Practical UML™: A Hands-On Introduction for Developers

by Randy Miller

Borland, 2003

Abstract: This tutorial provides a quick introduction to the Unified Modeling Language™

The heart of object-oriented problem solving is the construction of a model. The model abstracts the essential details of the underlying problem from its usually complicated real world. Several modeling tools are wrapped under the heading of the **UML**™, which stands for Unified Modeling Language™. The purpose of this course is to present important highlights of the UML.

At the center of the UML are its nine kinds of modeling diagrams, which we describe here.

- Use case diagrams
- Class diagrams
- Object diagrams
- Sequence diagrams
- Collaboration diagrams
- Statechart diagrams
- Activity diagrams
- Component diagrams
- Deployment diagrams

Why is UML important?

Let's look at this question from the point of view of the construction trade. Architects design buildings. Builders use the designs to create buildings. The more complicated the building, the more critical the communication between architect and builder. Blueprints are the standard graphical language that both architects and builders must learn as part of their trade.

Writing software is not unlike constructing a building. The more complicated the underlying system, the more critical the communication among everyone involved in creating and deploying the software. In the past decade, the UML has emerged as the software blueprint language for analysts, designers, and programmers alike. It is now part of the software trade. The UML gives everyone from business analyst to designer to programmer a common vocabulary to talk about software design.

The UML is applicable to object-oriented problem solving. Anyone interested in learning UML must be familiar with the underlying tenet of object-oriented problem solving -- it all begins with the construction of a model. A **model** is an abstraction of the underlying problem. The **domain** is the actual world from which the problem comes.

Models consist of **objects** that interact by sending each other **messages**. Think of an object as "alive." Objects have things they know (**attributes**) and things they can do (**behaviors** or **operations**). The values of an object's attributes determine its **state**.

Classes are the "blueprints" for objects. A class wraps attributes (data) and behaviors (methods or functions) into a single distinct entity. Objects are **instances** of classes.

Use case diagrams

Use case diagrams describe what a system does from the standpoint of an external observer. The emphasis is on *what* a system does rather than *how*.

Use case diagrams are closely connected to scenarios. A **scenario** is an example of what happens when someone interacts with the system. Here is a scenario for a medical clinic.

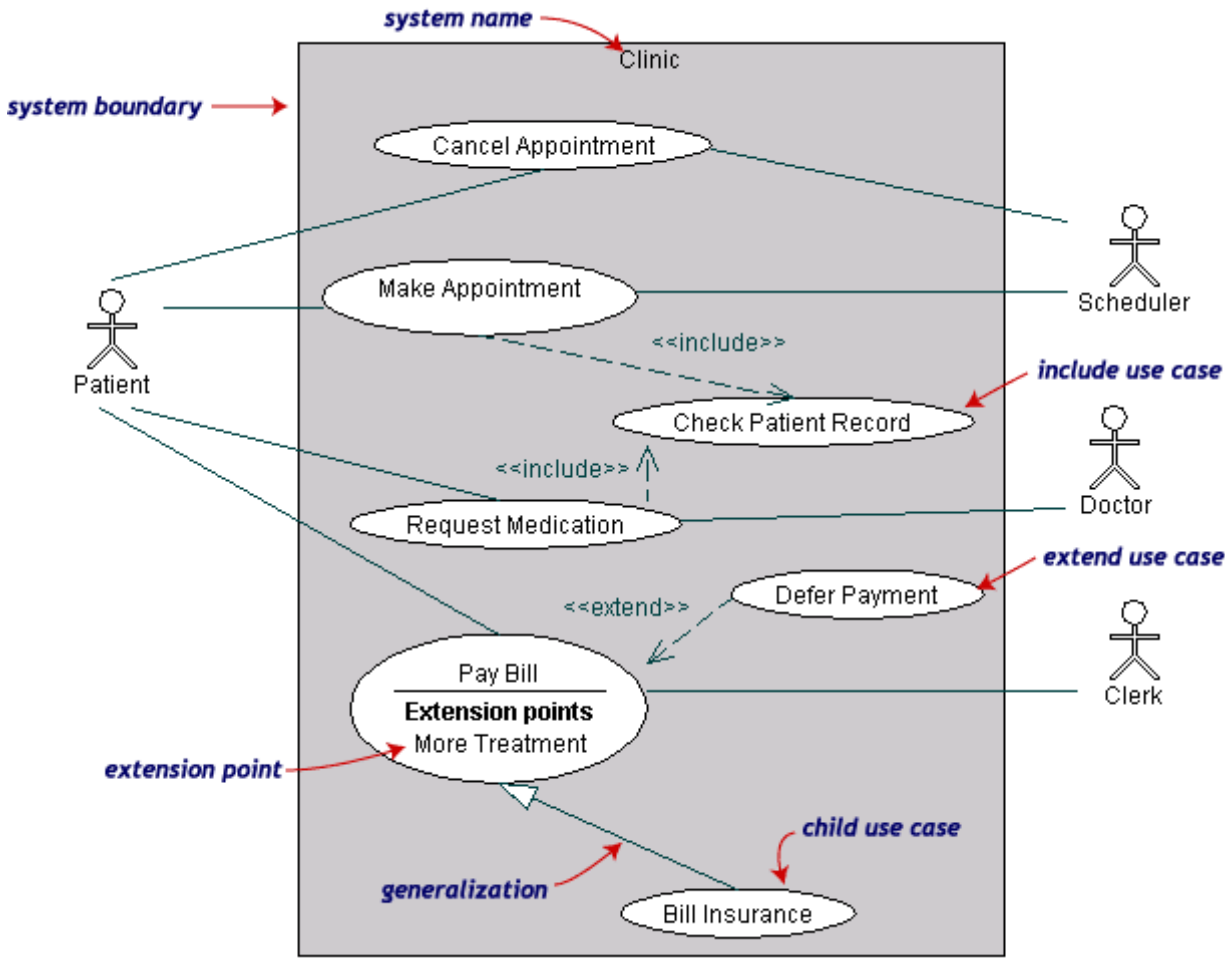
> "A patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot. "

A **use case** is a summary of scenarios for a single task or goal. An **actor** is who or what initiates the events involved in that task. Actors

are simply roles that people or objects play. The picture below is a **Make Appointment** use case for the medical clinic. The actor is a **Patient**. The connection between actor and use case is a **communication association** (or **communication** for short).

Use case diagrams give an outsider's view of a system. Every use case diagram has actors, use cases, and communications. A simple use case diagram can be expanded with additional features to display more information.

Medical clinic diagram, expanded



The following use case diagram expands the original medical clinic diagram with additional features.

A **system boundary** rectangle separates the clinic system from the external actors.

A use case **generalization** shows that one use case is simply a special kind of another. **Pay Bill** is a parent use case and **Bill Insurance** is

the child. A child can be substituted for its parent whenever necessary. Generalization appears as a line with a triangular arrow head toward the parent use case.

