



# J2EE

SUN® CERTIFIED ENTERPRISE ARCHITECT

## Object-Oriented Analysis and Design

### CERTIFICATION OBJECTIVES

- ✓ Two-Minute Drill
- Q&A Self Test

One of the fundamental challenges facing software architects is change. The need to develop maintainable software systems has driven interest in approaches to software development and design. Object-oriented technology has proven over time to be one of the most promising paradigms for design and implementation of large scale systems. The distinctive features of object-oriented technology include classes and instances and abstraction. We assume that most readers have some grounding in these concepts so we will concentrate on the information that will prove valuable for the certification exam.

## Analysis and Design of Object-Oriented Architecture

*Modeling* is a visual process used for creating in a preserved form the design and structure of an application. Before, during, and after development, it is typical and prudent to outline an application, depicting dependencies and relationships among the components and subsystems. Like any good development tool, today's modeling tools facilitate this process by tracking changes made in the model to reflect the cascading effects of changes. Use of modeling tools gives developers a high-level and accurate view of the system.

Modeling can be used at any point in a project. Most modeling tools can reengineer and use code as input to create a visual model. The standard for modeling tools is the Unified Modeling Language (UML). This standard unifies the many proprietary and incompatible modeling languages to create one modeling specification. Use of modeling tools for development projects is increasing. With the increasing complexity of enterprise Java applications and components, modeling is a virtual necessity. It can reduce development time while ensuring that code is well formed.

Modeling is useful whether the objective is to understand and modify an existing computer-based business system or to create an entirely new one. An obstacle to engineering successfully is the inability to analyze and communicate the numerous interactive activities that make up a business process. Conversational languages, such as English, are ambiguous and therefore ineffective for communicating such objectives and activities. Formal languages are unintelligible to most functional (business) experts. What is needed instead is a technique that structures conversational language to eliminate ambiguity, facilitating effective communication and understanding.

In a process model, extraneous detail is eliminated, thus reducing the apparent complexity of the system under study. The remaining detail is structured to eliminate any ambiguity, while highlighting important information. Graphics (pictures, lines, arrows, and other graphic standards) are used to provide much of the structure,

so most people consider process models to be pictorial representations. However, well-written definitions of the objects, as well as supporting text, are also critical to a successful model.

In engineering disciplines, the model is typically constructed before an actual working system is built. In most cases, modeling the target business process is a necessary first step in developing an application. The model becomes the road map that will establish the route to the final destination. Deciding the functionality of the target destination is essential. To be effective, it must be captured and depicted in detail.

In today's software development environment, we speak of *objects* as things that encapsulate *attributes* and *operations*. Before we proceed to the modeling standards being used today by software architects, let's begin with some basic definitions of object programming and its intending analysis, design, and lifecycle.

## Key Features of OOP: Objects and Classes

Object-oriented programming (OOP) is the methodology used for programming classes based on defined and cooperating objects. OOP is based on objects rather than procedural actions, data rather than logic. In days past, a program had been viewed as a logical procedure that used input data to process and produce output data. Object-oriented programming focuses on the objects we want to manipulate rather than the logic required to manipulate them. Object examples range from human beings (described by name, address, and so forth) to inanimate objects whose properties can be described and managed, such as the controls on your computer desktop—buttons, scroll bars, and so on.

Step one in OOP is to identify the objects to be manipulated and their relationships to each other—that is, *modeling*. Once you've identified an object, you generalize it as a class of objects and define the kind of data it contains and logic that can manipulate it. The logic is known as *methods*. A real instance of a class is called an *object* or an *instance of a class*. The object or class instance is executed on the computer. Its methods provide computer instructions, and the class object characteristics provide relevant data. You communicate with objects and they communicate with each other with defined interfaces called *messages*.

The concepts and rules used in OOP provide these important benefits:

- The concept of a data class makes it possible to define subclasses of data objects that share some or all of the main class characteristics. This is known as *inheritance*, and it is a property of OOP that facilitates thorough data analysis, reduces development time, and ensures more accurate coding.

- Since a class defines only the data it needs, when an instance of that class is run, the code will not be able to access other program data improperly. This characteristic of data hiding provides greater system security and avoids unintended data corruption.
- The definition of a class is reusable not only by the program for which it is initially created but also by other object-oriented programs. This facilitates distribution for use in other domains.
- The concept of data classes allows a programmer to create new datatypes that are not defined in the language itself.

### Defining Object-Oriented Analysis and Design

In terms of computing software, *analysis* is the development activity consisting of the discovery, modeling, specification, and evaluation of requirements. Object-Oriented analysis (OOA) is the discovery, analysis, and specification of requirements in terms of objects with identities that encapsulate properties and operations, message passing, classes, inheritance, polymorphism, and dynamic binding. Object-oriented design (OOD) is the design of an application in terms of objects, classes, clusters, frameworks, and their interactions.

In comparing the definition of traditional analysis with that of object-oriented analysis and design (OOAD), the only aspect that is new is thinking of the world or the problem in terms of *objects* and *object classes*. A *class* is any uniquely identified abstraction—that is, a model—of a set of logically related instances that share the same or similar characteristics. An *object* is any abstraction that models a single element, and the term *object* as mentioned is synonymous with *instance*. Classes have attributes and *methods*, as they are more commonly known.

### Project Lifecycle

The project lifecycle is a pivotal concept in terms of understanding what a project is; the lifecycle is a mapping of the progress of the project from start to finish. Projects, by definition, have a start and finish, like any good game. At the simplest level, projects have two phases: planning and executing. Planning and executing are okay for a simple, short-term project. Larger, long-term endeavors require another layer to be added to the lifecycle of the projects. This can be achieved by subdividing each phase: plan and execute into two further phases, leading to a lifecycle of analysis, design, development, and implementation.

Table 3-1 summarizes the classic project lifecycle phases and mentions activities to be planned and executed for each phase. UML deliverables mentioned in this table are discussed in the sections that follow.

For the sake of completeness, we should also mention the Unified Process—or RUP (Rational Unified Process), as it has been trademarked by Rational ([www.rational.com](http://www.rational.com)). The RUP is an incremental process used by development managers to manage a

**TABLE 3-1** Project Lifecycle Phases

Primary Phase	Subphase	Activities
Analysis	Requirements analysis	Take a concept statement and define detailed requirements and the externally visible characteristics of the system. Write a validation plan that maps to the requirements specification. Short form: Is it possible to resolve the requirements?
	System-context analysis	Define the context of the system via use cases and scenarios. External messages, events, and actions are defined. The system is treated as a black box. Use case and scenario models are the deliverables. For real-time systems, characterize the sequence and synchronization details of the messages/responses. Short form: What would the big picture solution look like?
	Model analysis	Identify the classes, objects, and associations that solve the problem, using class and object diagrams. Response behavior is modeled using state charts. Interaction among objects is shown with sequence or collaboration diagrams. Short form: A further refinement of the big picture solution arrived at by decomposing subsystems into high-level classes.
Design	Architectural design	Define the important architectural decisions. Physical architecture of the system is modeled using deployment diagrams, software component architecture is modeled using component diagrams, and concurrency models are captured using class diagrams identifying the active objects. Design patterns are used here as well. Note: One key element of design is that “hard” dependencies on specific hardware, software, and other infrastructure is fleshed out as we move closer to implementation. For example, an architect may decide to use BEA WebLogic as the J2EE server. A designer may find that, while trying to build some XML parsing components, a decision needs to be made about whether to use BEA-specific APIs or perhaps use JAXP APIs.

TABLE 3-1 Project Lifecycle Phases (continued)

Primary Phase	Subphase	Activities
	Mechanistic design	Define the collaborative behavior of classes and objects. This information is captured on class and object diagrams. Sequence and collaboration diagrams capture specific instances of collaborations and state charts are enhanced to define the full behavior.
	Detailed Design	Define the detailed behavior and structure of individual classes using activity diagrams and notations.
Development		Develop class code, database definition, and message structures in the target language, DBMS, and messaging system.
Implementation	Unit testing	Test the internal structure and behavior of each class.
	Integration testing	Test the integration of various components. This take place recursively at multiple levels of decomposition based on the scale of the system.
	Validation testing	Test the delivered system against the requirements as defined in the validation test plan.
	System delivered	Pass the delivered system and user guide and other operational documentation to the user and technical support staff.

software project. Using the RUP, the project is broken down into phases and iterations. The iterations are oriented toward decreasing risk. Each phase should deliver a product, usually software, that can be demonstrated and validated against the project’s requirements and use cases. The development manager uses iteration plans to manage the project. An iteration plan provides a detailed description of the upcoming phase of work. It defines the roles involved as well as activities and artifacts to be delivered in that iteration. The RUP outlines a set of criteria by which productivity and progress can be measured during the iteration. As with all planning tools, it defines specific start and end dates for product delivery.

The RUP identifies four phases for projects. Each phase focuses the team on an aspect of the project and has associated milestones.

- 1. **Inception** The focus of this phase is the project scope.
- 2. **Elaboration** The architecture as well as the requirements of the product being built must be defined by the end of this phase.
- 3. **Construction** The software must be developed or constructed in this phase.
- 4. **Transition** The software must be rolled out to users during this phase.

The RUP phases in some respects parallel the classic lifecycle phases—analysis, design, development, and implementation. They are, however, targeted at managing risks in project development. They consider that today’s development is iterative. They are a framework geared for project leaders as opposed to architects and developers. The RUP management discipline provides a process that software development managers use to produce an overall project plan. The plan must be focused on deliverables, and it must be measurable, flexible, and aligned to real progress. The plan also must define the responsibilities and dependencies of the development team.

## Unified Modeling Language

The Unified Modeling Language (UML) is a language used for specifying, constructing, visualizing, and documenting the components of a software system. The UML combines the concepts of Booch, Object Modeling Technique (“OMT”), and Object-Oriented Software Engineering (“OOSE”). The result is a standard modeling language. The UML authors targeted the modeling of concurrent and distributed systems; therefore, UML contains the elements required to address these domains. UML concentrates on a common model that brings together the syntax and semantics using a common notation.

This nonexhaustive treatment of UML is arranged in parts. First we describe the basic elements used in UML. Then we discuss relationships among elements. The follow-up is the resultant UML diagrams. Within each UML diagram type, the model elements that are found on that diagram are listed. It is important to note that most model elements are usable in more than one diagram.



