Syllabus : Unit – II : Introduction To Uml
Why we model – Conceptual model of UML – Architecture - Classes - Relationships - Common mechanisms - Diagrams – Class diagrams - Object diagrams

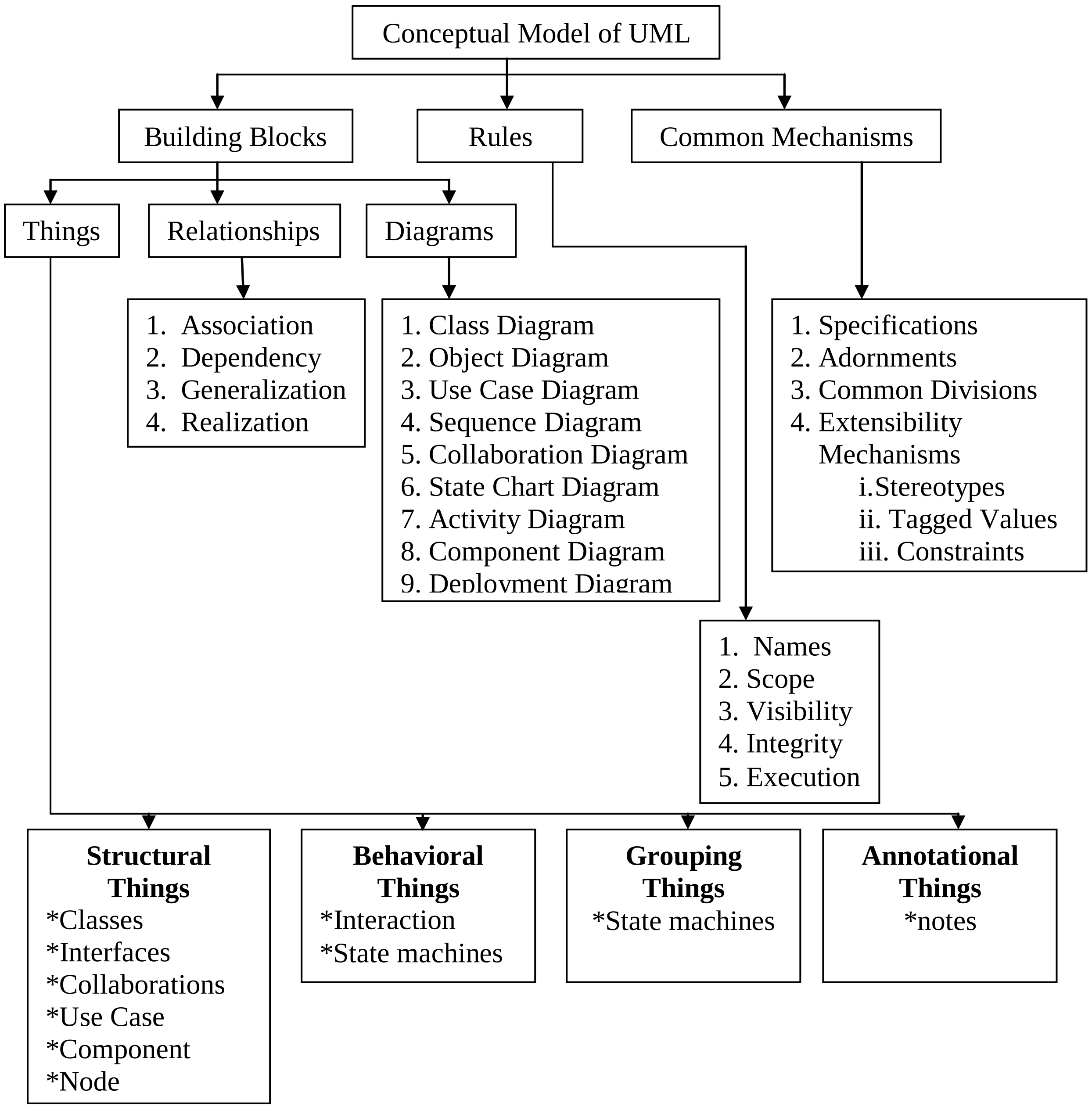
Introducing the UML

- A **modeling language** is a language whose vocabulary and rules focus on the conceptual and physical representation of a system.
- The UML is a language for
  - ✓ Visualizing
  - ✓ Specifying
  - ✓ Constructing
  - ✓ Documenting the artifact of a software-intensive system.
    - **Visualizing**
      - Some things are best modeled textually; others are best modeled graphically. In all interesting system, there are structures that transcend what can be represented in a programming language.
    - **Specifying**
      - Specifying means building models that are precise, unambiguous and complete. The UML addresses the specification of all the important analysis, design and implementation decision.
    - **Constructing**
      - The UML is not a visual programming language, but its models can be directly connected to a variety of programming language (Java, C++, ...)
    - **Documenting**
      - A software solution is made up by all sorts of artifacts in addition to raw executable code. These artifacts include:
        - Requirements
        - Architecture
        - Design
        - Source code
        - Project Plans
        - Tests
        - Prototypes
        - Releases
- **Benefits of UML**
  - ✓ Enterprise information systems
  - ✓ Banking and financial services
  - ✓ Telecommunications
  - ✓ Transportation
  - ✓ Defense/aerospace
  - ✓ Retail
  - ✓ Medical electronics
  - ✓ Scientific
  - ✓ Distributed Web-based services

Why we model

- **The UML models act as an architectural blueprint for software development.**
- **Good models:**
  - ✓ Identify requirements and communicate information.
  - ✓ Allows focus on how system components interact.
  - ✓ Allows you to see relationships among design components
  - ✓ Improves communication across your team through the use of common graphical language

Conceptual model of the UML



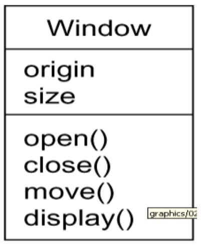
- **Conceptual model of the UML cover three major elements:**

  1. Things
  2. Relationships
  3. Diagrams

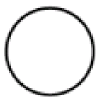
- **Things in the UML**
  - ✓ There are four kinds of things in the UML:
    - Structural things
    - Behavioral things
    - Grouping things
    - Annotational things
    -
  - ✓ **Structural things**

    - Structural things are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical. In all, there are seven kinds of structural things.
      1. Classes
      2. Interfaces
      3. Collaborations
      4. Use cases
      5. Active classes
      6. Components
      7. Nodes

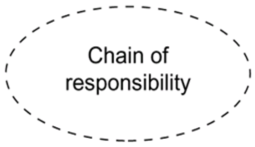
    - <u>Class</u>
      - Class is a description of a set of objects that share the same attributes, operations, relationships, and semantics.
      - A class implements one or more interfaces. Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations.



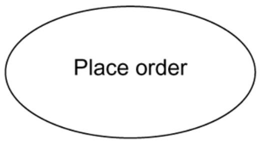
    - <u>Interface</u>
      - Interface is a collection of operations that specify a service of a class or component.
      - An interface therefore describes the externally visible behavior of that element.



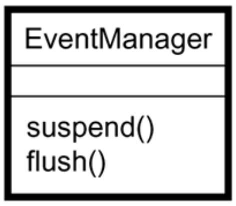
- Collaboration
  - Collaboration defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. Therefore, collaborations have structural, as well as behavioral, dimensions. A given class might participate in several collaborations.
  - Graphically, a collaboration is rendered as an ellipse with dashed lines, usually including only its name



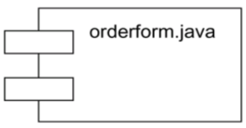
- Usecase
  - Use case is a description of set of sequence of actions that a system performs that yields an observable result of value to a particular actor
  - Use case is used to structure the behavioral things in a model.
  - A use case is realized by collaboration.
  - Graphically, a use case is rendered as an ellipse with solid lines, usually including only its name



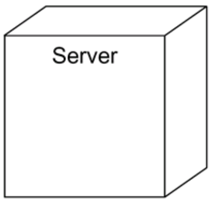
- Active class
  - Active class is just like a class except that its objects represent elements whose behavior is concurrent with other elements. Graphically, an active class is rendered just like a class, but with heavy lines, usually including its name, attributes, and operations



- Component
  - Component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces.
  - Graphically, a component is rendered as a rectangle with tabs.



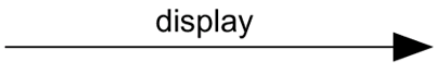
- <u>Node</u>
    - Node is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability.
    - Graphically, a node is rendered as a cube, usually including only its name.



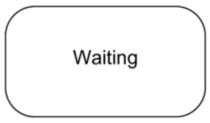
✓ **Behavioral Things**

- Behavioral things are dynamic part of system
    1. Interaction
    2. **2.** State machine

- <u>Interaction</u>
    - Interaction is a behavior that comprises a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose.
    - An interaction involves a number of other elements, including messages, action sequences and links.
    - Graphically a message is rendered as a directed line, almost always including the name of its operation.

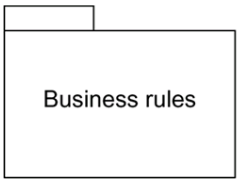


- <u>State Machine</u>
    - State machine is a behavior that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events.
    - State machine involves a number of other elements, including states, transitions, events and activities.
    - Graphically, a state is rendered as a rounded rectangle, usually including its name and its sub states.

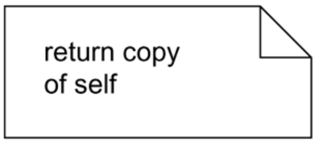


- ✓ **Grouping Things**
  - They are the organizational parts of UML models. These are the boxes into which a model can be decomposed.
  - There is one primary kind of grouping thing, namely, packages.

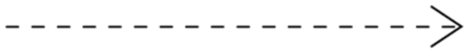
  - <u>Package</u>
    - A package is a general-purpose mechanism for organizing elements into groups. Structural things, behavioral things, and even other grouping things may be placed in a package.



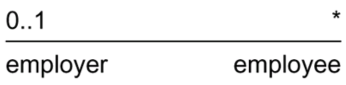
- ✓ **Annotational things**
  - Annotational things are the explanatory parts of UML models. These are the comments you may apply to describe about any element in a model.
  - <u>Note</u>
    - Note is simply a symbol for rendering constraints and comments attached to an element or a collection of elements.
    - Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment



- **Relationships**
  - ✓ There are four kinds of relationships in the UML:
    - ▪ Dependency
    - ▪ Association
    - ▪ Generalization
    - ▪ Realization
  - ✓ <u>Dependency</u>
    - Dependency is a semantic relationship between two things in which a change to one thing may affect the semantics of the other thing
    - Graphically a dependency is rendered as a dashed line, possibly directed, and occasionally including a label.



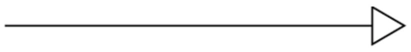
  - ✓ <u>Association</u>
    - Association is a structural relationship that describes a set of links, a link being a connection among objects.
    - Graphically an association is rendered as a solid line.



- Aggregation
  - Aggregation is a special kind of association, representing a structural relationship between a whole and its parts. Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent

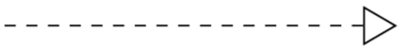
✓ Generalization

- It is a specialization/generalization relationship in which the specialized element (the child) builds on the specification of the generalized element (the parent).
- The child shares the structure and the behavior of the parent.
- Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent.



✓ Realization

- Realization is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out.
- Graphically a realization relationship is rendered as a cross between a generalization and a dependency relationship



- **Diagrams in the UML**

✓ A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and paths (relationships).

✓ We draw diagrams to visualize a system from different perspectives, so a diagram is a projection into a system.

✓ In theory, a diagram may contain any combination of things and relationships.

✓ UML includes the following nine diagrams:

- **<u>Structural Diagrams</u>**
  - The structural diagrams represent the static aspect of the system. These static aspects represent those parts of a diagram, which forms the main structure and are therefore stable.

1. Class diagram
   - Class diagrams are the most common diagrams used in UML.
   - Class diagram consists of classes, interfaces, associations, and collaboration.
   - Class diagrams basically represent the object-oriented view of a system, which is static in nature.
   - Active class is used in a class diagram to represent the concurrency of the system.

2. Object diagram
    - Object diagrams are a set of objects and their relationship is just like class diagrams. They also represent the static view of the system.
    - Object diagrams can be described as an instance of class diagram. Thus, these diagrams are more close to real-life scenarios where we implement a system.
    - The usage of object diagrams is similar to class diagrams but they are used to build prototype of a system from a practical perspective.

3. Component diagram
    - Component diagrams represent a set of components and their relationships.
    - These components consist of classes, interfaces, or collaborations.
    - Component diagrams represent the implementation view of a system.

4. Deployment Diagram
    - Deployment diagrams are a set of nodes and their relationships. These nodes are physical entities where the components are deployed.
    - Deployment diagrams are used for visualizing the deployment view of a system.
    - This is generally used by the deployment team.

- **Behavioral Diagrams**
    - Behavioral diagrams basically capture the dynamic aspect of a system. Dynamic aspect can be further described as the changing/moving parts of a system.

5. Use case diagram
    - Use case diagrams are a set of use cases, actors, and their relationships. They represent the use case view of a system.
    - A use case represents a particular functionality of a system.
    - Use case diagram is used to describe the relationships among the functionalities and their internal/external controllers. These controllers are known as actors.

6. Sequence diagram
    - A sequence diagram is an interaction diagram.
    - The diagram deals with some sequences, which are the sequence of messages flowing from one object to another.
    - Sequence diagram is used to visualize the sequence of calls in a system to perform a specific functionality.

7. <u>Collaboration diagram</u>
   - Collaboration diagram is another form of interaction diagram.
   - It represents the structural organization of a system and the messages sent/received.
   - Structural organization consists of objects and links.
   - The purpose of collaboration diagram is similar to sequence diagram.

8. <u>State chart diagram</u>
   - A state diagram shows a state machine, consisting of states, transitions, events, and activities.
   - State chart diagram is used to represent the event driven state change of a system. It basically describes the state change of a class, interface, etc.
   - State chart diagram is used to visualize the reaction of a system by internal/external factors.

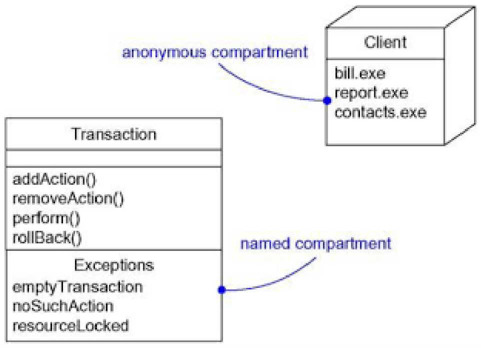
9. <u>Activity Diagram</u>
   - Activity diagram describes the flow of control in a system. It consists of activities and links. The flow can be sequential, concurrent, or branched.
   - Activity diagrams are used to visualize the flow of controls in a system. This is prepared to have an idea of how the system will work when executed.

- **Rules**
  - ✓ UML has a number of rules so that the models are semantically self-consistent and related to other models in the system harmoniously. UML has semantic rules for the following −
    - o Names - What you can call things, relationships, and diagrams
    - o Scope - The context that gives specific meaning to a name
    - o Visibility - How those names can be seen and used by others
    - o Integrity - How things properly and consistently relate to one another
    - o Execution - What it means to run or simulate a dynamic model

- **Common Mechanisms**
  - ✓ UML has four common mechanisms:
    - o Specifications
    - o Adornments
    - o Common Divisions
    - o Extensibility Mechanisms

  - ✓ <u>Specifications</u>
    - o In UML, behind each graphical notation, there is a textual statement denoting the syntax and semantics. The specifications provide a semantic backplane that contains all the parts of a system and the relationship among the different paths.