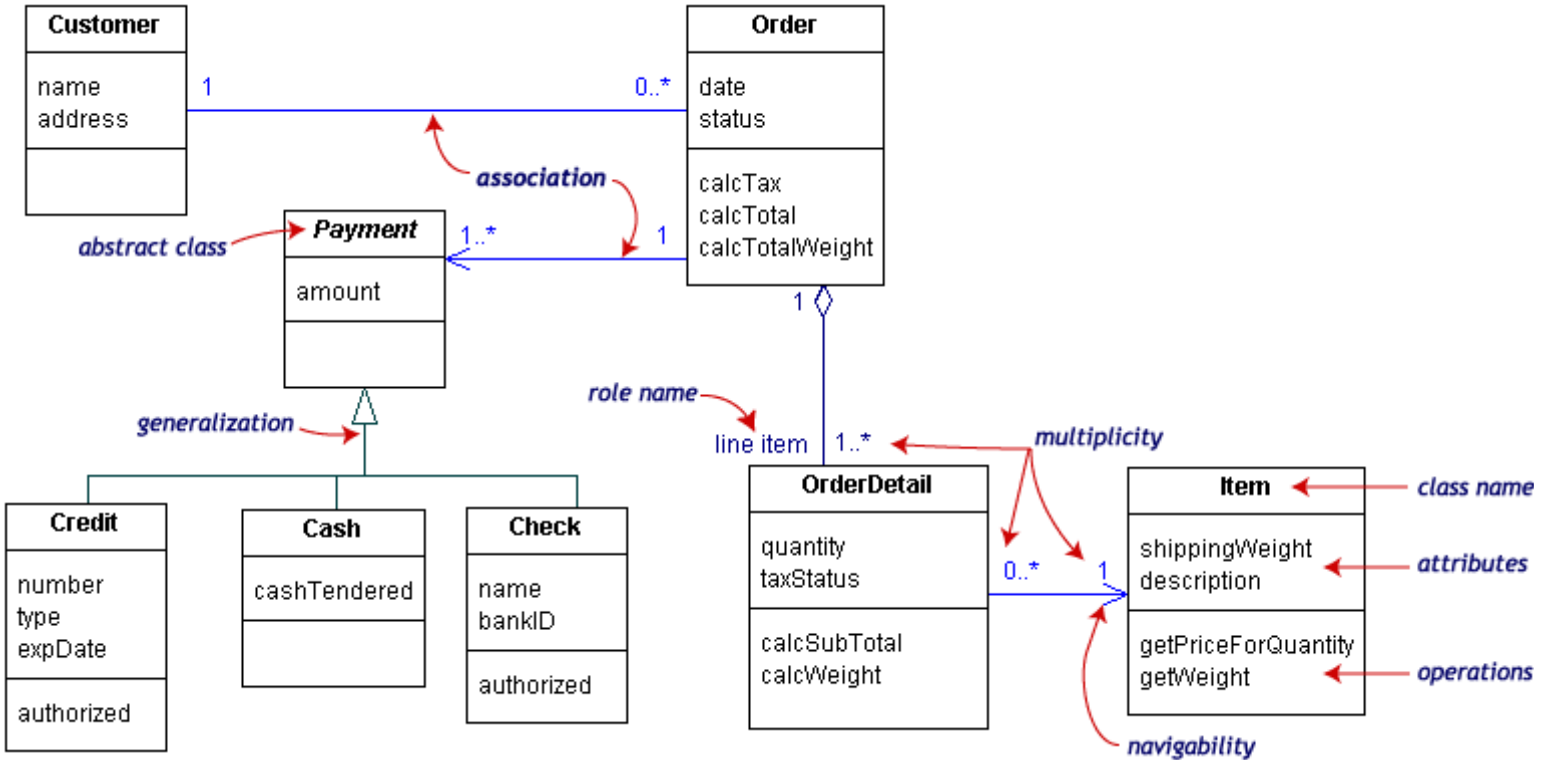
Include relationships factor use cases into additional ones. Includes are especially helpful when the same use case can be factored out of two different use cases. Both **Make Appointment** and **Request Medication** include **Check Patient Record** as a subtask. In the diagram, include notation is a dotted line beginning at base use case ending with an arrows pointing to the include use case. The dotted line is labeled <<include>>.

An **extend** relationship indicates that one use case is a variation of another. Extend notation is a dotted line, labeled <<extend>>, and with an arrow toward the base case. The **extension point**, which determines when the extended case is appropriate, is written inside the base case.

Class diagrams

A Class diagram gives 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 below models a customer order from a retail catalog. The central class is the **Order**. Associated with it are the **Customer** making the purchase and the **Payment**. A **Payment** is one of three kinds: **Cash**, **Check**, or **Credit**. The order contains **OrderDetails** (line items), each with its associated **Item**.



UML class notation is a rectangle divided into three parts: class name, attributes, and operations. Names of abstract classes, such as ***Payment***, are in italics. Relationships between classes are the connecting links.

Our class diagram has three kinds of relationships.

- **association** -- a relationship between instances of the two classes. There is an association between two classes if an instance of one class must know about the other in order to perform its work. In a diagram, an association is a link connecting two classes.

- **aggregation** -- an association in which one class belongs to a collection. An aggregation has a diamond end pointing to the part containing the whole. In our diagram, **Order** has a collection of **OrderDetails**.

- **generalization** -- an inheritance link indicating one class is a superclass of the other. A generalization has a triangle pointing to the superclass. ***Payment*** is a superclass of **Cash**, **Check**, and **Credit**.

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

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 bi-directional.

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 are single numbers or ranges of numbers. In our example, there can be only one **Customer** for each **Order**, but a **Customer** can have any number of **Orders**.

This table gives the most common multiplicities.

Multiplicities	Meaning
0..1	zero or one instance. The notation ***n . . m*** indicates ***n*** to ***m*** 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.

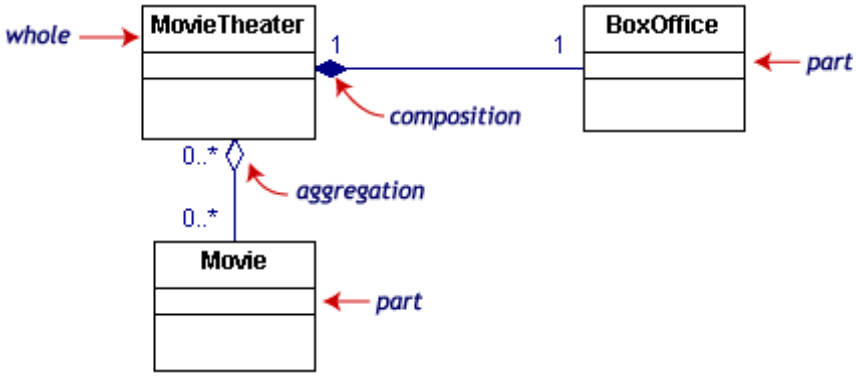
All class diagrams have classes, links, and multiplicities. But a class diagram can show even more information.

Composition and aggregation

Associations in which an object is part of a whole are aggregations. **Composition** is a strong association in which the part can belong to

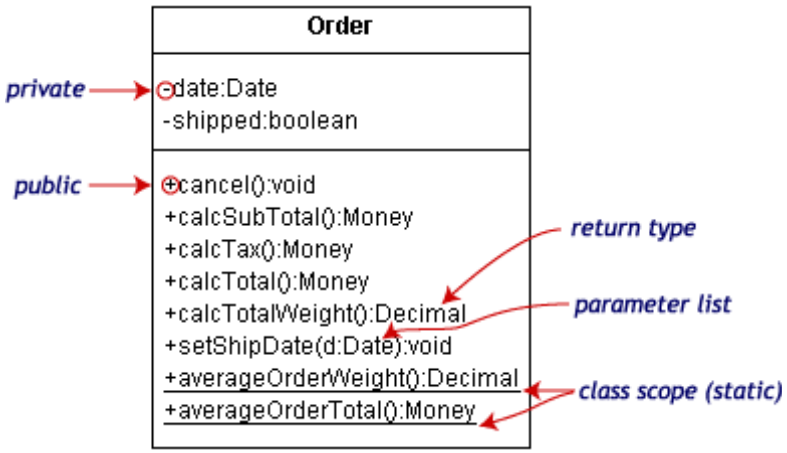
only one whole -- the part cannot exist without the whole. Composition is denoted by a filled diamond at the whole end.

