02b –UML Examples

Randy Miller

Introducing Unified Modelling Language -UML

The basis of object-oriented problem solving is to construct a model showing the essential details of the underlying problem. There are many tools/methods available that can be used to create such a model – the most popularly accepted is Unified Modelling Language (UML).

Why is UML important?

In the construction trade an Architects design buildings and the Builders use the designs to create the 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.
Believe it or not, writing software is not unlike constructing a building. The more complicated the building structure, the more critical the communication among everyone involved in creating and deploying the software.

UML has emerged as the software version of a blueprint language used by analysts, designers, and programmers alike. It is accepted as a standard design method giving everyone from business analyst to designer to programmer a common vocabulary to talk about software design.

In UML models consist of **objects** that interact. Objects have **attributes (their data**) and Methods (things they can do). The values of an object's attributes determine its **state**.

Classes are the "blueprints" for objects. A class keeps attributes (data) and methods together in a single entity. Objects are **instances** of classes.

The "cookie cutter"

You may think of a class definition as a cookie cutter in that it is a template or pattern maker that can create actual objects. The class is not an object itself- objects are instantiated (created) by the cookie cutter (class definition)

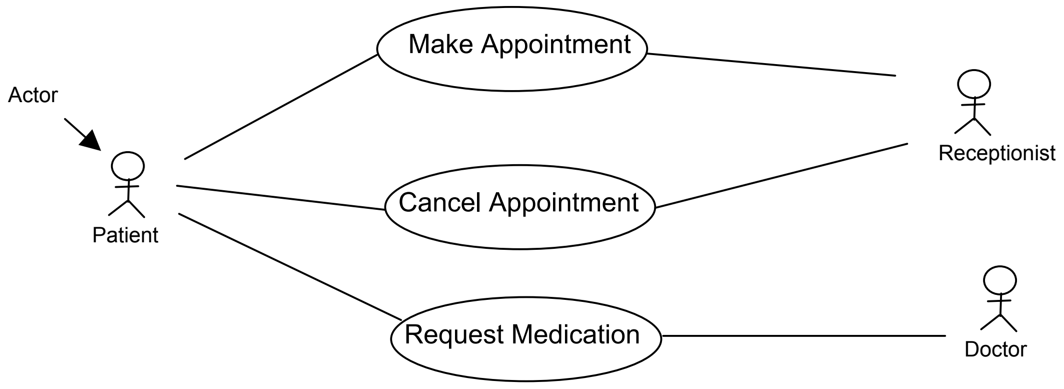
- **Use case diagrams**

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

  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 show 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 check-up. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot. "

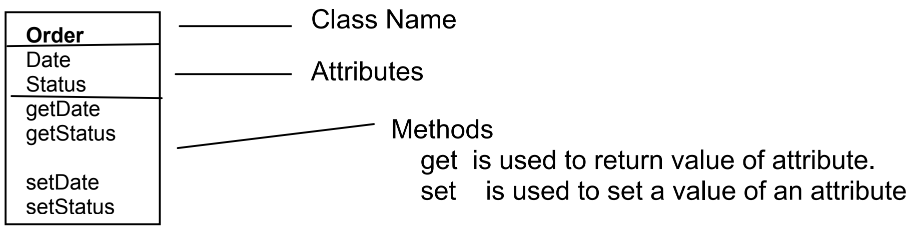


Actors are stick figures.
Use cases are ovals.
Communications are lines that link actors to use cases.
A use case diagram is a collection of actors, use cases, and their communications.
We've put Make Appointment as part of a diagram with three actors and three use cases.
Notice that a single use case can have multiple actors.