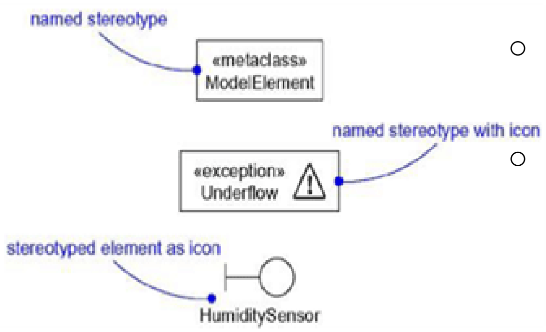
- ✓ Adornments
  - Adornments are textual or graphical items that are added to the elements basic notation to specify extra information. When using UML, always start with the basic notation of elements and add adornments to specify new information.
  - Example: adornments of an association.



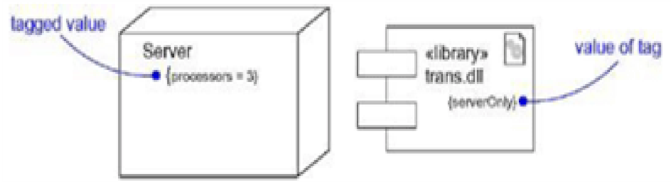
- ✓ Common Divisions
  - Object-oriented systems can be divided in many ways. The two common ways of division are:
    - **Division of classes and objects:** A class is an abstraction of a group of similar objects. An object is the concrete instance that has actual existence in the system.
    - **Division of Interface and Implementation:** An interface defines the rules for interaction. Implementation is the concrete realization of the rules defined in the interface.

- ✓ Extensibility Mechanisms
  - UML is an open-ended language. It is possible to extend the capabilities of UML in a controlled manner to suit the requirements of a system. The extensibility mechanisms are

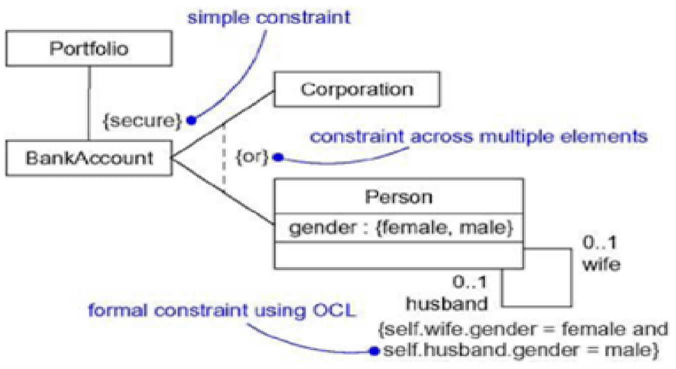
    - Stereotypes:
      - It extends the vocabulary of the UML, through which new building blocks can be created out of existing ones. (or)
      - UML provides basic notations for structural things, behavioral things, grouping things and annotational things. Sometimes we may need to extend i.e., create new vocabulary and look like primitive building blocks.
      - A stereotype allows us to create new building blocks. Using stereotypes we can use the basic elements but with special properties, semantics and notation.
      - Stereotypes are rendered as text inside guillemets(<< >>) or we can create new icons for stereotypes.



- Tagged Values:
  - It extends the properties of UML building blocks. (or)
  - Every element in UML has its own properties. For example a class has its own attributes and operations.
  - Using tagged values, we can represent new properties also called as metadata.
  - These properties apply to the element itself rather than its instance.
  - Tagged values are enclosed in braces { } and are written under the element name.

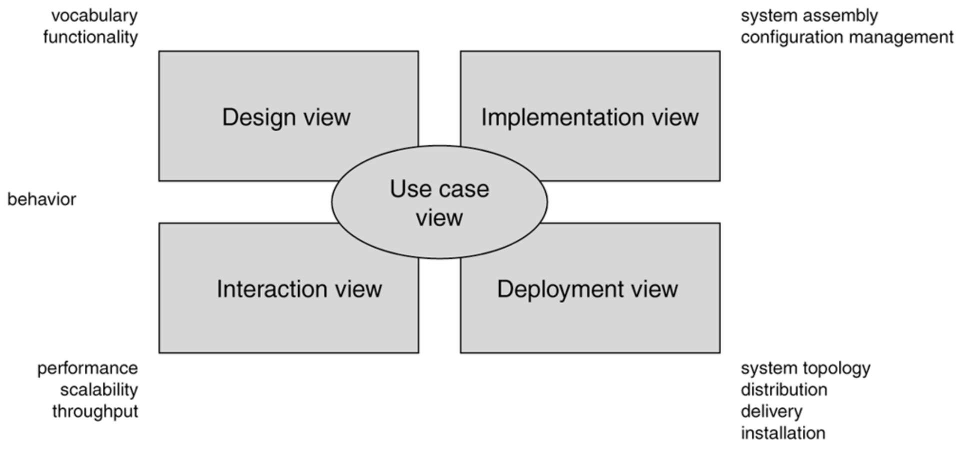


- Constraints :
  - It extends the semantics of UML building blocks. (or)
  - A constraint is used to add new semantics or change existing rules.
  - The constraints specify rules that must be followed by the elements in the model.
  - Represented as text inside braces { } and placed near to the associated element.


simple constraint
Portfolio
{secure}
BankAccount
Corporation
constraint across multiple elements
{or}
Person
gender : {female, male}
0..1
wife
0..1
husband
formal constraint using OCL
{self.wife.gender = female and
self.husband.gender = male}

Architecture

- Visualizing, specifying, constructing, and documenting a software-intensive system demands.
  - ✓ system be viewed from a number of perspectives
- Different stakeholders, end users, analysts, developers, system integrators, testers, technical writers, and project managers
  - ✓ each bring different agendas to a project
  - ✓ each looks at the system in different ways at different times over the project's life
- The most important artifact that can be used to manage these different viewpoints and so control development of a system throughout its life cycle.
- Concerned with structure, behavior, usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and trade-offs, and aesthetic concern.
- The architecture of a software-intensive system can best be described by five interlocking views. Each view is a projection into the organization and structure of the system, focused on a particular aspect of that system



- **<u>Use-Case View:</u>**
  - ✓ This view showing the behavior of the system as perceived by the external actors.
  - ✓ It exposes the requirements of the system.
  - ✓ With UML,
    - o The static aspects of this view are captured in use case diagrams
    - o The dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.
- **<u>Design View:</u>**
  - ✓ This view showing how the functionality is designed inside the system, in terms of the static structure and dynamic behaviour.
  - ✓ It captures the vocabulary of the problem space and solution space.
  - ✓ With UML,
    - o The static aspects of this view are captured in class and object diagrams
    - o The dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

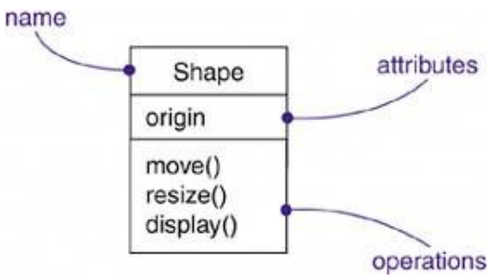
- **Implementation View:**
  - ✓ The implementation view of a system encompasses the artifacts that are used to assemble and release the physical system.
  - ✓ This view primarily addresses the configuration management of the system's releases.
  - ✓ With UML,
    - o The static aspects of this view are captured in component diagrams
    - o The dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

- **Interaction view (Process View):**
  - ✓ The interaction view of a system shows the flow of control among its various parts, including possible concurrency and synchronization mechanisms.
  - ✓ It encompasses the threads and processes that form the system's concurrency and synchronization mechanisms.
  - ✓ This view primarily addresses the performance, scalability, and throughput of the system.
  - ✓ With UML,
    - o The static and dynamic aspects of this view are captured in same way as design view, but with a focus on the active classes that represent these threads and processes

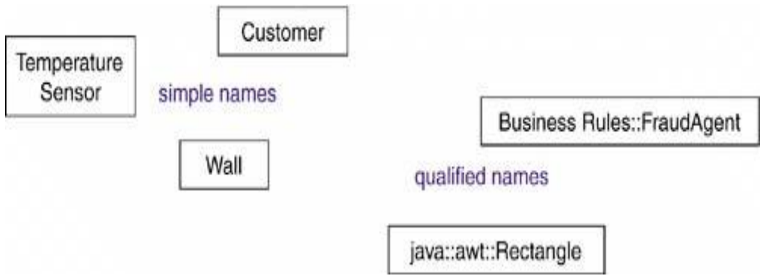
- **Deployment View:**
  - ✓ This view showing the deployment of the system in terms of the physical architecture.
  - ✓ It encompasses the nodes that form the system's hardware topology on which the system executes.
  - ✓ This view primarily addresses the distribution, delivery, and installation of the parts that make up the physical system.
  - ✓ With UML,
    - o The static aspects of this view are captured in deployment diagrams
    - o The dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

Classes

- ✓ Classes are the most important building block of any object-oriented system.
- ✓ A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics.
- ✓ A class implements one or more interfaces.
- ✓ We use classes to capture the vocabulary of the system we are developing. These classes may include abstractions that are part of the problem domain, as well as classes that make up an implementation.
- ✓ We can use classes to represent software things, hardware things, and even things that are purely conceptual.
- ✓ Graphically, a class is rendered as a rectangle.
- ✓ This notation permits we to visualize an abstraction apart from any specific programming language and in a way that lets you emphasize the most important parts of an abstraction: its name, attributes, and operations.

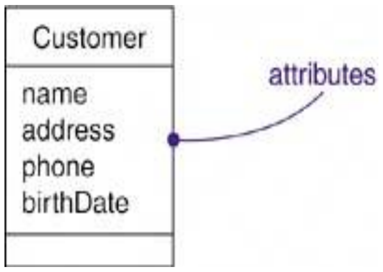


- ✓ **Terms and Concepts**
  - **Names**
    - Every class must have a name that distinguishes it from other classes.
    - A *name* is a textual string.
    - That name alone is known as
      - a *simple name*
      - a *qualified name* is the class name prefixed by the name of the package in which that class lives.
    - A class may be drawn showing only its name, as Figure shows.

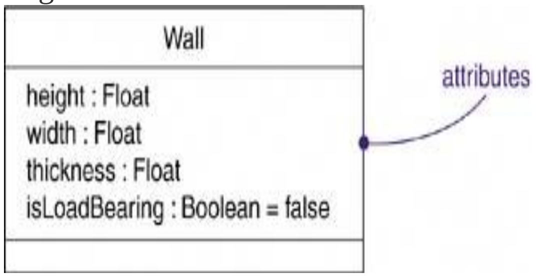


  - **Attributes**
    - An attribute is a named property of a class that describes a range of values that instances of the property may hold.
    - A class may have any number of attributes or no attributes at all.
    - An attribute represents some property of the thing we are modeling that is shared by all objects of that class.

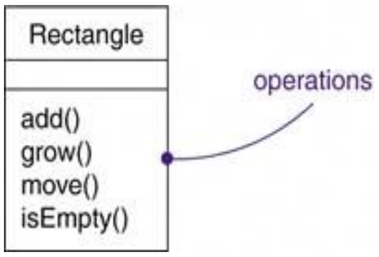
- Graphically, attributes are listed in a compartment just below the class name. Attributes may be drawn showing only their names, as shown in Figure



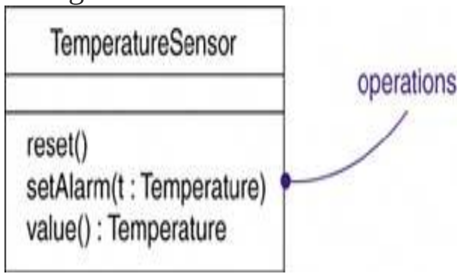
- Attributes with Signatures:



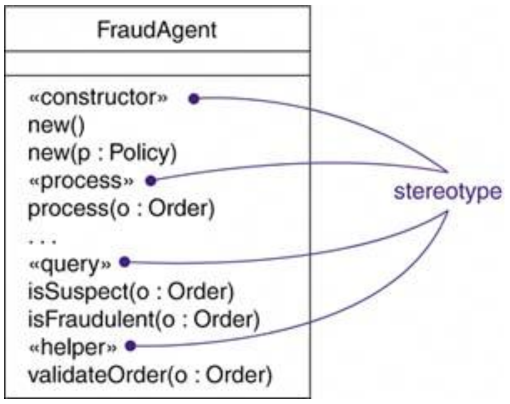
- **Operations**
  - An operation is the implementation of a service that can be requested from any object of the class to affect behavior.
  - A class may have any number of operations or no operations at all.
  - Graphically, operations are listed in a compartment just below the class attributes. Operations may be drawn showing only their names, as in Figure.



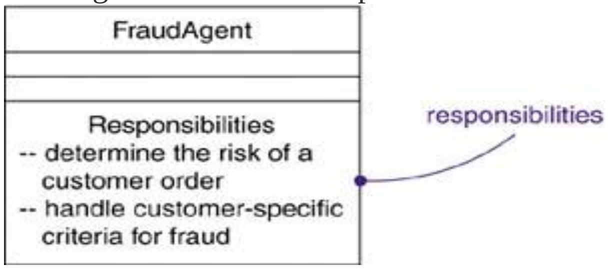
  - We can specify an operation by stating its signature, which includes the name, type, and default value of all parameters and a return type, as shown in Figure.



- **Organizing Attributes and Operations**
  - To better organize long lists of attributes and operations, you can also prefix each group with a descriptive category by using stereotypes, as shown in Figure



- **Responsibilities**
  - A responsibility is a contract or an obligation of a class.
  - When you create a class, we are making a statement that all objects of that class have the same kind of state and the same kind of behavior.
  - At a more abstract level, these corresponding attributes and operations are just the features by which the class's responsibilities are carried out.
  - A Wall class is responsible for knowing about height, width, and thickness;
  - A TemperatureSensor class is responsible for measuring temperature and raising an alarm if the temperature reaches a certain point.

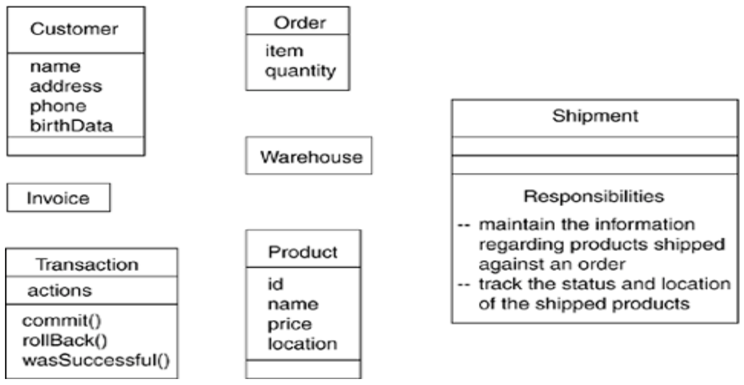

FraudAgent
Responsibilities
-- determine the risk of a customer order
-- handle customer-specific criteria for fraud
responsibilities

✓ **Common Modeling Techniques**

- **Modeling the Vocabulary of a System**
  - We use classes most commonly to model abstractions that are drawn from the problem or from the technology.
  - Each of these abstractions is a part of the vocabulary of the system; they represent the things that are important to users and to implementers.
  - For users, most abstractions are not that hard to identify.
  - They are drawn from the things that users already use to describe their system.
  - Techniques such as CRC cards and use case-based analysis are excellent ways to help users find these abstractions.
  - For implementers, these abstractions are typically just the things in the technology that are parts of the solution.