This diagram shows that a **BoxOffice** belongs to exactly one **MovieTheater**. Destroy the **MovieTheater** and the **BoxOffice** goes away! The collection of **Movies** is not so closely bound to the **MovieTheater**.



Class information: visibility and scope

The class notation is a 3-piece rectangle with the class name, attributes, and operations. Attributes and operations can be labeled according to access and scope. Here is a new, expanded **Order** class.



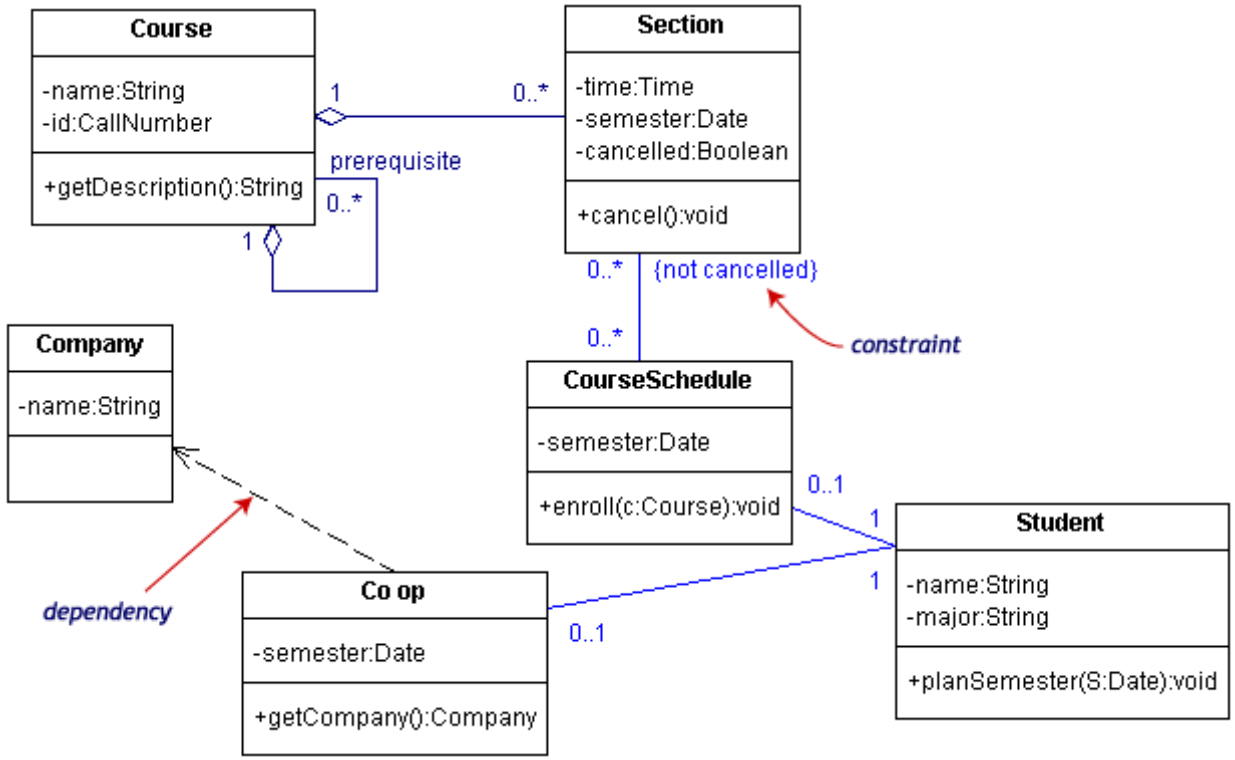
The illustration uses the following UML™ conventions.

- Static members are underlined. 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 appear in front of each member.

Symbol	Access
+	public
-	private
#	protected

Dependencies and constraints

A **dependency** is a relation between two classes in which a change in one may force changes in the other. Dependencies are drawn as dotted lines. In the class diagram below, **Co_op** depends on **Company**. If you decide to modify **Company**, you may have to change **Co_op** too.



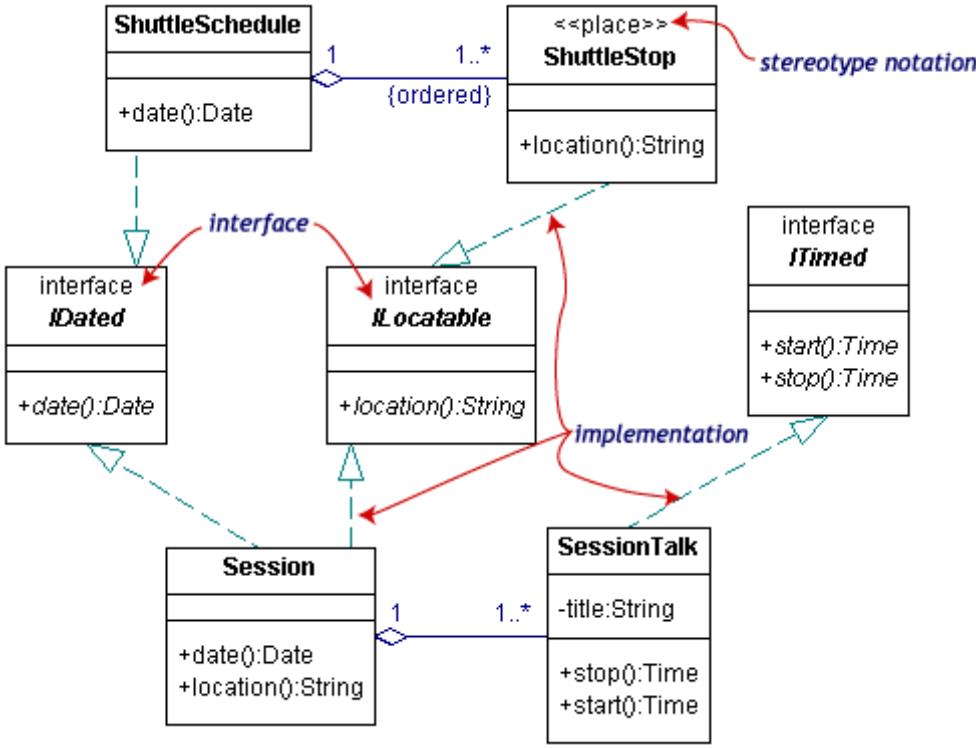
A **constraint** is a condition that every implementation of the design must satisfy. Constraints are written in curly braces { }. The

constraint on our diagram indicates that a **Section** can be part of a **CourseSchedule** only if it is not canceled.

Interfaces and stereotypes

An **interface** is a set of operation signatures. In C++, interfaces are implemented as abstract classes with only pure virtual members. in Java, they're implemented directly.

The class diagram below is a model of a professional conference. The classes of interest to the conference are **SessionTalk**, which is a single presentation, and **Session**, which is a one-day collection of related **SessionTalk**s. The **ShuttleSchedule** with its list of **ShuttleStop**s is important to the attendees staying at remote hotels. The diagram has one constraint, that the **ShuttleStop**s are ordered.



