Appendix B

# Quick Reference for UML

## Objectives

Upon completion of this appendix, you should be able to:

- Understand the fundamentals of UML
- Describe general UML elements
- Describe use case diagrams
- Describe class diagrams
- Describe object diagrams
- Describe collaboration diagrams
- Describe sequence diagrams
- Describe statechart diagrams
- Describe activity diagrams
- Describe component diagrams
- Describe deployment diagrams

# Additional Resources

**Additional resources –** The following references provide additional details on the topics described in this appendix:

- Booch, Grady, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Reading: Addison Wesley Longman, Inc., 1999.
- Folwer, Martin, with Kendall Scott. *UML Distilled (2nd ed)*. Reading: Addison Wesley Longman, Inc., 2000.
- The Object Management Group. “OMG Unified Modeling Language Specification,” version 1.4, September 2001, [`http://www.omg.org/technology/documents/formal/uml.htm`], accessed 22 April 2004.

**Note –** Additional UML resources are available online at: `http://www.omg.org/uml/`.

# UML Fundamentals

Unified Modeling Language (UML) is a graphical language for modeling software systems. UML is not a programming language, it is a set of diagrams that can be used to specify, construct, visualize, and document software designs. Software engineers use UML diagrams to construct and explain their software designs, similar to how a building architect uses blueprints to construct and explain their building designs. UML has diagrams to assist in every part of the application development process from requirements gathering through design, and into coding, testing, and deployment.

UML was developed in the early 1990s by three leaders in the object modeling world: Grady Booch, James Rumbaugh, and Ivar Jacobson. Their goal was to unify the major methods that they had previously developed to create a new standard for software modelling. UML is now the most commonly used modeling language. UML is currently maintained by the OMG. The UML specification is available on the OMG web site at `http://www.omg.org/uml/`.

The UML is not a process for how to do analysis and design. UML is only a set of tools to use in a process. UML is frequently used with a process such as the Unified Software Development Process. Sun Microsystems OO-226: *Object Oriented Analysis and Design Using UML*, is a five day course that teaches effective methods of analysis and design using UML language, the Unified Software Development Process (USDP) method, and software patterns.

UML defines nine standard types of diagrams. Table B-1 provides a list of these diagrams, an informal description, and use.

**Table B-1** Types of UML Diagrams

| Diagram Name | Description | Use |
|---|---|---|
| Use Case | A Use Case diagram is a simple diagram that shows who is using your system and what processes they will perform in the system. | Use the Use Case diagram to document the Requirements Gathering and Analysis workflows. Throughout the entire development, all work should be traceable back to the Use Case diagram. |
| Class | A Class diagram shows a set of classes in the system, and the associations and inheritance relationships between the classes. Class nodes might also contain a listing of attributes and operations. | Use the Class diagram for showing the structure of the system and what needs to be programmed. Most UML case tools can generate code based on the Class diagram. |
| Object | An Object diagram shows specific object instances and the links between them. An Object diagram represents a snapshot of the system objects at a specific point in time. | Use the Object diagram to clarify or validate the Class diagram. |
| Activity | An Activity diagram is essentially a flow chart with new symbols. This diagram represents the flow of activities in a process or algorithm. | Use the activity diagram to model real world business systems during the Requirements Gathering workflow. |
| Collaboration | The Collaboration diagram and the Sequence diagram both show processes from an object oriented perspective. The main difference is that the layout of the Collaboration diagram puts more focus on the objects, rather than the sequence. | Use the collaboration diagram to focus on the objects in a sequence. Collaboration diagrams are typically more difficult to read than Sequence diagrams. |

**Table B-1** Types of UML Diagrams (Continued)

| Diagram Name | Description | Use |
|---|---|---|
| Sequence | A Sequence diagram shows a process from an object oriented perspective by showing how a process is executed by a set of objects or actors. | Use the Sequence diagram for assigning responsibilities to classes by considering how they can work together to implement the processes in the system. This is essential. |
| Statechart | A Statechart diagram shows how a particular object changes behavioral state as various events happen to it. | Use the Statechart diagram to understand objects that change behavioral states in significant ways. |
| Component | A Component diagram shows the major software components of a system and how they can be integrated. Component diagrams can contain non-OO software components, such as legacy procedural code and web documents. | Use Component diagrams to show how all of the OO and non-OO components fit together in your system. It is also a good way to look at the high-level software structure of your system. |
| Deployment | A Deployment diagram shows the hardware nodes in the system. | Use the Deployment diagram to show how a distributed system will be configured. Software components might be displayed inside the hardware nodes to show how they will be deployed. |

# General Elements

In general, UML diagrams represent concepts, depicted as symbols (also called nodes), and relationships among concepts, depicted as paths (also called links), connecting the symbols. These nodes and links are specialized for the particular diagram. For example, in Class diagrams, the nodes represent object classes, and the links represent associations between classes and generalization (inheritance) relationships.

There are other elements that augment these diagrams, including: packages, stereotypes, annotations, constraints, and tagged values.

## Packages

In UML, packages enable you to arrange your modeling elements into groups. UML packages are a generic grouping mechanism and should not be directly associated with Java technology packages. However, you can use UML packages to model Java technology packages. Figure B-1 shows a package diagram that contains a group of classes in a Class diagram.

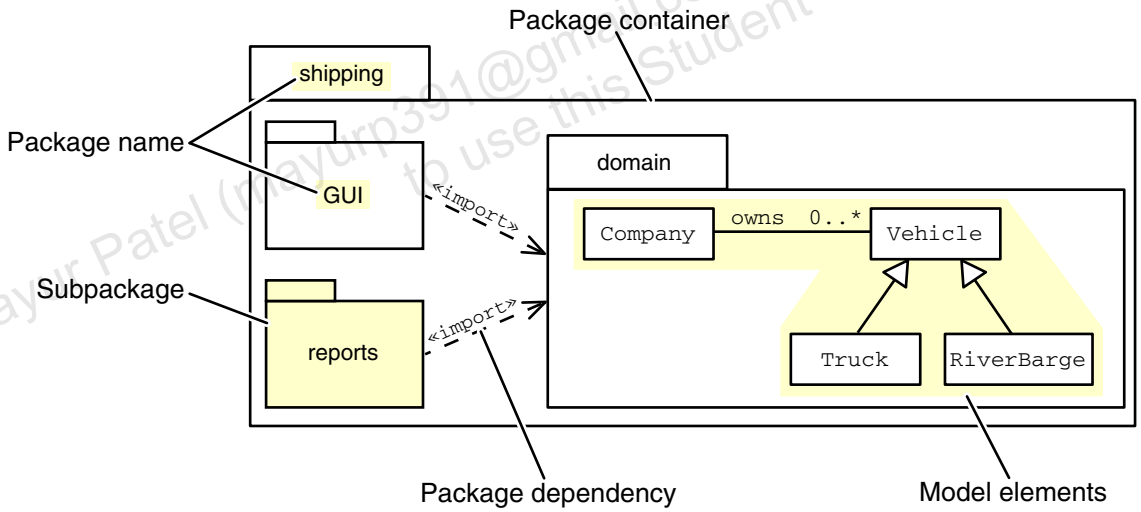


**Figure B-1** Example Package

## Mapping to Java Technology Packages

The mapping of UML packages to Java technology packages implies that the classes would contain the package declaration of package `shipping.domain`. For example, in the file `Vehicle.java`:

```
package shipping.domain;

public class Vehicle {
  // declarations
}
```

Figure B-1 on page B-6 also shows a simple hierarchy of packages. The `shipping` package contains the `GUI`, `reports`, and `domain` subpackages. The dashed arrow from one package to another indicates that the package at the tail of the arrow uses (imports) elements in the package at the head of the arrow. For example, `reports` uses elements in the `domain` package as follows:

```
package shipping.reports;

import shipping.domain.*;

public class VehicleCapacityReport {
  // declarations
}
```

**Note –** In Figure B-1 on page B-6, the `shipping.GUI` and `shipping.reports` packages have their names in the body of the package box, rather than in the head of the package box. This is done only when the diagram does not expose any of the elements in that package.

# Stereotypes

The designers of UML understood that they could not build a modeling language that would satisfy every programming language and every modeling need. They built several mechanisms into UML to enable modelers to create their own semantics for model elements (nodes and links). Figure B-2 shows the use of a stereotype tag «interface» to declare that the class node Set is a Java technology interface declaration. Stereotype tags can adorn relationships as well as nodes. There are over a hundred standard stereotypes. You can create your own stereotypes to model your own semantics.