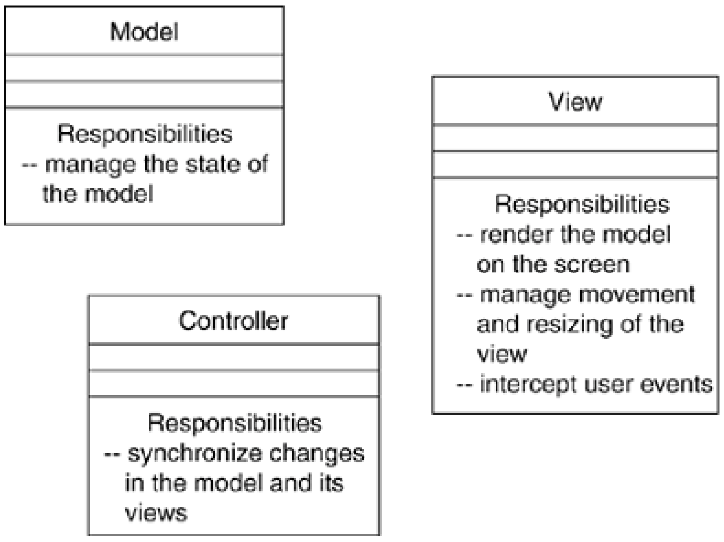
- To model the vocabulary of a system,
    - Identify those things that users or implementers use to describe the problem or solution. Use CRC cards and use case-based analysis to help find these abstractions.
    - For each abstraction, identify a set of responsibilities. Make sure that each class is crisply defined and that there is a good balance of responsibilities among all your classes.
    - Provide the attributes and operations that are needed to carry out these responsibilities for each class.

- **Example**
    - ✓ Figures show a set of classes drawn from a retail system, including Customer, Order, and Product.
    - ✓ This figure includes a few other related abstractions drawn from the vocabulary of the problem, such as Shipment (used to track orders), Invoice (used to bill orders), and Warehouse (where products are located prior to shipment). There is also one solution-related abstraction, TRansaction, which applies to orders and shipments.

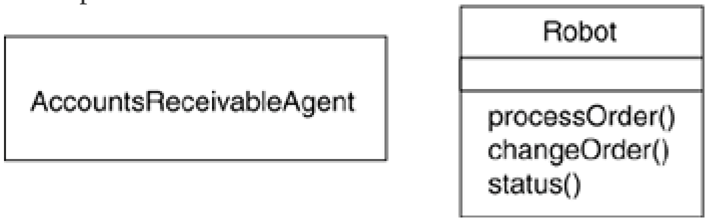


- **Modeling the Distribution of Responsibilities in a System**
    - Abstractions provide a balanced set of responsibilities. i.e. don't want any one class to be too big or too small.
        - If abstract classes that are too big, we will find that the models are hard to change and are not very reusable.
        - If abstract classes that are too small, we can reasonably manage or understand.
    - To model the distribution of responsibilities in a system,
        - Identify a set of classes that work together closely to carry out some behavior.
        - Identify a set of responsibilities for each of these classes.
        - Look at this set of classes as a whole, split classes that have too many responsibilities into smaller abstractions, collapse tiny classes that have trivial responsibilities into larger ones, and reallocate responsibilities so that each abstraction reasonably stands on its own.
        - Consider the ways in which those classes collaborate with one another, and redistribute their responsibilities accordingly so that no class within collaboration does too much or too little.

- **Example**

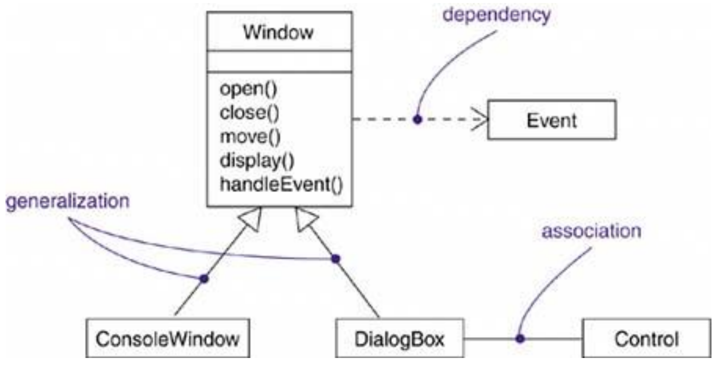

Model
Responsibilities
-- manage the state of the model
View
Responsibilities
-- render the model on the screen
-- manage movement and resizing of the view
-- intercept user events
Controller
Responsibilities
-- synchronize changes in the model and its views

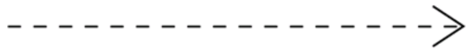
- **Modeling Nonsoftware Things**
  - Modeling abstractions are human or hardware classes
  - <u>To model nonsoftware things,</u>
    - Model the thing we are abstracting as a class.
    - If we want to distinguish these things from the UML's defined building blocks, create a new building block by using stereotypes to specify these new semantics and to give a distinctive visual cue.
    - If the thing we are modeling is some kind of hardware that itself contains software, consider modeling it as a kind of node as well, so that we can further expand on its structure.
  - Example

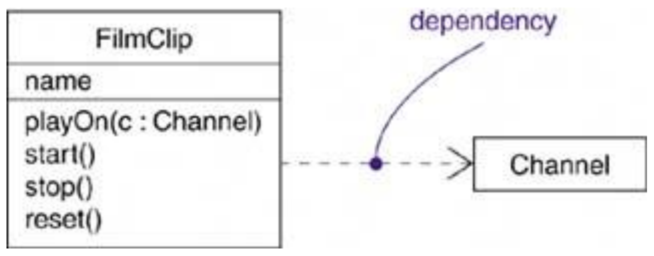


Relationships

- ✓ A relationship is a connection among things.
- ✓ In object oriented modeling, there are three kinds of relationships:
    - Dependency
    - Association
    - Generalization
- ✓ Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the kinds of relationships.
- ✓ The UML provides a graphical representation for each of these kinds of relationships, as Figures shows.

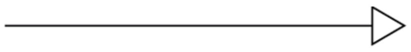


- ✓ **Terms and Concepts**
    - ✓ Dependency
        - o Dependency is a semantic relationship between two things in which a change to one thing may affect the semantics of the other thing.
        - o Graphically a dependency is rendered as a dashed line, possibly directed, and occasionally including a label.
        - o Example 
            - ✓ Most often, you will use dependencies between classes to show that one class uses operations from another class or it uses variables or arguments typed by the other class; see Figure

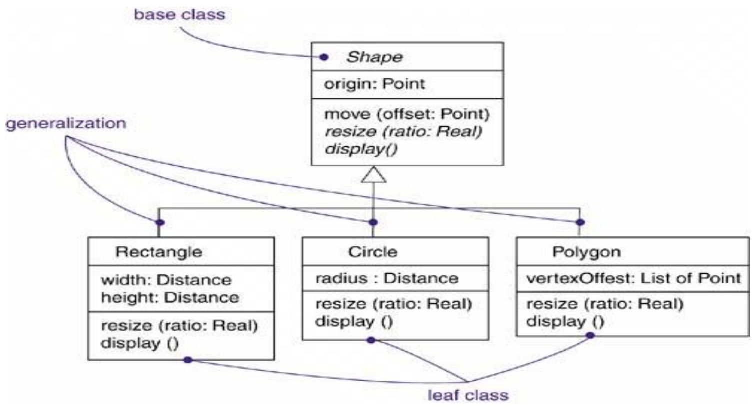


- ✓ Generalization
    - o It is a specialization/generalization relationship in which the specialized element (the child) builds on the specification of the generalized element (the parent).
    - o The child shares the structure and the behavior of the parent, but not the reverse.

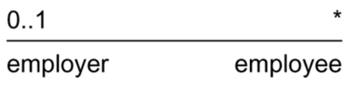
- Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent.



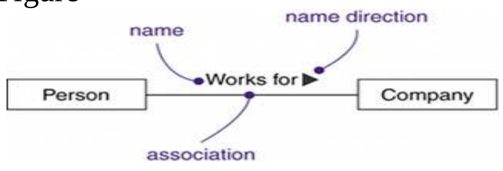
- Example



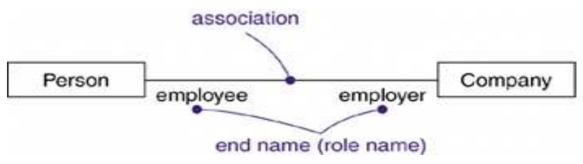
✓ <u>Association</u>

- Association is a structural relationship that describes a set of links, a link being a connection among objects.
- The link to other objects of the same class. An association that connects exactly two classes is called a binary association.
- The links to more than two classes; these are called n-ary associations.
- Graphically an association is rendered as a solid line.
    - 

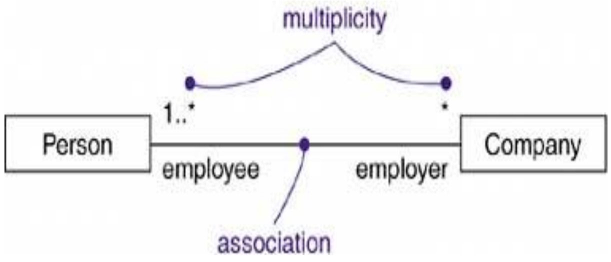
- Beyond this basic form, there are four adornments that apply to associations.
    - Name
        - An association can have a name, and we use that name to describe the nature of the relationship and we can give a direction to the name by providing a direction triangle that points in the direction we intend to read the name, as shown in Figure



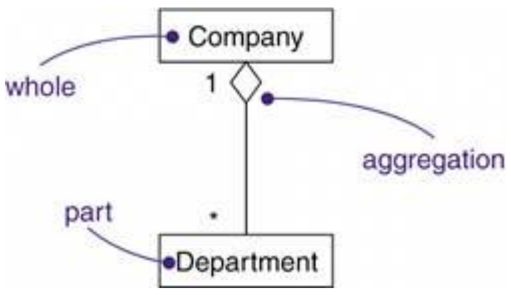
- Role
    - When a class participates in an association, it has a specific role that it plays in that relationship; a role is just the face the class at the far end of the association presents to the class at the near end of the association.



- Multiplicity
  - In many modeling situations, it's important for to state how many objects may be connected across an instance of an association. This "how many" is called the multiplicity of an association's role.
  - It represents a range of integers specifying the possible size of the set of related objects.
    - It is written as an expression with a minimum and maximum value, which may be the same; two dots are used to separate the minimum and maximum values.
      - The number of objects must be in the given range. We can show a multiplicity of exactly one (1), zero or one (0..1), many (0..*), or one or more (1..*).
  - For example, in Figure, each company object has as employee one or more person objects (multiplicity 1..*); each person object has as employer zero or more company objects (multiplicity *, which is equivalent to 0..*).



- Aggregation
  - Aggregation is a special kind of association, representing a structural relationship between a whole and its parts. Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent.

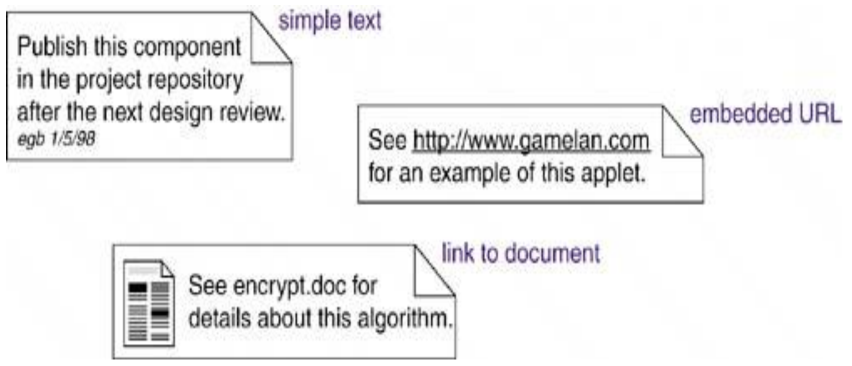


Common Mechanisms

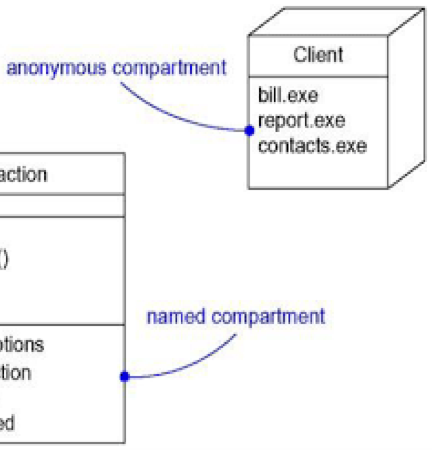
✓ UML has four common mechanisms:
- Specifications – It is a textual statement denoting the syntax and semantics.
- Adornments - textual or graphical items that are added to the elements basic notation to specify extra information.
- Common Divisions – Object-oriented systems can be divided in many ways.
- Extensibility Mechanisms - extend the capabilities of UML.

✓ **Terms and Concepts**

- Notes
  - A note is a graphical symbol for rendering constraints or comments attached to an element or a collection of elements. Graphically, a note is rendered as a rectangle with a dog-eared corner.

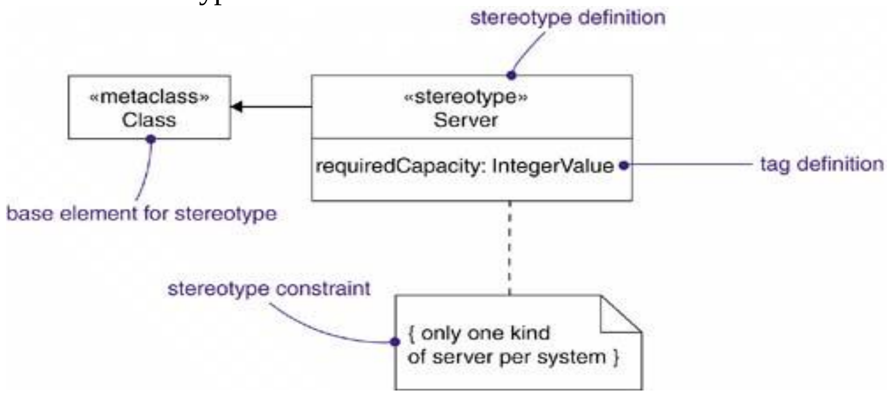

Publish this component in the project repository after the next design review.
egb 1/5/98
simple text
See http://www.gamelan.com for an example of this applet.
embedded URL
See encrypt.doc for details about this algorithm.
link to document

- Other Adornments
  - Adornments are textual or graphical items that are added to the elements basic notation to specify extra information. When using UML, always start with the basic notation of elements and add adornments to specify new information.
  - Example: adornments of an association.

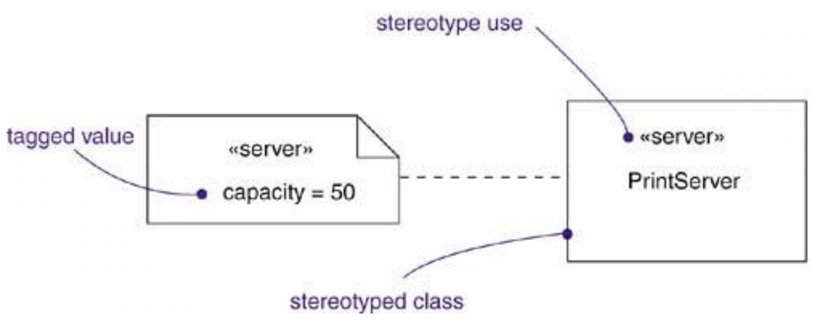


- Stereotypes:
  - It extends the vocabulary of the UML, through which new building blocks can be created out of existing ones. (or)
  - UML provides basic notations for structural things, behavioral things, grouping things and annotational things. Sometimes we may need to extend i.e., create new vocabulary and look like primitive building blocks.

