CHAPTER

13 ARCHITECTURAL DESIGN

KEY CONCEPTS

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. This description is extended by Freeman [Fre80]:

> [D]esign is an activity concerned with making major decisions, often of a structural nature. It shares with programming a concern for abstracting information representation and processing sequences, but the level of detail is quite different at the extremes. Design builds coherent, well-planned representations of programs that concentrate on the interrelationships of parts at the higher level and the logical operations involved at the lower levels.

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.

As we noted in Chapter 12, design is information driven. Software design methods are derived from consideration of each of the three domains of the analysis model. The data, functional, and behavioral domains serve as a guide for the creation of the software design.

Methods required to create "coherent, well-planned 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.

13.1 Software Architecture

In their landmark book on the subject, Shaw and Garlan [Sha96] discuss software architecture in the following manner:

> Ever since the first program was divided into modules, software systems have had architectures, and programmers have been responsible for the interactions among the modules and the global properties of the assemblage. Historically, architectures have been implicit—accidents of implementation, or legacy systems of the past. Good software developers have often adopted one or several architectural patterns as strategies for system organization, but they use these patterns informally and have no means to make them explicit in the resulting system.

Today, effective software architecture and its explicit representation and design have become dominant themes in software engineering.

13.1.1 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.

Quote:

"The architecture of a system is a comprehensive framework that describes its form and structure—its components and how they fit together."

Jerrold Grochow

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.

KEY POINT

Software architecture must model the structure of a system and the manner in which data and procedural components collaborate with one another.

But what about software architecture? Bass, Clements, and Kazman [Bas03] define this elusive term in the following way:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

The architecture is not the operational software. 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.

Quote:

"Marry your architecture in haste, repent at your leisure."

Barry Boehm

Some members of the software engineering community (e.g., [Kaz03]) make a distinction between the actions associated with the derivation of a software architecture (what we call "architectural design") and the actions that are applied to derive the software design. As one reviewer of a past edition noted:

> There is a distinct difference between the terms *architecture* and *design*. A *design* is an instance of an *architecture* similar to an object being an instance of a class. For example, consider the client-server architecture. I can design a network-centric software system in many different ways from this architecture using either the Java platform (Java EE) or Microsoft platform (.NET framework). So, there is one architecture, but many designs can be created based on that architecture. Therefore, you cannot mix "architecture" and "design" with each other.

Although we agree that a software design is an instance of a specific software architecture, the elements and structures that are defined as part of an architecture are the root of every design. Design begins with a consideration of architecture.

WebRef

Useful pointers to many software architecture sites can be obtained at **http://www.ewita.com/links/softwareArchitectureLinks.htm.**

13.1.2 Why Is Architecture Important?

In a book dedicated to software architecture, Bass and his colleagues [Bas03] identify three key reasons that software architecture is important:

- Software architecture provides a representation that facilitates communication among all stakeholders.

The architectural model provides a Gestalt view of the system, allowing the software engineer to examine it as a whole.

- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows.
- Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together" [Bas03].

The architectural design model and the architectural patterns contained within it are transferable. That is, architecture genres, styles, and patterns (Sections 13.2 through 13.6) 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.

13.1.3 Architectural Descriptions

Each of us has a mental image of what the word *architecture* means. The implication is that different stakeholders will see an 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 since 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. Each of these properties can have a significant impact on the system design. Stakeholders (e.g., project managers) view architectural decisions as the basis for allocating project resources and work tasks. These decisions may affect the sequence of tasks and 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 of a software-based system must exhibit characteristics that combine these metaphors. Tyree and Akerman [Tyr05] note this when they write:

> Developers want clear, decisive guidance on how to proceed with design. Customers want a clear understanding of the environmental changes that must occur and assurances that the architecture will meet their business needs. Other architects want a clear, salient understanding of the architecture's key aspects.

Each of these "wants" is reflected in a different metaphor represented using a different viewpoint.

The IEEE Computer Society has proposed IEEE-Std-1471-2000, *Recommended Practice for Architectural Description of Software-Intensive Systems,* [IEE00], with the following objectives: (1) to establish a conceptual framework and vocabulary for use during the design of software architecture, (2) to provide detailed guidelines for representing an architectural description, and (3) to encourage sound architectural design practices. An *architectural description* (AD) represents multiple views, where each view is "a representation of a whole system from the perspective of a related set of [stakeholder] concerns."

13.1.4 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. Therefore, 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.

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 an architecture should be defined 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

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.

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.