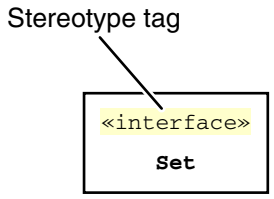


**Figure B-2** Example Stereotype

# Annotation

The designers of UML also built a method for annotating the diagrams into the UML language. Figure B-3 shows a simple annotation.

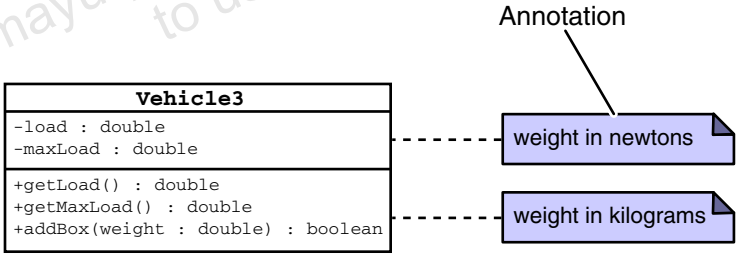


**Figure B-3** Example Annotation

Annotations can contain notes about the diagram as a whole, notes about a particular node or link, or even notes about an element within a node. The dotted link from the annotation points to the element being annotated. If there is no link from the annotation, then the note is about the whole diagram.

# Constraints

Constraints enable you to model certain conditions that apply to a node or link. Figure B-4 shows several constraints. The topmost constraint specifies that the `Player` objects must be stored in a persistent database. The middle constraint specifies that the captain and co-captain must also be members of the team's roster. The constraint on the bottom specifies the minimum number of players by gender.

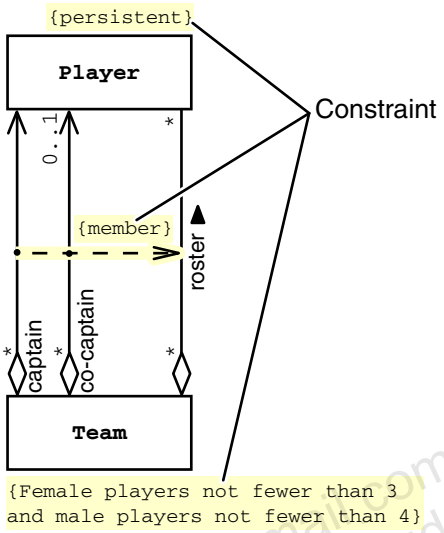


**Figure B-4** Example Constraints

# Tagged Values

Figure B-5 shows several examples of how tagged values enable you to add new properties to nodes in a diagram.

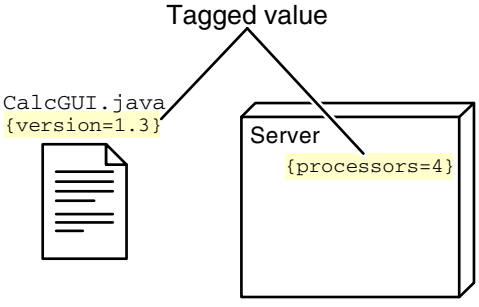


**Figure B-5** Example Tagged Values

# Use Case Diagrams

A Use Case diagram represents the functionality provided by the system to external users. The Use Case diagram is composed of actors, use case nodes, and their relationships. Actors can be humans or other systems.

Figure B-6 shows a simple banking Use Case diagram. An actor node can be denoted as a *stick figure* (as in the three Customer actors) or as a class node (see "Class Nodes" on page B-11) with the stereotype of `«actor»`. There can be a hierarchy of actors.

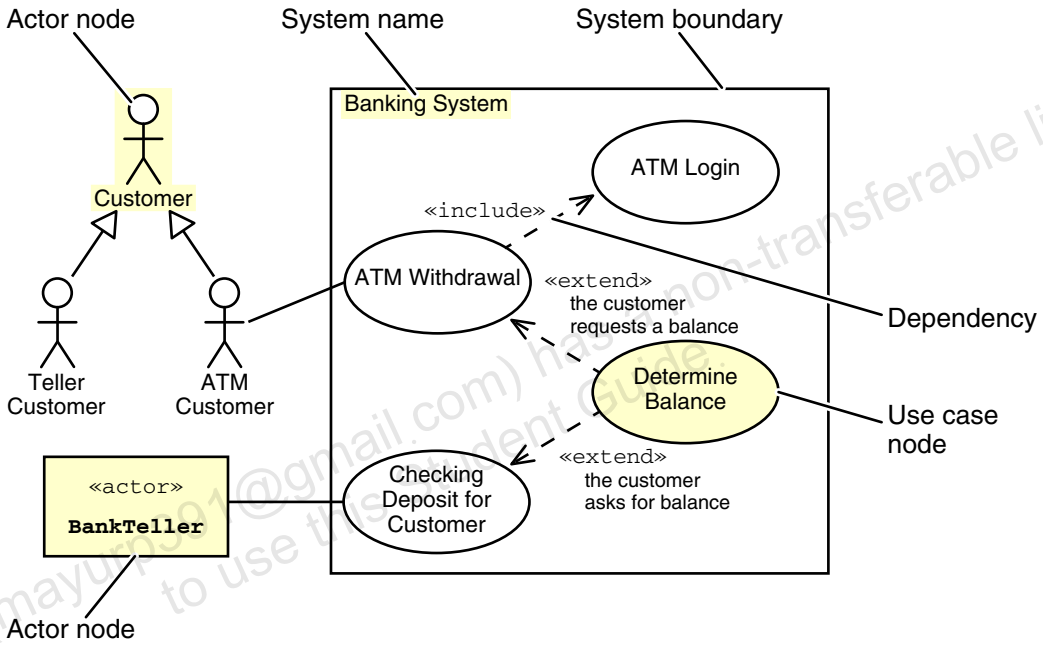


**Figure B-6** Use Case Diagram Example

A use case node is denoted by a labeled oval. The label indicates the activity summary that the system performs for the actor. Use case nodes are grouped into a system box, which is usually labeled in the top left corner. The relationship *actor uses the system to* is represented by the solid line from the actor to the use case node.

Use case nodes can depend on other use cases. For example, the ATM Withdrawl use case uses the ATM Login use case.

Use case nodes can also extend other use cases to provide optional functionality. For example, you can use the Determine Balance use case to extend the Checking Deposit for Customer use case.

# Class Diagrams

A Class diagram represents the static structure of a system. These diagrams are composed of classes and their relationships.

## Class Nodes

Figure B-7 shows several *class nodes*. You do not have to model every aspect of a class every time that class is used in a diagram.

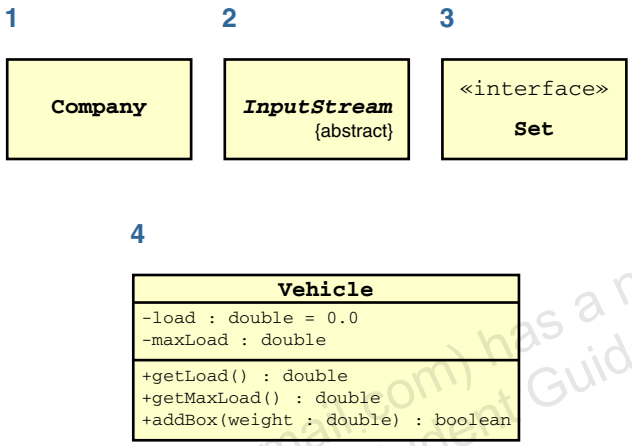


**Figure B-7** Several Class Nodes

A class node can be only the name of the class, as in Examples 1, 2, and 3 of Figure B-7. Example 1 is a concrete class, where no members are modeled. Example 2 is an abstract class (name is in italics). Example 3 is an interface. Example 4 is a concrete class, where members are modeled.

Figure B-8 shows the element of a class node.