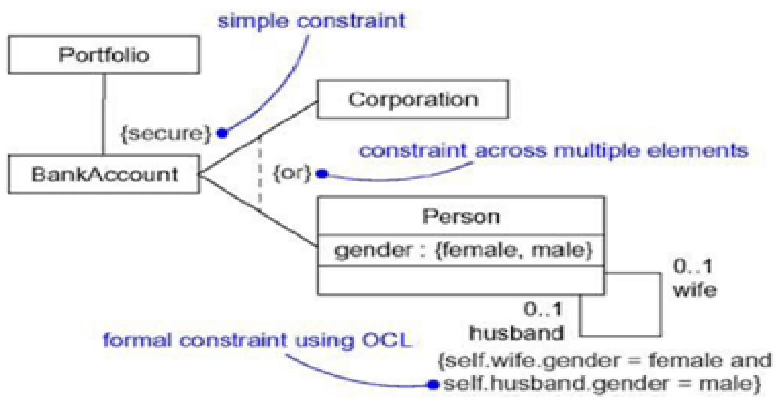
- A stereotype allows us to create new building blocks. Using stereotypes we can use the basic elements but with special properties, semantics and notation.
- Stereotypes are rendered as text inside guillemets(<< >>) or we can create new icons for stereotypes.


stereotype definition
«metaclass» Class
«stereotype» Server
requiredCapacity: IntegerValue
tag definition
base element for stereotype
stereotype constraint
{ only one kind of server per system }

- Tagged Values:
  - It extends the properties of UML building blocks. (or)
  - Every element in UML has its own properties. For example a class has its own attributes and operations.
  - Using tagged values, we can represent new properties also called as metadata.
  - These properties apply to the element itself rather than its instance.
  - Tagged values are enclosed in braces { } and are written under the element name.

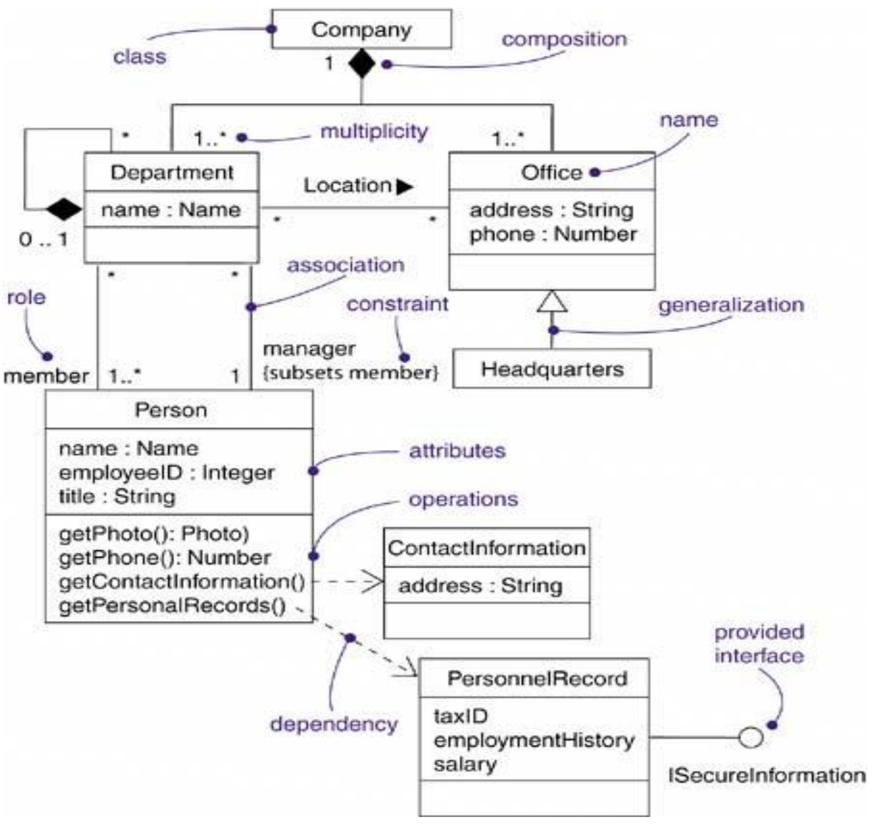


- Constraints :
  - It extends the semantics of UML building blocks. (or)
  - A constraint is used to add new semantics or change existing rules.
  - The constraints specify rules that must be followed by the elements in the model.
  - Represented as text inside braces { } and placed near to the associated element.


simple constraint
Portfolio
{secure}
BankAccount
Corporation
constraint across multiple elements
{or}
Person
gender : {female, male}
0..1
wife
0..1
husband
formal constraint using OCL
{self.wife.gender = female and self.husband.gender = male}

Class Diagrams

- ✓ Class diagrams are the most common diagram found in modeling object-oriented systems.
- ✓ A class diagram shows a set of classes, interfaces, and collaborations and their relationships.
- ✓ We use class diagrams to model the static design view of a system.
- ✓ Most part of class diagram involves modeling the vocabulary of the system, modeling collaborations, or modeling schemas.
- ✓ Class diagrams are also the foundation for component diagrams and deployment diagrams.
- ✓ Class diagrams are important not only for visualizing, specifying, and documenting structural models, but also for constructing executable systems through forward and reverse engineering.
- ✓ Example Class Diagram:



Terms and Concepts for class diagram:

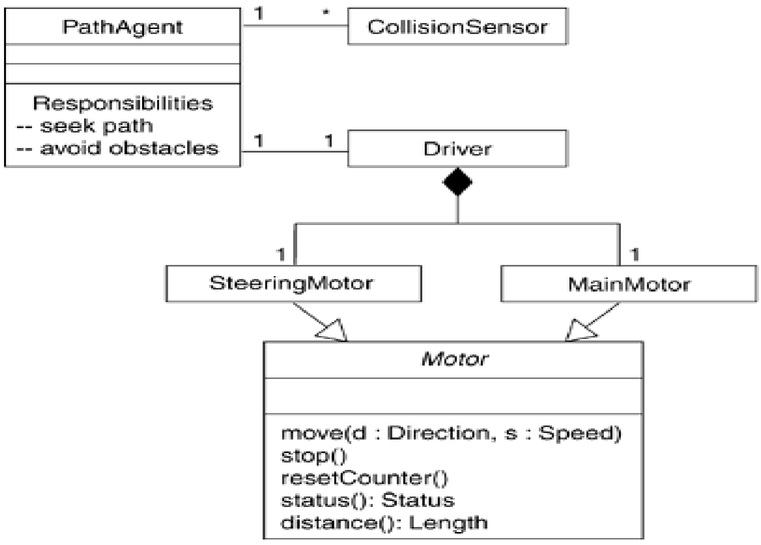
- ✓ A class diagram is a diagram that shows a set of classes, interfaces, and collaborations and their relationships.
- ✓ Graphically, a class diagram is a collection of vertices and arcs.

- ✓ **Common Properties**
  - A class diagram is just a special kind of diagram and shares the same common properties as do all other diagrams that is, a name and graphical content that are a projection into a model. What distinguishes a class diagram from other kinds of diagrams is its particular content.

- ✓ **Contents**
  - Class diagrams commonly contain the following things:
    - o Classes
    - o Interfaces
    - o Dependency, generalization, and association relationships
- ✓ **Common Uses**
  - To model the vocabulary of a system
  - To model simple collaborations
  - To model a logical database schema

<u>Common Modeling Techniques for class diagram:</u>

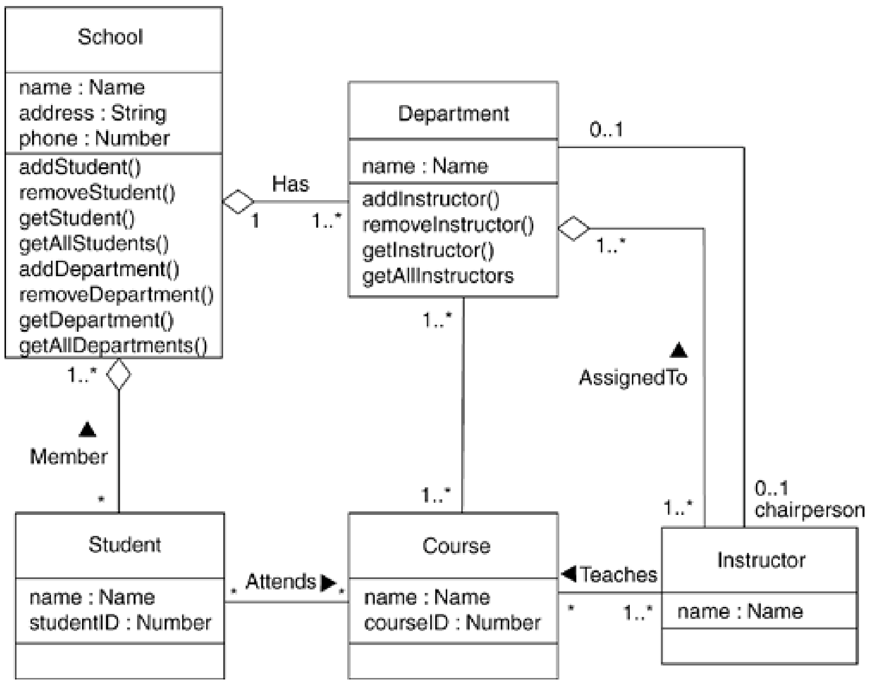
- ✓ **Modeling Simple Collaborations**
  - Identify the mechanism we had like to model.
  - For each mechanism, identify the classes, interfaces, and other collaborations that participate in this collaboration.
  - Use scenarios to walk through these things.
  - To populate these elements with their contents. For classes, start with getting a good balance of responsibilities. Then, over time, turn these into concrete attributes and operations.
  - For example,
    - o Figure shows a set of classes drawn from the implementation of an autonomous robot.
    - o The figure focuses on the classes involved in the mechanism for moving the robot along a path.
    - o We find one abstract class (Motor) with two concrete children, SteeringMotor and MainMotor. Both of these classes inherit the five operations of their parent, Motor. The two classes are, in turn, shown as parts of another class, Driver. The class PathAgent has a one-to-one association to Driver and a one-to-many association to CollisionSensor. No attributes or operations are shown for PathAgent, although its responsibilities are given.



- Class **PathAgent** collaborates with at least two other classes **Environment and GoalAgent** in a higher-level mechanism for managing the conflicting goals the robot might have at a given moment. Similarly, the classes **CollisionSensor and Driver** collaborate with another class **FaultAgent** in a mechanism responsible for continuously checking the robot's hardware for errors.

✓ **Modeling a Logical Database Schema**

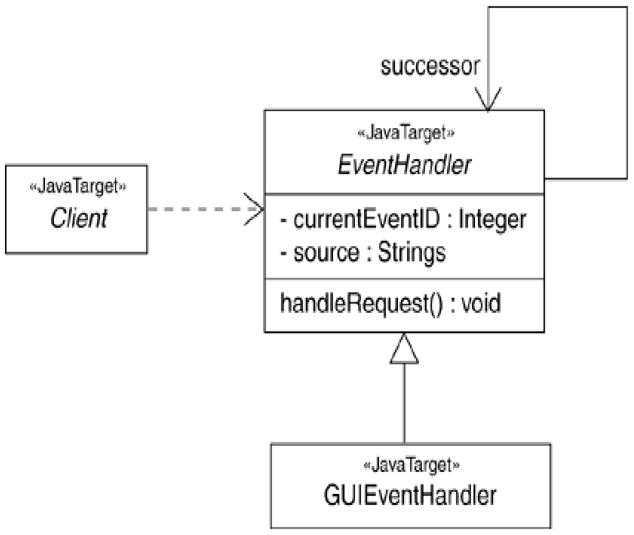
- Identify those classes in your model whose state must transcend the lifetime of their applications.
- Create a class diagram that contains these classes. We can define our own set of stereotypes and tagged values to address database-specific details.
- Expand the structural details of these classes.
- Watch for common patterns that complicate physical database design.
- Consider also the behavior of these classes by expanding operations that are important for data access and data integrity.
- Use tools to help we transform our logical design into a physical design.
- Example
    - Figure shows a set of classes drawn from an information system for a school.



✓ **Forward and Reverse Engineering**

- **Forward engineering:**
    - It is the process of transforming a model into code through a mapping to an implementation language.
    - To forward engineer a class diagram,
        - Identify the rules for mapping to your implementation language or languages of choice.

- Depending on the semantics of the languages we choose, we may want to constrain our use of certain UML features.
- Use tagged values to guide implementation choices in our target language.
- Use tools to generate code.
- Example
  - Figure shows a simple class diagram specifying an instantiation of the chain of responsibility pattern.



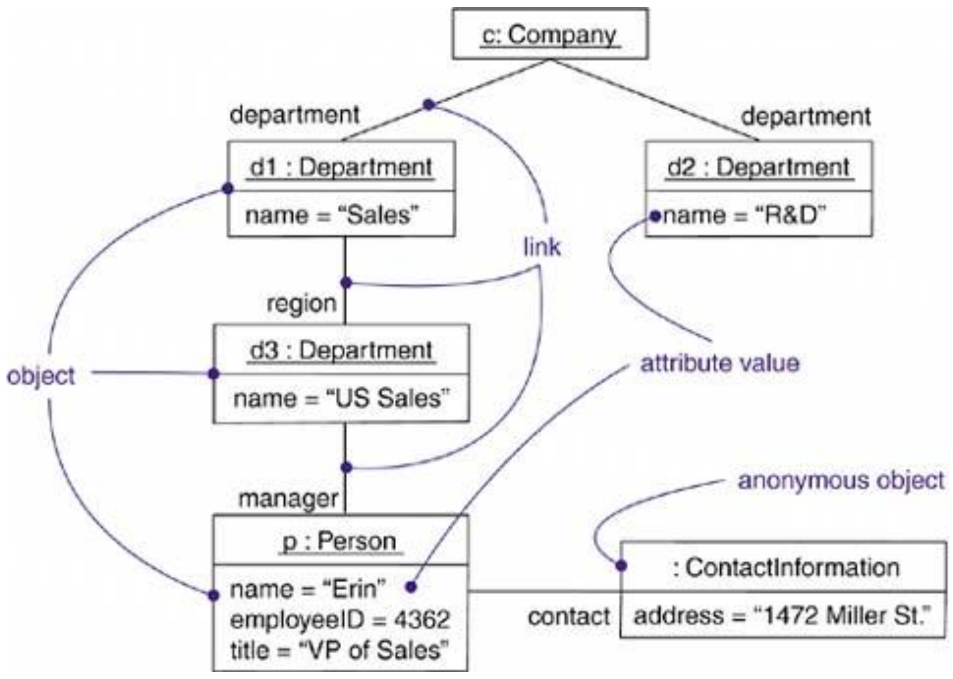
  - Forward engineering the class EventHandler yields the following code.

```
public abstract class EventHandler {
        EventHandler successor;
        private Integer currentEventID;
        private String source;
        EventHandler() {}
        public void handleRequest() {}
}
```

- **Reverse engineering:**
  - It is the process of transforming code into a model through a mapping from a specific implementation language.
  - To reverse engineer a class diagram,
    - Identify the rules for mapping from your implementation language or languages of choice.
    - Using a tool, point to the code we had like to reverse engineer. Use your tool to generate a new model or modify an existing one that was previously forward engineered.
    - Using our tool, create a class diagram by querying the model.
    - Manually add design information to the model to express the intent of the design that is missing or hidden in the code.

