Chapter 13 ARCHITECTURAL DESIGN

Chapter 13 ARCHITECTURAL DESIGN

Architectural design represents the 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.

Architectural design begins with data model and then proceeds to the derivation of one or more representations of the architectural structure of the system, including component properties and relationships (interactions) are described.

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.

Chapter 13 ARCHITECTURAL DESIGN

13.1 SOFTWARE ARCHITECTURE

13.1.1 What Is Architecture?

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.

Chapter 13 ARCHITECTURAL DESIGN

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.

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.

13.1.2 Why Is Architecture Important?

Bass and his colleagues identify three key reasons that software architecture is important:

Chapter 13 ARCHITECTURAL DESIGN

- **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.**
- **Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together".**

13.1.3 Architectural Descriptions

Different stakeholders will see an architecture from different viewpoints that are driven by different sets of concerns.

The architectural description needs to be concise and easy to understand since it forms the basis for negotiation particularly in determining system boundaries.

Chapter 13 ARCHITECTURAL DESIGN

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.

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.

As a system architect, you can use the template suggested in the sidebar to document each major decision. By doing this, you

Chapter 13 ARCHITECTURAL DESIGN

provide a rationale for your work and establish a historical record that can be useful when design modifications must be made. 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.

Chapter 13 ARCHITECTURAL DESIGN

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.

Chapter 13 ARCHITECTURAL DESIGN

13.2 ARCHITECTURAL GENRES

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.

Each style would have a structure that can be described using a set of predictable patterns.

In his evolving Handbook of Software Architecture , 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.

Chapter 13 ARCHITECTURAL DESIGN

13.3 ARCHITECTURAL STYLES

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.

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:

Chapter 13 ARCHITECTURAL DESIGN

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 13.1 illustrates a typical data-centered style.

Data-centered architectures promote integrability.

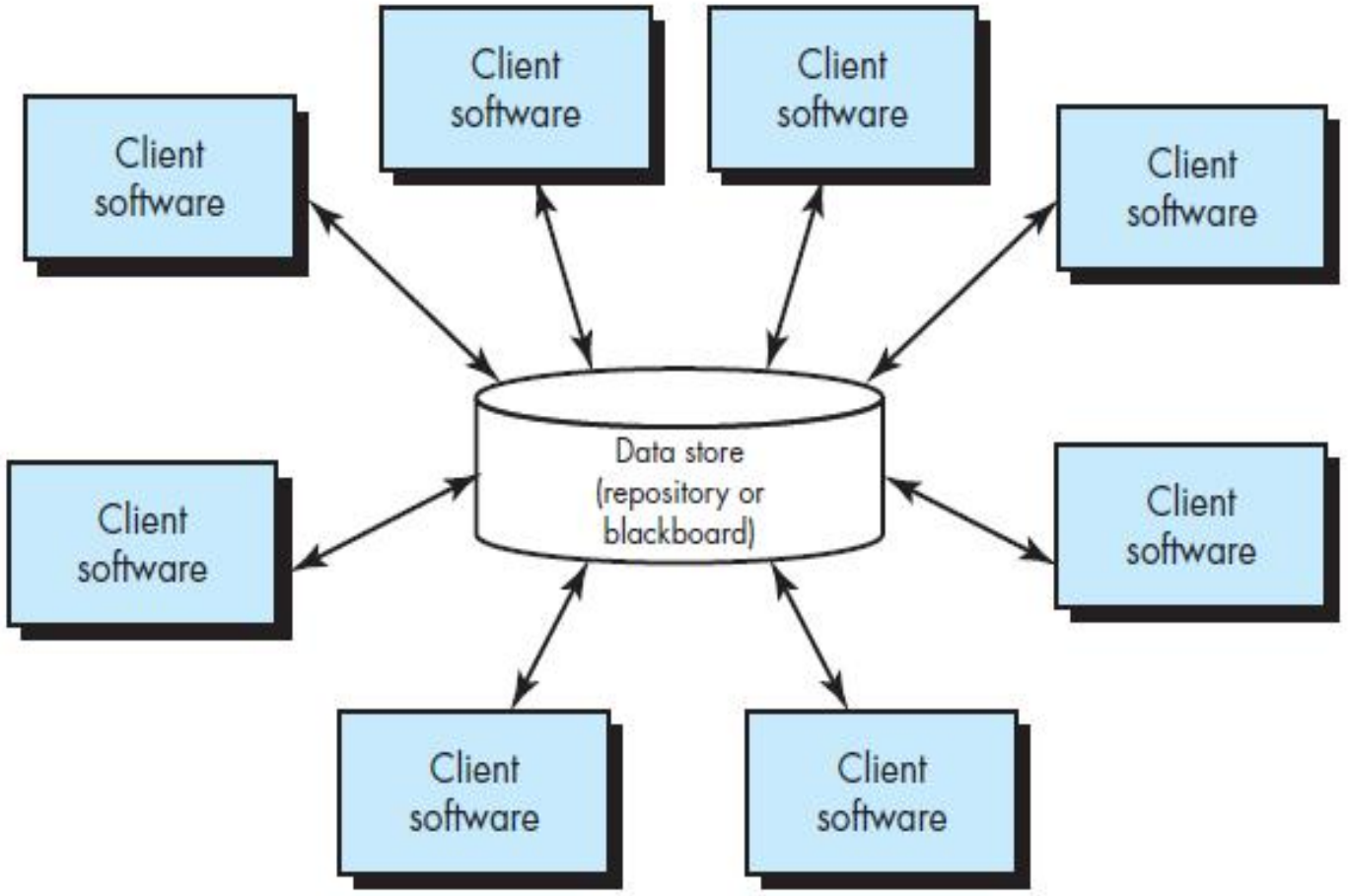
Data-Flow Architectures.This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern (Figure 13.2) has a set of components,called filters , connected by pipes that transmit data from one component to the next. Each filter works independently.

