITECTURE

The view of some agile developers is that architectural design is equated with "big design upfront." In their view, this leads to unnecessary documentation and the implementation of unnecessary features. However, most agile developers would agree [Fal10] that it is important to focus on software architecture when a system is complex (i.e., when a product has a large number of requirements, lots of stakeholders, or a large number of global users). For this reason, it is important to integrate new architectural design practices into agile process models.

To make early architectural decisions and avoid the rework required to correct the quality problems encountered when the wrong architecture is chosen, agile developers

need to anticipate architectural elements[2] and implied structure that emerges from the collection of user stories gathered (Chapter 7). By creating an architectural prototype (e.g., a *walking skeleton*) and developing explicit architectural work products to communicate the right information to the necessary stakeholders, an agile team can satisfy the need for architectural design.

Using a technique called *storyboarding*, the architect contributes architectural user stories to the project and works with the product owner to prioritize the architectural stories with the business user stories as "sprints" (work units) are planned. The architect works with the team during the sprint to ensure that the evolving software continues to show high architectural quality as defined by the nonfunctional product requirements. If quality is high, the team is left alone to continue development on its own. If not, the architect joins the team for the duration of the sprint. After the sprint is completed, the architect reviews the working prototype for quality before the team presents it to the stakeholders in a formal sprint review. Well-run agile projects make use of iterative work product delivery (including architectural documentation) with each sprint. Reviewing the work products and code as it emerges from each sprint is a useful form of architectural review.

Responsibility-driven architecture (RDA) is a process that focuses on when, how, and who should make the architectural decisions on a project team. This approach also emphasizes the role of architect as being a servant-leader rather than an autocratic decision maker and is consistent with the agile philosophy. The architect acts as facilitator and focuses on how the development team works to accommodate stakeholder's nontechnical concerns (e.g., business, security, usability).

Agile teams usually have the freedom to make system changes as new requirements emerge. Architects want to make sure that the important parts of the architecture were carefully considered and that developers have consulted the appropriate stakeholders. Both concerns may be satisfied by making use of a practice called *progressive sign-off* in which the evolving product is documented and approved as each successive prototype is completed [Bla10].

Using a process that is compatible with the agile philosophy provides verifiable sign-off for regulators and auditors, without preventing the empowered agile teams from making the decisions needed. At the end of the project, the team has a complete set of work products and the architecture has been reviewed for quality as it evolved.

10.3 ARCHITECTURAL STYLES

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,

2 An excellent discussion of architectural agility can be found in [Bro10a].

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 [Bas12].

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 refactored (Chapter 27), 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].

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], and (3) architectural patterns (Section 10.3.2) 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.

10.3.1 A Brief Taxonomy of Architectural Styles

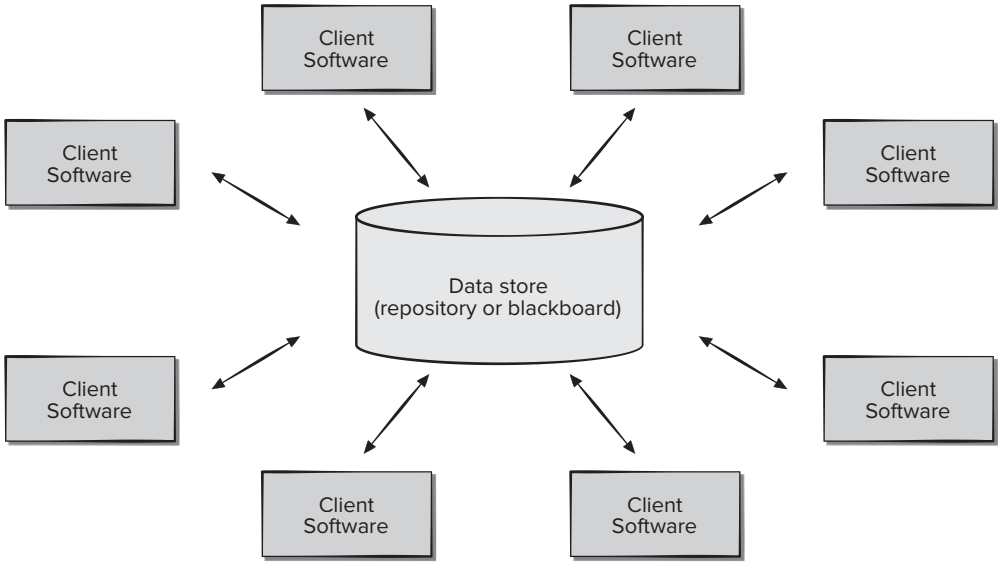
Although millions of computer-based systems have been created over the past 60 years, the vast majority can be categorized into one of a relatively small number of architectural styles.

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 that update, add, delete, or otherwise modify data within the store. Figure 10.1 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* [Bas12]. 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.

FIGURE 10.1

Data-centered architecture



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 10.2) 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.