Object Diagrams

- ✓ Object diagrams model the instances of things contained in class diagrams.
- ✓ An object diagram shows a set of objects and their relationships at a point in time.
- ✓ We use object diagrams to model the static design view or static process view of a system. This involves modeling a snapshot of the system at a moment in time and rendering a set of objects, their state, and their relationships.
- ✓ Object diagrams are not only important for visualizing, specifying, and documenting structural models, but also for constructing the static aspects of systems through forward and reverse engineering.
- ✓ Example Object Diagram:



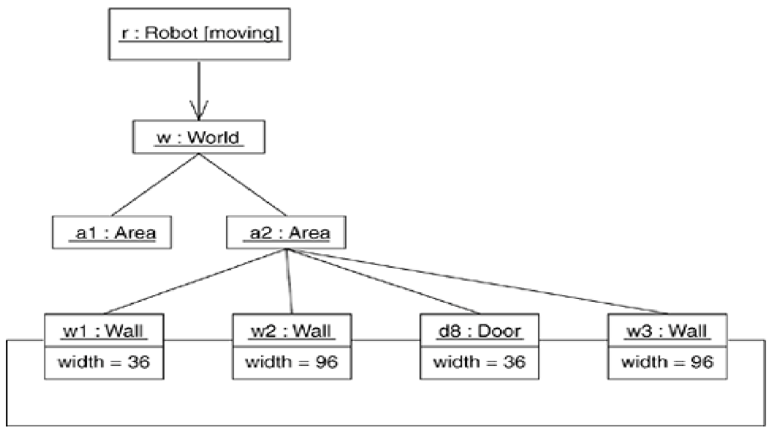
Terms and Concepts for object diagram:

- ✓ An object diagram is a diagram that shows a set of objects and their relationships at a point in time.
- ✓ Graphically, an object diagram is a collection of vertices and arcs.
- ✓ **Common Properties**
  - An object diagram is a special kind of diagram and shares the same common properties as all other diagrams that is, a name and graphical contents that are a projection into a model. What distinguishes an object diagram from all other kinds of diagrams is its particular content.
- ✓ **Contents**
  - Object diagrams commonly contain
    - Objects
    - Links
- ✓ **Common Uses**
  - To model object structures.

Common Modeling Techniques for class diagram:

✓ **Modeling Object Structures**

- Identify the mechanism we had like to model.
- Create a collaboration to describe a mechanism.
- For each mechanism, identify the classes, interfaces, and other elements that participate in this collaboration; identify the relationships among these things.
- Consider one scenario that walks through this mechanism.
- Expose the state and attribute values of each such object, as necessary, to understand the scenario.
- Similarly, expose the links among these objects, representing instances of associations among them.
- For example,
  - o The figure shows a set of objects drawn from the implementation of an autonomous robot.
  - o This figure focuses on some of the objects involved in the mechanism used by the robot to calculate a model of the world in which it moves.
  - o There are many more objects involved in a running system, but this diagram focuses on only those abstractions that are directly involved in creating this world view.



✓ **Reverse Engineering**

- It is the process of transforming code into a model through a mapping from a specific implementation language.
- To reverse engineer a object diagram,
  - o Chose the target we want to reverse engineer.
  - o Using a tool or simply walking through a scenario, stop execution at a certain moment in time.
  - o Identify the set of interesting objects that collaborate in that context and render them in an object diagram.
  - o As necessary to understand their semantics, expose these object's states.
  - o As necessary to understand their semantics, identify the links that exist among these objects.
  - o We will usually have to manually add or label structure that is not explicit in the target code. The missing information supplies the design intent that is only implicit in the final code.

Syllabus: Unit – III: Structural And Behavioral Modeling
Advance Classes - Advanced Relationships - Interfaces - Types &Roles - Packages - Interactions - Usecases - Usecase diagrams.

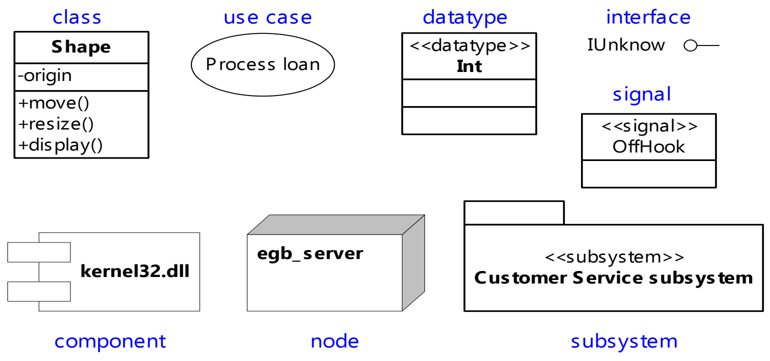
Advance Classes

✓ **Class**

- The fundamental building block in a object-oriented system is an object or class.
- However, in UML,
  - Class is not the only general building block.
  - It is only one of the general building blocks in UML, called as classifiers.

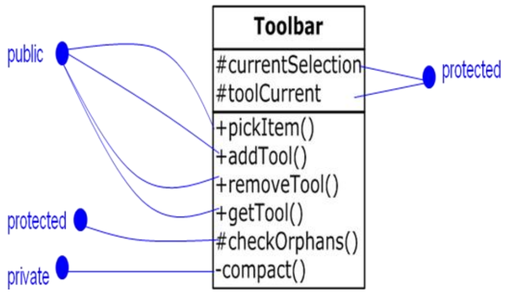
✓ **Classifier**

- A classifier is a mechanism which describes structural and behavioral features.
  - Class is the frequently used classifier.
  - Every classifier represents structural aspects in terms of properties and behavioral aspects in terms of operations.
  - Beyond these basic features, there are several advanced features like multiplicity, visibility, signatures, polymorphism and others.
- In general, all the modeling elements that can have instances are called classifiers.
- Class, Instance, Datatype, Signal, Component, Node, Use case, Subsystem are classifiers. (packages are not.)

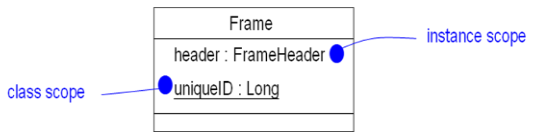


✓ **Special properties of attributes and operations**

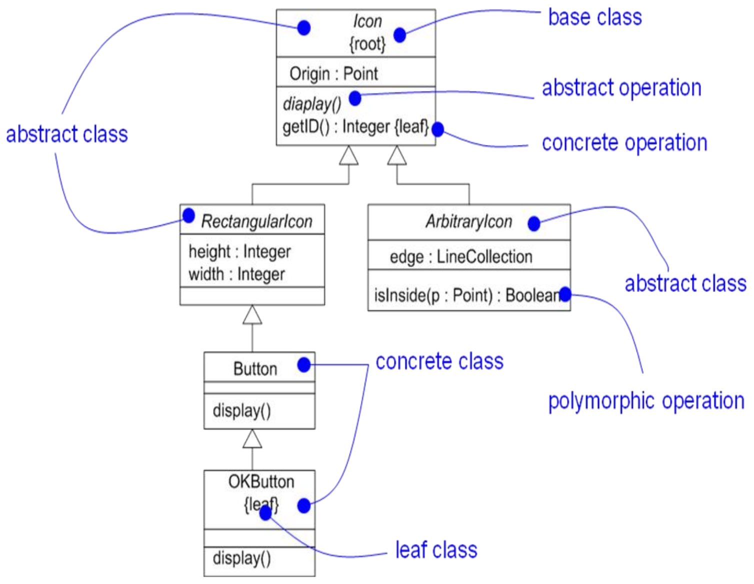
- Visibility
  - Public[+]: any outside classifier with visibility to the given classifier can use this feature.
  - Protected[#]: any descendant of the classifier can use the feature.
  - Private[-]: only the classifier itself can use the feature.



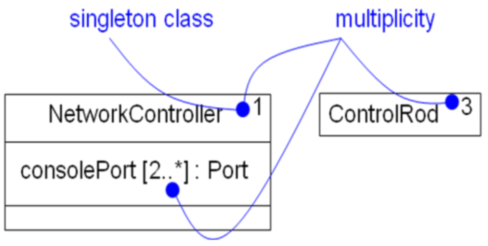
- Scope
  - The owner scope of a feature specifies whether the feature appears in each instance of the classifier or whether there is just a single instance of feature for all instances of the classifier.
    - ✓ Instance: each instance holds its own value.
    - ✓ Classifier: just one value for all instances. [static]



- Abstract, Root, Leaf and Polymorphic Elements

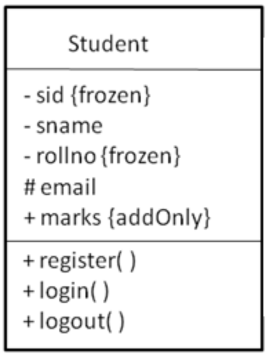


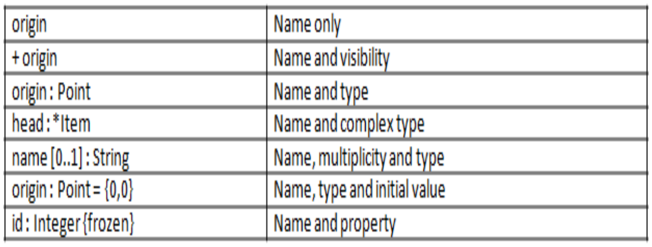
- Multiplicity
  - It's reasonable to assume that there may be any number of instances of classes.
  - The number of instances a class may have is called multiplicity.



- Attributes
  - The syntax of an attribute in the UML is
    [visibility] name [multiplicity][: type][= initial-value][{property-string }]
  - There are three defined properties
    - ✓ changeable : no restrictions on modifying the attribute's value

- ✓ addOnly : additional value may be added for attributes with a multiplicity > 1, but once created, a value may not be removed or altered.
- ✓ frozen : the attribute's value may not be changed after object is initialized. [const]
- ✓ Example:





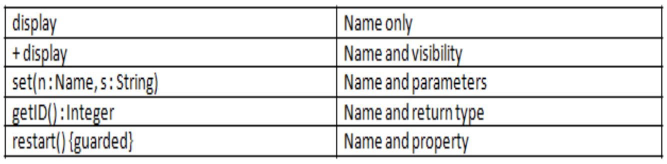
origin	Name only
+ origin	Name and visibility
origin : Point	Name and type
head : *Item	Name and complex type
name [0..1] : String	Name, multiplicity and type
origin : Point = {0,0}	Name, type and initial value
id : Integer {frozen}	Name and property

- Operations
  - The syntax of an operation in UML is
    [visibility] name [(parameter-list)][: return-type][{ property-string}]

    [direction] name : type [= default-value]

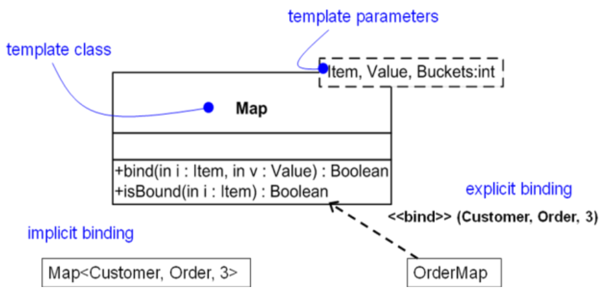
    in, out, inout : means parameter may be modified or not.

  - There are five defined properties
    - ✓ leaf : may not be overridden
    - ✓ isQuery : leave the state of subsystem unchanged.
    - ✓ sequential : only one flow is in the object at a time.
    - ✓ guarded : sequentializing all calls.
    - ✓ concurrent : treating the operation as atomic.
      - ➢ Note : 3. 4. 5. are for concurrence.
    - ✓ Example:



display	Name only
+ display	Name and visibility
set(n : Name, s : String)	Name and parameters
getID() : Integer	Name and return type
restart() {guarded}	Name and property

- Template Classes
  - Like template classes in C++ and Ada.
  - Cannot use a template directly; you have to instantiate it first.



```
template<class Item, class value, int Buckets>
Class Map
{
public:
        virtual Boolean bind(const Items, const Values&);
        virtual Boolean isBound(const Item&) const;
….
} // explicit binding
M: Map<Customer, Order, 3>; // implicit binding
```

- Standard elements
    - Metaclass : specify the classifier whose objects are all classes.
    - Powertype : whose objects are the children of a given parent.
    - Stereotype : may be applied to other elements.
    - Utility : whose attributes and operations are all class scoped.

Advanced Relationships

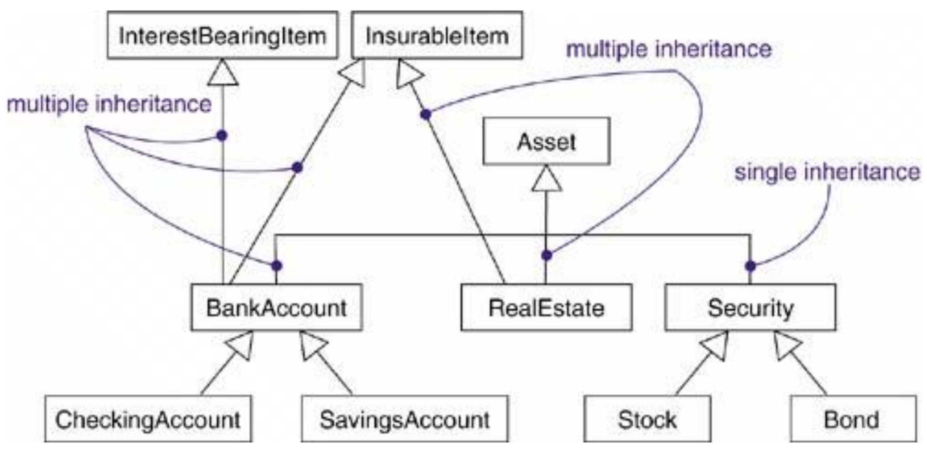
- ✓ A relationship is a connection among things. There are four most important relationships in object-oriented modeling:
    - Dependencies
    - Generalizations
    - Associations
    - Realizations

- ✓ **Dependency**
    - Specifying a change in the specification of one thing may affect another thing, but not necessarily the reverse.
    - Rendering as a dashed line [- - - - - ►]
    - UML defines a number of stereotypes.
    - There are eight stereotypes that apply to dependency relationships among classes and objects in class diagrams.
        - **«bind»** – Specifies that the source instantiates the target template using the given actual parameters.
        - **«derive»** – Specifies that the source may be computed from the target.
        - **«friend»** – Specifies that the source is given special visibility into the target.
        - **«instanceOf»** – Specifies that the source object is an instance of the target classifier.
        - **«instantiate»** – Specifies that the source creates instances of the target.
        - **«powertype»** – Specifies that the target is a powertype of the source; a powertype is a classifier whose objects are all the children of a given parent.
        - **«refine»** – Specifies that the source is at a finer degree of abstraction than the target.
        - **«use»** – Specifies that the semantics of the source element depends on the semantics of the public part of the target.