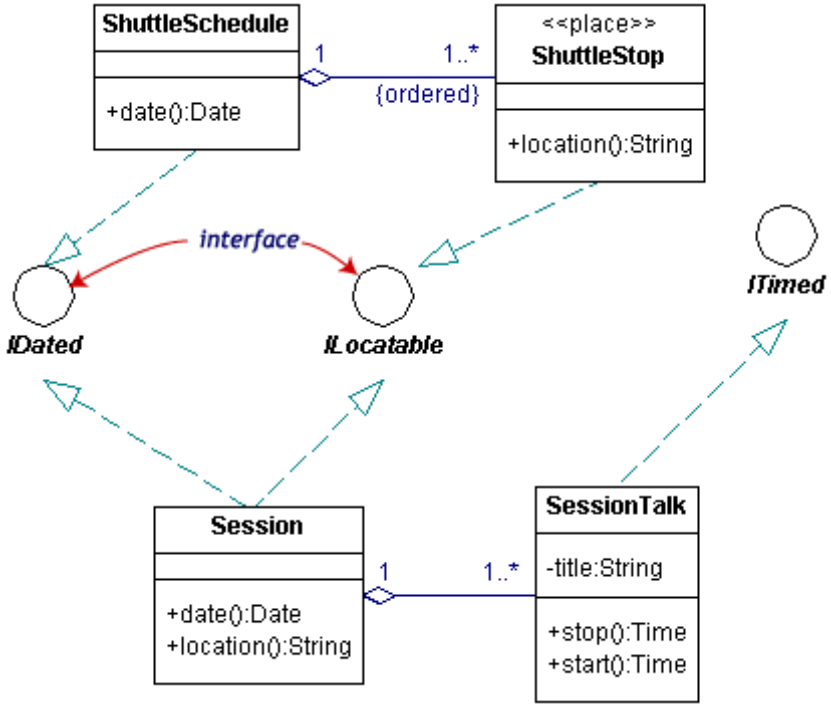
There are three interfaces in the diagram: ***IDated***, ***ILocatable***, and ***ITimed***. The names of interfaces typically begin with the letter **I**. Interface names along with their (abstract) operations are written in italics.

A class such as **ShuttleStop**, with operations matching those in an interface, such as **ILocatable**, is an **implementation** (or **realization**) of the interface.

The **ShuttleStop** class node has the **stereotype** << place>>. Stereotypes, which provide a way of extending UML, are new kinds of model elements created from existing kinds. A stereotype name is written above the class name. Ordinary stereotype names are enclosed in guillemets, which look like pairs of angle-braces. An interface is a special kind of stereotype.

There are two acceptable notations for interfaces in the UML. The first is illustrated above. The second uses the lollipop or circle

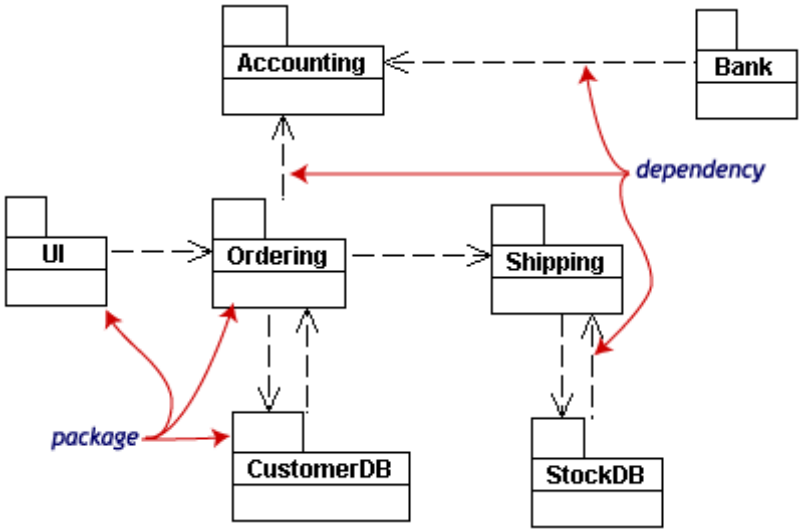
notation.



In circle notation, the interfaces are circles with lines connected to the implementing classes. Since it is more compact but leaves out some detail, the lollipop notation simplifies the original diagram.

Packages and object diagrams

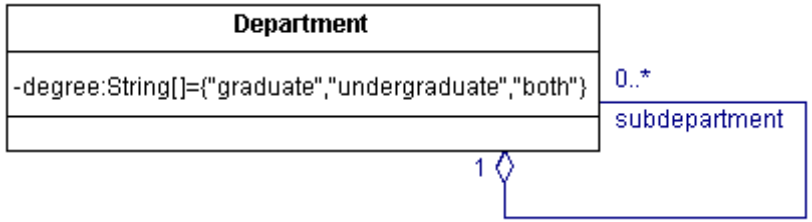
To simplify complex class diagrams, you can group classes into **packages**. A package is a collection of logically related UML elements. The diagram below is a business model in which the classes are grouped into packages.



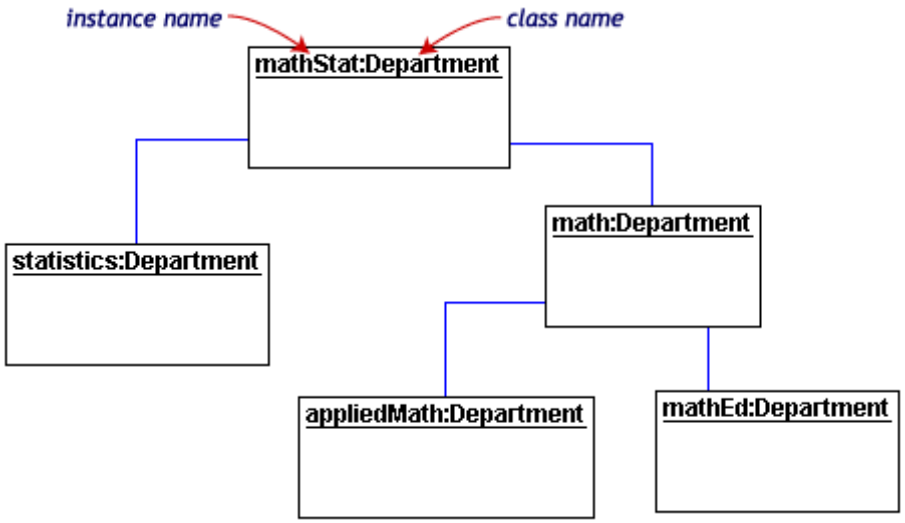
Packages appear as rectangles with small tabs at the top. The package name is on the tab or inside the rectangle. The dotted arrows are **dependencies**. One package depends on another if changes in the other could possibly force changes in the first.

Object diagrams show instances instead of classes. They are useful for explaining small pieces with complicated relationships, especially recursive relationships.

This small class diagram shows that a university **Department** can contain lots of other **Departments**.



The object diagram below instantiates the class diagram, replacing it by a concrete example.