13.2 Architectural Genres

Although the underlying principles of architectural design apply to all types of architecture, the architectural *genre* will often dictate the specific architectural approach to the structure that must be built. In the context of architectural design, *genre* implies a specific category within the overall software domain. Within each category, you encounter a number of subcategories. For example, within the genre of *buildings*, you would encounter the following general styles: houses, condos, apartment buildings, office buildings, industrial building, warehouses, and so on. Within each general style, more specific styles might apply (Section 13.3). Each style would have a structure that can be described using a set of predictable patterns.

KEY POINT

A number of different architectural styles may be applicable to a specific genre (also called an application domain).

In his evolving *Handbook of Software Architecture* [Boo08], Grady Booch suggests the following architectural genres for software-based systems that include artificial intelligence, communications, devices, financial, games, industrial, legal, medical, military, operating systems, transportation, and utilities, among many others.

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.**

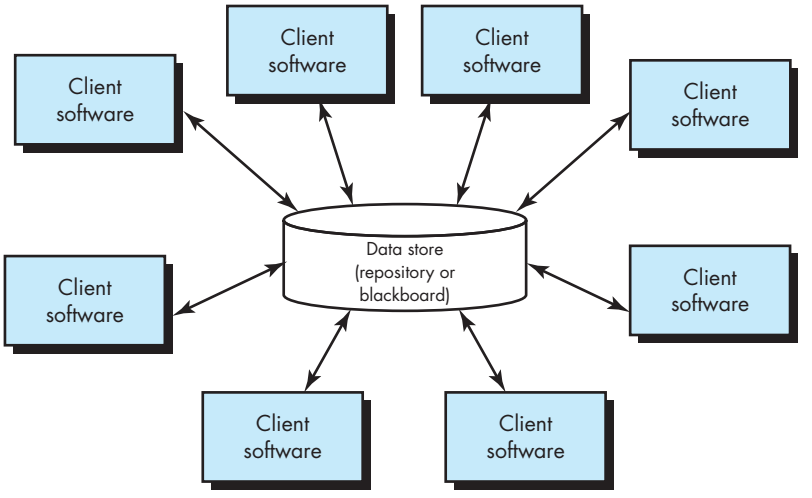
An architectural pattern, like an architectural style, imposes a transformation on the design of an architecture. However, a pattern differs from a style in a number of fundamental ways: (1) the scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture 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

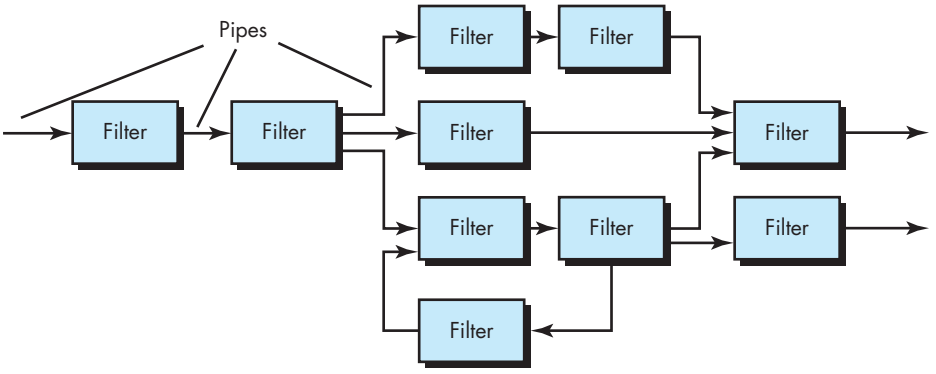
that update, add, delete, or otherwise modify data within the store. Figure 13.1 illustrates a typical data-centered style. Client software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or 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.

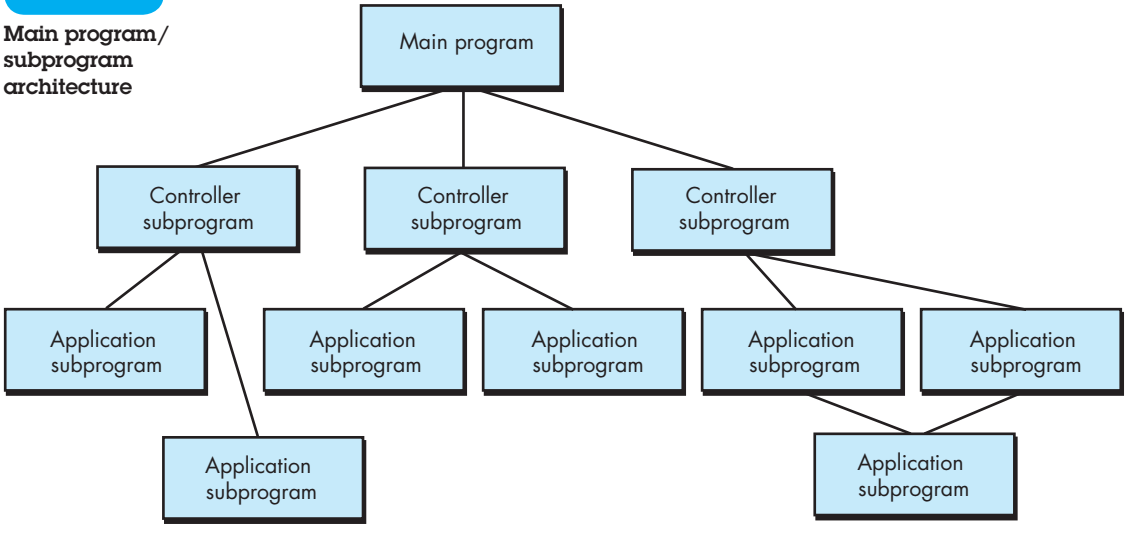
If the data flow degenerates into a single line of transforms, it is termed *batch sequential*. This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it.