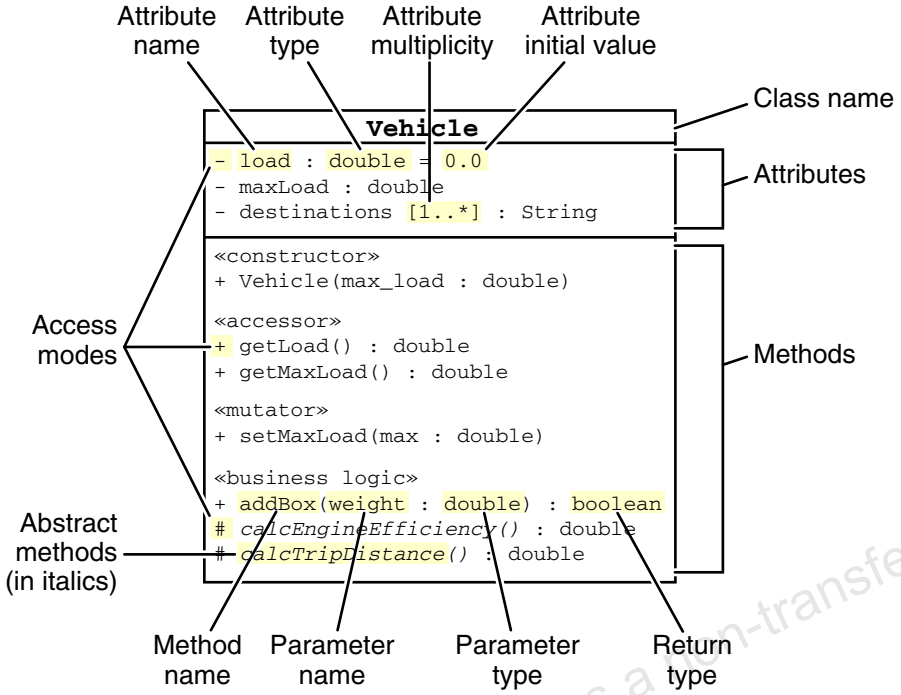


**Figure B-8** Class Node Elements

A fully specified class node has three fundamental compartments:

- The name of the class in the top is one compartment.
- The set of attributes under the first bar is a second compartment.

  An attribute is specified by five elements, including: `access mode`, `name`, `multiplicity`, `data type`, and `initial value`. All of the attribute elements are optional, except for the `name` element.
- The set of methods under the second bar is another compartment.

  A method is specified by four elements, including: `access mode`, `name`, `parameter list` (a comma-delimited list of parameter name and type), and the `return type`. All elements of a method are optional, except for the `name` element. If the return value is not specified, then no value is returned (`void`). The name of an abstract method is marked in italics.

You can use stereotypes to group attributes or methods together. For example, you can separate accessor, mutator, and business logic methods from each other for clarity. And because there is no UML-specific notation for constructors, you can use the `«constructor»` stereotype to mark the constructors in your method compartment.

Table B-2 shows the valid UML access mode symbols.

**Table B-2** UML Defined Access Modes and Their Symbols

| Access Mode | Symbol |
|---|---|
| Private | - |
| Package private | ~ |
| Protected | # |
| Public | + |

Figure B-9 shows an example class node with elements that have class (or static) scope. This is denoted by the underline under the element. For example, `counter` is a static data attribute and `getTotalCount` is a static method.

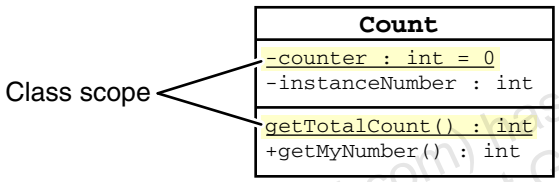


**Figure B-9** Class Node With Static Elements Example

You could write the `Count` class in the Java programming language as:

```
public class Count {
  private static int counter = 0;
  private int instanceNumber;

  public static int getTotalCount() {
    return counter;
  }
  public int getMyNumber() {
    return instanceNumber;
  }
}
```

# Inheritance

Figure B-10 shows class inheritance through the generalization relationship arrow.

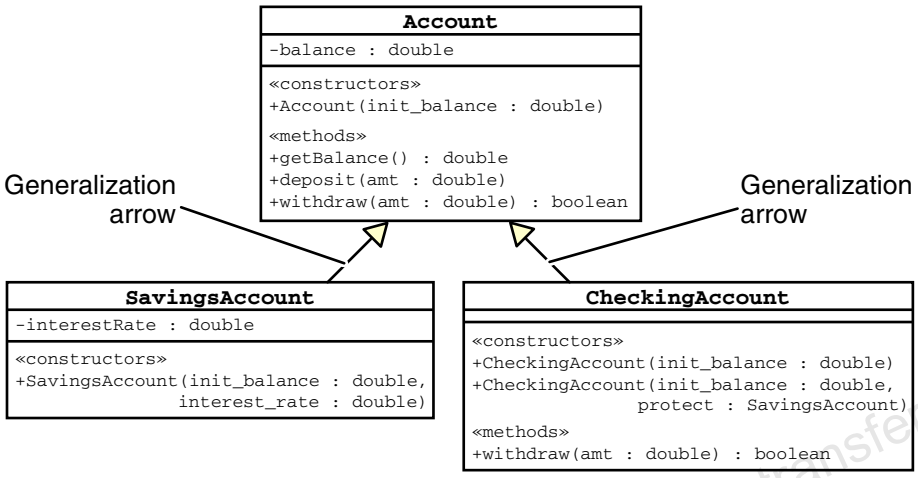


**Figure B-10** Class Inheritance Relationship

Class inheritance is implemented in the Java programming language with the `extends` keyword. For example:

```
public class Account {
  // members
}

public class SavingsAccount extends Account {
  // members
}

public class CheckingAccount extends Account {
  // members
}
```

# Interface Implementation

Figure B-11 shows how to use the Realization arrow to model a class that is implementing an interface.

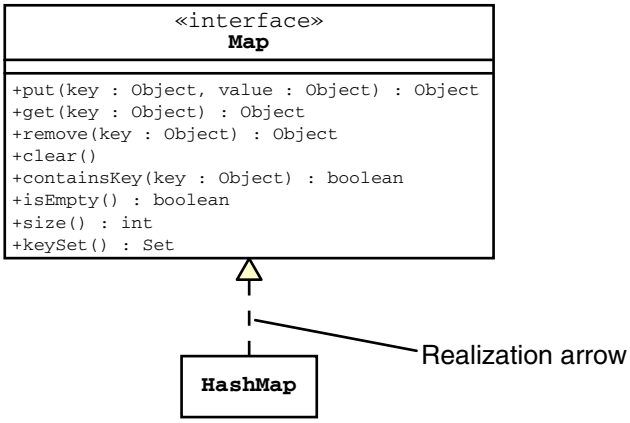


**Figure B-11** Class Implementing an Interface Example

An interface is implemented in the Java programming language with the `implements` keyword. For example:

```
public interface Map {
  // declaration here
}

public class HashMap implements Map {
  // definitions here
}
```

# Association, Roles, and Multiplicity

Figure B-12 shows an example association. An *association* is a link between two types of objects, and implies the ability for the code to navigate from one object to another.

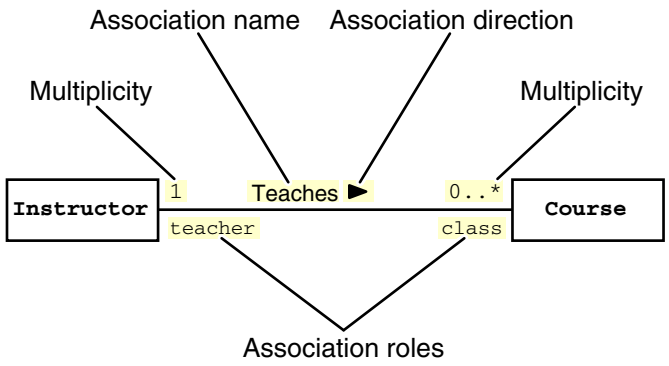


**Figure B-12** Class Associations and Roles

In this diagram, Teaches is the name of the association with a directional arrow pointing to the right. This association can be read as *an instructor teaches a course*. You can also attach roles to each end of the association. In Figure B-12, the teacher role indicates that the instructor is the teacher for a given course. All of these elements are optional if the association is obvious.

This example also demonstrates how many objects are involved in each role of the association. This is called *multiplicity*. In this example, there is only one teacher for every class, which is denoted by the `1` on the `Instructor` side of the association. Also, any given teacher might teach zero or more courses, which is denoted by the `0..*` on the `Course` side. You can leave out the multiplicity for a given role if it is always one. You can also abbreviate zero or more (`0..*`) using only an asterisk (`*`).