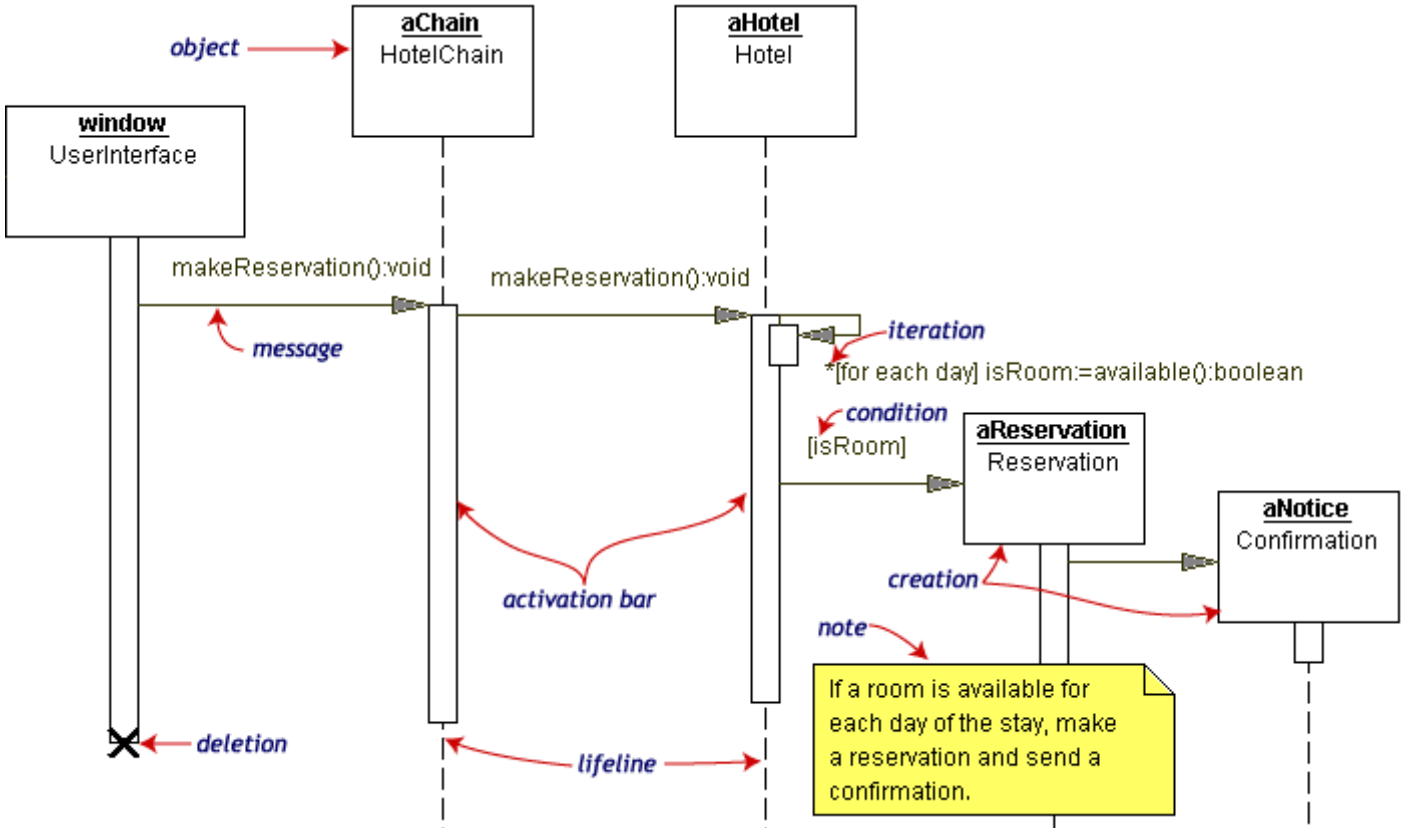
Each rectangle in the object diagram corresponds to a single instance. Instance names are underlined in UML diagrams. Class or instance names may be omitted from object diagrams as long as the diagram meaning is still clear.

Sequence diagrams

Class and object diagrams are static model views. **Interaction diagrams** are dynamic. They describe how objects collaborate.

A **sequence diagram** is an interaction diagram that details how operations are carried out -- what messages are sent and when. Sequence diagrams are organized according to time. The time progresses as you go 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.

Below is a sequence diagram for making a hotel reservation. The object initiating the sequence of messages is **a Reservation window**.



The **Reservation window** sends a `makeReservation()` message to a **HotelChain**. The **HotelChain** then sends a `makeReservation()` message to a **Hotel**. If the **Hotel** has available rooms, then it makes a **Reservation** and a **Confirmation**.

Each vertical dotted line 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.

In our diagram, the **Hotel** issues a **self call** to determine if a room is available. If so, then the **Hotel** creates a **Reservation** and a **Confirmation**. The asterisk on the self call means **iteration** (to make sure there is available room for each day of the stay in the hotel).

The expression in square brackets, [ ], is a **condition**.

The diagram has a clarifying **note**, which is text inside a dog-eared rectangle. Notes can be put into any kind of UML diagram.

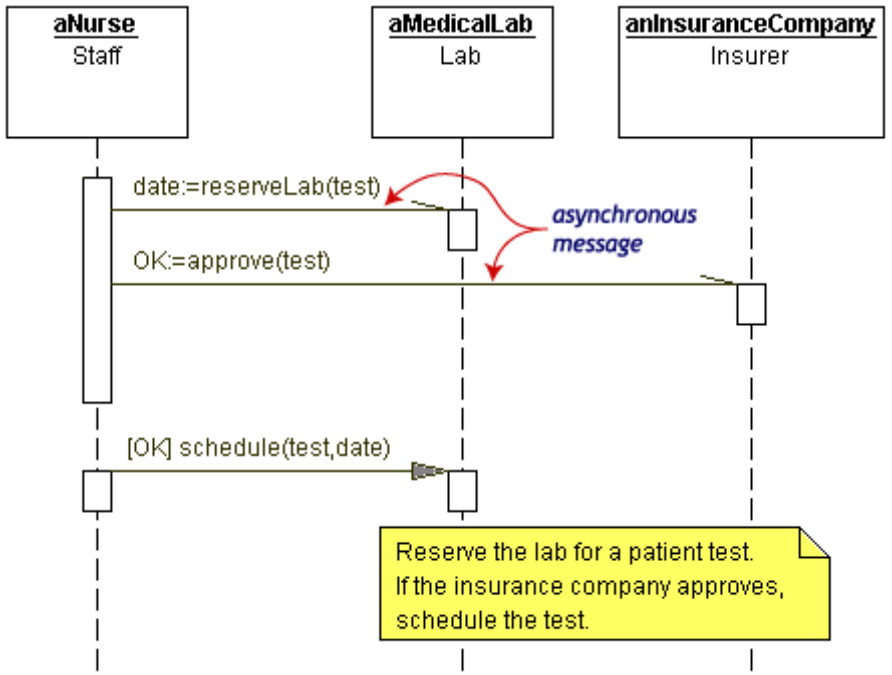
Sequence diagrams, collaboration diagrams, activity diagrams, and statechart diagrams give dynamic views of a model. They let you look inside to the mechanism for action inside the model. Sequence diagrams and collaboration diagrams focus on the messages involved in completing a single process. Statechart diagrams focus attention on a single object. Activity diagrams focus on the flow of activities in performing a single task.

In this page, we will see how to how asynchronous or concurrent activities are modeled in sequence diagrams and state diagrams.

Sequence diagrams with asynchronous messages

A message is **asynchronous** if it allows its sender to send additional messages while the original is being processed. The timing of an asynchronous message is independent of the timing of the intervening messages.