FIGURE 13.2
Data-flow architecture



Pipes and filters

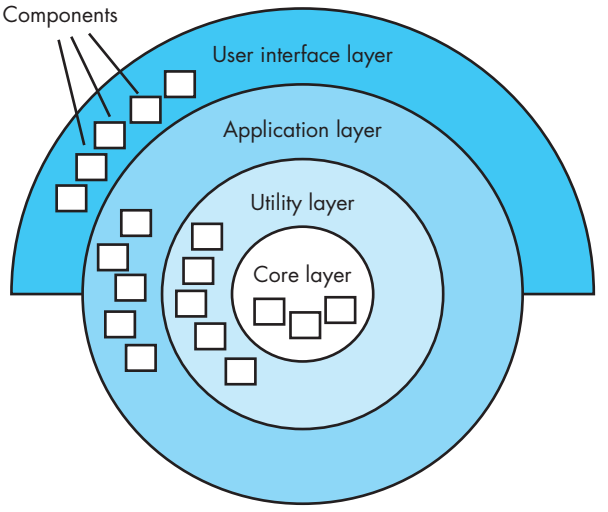
FIGURE 13.3
Main program/subprogram architecture



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.

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[3] 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 12.3]. 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.

reflect the domain and the domain logic of the business problem you want to solve with your application (Chapter 8).

A *problem frame* is a generalization of a class of problems that might be used to solve the problem at hand. There are five fundamental problem frames, and these are often associated with architectural styles: simple work pieces (tools), required behavior (data centered), commanded behavior (command processor), information display (observer), and transformation (pipe and filter variants).

Real-world problems often follow more than one problem frame, and as a consequence an architectural model may be a combination of different frames. 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) which manages access to the content (model) and instructs the information rendering model (view) to translate it for display by the browser software.

3 It can be argued that the *SafeHome* architecture should be considered at a higher level than the architecture noted. *SafeHome* has a variety of subsystems—home monitoring functionality, the company's monitoring site, and the subsystem running in the owner's PC. Within subsystems, concurrent processes (e.g., those monitoring sensors) and event handling are prevalent. Some architectural decisions at this level are made during product engineering, but architectural design within software engineering may very well have to consider these issues.

4 The MVC architecture is considered in more detail in Chapter 17.

Domain modeling can influence the choice of architectural style, particularly the core properties of domain objects. The domain objects that represent physical objects (e.g., sensors or drives) should be treated differently from those representing logical objects (e.g., schedules or workflows). Physical objects must obey stringent constraints like connection limitations or use of consumable resources. Logical objects may have softer real-time behaviors that can be canceled or undone. Domain-driven design is often best supported by a layered architectural style. [Eva04]

13.3.2 Architectural Patterns

Quote:

"Maybe it's in the basement. Let me go upstairs and check."

M. C. Escher

As the requirements model is developed, you'll notice that the software must address a number of 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 purchase our goods online?*

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.

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 16.

13.3.3 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 [Bas03] provide insight into an architectural style:

How do I assess an architectural style that has been derived?

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?

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 4) have become very popular. This implies the software architectures may need to evolve as each product increment is planned and implemented. In Chapter 12 we described this process as refactoring—improving the internal structure of the system without changing its external behavior.

13.4 Architectural Considerations

Buschmann and Henny [Bus10b, Bus10c] suggest several architectural considerations that can provide software engineers with guidance as architecture decisions are made.

? What issues should I consider as I develop a software architecture?