You can express multiplicity values as follows:

- A range of values – For example, `2..10` means at least 2 and up to 10
- A disjoint set of values – For example, `2,4,6,8,10`
- A disjoint set of values or ranges – For example, `1..3,6,8..11`

However, the most common values for multiplicity are exactly one (`1` or left blank), zero or one (`0..1`), zero or more (`*`), or one or more (`1..*`).

Associations are typically represented in the Java programming language as an attribute in the class at the tail of the relationship (specified by the direction indicator). If the multiplicity is greater than one, then some sort of collection or array is necessary to hold the elements.

Also, in Figure B-12 on page B-16 the association between an instructor and a course might be represented in the `Instructor` class as:

```
public class Instructor {
  private Course[]  classes = new Course[MAXIMUM];
}
```

or as:

```
public class Instructor {
  private List  classes = new ArrayList();
}
```

The latter representation is preferable if you do not know the maximum number of courses any given instructor might teach.

# Aggregation and Composition

An *aggregation* is an association in which one object contains a group of parts that make up the *whole* object (see Figure B-13). For example, a car is an aggregation of an engine, wheels, body, and frame. Composition is a specialization of aggregation in which the parts cannot exist independently of the whole object. For example, a human is a composition of a head, two arms, two legs, and a torso. If you remove any of these parts without surgical intervention, the whole person is going to die.

In the example in Figure B-13, a sports league, defined as a sports event that occurs seasonally every year, is a composition of divisions and schedules. A division is an aggregation of teams. A team might exist independently of a particular season. In other words, a team might exist for several seasons (leagues) in a row. Therefore, a team might still exist, even if a division is eliminated. Moreover, a game can only exist in the context of a particular schedule of a particular league.

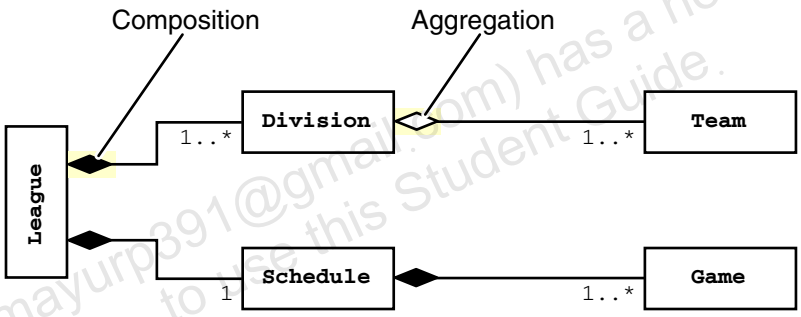


**Figure B-13** Aggregation and Composition Example

# Association Classes

An association between two classes might have properties and behavior of its own. Association classes are used to model this characteristic. For example, Figure B-14 shows that players might be required to register for a particular division within a sports league. The association class is attached to the association link by a dashed line.

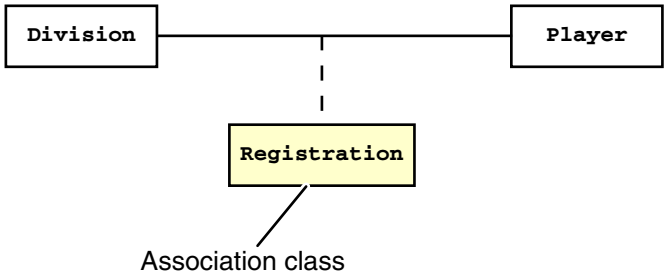


**Figure B-14** Simple Association Class

Figure B-15 shows an association class that is used by two associations and that has two private attributes. This example indicates that a `Game` object is associated with two opposing teams, and that each team will have a score and a flag specifying whether that team forfeited the game.

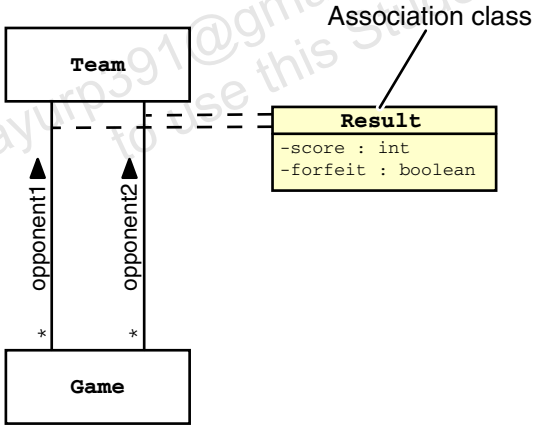


**Figure B-15** More Complex Association Class

You can use the Java programming language to represent an association class in several different ways. One way is to code the association class as a standard Java technology class. For example, `Registration` could be coded as follows:

```
public class Registration {
  private Division division;
  private Player player;
  // registration data
  // methods...
}
public class Division {
  // data
  public Registration retrieveRegistration(Player p) {
    // lookup registration info for player
    return new Registration(this, p, ...);
  }
  // methods...
}
```

Another technique is to code the association class attributes directly into one of the associated classes. For example, the `Game` class might include the score information as follows:

```
public class Game {
  // first opponent and score details
  private Team opponent1;
  private int opponent1_score;
  private boolean opponent1_forfeit;
  // second opponent and score details
  private Team opponent2;
  private int opponent2_score;
  private boolean opponent2_forfeit;
  // methods...
}
```

# Other Association Elements

There are several other parts of associations. This section presents the constraints and qualifiers elements.

An association constraint enables you to augment the semantics of two or more associations by attaching a dependency arrow between them and tagging that dependency with a constraint. For example, in Figure B-16, the captain and co-captain of a team are also members of the team's roster.

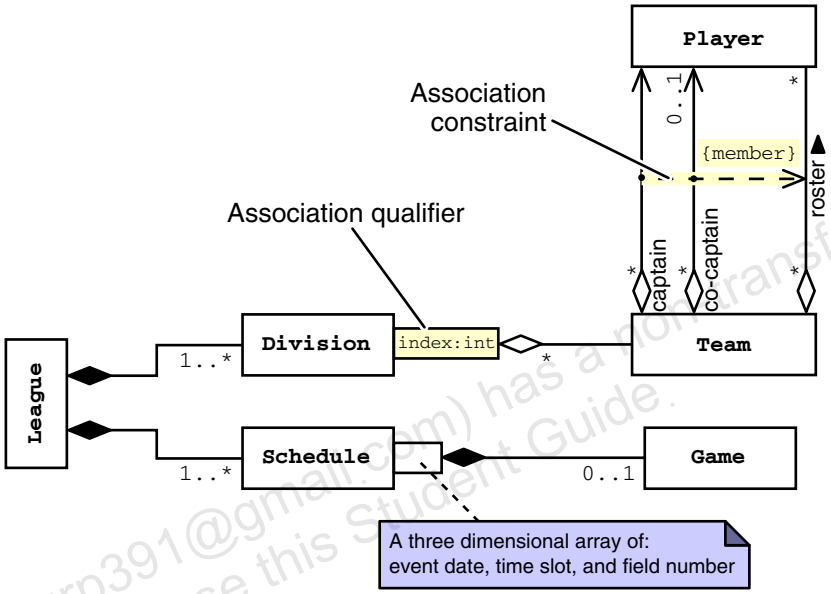


**Figure B-16** Other Associations Elements

An association qualifier provides a modeling mechanism to state that an object at one end of the association can look up another object at the other end. For example, a particular game in a schedule is uniquely identified by the following:

- An event date – Such as Saturday, August 12, 2000
- A time-slot – Such as 11:00 a.m. to 12:30 p.m.
- A particular field number – Such as field number 2

One particular implementation might be a three dimension array (for example, `Game[][][]`), in which each qualifier element (date, time-slot, and field number) is mapped to an integer index.

# Object Diagrams

An Object diagram represents the static structure of a system at a particular instance in time. These diagrams are composed of object nodes, associations, and sometimes class nodes.

Figure B-17 shows a hierarchy of objects that represent a set of teams in a single division in a soccer sports league. This diagram shows one configuration of objects at a specific point in the system. Object nodes only show instance attributes because methods are elements of the class definition. Also, an Object diagram does need not to show every associated object, it just needs to be representative.