The following sequence diagram illustrates the action of a nurse requesting a diagnostic test at a medical lab. There are two asynchronous messages from the **Nurse**: 1) ask the **MedicalLab** to reserve a date for the test and 2) ask the **InsuranceCompany** to approve the test. The order in which these messages are sent or completed is irrelevant. If the **InsuranceCompany** approves the test, then the **Nurse** will schedule the test on the date supplied by the **MedicalLab**.



The UML™ uses the following message conventions.

simple message which may be synchronous or asynchronous

simple message return (optional)

a synchronous message

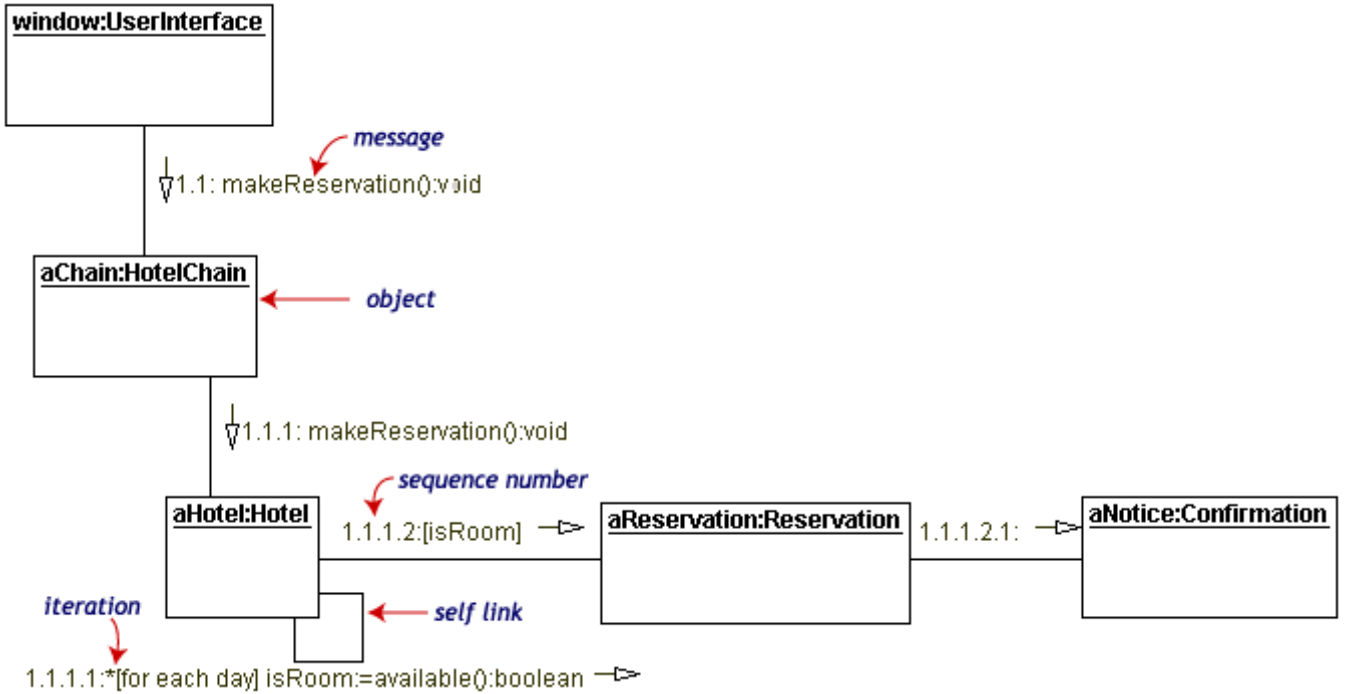
or

or

an asynchronous message

Collaboration diagrams

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 (:).

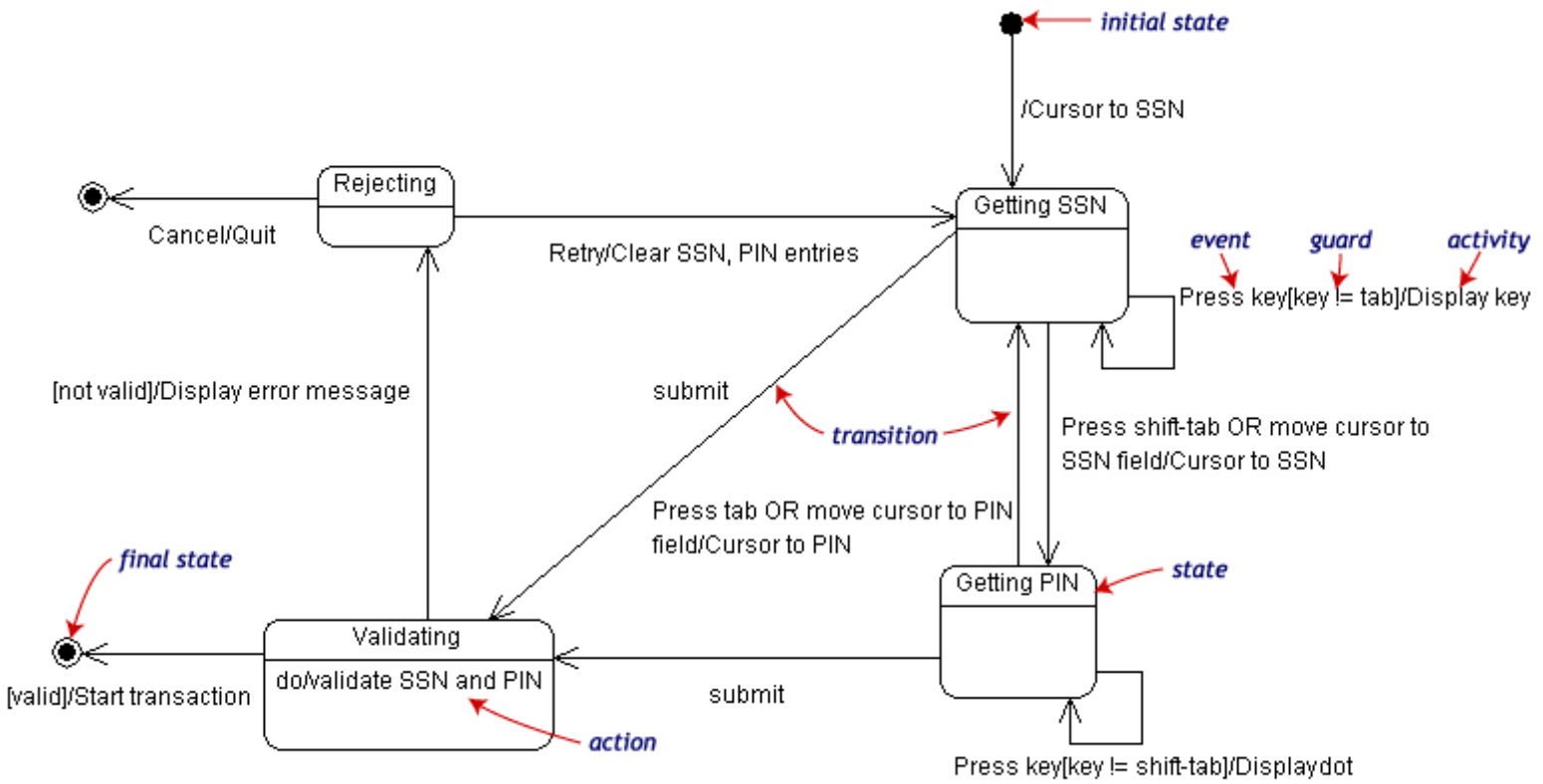
Each message in a collaboration diagram has a **sequence number**. The top-level message is numbered 1. Messages at the same level (sent during the same call) have the same decimal prefix but suffixes of 1, 2, etc. according to when they occur.