- **Economy**—Many software architectures suffer from unnecessary complexity driven by the inclusion of unnecessary features or nonfunctional requirements (e.g., reusability when it serves no purpose). The best software is uncluttered and relies on abstraction to reduce unnecessary detail.
- **Visibility**—As the design model is created, architectural decisions and the reasons for them should be obvious to software engineers who examine the model at a later time. Poor visibility arises when important design and domain concepts are poorly communicated to those who must complete the design and implement the system.
- **Spacing—**Separation of concerns in a design without introducing hidden dependencies is a desirable design concept (Chapter 12) that is sometimes referred to as *spacing*. Sufficient spacing leads to modular designs, but too much spacing leads to fragmentation and loss of visibility. Methods like domain-driven design can help to identify what to separate in a design and what to treat as a coherent unit.
- **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 the events that define the system's behavior is an emergent quality. It is very difficult to plan for every possible sequence of events. Instead the 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.

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 web-based version of this *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 web-based product.

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, since 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.

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].

13.5 Architectural Decisions

Decisions associated with system architecture capture key design issues and the rationale behind chosen architectural solutions. Some of these decisions include 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.

Quote:

"A doctor can bury his mistakes, but an architect can only advise his client to plant vines."

Frank Lloyd Wright

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.

A *guidance model* contains knowledge about architectural decisions required when applying an architectural style in a particular application genre. It is based 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

5 SOAD is analogous to the use of architecture patterns discussed in Chapter 16. Further information can be obtained at: http://soadecisions.org/soad.htm

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.

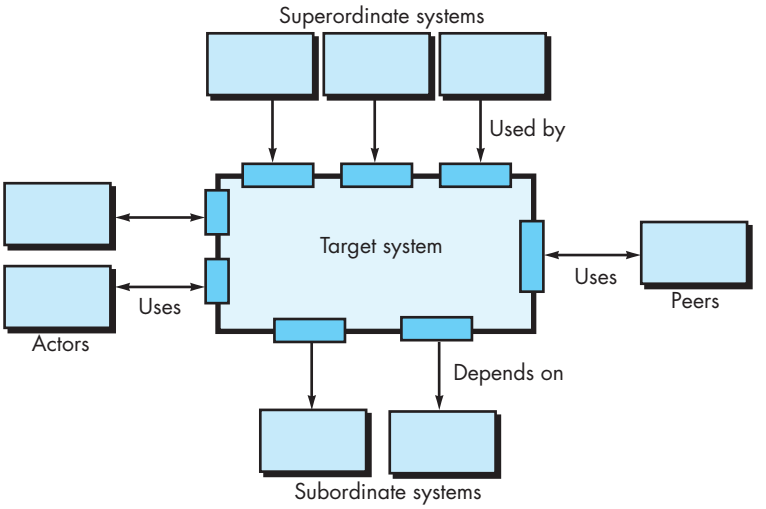


FIGURE 13.5
Architectural context diagram
Source: Adapted from [Bos00].

How do systems interoperate with one another?

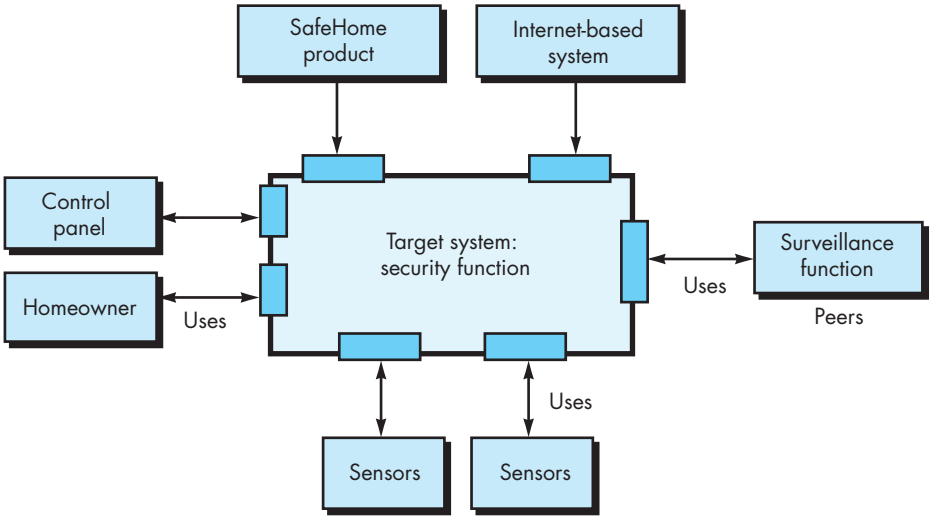
Referring to the figure, systems that interoperate with the *target system* (the system for which an architectural design is to be developed) are represented as:

- *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.