***UML is an evolving language. This chapter was written when OMG UML version 1.4 was current and 2.0 was in draft specification.***

## Elements Used in UML

In UML, an *element* is an atomic constituent of a model. A *model* element is an element that represents an abstraction drawn from the system being modeled. Elements are used in UML diagrams, which will be covered in the following sections. UML defines the following elements:

### Class

As mentioned, a class is any uniquely identified abstraction that models a single thing, and the term *object* is synonymous with *instance*. Classes have attributes and



methods. The class is represented in UML by a rectangle with three parts. The name part is required and contains the class name and other documentation-related information. For example, the name could be *data\_access\_object* <<java bean>>. The attributes part is optional and contains characteristics of the class. The operations part is also optional and contains method definitions. For example: method *(argument(s))* return type: *get\_order ( order\_id )* hashmap.

## Interface

An *interface* is a collection of operations that represent a class or that specify a set of methods that must be implemented by the derived class. An interface typically contains nothing but virtual methods and their signatures. Java supports interfaces directly. The interface is represented in UML by a rectangle with three parts. The name part, which is required, contains the class name and other documentation-related information. For example, the name could be *data\_access\_object* <<java bean>>. The attributes part (optional) contains characteristics of the class. The operations part (optional) contains method definitions. For example: method *(argument(s))* return type: *get\_order ( order\_id )* hashmap.

## Package

A *package* is used to organize groups of like elements. The package is the only group type element, and its function is to represent a collection of functionally similar classes. Packages can nest. Outer packages are sometimes called *domains*. Some outer packages are depicted by an “upside-down tuning fork” symbol, denoting them as *subsystems*. The package name is part of the class name—for example, given the class *accessdata* in the *ucny.trading.com* package, the fully qualified class name is *ucny.trading.com.accessdata*.

## Collaboration

*Collaboration* defines the interaction of one or more roles along with their contents, associations, relationships, and classes. To use collaboration, the roles must be bound to a class that supports the operations required of the role. A use of collaboration is shown as a dashed ellipse containing the name of the collaboration. A dashed line is drawn from the collaboration symbol to each of the objects, depending on whether it appears within an object diagram that participates in the collaboration. Each line is labeled by the role of the participant.

## Use Case

A *use case* is a description that represents a complete unit of functionality provided by something as large as a system or as small as a class. The result of this functionality



is manifested by a sequence of messages exchanged among the system (or class) and one or more outside actors combined with actions performed by another system (or class).

There are two types of use cases: *essential* and *real*. Essential use cases are expressed in an ideal form that remains free of technology and implementation detail. The design decisions are abstracted, especially those related to the user interface. A real use case describes the process in terms of its real design and implementation. Essential use cases are important early in the project. Their purpose is to illustrate and document the business process. Real use cases become important after implementation, as they document how the user interface supports the business processes documented in the essential use case. In either type, a use case is represented as a solid line ellipse containing the name of the use case. A stereotype keyword may be placed above the name, and a list of properties is included below the name.

## Component

The *component* represents a modular and deployable system part. It encapsulates an implementation and exposes a set of interfaces. The interfaces represent services provided by elements that reside on the component. A component is typically deployed on a node. A component is shown as a rectangle with two smaller rectangles extending from its left side. A component type has a type name: *component-type*. A component instance has a name and a type. The name of the component and its type may be shown as an underlined string, either within the component symbol or above or below it, with the syntax *component-name* ‘:’ *component-type*. Either or both elements are optional.

## Node

The *node* is a physical element object that represents a processing resource, generally having memory and processing capability—such as a server. Obviously, nodes include computers and other devices, but they can also be human resources or any processing resources. Nodes may be represented as types and instances. Runtime computational instances, both objects and component instances, may reside on node instances.

A node is typically depicted as a cube. A node type has a type name: *node-type*. A node instance has a name and a type name. The node may have an underlined name within the cube or below it. The name string has the syntax *name* ‘:’ *node-type*. The name is the name of the individual node, and the node-type says what kind of a node it is.

## State

The *state* is a condition that can occur during the life of an object. It can also be an interaction that satisfies some condition, performs some action, or waits for

some event. A *composite* state has a graphical decomposition. An object remains in a particular state for an interval of time. A state may be used to model the status of in-flight activity. Such an activity can be depicted as a state machine. A state is graphically shown as a rectangle with rounded corners. Optionally, it may have an attached name tab. The name tab is a rectangle and it contains the name of that state.

Relationships Used in UML

The object is the center of an object-oriented (OO) system. The OO model defines the system structure by describing objects (such as classes) and the relationships that exist among them. Class diagrams, as you will see, comprise classes, objects, and their relationships. The classes appear as rectangles that contain the class name. This rectangle is divided into sections, with the class name appearing in the first section, class attributes in the second section, class operations in the third, class exceptions in the fourth, and so on. The object names are underlined and have a colon as a suffix. As in any system, objects are connected by relationships.

UML defines and includes the types of relationships detailed in Table 3-2.

TABLE 3-2 UML Relationships

Relationship	Description	Notation
Generalization (aka Inheritance)	A specialized version of another class	Solid line with a closed arrowhead pointing to the more general class
Association	Uses the services of another class	Solid line connecting the associated classes, with an optional open arrowhead showing direction of navigation
Aggregation	A class “owns” another class	A form of association with an unfilled diamond at the “owner” end of the association
Composition	A class is composed of another class; refers to an aggregation within which the component parts and the larger encompassing whole share a lifetime	A form of aggregation, shown with either a filled diamond at the “composite” end, or with the composite graphically containing the “component”
Refinement	A refined version of another class; refinement within a given model can be shown as a dependency with the stereotype <<refines>> or one of its more specific forms, such as <<implements>>	Dashed line with a closed hollow arrowhead pointing to the more refined class
Dependency	A class dependent on another class.	Dashed line with an open arrowhead pointing to the dependency.

## Diagrams Used in UML

The following sections detail the graphical diagrams defined within UML.

### Use Case Diagram

The use case diagram shows actors, a set of use cases enclosed by a system boundary, communication or participation associations among the actors and the use cases, and generalizations among the use cases. See Figure 3-1.

### Class Diagram

The class diagram shows modeling elements. It may also contain types, packages, relationships, and even instances such as objects and links. A class is the descriptor for a set of objects that have a similar structure, behavior, and relationships. UML provides notation for declaring, specifying, and using classes. Some modeling elements that are similar to classes (such as types, signals, or utilities) are notated as stereotypes of classes. Classes are declared in class diagrams and used in most of the other diagrams. See Figure 3-2.

### Package Diagram

The package diagram is a mechanism used for dividing and grouping model elements such as classes. In UML, a folder represents a package. The package provides a name space so that two elements with the same name can exist by placing them in two separate packages. Packages can also be nested within other packages. Dependencies between two packages indicate dependencies between any two classes in the packages. See Figure 3-3.

**FIGURE 3-1**

Use case diagram

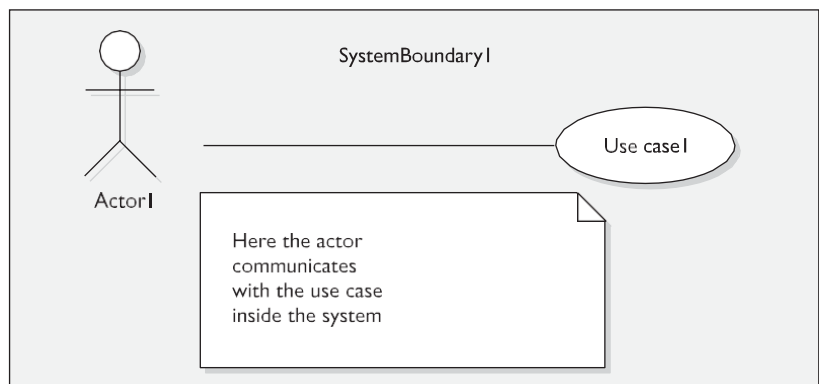


FIGURE 3-2 Class diagram

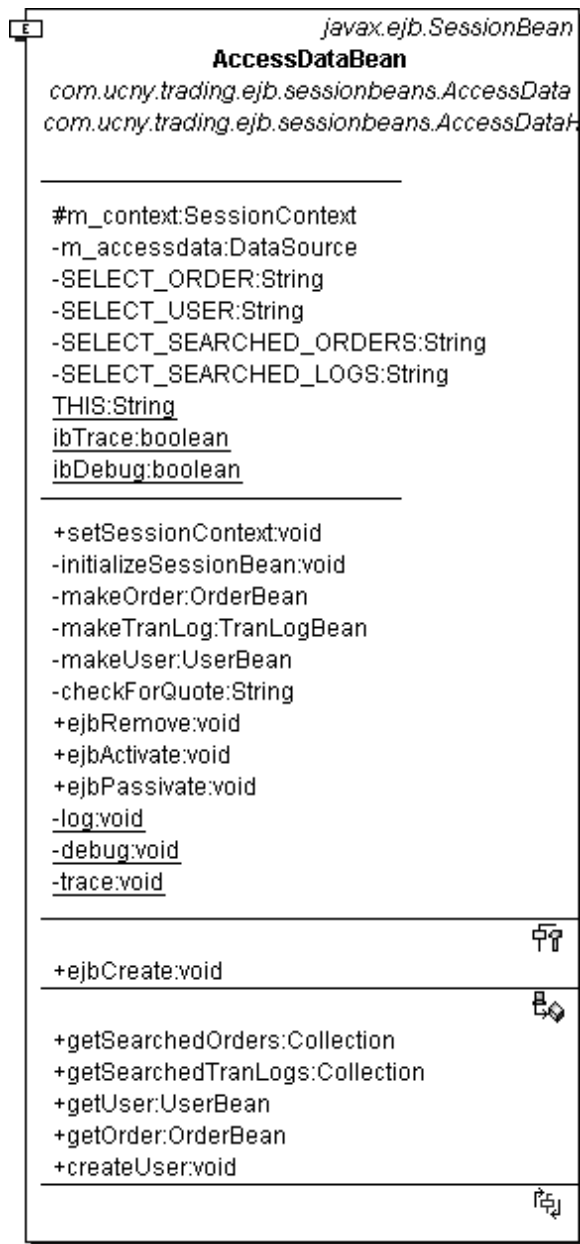
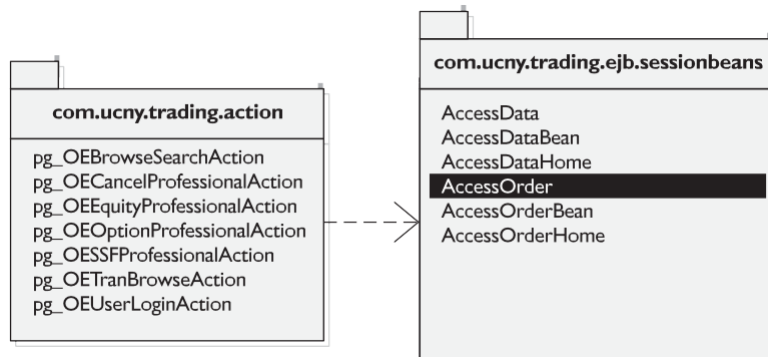


