CHAPTER 1 Introduction to Software Architecture

Objectives of this Chapter

- Introduce the relationship between software requirements and architecture
- Introduce the relationship between architecture styles and architecture
- Introduce the elements of software architecture
- Describe quality attributes and tradeoff analysis

1.1 Overview

The goal of software design is to build a model that meets all customer requirements and leads to successful implementation. As software systems continue to grow in scale, complexity, and distribution, their proper design becomes extremely important in software production. Any software, regardless of its application domain, should have an overall architecture design that guides its construction and development. The success of a software product or system largely depends on the success of its architecture design.

What is the architecture design? "The architecture design defines the relationship between major structural elements of the software, the styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented" (Garlan and Shaw, 1996). The architecture design representation is derived from the system requirement specification and the analysis model.

Who is responsible for developing the architecture design? Software architects and designers are involved in this process. They translate (map) the software system requirements into architecture design. During the translation process, they apply various design strategies to divide and conquer the complexities of an application domain and resolve the software architecture.

Why is software architecture design so important? There are several reasons. A poor design may result in a deficient product that does not meet system requirements, is not adaptive to future requirement changes, is not reusable, exhibits unpredictable behavior, or performs badly. Without proper planning in the architecture design stage, software production may be very inefficient in terms of time and cost. In contrast, a good software design reduces the risks associated with software production, helps development teams work together in an orderly fashion, makes the system traceable for implementation and testing, and leads to software products that have higher quality attributes.

When is software design conducted? Software design is an early phase of the Software Development Life Cycle (SDLC). During this phase, software designers model the system and assess its quality so that improvements may be made before the software goes into the production phase. As shown in Figure 1.1, SDLC consists of the following stages: software requirements analysis; software design (architecture and detailed); software development and implementation; and testing and quality assurance, maintenance, and evolution. The dashed box in Figure 1.1 depicts the scope of software design. The Software Requirements Specification (SRS) provides the input necessary for design. SRS is the result of requirements analysis; it records the functional and nonfunctional requirements that must be met by the software system.

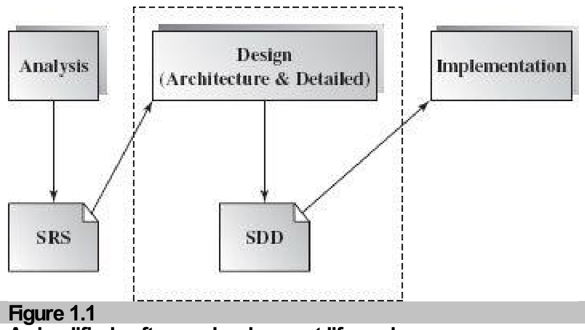


Figure 1.1
A simplified software development life cycle

What is the outcome of the software architecture design? Simply put, it is an overall representation of the software to be built. The IEEE Std 1016-IEEE Recommended Practice for Software Design Descriptions (SDD), shown in Figure 1.1, describes an organization for software design descriptions. The SDD serves as the blueprint for the implementation phase. It describes the elements of a system, the modules that compose each element, and the detailed information (such as data attributes, operations, and algorithms) of each module. The SDD is used as a template for software design description.

The following is a sample outline of SDD based on IEEE 1016.

- design overview, purpose, scope

- decomposition description (module, data, process)
- dependency and connection description (between modules, data, and processes)
- attributes
- user interface description
- detailed design (module and data)

Notice that the architecture design is a front strategy design for the detailed design. During the architecture design stage, a designer must specify user-accessible elements and the interconnections that are visible to stakeholders. Detailed design, also called tactical design, is concerned with local design constraints and the internal details of each element. For example, in the architecture design of a city traffic controller system, the designer can specify a priority queue that stores and dispatches incoming requests. In the detailed design, the designer must choose internal data structures from alternative solutions. For example, the priority queue can be implemented using a singly linked list, a doubly linked list, or an array. The designer must then document his reasons for selecting a particular internal data structure. In large-scale software design, the software architect may perform subsystem design before the detailed design.

We will now elaborate on the concepts of architecture design, which is the emphasis of this book. Take house construction as an analogy. Before construction begins, the builders need to know the requirements from customers and the architects must design blueprints. The architects have many options to choose from, such as the style (e.g., Victorian, Colonial, Cape Cod, etc.), functionality (e.g., vocational or residential), and features of the house (e.g., basement or sunroom). Similarly, the specifications of software elements, connectors, constraints (space, time, budget, etc.) and desired quality attributes (such as availability and performance) must be addressed in software design, and this is called the “software architecture,” or high-level design.

In practice, designers designate architecture styles by separating out common features of elements and connectors into “families of architecture.” Each style represents a layout topology of elements, and connectors and interactions among them. Each style also describes its semantic constraints and behaviors relating to data transfer and control transfer among the elements in the system, as well as the quality attributes tradeoff.

Software quality attributes include nonfunctional requirements such as performance, reliability, portability, usability, security, testability, maintainability, adaptability, modifiability, and scalability. Quality attributes are closely related to architecture styles in that each architecture style supports some quality features. An architecture style encapsulates the tradeoffs among many conflicting quality attributes. For example, with system performance, there is always a tradeoff between time/resources and system reliability and availability.

The rest of Chapter 1 is organized as follows: Section 1.2 elaborates on the notion of software architecture; Section 1.3 presents a general discussion of architecture styles; Section 1.4 discusses quality attributes; and Section 1.5 enumerates guidelines for software architects. The chapter concludes with a brief summary in Section 1.6.

1.2 Software Architecture: Bridging Requirements and Implementation

Software architecture plays a very important role in the Software Development Life Cycle. The architecture design provides a blueprint and guideline for developing a software system based on its requirement analysis specification. The architecture design embodies the earliest decisions that have a decisive impact on the ultimate success of the software product. The design shows how the system elements are structured, and how they work together. An architecture design must cover the software's functional and nonfunctional requirements as well. It serves as an evaluation and implementation plan for software development and software evolution.

The box-and-line diagram in Figure 1.2 shows what an architecture design typically looks like. Notice that it does not contain the complete set of information found in a development blueprint. For example, it does not provide enough guidelines for programmers to follow, nor does it describe any quality attributes. In Figure 1.2, each element (also called a “subsystem”) symbolizes a sole responsibility such as business logic processing, logic control, interface presentation, data gathering, and service brokering and mediating. This division of elements is based on their functionality, location, and runtime images. The elements may be in the form of modules, objects, packages, deployed elements, tasks, functions, processes, distributed programs, etc. The topology of the *static* structure focuses on the system composition configuration such as layered, flattened, star-typed, centralized, or distributed. The *dynamic* runtime connectors may be batch-sequential, multithreaded, explicit direct invocation, implicit indirect invocation (such as message queue or event notification), synchronous or asynchronous communication, peer-to-peer message exchange or message broadcasting, or another applicable coordination and cooperation mechanism among the elements.

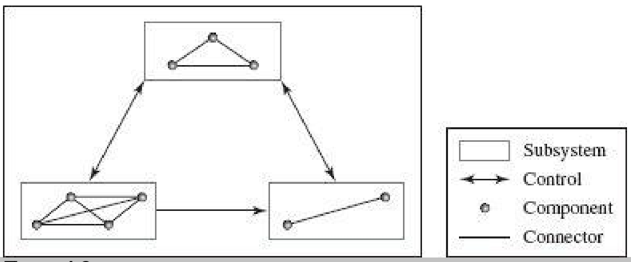


Figure 1.2

Box-and-line diagram showing subsystems

Figure 1.2 illustrates the idea of software architecture, but what is its formal definition? Here we list two definitions, one by IEEE and the other by Garlan and Shaw (1996).

IEEE Std 1471 defines system architecture as “the fundamental organization of a system embodied in its elements, their relationships to each other, and to the environment, and the principles guiding its design and evolution” (Maier, Emery, Hilliard).

Garlan and Shaw define software architecture as “the description of elements that comprise a system, the interactions and patterns of these elements, the principles that guide their composition, and the constraints on these elements” (1996).

In these definitions, the architecture elements can be a module, subsystem, object, or binary software such as a DLL component; a JavaBean, EJB, CORBA, or web component; or even a whole system. In this book we use “elements” to refer to the generic units of software architecture, and we use “component” as its synonym in discussions related to software architectures. Don't confuse this software component term with “component-based technology.”

Software design depends on the Software Requirement Specification (SRS) produced by analysis in the first step of SDLC. The requirements process covers information domain modeling, data modeling, function modeling, behavioral modeling, and user interface modeling. There are two aspects of software requirements: functional and nonfunctional. A functional requirement specifies the functionality of the software system whereas a nonfunctional requirement specifies system qualities, constraints, and behaviors.

There are many mechanisms used to specify a software requirement. The well-known box-and-line diagram in Figure 1.2 shows the conceptual analysis model of the system, which may be a starting point for software architecture design. However, a box-and-line diagram cannot fully capture the semantics of software architecture design because it does not provide the information necessary for software development in the next phase. Other descriptive notations also report the results of requirements analysis; these include Unified Modeling Language (UML) use-case specifications, Data Flow Diagrams (DFD), and State Transition Diagrams (STD). All these notations and tools can help software designers better understand the software requirements. However, they are conceptual models for analysis and not descriptions of the software architecture.

A complete software architecture specification must describe not only the elements and connectors between elements, but also the constraints and runtime behaviors so that developers know what and how the design should be implemented.

The following lists a software architect's tasks:

- Perform static partition and decomposition of a system into subsystems and communications among subsystems. A software element can be configured, delivered, developed, and deployed, and is replaceable in the future. Each element's interface encapsulates details and provides loose coupling with other elements or subsystems.
- Establish dynamic control relationships among different subsystems in terms of data flow, control flow orchestration, or message dispatching.
- Consider and evaluate alternative architecture styles that suit the problem domain at hand.
- Perform tradeoff analysis on quality attributes and other nonfunctional requirements during the selection of architecture styles. The selection of element type and connector type will have a direct impact on system properties and its quality attributes. Many quality attributes must be taken into account early in the design phase. For example, in order to increase a distributed system's extensibility, portability, or maintainability, software components and Web services may be the best choice of element types, and a loose connection among these elements may be most appropriate. The architects need to have stakeholders involved in this process.

The most important job of a software architect is to map the Software Requirements Specification to the software architecture design and guarantee that functional and nonfunctional requirements are met. If it is not possible to satisfy all requirements, system analysts and software architects can use the architecture designs to communicate with stakeholders.

1.3 Architecture Styles

An architecture style (also known as an “architecture pattern”) abstracts the common properties of a family of similar designs. An architecture style contains a set of rules, constraints, and patterns of how to structure a system into a set of elements and connectors. It governs the overall structure design pattern of constituent element types and their runtime interaction of flow control and data transfer. The key components of an architecture style are:

- elements that perform functions required by a system
- connectors that enable communication, coordination, and cooperation among elements
- constraints that define how elements can be integrated to form the system
- attributes that describe the advantages and disadvantages of the chosen structure

For example, in the data-centric style, the data store plays a central role and it is accessed frequently by other elements that modify data. In the dataflow style, input data is transformed by a series of computational or manipulative elements. In the call-and-return style, functions and procedures are the elements organized in a control hierarchy with a main program invoking several subprograms. In the object-oriented style, elements are represented as objects that encapsulate data and operations, and the communication among them is by message passing. In the layered style, each module or package completes tasks that progress in a framework from higher-level abstractions to lower-level implementations. All of these styles will be discussed in detail in later chapters.

For now, let us take a look at the multi-tier architecture style in detail. Multi-tier architecture is commonly used for distributed systems. It usually consists of three element types: client, middleware server, and data server.

Each element type serves a distinct function. The client element is responsible for GUI interface presentation, accepting user requests, and rendering results. The middleware element gets the requests from the client element, processes the requests based on the business logic, and sends a data request to the back-end tier. The data store server element manages data querying and updating. All three types of elements are connected via a network (e.g., the Internet). Many enterprise software architectures are of the multi-tier style because they share the same set of constraints.

Why are architecture styles so important? Because each style has a set of quality attributes that it promotes. By identifying the styles that a software architecture design supports, we can verify whether the architecture is consistent with the requirement specifications, and identify which tactics we can use to better implement the architecture.

Theoretically, an architecture style is a viewpoint abstraction for a software structure that is domain-independent. In most cases, a software system has its own application domain such as image processing, motor control, Web portal, expert system, or mail server. Each domain may have its own reference model. For instance, the Model-View-Controller (MVC) is widely adopted by designers of interactive systems. Such a reference model partitions the functionalities of a system into subsystems or software components. In many cases, a system can adopt heterogeneous architectures, i.e., more than one architecture style can coexist in the same design. It is also true that an architecture style maybe applied to many application domains.

1.4 Quality Attributes

Each architecture style has its advantages, disadvantages, and potential risks. Choosing the right style to satisfy required functions and quality attributes is very important. Quality attributes are identified in the requirement analysis process. Quality attributes can be categorized into the following three groups:

1. Implementation attributes (not observable at runtime)
   - *Interoperability:* universal accessibility and the ability to exchange data among internal components and with the outside world. Interoperability requires loose dependency of infrastructure.
   - *Maintainability and extensibility:* the ability to modify the system and conveniently extend it.
   - *Testability:* the degree to which the system facilitates the establishment of test cases. Testability usually requires a complete set of documentation accompanied by system design and implementation.
   - *Portability:* the system's level of independence on software and hardware platforms. Systems developed using high-level programming languages usually have good portability. One typical example is Java—most Java programs need only be compiled once and can run everywhere.
   - *Scalability:* a system's ability to adapt to an increase in user requests. Scalability disfavors bottlenecks in system design.
   - *Flexibility:* the ease of system modification to cater to different environments or problems for which the system was not originally designed. Systems developed using component-based architecture or service-oriented architecture usually possess this attribute.
2. Runtime attributes (observable at runtime)
   - *Availability:* a system's capability to be available 24/7. Availability can be achieved via replication and careful design to cope with failures of hardware, software, or the network.
   - *Security:* a system's ability to cope with malicious attacks from outside or inside the system. Security can be improved by installing firewalls, establishing authentication and authorization processes, and using encryption.
   - *Performance:* increasing a system's efficiency with regard to response time, throughput, and resource utilization, attributes which usually conflict with each other.
   - *Usability:* the level of human satisfaction from using the system. Usability includes matters of completeness, correctness, compatibility, as well as a friendly user interface, complete documentation, and technical support.
   - *Reliability:* the failure frequency, the accuracy of output results, the Mean-Time-to-Failure (MTTF), the ability to recover from failure, and the failure predictability.
   - *Maintainability (extensibility, adaptability, serviceability, testability, compatibility, and configurability):* the ease of software

system change.