If the data flow degenerates into a single line of transforms, it is termed batch sequential .

Chapter 13 ARCHITECTURAL DESIGN

FIGURE 13.1

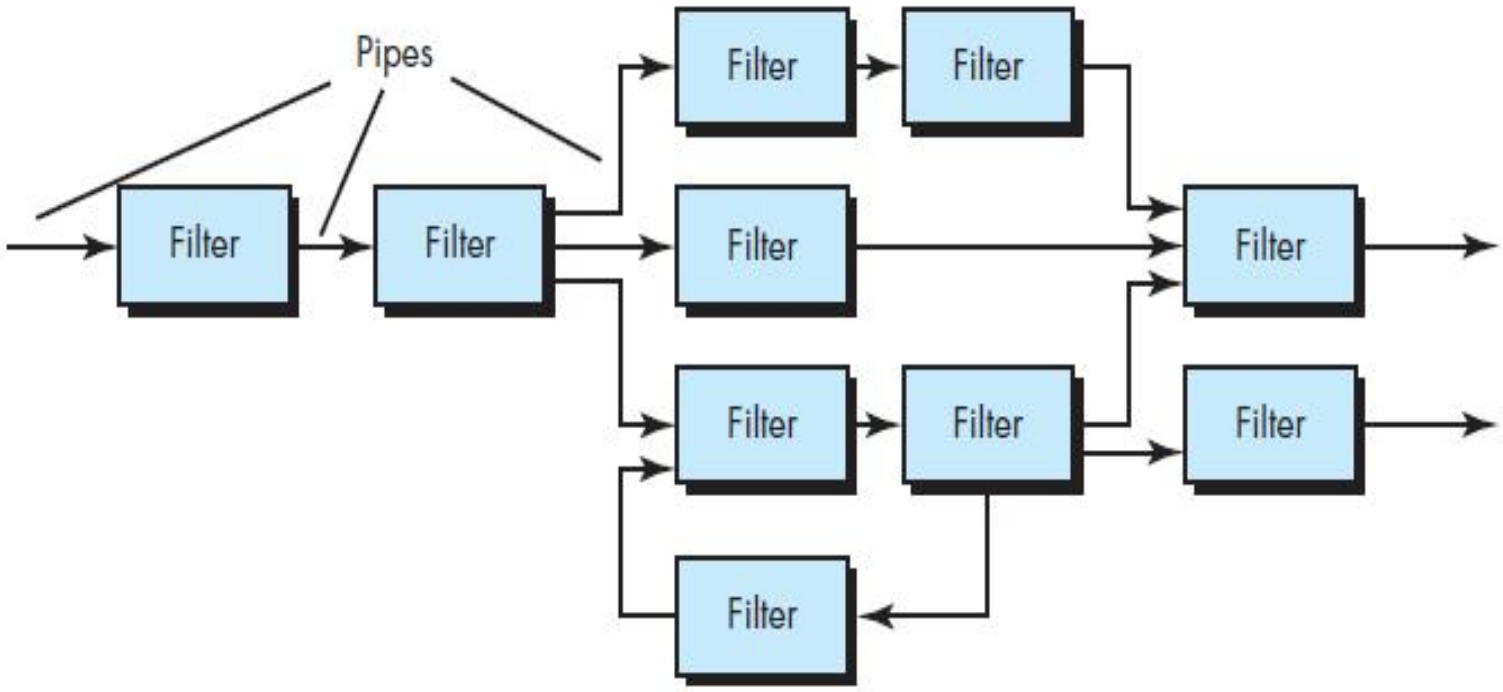
Data-centered architecture



Chapter 13 ARCHITECTURAL DESIGN

FIGURE 13.2

Data-flow architecture



Pipes and filters

Chapter 13 ARCHITECTURAL DESIGN