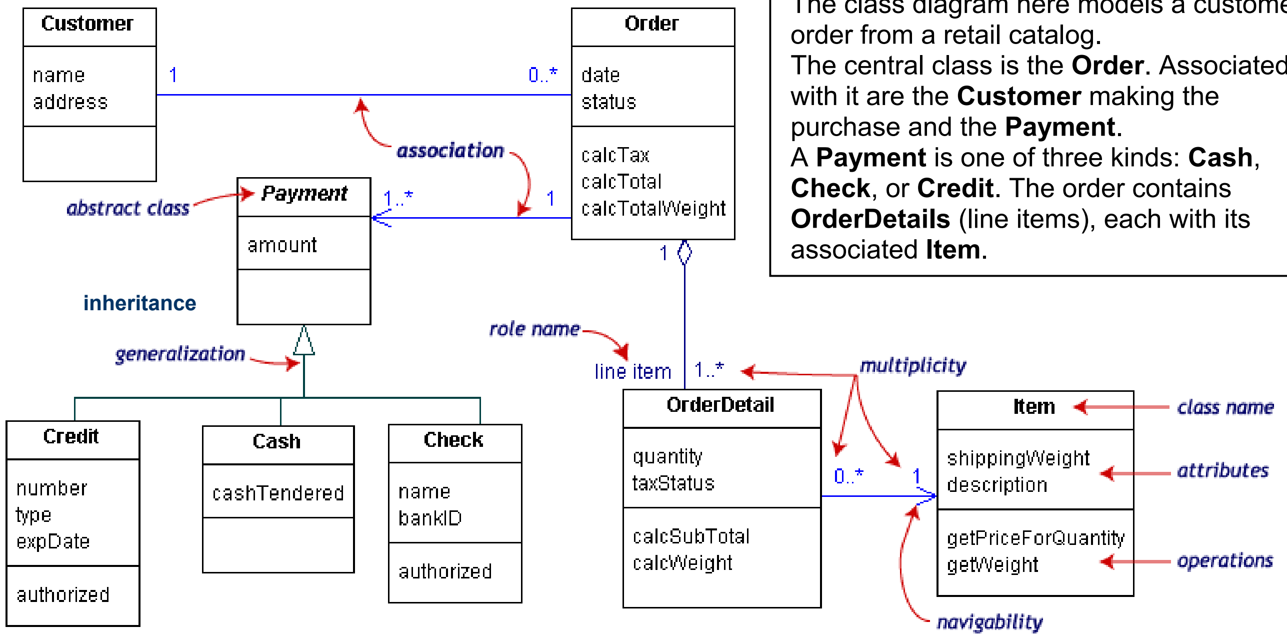
- ## Class Diagrams

  A Class diagram gives an overview of a system by showing its classes and the relationships among them. Class diagrams are static because they display what interacts but not what happens when they do interact.

  A Class is shown as a rectangle divided into three parts: class name, attributes, and Methods. Relationships between classes are the connecting links.



Order
Date
Status
getDate
getStatus
setDate
setStatus
Class Name
Attributes
Methods
get is used to return value of attribute.
set is used to set a value of an attribute



The class diagram here 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**.

This class diagram has three kinds of relationships.

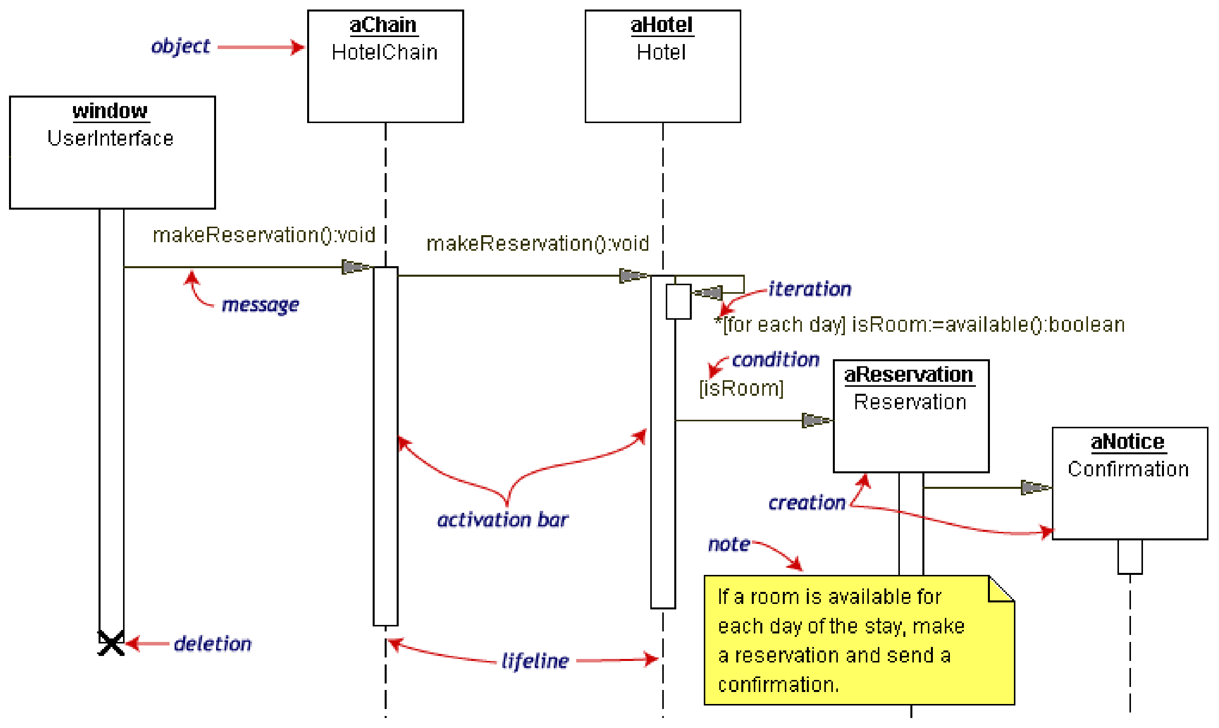
association
  a relationship between instances of the two classes. There is an association between two classes if an instance of one class must know about the other in order to perform its work. In a diagram, an association is a link connecting two classes.