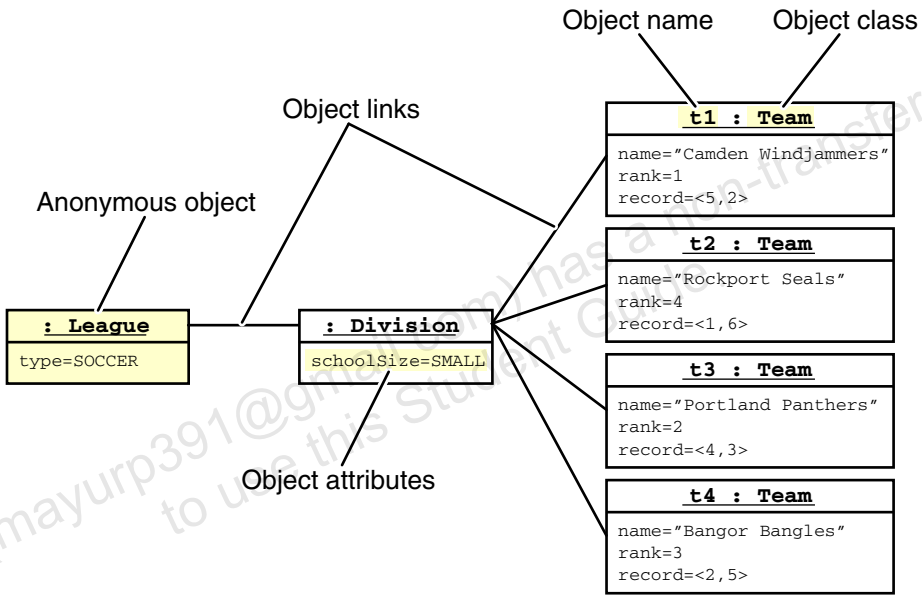


**Figure B-17** Object Diagram Example

Figure B-18 shows two objects, c1 and c2, with their instance data. They refer to the class node for Count. The dependency arrow indicates that the object is an instance of the class Count. The objects do not include the counter attribute because it has class scope.

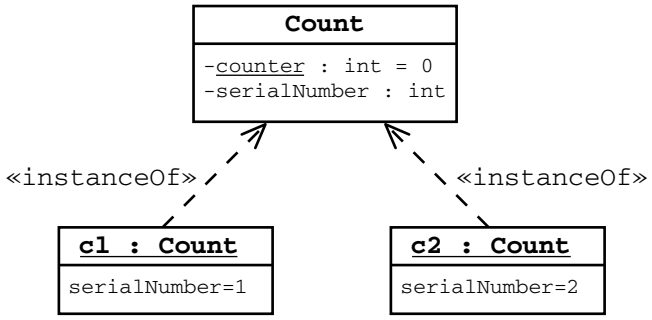


**Figure B-18** Object Diagram Example

# Collaboration Diagrams

A Collaboration diagram represents a particular behavior shared by several objects. These diagrams are composed of objects, their links, and the message exchanges that accomplish the behavior.

Figure B-19 shows a Collaboration diagram in which an actor initiates a login sequence within a web application using a servlet.

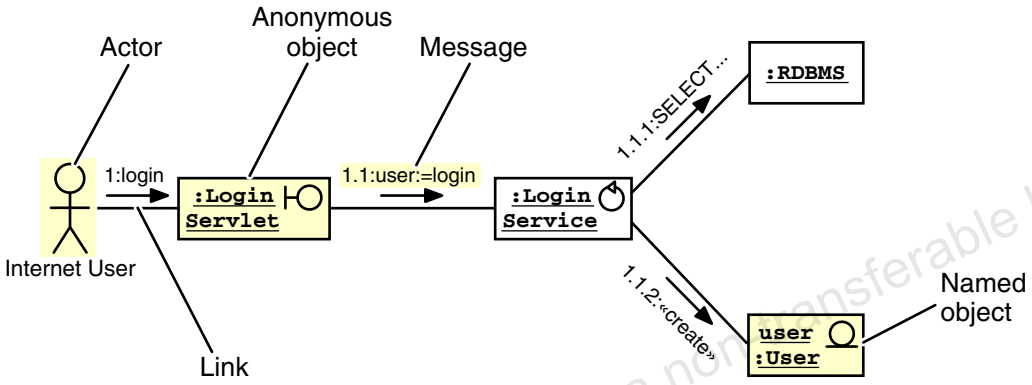


**Figure B-19** User-Driven Collaboration Diagram

The servlet uses an object of the `LoginService` class to perform the lookup of the username, verify the password, and create the `User` object. The links between objects show the dependencies and collaborations between these objects. Messages between objects are shown by the messages on the links. Messages are indicated by an arrow in the direction of the message and a text string declares the type of message. The text of this message string is unrestricted. Messages are also labeled with a sequence number so you can see the order of the message calls.

Figure B-20 shows a more elaborate Collaboration diagram. In this diagram, some client object initiates an action on a session bean. This session bean then performs two database modifications within a single transaction context.

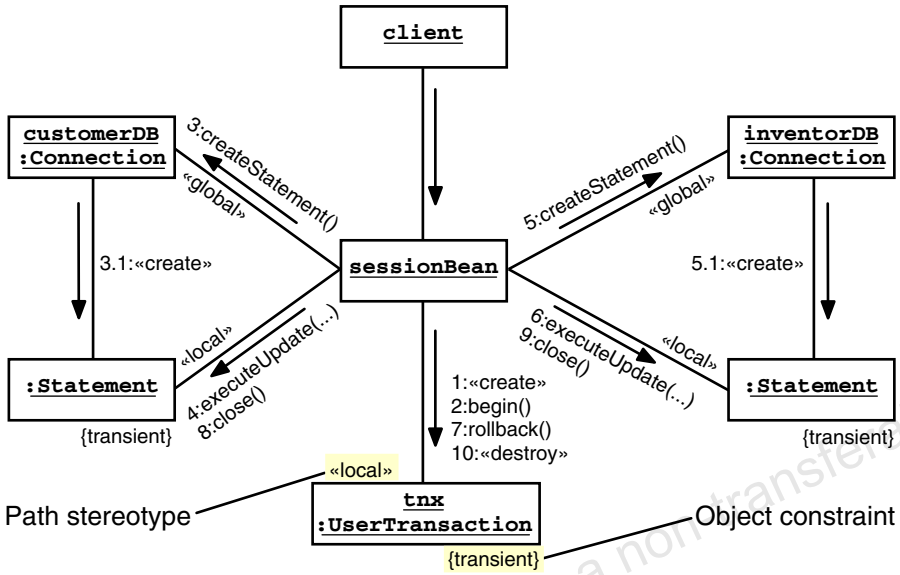


**Figure B-20** Collaboration Diagram

You can label the links with a stereotype to indicate if the object is global or local to the call sequence. In this example, the connection objects are global, and the statement and transaction objects are local.

You can also label objects with a constraint to indicate if the object is transient.

# Sequence Diagrams

A Sequence diagram represents a time sequence of messages exchanged between several objects to achieve a particular behavior. A Sequence diagram enables you to understand the flow of messages and events that occur in a certain process or collaboration. In fact, a Sequence diagram is just a time-ordered view of a Collaboration diagram, as shown in Figure B-19 on page B-24.

Figure B-21 shows a Sequence diagram in which an actor initiates a login sequence within a web application, which uses a servlet. This diagram is equivalent to the Collaboration diagram in Figure B-19 on page B-24. An important aspect of this type of diagram is that time is moving from top to bottom. The Sequence diagram shows the time-based interactions between a set of objects or roles. Roles can be named or anonymous, and usually have a class associated with them. Roles can also be actors.