Call and Return Architectures. This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. For example:Main program/subprogram architectures(Figure 13.3)

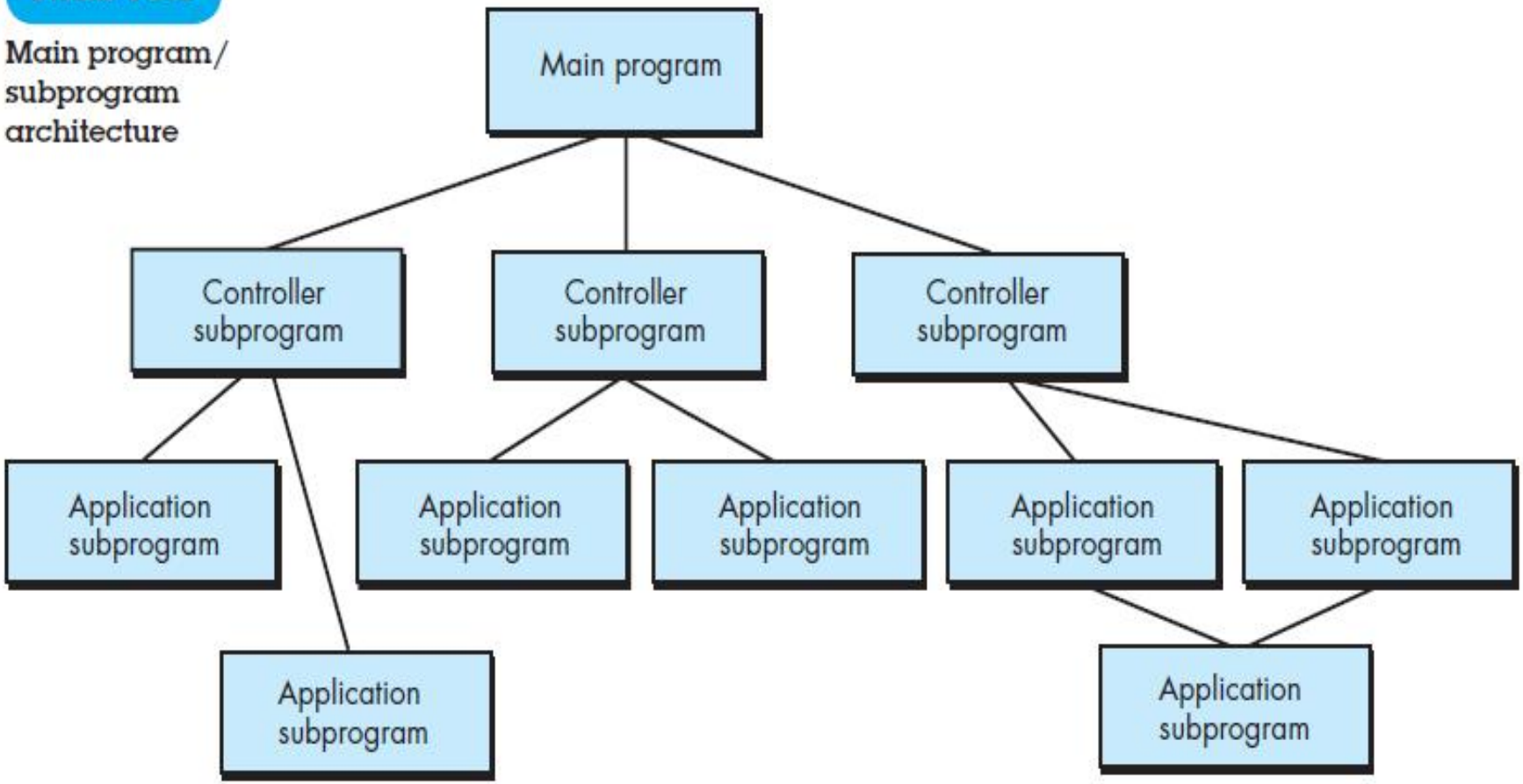
Object-Oriented Architectures. The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components are accomplished via message passing.