FIGURE 10.2 **Data-flow architecture**

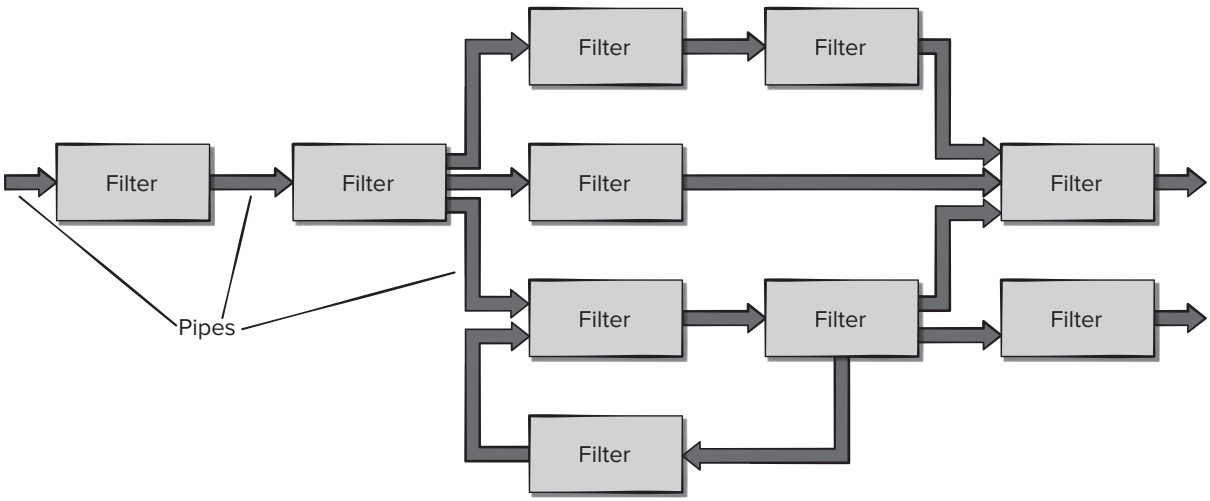
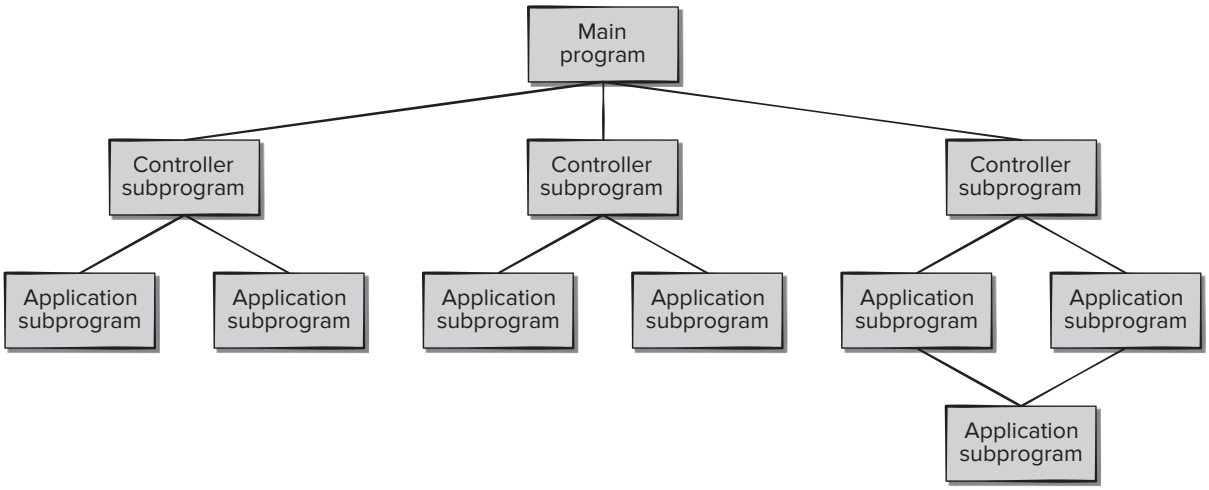


FIGURE 10.3 Main program/subprogram architecture



Call-and-Return Architectures. This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. Two substyles [Bas12] that exist within this category:

- *Main program/subprogram architectures.* This classic program structure decomposes function into a control hierarchy where a "main" program invokes several program components, which in turn may invoke still other components. Figure 10.3 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.

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. Figure 10.4 contains a UML communication diagram that shows the message passing for the login portion of a system implemented using an object-oriented architecture. Communications diagrams are described in more details in Appendix 1 of this book.

Layered Architectures. The basic structure of a layered architecture is illustrated in Figure 10.5. 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.

Model-View-Controller (MVC) architecture [Kra88] is one of a number of suggested mobile infrastructure models often used in Web development. The *model* contains all application-specific content and processing logic. The *view* contains all

