**UML:unified modeling language**

UML is language that unifies the best engineering practices for modeling systems

• It is a language for capturing knowledge and expressing knowledge

• Its purpose is for modeling of systems

• It provides a visual illustration of that model

• It is a collection of best practices

• It is used to produce a set of artifacts that can be delivered

• It has a world-wide support

**Diagrams of UML**

- behavior diagrams

- use-case diagrams//user-case inside actors outside

- statechart diagrams

- activity diagrams

- static-structure diagrams

- obeject diagrams

- class diagrams

- interaction or dynamic diagrams

- sequence diagrams

- collaboration diagrams//合作

- implementation diagrams

- component diagrams

- deployment diagrams

**UML:Diagrams**

 Use Case

• Show a set of use cases and actors and their relationships

 Object

• Show a set of objects and their relationships

 Class

• Show a set of classes, interfaces and collaborations and their relationships

 Interaction

• Sequence and collaboration

• Show an interaction, consisting of a set of objects and their relationships

 State

• Shows behavior a class or use case.

• Different notation

 Activity

• Show the flow from activity to activity within a system

 Component

• Show the organizations and dependencies among a set of components (subsystem)

 Deployment

• Show the configuration of run-time processing nodes and components that live on them

**1. Use-case Diagram**

 A use-case diagram describes the functionality and users (actors) of the system.

 It is used to show the relationships between the actors that use the system and the use cases they use

 A single use case can also be thought as a procedure by which an external actor can use the system

 Taken together, the use cases define the full functionality of the system from an outside perspective.