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. Analyse the model-view-controller (MVC) architecture.

Chapter 13 ARCHITECTURAL DESIGN

FIGURE 13.3

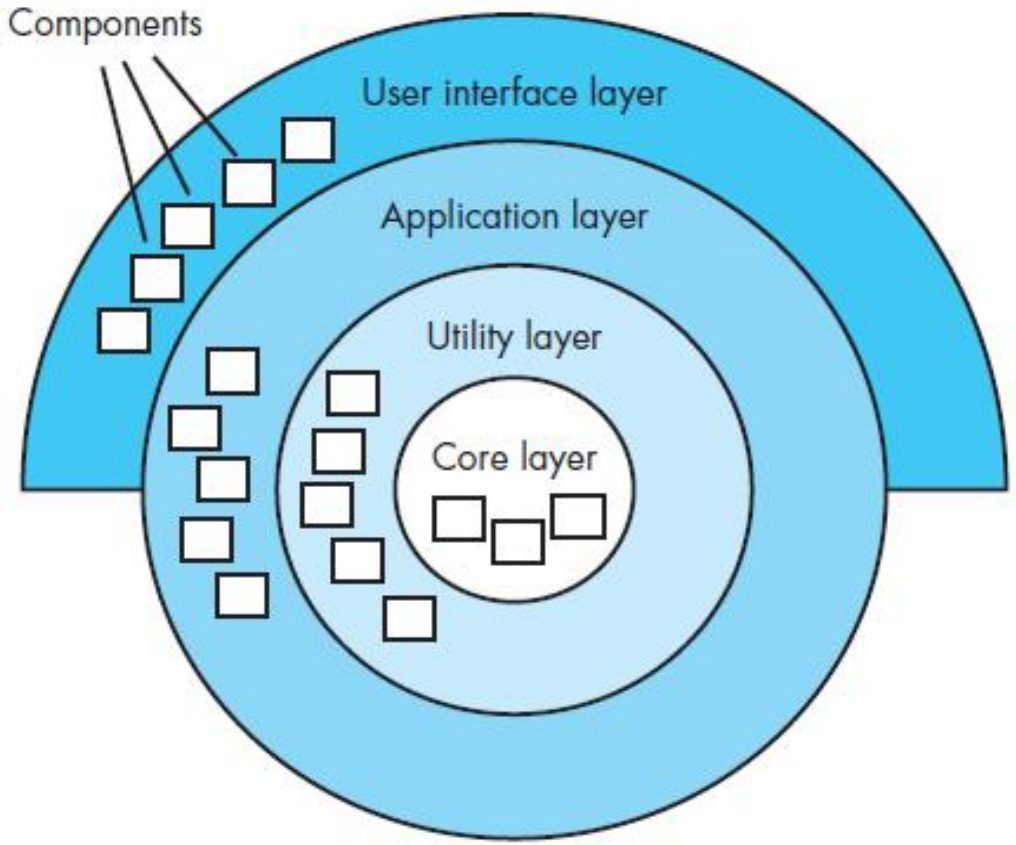
Main program/ subprogram architecture



Chapter 13 ARCHITECTURAL DESIGN

FIGURE 13.4

Layered architecture



Chapter 13 ARCHITECTURAL DESIGN

13.3.2 Architectural Patterns

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.

Chapter 13 ARCHITECTURAL DESIGN

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.