FIGURE 3-3

Package diagram



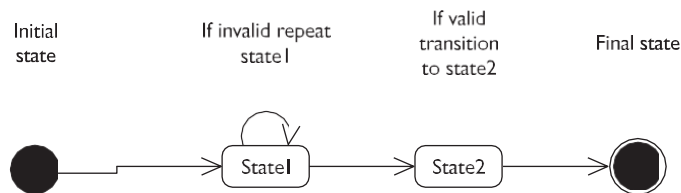
## State Diagram

The state diagram is a two-part diagram showing states and transitions. It shows states connected by physical containment and tiling. The entire state diagram is attached through the model to a class or a method—that is, an operation implementation. See Figure 3-4.

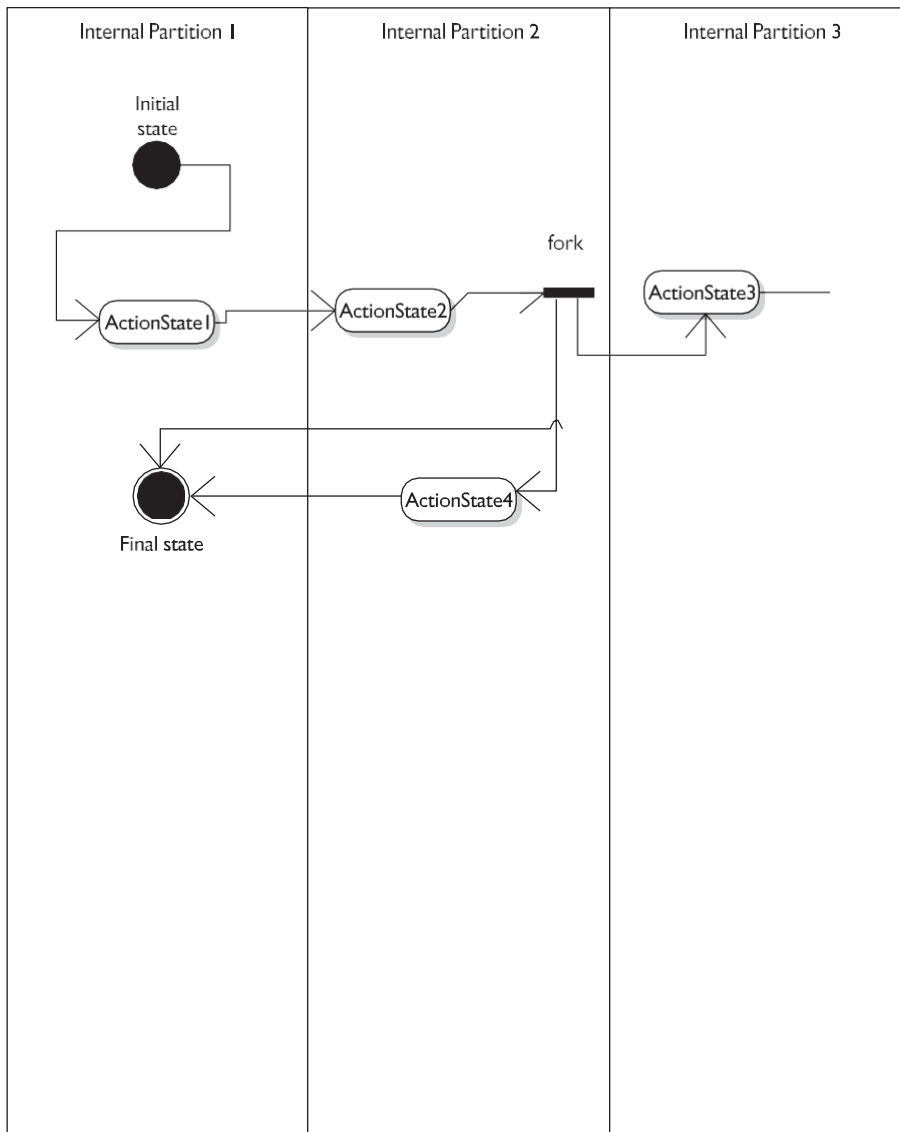
## Activity Diagram

An activity diagram is a special case of a state diagram in which all or most of the states are action states and in which all or most of the transitions are triggered by completion of the actions in the source states. The entire activity diagram is attached via the model to a class or to the implementation of an operation or a use case. This diagram concentrates on activity driven by internal processing as opposed to external forces. Activity diagrams are used for situations in which all or most of the events represent the completion of internal actions. Alternatively, ordinary state diagrams are used for situations in which *asynchronous* events occur. See Figure 3-5.

FIGURE 3-4 State diagram



**FIGURE 3-5** Activity diagram



## Sequence Diagram

A sequence diagram describes how groups of objects collaborate in some behavior over time. It records the behavior of a single use case. It displays objects and the

messages passed among these objects in the use case. A design can have lots of methods in different classes. This makes it difficult to determine the overall sequence of behavior. This diagram is simple and logical, so as to make the sequence and flow of control obvious. See Figure 3-6.

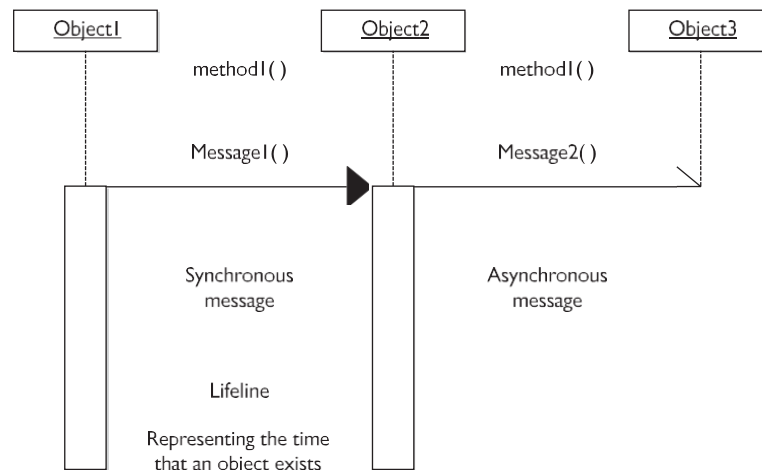
## Collaboration Diagram

A collaboration diagram models interactions among objects; objects interact by invoking messages on each other. A collaboration diagram groups together the interactions among different objects. The interactions are listed as numbered interactions that help to trace the sequence of the interactions. The collaboration diagram helps to identify all the possible interactions that each object has with other objects. See Figure 3-7.

## Component Diagram

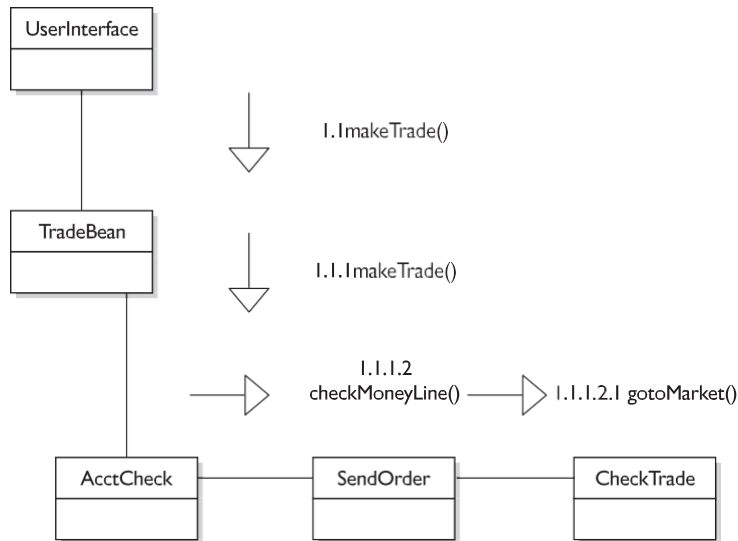
The component diagram represents the high-level parts that make up the modeled application. This diagram is a high-level depiction of the components and their relationships. A component diagram depicts the components refined post development or construction phase. See Figure 3-8.