FIGURE 10.4

UML communication diagram

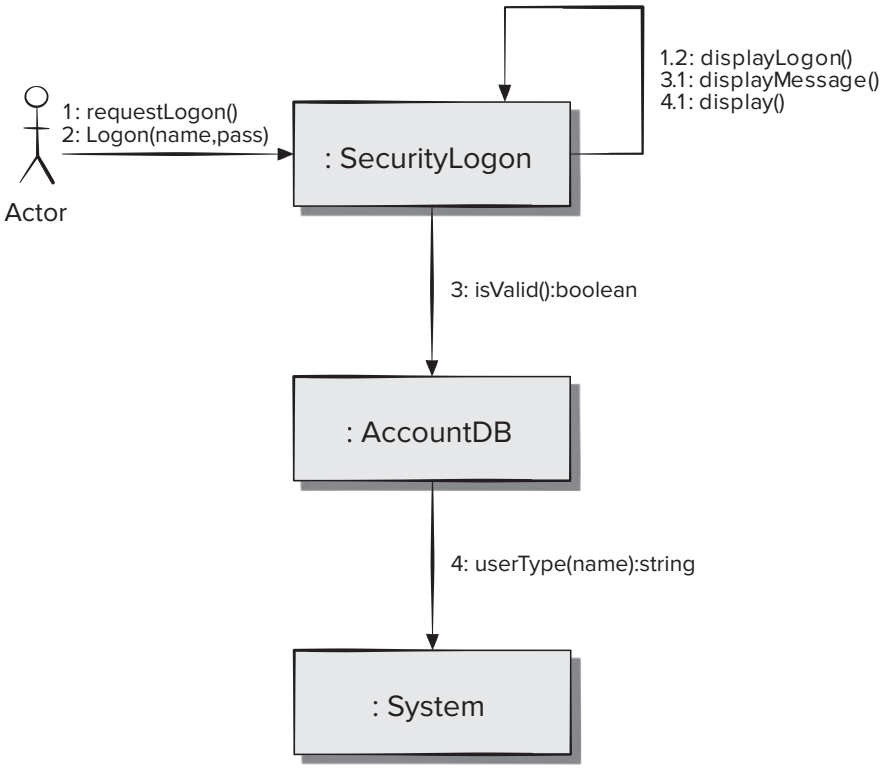


FIGURE 10.5

Layered architecture

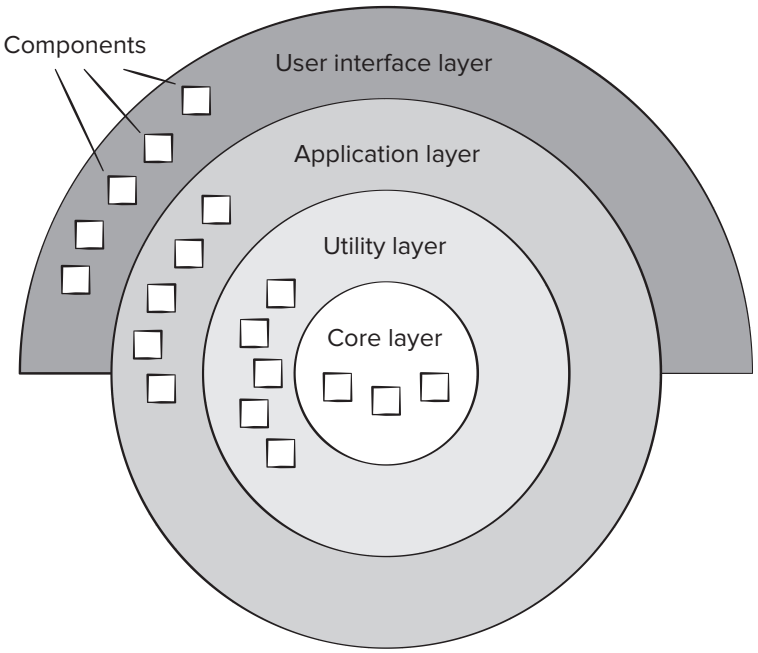
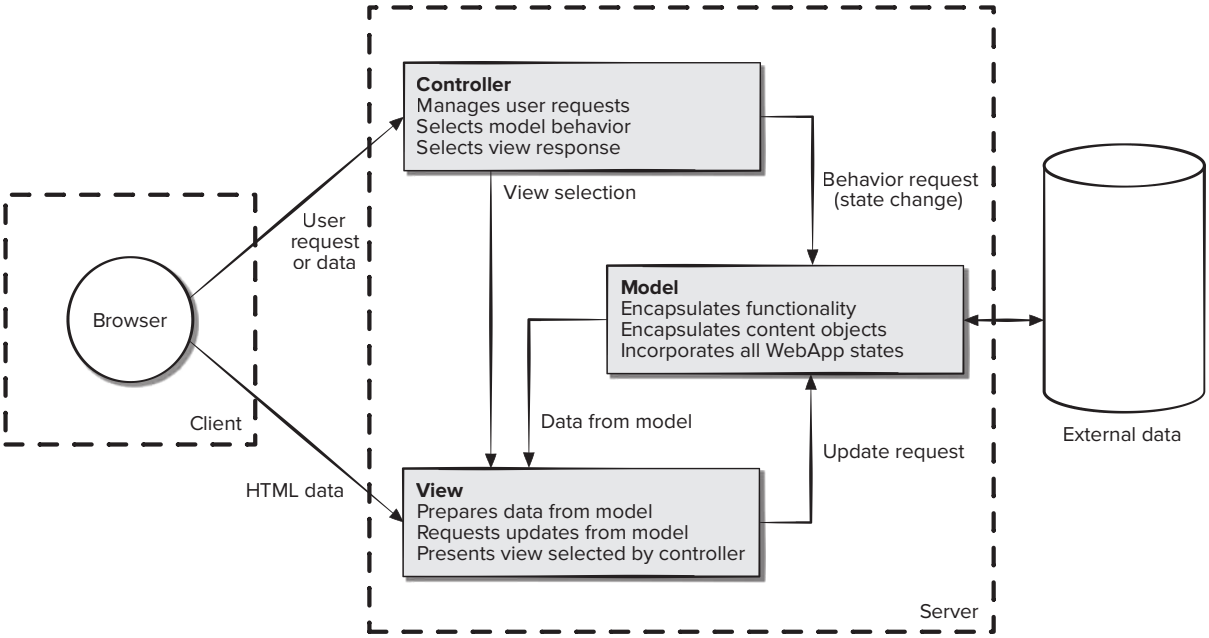


FIGURE 10.6 The MVC architecture



Source: Adapted from Jacyntho, Mark Douglas, Schwabe, Daniel and Rossi, Gustavo, "An Architecture for Structuring Complex Web Applications," 2002, available at http://www-di.inf.puc-rio.br/schwabe/papers/OOHDMJava2%20Report.pdf

interface-specific functions and enables the presentation of content and processing logic required by the end user. The *controller* manages access to the model and the view and coordinates the flow of data between them. A schematic representation of the MVC architecture is shown in Figure 10.6.