3. Business attributes
   - *Time to market:* the time it takes from requirements analysis to the date a product is released.
   - *Cost:* the expense of building, maintaining, and operating the system.
   - *Lifetime:* the period of time that the product is “alive” before retirement.

In many cases, no single architecture style can meet all quality attributes simultaneously. Software architects often need to balance tradeoffs among attributes. Typical quality attribute tradeoff pairs include the following:

- *Tradeoff between space and time.* For example, to increase the time efficiency of a hash table means a decrease in its space efficiency.

- *Tradeoff between reliability and performance.* For instance, Java programs are well protected against buffer overflow due to security measures such as boundary checks on arrays. Such reliability features come at the cost of time efficiency, compared with the simpler and faster C language which provides the “dangerous,” yet efficient, pointers.

- *Tradeoff between scalability and performance.* For example, one typical approach to increase the scalability of a service is to replicate servers. To ensure consistency of all servers (e.g., to make sure that each server has the same logically consistent data), performance of the whole service is compromised.

When an architecture style does not satisfy all the desired quality attributes, software architects work with system analysts and stakeholders to nail down the priority of quality attributes. By enumerating alternative architecture designs and calculating a weighted evaluation of quality attributes, software architects can select the optimal design.

1.5 Software Architecture Design Guidelines

In the following section we provide several rules of thumb to help software developers better understand requirements, identify the right architecture styles to decompose a complex system into its constituent elements, choose the proper element and connector types, meet stakeholders’ requirements for quality attributes, and provide proper execution tactics for efficient implementation.

- *Think of what to do before thinking of how to do it.* Functional and nonfunctional requirements should be identified, verified, and validated before architecture and detailed design work is done. Using an abstract architecture design to communicate with stakeholders helps avoid the need to overhaul the system design in later stages of the software development cycle.

A successful architecture design relies on inherent iterative requirement analysis. Notice that different stakeholders of software systems have their own concerns. Software architects need to confirm what is needed and what can be traded off. For example, the investors of a project are usually concerned with the system release date, budget, usability, and so on; whereas the end users of the same project are concerned with performance, reliability, and usage scenarios. Thus, the software architect must be concerned with tradeoff analysis of the quality attributes, as well as the completeness and consistency of the architecture. Software developers, on the other hand, focus on implementation and are concerned with whether the software design is detailed enough for coding. Software project managers may be concerned with software architecture evolution and maintenance in the future.

- *Think of abstract design before thinking of concrete design.* Always start with an abstract design that specifies interfaces of components and abstract data types. Use multiple levels of abstraction if necessary. Make implementation decisions based on the abstract interfaces instead of the concrete ones because those are more stable—they are the contracts between service providers and service requesters, so they are defined at the early stages of the software development cycle.

- *Think of nonfunctional requirements early in the design process.* When you map functional requirements to an architecture design, you should consider nonfunctional requirements as well. Communicate with stakeholders and document their preferences for quality attributes. If it is not possible to find a design that meets all quality attributes, try to find the right balance of quality attributes and consider heterogeneous architecture styles when necessary.

- *Think of software reusability and extensibility as much as possible.* For most software systems, it is likely that new functionalities will be added after the systems are deployed. You need to consider how to reuse existing software components to increase the reliability and cost-effectiveness of new systems. Always try hard to make software extensible in the future.

- *Try to promote high cohesion within each element and loose coupling between elements.* A highly coherent subsystem, component, or module performs one sole function. For example, in object-oriented design, if a class is assigned to bear two unrelated responsibilities, it is regarded as incoherent. You must consider cohesion factors during the very early stages of the design process. Low cohesion of a system implies that functional composition is not designed well; for example, a single function can be scattered across a large number of different components, making it very hard to maintain.
  Each architecture style should show a clear division between elements to guarantee loose coupling. In most cases, loose coupling means less interdependency between components, so the change of one component is not likely to cause ripple-changes of other components. The coupling attribute can be measured by interface signature counts. Message passing and asynchronous communication are good examples of loose coupling between the service requestor and the service provider. For example, an email conversation has a much looser tie than that of a phone conversation.

- *Tolerate refinement of design.* Never expect to have software design completely perfect in one step. You may need to use prototyping and iteration to refine the design.

- *Avoid ambiguous design and over-detailed design.* Ambiguous design lacks constraints and over-detailed design restricts implementation.

How are architecture designs described? UML notation is one of many solutions, in addition to text documentation, that are available to designers. UML provides graphic notations that are available to architects and designers in nearly every stage of SDLC, e.g., use case diagrams for documenting system requirements, class diagrams for describing the logical structure of a system, and state machine diagrams and interaction diagrams for specifying the dynamic behaviors of a system. The "4+1" view model, developed by P. B. Kruchten, is a way to show different views of a software system, from the perspective of different stakeholders. It is especially useful in describing a complete set of functional and nonfunctional requirements. Another choice is to use Architecture Description Languages (ADL) to formally specify the structure and semantics of software architecture.

1.6 Summary

Software architecture design has emerged as an important part of software development. A software architecture specification consists of software elements, connectors and collaborations among the elements, and desired quality attributes. An architecture style is a set of rules, constraints, or patterns that guide how to structure a system into a set of elements and connectors, and how to govern overall structure design patterns of constituent element types and their runtime interaction. One specific architecture style may not satisfy all of the system's desired quality attributes, in which case tradeoffs must be made. Thus, how to properly balance quality attributes is an important design issue.

1.7 Self-Review Questions

1. The constituent elements of software architecture are software elements and their connections.
   a. True
   b. False
2. Software architecture design involves many software design methodologies and architecture styles.
   a. True
   b. False
3. The purpose of the software design phase is to produce a software requirement specification.
   a. True
   b. False
4. Object-oriented design is a design methodology.
   a. True
   b. False
5. Pipe-and-filter is one of the architecture styles.
   a. True
   b. False
6. Software architecture is a static software structure description.
   a. True
   b. False
7. Software quality attributes must satisfy functional requirements.
   a. True
   b. False
8. Architecture styles contribute to software quality attributes.
   a. True
   b. False
9. Software architecture = software architecture styles.
   a. True
   b. False
10. Software architecture design is based on the software requirement specification.
    a. True
    b. False

Answers to the Self-Review Questions

1. b 2. a 3. b 4. a 5. a 6.b 7.b 8. a 9.b 10. a

References

Garlan David and Mary Shaw. *Software Architecture: Perspectives on an Emerging Discipline.* Upper Saddle River, NJ: Prentice Hall, 1996, 1-4.

Maier M.W., D. Emery, R. Hilliard. "Software architecture: introducing IEEE Standard 1471." IEEE Xplore. Vol. 34, No. 4 (April 2001), http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/2/19820/00917550.pdf? temp=x.

Suggested Reading

Reekie John and Rohan McAdam. *A Software Architecture Primer.* Angophora Press, 2006.

CHAPTER 2 Software Architecture Design Space

Objectives of this Chapter

- Introduce major perspectives on, and structures used in, software architecture
- Introduce major element and connector types used in software architecture
- Introduce the iterative refinement process for software architecture design

2.1 Overview

A software architect is responsible for proposing a concrete architecture that best supports the detailed design and implementation of a specific project. Software architects must know what design alternatives are available to them, and which one will best support the functional and nonfunctional requirements. To put it another way, a software architect must understand the software architecture's design space.

In its simplest form, a software architecture design is a set of software elements connected by a set of connectors. From a dynamic structure point of view, a software element can be a process, an object, an instance of a software component, or a service. Different software elements may run on different hardware and software platforms and may be implemented in different programming languages or on different software frameworks. Two software elements can run in the same process, on the same computer system, within an intranet, or distributed over the Internet. Depending on their relative location, the connectors between a pair of software elements can be implemented in various forms including local method invocations, remote method invocations, service calls, and messaging through a message queue. The connectors can also work in synchronous or asynchronous nodes.

In terms of the static structure, a software element can be a package, a class, a component, or a loadable library. Correspondingly, a connector can be an import clause, inheritance clause, interface specification, pipe, or filter.

Today's software industry is characterized by constantly changing project requirements. An organization's expansion or merger may lead to heterogeneous intranet IT infrastructure, just as B2B (business-to-business) integration may make integration across the Internet critical to its smooth operation. A good software architecture should be able to easily adapt to these changing environments without the need for major reengineering of corresponding software systems.

Over the last decade, information technology has gone through significant changes. Component-based software engineering calls for the use of software frameworks. For example, technologies such as. NET and J2EE (Java 2 Enterprise Edition) have greatly enhanced the level of encapsulation. Web services and service-oriented architectures have brought us more flexible connector implementation technologies and software architecture varieties.

In the rest of this chapter we discuss the design space for software architectures and put in perspective the fundamental concepts behind the latest implementation technologies.

2.2 Types of Software Structures

As indicated previously, a software architecture design can be described with various software structures, each from a different perspective. It may be described in terms of software code units like source/binary code files, software modules, or software component deployment units; this is known as the static structure. It may also be described based on the runtime dynamic structure, in which the software elements are threads, processes, sessions, transactions, objects, or software component instances at execution time. Furthermore, an allocation structure may also be used to describe the project management structure of an architecture design. These different types of structures use different connector types and different performance attributes. We provide more details about these structural perspectives in the following subsections.

2.2.1 Software Static Structure

A software project is typically implemented in multiple files. This includes static file types such as executable files; library files; binary software component modules (usually in the form of DLLs [dynamic linking libraries], JavaBeans, and Enterprise JavaBeans); deployment descriptors; and other resource files.

At software development time, the main software elements are source code modules or files. Each module has assigned functional and nonfunctional attributes, and the public APIs (application programming interfaces), defined for each module separate the module's interfaces and implementations. The connectors at this level are module dependent. Module A is connected to module B if, and only if, A needs to invoke some methods in B during execution. Such connectors may exhibit the following attributes:

- *Direction:* If module A invokes a method of module B during execution, there is a unidirectional connector from module A to module B.

- *Synchronization:* A method invocation can be synchronous or asynchronous.

- *Sequence:* Some connectors must be used in a particular sequence. For example, module A may invoke a method of module B and pass a callback reference during the invocation. Later, some events in module B may trigger a callback to module A. Both of these method invocations are represented by their connector abstractions, and a sequence attribute associated with them consists of a sequence ID and number. In this case both connectors will have the same sequence ID but different sequence numbers, which indicates the order of method invocation. Note that the terms *method* and *method invocation* are used in a very general sense in this chapter. Normally, classes and methods will only be available at the detailed design phase, which takes place after a software architecture design has been chosen.

At software deployment time, the elements are binary versions of the project modules and files. Several source code modules may be packaged into the same deployment unit, but the connectors in the deployment structures are the same as those for the source module structures. Let us look at the software structure Java.

Classes are the basic building blocks of Java software. A Java program is a hierarchical collection of one or more *classes.* A large program consists of thousands of classes. *Files* are the *compilation units* in Java; that is, each file can be separately compiled. Packages allow the grouping of closely related classes and interfaces. Thus, they support the hierarchical and static organization of a large Java program as "logical and name space" managing units.

Package declarations are file-based, meaning that all classes in the same file belong to the same package (name space), if the source file contains a package declaration. When the package declaration is absent from a file, all the classes contained therein belong to an unnamed (anonymous) package. When packages are used, source and class files must be placed in directories whose structures match the structures of the packages. Naming the classes inside a package can be done by fully qualifying the name as follows: `package-name.class-name`. Alternatively, we can *import* a package, one of its subunits, or all of its classes.

Java units declared inside other units, such as packages, classes, or interfaces, yield a tree-like hierarchy. In contrast, importing separately compiled units defines a linear partial ordering which, when combined with the tree structure of subunits, defines the *software static structure.*

The software static structure refers to the organization of physical software modules and their interrelations and this structure plays a critical role in software architecture design. Static structure affects the architecture's clarity, construction strategy, maintenance, reengineering, reusability, etc. It plays a crucial role in the management of large software systems because it deals with the packaging of software modules in order to facilitate system construction and maintenance through a clear portrayal of intermodule relations. The fact that systems are developed incrementally increases the need for tight control of this structure in the physical software element.

Managing static structures involves layers of abstraction and of refinement showing visibility and encapsulation, respectively. These two notions define different kinds of hierarchical relations as described in the following:

- A linear client-server relation is formed when a component provides primitive abstractions to another component. In this sense components may refer to abstractions that, once defined, may be used throughout the entire design (at all levels). Layers of abstractions are connected when a module, the client, explicitly requests to use the facilities or abstractions provided by another module, the server. This relationship forms a linear hierarchy, whereby visibility is not transitive. Note that in support for reusability, server units must not know the identity of the client modules.

- A tree-like hierarchy of refinement relations is formed when an abstraction (i.e., a component) is implemented, and recursively divides into subcomponents. A refinement relation specifies how a module (parent) is decomposed into a refinement module (child). This relationship always defines a tree-like hierarchy. Inheritance is a special case of refinement relations.

Figure 2.1 illustrates the client-server and refinement relations essential for specifying the static structure. A server unit is an independently compiled unit available for use in the given scope, whereas a subunit (and its sub-subunits, and so on) is a refinement component of another unit, and hence exists only within the context of that unit.