Evolutionary process models (Agile model, 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(read by yourselves)

Chapter 13 ARCHITECTURAL DESIGN

Buschmann and Henny suggest several architectural considerations that can provide software engineers with guidance as architecture decisions are made:

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).

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.

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.

Chapter 13 ARCHITECTURAL DESIGN

Symmetry —Architectural symmetry implies that a system is consistent and balanced in its attributes. Symmetric designs are easier to understand,comprehend, and communicate.

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.

These considerations do not exist in isolation. They interact with each other and are moderated by each other.

13.5 ARCHITECTURAL DECISIONS(read by yourselves)

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

Chapter 13 ARCHITECTURAL DESIGN

selection of structural elements（component） and their interfaces as defined by their intended collaborations, and the composition of these elements（components） into increasingly larger subsystems.

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 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.

Chapter 13 ARCHITECTURAL DESIGN

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.

Chapter 13 ARCHITECTURAL DESIGN

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.

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,Subordinate systems,Peer-level systems,Actors.*

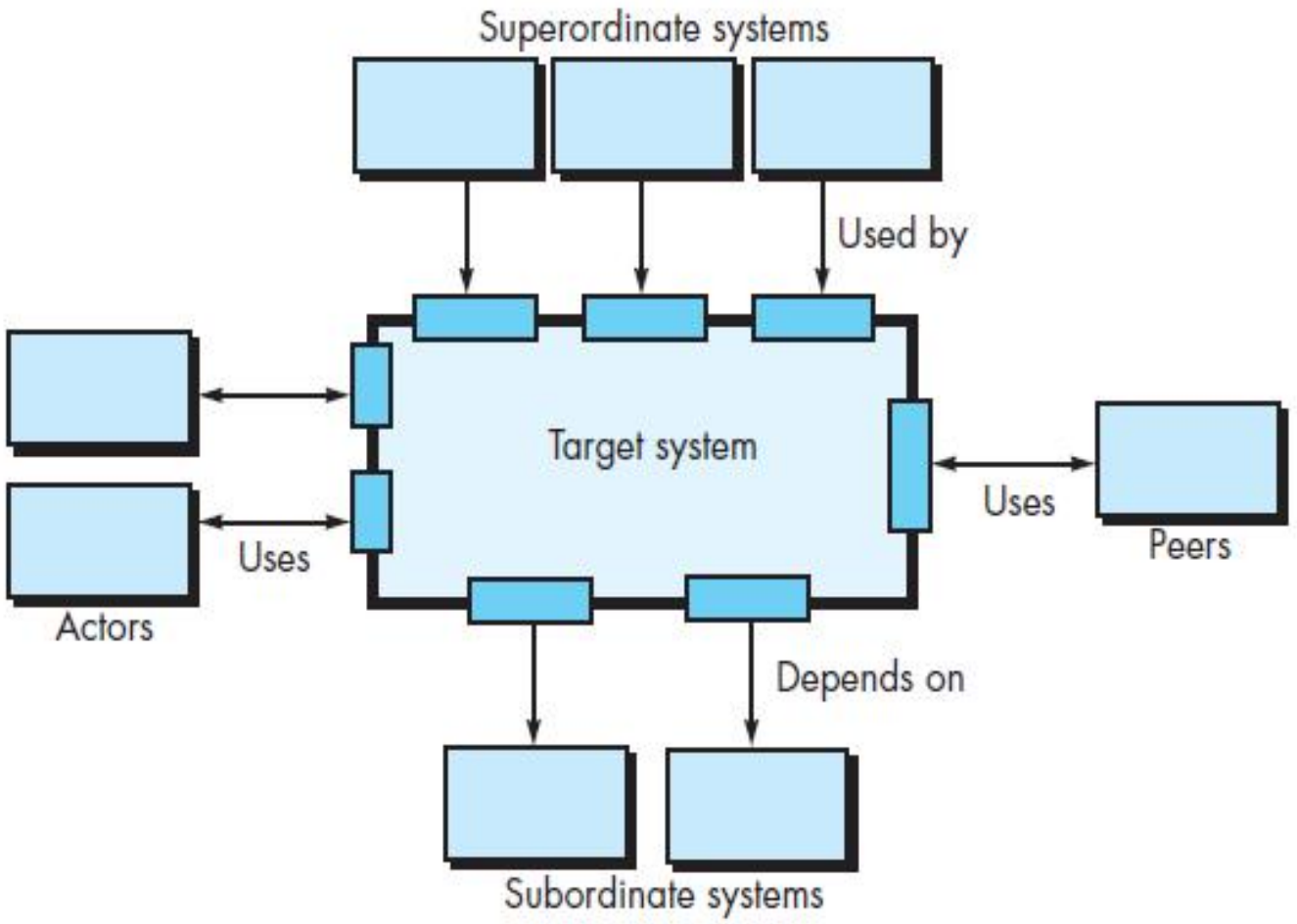
Each of these external entities communicates with the target system through an interface (the small shaded rectangles).