Referring to the figure, user requests are handled by the controller. The controller also selects the view object that is applicable based on the user request. Once the type of request is determined, a behavior request is transmitted to the model, which implements the functionality or retrieves the content required to accommodate the request. The model object can access data stored in a corporate database, as part of a local data store, or as a collection of independent files. The data developed by the model must be formatted and organized by the appropriate view object and then transmitted from the application server back to the client-based browser for display on the customer's machine.

These architectural styles are only a small subset of those available.[3] 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.

3 See [Roz11], [Tay09], [Bus07], [Gor06], or [Bas12] for a detailed discussion of architectural styles and patterns.

SAFEHOME

Choosing an Architectural Style

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

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

The conversation:

Ed (frowning): We've been modeling the security function using UML . . . you know, classes, relationships, that sort of stuff. So I guess the object-oriented architecture is the right way to go.

Jamie: But . . .?

Ed: But . . . I have trouble visualizing what an object-oriented architecture is. I get the call-and-return architecture, sort of a conventional process hierarchy, but OO . . . I don't know, it seems sort of amorphous.

Jamie (smiling): Amorphous, huh?

Ed: Yeah . . . what I mean is I can't visualize a real structure, just design classes floating in space.

Jamie: Well, that's not true. There are class hierarchies . . . think of the hierarchy (aggregation) we did for the **FloorPlan** object [Figure 9.3]. An OO architecture is a combination of that structure and the interconnections—you know, collaborations—between the classes. We can show it by fully describing the attributes and operations, the messaging that goes on, and the structure of the classes.

Ed: I'm going to spend an hour mapping out a call-and-return architecture; then I'll go back and consider an OO architecture.

Jamie: Doug'll have no problem with that. He said that we should consider architectural alternatives. By the way, there's absolutely no reason why both of these architectures couldn't be used in combination with one another.

Ed: Good. I'm on it.

Choosing the right architecture style can be tricky. Real-world problems often follow more than one problem frame, and a combination architectural model may result. For example, the model-view-controller (MVC) architecture used in WebApp design[4] might be viewed as combining two problem frames (command behavior and information display). In MVC, the end user's command is sent from the browser window to a command processor (controller) that manages access to the content (model) and instructs the information rendering model (view) to translate it for display by the browser software.

10.3.2 Architectural Patterns

As the requirements model is developed, you'll notice that the software must address several broad problems that span the entire application. For example, the requirements model for virtually every e-commerce application is faced with the following problem: *How do we offer a broad array of goods to many different customers and allow those customers to find and purchase our goods easily?*

The requirements model also defines a context in which this question must be answered. For example, an e-commerce business that sells golf equipment to consumers will operate in a different context than an e-commerce business that sells high-priced industrial equipment to medium and large corporations. In addition, a set of limitations and constraints may affect the way you address the problem to be solved.

4 The MVC architecture is considered in more detail in Chapter 13.

Architectural patterns address an application-specific problem within a specific context and under a set of limitations and constraints. The pattern proposes an architectural solution that can serve as the basis for architectural design.

Previously in this chapter, we noted that most applications fit within a specific domain or genre and that one or more architectural styles may be appropriate for that genre. For example, the overall architectural style for an application might be call and return or object oriented. But within that style, you will encounter a set of common problems that might best be addressed with specific architectural patterns. Some of these problems and a more complete discussion of architectural patterns are presented in Chapter 14.

10.3.3 Organization and Refinement

Because the design process often leaves you with a number of architectural alternatives, it is important to establish a set of design criteria that can be used to assess an architectural design that is derived. The following questions [Bas12] provide insight into an architectural style:

> **Control.** How is control managed within the architecture? Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy? How do components transfer control within the system? How is control shared among components? What is the control topology (i.e., the geometric form that the control takes)? Is control synchronized, or do components operate asynchronously?
>
> **Data.** How are data communicated between components? Is the flow of data continuous, or are data objects passed to the system sporadically? What is the mode of data transfer (i.e., are data passed from one component to another or are data available globally to be shared among system components)? Do data components (e.g., a blackboard or repository) exist, and if so, what is their role? How do functional components interact with data components? Are data components passive or active (i.e., does the data component actively interact with other components in the system)? How do data and control interact within the system?

The answers to these questions provide the designer with an early assessment of design quality and lay the foundation for more detailed analysis of the architecture.

Evolutionary process models (Chapter 2) have become very popular. This implies the software architectures may need to evolve as each product increment is planned and implemented. In Chapter 9, we described this process as refactoring—improving the internal structure of the system without changing its external behavior.

10.4 Architectural Considerations

Buschmann and Henny [Bus10a, Bus10b] suggest several architectural considerations that can provide software engineers with guidance as architecture decisions are made.

- **Economy.** The best software is uncluttered and relies on abstraction to reduce unnecessary detail. It avoids complexity due to unnecessary functions and features.

- **Visibility.** As the design model is created, architectural decisions and the reasons for them should be obvious to software engineers who examine the model later. Important design and domain concepts must be communicated effectively.
- **Spacing.** Separation of concerns (Chapter 9) in a design is sometimes referred to as *spacing*. Sufficient spacing leads to modular designs, but too much spacing leads to fragmentation and loss of visibility.
- **Symmetry.** Architectural symmetry implies that a system is consistent and balanced in its attributes. Symmetric designs are easier to understand, comprehend, and communicate. As an example of architectural symmetry, consider a **customer account** object whose life cycle is modeled directly by a software architecture that requires both *open()* and *close()* methods. Architectural symmetry can be both structural and behavioral.
- **Emergence.** Emergent, self-organized behavior and control are often the key to creating scalable, efficient, and economic software architectures. For example, many real-time software applications are event driven. The sequence and duration of these events that define the system's behavior is an emergent quality. Because it is very difficult to plan for every possible sequence of events, a system architect should create a flexible system that accommodates this emergent behavior.