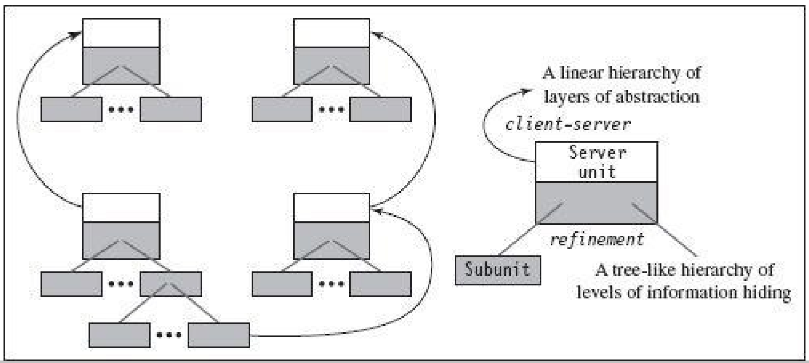

A linear hierarchy of layers of abstraction
client-server
Server unit
refinement
Subunit
A tree-like hierarchy of levels of information hiding

Figure 2.1
A summary of client-server and refinement relationships

2.2.2 Software Runtime Structure

At runtime a project consists of one or more threads, processes, functional units, and data units. These elements may run on the same computer or on multiple computers across a network. The same element in a code structure may implement or support multiple runtime elements. For example, in a client-server application, the same client module may run on many client computers. Conversely, several code structure elements may implement or support a single runtime element. For example, many threads will run multiple methods from different classes that maybe packaged in different code units.

The connectors at this level inherit attributes from their source-code structure counterparts, along with the following other attributes:

- *Multiplicity:* One element can be connected to multiple elements if it needs to invoke methods of multiple elements at runtime.

- *Distance and connection media:* Two connected elements may communicate in the same thread, in the same process, on the same computer, or on different computers across a network. Based on the distance between two elements, the communication media may vary from copper/optical cable or wireless based LAN to the Internet.

- *Universally invocable:* A connector with this attribute set to true allows any external software system, no matter what hardware/ software platforms they run on and in which programming languages or software frameworks they are developed, to invoke the method at the connector's target. This attribute is critical for heterogeneous enterprise information systems that must be integrated efficiently.

- *Self-descriptive:* A connector with this attribute set to true can allow an external software system to invoke its target method without the pre-installation of any software specific to the method. This attribute allows clients to choose service providers dynamically. It also allows software systems developed at different companies at potentially different times to dynamically interact with each other. For example, agents from different companies may be able to collaborate without special software installation.

2.2.3 Software Management Structure

A large software project is normally designed and implemented by several project teams, each having its well-defined responsibilities at specific SDLC process stages. At this level, each element consists of manipulation (design, implementation, debugging, etc.) of specific code units assigned to each project team, and the connectors are derived from runtime dependency among the code units and software process dependencies. Some software architectures are best implemented by a particular software management structure. Software management structures are also used for project resource allocation.

Software runtime structures serve as the technical backbone of architecture designs and provide the basis from which other structures are derived.

In this book we focus on software runtime structures and their efficient mapping to the best implementation technologies.

2.3 Software Elements

At runtime each software element has well-defined functions and connects to the other elements into a dependency graph through connectors. The elements of a software architecture are usually refined through multiple transformation steps based on their attributes and the project requirement specifications.

Depending on each software element's assigned function, there may be different synchronization and performance constraints. For example, some elements are reentrant objects or software components (meaning that multiple threads can execute in an element concurrently without interfering with each other) while some elements are not reentrant and no more than one thread may execute in it at any time. Depending on the multiplicity of an element, it could be invoked by a limited number of other elements at execution time, or it could be invoked by unlimited elements, as in the case of a server element. In the latter case, scalability, response time, and throughput become important performance constraints and must be considered during the element's implementation.

The following are the basic guidelines for mapping runtime elements into their implementations:

- If an element is reentrant, it can be implemented by a thread or a process. Reentrant elements are usually more efficient because they avoid many synchronization issues and support shared thread or process pools. However, business logics may not allow some elements to be reentrant.

- If an element is not reentrant and multiple threads or processes need to communicate with it, it must be run on separate threads or processes in order to be thread-safe (meaning that the system behavior is not affected by the multiplicity of threads executing concurrently).

- If an element has high multiplicity and its performance is important to the global system performance, an application server (a software system running business logics) should be used for the element's implementation so that it can take advantage of thread and resource pooling, data caching, and dynamic element life cycle management to conserve resources.

- If the elements contain heavy computations for deployment at a particular location, a cluster of processors will enhance CPU data processing power. The cluster size and the elements' mapping to the cluster computers should be done carefully to balance each cluster's computation load and minimize the total communication traffic on the cluster's network.

- If an element is assigned complex but well-defined functions, similar to those of some commercial off-the-shelf software components, and the performance of this element is not critical, then it is more cost-effective to use an existing software component to implement the element's functions.

- A complex element can be expanded into a subsystem with its own elements and connectors. A well-defined interface should be used to encapsulate the subsystem's design and implementation details from the existing architecture.

- A complex element can be transformed into a sequence of vertical layered elements if each layer provides a virtual machine or interface to its immediate upper-layer element, and each layered element hides away some low-level system details from the upper layers.

- A complex element can be transformed into a sequence of horizontally tiered elements if the business logic can be achieved by processing data with a sequence of discrete processing stages, and these processing stages can be implemented by tiered elements with well-defined interfaces and balanced workloads.

2.4 Software Connectors

The connectors in a software architecture are refined during the design process and are heavily influenced by a project's deployment environment. In the most abstract form, a connector indicates the necessity during system execution for one of the elements to send a message to another element and potentially get a return message. During software architecture refinement, if two elements are mapped to a single process, the connector can be mapped to a local method invocation. If two elements are mapped to two different processes on the same computer, the connector can be mapped to a local message queue or a pipe. If the two elements are mapped to two different computers, then remote method invocation or web service invocation can be used to refine the connector between them.

Software connectors are classified according to many attributes, including synchronization mode, initiator, implementation type, active time span, fan-out, information carrier, and environment. Based on the connector's *synchronization mode*, we can classify all connectors into two categories: *blocking connectors* and *non-blocking connectors*, as shown in Figure 2.2 (a). A blocking connector allows one of its incident elements to send a request (method call or message) to another and wait for a response (method return value or message). The element will be blocked from further execution until it receives a response. A non-blocking connector allows one of its incident elements to send a request (method call or message) to another and then continue its execution without waiting for a response.

Based on the connector's *initiator*, we can classify all connectors into two categories: *one-initiator connectors* and *two-initiator connectors*, as shown in Figure 2.2 (b). An initiator is an incident element of a connector that can make a request to its partner. A one-initiator connector allows only one of its two incident elements to make a request to the other element, but not the another way around. A two-initiator connector allows either one of its two incident elements to make a request to the other element. For a system to support callback between its two subsystems, the two subsystems must be connected by a two-initiator connector.

The information flow on a connector can be implemented using various *information carriers*, as shown in Figure 2.2 (c). If the two incident elements are in the same process, say as two threads, they may use a shared variable to exchange information. If they are mapped to different processes on the same processor, then resources like pipes, files, or local message queues may be used to implement the connector. *Method* invocations and *message* passing are more common and more structured ways for carrying information. Remote method invocation and messaging can also allow communication among elements deployed on different processors. Figure 2.3 shows that a message system, consisting of a message sender module and a message receiver module connected by a network, is used to implement a one-initiator connector for subsystem 1 to send messages/requests to subsystem 2. A message format must be defined so both the sender and the receiver can understand the messages, and a protocol must be adopted to determine the proper handshaking and synchronization between the two parties. The two small circles in the arrows connecting the message system to the two subsystems represent the two interfaces that connect the message system to its incident elements.

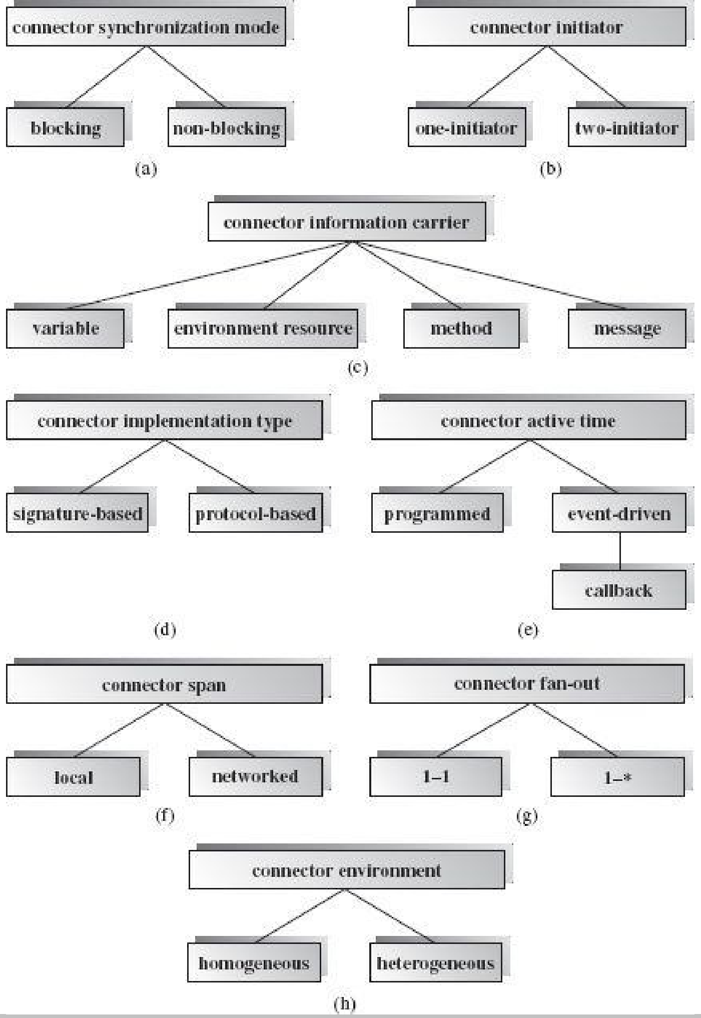


Figure 2.2
Connector type classification

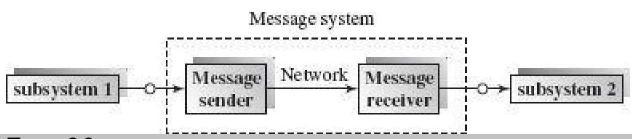


Figure 2.3
A simple message system

Based on the *implementation type*, a connector may be classified as *signature-based* or *protocol-based*, as shown in Figure 2.2 (d). For signature-based connectors, the method's name indicates an operation, and the parameters carry argument values for the operation's execution. If we assign one or more method parameter to indicate operation types, the connector can be used to implement protocols.

Whereas signature-based connectors can only be used to request one type of operation, a protocol-based connector can implement multiple operation types with a single binding signature. Furthermore, a protocol-based connector can support new operation types after the system interfaces are fixed. The connectors between an interpreter subsystem and its client subsystems are protocol-based. Message-based connectors support more flexible forms of protocols where all information about operations, arguments, and return values are coded in message formats and handshaking conventions among the involved parties. The HTTP protocol between web servers and web browsers is a familiar example of implementing a protocol-based connector.

Connector active time refers to when an operation request or message is sent over a connector. Connectors may be classified into *programmed connectors* and *event-driven connectors*, as shown in Figure 2.2 (e). Normally a method call will be made at a time specified during programming time: When execution comes to a line in a particular method, a call is made to another method. But for real-time systems, reactive systems, and any system with graphic user interfaces, an event-driven programming model becomes a much more flexible connection mechanism. One element will function as an event source, and all elements that need be notified of the event will register as listeners of the event source. When the event happens, all the registered listener elements will be notified for potential reaction.

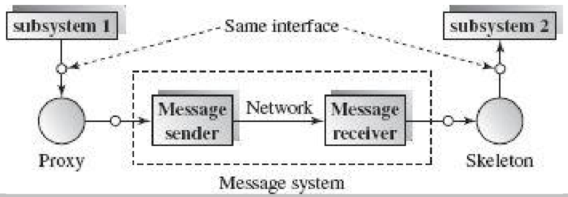


Figure 2.4
Networked connector implemented with the proxy design pattern

Method callback can be considered a special case of this event-driven mechanism. One element calls a method of another element, with one of its parameters passing a callback object/method (listener object) reference. When some event happens in the latter element, it will use the saved reference to call back and notify the first element. Event-driven connectors support late binding among subsystems.

Based on the *connector span* between incident elements, we classify the connectors as *local connectors* or *networked connectors*, as shown in Figure 2.2 (f). This connector attribute depends on whether the incident elements are located in the same processor. This attribute has major impact in the connector's implementation technologies. Networked connectors are normally implemented with the proxy design pattern to support object-oriented programming paradigm in a distributed environment. Suppose that there is a one-initiator and networked connector from subsystem 1 to subsystem 2, and subsystem 2 exposes an interface for subsystem 1 to invoke. A pair of proxy and skeleton objects will be generated from the interface by some technology-dependent tool, and they will be connected by a message system, as shown in Figure 2.4. The proxy object exposes the same interface as subsystem 2, and it is located in the same process as subsystem 1. Subsystem 1 gives the illusion that the proxy object is subsystem 2 deployed in the same process. When subsystem 1 invokes a method of the proxy object, the method body will send the operation and arguments to its skeleton partner, which is deployed in the same process as subsystem 2, over the network through the message system. The skeleton will then make the corresponding local method call against subsystem 2 and send the return value or any exceptions back to the proxy object over the message system. The proxy object will then send the received return value as its own to subsystem 1. One major advantage of this approach is that neither of the subsystems need be network-enabled at their design and implementation time.

Based on *connector fan-out* (the number of elements one element can connect to) we classify connectors as *1-1 connectors* and *1-* connectors*, as shown in Figure 2.2 (g). The 1-1 connectors are for connecting two elements only. The 1-* connectors are for connecting one element with a variable number of elements of the same type. For example, a web server and web browsers are connected with a 1-* connector, as are the server and clients in a client-server architecture. A connector's fan-out attribute may significantly impact connector implementation technology and performance.

Based on *connector environment*, which is the implementation technology or supporting platforms of a connector's two incident elements, we classify connectors into *homogeneous connectors* and *heterogeneous connectors*, as shown in Figure 2.2 (h). The incident elements of a homogeneous connector are implemented with the same programming language and software framework and run on the same operating system. The incident elements of a heterogeneous connector may be implemented with different programming languages or software frameworks and may run on different operating systems. CORBA, web services, and messaging are typical implementation technologies for heterogeneous connectors.

Heterogeneous connectors are usually implemented with the broker design pattern. This means that a message system might be implemented with the message sender and receiver modules implemented in different programming languages or on different platforms. Suppose that the two subsystems in Figure 2.4 are implemented in different programming languages and deployed on different platforms of two networked computer systems. This illustrates the broker design pattern with the following modifications.