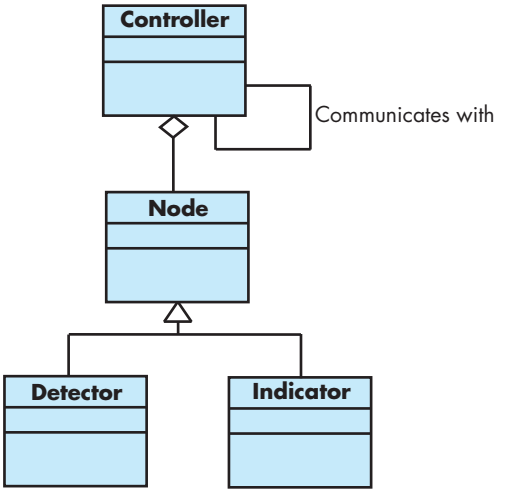


FIGURE 13.7
UML relationships for *SafeHome* security function archetypes
Source: Adapted from [Bos00].

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).

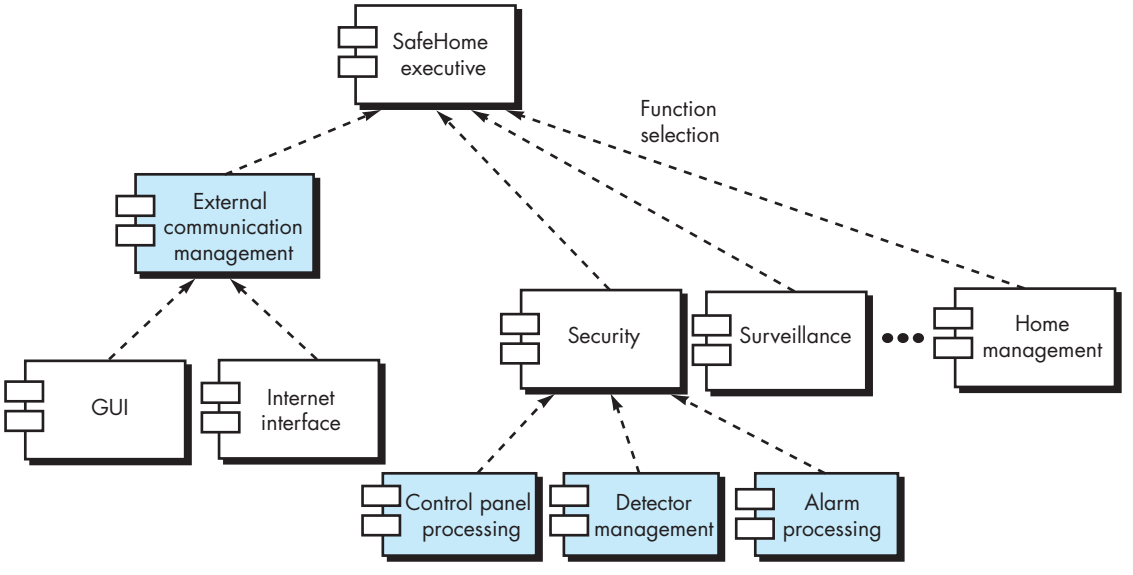
Continuing the *SafeHome* home security function example, you might define the set of top-level components that address the following functionality:

- *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



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

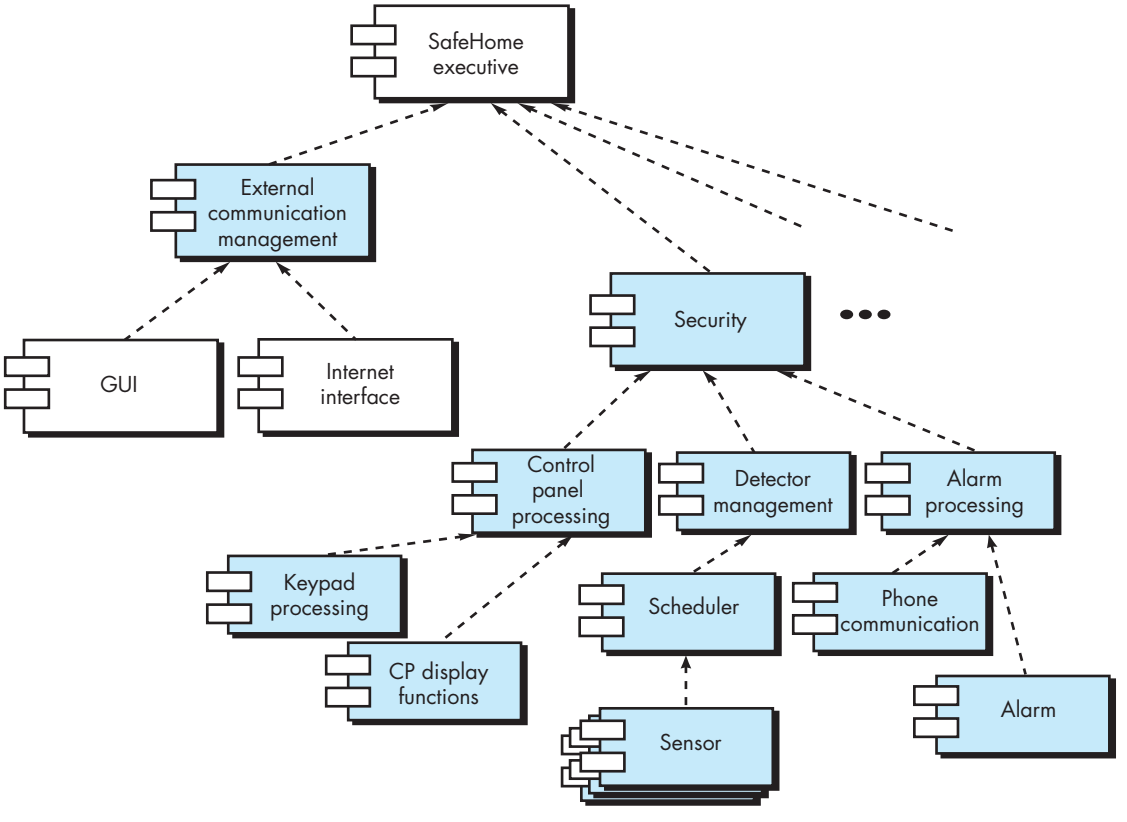
13.6.4 Describing Instantiations of the System