Two concepts in a use-case diagram

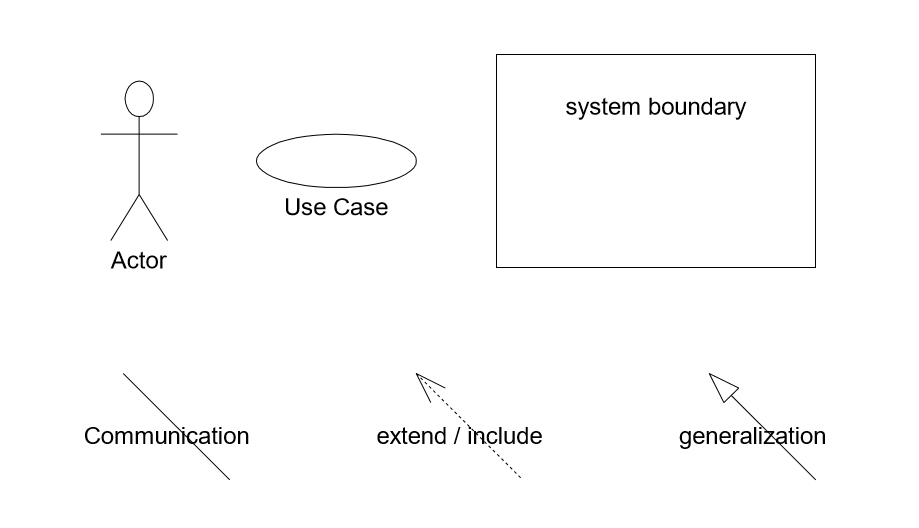
 Actor

• Represents users of the system, including human and other systems

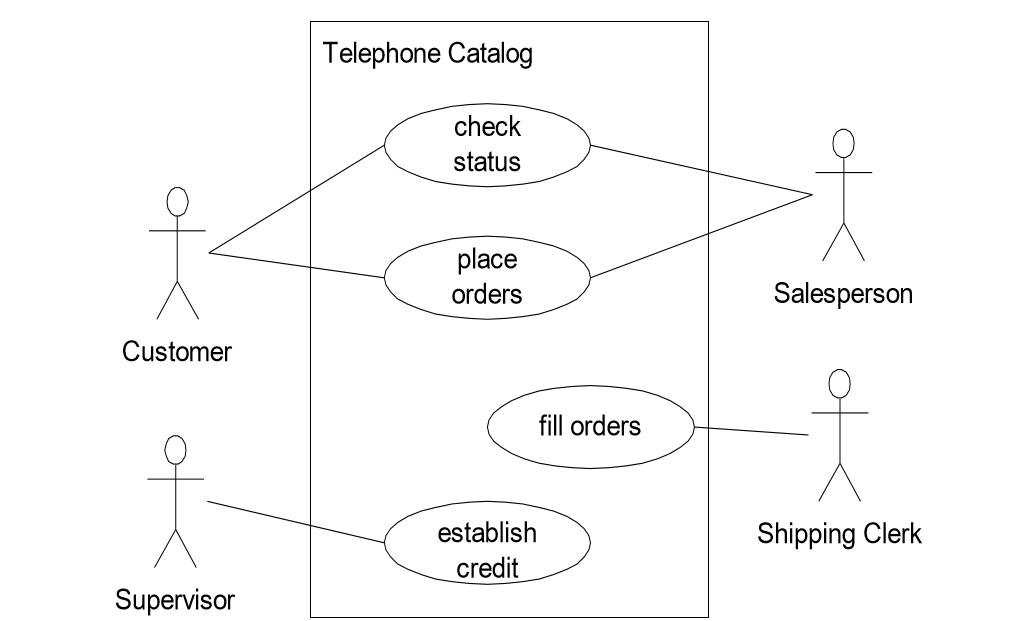
 Use Case

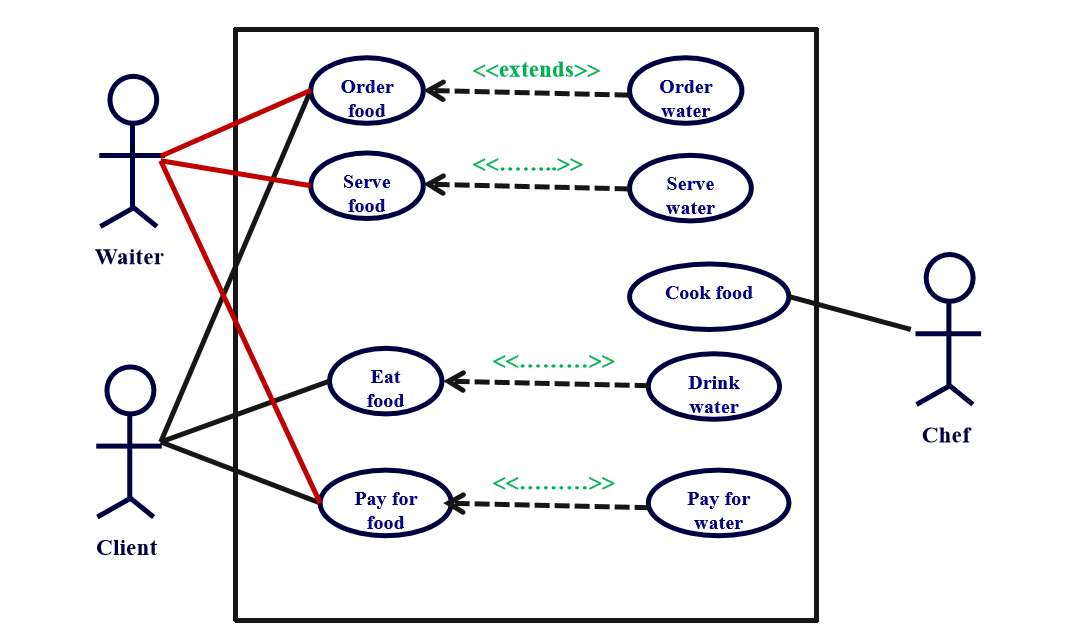
• Represents services or functionality provided by the system to the users

• The use case is attached to at least one actor, represented by a stick man, by a line called a “communicates” relationship



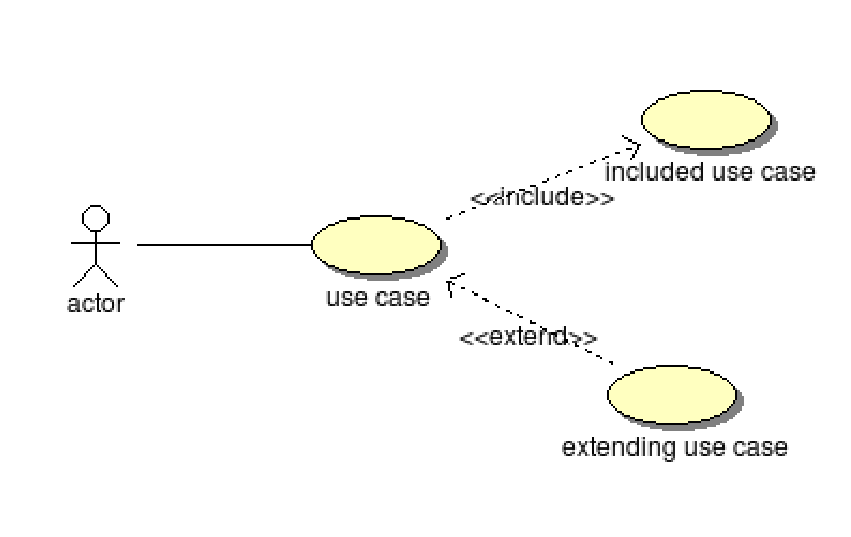
eg.

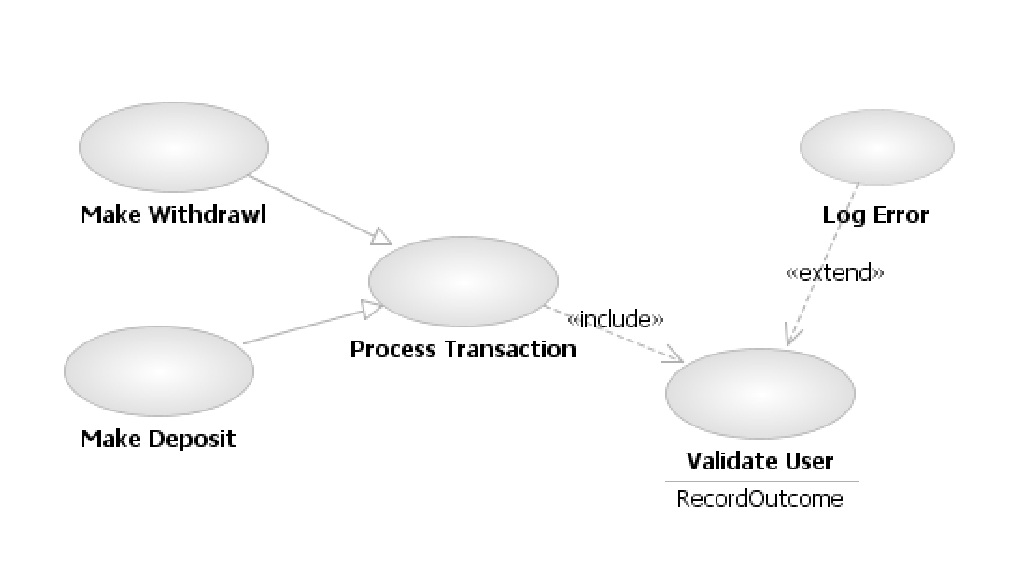


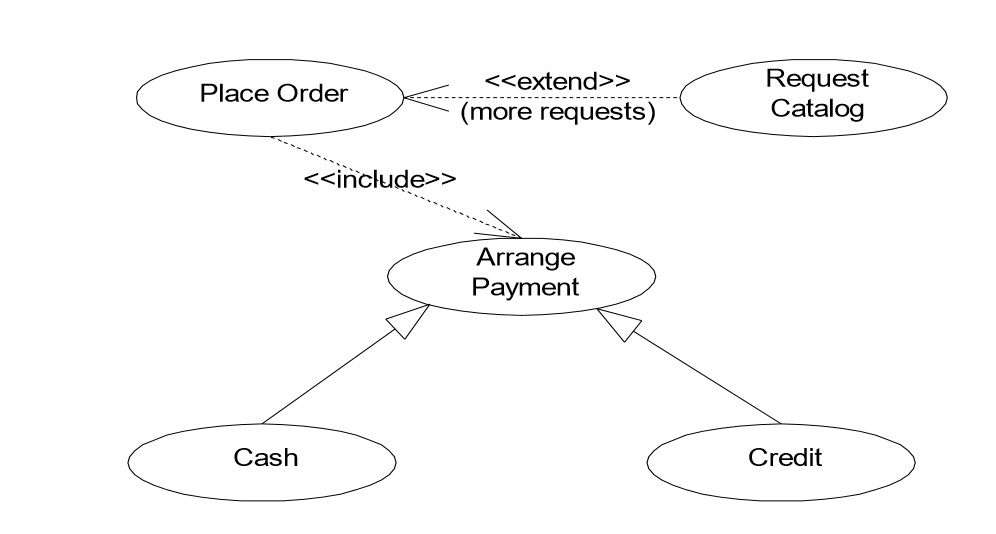


 A use case can be related to another use case by either an “include” or an “extend” relationship.  An “include” relationship indicates that the procedure of the used case is part of the procedure of the using use case

 An “extend” relationship indicates that one use case (extension) extends the behavior of another use case (base). If the procedure of the use case is an alternative or a partial alternative course to a defined in another use case.