Chapter 13 ARCHITECTURAL DESIGN

FIGURE 13.5

Architectural context diagram

Source: Adapted from [Bos00].



Chapter 13 ARCHITECTURAL DESIGN

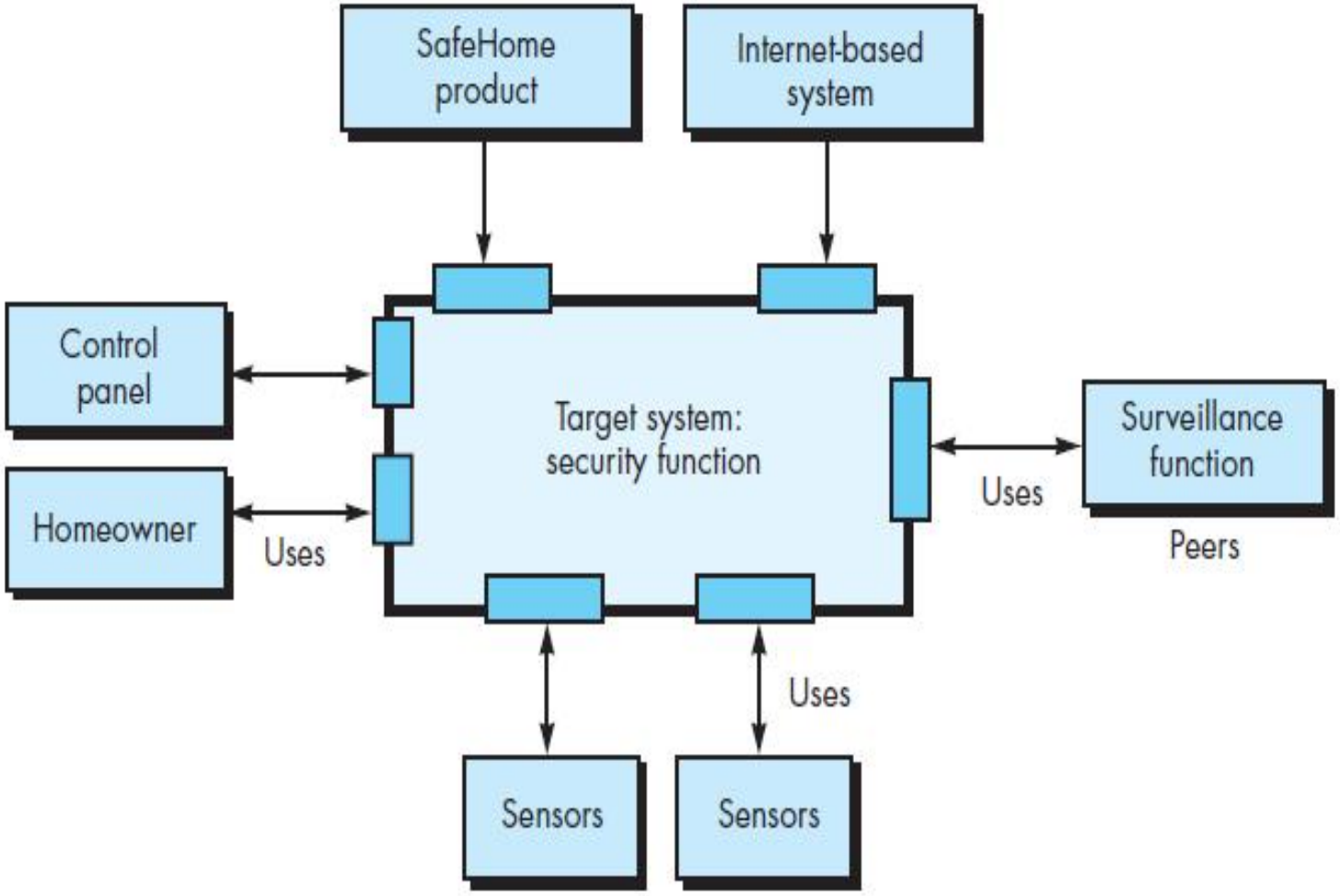
To illustrate the use of the ACD, consider the home security function of the Safe-Home product. The overall SafeHome product controller and the Internet-based system are both superordinate to the security function and are shown above the function in Figure 13.6 . The surveillance function is a peer system 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.