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 13.9 illustrates an instantiation of the *SafeHome* architecture for the security system. Components shown in Figure 13.8 are elaborated to show additional detail. For example, the *detector management* component interacts with

FIGURE 13.9 An instantiation of the security function with component elaboration



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.

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 8 and 9) 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/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.[10]

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 and 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 are going to 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, three years. You know, new versions, features, that sort of thing. We build a set of change scenarios. We also develop a set of quality scenarios that define the attributes we'd like to see in the software architecture.

Jamie: And we apply them to the alternatives.

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

13.7.1 Architectural Description Languages

Architectural description language (ADL) provides a semantics and syntax for describing a software architecture. Hofmann and his colleagues [Hof01] suggest that an ADL should provide the designer with the ability to decompose architectural components, compose individual components into larger architectural blocks, and represent interfaces (connection mechanisms) between components. Once descriptive, language-based techniques for architectural design have been

10 The *software architecture analysis method* (SAAM) is an alternative to ATAM and is well worth examining by those readers interested in architectural analysis. A paper on SAAM can be downloaded from www.sei.cmu.edu/publications/articles/saam-metho-propert-sas.html.

SOFTWARE TOOLS

Architectural Description Languages

The following summary of a number of important ADLs was prepared by Rickard Land [Lan02] and is reprinted with the author's permission. It should be noted that the first five ADLs listed have been developed for research purposes and are not commercial products.

xArch (http://www.isr.uci.edu/projects/xarchuci/) a standard, extensible XML-based representation for software architectures.

UniCon (www.cs.cmu.edu/~UniCon) is "an architectural description language intended to aid designers in defining software architectures in terms of abstractions that they find useful."

Wright (www.cs.cmu.edu/~able/wright/) is a formal language including the following elements: *components* with *ports*, *connectors* with *roles*, and *glue* to attach roles to ports. Architectural styles can be formalized in the language with predicates, thus allowing for static checks to determine the consistency and completeness of an architecture.

Acme (www.cs.cmu.edu/~acme/) can be seen as a second-generation ADL, in that its intention is to identify a kind of least common denominator for ADLs.

UML (www.uml.org/) includes many of the artifacts needed for architectural descriptions—processes, nodes, views, etc. For informal descriptions, UML is well suited just because it is a widely understood standard. It, however, lacks the full strength needed for an adequate architectural description.

established, it is more likely that effective assessment methods for architectures will be established as the design evolves.

13.7.2 Architectural Reviews

Architectural reviews are a type of specialized technical review (Chapter 20) 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. The most common architectural review techniques used in industry are: experienced-based reasoning,[11] prototype evaluation, scenario review (Chapter 9), and use of checklists.[12] 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 14). Software engineers who conduct architectural reviews note that architectural work products are sometimes missing or inadequate, thereby making reviews difficult to complete [Bab09].

11 *Experience-based reasoning* compares the new software architecture to an architecture used to create a similar system in the past.

12 Representative checklists can be found at http://www.opengroup.org/architecture/togaf7-doc/arch/p4/comp/clists/syseng.htm

13.8 Lessons Learned

Software-based systems are built by people with a variety of different needs and points of view. Therefore, a software architect should build consensus among members of the software team (and other stakeholders) in order to achieve the architectural vision for the final software product [Wri11].