 The arrow is, in effect, an unconditional call to the procedure in the used use case

 An “extend” relationship is used from the “extending” use case to the “extended” use case

 The procedure in the “extending” use case then replaces all or part of the procedure in the “extended” use case under conditions specified in the “extending” use case

**2. Class Diagram**

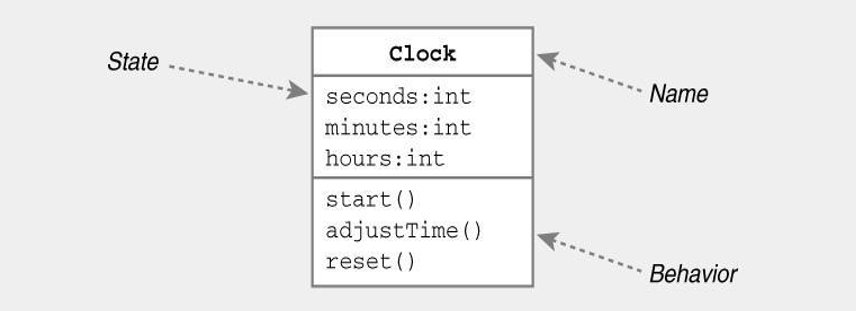
 Represent the (static) structure of the system

 General In Java

• Name Name

• State Variables

• Behavior Methods



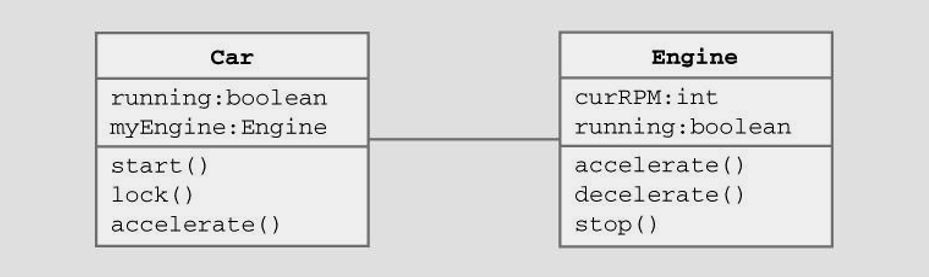
**Relationships Between Classes**

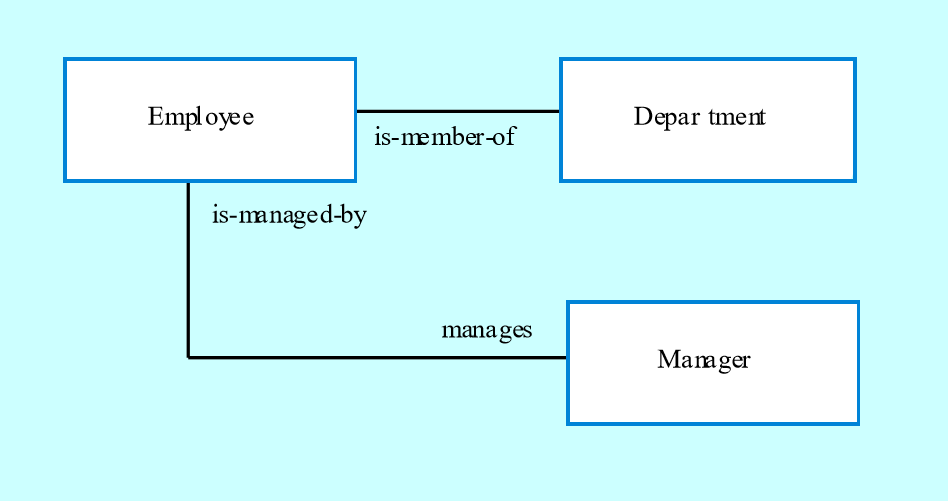
Relationships between classes show what links can exist between objects and define constraints on those links including the relative quantity of instances linked by an association. • Association or • Permanent, structural, “has a” • Solid line (arrowhead optional) • Dependency • Temporary, “uses a” • Dotted line with arrowhead • Generalization • Inheritance, “is a” • Solid line with open (triangular) arrowhead • Implementation • Dotted line with open (triangular) arrowhead

**Association**

Car and Engine classes know about each other

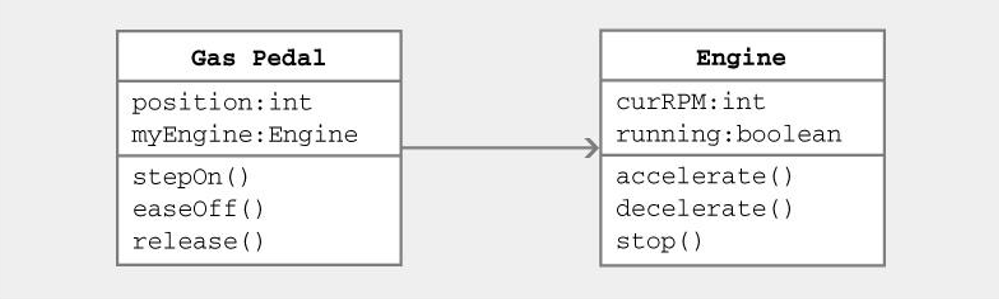
 Denotes permanent, structural relationship  State of class A contains class B  Represented by solid line (arrowhead optional)





Associations / Navigation Information

 Can indicate direction of relationship  Represented by solid line with arrowhead

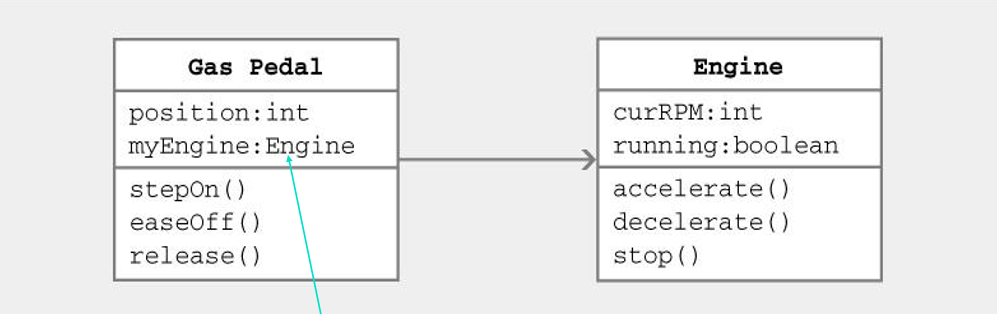


Gas Pedal class knows about Engine class Engine class doesn’t know about Gas Pedal class