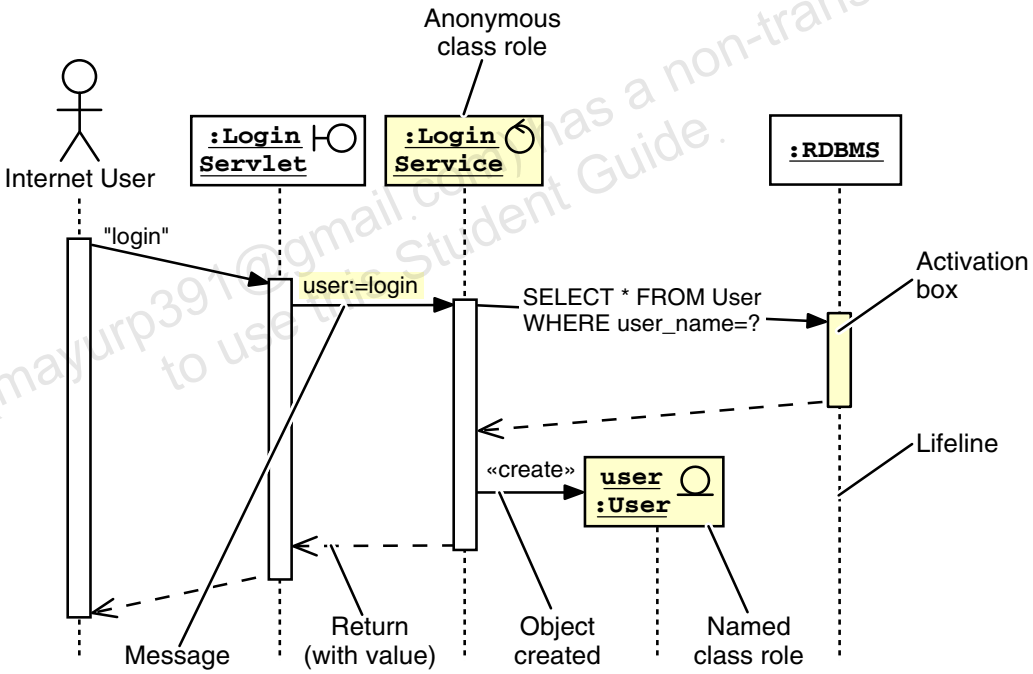


**Figure B-21** User-Driven Sequence Diagram Example

Notice the message arrows between the servlet and the service object. The arrow is perfectly horizontal, which indicates that the message is probably implemented by the local method call. Notice that the message arrow between the actor and the servlet is angled, which indicates that the message is sent between components on different machines. For example, an HTTP message from the user's web browser to the web container that handles the login servlet.

Figure B-22 shows a more elaborate Sequence diagram. This diagram is equivalent to the Collaboration diagram in Figure B-20 on page B-25.

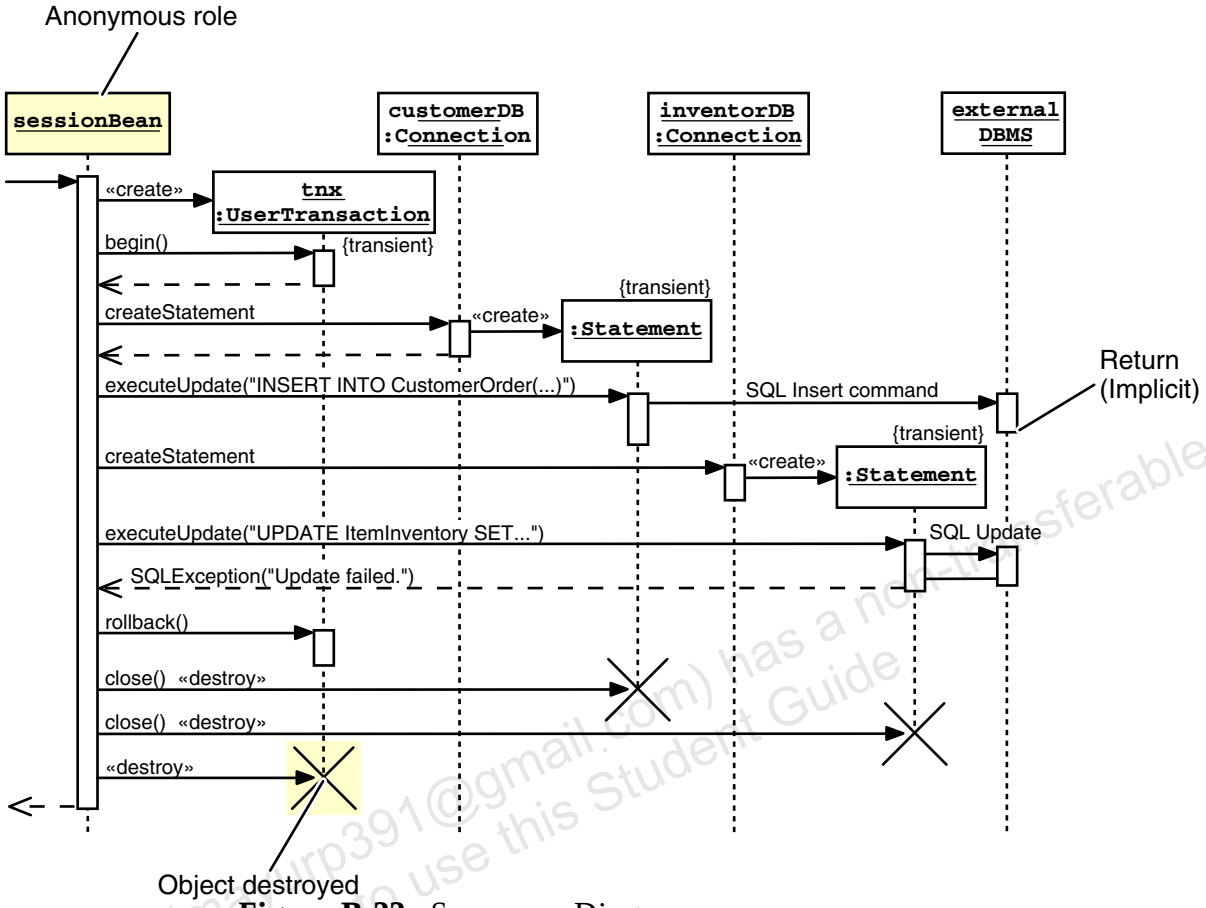


**Figure B-22** Sequence Diagram

This example shows a few more details about Sequence diagrams. First, the return arrow is not always important. A return arrow is implicit at the end of the activation bar. Also, Sequence diagrams can show the creation and destruction of objects explicitly. Every role has a lifeline that extends from the base of the object node vertically. Roles at the top of the diagram existed before the entry message (into the left-most role). Roles that have a message arrow pointing to the head of the role node with the «create» message are created during the execution of the sequence. The destruction of an object is shown with a large cross that terminates the role's lifeline.

**Note –** Sequence diagrams can also show asynchronous messages. This type of message uses a solid line with stick arrow head: ⟶.

# Statechart Diagrams

A Statechart diagram represents the states and responses of a class to external and internal triggers. You can also use a Statechart diagram to represent the life cycle of an object. The definition of an object state is dependent on the type of object and the level of depth you want to model.

**Note –** A Statechart diagram is recognized by several other names, including State diagram and State Transition diagram.

Figure B-23 shows an example Statechart diagram. Every Statechart diagram should have an initial state (the state of the object at its creation) and a final state. By definition, no state can transition into the initial state and the final state cannot transition to any other state.

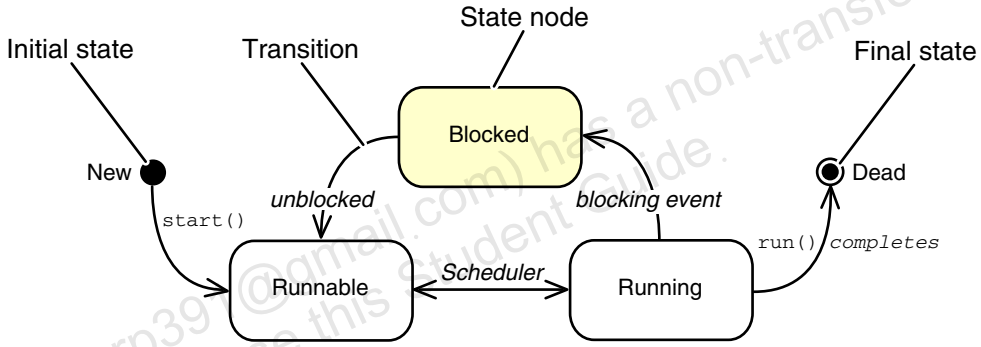


**Figure B-23** Statechart Diagram Example

There is no pre-defined way to implement a Statechart diagram. For complex behavior, you might consider using the State design pattern.

# Transitions

A transition has five elements:

- Source state – The state affected by the transition
- Event trigger – The event whose reception by the object in the source state makes the transition eligible to fire, providing its guard condition is satisfied
- Guard condition – A Boolean expression used to determine if the state transition should be made when the event trigger occurs
- Action – A computation or operation performed on the object that makes the state transition
- Target state – The state that is active after the completion of the transition

Figure B-24 shows a detailed transition.