- There are two stereotypes that apply to dependency relationships among packages.
  - **«access»** – Specifies that the source package is granted the right to reference the elements of the target package.
  - **«import»** – A kind of access that specifies that the public contents of the target package enter the flat namespace of the source, as if they had been declared in the source.
- Two stereotypes apply to dependency relationships among use cases:
  - **«extend» –** Specifies that the target use case extends the behavior of the source.
  - **«include» –** Specifies that the source use case explicitly incorporates the behavior of another use case at a location specified by the source.
- There are three stereotypes when modeling interactions among objects.
  - **«become»** – Specifies that the target is the same object as the source but at a later point in time and with possibly different values, state, or roles.
  - **«call»** – Specifies that the source operation invokes the target operation.
  - **«copy»** – Specifies that the target object is an exact, but independent, copy of the source.
- There is one stereotype when modeling state machine among objects.
  - **«send»** – Specifies that the source operation sends the target event
- There is one stereotype when modeling system into sub system among objects.
  - **«trace»** – Specifies that the target is an historical ancestor of the source.

✓ **Generalizations**

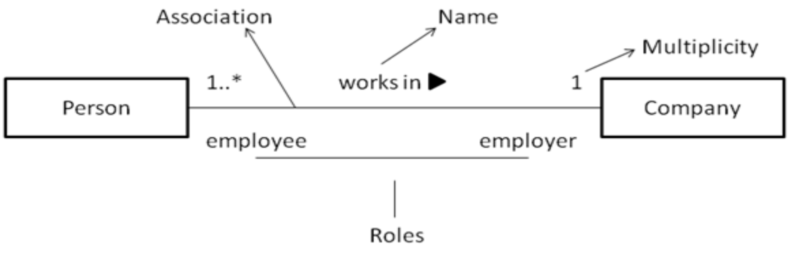
- A generalization relationship represents generalization-specialization relationship between classes.
- The class with the general structure and behavior is known as the parent or superclass and the class with specific structure and behavior is known as the child or subclass.
- Rendering as a dashed line [———►]
- Consider the below class hierarchy:



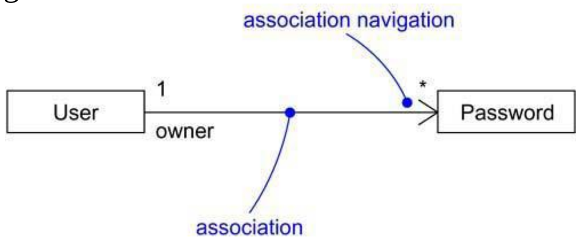
- UML defines **one stereotype and four constraints** that may be applied to generalization relationships.
    - **«implementation»** – Specifies that the child inherits the implementation of the parent but does not make public nor support its interfaces, thereby violating substitutability.
        - ✓ **{complete}** – Specifies that all children in the generalization have been specified in the model and that no additional children are permitted.
        - ✓ **{incomplete}** – Specifies that not all children in the generalization have been specified and that additional children are permitted.
        - ✓ **{disjoint}** – Specifies that objects of the parent may have no more than one of the children as a type.
        - ✓ **{overlapping}** – Specifies that objects of the parent may have more than one of the children as a type.

✓ **Associations**

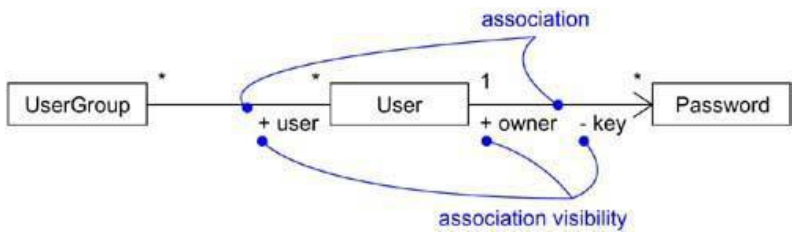
- Association is a structural relationship which denotes a connection between two or more things.
- The association relationship can represent either physical or logical connections between things.
- Rendering as a dashed line [————]
- The four basic adornments for an association relationship are: name, role at each end of the association, multiplicity at each end of the association and aggregation.



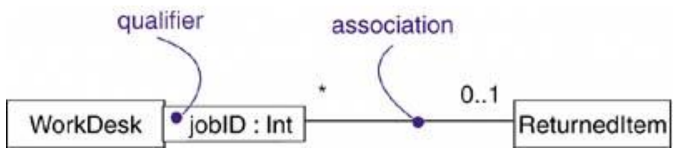
- Over these basic features, there are other advanced features like: navigation, visibility, qualification, composition and association classes.
    - **Navigation: Unless** otherwise specified, navigation across an association is bidirectional. Unidirectional navigation is also possible where reverse navigation is not desirable such as where security concern is an important thing. A Unidirectional navigation is shown in Figure



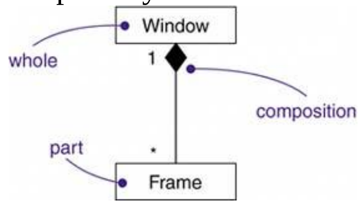
- **Visibility:** Objects at that end are not accessible to any objects outside the association
  - ✓ Three levels of visibility for an association possible in UML.
    - ➢ Public visibility indicates that objects can access any one. (default)
    - ➢ Private visibility indicates that objects at that end are not accessible to any objects outside the association;
    - ➢ Protected visibility indicates that objects at that end are not accessible to any objects outside the association, except for children of the other end.



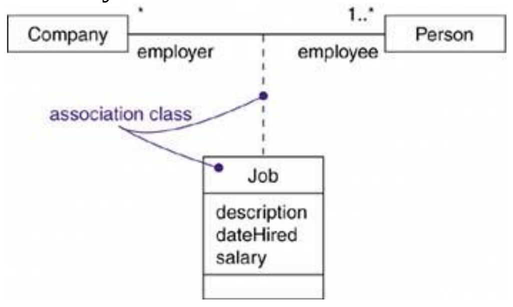
- **Qualification:** This is an attribute whose values partition the set of objects related to an object across an association.



- **Composition:** A form of an aggregation with strong ownership and coincident lifetime of the parts by the whole



- **Association Classes:** In an association between two classes, the association itself might have properties. An association class can be seen as an association that also has class properties, or as a class that also has association properties. We render an association class as a class symbol attached by a dashed line to an association.

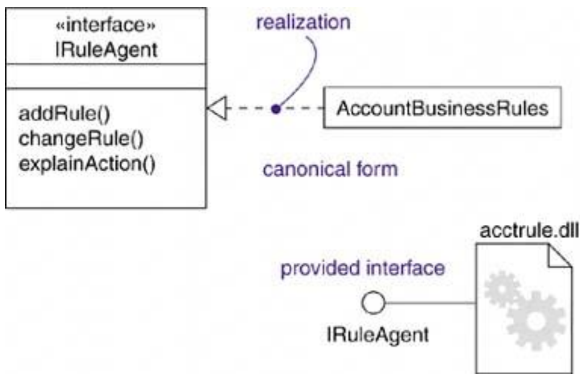


- ✓ **Five constraints** that can be applied to association relationships.
  - ➢ **{implicit}** – Specifies that the relationship is not manifest but, rather, is only conceptual.
  - ➢ **{ordered}** – Specifies that the set of objects at one end of an association are in an explicit order.
- ✓ Constraints that **relate to the changeability of the instances** of an association.
  - ➢ **{changeable}** – Links between objects may be added, removed, and changed freely.
  - ➢ **{addOnly}** – New links may be added from an object on the opposite end of the association.
  - ➢ **{frozen}** – A link, once added from an object on the opposite end of the association, may not be modified or deleted.
- ✓ Constraint **for managing related sets** of associations.
  - ➢ **{xor}** – Specifies that, over a set of associations, exactly one is manifest for each associated object.

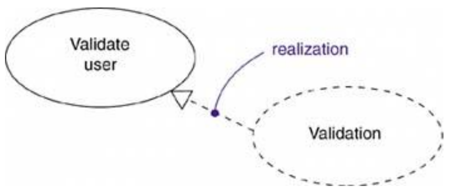
✓ **Realizations**

- A realization is a semantic relationship between classifiers in which one classifier specifies a contract that another classifier guarantees to carry out
- Used in two circumstances
  - ▪ In the context of interfaces
  - ▪ In the context of collaborations
- Graphically rendered as dashed directed line with a large open arrowhead [- - - - - - - ▷]

  - ▪ Realization of an Interface

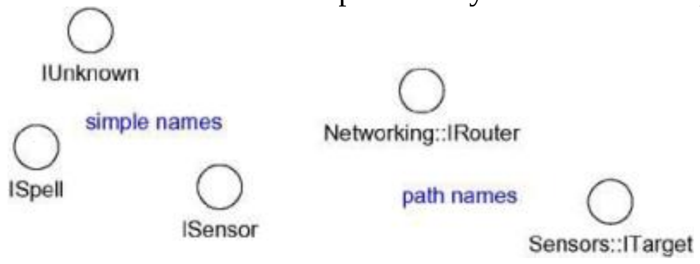


  - ▪ Realization of a Use Case

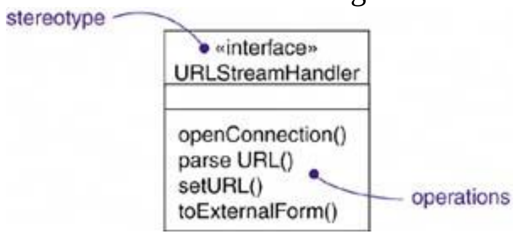


Interfaces - Types &Roles

- ✓ Interface
  - An interface is a collection of operations that are used to specify a service of a class or a component.
- ✓ Type
  - A type is a stereotype of a class used to specify a domain of objects, together with the operations (but not the methods) applicable to the object.
- ✓ Role
  - A role is the behavior of an entity participating in a particular context. An interface may be rendered as a stereotyped class in order to expose its operations and other properties.
- ✓ Names
  - Every interface must have a name that distinguishes it from other interfaces.
  - A name is a textual string. That name alone is known as a simple name;
  - A path name is the interface name prefixed by the name of the package.

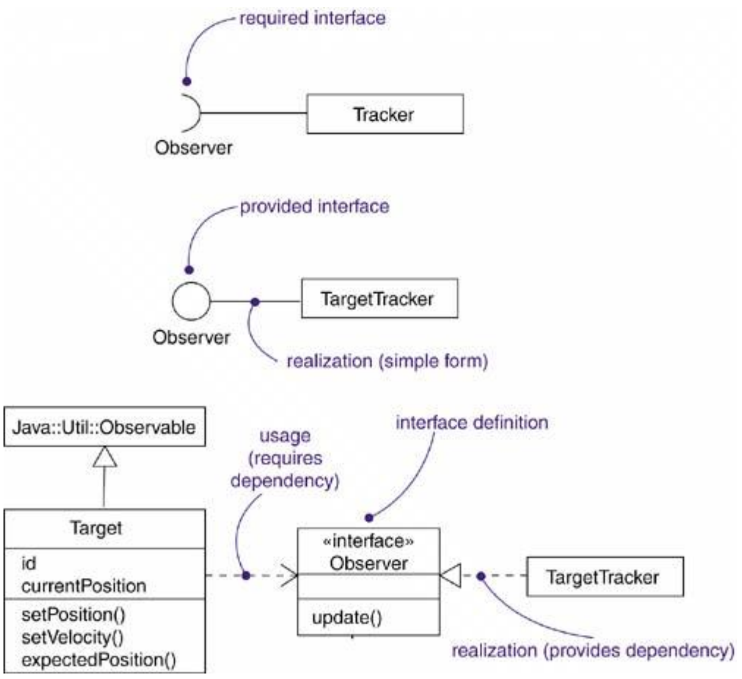


- ✓ Operations
  - An interface is a named collection of operations used to specify a service of a class or of a component.
  - Unlike classes or types, interfaces do not specify any structure, nor do they specify any implementation
  - These operations may be adorned with visibility properties, concurrency properties, stereotypes, tagged values, and constraints.
  - We can render an interface as a stereotyped class, listing its operations in the appropriate compartment. Operations may be drawn showing only their name, or they may be augmented to show their full signature and other properties.



- ✓ Relationships
  - Like a class, an interface may participate in generalization, association, and dependency relationships. In addition, an interface may participate in realization relationships.
  - An interface specifies a contract for a class or a component without dictating its implementation. A class or component may realize many interfaces.
  - We can show that an element realizes an interface in two ways.
    - ▪ First, you can use the simple form in which the interface and its realization relationship are rendered as a lollipop sticking off to one side of a class or component.

- Second, you can use the expanded form in which you render an interface as a stereotyped class, which allows you to visualize its operations and other properties, and then draw a realization relationship from the classifier or component to the interface.



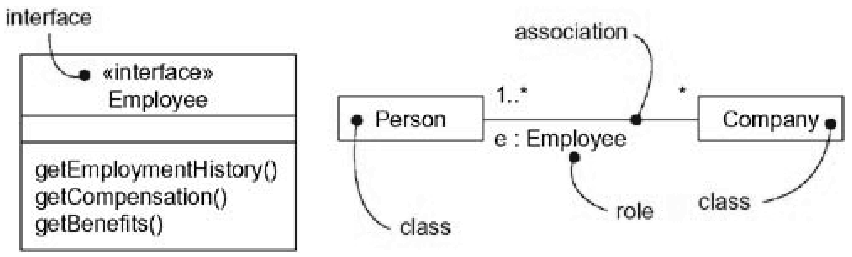
✓ Understanding an Interface

- In the UML, you can supply much more information to an interface in order to make it understandable and approachable.
- First, you may attach pre- and postconditions to each operation and invariants to the class or component as a whole. By doing this, a client who needs to use an interface will be able to understand what the interface does and how to use it, without having to dive into an implementation.
- We can attach a state machine to the interface. You can use this state machine to specify the legal partial ordering of an interface's operations.
- We can attach collaborations to the interface. You can use collaborations to specify the expected behavior of the interface through a series of interaction diagrams.

✓ **Types and Roles**

- A role names a behavior of an entity participating in a particular context. Stated another way, a role is the face that an abstraction presents to the world.
- For example, consider an instance of the class Person. Depending on the context, that Person instance may play the role of Mother, Comforter, PayerOfBills, Employee, Customer, Manager, Pilot, Singer, and so on.
- When an object plays a particular role, it presents a face to the world, and clients that interact with it expect a certain behavior depending on the role that it plays at the time.
- An instance of Person in the role of Manager would present a different set of properties than if the instance were playing the role of Mother.

- In the UML, you can specify a role an abstraction presents to another abstraction by adorning the name of an association end with a specific interface.

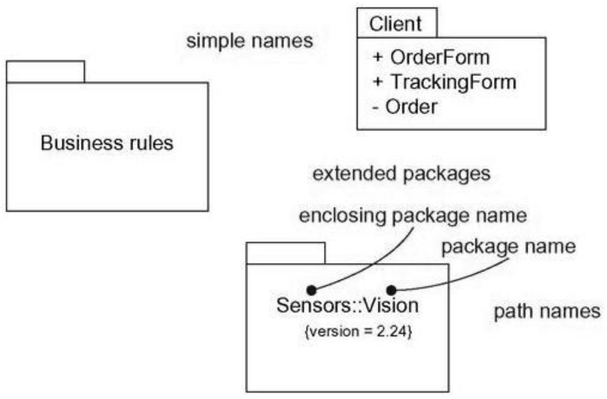


✓ Common Modeling Techniques
- Modeling the Seams in a System modeling the Seams in a System
- Modeling Static and Dynamic Types modeling Static and Dynamic Types

Packages

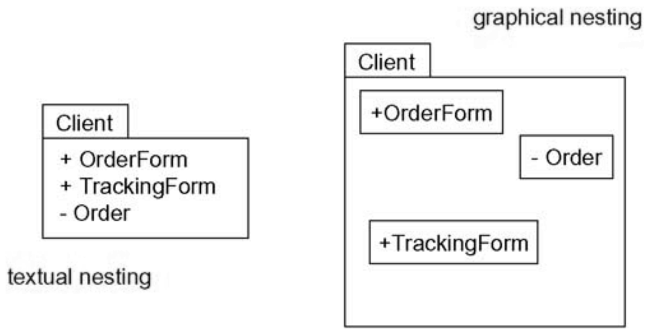
✓ A package is a general-purpose mechanism for organizing elements into groups.
✓ Graphically, a package is rendered as a tabbed folder.