Associations / Navigation Information

State of Gas Pedal class contains instance of Engine class can invoke its methods

 Denotes “has-a” relationship between classes  “Gas Pedal” has a “Engine”



Multiplicity of Associations

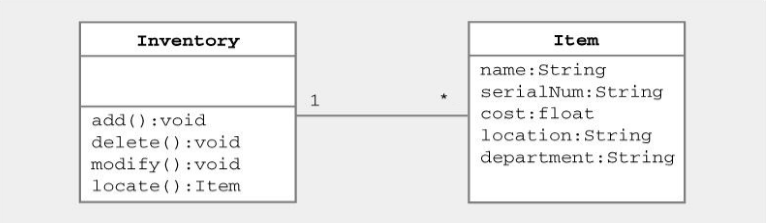
 Some relationships may be quantified  Multiplicity denotes how many objects the source object can legitimately reference  Notation • \* 0, 1, or more • 5 5 exactly • 5..8 between 5 and 8, inclusive • 5..\* 5 or more

Multiplicity of Associations

 Many-to-one • Bank has many ATMs, ATM knows only 1 bank

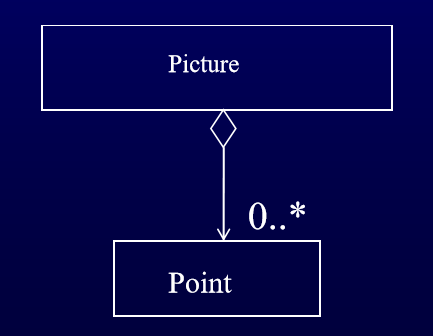


 One-to-many • Inventory has many items, items know 1 inventory



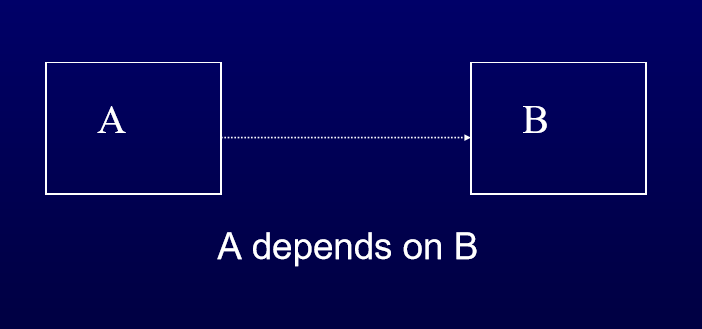
Association - Aggregation

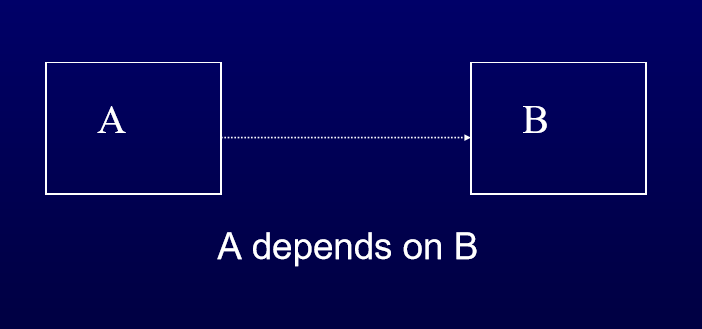
 Special case of association denotes a “consists of” hierarchy • Aggregate is the parent class • Components are the children class  Represented by line ending in open diamond

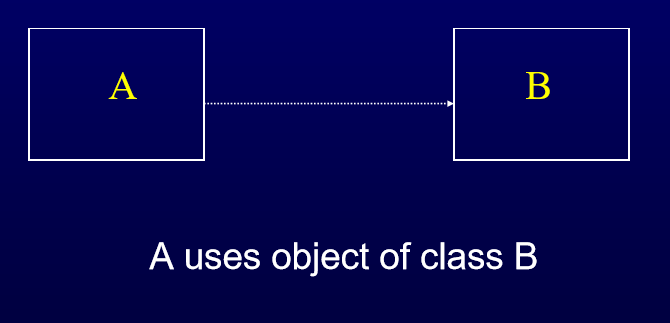


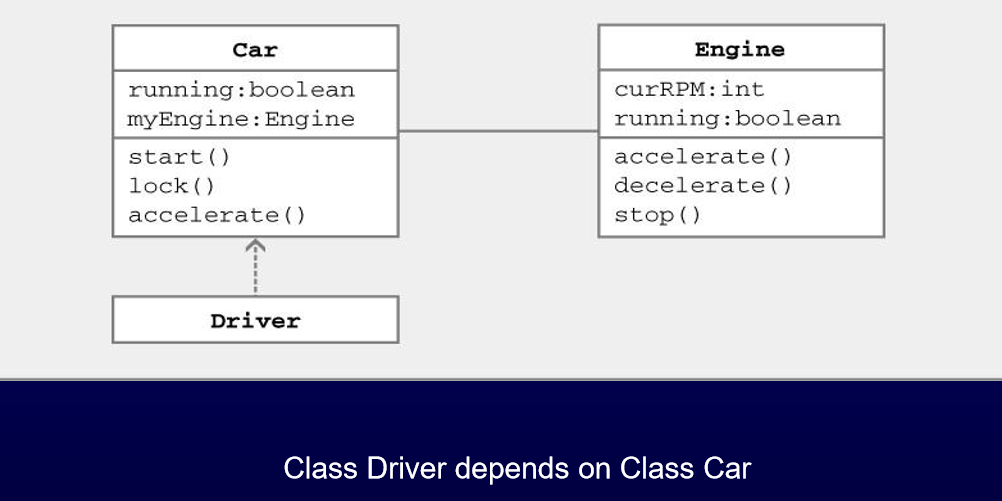
Dependency

 Denotes dependence between classes  Always directed (Class A depends on B)  Represented by dotted line with arrowhead