Statechart diagrams

Objects have behaviors and state. The state of an object depends on its current activity or condition. A **statechart diagram** shows the possible states of the object and the transitions that cause a change in state.

Our example diagram models the login part of an online banking system. Logging in consists of entering a valid social security number and personal id number, then submitting the information for validation.

Logging in can be factored into four non-overlapping states: **Getting SSN**, **Getting PIN**, **Validating**, and **Rejecting**. From each state comes a complete set of **transitions** that determine the subsequent state.



States are rounded rectangles. Transitions are arrows from one state to another. Events or conditions that trigger transitions are written beside the arrows. Our diagram has two self-transition, one on **Getting SSN** and another on **Getting PIN**.

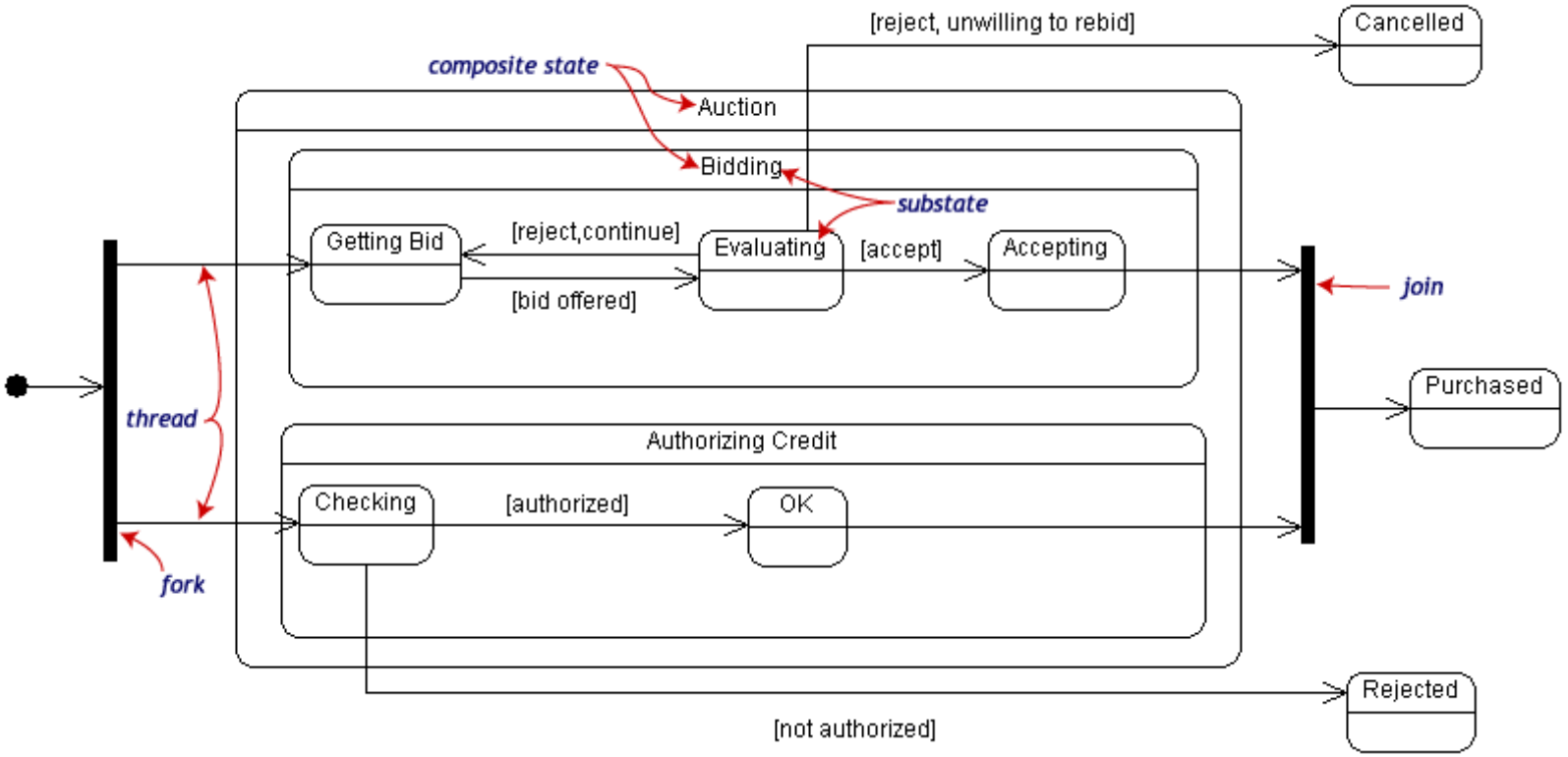
The initial state (black circle) is a dummy to start the action. Final states are also dummy states that terminate the action.

The action that occurs as a result of an event or condition is expressed in the form `/action`. While in its **Validating** state, the object does not wait for an outside event to trigger a transition. Instead, it performs an activity. The result of that activity determines its subsequent state.

Concurrency and asynchronization in statechart diagrams

States in statechart diagrams can be nested. Related states can be grouped together into a single **composite state**. Nesting states is necessary when an activity involves concurrent or asynchronous subactivities.

The following statechart diagram models an auction with two concurrent threads leading into two substates of the composite state **Auction**: **Bidding** and **Authorizing Credit**. **Bidding** itself is a composite state with three substates. **Authorizing Credit** has two substates.



Entering the **Auction** requires a fork at the start into two separate threads. Unless there is an abnormal exit (**Cancelled** or **Rejected**), the exit from the **Auction** composite state occurs when both substates have exited.