First, the proxy object and the message sender module will be implemented in the same programming language, and run on the same computer system, as subsystem 1. The skeleton object and the message receiver module will be implemented in the same programming language, and run on the same computer system, as subsystem 2. Second, an application-level protocol will be defined to represent operations and argument values in a platform-and language-independent way. Both the proxy object and the skeleton object will support data marshaling (transforming data from a platform-or language-dependent form to the platform-and language-independent form) and unmarshaling (transforming data from the platform-and language-independent form to a platform, and language-dependent form). When the proxy object receives a method call from subsystem 1, it will marshal the argument values and send the resulting values and operation name to the skeleton object on the other side of the network, which will unmarshal the argument values into the form used by subsystem 2. Upon receiving the return value, the skeleton object will marshal it and send it back to the proxy object, which will then unmarshal the return value into the form used by subsystem 1 and return it as its own return value.

2.5 An Agile Approach to Software Architecture Design

Traditional software architecture designs, fundamentally based on a waterfall model (a linear process without integrating feedbacks), do not emphasize the iterative refinement nature and do not use element and connector attributes to capture the key architecture requirements of a software project. As a result there is big gap between a project's requirement specification and a concrete software architecture for its detailed design and implementation. Another weak point of traditional architecture design is that if the deployment environment changes, which is happening more often with the economy's globalization, the architecture design must start from scratch.

This book adopts an iterative, agile approach for developing software architectures that maximizes the reuse of architecture, design, and implementation investments. Given a project specification, an abstract high-level software architecture will first be proposed, and attributes will be identified for its elements and connectors. This abstract software architecture will generally be free of deployment considerations. The architecture will then go through multiple refinement processes to support particular deployment constraints. The unique features of this approach include the delayed binding of software connectors for more flexible implementation decisions and the seamless integration of multiple architecture styles in realizing different subsystems or levels of the same system.

In this section we incrementally extend an artificial system into a complex software architecture integrating multiple architecture styles. The resulting system is very similar to current web architecture. This example will illustrate how specification attributes can be used to refine an existing design recursively, to achieve the design objectives. This example also applies the most important software architecture styles, thus serving as a preview before their formal treatment in the following chapters.

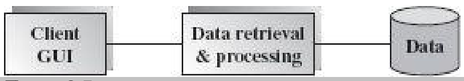


Figure 2.5
Stand-alone data presenter

Let us start with the design of an architecture for presenting data in a database to a client. This is a stand-alone application for serving data to a single user. Figure 2.5 shows a possible design, in which the *client GUI* module receives data retrieval criteria from the client and presents the selected data to the client in a graphical user interface. The *data retrieval & processing* module retrieves data from the database following client criteria and preprocesses the retrieved data. These two modules are supposed to run in different threads of the same process.

Now suppose the requirement specification changes and the application needs to run on a server to present the data over the Internet to multiple clients with matching client software. The connector between the *client GUI* module and the *data retrieval & processing* module now has a new *networked* attribute, as shown in Figure 2.6. Because all the clients will use the same client software to access the data server, the modules can be implemented in the same programming language using remote method invocation technology. If both of the modules are implemented in Java, then Java RMI can be used to implement the networked connector. If both modules are implemented in Windows, then Microsoft. NET remote invocation can be used to implement the networked connector.

In both of these examples, the connector between the *client GUI* module and the *data retrieval & processing* module is one-initiator, networked, and signature-based.

Now suppose we decide to support *client GUI* devices from third parties and present data in formats customizable on the server. Because we don't have control over the implementation technologies of the *client GUI* module, a message-and protocol-based connector can provide the needed flexibility. We use HTTP as the application-level protocol on top of the TCP/IP network connection between the client-side *client GUI* module and the server-side modules. For flexible data presentation that is modifiable on the server, we adopt the HTML markup language to specify how to present data to the client and submit client requests to the server through the HTTP protocol. As a result we introduce two tiers for data presentation: the server-side data presentation tier, implemented by the *HTML generator* module on the server, for dynamically generating HTML files; and the client-side data presentation tier, implemented by the *HTML presenter* module on the client side, for rendering data to the user according to the HTML markups. This is shown in Figure 2.7.

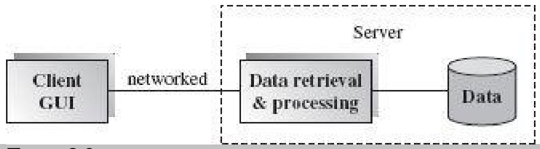


Figure 2.6
Networked data presenter

Now suppose we need to support significant data processing capability on the server. We apply the divide-and-conquer approach to the software architecture and divide the server-side software into three tiers: the presentation tier for generating HTML files, the business logic tier for serious data processing, and the data source tier for documents and data persistency. If we use the layered architecture style to design and implement the *HTML generator* and the *Data retrieval & processing* modules, our server-side presentation tier and business logic tier will be very similar to the web server and the application server of a typical web architecture. See Figure 2.8.

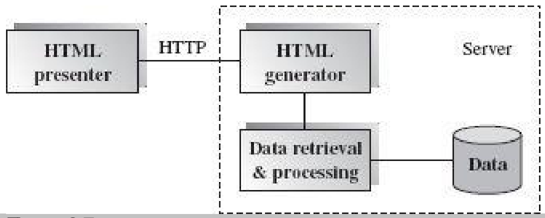


Figure 2.7
HTML and HTTP based data presenter

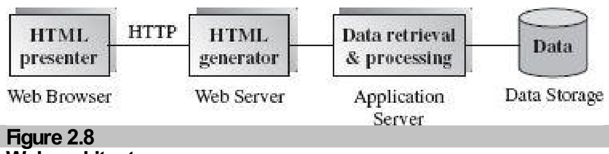


Figure 2.8
Web architecture

2.6 Summary

Software architecture determines the overall structure of a software system and greatly impacts its quality. The architecture can be viewed from multiple perspectives including the code structure (or static structure), runtime structure (or dynamic structure), and management structure (or deployment structure). Each type of structure consists of elements and connectors and their constraint attributes, which are derived from the requirements specification. To minimize the impact of changing project requirements and to maximize the ability to reuse design and implementation, an architect should adopt an iterative process during the design phase. Initial architecture designs should focus on the core functional and nonfunctional requirements; the resulting complex elements can then be refined into subsystems with their own architecture designs. A good architecture solution is typically based on multiple architecture styles for different subsystems or for different system abstraction levels.

2.7 Self-Review Questions

1. Which of the following structures describe the static properties of software architecture?
   a. Software code structure

   b. Software runtime structure
   c. Software deployment structure
   d. Software management structure
2. Which of the following structures describe the dynamic properties of software architecture?
   a. Software code structure
   b. Software runtime structure
   c. Software deployment structure
   d. Software management structure
3. Different architecture structures have different element and connector types.
   a. True
   b. False
4. Element and connector attributes are derived from the project requirements.
   a. True
   b. False
5. Architecture design is about choosing the right single architecture style for a project.
   a. True
   b. False
6. Divide-and-conquer is not a suitable methodology for architecture design.
   a. True
   b. False
7. Deployment decisions should be reflected in early architecture designs.
   a. True
   b. False

Answers to the Self-Review Questions

1. a, c, d 2. b 3. a 4. a 5.b 6.b 7.b

2.8 Exercises

1. Name at least one technology that can implement universally invocable connectors.
2. What types of connectors are used in standard four-tiered web architecture?
3. Name at least one technology that can implement self-descriptive connectors.
4. Is class inheritance a type of software architecture connector?
5. What are the main approaches to agile software architecture design?
6. What are the major types of connectors used in a university's online registration system?

2.9 Design Exercises

1. Design a high-level software architecture for a typical web-based business, and identify its major elements and connectors.
2. Design a high-level software architecture for a university's online registration system, and identify its major elements and connectors.

Suggested Reading

Bass, Len, Paul Clements, and Rick Kazman. *Software Architecture in Practice.* 2nd ed. SEI Series in Software Engineering, vol. 21, no. 26. Addison-Wesley, 2003.

CHAPTER 3 Models for Software Architecture

Objectives of this Chapter

- Introduce concepts of software architecture view models
- Discuss the UML notations as modeling tools for software architecture specification
- DiscussADLas a modeling tool for software architecture specification

3.1 Overview

Software architecture specifies a high level of software system abstraction by employing decomposition, composition, architecture styles, and quality attributes. Every software architecture must describe its collection of components and the connections and interactions among these components. It must also specify the deployment configuration of all components and connections. Additionally, a software architecture design must conform to the project's functional and nonfunctional requirements.

There are many ways to describe software architecture. Box-and-line diagrams are often used to describe the business concepts and processes during the analysis phase of the software development lifecycle. These diagrams come with descriptions of components and connectors, as well as other descriptions that provide common intuitive interpretations. Box-and-line diagrams will be used throughout this book for specification purposes.

Figure 3.1 presents a box-and-line diagram for an online shopping business where customers browse the catalog and put their selected items in a shopping cart. After a customer checks out, the system examines the customer's credit record, updates the inventory, and notifies the shipping department to process the order.

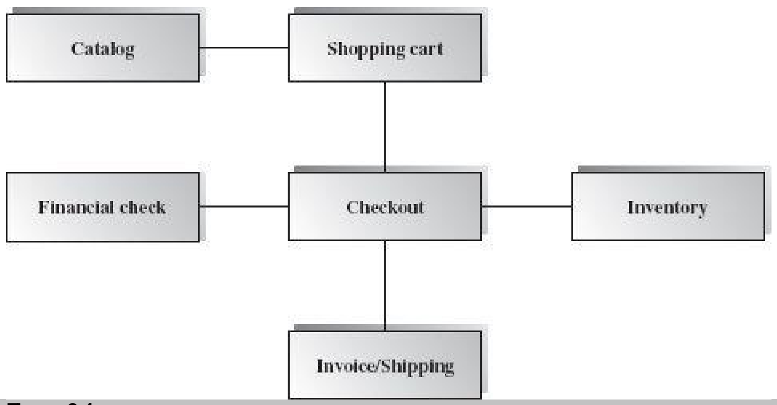


Figure 3.1
Box-and-line diagram

Lines in the box-and-line diagrams indicate the relationship among components. Notice that, unlike UML, the semantics of lines may vary—they may refer to dependency, control flow, data flow, etc. Lines may be associated with arrows to indicate the process direction and sequence.

A box-and-line diagram can be used as a business concept diagram describing its application domain and process concepts. This type of diagram helps us to understand the business concepts and to derive other software modeling and design diagrams such as UML. UML is one of the object-oriented solutions used in software modeling and design. We discuss this further in Section 3.2.

The 4+1 view model is another way to show the functional and nonfunctional requirements of a software project. There are five views in the model: the logical view, the process view, the development view, the physical view, and the user interface view. The logical view is used to identify software modules and their boundaries, interfaces, external environment, usage scenarios, etc. The process view addresses nonfunctional requirements such as module communication styles and performance issues at runtime. The development view organizes the software units in well-defined ways according to the actual file or directory structure. The physical view specifies the deployment infrastructure in terms of software, hardware, networking configurations, and installation for delivery purposes. All of these views work with, and are validated by, the scenarios view. The user interface view provides the look and feel of the system, and it may also impact other views. We explore the details of the 4+1 view model in Section 3.3.

The Architecture Description Language (ADL) is another way to describe software architecture formally and semantically. A simple software architecture specification example is demonstrated in Section 3.4.

3.2 UML for Software Architecture

The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. UML offers a standard way to draw a system's blueprints including conceptual notions such as business processes and system functions as well as concrete designs such as programming language statements, database schemas, and reusable software components. UML is a typical object-oriented analysis and design notation tool that provides many useful analysis diagrams and even generates code skeleton.

UML is widely used as a guideline for software requirement analysis and design documents, which are the basis for software development. Typically, UML can be used to model the problem domain; describe the user requirements; identify significant architecture elements during software design, such as classes and objects; describe behavior and interactions among these elements; and organize the software structure, specify its constraints, describe the necessary attributes, and more.

UML provides several modeling diagrams that can be grouped into two major categories: structural (static) and behavioral (dynamic). Structural software architecture describes the static structure of all software elements in a system: class hierarchy, class library structure, and relationships between classes such as inheritance (is a), aggregation (has a), association (uses a), and messaging (method invocation). Static structural UML diagrams depict the control flow between system elements, and are time-independent. These can be class diagrams, component diagrams, deployment diagrams, etc.

A dynamic software architecture describes the behavior of objects (i.e., instances of classes) in the system such as object collaboration, interaction, activity, and concurrency. The related UML diagrams are sequence diagrams, collaboration diagrams, activity diagrams, etc.

They are many UML IDE (Interactive Development Environment) tools available; some of them are open source. The most popular UML tools are Rational Rose, Boland Together, and Microsoft Visio. Many of these are capable of mapping from UML diagrams directly to coding framework in popular programming languages such as C++, C#, and Java.

The following table summarizes the 13 UML 2.0 diagrams in the structural and behavioral categories:

1. Structural (Static) Diagrams

Diagram	Description
Class	An overview of classes for modeling and design. It shows how classes are statically related, but not how classes dynamically interact with each other.
Object	Objects and their relationship at runtime. An overview of particular instances of a class diagram at a point of time for a specific case. It is based on the class diagram.
Composite structure	Describes the inner structure of a component including all classes within the component, interface of the component, etc.
Component	Describes all components in a system, their interrelationships, interactions, and the interface of the system. It is an outline of the composition structure of components or modules.
Package	Describes the package structure and organization. It covers classes in the package and packages within another package.
Deployment	Describes system hardware, software, and network connections for distributed computing. It covers server configuration and network connections between server nodes in real-world setting.

2. Behavioral (Dynamic) Diagrams

Diagram	Description
Use case	Derived from use-case study scenarios. It is an overview of use cases, actors, and their communication relationships to demonstrate how the system reacts to requests from external users. It is used to capture system requirements.
Activity	Outline of activity—s data and control flow among related objects. An activity is an action for a system operation or a business process, such as those outlined in the use-case diagram. It also covers decision points and threads of complex operation processes. It describes how activities are orchestrated to achieve the required functionality.
State machine	Describes the life cycle of objects using a finite state machine. The diagram consists of states and the transitions between states. Transitions are usually caused by external stimuli or events. They can also represent internal moves by the object.
Sequence	Describes time sequence of messages passed among objects in a timeline.
Interaction overview	Combines activity and sequence diagrams to provide control flow overview of the system and business process.
Communication	Describes the sequence of message passing among objects in the system. Equivalent to sequence diagram, except that it focuses on the object—s role. Each communication link is associated with a sequence order number plus the passed messages.
Time sequence	Describes the changes by messages in state, condition, and events.