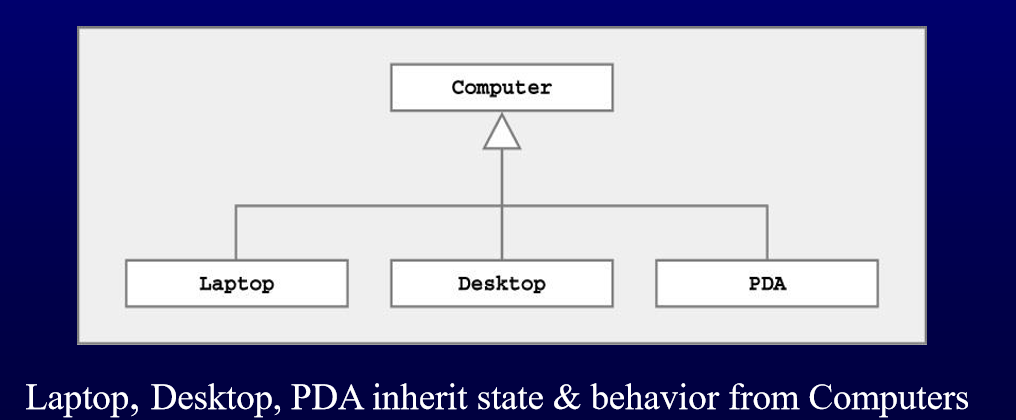






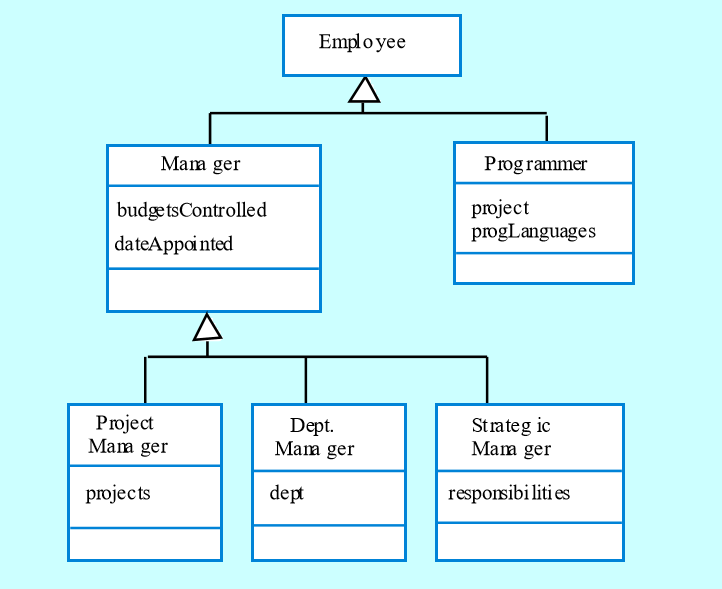
Generalization

 Denotes inheritance between classes  Can view as “is-a” relationship  Represented by line ending in (open) triangle



Generalisation and inheritance

 Objects are members of classes that define attribute types and operations.  Classes may be arranged in a class hierarchy where one class (a super-class) is a generalisation of one or more other classes (sub-classes).  A sub-class inherits the attributes and operations from its super class and may add new methods or attributes of its own.  Generalisation in the UML is implemented as inheritance in OO programming languages.



Advantages of inheritance

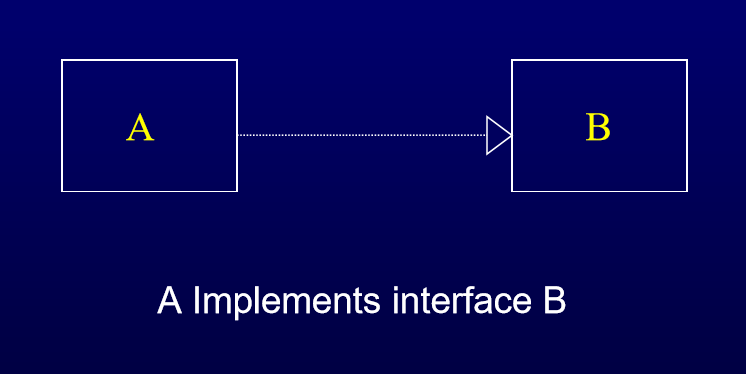
 It is an abstraction mechanism which may be used to classify entities.  It is a reuse mechanism at both the design and the programming level.  The inheritance graph is a source of organisational knowledge about domains and systems.

Problems with inheritance

 Object classes are not self-contained. they cannot be understood without reference to their super-classes.  Designers have a tendency to reuse the inheritance graph created during analysis. Can lead to significant inefficiency.  The inheritance graphs of analysis, design and implementation have different functions and should be separately maintained.

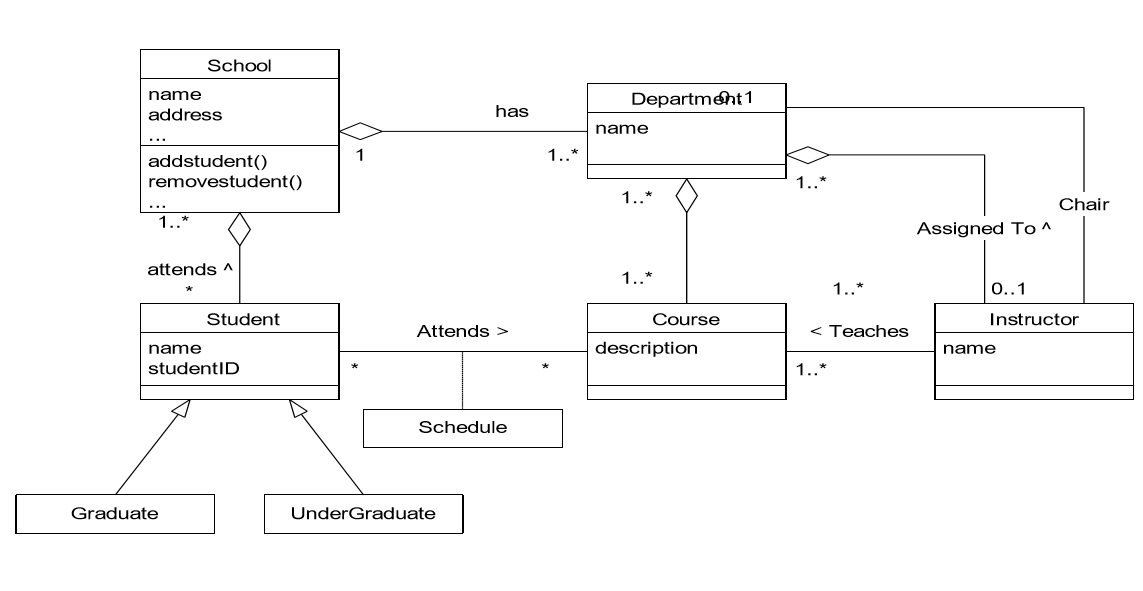
Implementation

 Denotes class implements Java interface  Represented by dotted line ending in (open) triangle



Identifying Classes

Develop an automated student registration system. The students registration system identify the School (i.e. Arts & Sciences, Engineering, Fine Arts, etc.) in which the student is registered. It also shall Identify the current classes offered by each department and the instructor for each class.