WebRef

Examples of software architectural design lessons learned can be found at **http://www.sei.cmu.edu/library/abstracts/news-at-sei/01feature200707.cfm**

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 should work to achieve consensus that the software architecture chosen has distinct advantages over any other alternatives.

Wright [Wri11] suggests the use of several *decision analysis and resolution* (DAR) methods that may help to counteract some hindrances to collaboration. These methods can help increase active team member participation and increase the likelihood of their buy-in to the final decision. DAR methods help team members to consider several viable architectural alternatives in an objective manner. Three representative examples of DAR methods are:

WebRef

A discussion of pattern-based architecture reviews appears at **http://www.infoq.com/articles/ieee-pattern-based-architecture-reviews**

- **Chain of causes.** This technique is a form of root cause[13] analysis in which the team defines an architectural goal or effect and then enunciates the related actions that will cause the goal to be achieved.
- **Ishikawa fishbone.**[14] This is a graphical technique that identifies the many possible actions or causes required to achieve a desired architectural goal.
- **Mind mapping or spider diagrams.**[15] This diagram is used to represent words, concepts, tasks, or software engineering artifacts arranged around a central key word, constraint, or requirement.

13.9 Pattern-based Architecture Review

Formal technical reviews (Chapter 20) 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,

13 Further information can be obtained at: http://www.thinkreliability.com/Root-Cause-Analysis-CM-Basics.aspx

14 Further information can be obtained at: http://asq.org/learn-about-quality/cause-analysis-tools/overview/fishbone.html

15 Further information can be obtained at: http://mindmappingsoftwareblog.com/5-best-mind-mapping-programs-for-brainstorming/

a lightweight architectural review process known as *pattern-based architecture review* (PBAR) might be the best option.

PBAR is an evaluation method that leverages the relationship between architectural patterns[16] and software 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*[17] 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 9).
2. Discuss a diagram of the system's architecture in relation to its requirements.
3. Help the reviewers 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 makes 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.

13.10 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,

16 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 16.

17 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.

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].

WebRef
An overview of architecture conformance checking appears at **http://www.cin.ufpe.br/~fcf3/Arquitetura%20de%20Software/arquitetura/getPDF3.pdf**

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.

SOFTWARE TOOLS

Architectural-Conformance Tools

Lattix Dependency Manager (http://www.lattix.com/). This tool includes a simple language to declare design rules that the implementation must follow, detects violations in design rules, and visually represents them as a dependency-structure matrix.

Source Code Query Languages (http://www.semmle.com/). This tool can be used to automate software development tasks such defining and checking architectural constraints and makes use of a Prolog-like to define recursive queries on the inheritance hierarchy of object-oriented systems.

Reflexion Models (http://www.iese.fraunhofer.de/en/competencies/architecture/tools_architecture.html#contentPar_textblockwithpics). The SAVE tool can be used to allow software engineers to build a high-level model that captures the architecture of a system and then define the relations between this model and the source code. SAVE will then identify missing or erroneous relations between the model and the code.

13.11 AGILITY AND ARCHITECTURE

In the view of some proponents of agile development, 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 do 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, many stakeholders, or wide geographic distribution). For this reason, there is a need to integrate new architectural design practices into agile process models.

WebRef
A discussion of the role of architecture in agile software processes **http://msdn.microsoft.com/en-us/architecture/ff476940.aspx**

In order to make early architectural decisions and avoid the rework required and/or the quality problems encountered required when the wrong architecture is chosen, agile developers should anticipate architectural elements[18] and structure based on an emerging collection of user stories (Chapter 5). By creating an

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

architectural prototype (e.g., a walking skeleton) and developing explicit architectural work products to communicate to the necessary stakeholders, an agile team can satisfy the need for architectural design.

Agile development gives software architects repeated opportunities to work closely with the business and technical teams to guide the direction of a good architectural design. Madison [Mad10] suggests the use of a hybrid framework that contains elements of Scrum, XP, and sequential project management.[19] In this framework up-front planning sets the architectural direction, but moves quickly into storyboarding [Bro10b].

During 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. 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 require the iterative delivery of work products (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 architectural decision-making. It addresses when and how architectural decisions should be made and who on the project team makes them. 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 with stakeholder concerns from outside the team (e.g., business, security, infrastructure).

Agile teams insist on the freedom to make 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 agile teams from making decisions as 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 evolves.

19 Scrum and XP are agile process models and are discussed in Chapter 5.

13.12 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.

A number of 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.

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.

Problems and Points to Ponder

13.1. 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?

13.2. Present two or three examples of applications for each of the architectural styles noted in Section 13.3.1.

13.3. Some of the architectural styles noted in Section 13.3.1 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?

13.4. 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 for its counterparts.