**FIGURE 3-6** Sequence diagram





**FIGURE 3-7** Collaboration diagram



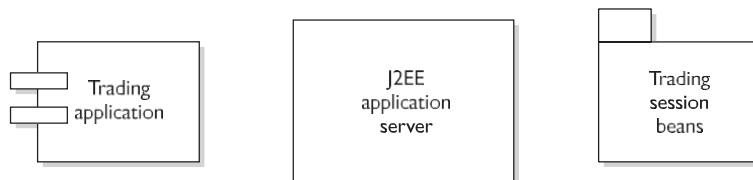
## Deployment Diagram

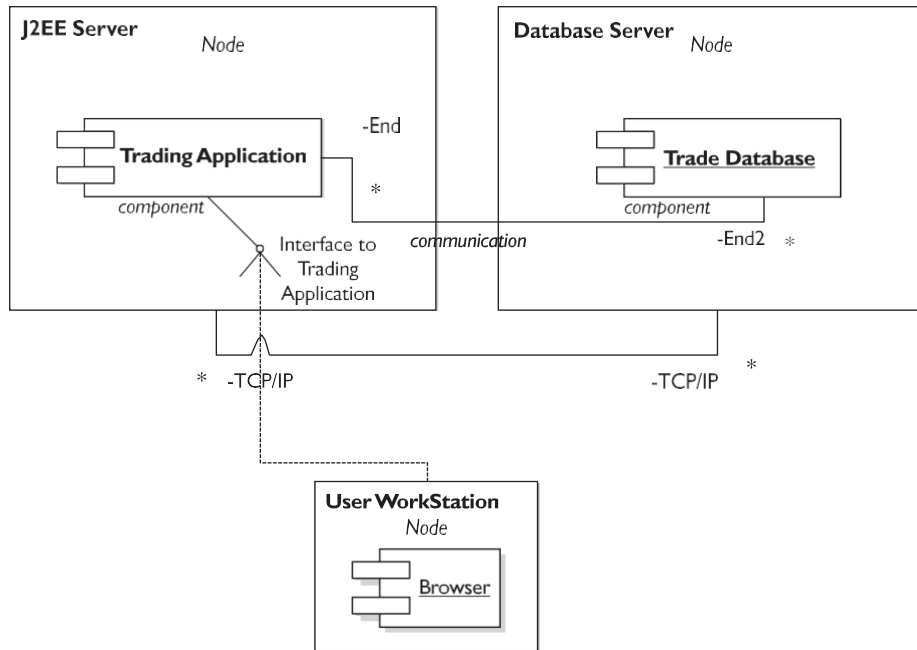
A deployment diagram captures the configuration of the runtime elements of the application. This diagram is obviously most useful when an application is complete and ready to be deployed. See Figure 3-9.

## Stereotypes

A *stereotype* is a new class of modeling element that is introduced during modeling time. Certain restrictions are in place: stereotypes must be based on certain existing classes in the meta model, and they may extend those classes only in certain predefined ways. They provide an extensibility mechanism for UML.

**FIGURE 3-8** Component diagram



**FIGURE 3-9** Deployment diagram

## Practical Use of UML Diagrams

The scope of a typical software system is one of the barriers preventing the thorough understanding necessary for effective maintenance of systems. Even standard visualization approaches such as graphs and flow charts are overwhelming when attempting to depict a system. As you start to analyze such a system, you often want to begin with a high-level understanding of the overall structure and design of the system. You then delve into lower-level details once you have bounded the problem at hand. And at other times, the scope of the problem requires that you continue to work from the higher-level view.

UML provides a number of abstraction mechanisms to help you study the high-level architecture of your software. Within the Unified Modeling Language notation, diagrams are the primary representation of a system. UML will help you understand the objects, interactions, and relationships of your system software and hardware.

## Use Case Diagram

The use case lends itself to a problem-centric approach to analysis and design, providing an understanding and a model of your system from a high-level business perspective—that is, how a system or business works or how you wish it to work. The use case diagram represents the functionality of a system as displayed to external interactions as *actors* of the system. A use case view represents the interface or interfaces that a system makes visible to the outside world, the external entities that interact with it, and their interrelationships.

Each use case step is either automated or manual. The objective of each step is to make a business decision or carry out a action. We typically assign responsibility for each business decision and action either to the system in the case of an automated action or to the actor in the manual case. This responsibility impacts the system delivered because the automated steps manifest themselves as system operations to make these decisions or execute these actions.

The diagram represents the processes within the system, which are visible to the outside world—that is, the actors of the system being modeled and the relationships among them.

Use cases are the functions or services of the system—those that are visible to its actors. They constitute a complete unit of functionality provided by a system as manifested by sequences of messages exchanged among the system and one or more actors together with actions performed by the system.

Actors are representative of the role of an object outside of a system that interacts directly with it as part of a complete work unit. An actor element characterizes the role played by an outside object, where one physical object may play multiple positions. For example, one entity may actually play different positions and assume different identities.

You can think of use case as a model that describes the processes of a business—order processing, for example—and its interactions with external parties such as clients and vendors. It is helpful in identifying the fundamental components of a system, namely the following:

- The business processes of the system
- External entities of the system
- The relationships among them

Use case diagrams are closely connected to *scenarios*. A scenario is an example of what happens when someone interacts with the system. For example, here is a scenario for a security trade: a trader accesses an Internet-based system and chooses the type of security he or she wants to trade. See Figure 3-10.

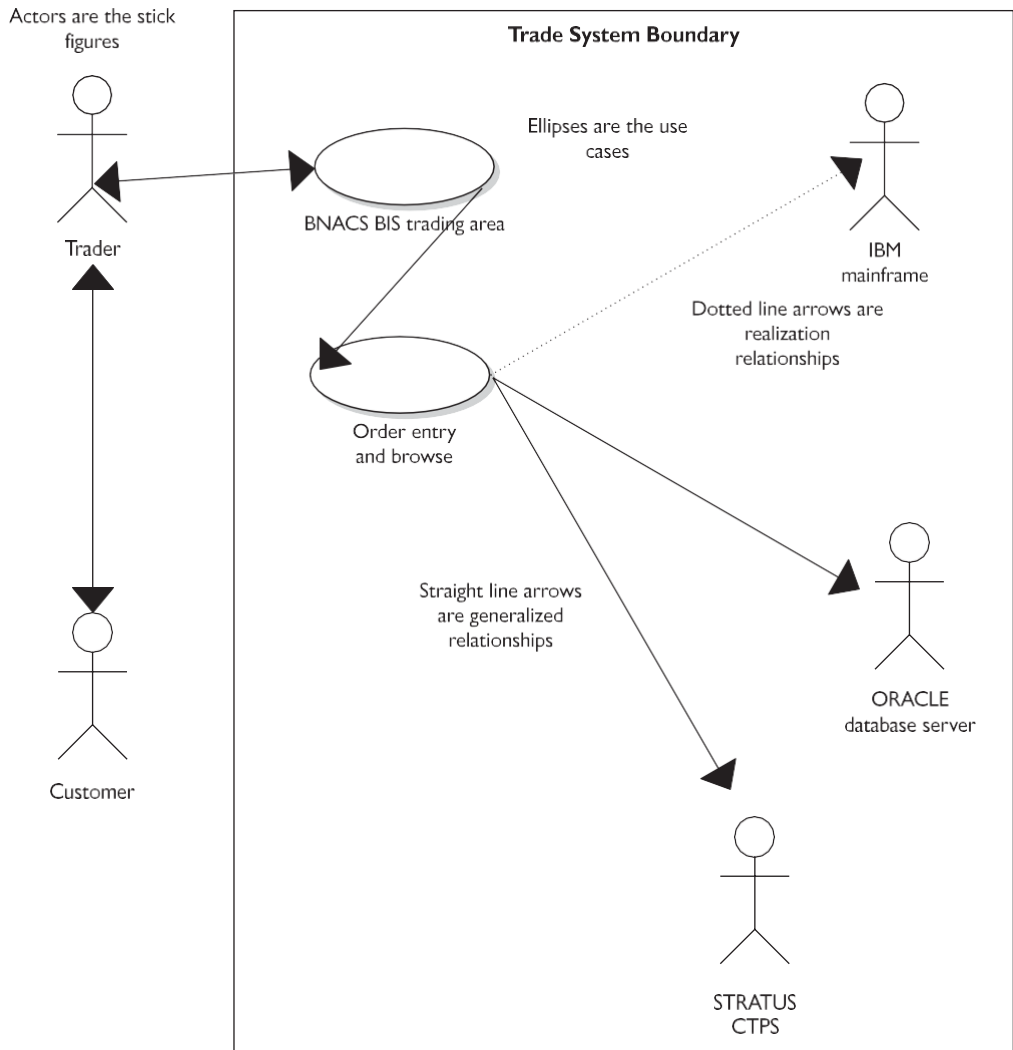
**FIGURE 3-10** Annotated use case diagram

Figure 3-10 shows a trade use case for the online trading site. The actor is a trader. The connection between actor and use case is a communication association. Actors are represented by stick figures. Use cases are represented by ovals. A common issue regarding drawing use cases is having two “actions” tied to each other, essentially showing a “flowchart.” In Figure 3-10, the trading system menu is invoked before the

“order browse” functionality and subsequent calls to the Stratus CTPS and Oracle database. Communications are represented by lines that link actors to use cases.

A use case diagram is a collection of actors, use cases, and their communications. A single use case can have multiple actors. A system boundary rectangle separates the system from the external actors. A use case generalization shows that one use case is a special kind of another use case. Use case diagrams are important to use when you are

- Determining new requirements
- Communicating with clients—their simplicity makes use case diagrams a good way to communicate the system to users
- Validating the system—the different scenarios for a use case make a good set of test cases

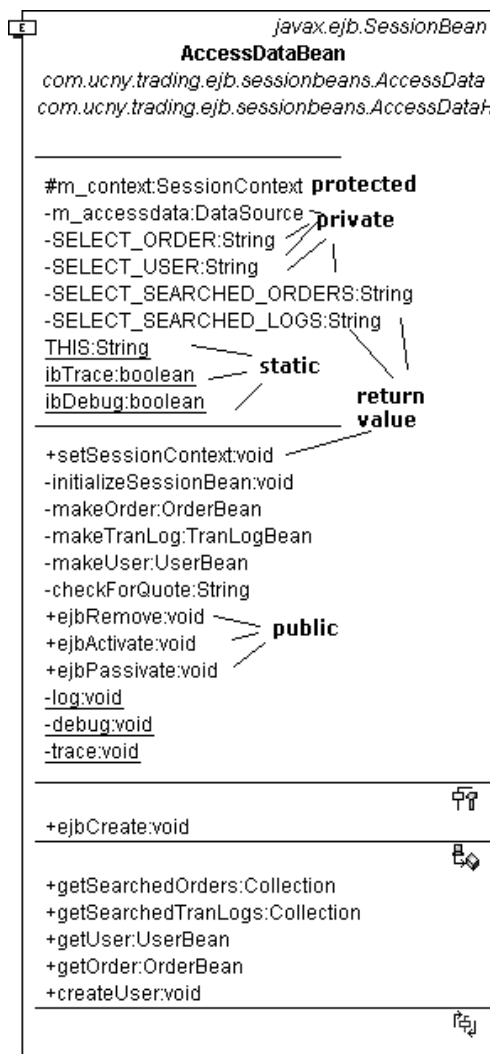
## Class Diagram

A class diagram provides an overview of a system by showing its classes and the relationships among them. Class diagrams are static; they display what interacts but not what happens when they do interact. The class diagram shown in Figure 3-11 models an EJB session bean used to order equities from a securities market. The central method is `makeOrder` that creates and returns an `orderBean`. Associated with it is the `makeUser` that creates and returns a `userBean`. UML class notation is a rectangle divided into three parts: class name, attributes, and operations. Names of abstract classes, such as *com.ucny.trading.ejb.sessionbeans.AccessData*, are in italics. Relationships among classes are the connecting links.