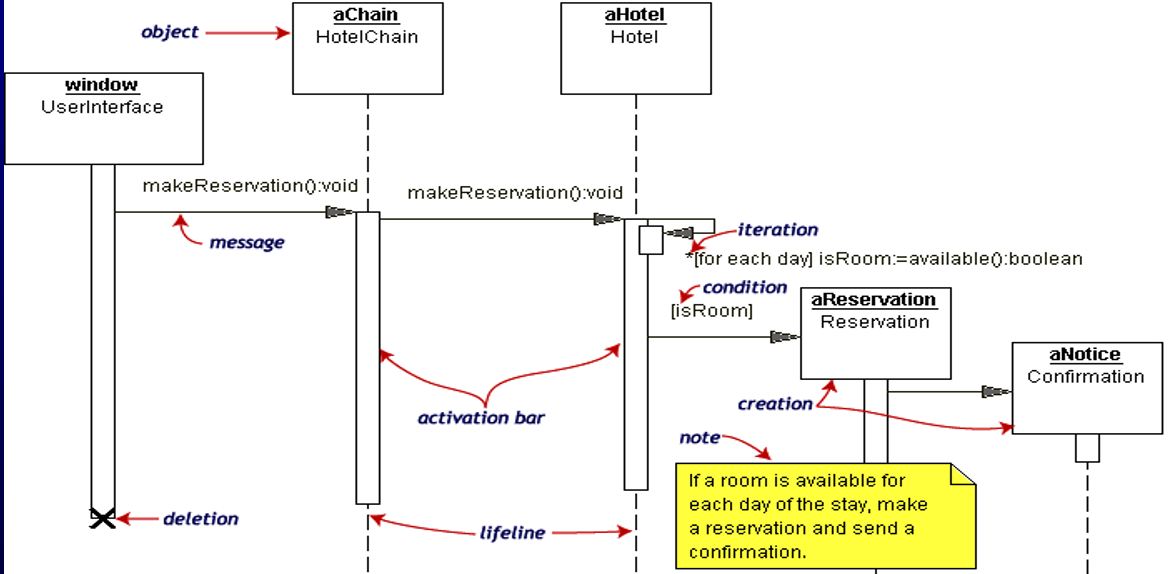


Interaction Diagrams

 Sequence diagrams, collaboration diagrams, or both diagrams can be used to demonstrate the interaction of objects in a use case. Sequence diagrams generally show the sequence of events that occur.  Collaboration diagrams demonstrate how objects are statically connected.  Both diagrams are relatively simple to draw and contain similar elements

**3. Sequence diagram**

 Sequence diagrams demonstrate the behavior of objects in a use case by describing the objects and the messages they pass.  the diagrams are read left to right and descending.



Sequence Diagram

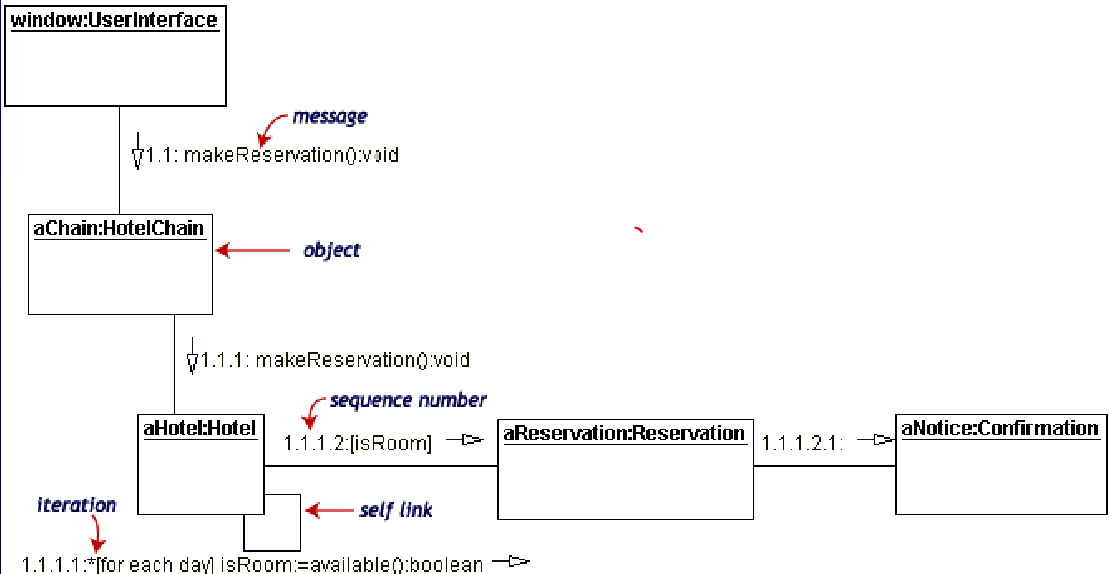
 The Reservation window sends a makeReservation() message to a HotelChain. The HotelChain then sends a makeReservation() message to a Hotel. If the Hotel has available rooms, then it makes a Reservation and a Confirmation.  Each vertical dotted line is a lifeline, representing the time that an object 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 dogeared rectangle. Notes can be put into any kind of UML diagram.

**4. Collaboration Diagram**

 Collaboration diagrams are also relatively easy to draw.  They show the relationship between objects and the order of messages passed between them.  The objects are listed as icons and arrows indicate the messages being passed between them.  The numbers next to the messages are called sequence numbers.  As the name suggests, they show the sequence of the messages as they are passed between the objects.