These considerations do not exist in isolation. They interact with each other and are moderated by each other. For example, spacing can be both reinforced and reduced by economy. Visibility can be balanced by spacing.

The architectural description for a software product is not explicitly visible in the source code used to implement it. As a consequence, code modifications made over time (e.g., software maintenance activities) can cause slow erosion of the software architecture. The challenge for a designer is to find suitable abstractions for the architectural information. These abstractions have the potential to add structuring that improves readability and maintainability of the source code [Bro10b].

SAFEHOME

Evaluating Architectural Decisions

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

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

The conversation:

Ed: I finished my call-return architectural model of the security function.

Jamie: Great! Do you think it meets our needs?

Ed: It doesn't introduce any unneeded features, so it seems to be economic.

Jamie: How about visibility?

Ed: Well, I understand the model, and there's no problem implementing the security requirements needed for this product.

Jamie: I get that you understand the architecture, but you may not be the programmer for this part of the project. I'm a little worried

about spacing. This design may not be as modular as an object-oriented design.

Ed: Maybe, but that may limit our ability to reuse some of our code when we have to create the mobile version of *SafeHome*.

Jamie: What about symmetry?

Ed: Well, that's harder for me to assess. It seems to me the only place for symmetry in the security function is adding and deleting PIN information.

Jamie: That will get more complicated when we add remote security features to the mobile app.

Ed: That's true, I guess.

(They both pause for a moment, pondering the architectural issues.)

Jamie: *SafeHome* is a real-time system, so state transition and sequencing of events will be tough to predict.

Ed: Yeah, but the emergent behavior of this system can be handled with a finite state model.

Jamie: How?

Ed: The model can be implemented based on the call-return architecture. Interrupts can be handled easily in many programming languages.

Jamie: Do you think we need to do the same kind of analysis for the object-oriented architecture we were initially considering?

Ed: I suppose it might be a good idea, because architecture is hard to change once implementation starts.

Jamie: It's also important for us to map the nonfunctional requirements besides security on top of these architectures to be sure they have been considered thoroughly.

Ed: Also, true.

10.5 ARCHITECTURAL DECISIONS

Decisions about system architecture identify key design issues and the rationale behind chosen architectural solutions. System architecture decisions encompass software system organization, selection of structural elements and their interfaces as defined by their intended collaborations, and the composition of these elements into increasingly larger subsystems [Kru09]. In addition, choices of architectural patterns, application technologies, middleware assets, and programming language can also be made. The outcome of the architectural decisions influences the system's nonfunctional characteristics and many of its quality attributes [Zim11] and can be documented with *developer notes*. These notes document key design decisions along with their justification, provide a reference for new project team members, and serve as a repository for lessons learned.

In general, software architectural practice focuses on architectural views that represent and document the needs of various stakeholders. It is possible, however, to define a *decision view* that cuts across several views of information contained in traditional architectural representations. The decision view captures both the architecture design decisions and their rationale.

Service-oriented architecture decision (SOAD)[5] modeling [Zim11] is a knowledge management framework that provides support for capturing architectural decision dependencies in a manner that allows them to guide future development activities.

5 SOAD is analogous to the use of architecture patterns discussed in Chapter 14.

A SOAD *guidance model* contains knowledge about architectural decisions required when applying an architectural style in a particular application genre. It is based on architectural information obtained from completed projects that employed the architectural style in that genre. The guidance model documents places where design problems exist and architectural decisions must be made, along with quality attributes that should be considered in selecting from among potential 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.

A SOAD *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.

10.6 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.

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.

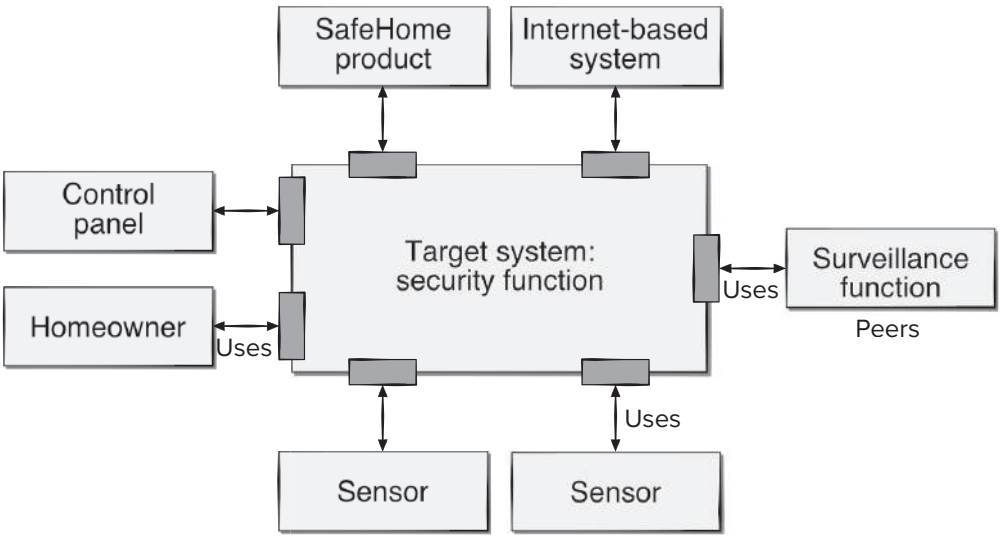
Several questions [Boo11b] must be asked and answered as a software engineer creates meaningful architectural diagrams. Does the diagram show how the system responds to inputs or events? What visualizations might there be to help emphasize areas of risk? How can hidden system design patterns be made 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.