A class diagram can have three kinds of relationships:

- *Association* is a relationship between instances of the two classes. An association exists between two classes if an instance of one class must know about the other to perform its work. In a diagram, an association is a link connecting two classes.
- *Aggregation* is an association in which one class belongs to a collection. An aggregation shows a diamond end pointing to the part containing the whole.
- *Generalization* is an inheritance link indicating one class is a superclass of another. A generalization shows a triangle pointing to the superclass.

An association has two ends. An end may include a role name to clarify the nature of the association. For example, an *OrderDetail* is a line item of each *Order*.

**FIGURE 3-11** Annotated class diagram

A navigability arrow on an association shows which direction the association can be traversed or queried. An *OrderDetail* can be queried about its *Item*, but not the other way around. The arrow also lets you know who “owns” the association’s implementation; in this case, *OrderDetail* has an *Item*. Associations with no navigability arrows are bidirectional.

The multiplicity of an association end is the number of possible instances of the class associated with a single instance of the other end. Multiplicities, shown in the following table, are single numbers or ranges of numbers. In our example, there can be only one *User* for each *Order*, but a *User* can have any number of *Orders*.

Multiplicities	Meaning
0..1	Zero or one instance; the notation <i>n</i> . . <i>m</i> indicates <i>n</i> to <i>m</i> instances
0..* or *	No limit on the number of instances (including none)
1	Exactly one instance
1..*	At least one instance

Every class diagram has classes, associations, and multiplicities. Navigability and roles are optional items placed in a diagram to provide clarity. The class notation is a three-piece rectangle with the class name, attributes, and operations. Attributes and operations can be labeled according to access and scope.

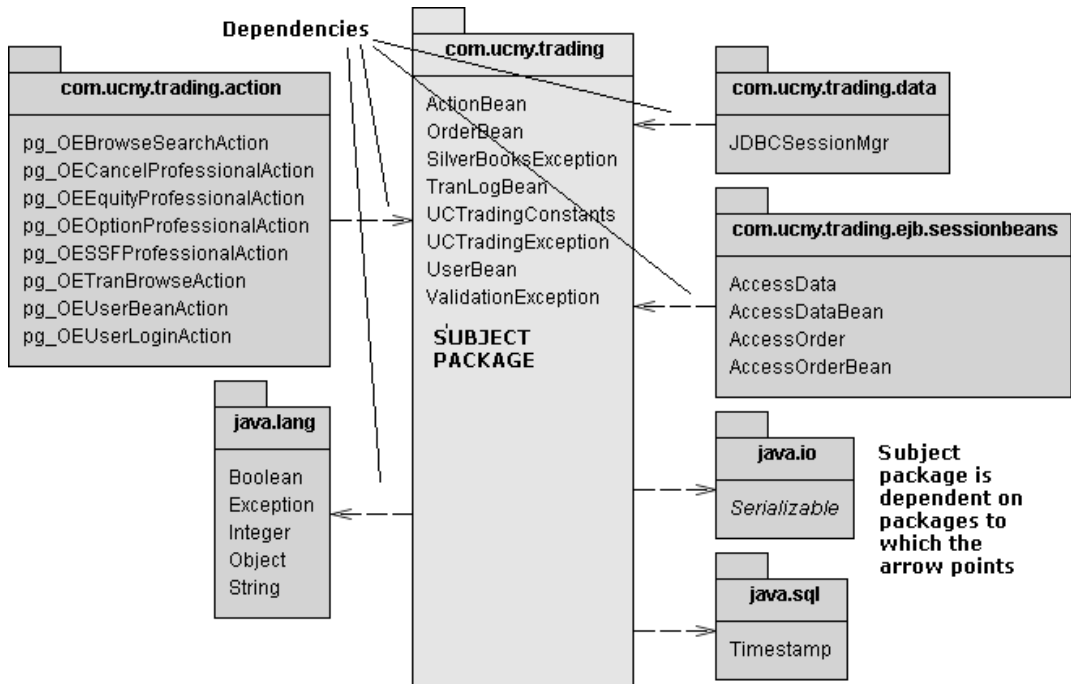
It is preferable that you name classes as singular nouns, such as *User* instead of *Users*. Static members are underlined, and Instance members are not. The operations follow this form: <access specifier> <name> ( <parameter list> ): <return type>. The parameter list shows each parameter type preceded by a colon. Access specifiers, shown in the following, appear in front of each member.

Symbol	Access
+	Public
-	Private
#	Protected

Package Diagram

To simplify complex class diagrams, you can group classes into *packages*. A package is a collection of logically related UML elements. The diagram shown in Figure 3-12 is a business model in which the classes are grouped into packages. Packages appear as rectangles with small tabs at the top. The package name is on the tab or inside the rectangle. The dotted arrows show dependencies. One package depends on another if changes in the other could possibly force changes in the first. Object diagrams show instances instead of classes. They are useful for explaining small pieces with complicated relationships, especially recursive relationships.



**FIGURE 3-12** Annotated package diagram

## Sequence Diagrams


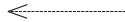




The sequence diagram shows the explicit series of interactions as they flow through the system to cause the desired objective or result. The sequence view is especially useful in systems with time-dependent functionality (such as real-time applications) and for complex scenarios where time dependencies are critical. It has two dimensions:

- One that represents time
- Another that represents the various objects participating in a sequence of events required for a purpose

Usually, only the sequence of events to which the objects of the system are subject is important; in real-time applications, the time axis is an important measurement. This view identifies the roles of the objects in your system through the sequence of states they traverse to accomplish the goal. This view is an event-driven perspective of the system. The relationships among the roles is not shown.

Class and object diagrams present static views. Interaction diagrams are dynamic. They describe how objects collaborate or interact. A sequence diagram is an interaction diagram that details the functionality and messages (requests and responses) and their timing. The time progresses as you move down the page. The objects involved in the operation are listed from left to right according to when they take part in the message sequence. Figure 3-13 shows a sequence diagram that illustrates the software calls and hardware used to service the calls in a sequence of time, with synchronous messages between each object in the diagram.

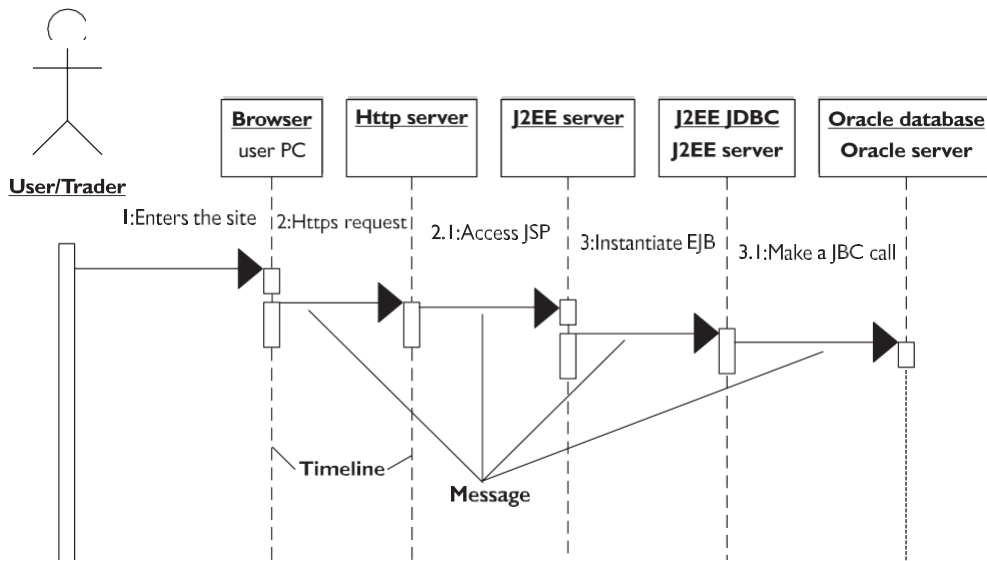
Each vertical dotted line in Figure 3-13 is a *lifeline*, representing the time that an object exists. Each arrow is a message call. An arrow goes from the sender to the top of the activation bar of the message on the receiver’s lifeline. The activation bar represents the duration of execution of the message. The sequence diagram can have a clarifying note, text inside a dog-eared rectangle. Notes can be put into any kind of UML diagram. The UML uses the following message conventions for sequence diagrams:

Symbol	Meaning
	Simple message that may be synchronous or asynchronous
	Simple message return (optional)
	A synchronous message
 or  or 	An asynchronous message

Collaboration

Collaboration diagrams are also interaction diagrams. They convey the same information as sequence diagrams, but they focus on object roles instead of the times that messages are sent. In a sequence diagram, object roles are the vertices and messages are the connecting links. The object-role rectangles are labeled with either class or object names (or both). Class names are preceded by colons (:). Each message in a collaboration diagram has a sequence number. The top-level message is number 1. Messages at the same level (sent during the same call) have the same decimal prefix but suffixes of 1, 2, 3, and so on, according to when they occur.

The collaboration diagram is similar to the sequence diagram in terms of the information displayed, but it’s different in its depiction. A collaboration diagram shows the relationships among objects. It is intended to assist in the understanding

**FIGURE 3-13** Annotated sequence diagram

the effects on a given object. It provides a procedural perspective rather than a chronological view. A collaboration diagram shows interactions organized around the objects in a particular interaction, especially their links to one another. A collaboration diagram shows the relationships among the object roles.