Collaboration Diagram

 There are many acceptable sequence numbering schemes in UML.  A simple 1, 2, 3... format can be used, as the example below shows, or for more detailed and complex diagrams a 1, 1.1 ,1.2, 1.2.1... scheme can be used.



Collaboration Diagram

 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.

**5. Statechart Diagram**

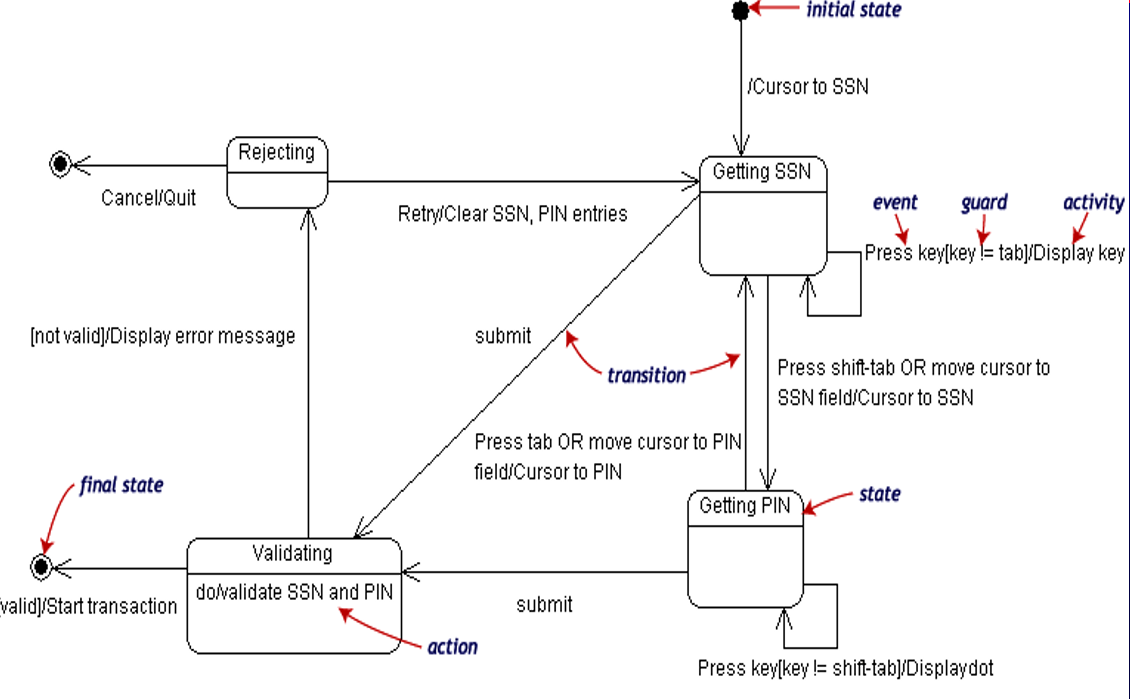
 A statechart diagram describes how the functionality of an object depends on its state and how its state changes as a result of the events that it receives.  It is used to show the dependency of operations on the order of their calling.

Statechart Diagram

 The concepts and paths in this diagram are: • State: represents an abstraction of the attribute values of an object. • Event: represents a condition that can be detected by the object • Transition: represents a response by an object to an event received by change in state. • Action: represents a set of operation that is done inside of a state or on a transition.

Statechart Diagram

 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 nonoverlapping states: Getting SSN, Getting PIN, Validating, and Rejecting. From each state comes a complete set of transitions that determine the subsequent state.



Statechart diagram

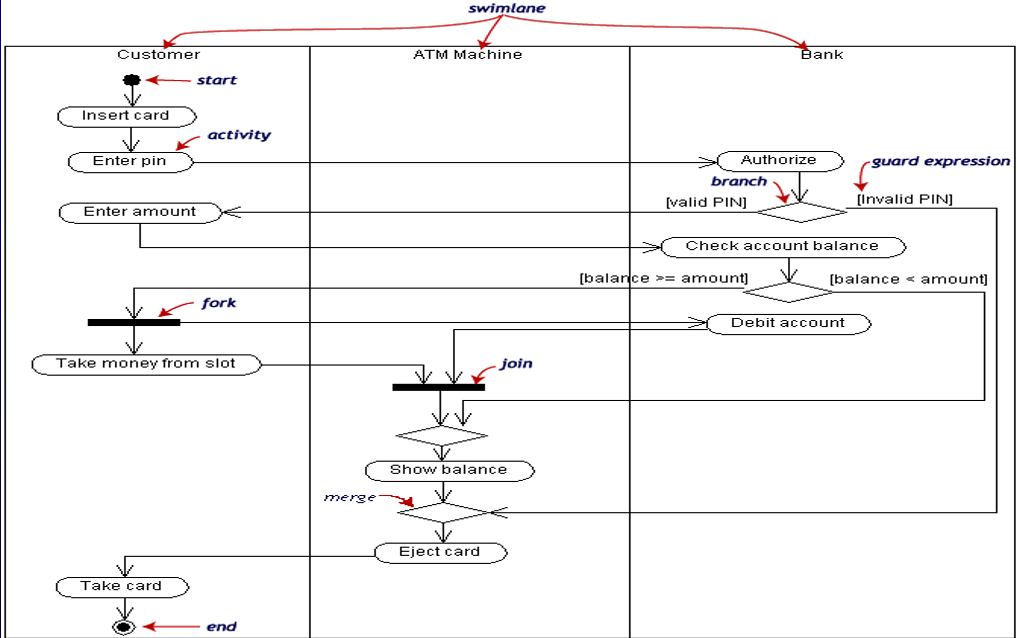
 States are rounded rectangles. Transitions are arrows from one state to another. Events or conditions that trigger transitions are written beside the arrows. Our diagram has two self-transition, one on Getting SSN and another on Getting PIN.  The initial state (black circle) is a dummy to start the 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.

**6. Activity Diagram**

 A special form of state machine. • Transitions are triggered by completion of activity • Real world continuous operation  Similar to what control flow is trying to accomplish with out the clutter.

Activity Diagram

 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 Diagram

 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.

**7. Component Diagram**

 Component diagrams fall under the category of an implementation diagram, a kind of diagram that models the implementation and deployment of the system.  A Component Diagram, in particular, is used to describe the dependencies between various software components such as the dependency between executable files and source files.  This information is similar to that within makefiles, which describe source code dependencies and can be used to properly compile an application.