10.6.1 Representing the System in Context

UML does not contain specific diagrams that represent the system in context. Software engineers wishing to stick with UML and represent the system in context would do so with a combination of use case, class, component, activity, sequence, and collaboration diagrams. Some software architects may make use of an *architectural context diagram* (ACD) to model the manner in which software interacts with entities external to its boundaries. An architectural context diagram for the *SafeHome* security functions is shown in Figure 10.7.

FIGURE 10.7

Architectural context diagram for the *SafeHome* security function



To illustrate the use of the ACD, consider the home security function of the *SafeHome* product shown in Figure 10.7. The overall *SafeHome* product controller and the Internet-based system are both superordinate to the security function and are shown above the function. 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 produce and consume information that is, respectively, used and produced by the security software. Finally, sensors are used by the security software and are shown as subordinate to it (by drawing them below the target system).

As part of the architectural design, the details of each interface shown in Figure 10.7 would have to be specified. All data that flow into and out of the target system must be identified at this stage.

10.6.2 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 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.

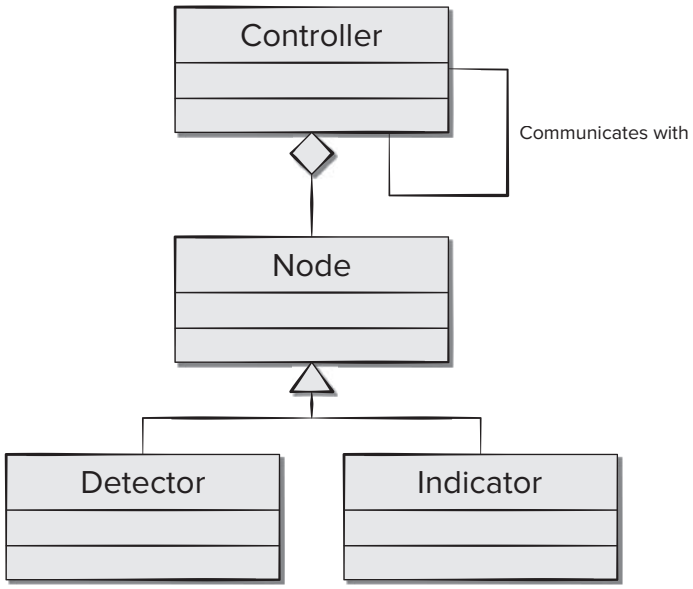
Each of these archetypes is depicted using UML notation, as shown in Figure 10.8. Recall that the archetypes form the basis for the architecture but are abstractions that must be further refined as architectural design proceeds. For example, **Detector** might be refined into a class hierarchy of sensors.

10.6.3 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? 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.

FIGURE 10.8

UML relationships for *SafeHome* security function archetype
Source: Adapted from Bosch, Jan, Design & Use of Software Architectures. Pearson Education, 2000.



6 If a conventional (non-object-oriented) approach is chosen, components may be derived from the subprogram calling hierarchy (see Figure 10.3).

The interfaces depicted in the architecture context diagram (Section 10.6.1) 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.

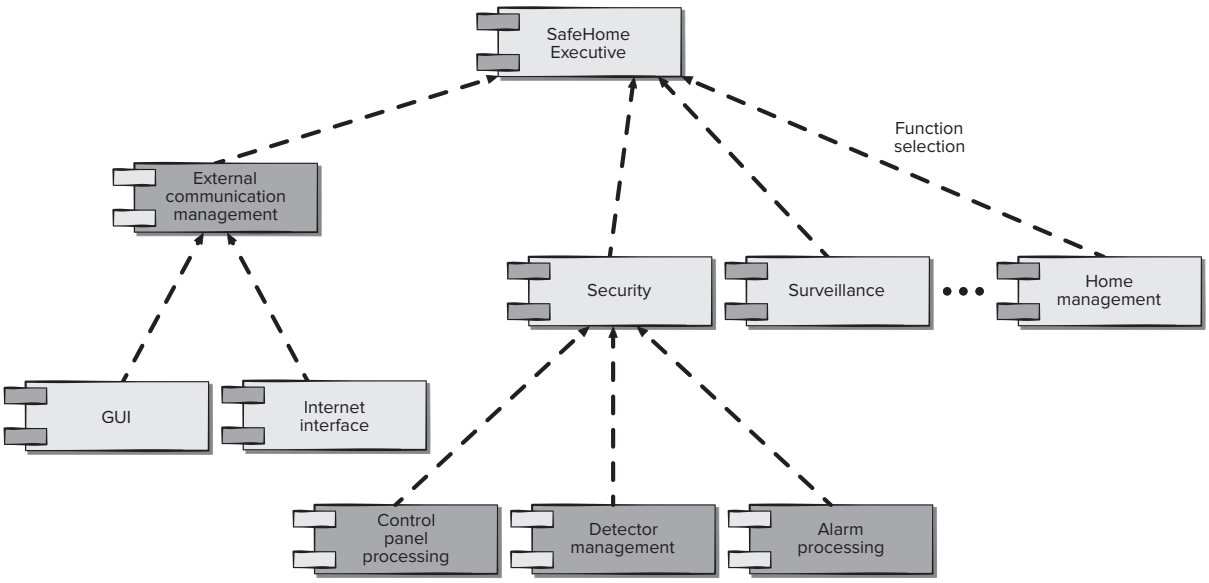
Continuing the *SafeHome* home security function example, you might define the set of top-level components that addresses the following functionality:

- ***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 11).