In the following section, we provide details of the aforementioned diagrams.

3.2.1 Structural Diagrams

The structural description diagrams comprise class and object diagrams; component, structure, and package diagrams; and deployment diagrams. We discuss each in turn.

3.2.1.1 Class Diagram

The class diagram provides a static view of the system. It captures the vocabulary of the designed system. It is the foundation diagram of the system design and the most frequently used UML diagram as well.

Class diagrams can be derived from use-case diagrams or from text analysis of the given problem domain. A class diagram is generated by system analysts and designers and will be iteratively refined in the subsequent phases during the software development life cycle.

Class diagrams describe each individual class with its type, interface, properties, and methods. The accessibility (visibility) of each attribute and operation can also be specified. Popular accessibility modifiers include *private, public, protected*, and *default*.

One important part of a class diagram is the interface of each class. A class interface provides the behavioral contracts that the class must support.

There are three main relationships among classes: inheritance, aggregation, and association. These relationships can be represented graphically in a class diagram. For each relationship, multiplicities among classes can also be denoted. Typical multiplicity types include one-to-one, one-to-many, and many-to-many mappings. In UML multiplicity notations, 1 stands for one instance, 0 stands for no instance, 0..1 stands for zero or one instance, and 1..* stands for at least one instance.

Figure 3.2 shows a class diagram for an online purchase order processing system. Here we see all kinds of relationships among classes such as inheritance (represented using hollow triangle arrows), aggregation (represented using hollow diamond arrows), and association (lines without arrows). The multiplicity indicators are also shown. Generally, the diagram describes the logical structure of a purchase order system that consists of six classes. The `customer` class is the base class of `new` and `existing`. A `customer` can place zero or more `orders`. Each `order` consists of multiple `itemlines`, which in turn, contain `items`.

A class diagram may be refined from time to time during the software development life cycle. Object diagrams and component structure diagrams can be derived directly from a class diagram. Other dynamic behavioral diagrams such as sequence diagrams and communication (collaboration) diagrams are also based on the class diagram.

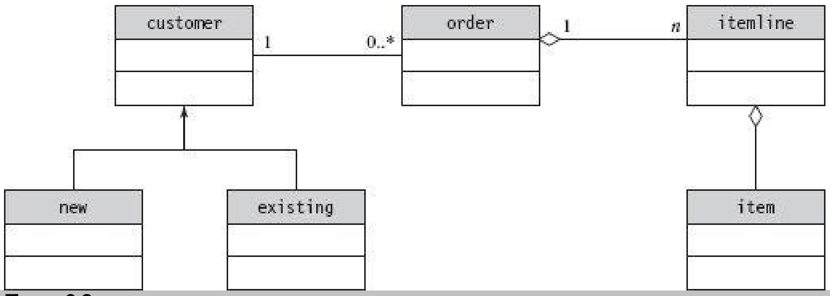


Figure 3.2
Class diagram

We will revisit class diagrams in Chapter 4 when we discuss the methodology of object-oriented design in detail. We will explore many issues related to specifying relationships among classes (e.g., inheritance, dependency, association, aggregation, and composition) in establishing the logical model of a system.

3.2.1.2 Object Diagram

Objects are the instances of classes. The object diagram is used to describe a sample subset of objects in the system at a specific point in time. This diagram shows a snapshot of class instance connection and interaction. It is derived from the preceding class diagram and is a concrete example of the class diagram at runtime. Many other behavioral diagrams (sequence diagrams, communication diagrams, and interaction diagrams) may make reference to the object diagram.

Figure 3.3 shows an object diagram based on the class diagram in Figure 3.2. Each rectangular box in the diagram represents an object that is an instance of some class. The diagram tells us that the customer with identification #1234 has ordered two items: book and gift.

3.2.1.3 Composite Structure Diagram

The composite structure diagram is used to describe the composition of interconnected elements or the collaboration of runtime instances. There are two basic notations in a composite structure diagram: collaboration (represented using a dashed eclipse) and structured class (represented using a rectangular box). Each structure class may have an annotation that indicates its role in the collaboration. For example, Figure 3.4 describes two classes involved in an `OrderProcess` collaboration. The `Customer` class plays the role of “buyer” and the `Order Processing System` plays the role of “seller.” Notice that `OrderProcess` is neither a class nor an object, it is a collaboration.

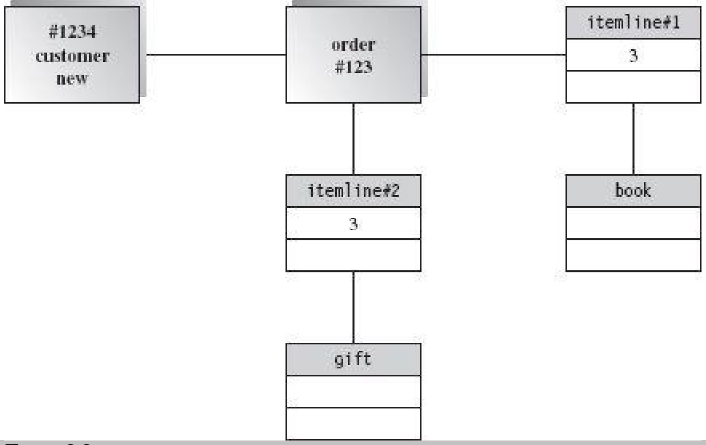


Figure 3.3
Object diagram

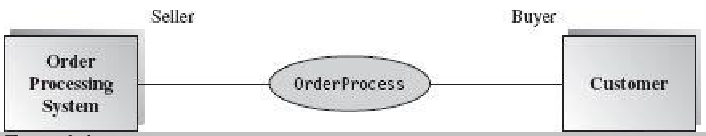


Figure 3.4
Simple composite structure

3.2.1.4 Component Diagram

A component is neither a class nor an object. A component is a deployable, reusable building block used in software design and development. For example, a JavaBean component is deployed in a jar file, an EJB component is deployed in an ear file, and a. NET component is deployed in a. dll file. Each component has an interface to expose its services and hide its implementations. The interface is the contract between a reusable component and its clients.

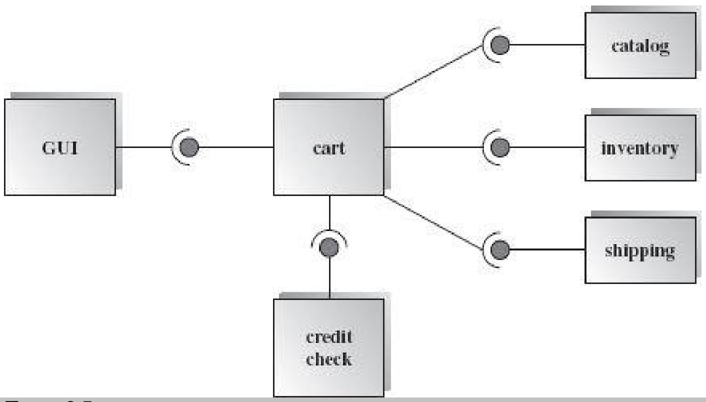


Figure 3.5
Component diagram

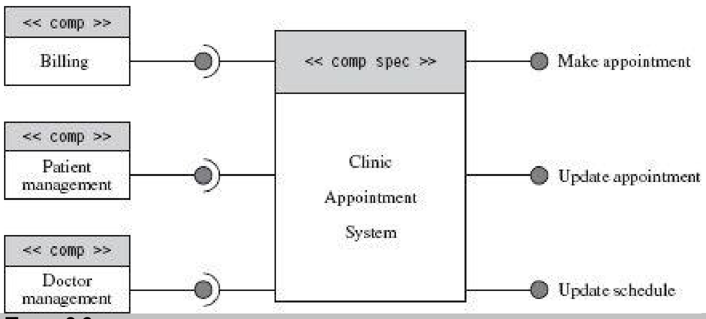


Figure 3.6
Another example of a UML 2 component diagram

UML 2.0 introduced a new notation for components and their connections. A lollipop shape of a component represents an implemented interface. A cup shape represents the required interface, and the required interface must be provided by some other components. In a component diagram, some of the components may exist and be available in-house or on the market. Other components are designed and developed by those working on the project.

Figure 3.5 shows a component diagram for a shopping cart application. The *cart* component provides services to front-end GUI interfaces such as ASP, JSP, or PHP web pages. The *cart* component itself may need services from other components: *catalog, inventory, shipping*, and *credit-check.*

The component diagram in Figure 3.6 shows four components: the *clinic* component, the *billing* component, the *patient* component, and the *doctor* component. The *clinic* component provides the following services (in the form of interface): *make appointment* by *patient, update appointment* by *patient*, and *update schedule* by *doctor.* The *clinic* component also needs services from the *billing* component, *patient* component, and *doctor* component.

3.2.1.5 Package Diagram

A package is represented by a tabbed folder that indicates where all included classes and subpackages reside. Packages play a similar role as a directory for grouping files in a file system; they allow the organization of all closely related classes in one “container.”

For example, namespaces in. NET and packages in Java provide well-formed structures for class accessibility and class correlations. We can organize functionally related classes in the same package so that these classes can access each other within a default accessibility or visibility. We can also organize related packages in a same parent package to build a class and package hierarchy just like. NET class library and Java API. Another reason for using the package organization is namespace sharing; in this way, all classes in the same package have a unique name but they may have the same name in different packages (namespaces).

A package diagram shows the dependency relationship between packages in which a change of one package may result in changes in other packages. The package diagram may also specify the contents of a package, i.e., the classes that constitute a package and their relationships. The use of package diagrams to represent system structures can help reduce the dependency complexity and simplify relationships among groups of classes.

Figure 3.7 shows a simple package diagram in which the *checkout* package, containing all *checkout-related* classes, depends on classes grouped in the *shopping cart* package. Same with the user *interface* package which has all GUI presentation classes to render the *catalog* and *shopping cart.* The package diagram also describes the dependency relationship of the package units. This diagram is often used for component-based software architecture design.

3.2.1.6 Deployment Diagram

A deployment diagram depicts the physical configuration of the software system deployed on hardware server nodes and the network between the nodes (defined as protocols). This diagram is generated in the later phase of the software development life cycle. All components in the system must be deployed on servers to provide services via network protocols. Component diagrams are the basis for deployment diagrams.

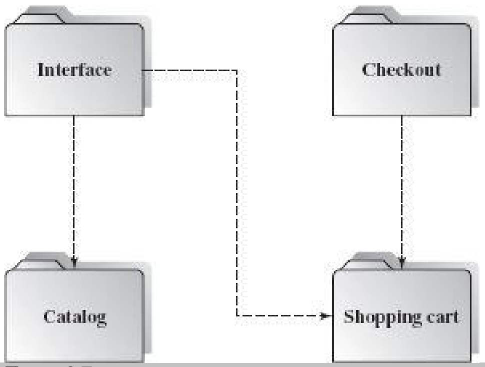


Figure 3.7
Package diagram

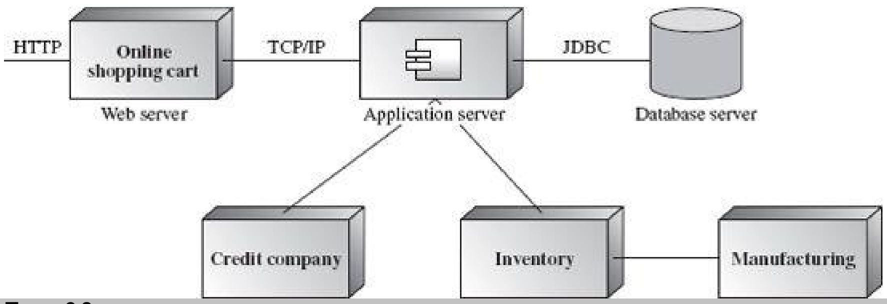


Figure 3.8
Deployment diagram

UML uses a cube symbol to represent a computing resource node; such a resource can be a hardware device or a deployed software subsystem. For example, data servers, web servers, and application servers can be nodes and are described by cubes in a deployment diagram. The link between nodes is the network connection depicted by the network protocol. The deployment diagram is used widely to model and design distributed software systems.

Figure 3.8 shows a deployment diagram in which a shopping cart is deployed in a web server, the business transaction component is deployed in a separate application server, and the database is available in a data server. The other services are available from three components deployed by the corresponding service providers.

3.2.2 Behavioral Diagrams

The behavior description diagrams comprise use case diagrams, activity diagrams, state machines, interaction diagrams, sequence diagrams, collaboration diagrams, and timing diagrams. We discuss each in turn.

3.2.2.1 Use Case Diagram

Use case diagrams describe user requirements in terms of system functionality as a contract between the users (actors) and the software system. This diagram consists of actors, use cases, and the links between them. An example of a use case is shown in Figure 3.9.

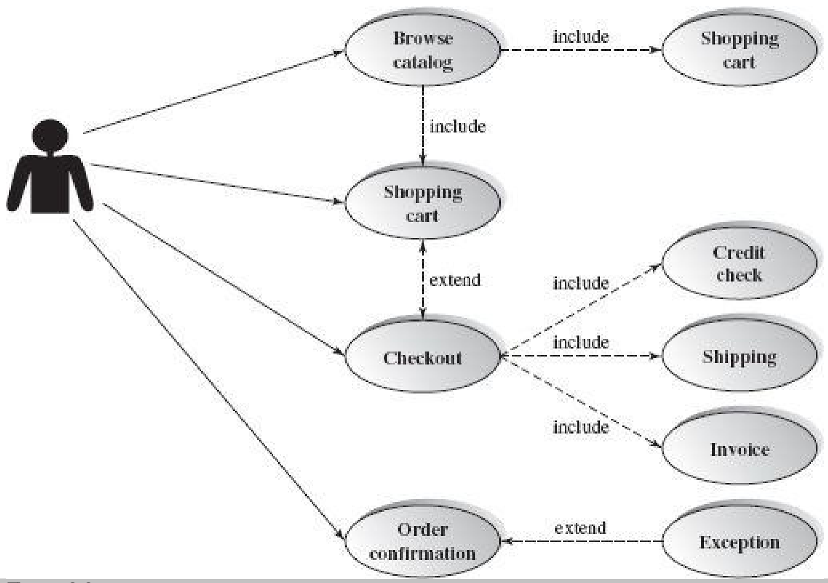


Figure 3.9
Use case diagram

An *actor* in a use case diagram is an external "user"; this may be a human end user, an application, an external device, or another system.