Other an example is books management system(relate to e-book,campus card, education management system ....)

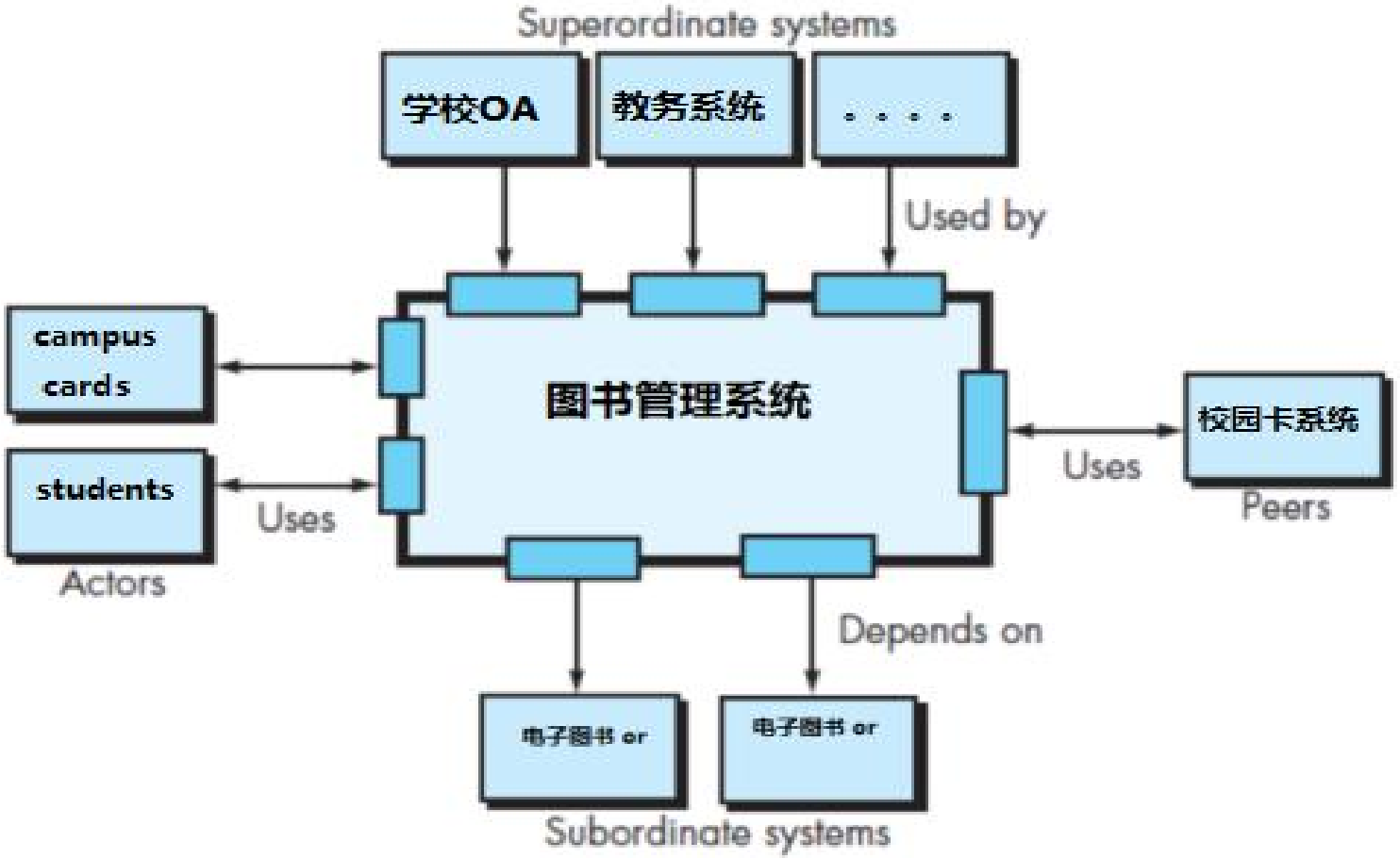
Chapter 13 ARCHITECTURAL DESIGN

FIGURE 13.6

Architectural context diagram for the *SafeHome* security function



Chapter 13 ARCHITECTURAL DESIGN



Chapter 13 ARCHITECTURAL DESIGN

Exercise:Give an other example to explain *Superordinate systems,Subordinate systems,Peer-level systems,Actors.*

As part of the architectural design, the details of each interface shown in Figure 13.5 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（Read by yourselves）

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.

Chapter 13 ARCHITECTURAL DESIGN

In many cases, archetypes can be derived by examining the analysis classes defined as part of the requirements model. you might define the following archetypes:

- **business analysis classes(entity classes).**
- **Detector. For exampel :sensor(device classes).**
- **Indicator. An abstraction that represents all mechanisms (e.g., alarm siren,flashing lights, bell) for indicating that an event is occurring(event classes).**
- **Controller. An abstraction that depicts the mechanism. If controllers reside on a network, they have the ability to communicate with one another (interface or control classes)**

Chapter 13 ARCHITECTURAL DESIGN

13.6.3 Refining the Architecture into Components（Read by yourselves）

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.

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.

Chapter 13 ARCHITECTURAL DESIGN

Another source is the infrastructure domain.The architecture must accommodate many infrastructure components that enable application. 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.

13.6.4 Describing Instantiations of the System（Read by yourselves）

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.

refer to the appendix “13章13.6节.architecture design example.doc”

Chapter 13 ARCHITECTURAL DESIGN

13.6.5 Architectural Design for Web Apps（Read by yourselves）

WebApps 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.

Chapter 13 ARCHITECTURAL DESIGN

The architectural design of a WebApp is also influenced by the structure 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.

13.6.6 Architectural Design for Mobile Apps（Read by yourselves）

Mobile apps 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.

Chapter 13 ARCHITECTURAL DESIGN

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. A number of considerations 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).

Chapter 13 ARCHITECTURAL DESIGN

13.7 ASSESSING ALTERNATIVE ARCHITECTURAL DESIGNS（Read by yourselves）