The overall architectural structure (represented as a UML component diagram) is illustrated in Figure 10.9. 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 and disarm the security

FIGURE 10.9 **Overall architectural structure for *SafeHome* with top-level components**



function. The *detector management* component polls sensors to detect an alarm condition, and the *alarm processing* component produces output when an alarm is detected.

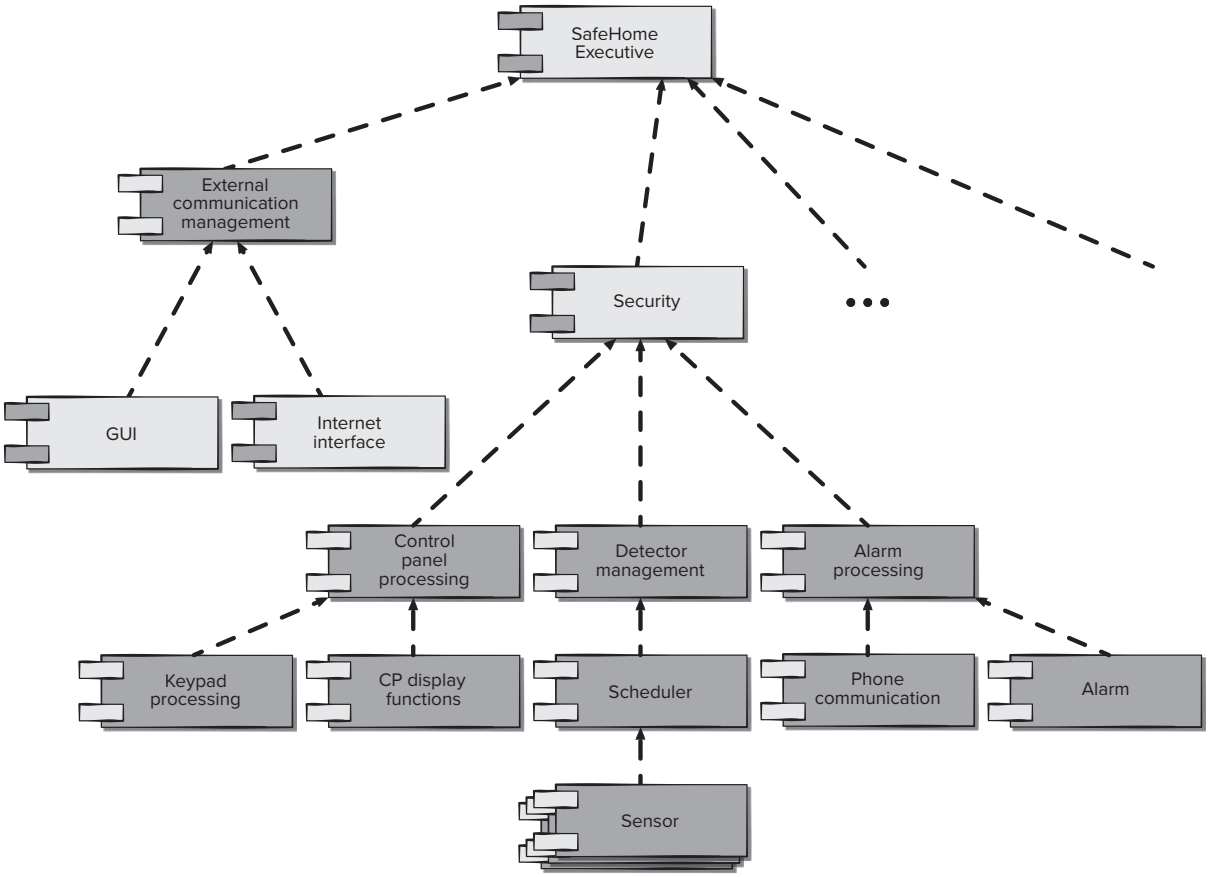
10.6.4 Describing Instantiations of the System

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 10.10 illustrates an instantiation of the *SafeHome* architecture for the security system. Components shown in Figure 10.9 are elaborated to show additional detail. For example, the *detector management* component interacts with 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 10.10.

FIGURE 10.10 **An instantiation of the security function with component elaboration**



10.7 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."

The big question for a software architect and the software engineers who will work to build a system is simple: Will the architectural bet pay off?

To help answer this question, architectural design should result in a number of architectural alternatives that are each assessed to determine which is the most appropriate for the problem to be solved.

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:

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.
3. **Describe the architectural styles and patterns that have been chosen to address the scenarios and requirements.** The architectural style(s) should be described using one of the following architectural views:
   - *Module view* for analysis of work assignments with components and the degree to which information hiding has been achieved.
   - *Process view* for analysis of system performance.
   - *Data flow view* for analysis of the degree to which the architecture meets functional requirements.
4. **Evaluate quality attributes by considering each attribute in isolation.** The number of quality attributes chosen for analysis is a function of the time available for review and the degree to which quality attributes are relevant to the system at hand. 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.

SAFEHOME

Architecture Assessment

The scene: Doug Miller's office as architectural design modeling proceeds.

The players: Vinod, Jamie, and Ed, members of the *SafeHome* software engineering team. Also Doug Miller, manager of the software engineering group.

The conversation:

Doug: I know you guys are deriving a couple of different architectures for the *SafeHome* product, and that's a good thing. I guess my question is, how are we going to choose the one that's best?

Ed: I'm working on a call-and-return style, and then either Jamie or I will derive an OO architecture.