Each use case is a meaningful operational work scenario. That is, use cases are scenario-oriented in the sense that each is a sequence of working steps to be carried out by classes in order to provide the required system functionality. The detailed steps are specified in a separate note including the pre-and post-conditions of the action in the sequence.

The (simple) connection link from actor to use case shows the direction of actors using the use case. An `<<include>>` link, a special kind of link, from one use case to another, indicates that the first use case reuses or includes the second use case and is needed to complete the work necessary to fulfill the requirement. The `<<include>>` link is a dashed line with an arrow pointing to the reused use case. An `<<extend>>` link, another special link, shows a newly created optional use case from an existing use case. It covers alternative cases that may or may not necessarily take place. An `<<extend>>` link is also a dashed directed line with an arrow pointing toward the extended use case; these special links are labeled accordingly.

A complete use case diagram describes a set of scenarios; scenarios may have a set of subordinate, or lower-level, use cases. Use case diagrams are used in the early stages of the software development life cycle, such as analysis and design. These diagrams are one of the most frequently used UML diagrams for object-oriented system analysis and design.

System analysts employ use case diagrams to capture and verify user requirements. Architects and designers use these diagrams to derive structural diagrams (e.g., class diagrams) and behavioral diagrams (e.g., sequence diagrams and communication diagrams).

3.2.2.2 Activity Diagram

An activity diagram is used to describe complex business processes. This diagram typically involves complex workflow, decision making, concurrent executions, exception handling, process termination, etc. An activity diagram is a workflow-oriented diagram describing the steps in a single process. One activity diagram corresponds to one business process. There are many activities in a business process, and this diagram explores their dependency within the process.

UML activity diagrams use a rounded rectangle to represent an activity. Each activity diagram has a starting point and one or more finishing points. A small diamond represents a decision point in the diagram. Activity diagrams support parallel processing using a pair of black horizontal bars to indicate the corresponding fork/join actions in such pathways. UML activity diagrams also support communication between two concurrent flows by sending signals from one path to another. (This is called an event and is noted as a pair of convex polygons.)

An activity diagram gives a detailed dynamic view of a specific task or process within a system so that the software developer can easily recognize the implementation requirements. This diagram is a basis for a communication diagram and other dynamic interaction diagrams.

Figure 3.10 shows a partial set of activities in a purchase-order-processing system. The first black bar splits (forks) two concurrent activities that can be executed in parallel. The shipping/handling activity will not start until both of them complete and join together (indicated by the second bar at the bottom).

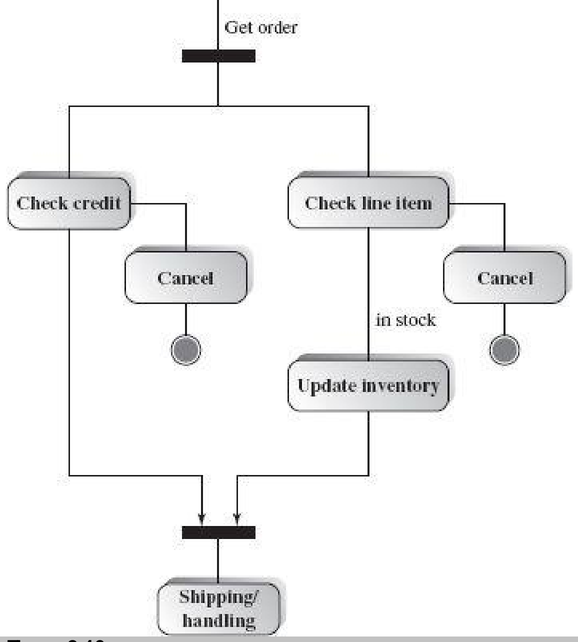


Figure 3.10
Activity diagram

3.2.2.3 State Machine Diagram

A state machine diagram, called a state chart in UML 1. x, is widely used for embedded systems and device software analysis and design. It is an event-oriented diagram in which the system's elements change their states in response to external or internal stimuli (such as time events or other system events). These diagrams are ideal for specifying the internal behavior of objects.

In a state machine diagram, a state is a rounded rectangle with three subdivisions: state name, state variables, and state activities. A state is a situation in which an object meets conditions, takes actions, and waits for a new event. When a new event takes place in the current state, the machine will perform specified actions and then will enter a new state (the next state).

A complex composite state may have a subordinate state diagram. The sub-states in a composite state may be transited from one to the next, sequentially or concurrently.

Each state machine diagram has one starting point in a solid black circle and has one or more endpoints, the latter indicated by eye-circles. The transition links between states are solid lines with arrowheads to indicate direction. State diagrams help software developers understand how a system responds and handles external events and the corresponding event-triggering condition.

Figure 3.11 shows a state machine diagram that depicts a login process. Initially, the state machine executes a busy loop to wait for user login, and then the username/password pair is verified. If the pair matches the system records, the login is confirmed; otherwise the login is rejected.

3.2.2.4 Interaction Overview Diagram

An interaction overview diagram describes the control flow of the interactions rather than the message. It is a variant of the activity diagram. The nodes in an interaction overview diagram represent either a reference to an existing diagram (ref), a basic interaction element [activity diagram(ad)], or a sequence diagram(sd)].

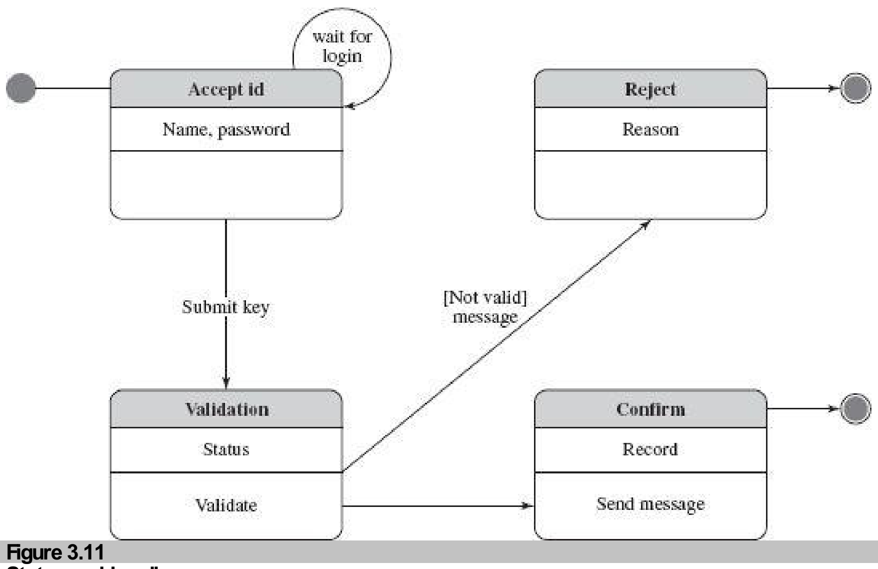


Figure 3.11
State machine diagram

Each node (or frame) can be an interaction diagram such as a sequence diagram, communication diagram, activity diagram, or nested interaction overview diagram. A reference node, indicated by "ref" in the upper left-hand corner of the frame, points to an existing diagram, while the basic element displays the frame's dynamic interaction diagram. A basic element is indicated by an "ad" label for an activity diagram, an "sd" label for a sequence diagram, or a "cd" label for a communication diagram, and so on. The interaction overview diagram is a high-level abstraction of an interaction overview description.

Figure 3.12 presents an example interaction overview diagram showing reference diagrams that point to other UML diagrams and one "ad" diagram displaying a detailed activity diagram.

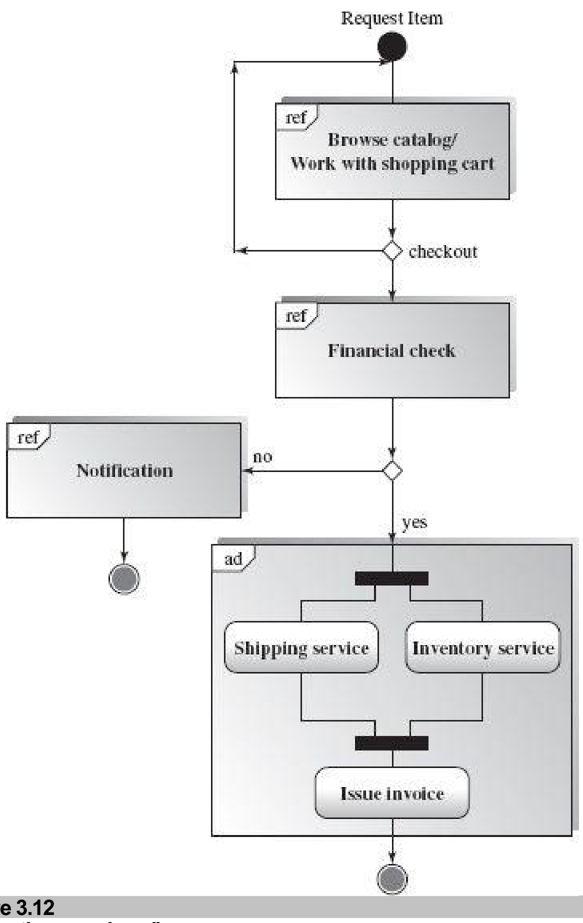


Figure 3.12
Interaction overview diagram

3.2.2.5 Sequence Diagram

A sequence diagram is one of the most important and widely used UML diagrams for software system analysis and design. It is a time-oriented interaction diagram showing the chronological sequence of messages between objects. Usually, one sequence diagram corresponds to one use case.

Each participant object in this diagram has a vertical timeline for its life cycle. Time goes forward along with the downward timeline. Each vertical timeline has a number of narrow rectangular boxes (called activations) representing the object-activated state in which it receives or sends out messages. Each activation box may also have a self-recursive directed link pointed back to itself, indicating that the object passes messages to itself. An activation may also branch or fork many separate lifelines for the if-selection scenario conditions; eventually all forked lines will join together.

Passing messages between objects is represented by a horizontal arrow link from the source to the destination. A simple transfer message line, represented by a solid line with an arrowhead, transfers control from one object to the other. An object can send a synchronous message to another object by a line with a full arrowhead. A synchronous message means that the sender must wait for a reply from the target object before it can move forward in the timeline. An object can also send an asynchronous message to another object, which is indicated by a line with a half arrowhead. The sender of an asynchronous message can continue its work down the timeline without waiting for a return message from the target object.

Figure 3.13 shows a simplified sequence diagram for online shopping. The sequence of message exchanges starts from the *cart* object. After a *browse* message, the *cart* object sends a message to *checkcart* to check out. *Checkcart* does a calculation (i.e., via self-message *calc)* and then sends a message to *inventory*. The message from *heckcart* to *inventory* is a synchronous one, because *checkcart* has to wait for the *response* message (represented using a dotted arrow line). Then *checkcart* contacts *shipping* and finally sends back a message to the actor that initiated the use case.

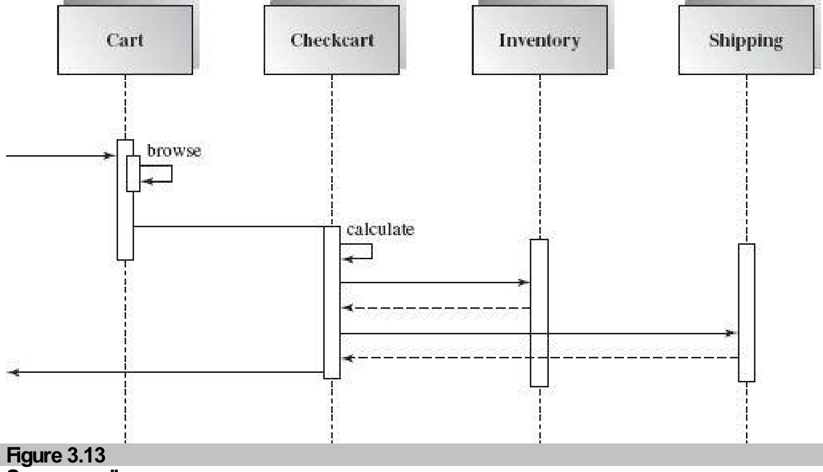


Figure 3.13
Sequence diagram

3.2.2.6 Communication or Collaboration Diagram

The UML communication diagram, known as the collaboration diagram in UML 1. x, is a message-oriented diagram that describes all message passing sequences, flow control, object coordination, etc., among the objects that participate in certain use cases. It summarizes how objects in the system receive and send messages. It is an extension of the static object diagram in which the links between objects represent association relationships. Above the links in a communication diagram are the numbered messages, indicating the order in which they are sent or received. The messages tell the receiver to perform an operation with specified arguments. Every communication diagram is equivalent to a sequence diagram, i.e., a communication diagram can be converted to an equivalent sequence diagram and vice versa. These two types of diagrams provide a message-oriented and time-oriented view, respectively.

Figure 3.14 shows an example of a communication diagram. It is equivalent to the sequence diagram shown in Figure 3.13 except that message names are given.