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**.

inheritance
  an inheritance link indicating one class is a baseclass (parent) of the other. Inheritance has an arrow with empty triangle pointing to the baseclass. ***Payment*** is a baseclass 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**.

- **Sequence diagrams**
  Class and object diagrams are static model views.
  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**.



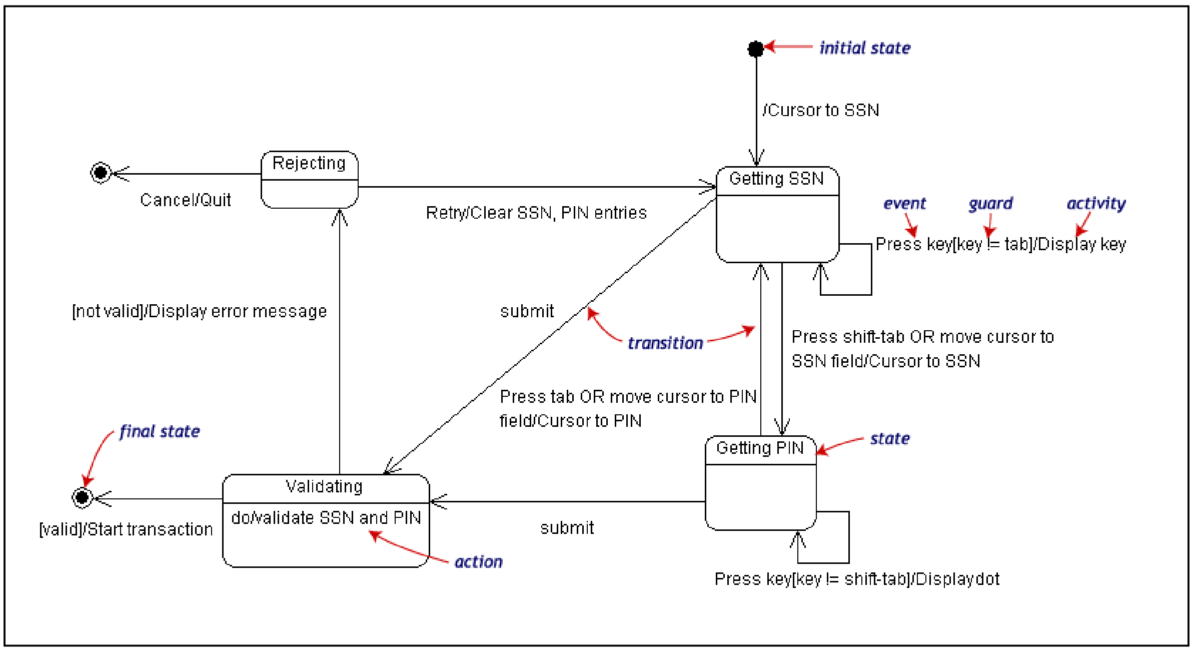
The **Reservation window** sends a makeReservation() message to a **HotelChain**. The **HotelChain** then sends a makeReservation() message to a **Hotel**. If the **Hotel** has available rooms, then it makes a **Reservation** and a **Confirmation**.
Each vertical dotted line is a **lifeline**, representing 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 this 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.

- **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.

  This 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.