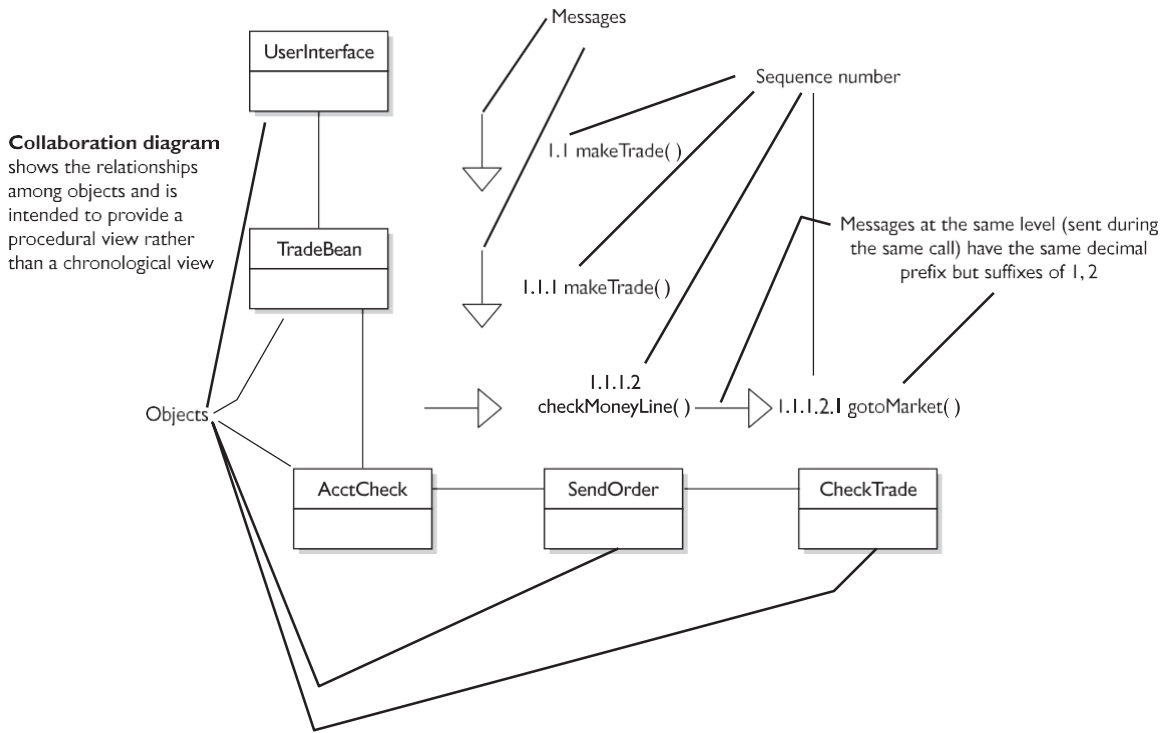
The collaboration diagram shown in Figure 3-14 shows you a model of the behavior of the objects in the trading system and the messages involved in accomplishing a purpose—in this case, making a trade (checking the trader account for sufficient funds and sending the order to the market place), projected from the larger trading system of which this collaboration is just a part. It is a representation of a set of participants and relationships that are meaningful for a given set of functionality.

The description of behavior itself involves two characteristics:

- The structural description of its participants
- The behavioral description of its execution

These two characteristics are combined, but they can be separated, because at times it is useful to describe the structure and behavior separately.

Collaboration diagrams can be enhanced by the inclusion of the dynamic behavior of the message sequences exchanged among objects to accomplish a specific purpose.

**FIGURE 3-14** Annotated collaboration diagram

This is called an *interaction*, and it helps in understanding the dynamics of the system and its participating objects.

## State

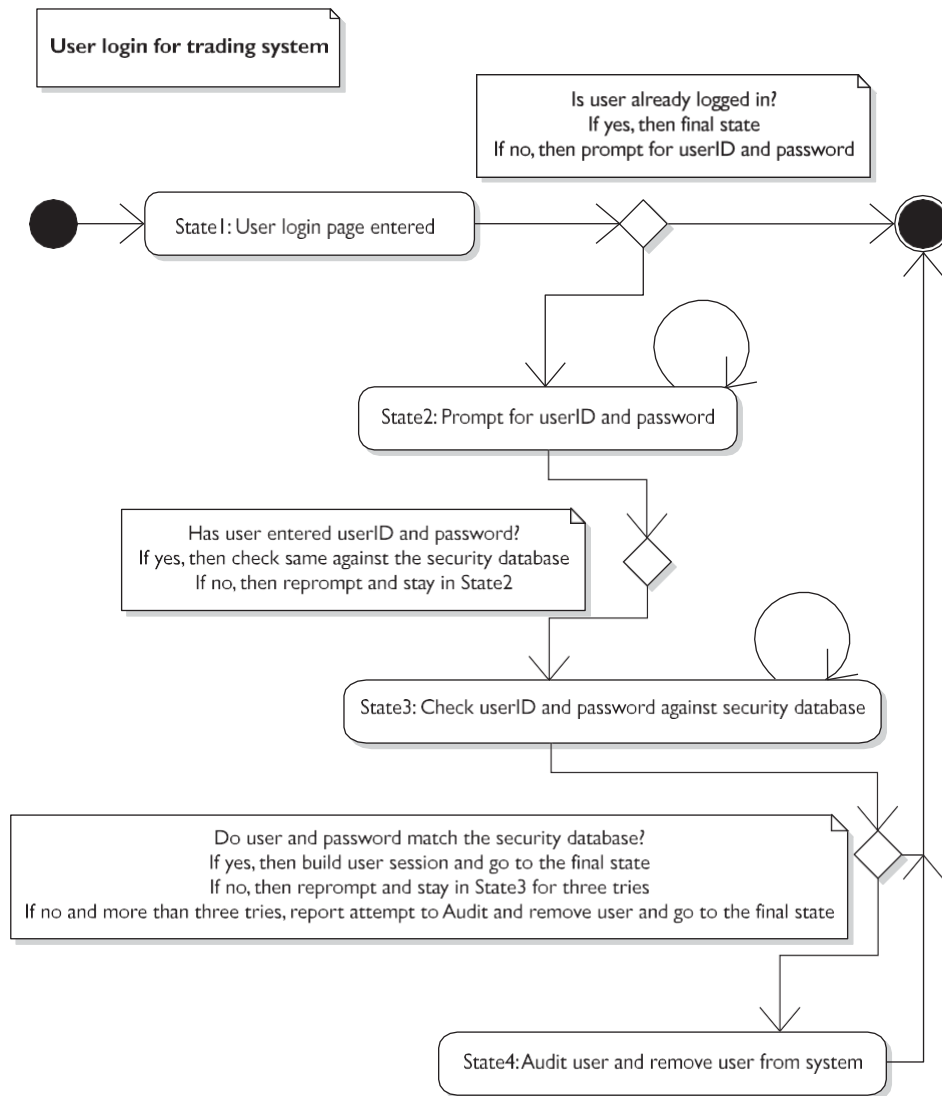
Objects have *state* or *status*. The state of an object depends on the current activity or condition. A state diagram illustrates the states of the object and the input and transitions that cause changes in the state. The state diagram shows the sequences of states that an object passes through during its lifetime. They correspond to prompts for input coupled with the responses and actions.

A *state machine* is a diagram of states and transitions that describe the response of an object of a given class to the receipt of external stimuli, and it is generally attached to a class or a method. A state diagram represents a state machine: a state being a condition during the life of an object or an interaction during which it satisfies some condition, performs some action, or waits for some event. A state may correspond to ongoing activity. Such activity is expressed as a nested state machine. For example, you may reprompt the user to enter missing form items that are required to process

a transaction, such as user login. Alternatively, ongoing activity may be represented by a pair of actions—one that starts the activity on entry to the state and one that terminates the activity on exit from the state.

The example state diagram shown in Figure 3-15 models the login part of an online trading system. Logging in consists of entering a valid user ID and password, and then submitting the information for validation against a security database of

**FIGURE 3-15** Annotated state diagram



valid users and their passwords. Logging in can be factored into four nonoverlapping states: checking whether user ID is logged in, getting user ID and password, validating same, and rejecting/accepting the user. From each state comes a complete set of transitions that determine the subsequent state.

### Activity Diagram

An activity diagram is essentially a fancy flowchart. Activity diagrams and state diagrams are related. An activity diagram—in a similar manner to the relationship between an object and class diagram—is a special case of a state diagram in which all the states are action states and all the transitions are triggered by completion of the actions in the source states. The entire activity diagram is attached to a class or a use case. The purpose of this diagram is to focus on the functionality that flows from internal processing. Activity diagrams are used in situations for which the events represent the completion of internally generated actions—that is, procedure flow. State diagrams, on the other hand, are used in situations for which asynchronous events predominate. Figure 3-16 shows the process for making a trade.