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.

The Software Engineering Institute (SEI) has developed an architecture tradeoff analysis method (ATAM) that establishes an iterative evaluation process for software architectures. The design analysis activities that follow are performed iteratively.

13.7.1 Architectural Description Languages

Architectural description language (ADL) provides a semantics and syntax for describing a software architecture. Hofmann and his

Chapter 13 ARCHITECTURAL DESIGN

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.

colleagues 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.

13.7.2 Architectural Reviews

Architectural reviews are a type of specialized technical review 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.

Chapter 13 ARCHITECTURAL DESIGN

The most common architectural review techniques used in industry are: experienced-based reasoning, prototype evaluation,scenario review (Chapter 9), and use of checklists(*http://www.opengroup.org/architecture/togaf7-doc/arch/p4/comp/clists/syseng.htm*). 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.

13.8 LESSONS LEARNED（Read by yourselves）

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.

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.

13.9 PATTERN-BASED ARCHITECTURE REVIEW（Read by yourselves）

Formal technical reviews 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 that leverages the relationship between architectural patterns 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

Chapter 13 ARCHITECTURAL DESIGN

walking skeleton is completed. The PBAR encompasses the following iterative steps:

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.**

Chapter 13 ARCHITECTURAL 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 OR REVIEW

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

Chapter 13 ARCHITECTURAL DESIGN

architecture. If architecture is not checked for conformance periodically, uncontrolled deviations can cause architecture erosion and affect the quality of the system.

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.

13.11 AGILITY AND ARCHITECTURE（Read by yourselves）

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 that it is important to focus on software architecture when a system is

Chapter 13 ARCHITECTURAL DESIGN

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.

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 and structure. By creating an 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.

Chapter 13 ARCHITECTURAL DESIGN

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.

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.