Doug: Okay, and how do we choose?

Jamie: I took a CS course in design in my senior year, and I remember that there are a number of ways to do it.

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. So we 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. Make 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: Aren't scenarios 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, 3 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 defines 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.

10.7.1 Architectural Reviews

Architectural reviews are a type of specialized technical review (Chapter 16) that provide a means of assessing the ability of a software architecture to meet the system's quality requirements (e.g., scalability or performance) and to identify any potential

risks. Architectural reviews have the potential to reduce project costs by detecting design problems early.

Unlike requirements reviews that involve representatives of all stakeholders, architecture reviews often involve only software engineering team members supplemented by independent experts. However, software-based systems are built by people with a variety of different needs and points of view. Architects often focus on the long-term impact of the system's nonfunctional requirements as the architecture is created. Senior managers assess the architecture within the context of business goals and objectives. Project managers are often driven by short-term considerations of delivery dates and budget. Software engineers are often focused on their own technology interests and feature delivery. Each of these (and other) constituencies must agree that the software architecture chosen has distinct advantages over any other alternatives. Therefore, a wise software architect should build consensus among members of the software team (and other stakeholders) to achieve the architectural vision for the final software product [Wri11].

The most common architectural review techniques used in industry are: experience-based reasoning, prototype evaluation, scenario review (Chapter 8), and use of checklists. Many architectural reviews occur early in the project life cycle; they should also occur after new components or packages are acquired in component-based design (Chapter 11). One of the most commonly cited problems facing software engineers when conducting architectural reviews is missing or inadequate architectural work products, thereby making review difficult to complete [Bab09].

10.7.2 Pattern-Based Architecture Review

Formal technical reviews (Chapter 16) can be applied to software architecture and provide a means for managing system quality attributes, uncovering errors, and avoiding unnecessary rework. However, in situations in which short build cycles, tight deadlines, volatile requirements, and/or small teams are the norm, a lightweight architectural review process known as *pattern-based architecture review* (PBAR) might be the best option.

PBAR is an evaluation method based on architectural patterns[7] that leverages the patterns' relationships to quality attributes. A PBAR is a face-to-face audit meeting involving all developers and other interested stakeholders. An external reviewer with expertise in architecture, architecture patterns, quality attributes, and the application domain is also in attendance. The system architect is the primary presenter.

A PBAR should be scheduled after the first working prototype or *walking skeleton*[8] is completed. The PBAR encompasses the following iterative steps [Har11]:

1. Identify and discuss the quality attributes most important to the system by walking through the relevant use cases (Chapter 8).
2. Discuss a diagram of the system's architecture in relation to its requirements.

---

7 An *architectural pattern* is a generalized solution to an architectural design problem with a specific set of conditions or constraints. Patterns are discussed in detail in Chapter 14.

8 A *walking skeleton* contains a baseline architecture that supports the functional requirements with the highest priorities in the business case and the most challenging quality attributes.

3. Help the reviewer identify the architecture patterns used and match the system's structure to the patterns' structure.
4. Using existing documentation and past use cases, examine the architecture and quality attributes to determine each pattern's effect on the system's quality attributes.
5. Identify and discuss all quality issues raised by architecture patterns used in the design.
6. Develop a short summary of the issues uncovered during the meeting, and make appropriate revisions to the walking skeleton.

PBARs are well suited to small, agile teams and require a relatively small amount of extra project time and effort. With its short preparation and review time, PBAR can accommodate changing requirements and short build cycles and, at the same time, help improve the team's understanding of the system architecture.

10.7.3 Architecture Conformance Checking

As the software process moves through design and into construction, software engineers must work to ensure that an implemented and evolving system conforms to its planned architecture. Many things (e.g., conflicting requirements, technical difficulties, deadline pressures) cause deviations from a defined architecture. If architecture is not checked for conformance periodically, uncontrolled deviations can cause *architecture erosion* and affect the quality of the system [Pas10].

Static architecture-conformance analysis (SACA) assesses whether an implemented software system is consistent with its architectural model. The formalism (e.g., UML) used to model the system architecture presents the static organization of system components and how the components interact. Often the architectural model is used by a project manager to plan and allocate work tasks, as well as to assess implementation progress.

10.8 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.

Architectural design can coexist with agile methods by applying a hybrid architectural design framework that makes use of existing techniques derived from popular agile methods. Once an architecture is developed, it can be assessed to ensure conformance with business goals, software requirements, and quality attributes.

Several different architectural styles and patterns are available to the software engineer and may be applied within a given architectural genre. Each style describes a system category that encompasses a set of components that perform a function required by a system; a set of connectors that enable communication, coordination, and cooperation among components; constraints that define how components can be integrated

to form the system; and 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.

PROBLEMS AND POINTS TO PONDER

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

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

10.3. 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?

10.4. The terms *architectural style, architectural pattern,* and *framework* (not discussed in this book) are often encountered in discussions of software architecture. Do some research, and describe how each of these terms differs from its counterparts.

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

10.6. Research the ATAM (using [Kaz98]), and present a detailed discussion of the six steps presented in Section 10.7.1.

10.7. If you haven't done so, complete Problem 8.3. Use the design approach described in this chapter to develop a software architecture for the pothole tracking and repair system (PHTRS).

10.8. Use the architectural decision template from Section 10.1.4 to document one of the architectural decisions for PHTRS architecture developed in Problem 10.7.

10.9. Select a mobile application you are familiar with, and assess it using the architecture considerations (economy, visibility, spacing, symmetry, emergence) from Section 10.4.

10.10. List the strengths and weakness of the PHTRS architecture you created for Problem 10.7.