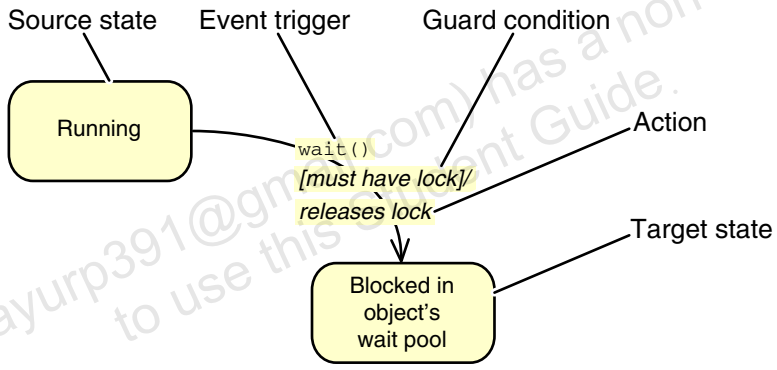


**Figure B-24** State Transition Example

# Activity Diagrams

An Activity diagram represents the activities or actions of a process without regard to the objects that perform these activities.

Figure B-25 shows the elements of an Activity diagram. An Activity diagram is similar to a flowchart. There are activities and transitions between them. Every Activity diagram starts with a single start state and ends in a single stop state.

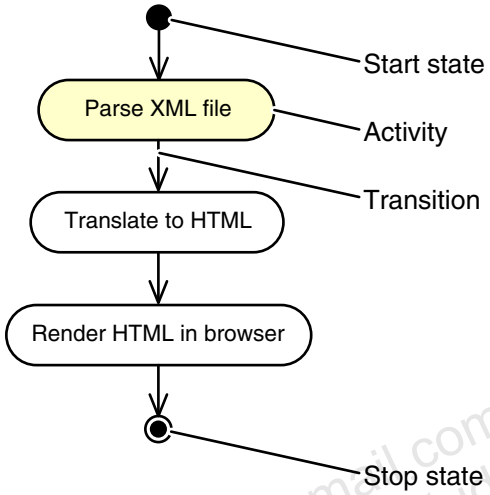


**Figure B-25** Activities and Transitions

Figure B-26 demonstrates branching and looping in Activity diagrams. The diagram models the higher level activity of `Verify availability` of products in a purchase order. The top-level branch node forms a simple while loop. The guard on the transition below the branch is `true` if there are more products in the order to be processed. The `else` transition halts this activity.

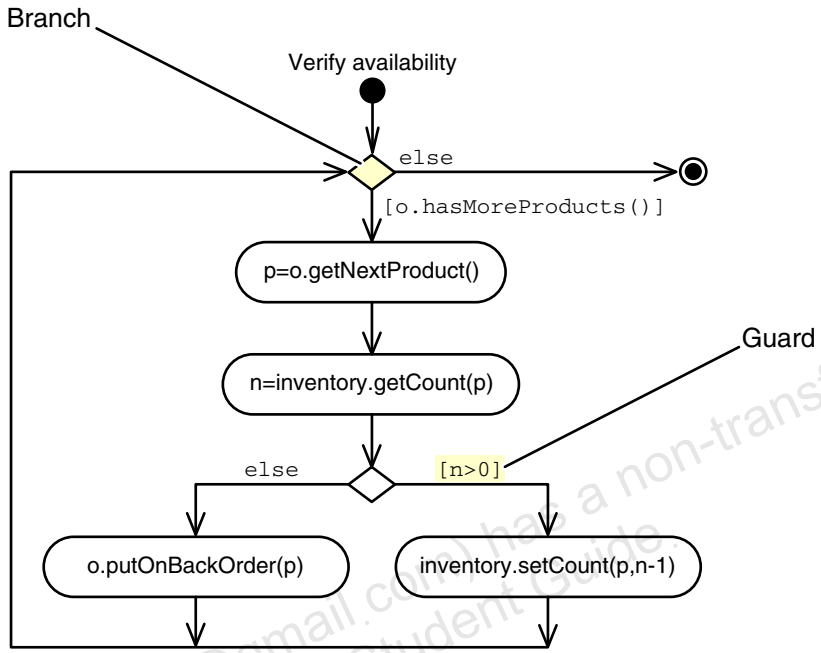


**Figure B-26** Branching and Looping

Figure B-27 shows a richer Activity diagram.

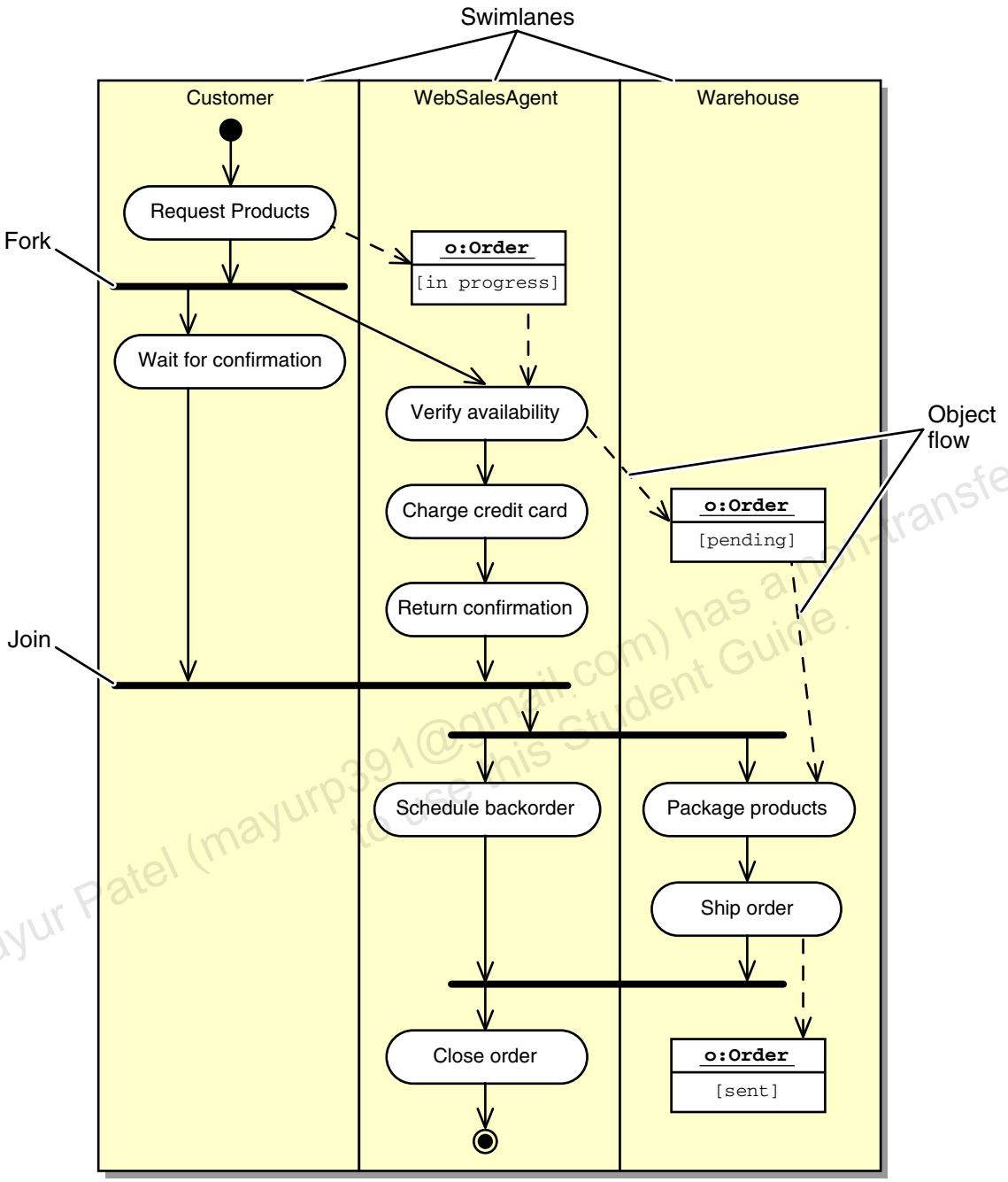


**Figure B-27** Activity Diagram Example

In the example in Figure B-27 on page B-32, swim lanes are used to isolate the actor of a given set of activities. These actors might include humans, systems, or organization entities. This diagram also demonstrates the ability to model concurrent activities. For example, the Customer initiates the purchase of one or more products on the company's web site. The Customer then waits as the WebSalesAgent software begins to process the purchase order. The fork bar splits a single transition into two or more transitions. The corresponding join bar must contain the same number of inbound transitions.