Component Diagram: Notation

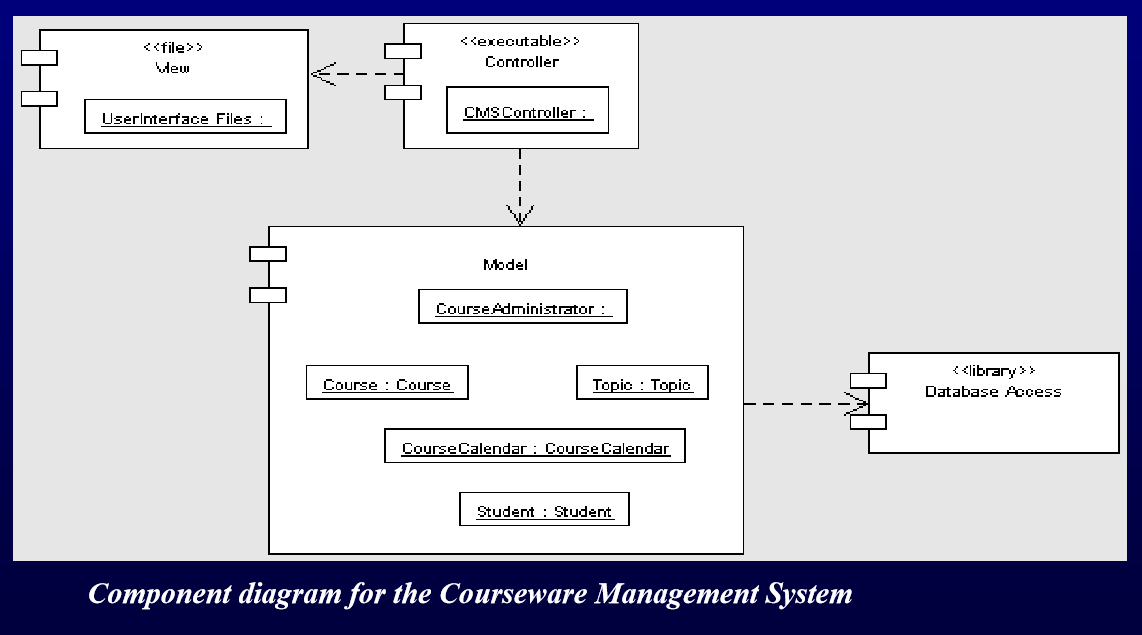
 A component represents a software entity in a system. Examples include source code files, programs, documents, and resource files. A component is represented using a rectangular box, with two rectangles protruding from the left side, as seen in the image below.



Component diagram: Notation

 A Dependency is used to model the relationship between two components. The notation for a dependency relationship is a dotted arrow, pointing from a component to the component it depends on.





Component Diagram: Example

 The previous figure shows the Component diagram for the Model layer of the Courseware Management System.  The diagram shows the different components, such as CourseAdministrator, Course, Topic, Tutor, and so forth in the Model layer and how the Controller layer component interacts with these components.  The diagram also depicts a database access component that represents a library component that the Model layer components will use to interact with a database.

**8. Deployment Diagram**

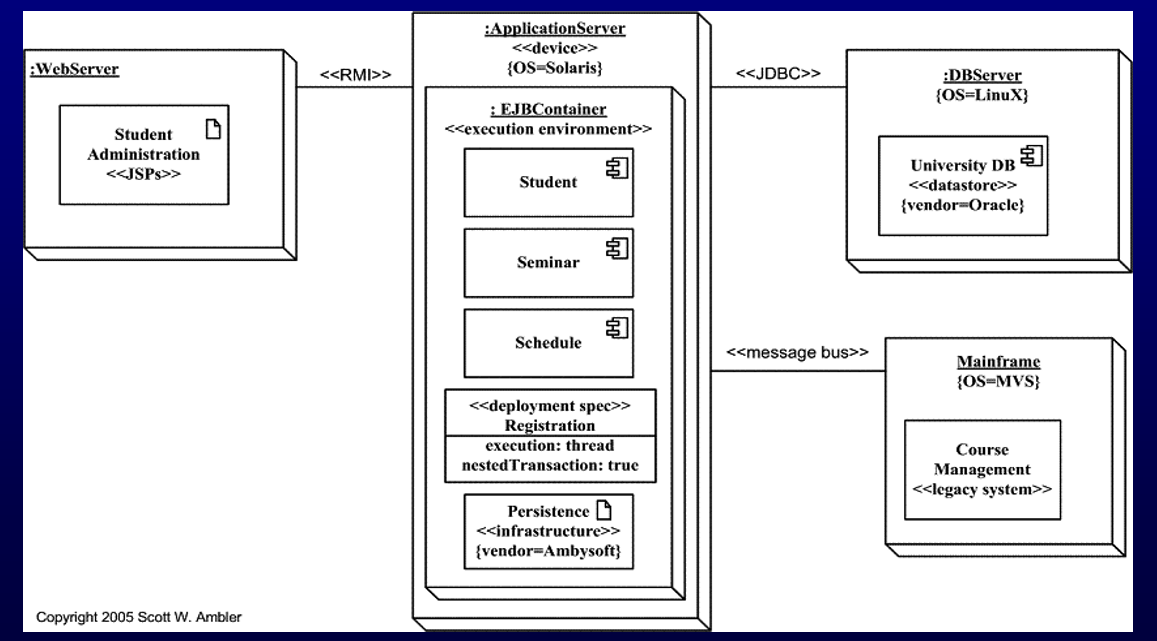
 Deployment diagrams are another model in the implementation diagram category.  The Deployment diagram models the hardware used in implementing a system and the association between those hardware components.  Components can also be shown on a Deployment diagram to show the location of their deployment.  Deployment diagrams can also be used early on in the design phase to document the physical architecture of a system.

Deployment Diagram: Notation

 A Component  A node represents a piece of hardware in the system. This entity is represented by a three-dimensional cube.



An association, drawn as a solid line between two Nodes, indicates a line of communication between the hardware elements



The previous figure presents an example of a fully rendered UML 2 deployment diagram for the student administration application. The three-dimensional boxes represent nodes, either software or hardware. Physical nodes should be labeled with the stereotype device, to indicate that it’s a physical device such as a computer or switch. As you can see I didn’t indicate that WebServer is a device – it will at least be some sort of software artifact and very well may be one or more physical devices as well but my team hasn’t made that decision yet. Remember, models evolve over time. Connections between nodes are represented with simple lines, and are assigned stereotypes such as RMI and message bus to indicate the type of connection.