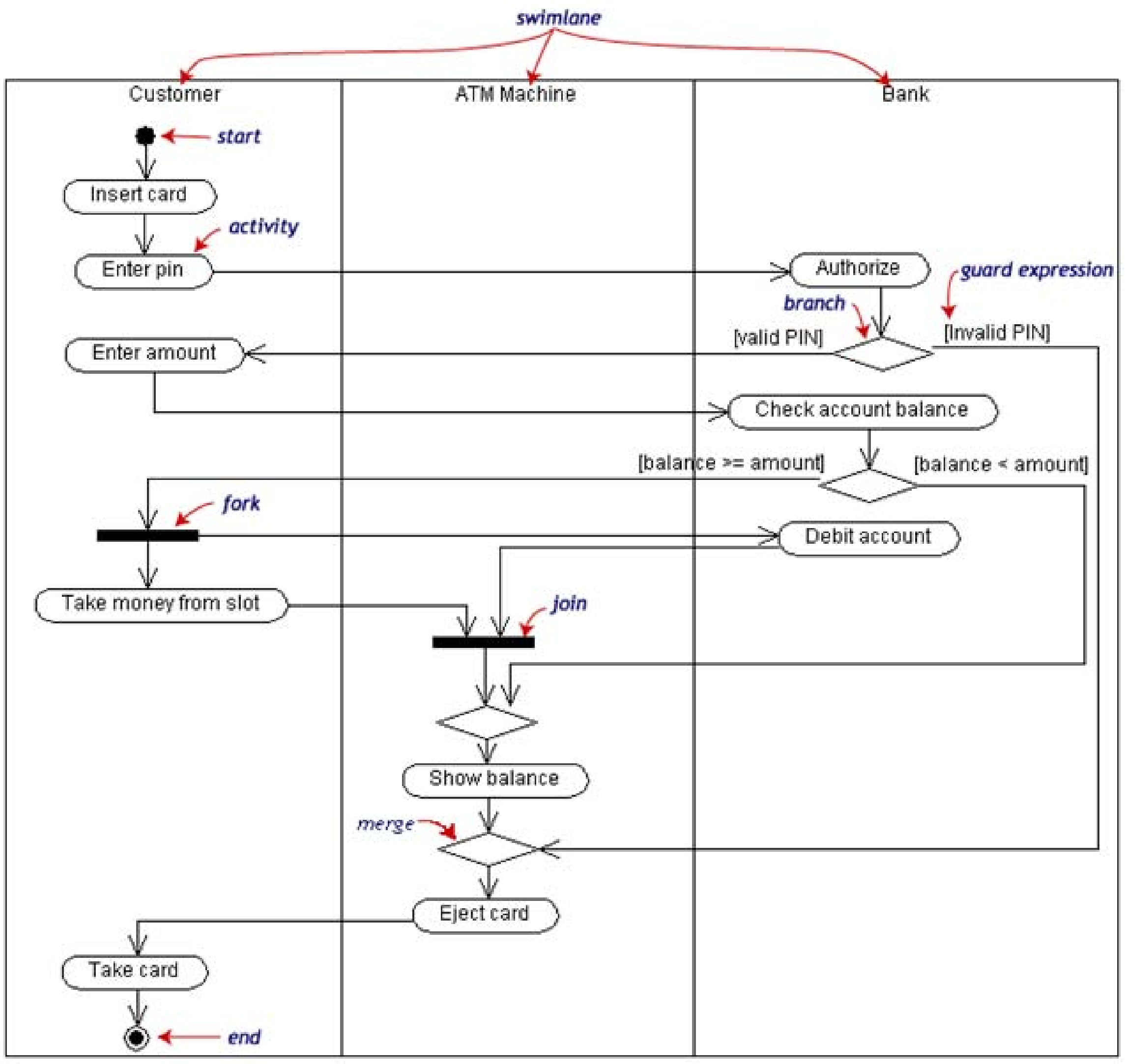


initial state
/Cursor to SSN
Rejecting
Getting SSN
Cancel/Quit
Retry/Clear SSN, PIN entries
event
guard
activity
Press key[key != tab]/Display key
[not valid]/Display error message
submit
transition
Press shift-tab OR move cursor to SSN field/Cursor to SSN
Press tab OR move cursor to PIN field/Cursor to PIN
final state
Getting PIN
state
Validating
do/validate SSN and PIN
[valid]/Start transaction
submit
action
Press key[key != shift-tab]/Display dot

  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.

- ## Activity diagrams

An **activity diagram** is essentially a 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.



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.