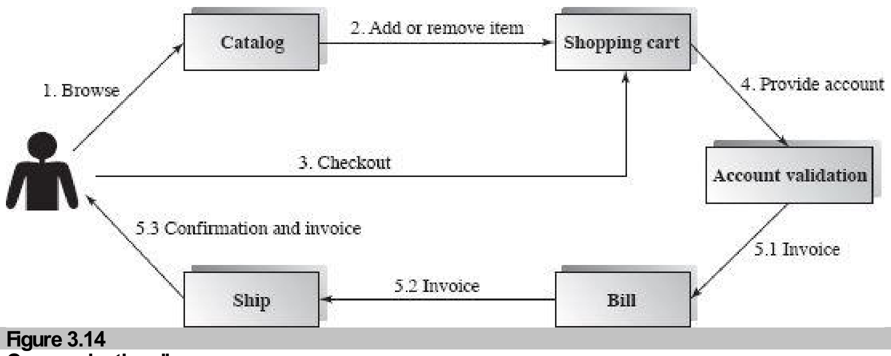


Figure 3.14
Communication diagram

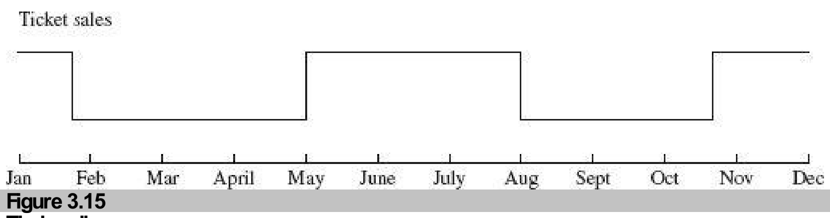


Figure 3.15
Timing diagram

3.2.2.7 Timing Diagram

The timing diagram is a new diagram in UML 2.0. It combines the state diagram and time sequences to show the dynamic view of state change caused by external events over time. It is often used in time-critical systems such as real-time operating systems, embedded system designs, etc. Figure 3.15 shows a timing diagram for seasonal discount air ticket prices.

3.3 Architecture View Models

A model is a complete, simplified description of a system from a particular perspective or viewpoint. There is no single view that can present all aspects of complex software to stakeholders. View models provide partial representations of the software architecture to specific stakeholders such as the system users, the analyst/designer, the developer/programmer, the system integrator, and the system engineer. Software designers can organize the description of their architecture decisions in different views. Stakeholders can use a view to find what they need in the software architecture.

The 4+1 view model was originally introduced by Philippe Kruchten (Kruchten, 1995). The model provides four essential views: the logical view, the process view, the physical view, and the development view. The logical view describes, for example, objects and their interactions; the process view describes system activities, their concurrency and synchronization; the physical view describes the mapping of the software onto the hardware, the server, and the network configuration; and the development view describes the software's static structure within a given development environment.

There is also another view called the scenario view; this view describes the scenarios that capture the most important aspects of the functional requirements, drive the system design, and validate the system. The 4+1 view model is a multiple-view model that addresses different aspects and concerns of the system. The 4+1 view model standardizes the software design documents and makes the design easy to understand by all stakeholders.

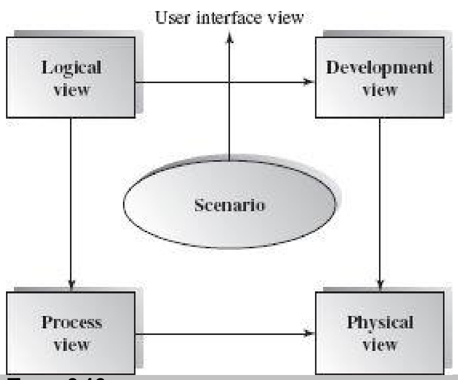


Figure 3.16
4+1 view model

We extend the 4+1 view model with one more view, the user interface (UI) view. The UI view, for end users of the software system, describes the graphical user interface to verify and validate the user interface requirements; these significantly impact the system usability and other quality attributes.

Figure 3.16 shows the 4+1 view model extended with our fifth view, the user interface view. The scenario view is coherent with the other four views, whereas the user interface view complies with the scenario view and is supported by the other four views.

3.3.1 The Scenario View

The scenario view describes the functionality of the system, i.e., how the user employs the system and how the system provides services to the users. This view provides a foundation for the other four views and lets them work together seamlessly and coherently. It helps designers to discover architecture elements during the design process and to validate the architecture design afterward. So, the scenario view helps to make the software architecture consistent with functional and nonfunctional requirements.

The UML use case diagram and other verbal documents are used to describe this view. The stakeholders of this view are end-users, architects, developers, and all users of the other four views. Figure 3.9 on page 49 describes a use case diagram for online shopping.

The scenario view is used to drive architecture design in the earlier phases of software development and is also used for software validation at later phases of the development cycle.

3.3.2 The Logical or Conceptual View

The logical view is based on application domain entities necessary to implement the functional requirements. It focuses on functional requirements, the main building blocks, and key abstractions of the system. The logical view is an abstraction of the system's functional requirements. It is typically used for object-oriented (OO) modeling from which the static and dynamic system structures emerge. The logical view specifies system decomposition into conceptual entities (such as objects) and connections between them (such as associations). This view helps to understand the interactions between entities in the problem space domain of the application and their potential variation. In an object-oriented system, the architecture elements may be classes and objects.

The logical view is typically supported by UML static diagrams including class/object diagrams and UML dynamic diagrams such as the interaction overall diagram, sequence diagram, communication diagram, state diagram, and activity diagram. The class diagram is used to describe the conceptual or logical view. A class diagram defines classes and their attributes, methods, and associations to other classes in the system. A class diagram is static in the sense that it does not describe any user interaction nor any sequence of module interaction in the system.

A block diagram can also be used to provide an overview of the whole system. A sequence diagram shows how objects in the system interact. A communication diagram shows system objects and the message passing between them in time order.

In summary, the logical view points out all major tasks the system must complete, and presents the major components and their static relationships. The stakeholders of the logical view are the end-users, analysts, and designers. We can apply an object-oriented design methodology in the logical view because the view itself is object-oriented.

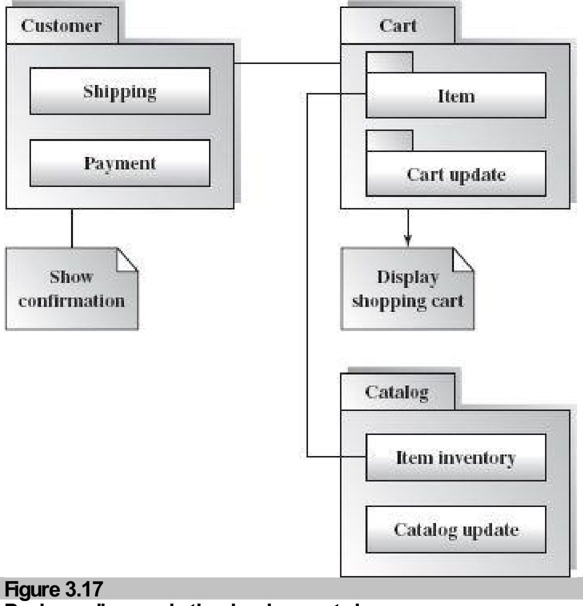


Figure 3.17
Package diagram in the development view

3.3.3 The Development or Module View

The development view derives from the logical view and describes the static organization of the system modules. Modules such as namespaces, class library, subsystem, or packages are building blocks that group classes for further development and implementation. This view addresses the subsystem decomposition and organizational issue. The software is packaged and partitioned into small units such as program libraries or subsystems created by many teams of developers. Each package has its own visibility and accessibility as package or default scope visibility (see static structure discussion on page 42).

The development view maps software component elements to actual physical directories and files. UML diagrams such as package diagrams and component diagrams are often used to support this view. The stakeholders of this view can be programmers and software project managers.

Figure 3.17 shows a simple development view using a package diagram.

3.3.4 The Process View

The process view focuses on the dynamic aspects of the system, i.e., its execution time behavior. This view also derives from the logical view. It is an abstraction of processes or threads dealing with process synchronization and concurrency. It contributes to many nonfunctional requirements and quality attributes such as scalability and performance requirements.

The process view looks at the system's processes and the communications among them. A software system can be decomposed into many runtime execution units. How to organize all execution units at runtime is presented in this view. The quality attributes such as performance, scalability, concurrency, synchronization, distribution, and system throughput are all addressed in the process view. This view maps functions, activities, and interactions onto runtime implementation with a focus on nonfunctional requirements as well as the implementation of the functional requirements.

The process view takes care of the concurrency and synchronization issues between subsystems. It can be described at several levels of abstraction, from independently executing logical networks of communicating programs to basic tasks running within the same processing node. The process view must also address nonfunctional requirements such as multithreading and synchronous/asynchronous communications for performance and availability. The UML activity diagram and interaction overview diagram support this view.

Figure 3.18 presents an activity diagram that documents the process view. Notice that after the *check credit* step, two processes are spawned to run concurrently.

The stakeholders of this view are the developers and integrators. Many architecture styles such as pipe and filter, multi-tier, and others can be applied in the process view.

3.3.5 The Physical View

The physical view describes installation, configuration, and deployment of the software application. It concerns itself with how to deliver the deploy-able system. The physical view shows the mapping of software onto hardware. It is particularly of interest in distributed or parallel systems. The components are hardware entities (processors), and the links are communication pathways; together these specify how the various elements such as communication protocols and middleware servers found in the logical, process, and development views are mapped onto the various nodes in the runtime environment.

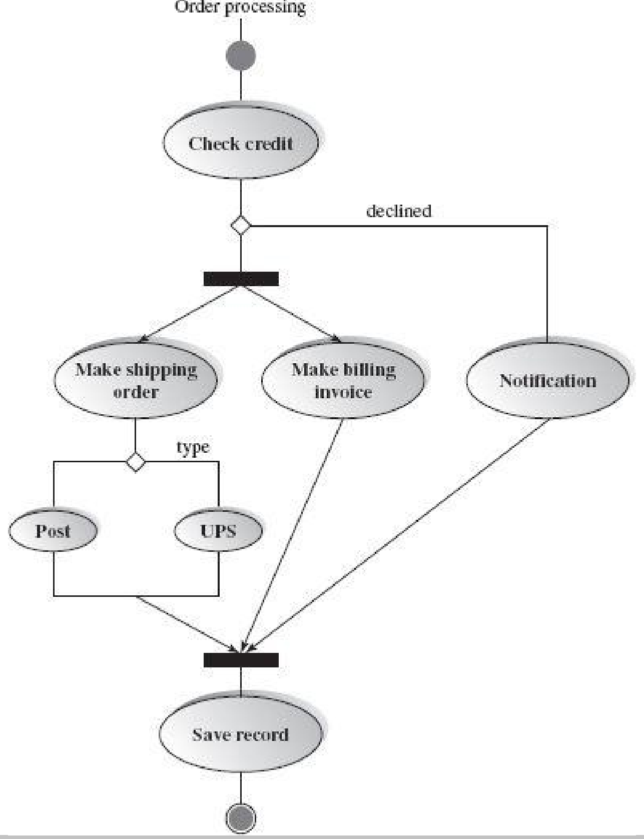


Figure 3.18
Activity diagram in the process view

A physical view also takes into account the system's nonfunctional requirements such as system availability, reliability (fault-tolerance), throughput performance, scalability performance, and security. For example, software can be delivered in different hardware and networking layouts, which will result in significant differences in these quality attributes.

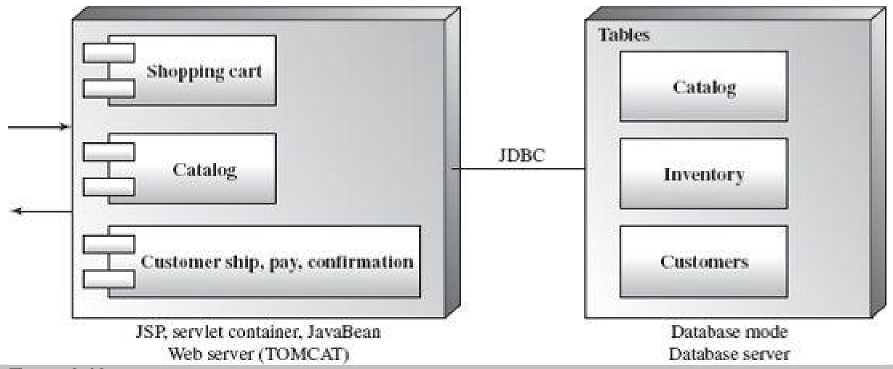


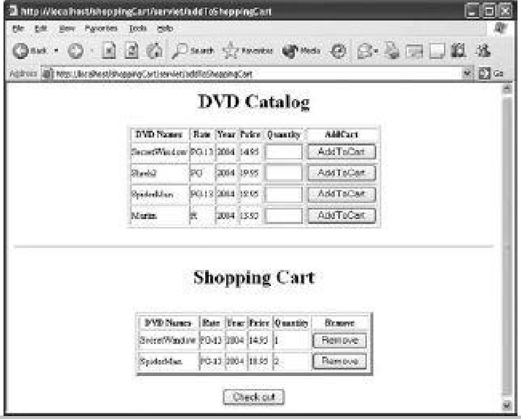
Figure 3.19
Deployment diagram in the physical view

The system topology in terms of hardware, network, and other infrastructure is all part of this view. This view explains the nonfunctional requirements and quality attributes such as performance, availability, reliability, and scalability. The physical view must address network connections and communication protocols such as server nodes and multi-tier distributed environment configurations. The UML deployment diagrams and other documentation are often used to support this view.

The stakeholders of this view are system installers, system administrators, system engineers, and operators. Figure 3.19 presents an example of a deployment diagram showing that the order processing system is deployed on two servers.

3.3.6 The User Interface View

The User Interface (UI) view is an extended view that provides a clear user-computer interface view and hides implementation details. This view may be provided as a series of screen snapshots or a dynamic, interactive prototype demo. Any modification on this view will have direct impact on the scenarios view. The screenshot in Figure 3.20 shows a GUI user interface for an online shopping cart.


http://localhost/shoppingCart/servlet/addToShoppingCart

DVD Catalog

DVD Names	Rate	Year	Price	Quantity	AddCart
SecretWindow	PG-13	2004	14.95		AddToCart
Shrek2	PG	2004	19.95		AddToCart
SpiderMan	PG-13	2004	12.95		AddToCart
Martin	R	2004	13.50		AddToCart

Shopping Cart

DVD Names	Rate	Year	Price	Quantity	Remove
SecretWindow	PG-13	2004	14.95	1	Remove
SpiderMan	PG-13	2004	18.95	2	Remove

Check out

Figure 3.20
Forms in the user interface view

In summary, the 4+1 view is an architecture verification technique for studying and documenting software architecture design. Each view provides a window into the different aspects of the system. The 4+1 view covers all aspects of a software architecture for all stakeholders. The views are interconnected; thus, based on the scenarios view, we can start with the logical view, move to the development or process view, and finally go to the physical view. The user interface view is also established during this process.