13.5. Select an application with which you are familiar. Answer each of the questions posed for control and data in Section 13.3.3.

13.6. Research the ATAM (using [Kaz98]) and present a detailed discussion of the six steps presented in Section 13.7.1.

13.7. If you haven't done so, complete Problem 9.5. Use the design approach described in this chapter to develop a software architecture for the PHTRS.

13.8. Use the architectural decision template from Section 13.1.4 to document one of the architectural decisions for PHTRS architecture developed in Problem 13.7.

13.9. Select a mobile application you are familiar with, assess it using the architecture considerations (economy, visibility, spacing, symmetry, emergence) from Section 13.4.

13.10. List the strengths and weakness of the PHTRS architecture you created for Problem 13.7.

13.11. Create a dependency structure matrix[20] for the software PHTRS architecture created for Problem 13.7.

13.12. Pick an agile process model from Chapter 5 and identify the architectural design activities that are included.

Further Readings and Information Sources

The literature on software architecture has exploded over the past decade. Varma (*Software Architecture: A Case Based Approach,* Pearson, 2013) presents architecture in the context of a series of case studies. Books by Bass and his colleagues (*Software Architecture in Practice,* 3rd ed., Addison-Wesley, 2012), Gorton (*Essential Software Architecture,* 2nd ed., Springer, 2011), Rozanski and Woods (*Software Systems Architecture,* 2nd ed., Addison-Wesley, 2011), Eeles and Cripps (*The Process of Software Architecting,* Addison-Wesley, 2009), Taylor and his colleagues (*Software Architecture,* Wiley, 2009), Reekie and McAdam (*A Software Architecture Primer,* 2nd ed., Angophora Press, 2006), and Albin (*The Art of Software Architecture,* Wiley, 2003), present worthwhile treatments of an intellectually challenging topic area.

Buschman and his colleagues (*Pattern-Oriented Software Architecture,* Wiley, 2007) and Kuchana (*Software Architecture Design Patterns in Java,* Auerbach, 2004) discuss pattern-oriented aspects of architectural design. Knoernschilf (*Java Application Architecture: Modularity Patterns with Examples Using OSGi*, Prentice Hall, 2012), Rozanski and Woods (*Software Systems Architecture*, 2nd ed., Addison-Wesley, 2011), Henderikson (*12 Essential Skills for Software Architects,* Addison-Wesley, 2011), Clements and his colleagues (*Documenting Software Architecture: View and Beyond,* 2nd ed., Addison-Wesley, 2010), Microsoft (*Microsoft Application Guide*, Microsoft Press, 2nd ed., 2009), Fowler (*Patterns of Enterprise Application Architecture,* Addison-Wesley, 2003), Bosch [Bos00], and Hofmeister and his colleagues [Hof00] provide in-depth treatments of software architecture.

Hennesey and Patterson (*Computer Architecture,* 5th ed., Morgan-Kaufmann, 2011) take a distinctly quantitative view of software architectural design issues. Clements and his colleagues (*Evaluating Software Architectures,* Addison-Wesley, 2002) consider the issues associated with the assessment of architectural alternatives and the selection of the best architecture for a given problem domain.

20 Use Wikipedia as a starting point to obtain further information about the DSM at: http://en.wikipedia.org/wiki/Design_structure_matrix.

Implementation-specific books on architecture address architectural design within a specific development environment or technology. Erl (*SOA Design Patterns*, Prentice Hall, 2009) and Marks and Bell (*Service-Oriented Architecture,* Wiley, 2006) discuss a design approach that links business and computational resources with the requirements defined by customers. Bambilla et al. (*Model-Driven Software Engineering in Practice*, Morgan Claypool, 2012) and Stahl and his colleagues (*Model-Driven Software Development,* Wiley, 2006) discuss architecture within the context of domain-specific modeling approaches. Radaideh and Al-ameed (*Architecture of Reliable Web Applications Software,* IGI Global, 2007) consider architectures that are appropriate for WebApps. Esposito (*Architecting Mobile Solutions for the Enterprise*, Microsoft Press, 2012) discusses architecting mobile applications. Clements and Northrop (*Software Product Lines: Practices and Patterns,* Addison-Wesley, 2001) address the design of architectures that support software product lines. Shanley (*Protected Mode Software Architecture,* Addison-Wesley, 1996) provides architectural design guidance for anyone designing PC-based real-time operating systems, multitask operating systems, or device drivers.

Current software architecture research is documented yearly in the *Proceedings of the International Workshop on Software Architecture,* sponsored by the ACM and other computing organizations, and the *Proceedings of the International Conference on Software Engineering*.

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