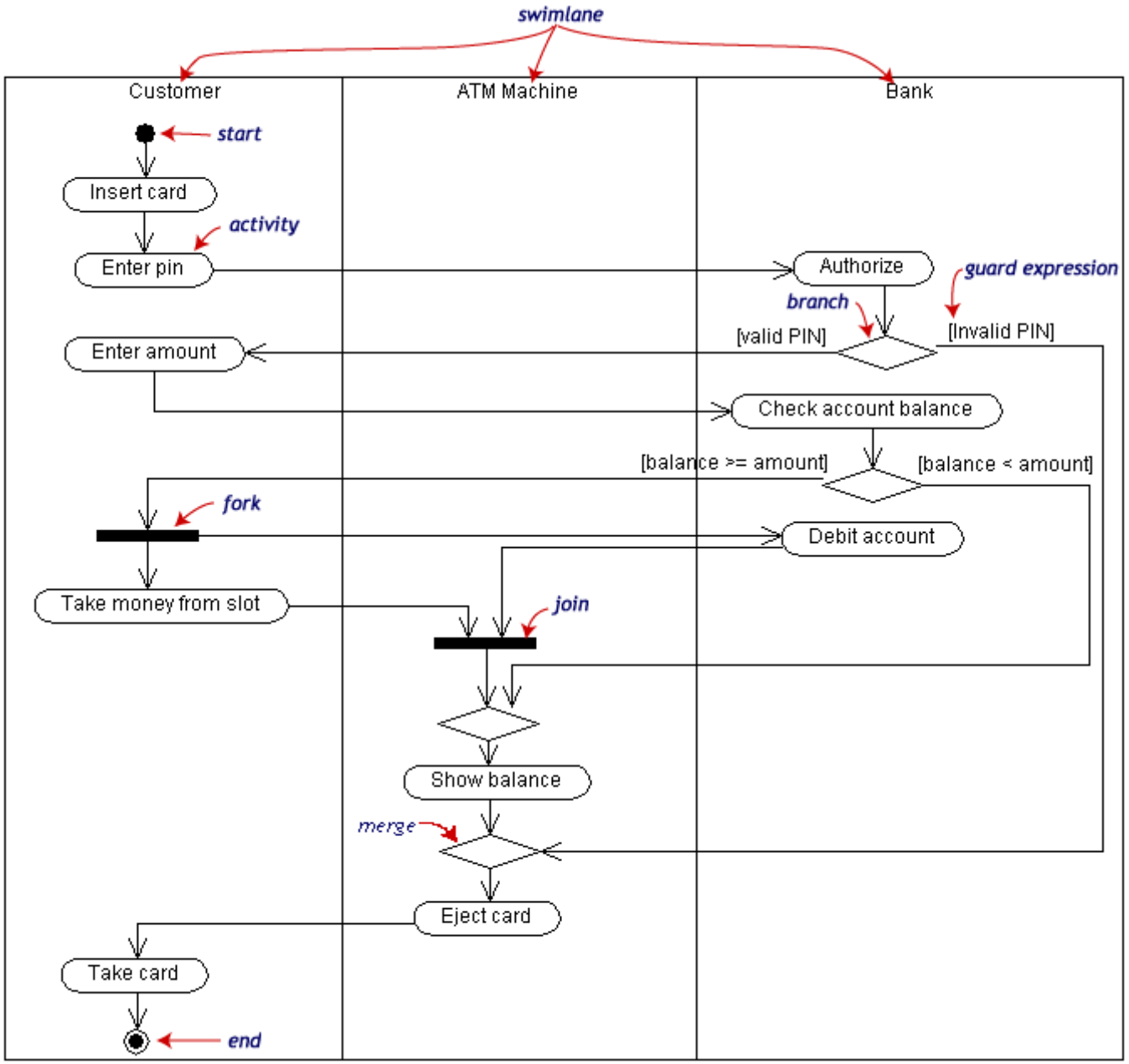
Activity diagrams

An **activity diagram** is essentially a fancy flowchart. Activity diagrams and statechart diagrams are related. While a statechart diagram focuses attention on an object undergoing a process (or on a process as an object), an activity diagram focuses on the flow of activities involved in a single process. The activity diagram shows the how those activities depend on one another.

For our example, we used the following process.

> "Withdraw money from a bank account through an ATM."

The three involved classes (people, etc.) of the activity are **Customer**, **ATM**, and **Bank**. The process begins at the black start circle at the top and ends at the concentric white/black stop circles at the bottom. The activities are rounded rectangles.


swimlane
Customer
ATM Machine
Bank
start
Insert card
activity
Enter pin
Authorize
guard expression
branch
[valid PIN]
[Invalid PIN]
Enter amount
Check account balance
[balance >= amount]
[balance < amount]
fork
Debit account
Take money from slot
join
Show balance
merge
Eject card
Take card
end

Activity diagrams can be divided into object **swimlanes** that determine which object is responsible for which activity. A single **transition** comes out of each activity, connecting it to the next activity.

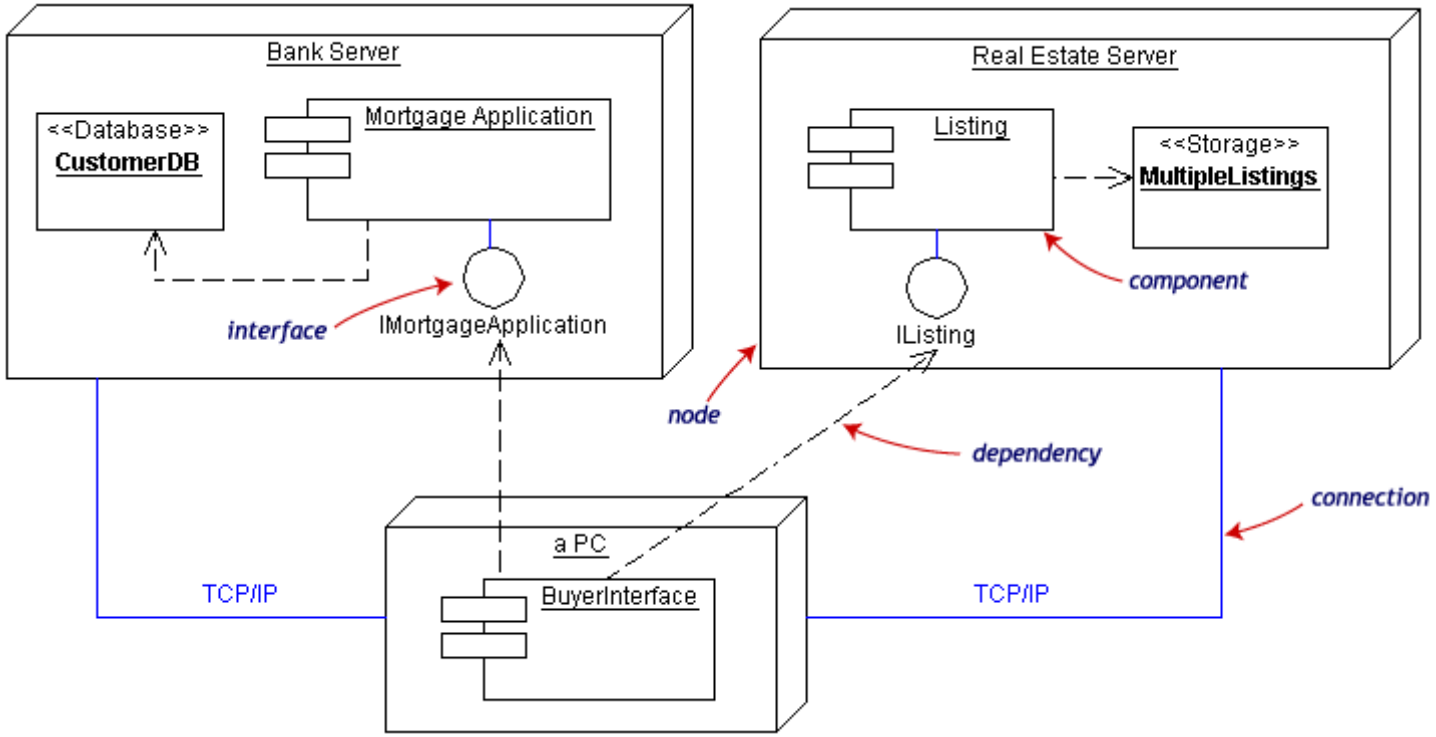
A transition may **branch** into two or more mutually exclusive transitions. **Guard expressions** (inside [ ]) label the transitions coming out of a branch. A branch and its subsequent **merge** marking the end of the branch appear in the diagram as hollow diamonds.

A transition may **fork** into two or more parallel activities. The fork and the subsequent **join** of the threads coming out of the fork appear in the diagram as solid bars.

Component and deployment diagrams

A **component** is a code module. Component diagrams are physical analogs of class diagram. **Deployment diagrams** show the physical configurations of software and hardware.

The following deployment diagram shows the relationships among software and hardware components involved in real estate transactions.



The physical hardware is made up of **nodes**. Each component belongs on a node. Components are shown as rectangles with two tabs at the upper left.