✓ **Names**
- Every package must have a name that distinguishes it from other packages. A name is a textual string. That name alone is known as a simple name; a path name is the package name prefixed by the name of the package in which that package lives
- We may draw packages adorned with tagged values or with additional compartments to expose their details.



✓ **Owned Elements**
- A package may own other elements, including classes, interfaces, components, nodes, collaborations, use cases, diagrams, and even other packages.
- Owning is a composite relationship, which means that the element is declared in the package. If the package is destroyed, the element is destroyed. Every element is uniquely owned by exactly one package.
- Elements of different kinds may have the same name within a package. Thus, you can have a class named Timer, as well as a component named Timer, within the same package.
- Packages may own other packages. This means that it's possible to decompose your models hierarchically.

- We can explicitly show the contents of a package either textually or graphically.

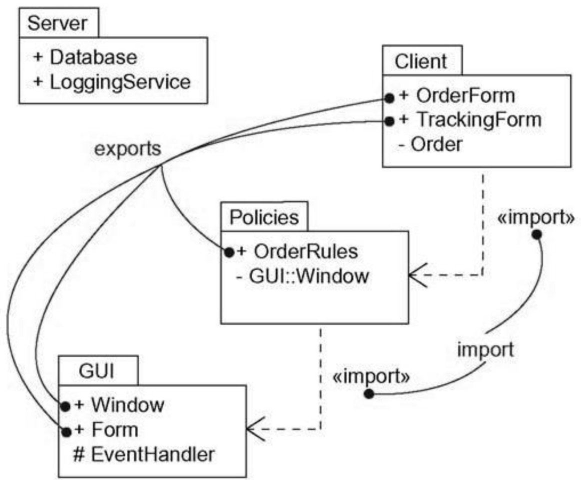


✓ **Visibility**

- We can control the visibility of the elements owned by a package just as you can control the visibility of the attributes and operations owned by a class.
- Typically, an element owned by a package is public, which means that it is visible to the contents of any package that imports the element's enclosing package.
- Conversely, protected elements can only be seen by children, and private elements cannot be seen outside the package in which they are declared.
- We specify the visibility of an element owned by a package by prefixing the element's name with an appropriate visibility symbol.

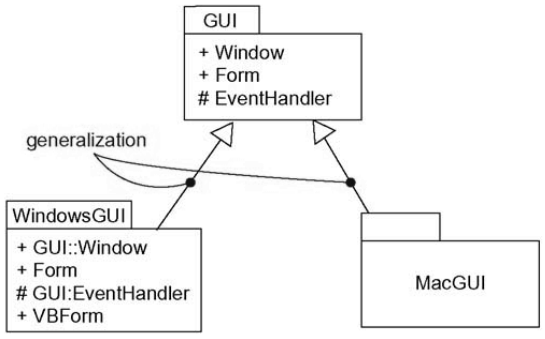
✓ **Importing and Exporting**

- In the UML, you model an import relationship as a dependency adorned with the stereotype import.
- Actually, two stereotypes apply here
  - **import and access** and both specify that the source package has access to the contents of the target.
    - ✓ Import adds the contents of the target to the source's namespace
    - ✓ Access does not add the contents of the target
- The public parts of a package are called its exports.
- The parts that one package exports are visible only to the contents of those packages that explicitly import the package.
- Import and access dependencies are not transitive.



✓ **Generalization**

- There are two kinds of relationships you can have between packages: import and access dependencies used to import into one package elements exported from another and generalizations, used to specify families of packages
- Generalization among packages is very much like generalization among classes
- Packages involved in generalization relationships follow the same principle of substitutability as do classes. A specialized package (such as Windows GUI) can be used anywhere a more general package (such as GUI) is used.



✓ **Standard Elements**

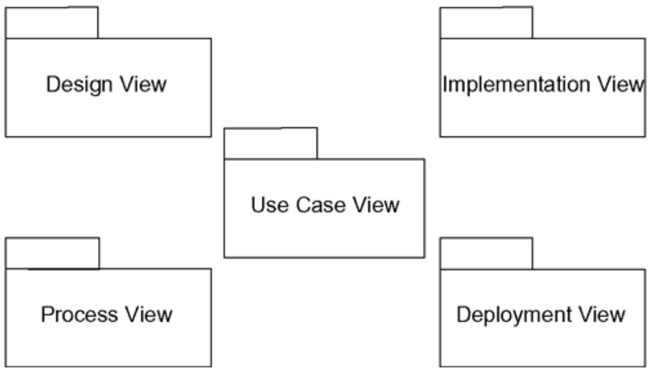
- All of the UML's extensibility mechanisms apply to packages. Most often, you'll use tagged values to add new package properties (such as specifying the author of a package) and stereotypes to specify new kinds of packages (such as packages that encapsulate operating system services).
- The UML defines five standard stereotypes that apply to packages:
  - facade - Specifies a package that is only a view on some other package.
  - Framework - Specifies a package consisting mainly of patterns.
  - Stub - Specifies a package that serves as a proxy for the public contents of another package.
  - Subsystem - Specifies a package representing an independent part of the entire system being modelled.
  - System - Specifies a package representing the entire system being modelled.

✓ **<u>Common Modeling Techniques</u>**

- **Modeling Groups of Elements**
  - The most common purpose for which you'll use packages is to organize modeling elements into groups that you can name and manipulate as a set.
  - There is one important distinction between classes and packages:
  - Packages have no identity (meaning that you can't have instances of packages, so they are invisible in the running system);
  - Classes do have identity (classes have instances, which are elements of a running system).
  - To model groups of elements,
    - ✓ Scan the modeling elements in a particular architectural view and look for clumps defined by elements that are conceptually or semantically close to one another.
    - ✓ Surround each of these clumps in a package.
    - ✓ For each package, distinguish which elements should be accessible outside the package. Mark them public, and all others protected or private. When in doubt, hide the element.

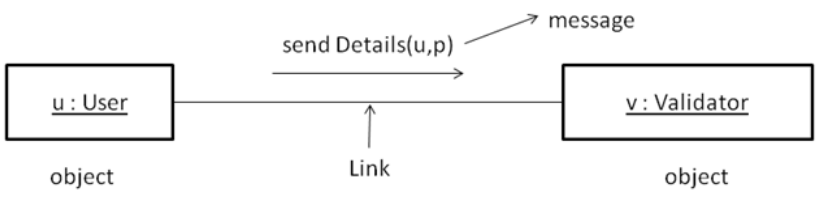
- ✓ Explicitly connect packages that build on others via import dependencies
- ✓ In the case of families of packages, connect specialized packages to their more general part via generalizations

- **Modeling Architectural Views**
  - We can use packages to model the views of architecture.
  - Remember that a view is a projection into the organization and structure of a system, focused on a particular aspect of that system.
  - This definition has two implications. First, you can decompose a system into almost orthogonal packages, each of which addresses a set of architecturally significant decisions.(design view, a process view, an implementation view, a deployment view, and a use case view)
  - Second, these packages own all the abstractions germane to that view.(Implementation view)
  - To model architectural views,
    - ✓ Identify the set of architectural views that are significant in the context of your problem. In practice, this typically includes a design view, a process view, an implementation view, a deployment view, and a use case view.
    - ✓ Place the elements (and diagrams) that are necessary and sufficient to visualize, specify, construct, and document the semantics of each view into the appropriate package.
    - ✓ As necessary, further group these elements into their own packages.
    - ✓ There will typically be dependencies across the elements in different views. So, in general, let each view at the top of a system be open to all others at that level.

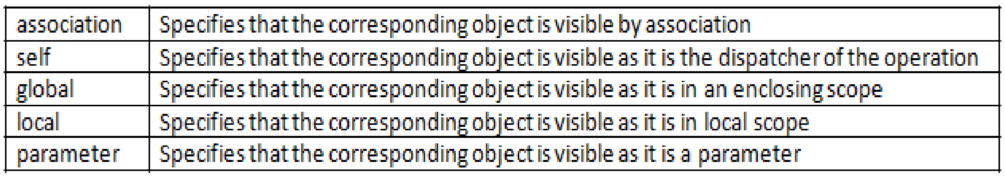


Interactions

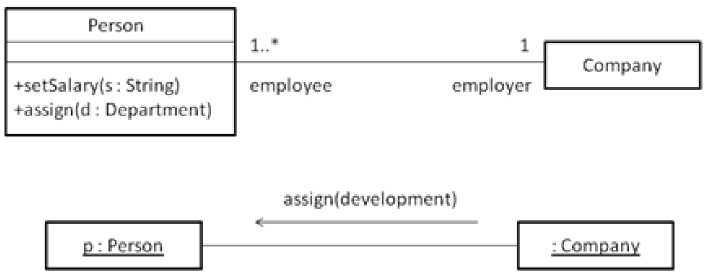
- ✓ An interaction is a behavior that contains a set of messages exchanged among a set of objects within a context to accomplish a purpose. A message is specification of a communication between objects that conveys information with the expectation that the activity will succeed.
- ✓ In UML, the dynamic aspects of a system can be modeled using interactions. Interactions contain messages that are exchanged between objects.
- ✓ A message can be an invocation of an operation or a signal. The messages may also include creation and destruction of other objects.
- ✓ We can use interactions to model the flow of control within an operation, a class, a component, a use case or the system as a whole.
- ✓ Using interaction diagrams, we can model these flows in two ways: one is by focusing on how the messages are dispatched across time and the second is by focusing on the structural relationships between objects and then consider how the messages are passed between the objects.
- ✓ Graphically a message is rendered as a directed line with the name of its operation as show below:



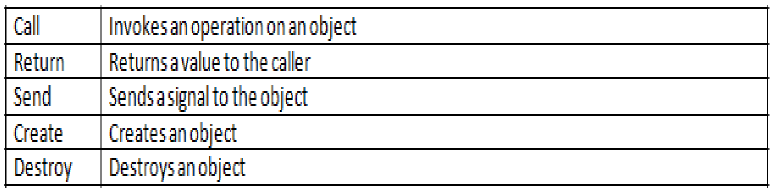
- ✓ **Objects and Roles**
  - The objects that participate in an interaction are either concrete things or prototypical things. As a concrete thing, an object represents something in the real world. For example, p an instance of the class Person, might denote a particular human. Alternately, as a prototypical thing, p might represent any instance of Person.
- ✓ **Links**
  - A link is a semantic connection among objects. In general, a link is an instance of association. Wherever, a class has an association with another class, there may be a link between the instances of the two classes. Wherever there is a link between two objects, one object can send messages to another object. We can adorn the appropriate end of the link with any of the following standard stereotypes:



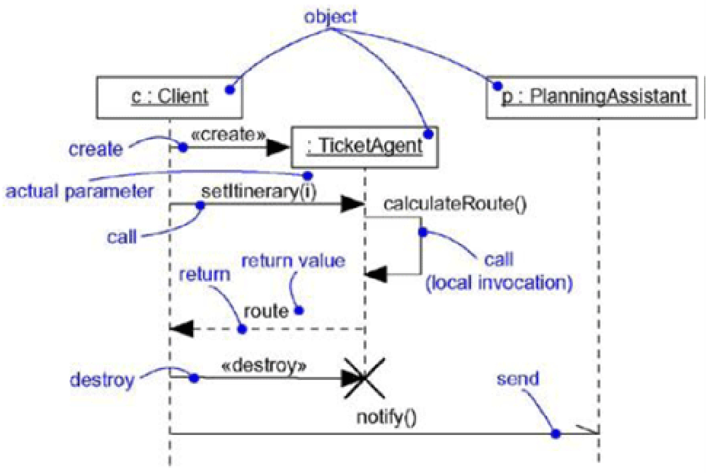
association	Specifies that the corresponding object is visible by association
self	Specifies that the corresponding object is visible as it is the dispatcher of the operation
global	Specifies that the corresponding object is visible as it is in an enclosing scope
local	Specifies that the corresponding object is visible as it is in local scope
parameter	Specifies that the corresponding object is visible as it is a parameter



- **Messages**
  - A message is the specification of communication among objects that conveys information with the expectation that activity will succeed. The receipt of a message instance may be considered an instance of an event.
  - When a message is passed, the action that results is an executable statement that forms an abstraction of a computational procedure. An action may result in a change of state. In UML, we can model several kinds of actions like:

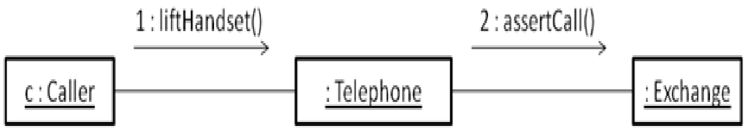


Call	Invokes an operation on an object
Return	Returns a value to the caller
Send	Sends a signal to the object
Create	Creates an object
Destroy	Destroys an object



- **Sequencing**
  - When an object passes a message to another object, the receiving object might in turn send a message to another object, which might send a message to yet a different object and so on.
  - This stream of messages forms a sequence. So, we can define a sequence as a stream of messages. Any sequence must have a beginning. The start of every sequence is associated with some process or thread.
  - To model the sequence of a message, we can explicitly represent the order of the message relative to the start of the sequence by prefixing the message with a sequence number set apart by a colon separator.
  - Most commonly, we can specify a procedural or nested flow of control, rendering using a filled solid arrowhead. Less common but also possible, you can specify a flat flow of control, rendered using a stick arrowhead.

- We will use flat sequences only when modeling interactions in the context of use cases that involve the system as a whole, together with actors outside the system.
- In all other cases, we will use procedural sequences, because they represent ordinary, nested operation calls of the type we find in most programming languages.