The 4+1 view architecture model is available in the Rational Rose IDE kit.

3.4 Architecture Description Language (ADL)

An ADL is a notation specification providing syntax and semantics for defining software architecture. It also provides designers with the ability to decompose components, combine components, and define interfaces of components. An ADL is a formal specification language with well-defined syntax and semantics used to describe architecture components and their connections, interfaces, and configurations.

Garlan and Shaw (1996) list the following requirements for an ADL:

- *Composition*: “It should be possible to describe a system as a composition of independent components and connections.” Large systems should be built from constituent elements, and it should be possible to consider each element independently of the system.
- *Abstraction*: “It should be possible to describe the components and their interactions in a way that describes their abstract roles in a system.” It should not be necessary to consider implementation issues while specifying the architecture.
- *Reuse*: Reusability should be built-in at the component and connection level. The derivation of architecture patterns should also be supported to facilitate the reuse of architecture descriptions.
- *Configuration*: Architecture descriptions should enable comprehension and modification of an architecture without examination of each component and connector.
- *Heterogeneity*: “It should be possible to combine multiple, heterogeneous architectural descriptions.”
- *Analysis*: The use of an ADL should facilitate the analysis of an architecture design. Analysis might include consideration of throughput, deadlock, input/output behavior, and scheduling.

A number of ADLs have been proposed over the last few years. These include UniCon and Wright, both from CMU; C2sadel from UC Irvine and USC; Rapide from Stanford; and Darwin from Imperial College, London. Acme is another ADL available in the research community. UML can also provide many of the artifacts needed for architecture descriptions, but is not a complete or sufficient ADL.

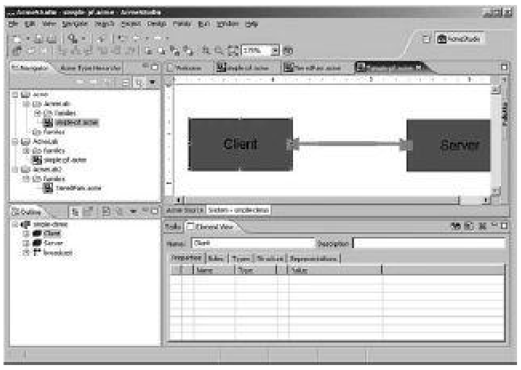
Let's take a close look at one of the ADLs listed earlier. Acme provides a formal and semiformal way to describe a software architecture as a static structure in a high-level of abstraction. This ADL provides many building block elements for architecture design description. Three of these basic elements are components, connectors, and systems; additional elements include ports, roles, representations, and rep-maps. Component elements can be any computing or data store units. Connectors represent interactions among components. The connector elements implement the communication and interactions among components in a system, such as synchronous or asynchronous communications.

Components and connectors in Acme play the same roles as boxes and lines in the block (box-and-line) diagram. However, they have a more specific purpose. For example, components have interfaces that are defined by a set of ports. Each port identifies a contact point between the component and its clients. A component may provide multiple interfaces by using different types of ports. A simple port may represent only a single procedure signature.

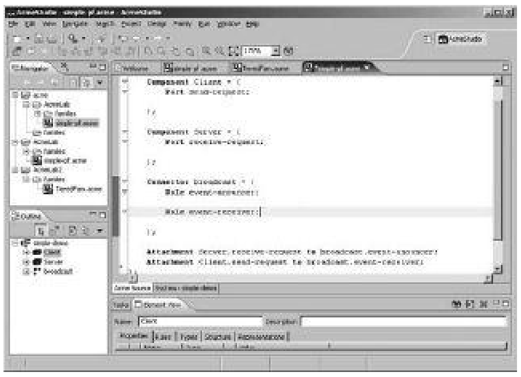
Connectors also have interfaces defined by a set of roles. Each role represents an interaction participant of the connector. A simple binary connector has two roles, such as the *caller* and *callee* roles of an RPC connector, the *reading* and *writing* roles of a pipe, or the *sender* and *receiver* roles of a message passing connector. Multiple role connectors may serve multiple roles; for example an event broadcast connector might have a single *event-publisher* role and multiple *event-receiver* roles.

Here, we present several examples showing how the Acme ADL and the AcmeStudio tool work together.

As shown in the following screenshot, a user can conveniently construct an architecture design using the AcmeStudio tool. The simple design consists of two components, a client and a server, connected by a connector. This architecture style is the client/server model. We can define the name and properties of each component and the connector in the diagram. The diagram can be translated by AcmeStudio to a specification in the Acme language. Any constraints preset by designers in the specification can be examined by AcmeStudio. The AcmeStudio also has plug-ins for many other checks, such as portability analysis. The Acme specification can also be used to construct a code framework in popular programming languages such as C++.



The next screenshot shows the architecture description (in Acme) generated automatically by AcmeStudio.

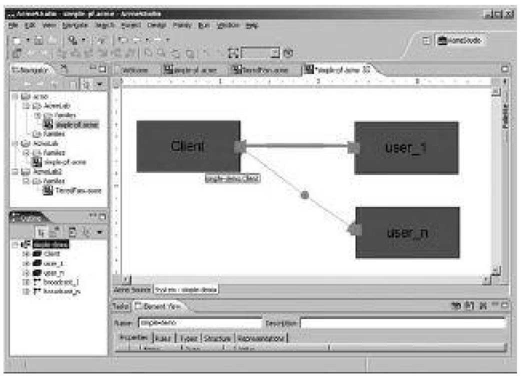


```
Component Client = {
    Port send-request;

};

Component Server = {
    Port receive-request;

};

Connector broadcast = {
    Role event-announcer;

    Role event-receiver;

};

Attachment Server.receive-request to broadcast.event-announcer;
Attachment Client.send-request to broadcast.event-receiver;
```

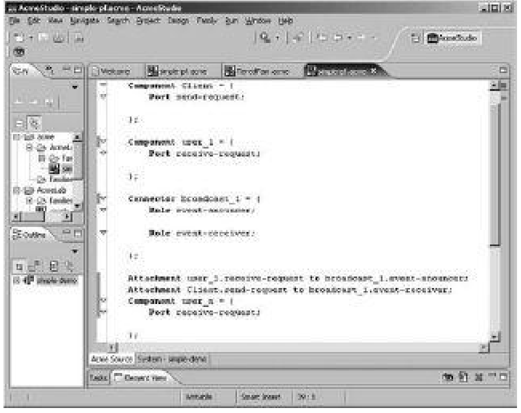
The architecture description generated by AcmeStudio may be translated into C++ or Java implementation as illustrated here.

```
System event-listener =
{
    Component Client = { Port send-request; };
    Component server = { Port receive-request; };
    Connector broadcast = {
      Roels {event-announcer, event-receiver}
    };
    Attachments {
      Server.receive-request to broadcast.event-anouncer;
      Client.send-request to broadcast.event-receiver;
    }
}
```

The next screenshot shows an example of multiple users with a single-server architecture design.



There are multiple event-receivers for a single event-announcer in this simple architecture. The following is the ADL architecture description generated by AcmeStudio.



```
Component Client = {
    Port send-request;

};

Component user_1 = {
    Port receive-request;

};

Connector broadcast_1 = {
    Role event-announcer;

    Role event-receiver;

};

Attachment user_1.receive-request to broadcast_1.event-announcer;
Attachment Client.send-request to broadcast_1.event-receiver;
Component user_n = {
    Port receive-request;

};
```

From the AcmeStudio demo we see that an ADL can be used to describe an architecture design using a description language with its own syntax and grammar. A software architecture specified by an ADL can be used to generate the implementation code, which makes the developer's job much easier.

3.5 Summary

Decomposition is the most important step in software design. The separation of software architecture into multiple views helps by reducing complexity; it also allows for the optimization of design guidelines such as low coupling and high cohesion within and between elements of a system. Multiple architecture views can also help with design trade-offs between software quality attributes.

An architecture design is based on the software requirement specification and generated in the design phase of the software engineering life cycle. The software architecture must satisfy the functional requirements as well as the nonfunctional requirements.

UML is a common modeling language for specifying complex software systems. It is widely accepted in the software industry for software analysis and design. This chapter discussed how to use UML to specify software architectures, to simplify the OO design, and to improve software design quality. UML notations are useful tools in describing software architectures. UML is also used in the 4+1 view model. This view model describes not only the functional requirements but deals with nonfunctional requirements as well. The 4+1 view model is a blueprint for the next phase of software development.

This model derives the software architecture's description from several viewpoints. Different views represent different aspects of the software; all views are based on use cases or the scenario view. The user interface view (added by the authors) is an end-user external view supported by all the other views, including the scenario view.

In this chapter, we also introduced Architecture Description Languages (ADLs); these are formal or semi-formal notations to describe software architecture.

In the next part of this book, we will discuss all the architecture styles in detail and you will find a case study chapter showing different styles applied to the same problem, which results in different quality attributes.

3.6 Self-Review Questions

1. Which of the following notations is used to support the logical view?
    a. Sequence diagram
    b. Collaboration diagram
    c. State diagram
    d. All of the above
2. Which of the following notations is used to support the physical view?
    a. Sequence diagram
    b. Collaboration diagram
    c. State diagram
    d. None of the above
3. Activity diagrams are used to support the process view.
    a. True
    b. False
4. Deployment diagrams are used to support the physical view.
    a. True
    b. False
5. Component diagrams are used to support the development view.
    a. True
    b. False
6. The software submodules and their interfaces are described in the logical view.
    a. True
    b. False
7. Concurrency control activity is part of the process view.
    a. True
    b. False
8. System and network configuration decisions are part of the physical view

   a. True
   b. False
9. Software architecture is concerned only with functional requirements.
   a. True
   b. False
10. Prototyping can be used to support UI design.
    a. True
    b. False
11. ADL is a programming language.
    a. True
    b. False
12. ADL can produce target code.
    a. True
    b. False
13. ADL is used only for software architecture specification.
    a. True
    b. False
14. UML diagrams are used for system analysis and design.
    a. True
    b. False
15. Use case diagrams are generated in the early stages of the SDLC, whereas deployment diagrams are generated in the later stages of the SDLC.
    a. True
    b. False
16. Composite structure diagrams are based on object diagrams
    a. True
    b. False
17. Component diagrams are based on object diagrams.
    a. True
    b. False
18. A UML diagram must provide a complete view of the entire software system.
    a. True
    b. False
19. A component is a class or an object.
    a. True
    b. False
20. Asynchronous message invocation can be expressed in sequence diagrams.
    a. True
    b. False
21. Conditional branching can be represented in sequence diagrams.
    a. True
    b. False
22. An activation in an object lifeline may have its own cycle message pointed back to itself in a sequence diagram.
    a. True
    b. False
23. An interaction overview diagram is based on all low-level interaction diagrams.
    a. True
    b. False

Answers to the Self-Review Questions

1. d 2. d 3. a 4. a 5. a 6. a 7. a 8. a 9. b 10. a 11. b 12. a 13. b 14. a 15. a 16. a 17. a 18. a 19. b 20. a 21. a 22. a 23. a

3.7 Exercises

1. List all interaction UML diagrams.
2. List all structural UML diagrams.
3. List all early phase SDLC UML diagrams.
4. List all late phase SDLC UML diagrams.
5. Describe the relationship between sequence diagrams, communication diagrams, and interaction diagrams.
6. Enumerate the problem domains suitable to state machine diagrams.
7. List problem domains suitable to time diagrams.
8. In what case is the activity diagram a good choice?
9. What is ADL?
10. What is the 4+1 view model?
11. Describe the logical view in the 4+1 view model.
12. Does the 4+1 model work only with object-oriented design methodology?
13. Can ADL support a non-object-oriented model system?
14. Which diagrams are the static UML diagrams?
15. Which diagrams are the dynamic UML diagrams?
16. Is the component diagram used only for component-based design?
17. Is a 4+1 view model changed once it is released?

18. Is an ADL specification changed once it is released?

3.8 Design Exercises

1. Draw a use case diagram for an ATM machine transaction application software.
2. Draw a class diagram and object diagram for an ATM machine transaction application software.
3. Draw a state machine diagram and a sequence diagram for an ATM transaction application software.
4. Draw a use case diagram for a student online registration software system.
5. Draw a class diagram and an object diagram for a student online registration software system.
6. Draw a sequence diagram and a communication diagram for a student online registration software system.
7. Draw package and deployment diagrams for a student online registration software system.
8. Draw a component diagram for a hotel reservation system. Assume there are four components: customer, reservation, hotel, and billing. The billing component is available for reuse.
9. Use the 4+1 view model to describe an online bookstore software architecture.
10. Use the 4+1 view model to describe an inventory control software architecture.
11. Make the inventory control system part of the online bookstore system.
12. Use ADL to describe an online camping registration system for a national park.

3.9 Challenge Exercises

1. Use UML to model the software architecture of an online trusted payment system. The system users are buyer, seller, payer, and security trustee. There are many e-payment selections available. The buyers may be consumers, corporations, or organizations. The sellers may be merchants, service providers, and others. The payers may be banks, financial services, credit card companies, etc. The trustee may be a security service, transaction auditing company, etc.
   The system supports payment selection, security services, transaction protection, and process flow management. The system administrator is also a special system user.
2. Use UML to model the nationwide chain motel online reservation system. The system provides the reservation request service for room selection, date selection, reservation deposit handling, reservation cancellation, and reservation confirmation. The system also provides online information about location, direction, facility, motel photos, etc.

References

Garlan, David and Mary Shaw. *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ: Prentice Hall, 1996, 39-40.

Kruchten, Philippe. *“Architectural Bluprint—The '4+1' View Model of Software Architecture.”* IEEE Software, vol. 12, no. 6. (1995): 42-50.

Suggested Reading

AcmeWEB. “The Acme Project.” ABLE Project, Carnegie Mellon University, 2006, http://www.cs.cmu.edu/~acme/ (accessed in 2007).

The Object Management Group. “Unified Modeling Language.” *Catalog of OMG Modeling and Metadata Specifications,* http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML (accessed in 2007).

The Object Management Group. “UML 2.0 Specification.” Unified Modeling Language (UML), 2004, http://www.omg.org/technology/documents/formal/uml.htm.

Sparx Systems Pty Ltd. “UML Tutorial.” 2005, http://www.sparxsystems.com.au/UML_Tutorial.htm.