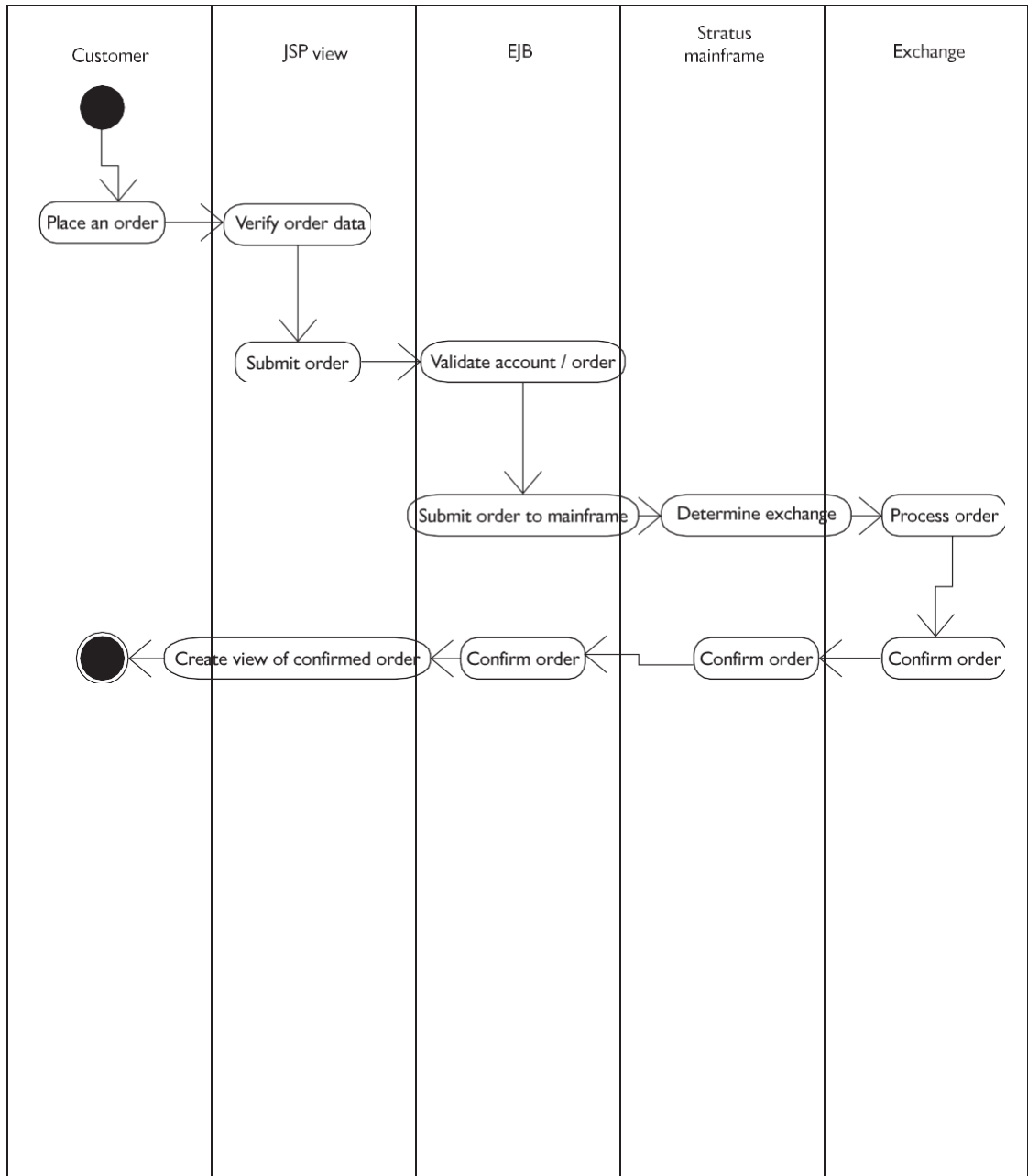
### Component

Component diagrams are physical versions of class diagrams. A component diagram shows the relationships and dependencies between software components, including Java source code components, Java class components, and Java deployable components—JAR (Java Archive) files. Within the deployment diagram, a software component may be represented as a component type.

With respect to Java and J2EE, some components exist at compile time (such as *makeTrade.java*), some exist at archive time (*makeTrade.class*), and some exist at runtime (*Trade.ear*); some exist at more than one time. So you can say that a compile-only component is one that is meaningful only at compile time; the runtime component in this case would be an executable program. You can think of this diagram as a kind of compile, JAR, and deploy description.

### Deployment Diagram

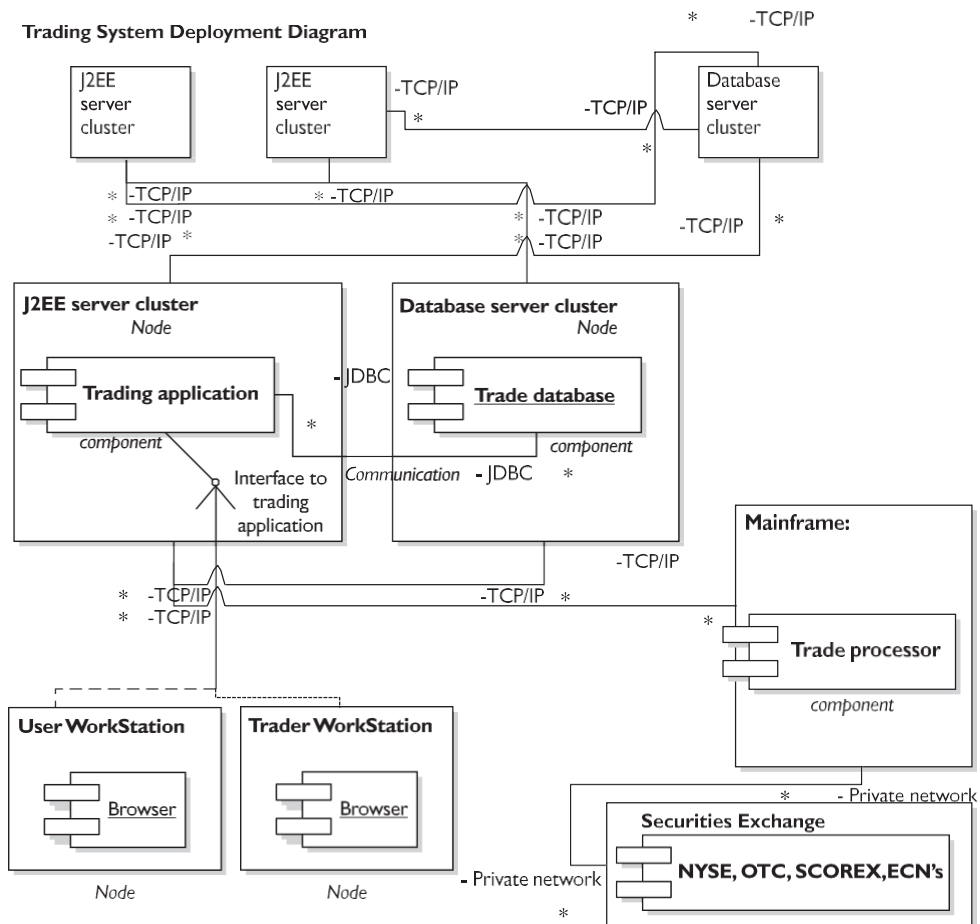
Deployment diagrams show the physical configurations of software and hardware. The deployment diagram complements the component diagram. It shows the configuration of runtime processing elements such as servers and other hardware and the software components, processes, and objects that they comprise. Software component instances represent runtime manifestations of classes. Components that do not exist as runtime entities (such as *makeTrade.java*) do not appear on these diagrams; they are shown on component diagrams. A deployment diagram is a graphical representation of nodes connected by communication links or associations. Nodes may contain

**FIGURE 3-16** Annotated activity diagram



component instances, which indicate that the component resides and runs on the node. Components may contain objects, which indicate that the object is part of the component. The deployment diagram can be used to show which components run on which nodes. The migration of components from node to node or objects from component to component may also be represented. The deployment diagram shown in Figure 3-17 depicts the relationships among software and hardware components involved in security trading transactions.

**FIGURE 3-17** Annotated deployment diagram



## **CERTIFICATION SUMMARY**

The UML is a language used for specifying, constructing, visualizing, and documenting the components of a software system. The primary design goals of the UML areas follow:

- Provide users with a visual modeling language to develop and exchange comprehensive models.
- Provide mechanisms for extensibility and specialization that extend the core concepts.
- Create a standard specification that is independent of particular computing languages.
- Provide a formal base for a modeling language.
- Support high-level development concepts such as components, collaborations, frameworks, and patterns.
- Integrate best practices.



## TWO-MINUTE DRILL

UML defines the following elements:

- ❑ **Class** Any uniquely identified abstraction that models a single thing, where the term *object* is synonymous with *instance*. Classes have *attributes* and *methods*.
- ❑ **Interface** A collection of operations that represents a class or specifies a set of methods that must be implemented by the derived class. An interface typically contains nothing but virtual methods and their signatures.
- ❑ **Package** Used to organize groups of like kind elements. The package is the only group type element and its function is to represent a collection of functionally similar classes.
- ❑ **Collaboration** Defines the interaction of one or more roles along with their contents, associations, relationships, and classes.
- ❑ **Use Case** A description that represents a complete unit of functionality provided by something as large as a system or as small as a class.
- ❑ **Component** Represents a modular and deployable system part. It encapsulates an implementation and exposes a set of interfaces.
- ❑ **Node** A physical element object that represents a processing resource, generally having memory and processing capability, such as a server.
- ❑ **State** A condition that can occur during the life of an object. It can also be an interaction that satisfies some condition, performs some action, or waits for some event.

UML defines the following relationships:

- ❑ **Generalization** A specialized version of another class.
- ❑ **Association** Uses the services of another class.
- ❑ **Aggregation** A class “owns” another class.
- ❑ **Composition:** A class is composed of another class. Refers to an aggregation within which the component parts and the larger encompassing whole share a lifetime.
- ❑ **Refinement** A refined version of another class.
- ❑ **Dependency** A class dependent on another class.

UML defines the following diagrams:

- ❑ **Use case diagram** Used to identify the primary elements and processes that form the system. The primary elements are termed as *actors* and the processes are called *use cases*. The use case diagram shows which actors interact with each use case.
- ❑ **Class diagram** Used to define a detailed design of the system. Each class in the class diagram may be capable of providing certain functionalities. The functionalities provided by the class are termed *methods* of the class.
- ❑ **Package diagram** Groups objects or classes.
- ❑ **State diagram** Represents the different states that objects in the system undergo during their lifecycle. Objects in the system change states in response to events.
- ❑ **Activity diagram** Captures the process flow of the system. An activity diagram also consists of activities, actions, transitions, and initial and final states.
- ❑ **Sequence diagram** Represents the interaction between different objects in the system. The important aspect of a sequence diagram is that it is time ordered. Objects in the sequence diagram interact by passing messages.
- ❑ **Collaboration diagram** Groups together the interactions between different objects. The interactions are listed as numbered interactions that help to trace the sequence of the interactions. The collaboration diagram helps to identify all the possible interactions that each object has with other objects.
- ❑ **Component diagram** Represents the high-level parts that make up the system. This diagram depicts what components form part of the system and how they are interrelated. It depicts the components culled after the system has undergone the development or construction phase.
- ❑ **Deployment diagram** Captures the configuration of the runtime elements of the application. This diagram is useful when a system is complete and ready for deployment.

UML can be used to view a system from various perspectives:

- ❑ **Design view** Structural view of the system; class diagrams and package diagrams form this view of the system.
- ❑ **Process view** Dynamic behavior of a system; state diagram, activity diagram, sequence diagram, and collaboration diagram form this view.