# Component Diagrams

A Component diagram represents the organization and dependencies among software implementation components.

Figure B-28 shows three types of icons that can represent software components. Example 1 is a generic icon. Example 2 is an icon that represents a source file. Example 3 is an icon that represents a file that contains executable (or object) code.

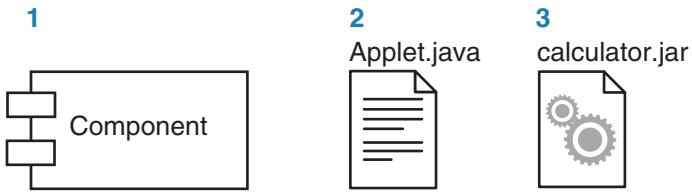


**Figure B-28** Component Nodes Example

Figure B-29 shows the dependencies of packaging an HTML page that contains an applet. The HTML page depends on the JAR file, which is constructed from a set of class files. The Class files are compiled from the corresponding source files. You can use a tagged value to indicate the source control version numbers on the source files.

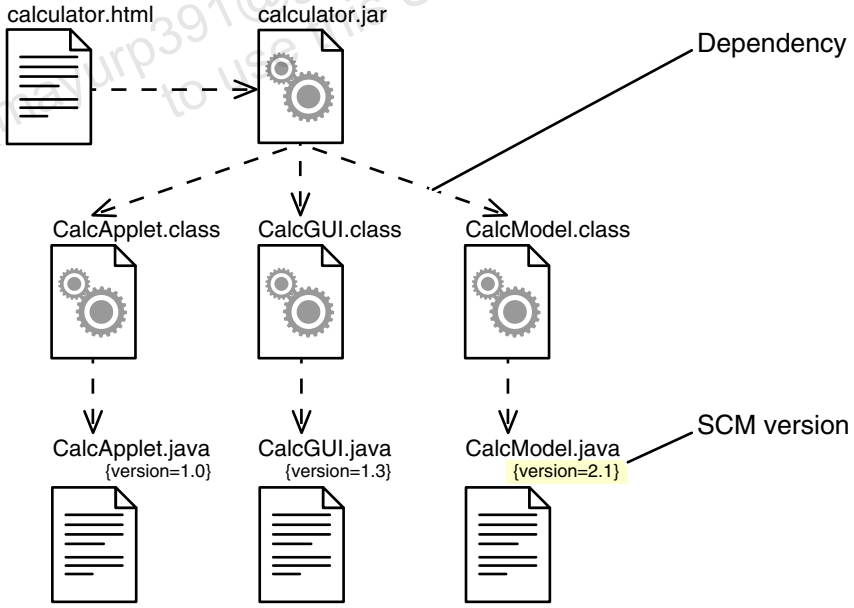


**Figure B-29** Component Diagram Example

**Note –** SCM is Source Control Management.

Figure B-30 shows another Component diagram. In this diagram, several components have an interface connector. The component attached to the connector implements the named interface. The component that has an arrow pointing to the connector depends on the fact that component realizes that interface.

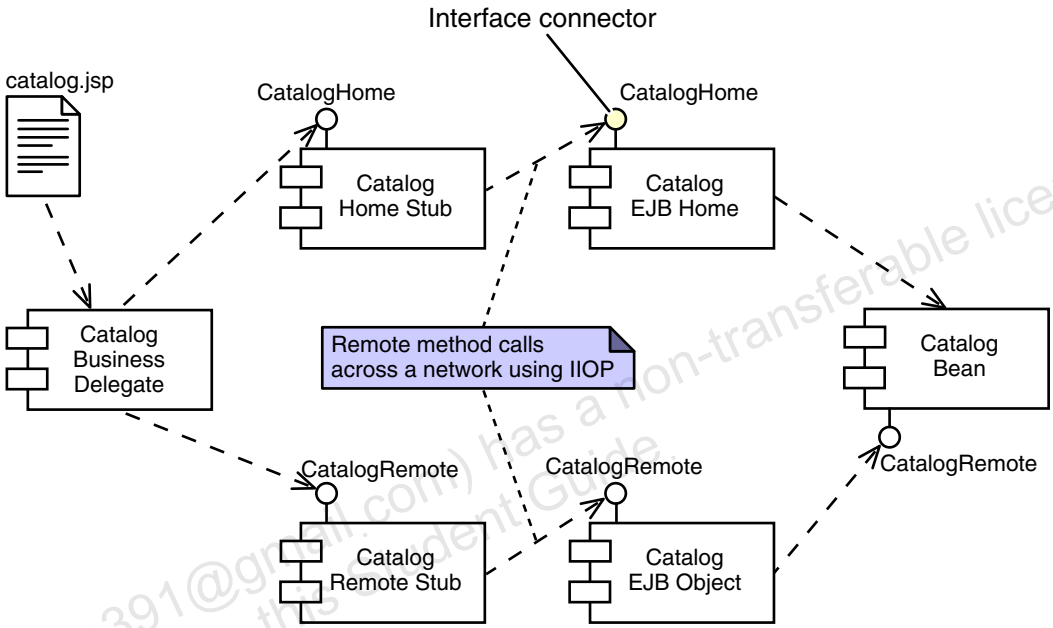


**Figure B-30** Component Diagram With Interfaces

In this J2EE technology example, the web tier includes a catalog JSP, which uses a catalog Business Delegate JavaBeans component to communicate to the Enterprise JavaBeans™ (EJB™) technology tier. Every enterprise bean must include two interfaces. The home interface enables the client to create new enterprise beans on the EJB server. The remote interface enables the client to call the business logic methods on the (remote) enterprise bean. The business delegate communicates with the catalog bean through local stub objects that implement the proper home and remote interfaces. These objects communicate over a network using the Internet Inter-ORB protocol with remote *skeletons*. In EJB technology terms, these objects are called `EJBHome` and `EJBObject`. These objects communicate directly with the catalog bean that implements the true business logic.

# Deployment Diagrams

A Deployment diagram represents the network of processing resource elements and the configuration of software components on each physical element.

A Deployment diagram is composed of hardware nodes, software components, software dependencies, and communication relationships. Figure B-31 shows an example in which the client machine communicates with a web server using HTTP over TCP/IP.

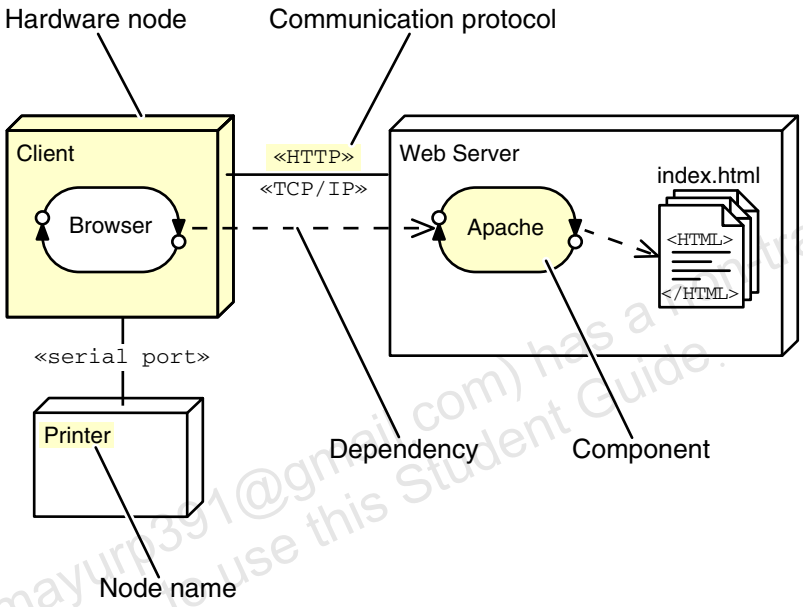


**Figure B-31** Deployment Diagram Example

The client is running a web browser, which is communicating with an Apache web server. Therefore, the browser component depends on the Apache component. Similarly, the Apache application depends on the HTML files that it serves. The client machine is also connected to local printer using a parallel port.

You can use a Deployment diagram to show how the logical tiers of an application architecture are configured into a physical network.