- ❑ **Component view** Software and hardware modules of the system modeled using the component diagram.
- ❑ **Deployment view** The deployment diagram of UML is used to combine component diagrams to depict the implementation and deployment of a system.
- ❑ **Use Case view** View a system from this perspective as a set of activities or transactions; use case diagrams.

## SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all the choices carefully because there may be more than one correct answer. Choose all correct answers for each question.

1. Which one of the following items is *not* one of the phases of the Unified Process?
  - A. Inception
  - B. Design
  - C. Construction
  - D. Transition
2. What *is* true about a use case?
  - A. It is a complete end-to-end business process that satisfies the needs of a user.
  - B. It is a description that represents a complete unit of functionality provided by something as large as a system or as small as a class.
  - C. It defines the interaction of one or more roles along with their contents, associations, relationships, and classes.
  - D. It is a collection of operations that represents a class or specifies a set of methods that must be implemented by the derived class.
3. Which item is *not* true when speaking of a class?
  - A. A class is a nonunique structure.
  - B. An *instance* is one computer executable copy of a class, also referred to as an *object*.
  - C. Multiple instances of a particular class can exist in a computer's main memory at any given time.
  - D. A class is a structure that defines the attribute data and the methods or functions that operate on that data.
4. What is *not* true about use cases?
  - A. There are three types of use cases: essential, real, and virtual.
  - B. A virtual use case describes the user's virtual view of the problem and is technology independent.
  - C. A real use case describes the process in terms of its real design and implementation.
  - D. Essential use cases are of importance early in the project. Their purpose is to illustrate and document the business process.

5. What is *not* true about a sequence diagram?
  - A. It has two dimensions.
  - B. One sequence diagram dimension represents time.
  - C. One sequence diagram dimension represents the different objects participating in a sequence of events required for a purpose.
  - D. Sequence diagrams are static model views.
6. Which item is *not* an example of things that a state diagram could effectively model?
  - A. Life could be modeled: birth, puberty, adulthood, death.
  - B. A computer system infrastructure.
  - C. A banking transaction.
  - D. A soccer match could be modeled: start, half time, injury time, end.
7. What is *not* true about a collaboration diagram?
  - A. A collaboration diagram models interactions among objects, and objects interact by invoking messages on each other.
  - B. A collaboration diagram groups together the interactions among different objects.
  - C. The interactions in a collaboration diagram are listed as alphabetically collated letters that help to trace the sequence of the interactions.
  - D. The collaboration diagram helps to identify all the possible interactions that each object has with other objects.
8. What item is *not* true about a component?
  - A. A component represents a modular and deployable system part. It encapsulates an implementation and exposes a set of interfaces.
  - B. The component interfaces represent services provided by elements that reside on the component.
  - C. A node may be deployed on a component.
  - D. A component is shown as a rectangle with two smaller rectangles extending from its left side. A component type has a type name *component-type*.
9. Which item(s) is *not* part of a class in a UML class diagram?
  - A. Name
  - B. Attributes
  - C. Method
  - D. Comments



10. Which item is *not* one of the three kinds of relationships a class diagram can have?
  - A. Association
  - B. Aggregation
  - C. Generalization
  - D. Specialization
11. In a class diagram, what does a line with an arrow from one class to another denote?
  - A. Attribute visibility
  - B. Class visibility
  - C. Method visibility
  - D. Global visibility
12. What is *not* a type of visibility between objects?
  - A. Local
  - B. Method
  - C. Attribute
  - D. Global
13. Which statement is *not* true about state machine and state diagrams?
  - A. A state machine is basically a diagram of states and transitions that describes the response of an object of a given class to the receipt of external stimuli, and it is generally attached to a class or a method.
  - B. The state diagram shows the sequences of states that an object passes through during its lifetime.
  - C. A state diagram represents a state machine: a state being a condition during the life of an object or an interaction during which it satisfies some condition, performs some action, or waits for some event.
  - D. State diagrams are used in situations for which all or most of the events represent the completion of internally generated actions (that is, procedural flow of control).
14. Which of the following UML diagrams may be best suited for a business analyst?
  - A. Deployment
  - B. Class
  - C. Use case
  - D. Activity
  - E. Collaboration
  - F. Sequence

15. In a UML class diagram, Private, Protected, and Public attributes are shown by which one of the following sets of symbols?
- A. -, +, #
  - B. +, -, hash
  - C. #, -, +
  - D. -, #, +
  - E. +, #, -
  - F. #, +, -

## SELF TEST ANSWERS

1. ☐ **B** is correct because design is not a phase in the unified process.  
**B A, C, and D** are incorrect because the phases of the unified process include inception, whose focus is the scope of the project; elaboration, in which the architecture and the requirements of the product being built must be defined by the end of this phase; construction, during which the software must be developed or constructed; and transition, during which the software must be rolled out to users.
2. ☐ **A and B** are correct because a use case is a complete end-to-end business process that satisfies the needs of a user. It is also a description that represents a complete unit of functionality provided by something as large as a system or as small as a class.  
**B C and D** are incorrect because a collaboration defines the interaction of one or more roles along with their contents, associations, relationships, and classes. A class diagram is a collection of operations that represents a class or specifies a set of methods that must be implemented by the derived class.
3. ☐ **A** is correct because a class is unique.  
**B B, C, and D** are incorrect because they are true. A class is a unique structure that defines the attribute data and the methods or functions that operate on that data. An instance is one computer executable copy of a class, also referred to as an object. Multiple instances of a particular class can exist in a computer's main memory at any given time.
4. ☐ **A and B** are correct because they are false. There are two types of use cases: essential and real.  
**B C and D** are incorrect because they are true. Essential use cases are expressed in an ideal form that remains free of technology and implementation detail. The design decisions are abstracted, especially those related to the user interface. A real use case describes the process in terms of its real design and implementation. Essential use cases are of importance early in the project. Their purpose is to illustrate and document the business process. Real use cases become important after implementation, as they document how the user interface supports the business process documented in the essential use case.
5. ☐ **D** is correct because it is false. Class and object diagrams are static model views; sequence diagrams are dynamic.  
**B A, B, and C** are incorrect because they are true. The sequence diagram shows the explicit sequence of interactions as they flow through the system to affect a desired operation or result. It has two dimensions; one dimension represents time, and another dimension represents the different objects participating in a sequence of events required for a purpose. Class and object diagrams are static model views.

6. ☐ **B** is correct because it is false. A computer system infrastructure does not have dynamic states; it is more or less static and the modeler would use a deployment diagram to depict the infrastructure.  
**B** **A**, **C**, and **D** are incorrect because they are true. Life could be modeled. A banking transaction and a soccer match could also be modeled.
7. ☐ **C** is correct because it is false. The interactions in a collaboration diagram are listed as numbered interactions that help to trace the sequence of the interactions.  
**B** **A**, **B**, and **D** are incorrect because they are true. A collaboration diagram models interactions among objects, and objects interact by invoking messages on each other. A collaboration diagram groups together the interactions among different objects. The interactions in a collaboration diagram are listed as numbered interactions that help to trace the sequence of the interactions.
8. ☐ **C** is correct because it is false. A component may be deployed on a node.  
**B** **A**, **B**, and **D** are incorrect because they are true. A component represents a modular and deployable system part. It encapsulates an implementation and exposes a set of interfaces. The interfaces represent services provided by elements that reside on the component. A component is shown as a rectangle with two smaller rectangles extending from its left side.
9. ☐ **D** is correct because it is false. A comment is not part of a UML class diagram.  
**B** **A**, **B**, and **C** are incorrect because they are true. UML class notation is a rectangle divided into three parts that include class name, attributes, and operations.
10. ☐ **D** is correct because it is false. Specialization is not a relationship type.  
**B** **A**, **B**, and **C** are incorrect because they are true. Association is a relationship between instances of the two classes. An association exists between two classes if an instance of one class must know about the other to perform its work. In a diagram, an association is a link connecting two classes. Aggregation is an association in which one class belongs to a collection. An aggregation has a diamond end pointing to the part containing the whole. Generalization is an inheritance link indicating one class is a superclass of the other. A generalization has a triangle pointing to the superclass.
11. ☐ **A** is correct.  
**B** **B**, **C**, and **D** are incorrect.
12. ☐ **B** is correct.  
**B** **A**, **C**, and **D** are incorrect.

13. ☐ **D** is correct because it is false. Activity diagrams are used in situations for which all or most of the events represent the completion of internally generated actions (that is, procedural flow of control). State diagrams, on the other hand, are used in situations for which asynchronous events predominate.
- B** **A**, **B**, and **C** are incorrect because they are true. The state diagram shows the sequences of states through which an object passes during its lifetime. They correspond to prompts for input couples with the responses and actions. A state machine is basically a diagram of states and transitions that describe the response of an object of a given class to the receipt of external stimuli, and it is generally attached to a class or a method. A state diagram represents a state machine: a state being a condition during the life of an object or an interaction during which it satisfies some condition, performs some action, or waits for some event.
14. ☐ **C** is correct because use case diagrams show a set of use cases and actors and their relationships. Use case diagrams show the static view of a system. These diagrams are especially important in organizing and modeling the behaviors of a system. Use case diagrams are frequently used by business analysts to capture business requirements of a system.
- B** **A**, **B**, **D**, **E**, and **F** are incorrect. Deployment diagrams show the configuration of runtime processing nodes and the components that live within these nodes. Deployment diagrams address the static view of the architecture. Architects frequently use deployment diagrams. A class diagram shows a set of classes, interfaces, and collaborations and their relationships. Class diagrams address the static design view of a system. Software designers frequently use class diagrams. Activity diagrams are a special kind of state chart diagram that shows the flow from activity to activity within the system. This type of diagram is important in modeling the function of a system and emphasizing the flow of control among objects. Designers and developers frequently use activity diagrams. A collaboration diagram is an interaction diagram that emphasizes the structural organization of objects that send and receive messages. Designers and developers frequently use interaction diagrams.
15. ☐ **D** is correct because in UML notation, access modifiers are shown by the -, #, and + symbols to represent private, protected, and public, respectively.
- B** **A**, **B**, **C**, **E**, and **F** are incorrect because they do not have the right combination.