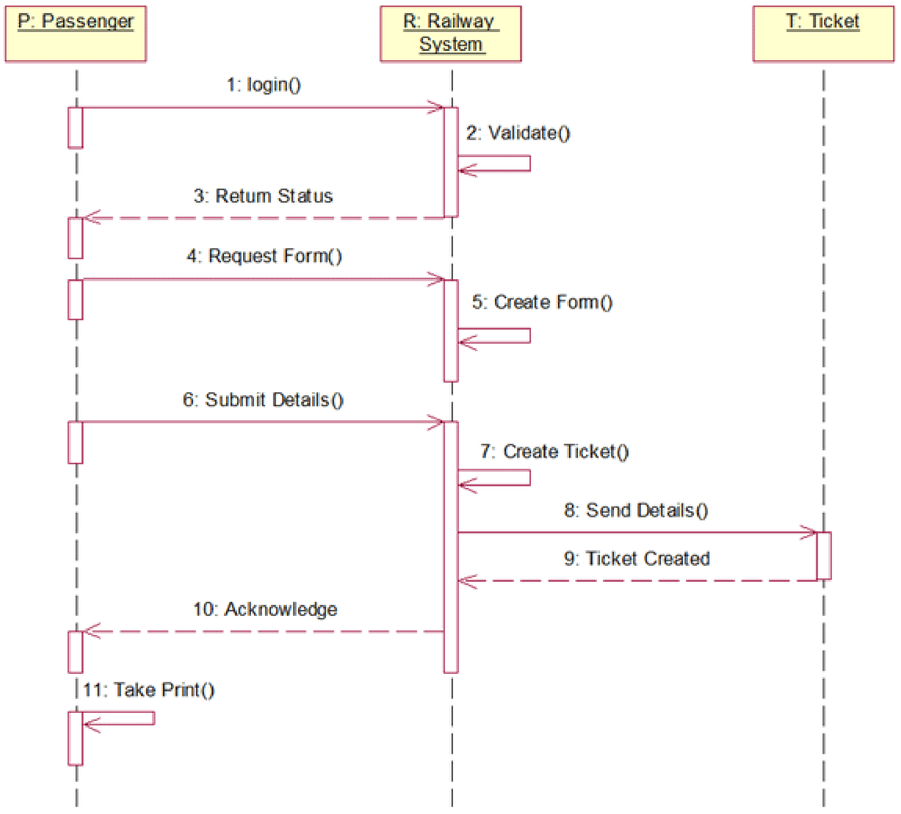
✓ **Representation**

- When we model an interaction, we typically include both objects and messages.
- We can visualize those objects and messages involved in an interaction in two ways: by emphasizing the time ordering of messages and by emphasizing the structural organization of the objects that send and receive messages.
- In UML, the first kind of representation is called a sequence diagram and the second kind of representation is called a collaboration diagram. Both sequence and collaboration diagrams are known as interaction diagrams.
- Sequence diagrams and collaboration diagrams are isomorphic, meaning that we can take one and transform it into the other without loss of information. Sequence diagram lets us to model the lifeline of an object.
- An object's lifeline represents the existence of the object at a particular time. A collaboration diagram lets us to model the structural links that may exist among the objects in the interaction.

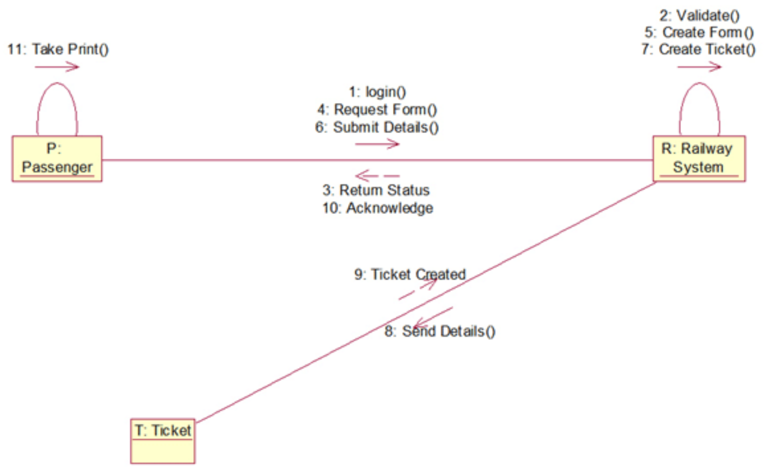
✓ **Common Modeling Techniques**

- **Modeling a flow of control**
  - Set the context for the interaction, whether it is the system as a whole, a class or an individual operation.
  - Identify the objects and their initial properties which participate in the interaction.
  - Identify the links between objects for communication through messages.
  - In time order, specify the messages that pass from object to object. Use parameters and return values as necessary.
  - To add semantics, adorn each object at every moment in time with its state and role.

- Consider the following example of railway reservation system's sequence diagram:

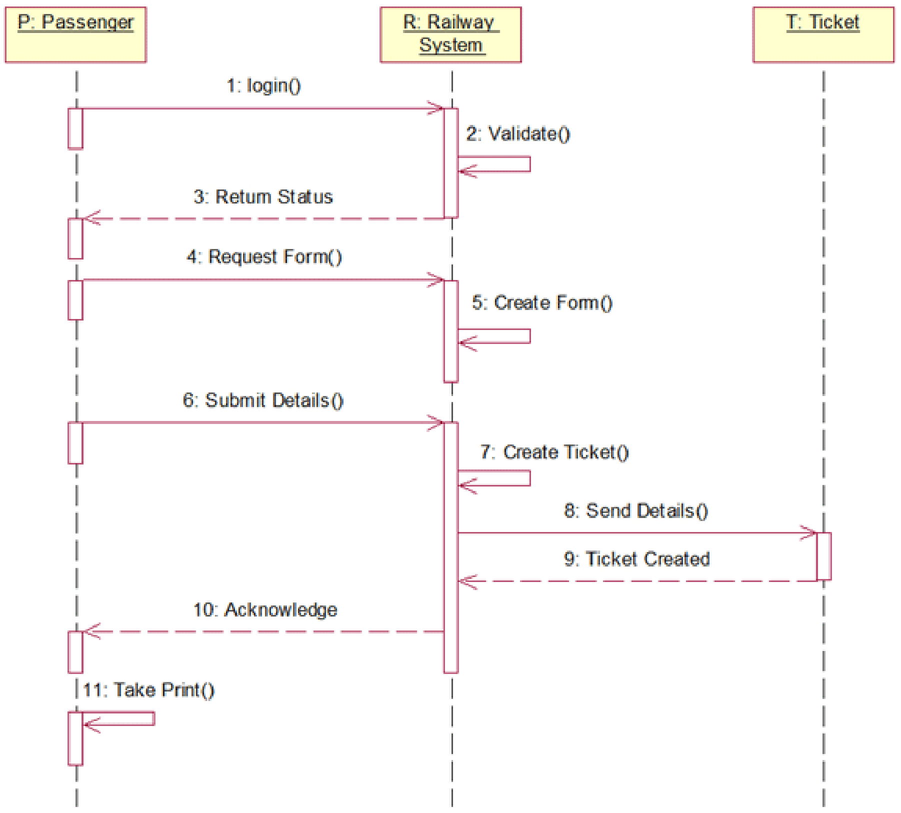


- Consider the following example of railway reservation system's collaboration diagram:



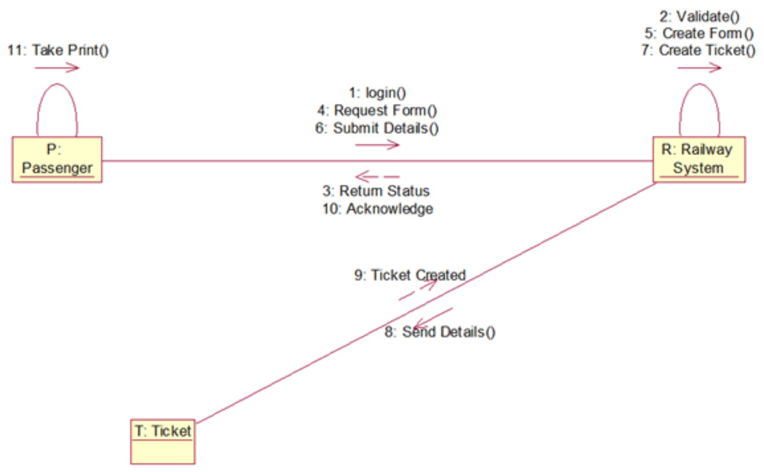
Interaction diagrams

- ✓ An interaction diagram represents an interaction, which contains a set of objects and the relationships between them including the messages exchanged between the objects.
- ✓ A sequence diagram is an interaction diagram in which the focus is on time ordering of messages.
- ✓ Collaboration diagram is another interaction diagram in which the focus is on the structural organization of the objects.
- ✓ Both sequence diagrams and collaboration diagrams are isomorphic diagrams.
- ✓ **Common Properties**
  - Interaction diagrams share the properties which are common to all the diagrams in UML. They are: a name which identifies the diagram and the graphical contents which are a projection into the model.
- ✓ **Contents**
  - Interaction diagrams commonly contain:
    - Objects
    - Links
    - Messages
- ✓ **Sequence Diagrams**
  - A sequence diagram is one of the two interaction diagrams. The sequence diagram emphasizes on the time ordering of messages.
  - In a sequence diagram, the objects that participate in the interaction are arranged at the top along the x-axis.
  - Generally, the object which initiates the interaction is placed on the left and the next important object to its right and so on.
  - The messages dispatched by the objects are arranged from top to bottom along the y-axis. This gives the user the detail about the flow of control over time.
  - Sequence diagram has two features that distinguish them from collaboration diagrams.
    - First, there is the object lifeline, which is a vertical dashed line that represents the existence of an object over a period of time. Most of the objects are alive throughout the interaction. Objects may also be created during the interaction with the receipt of the message stereotyped with create. Objects may also be destroyed during the interaction with the receipt of the message stereotyped with destroy.
    - Second, there is focus of control which is represented as a thin rectangle over the life line of the object. The focus of control represents the points in time at which the object is performing an action. We can also represent recursion by using a self message.
  - Consider the following example of railway reservation system's sequence diagram:



✓ **Collaboration Diagrams**

- A collaboration diagram is one of the two interaction diagrams. The collaboration diagram emphasizes on the structural organization of the objects in the interaction.
- A collaboration diagram is made up of objects which are the vertices and these are connected by links. Finally, the messages are represented over the links between the objects. This gives the user the detail about the flow of control in the context of structural organization of objects that collaborate.
- Collaboration diagram has two features that distinguish them from the sequence diagrams.
    - First, there is a path which indicates one object is linked to another.
    - Second, there is a sequence number to indicate the time ordering of a message by prefixing the message with a number.
- Consider the following example of railway reservation system's collaboration diagram:



✓ **Common Uses**

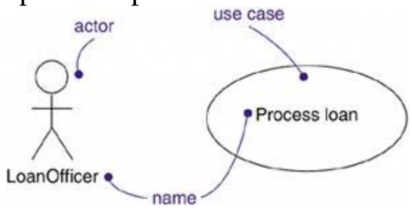
- To model flows of control by time ordering
- To model flows of control by organization

✓ **Common Modeling Techniques**

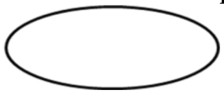
- **Modeling flow of control by time ordering**
  - Set the context for the interaction, whether it is a system, subsystem, operation or class or one scenario of a use case or collaboration.
  - Identify the objects that take part in the interaction and lay them out at the top along the x-axis in a sequence diagram.
  - Set the life line for each object.
  - Layout the messages between objects from the top along the y-axis.
  - To visualize the points at which the object is performing an action, use the focus of control.
  - To specify time constraints, adorn each message with the time and space constraints.
  - To specify the flow of control in a more formal manner, attach pre and post conditions to each message.
- **Modeling flow of control by organization**
  - Set the context for the interaction, whether it is a system, subsystem, operation or class or one scenario of a use case or collaboration.
  - Identify the objects that take part in the interaction and lay them out in a collaboration diagram as the vertices in a graph.
  - Set the initial properties of each of these objects.
  - Specify the links among these objects.
  - Starting with the messages that initiate the interaction, attach each subsequent message to the appropriate link, setting its sequence number, as appropriate. Use Dewey numbering system to specify nested flow of control.
  - To specify time constraints, adorn each message with the time and space constraints.
  - To specify the flow of control in a more formal manner, attach pre and post conditions to each message.

Use cases

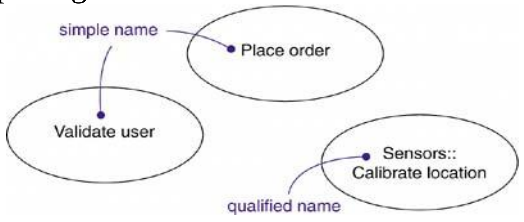
- ✓ The use cases represent aspects of the behavior of the class.
- ✓ A use case involves the interaction of actors and the system or other subject.
- ✓ An actor represents a coherent set of roles that users of use cases play when interacting with these use cases. Actors can be human or they can be automated systems.
- ✓ For example, in modelling a bank, processing a loan involves, among other things, the interaction between a customer and a loan officer.
- ✓ The UML provides a graphical representation of a use case and an actor.



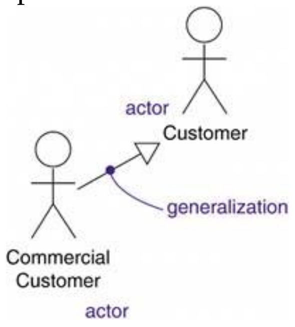
- ✓ **Terms and Concepts**
  - A use case is a description of a set of sequences of actions, including variants that a system performs to yield an observable result of value to an actor.
  - Graphically, a use case is rendered as an ellipse.



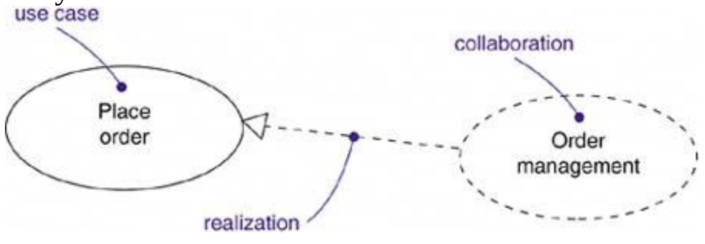
  - **Name**:
    - Every use case must have a name that distinguishes it from other use cases. A name is a textual string. That name alone is known as a simple name; a qualified name is the use case name prefixed by the name of the package in which that use case lives.



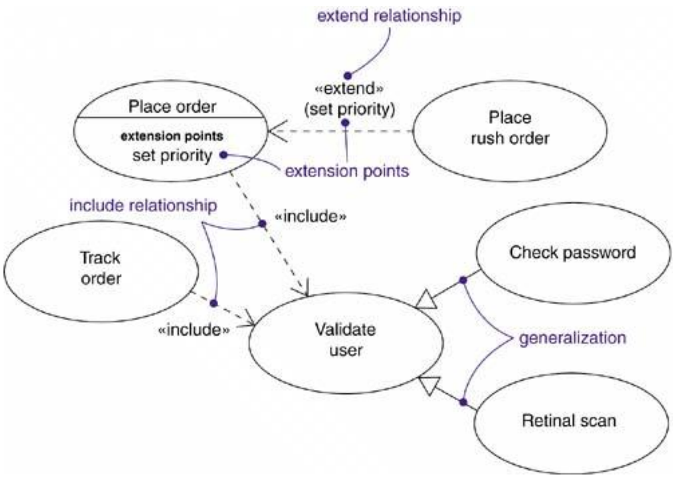
  - **Actors:**
    - An actor represents a coherent set of roles that users of use cases play when interacting with these use cases.
    - Typically, an actor represents a role that a human, a hardware device, or even another system plays with a system.
    - Actors are not actually part of the software application. They live outside the application within the surrounding environment.
    - Example: bank_customer, bank_manager and load_officer
    - Graphically represented as stick man



- **Flowevents:**
  - A flow of events is a textual description embodying sequence of events with regards to the use case and is part of the use case specification.
  - Flow of events is understood by the customer.
  - Flow of events describes how and when the use case starts and ends, when the use case interacts with the actors, and the information exchanged between an actor and the use case.
  - Types Flow of Events
    - ✓ Main flow of events
      - ➢ The use case starts when the system prompts the Customer for a PIN number. The Customer can now enter a PIN number via the keypad. The Customer commits the entry by pressing the Enter button. The system then checks this PIN number to see if it is valid. If the PIN number is valid, the system acknowledges the entry, thus ending the use case.
    - ✓ Exceptional flow of events
      - ➢ 1. The Customer can cancel a transaction at any time by pressing the Cancel button, thus restarting the use case. No changes are made to the Customer's account.
      - ➢ 2. The Customer can clear a PIN number anytime before committing it and re-enter a new PIN number.
      - ➢ 3. If the Customer enters an invalid PIN number, the use case restarts. If this happens three times in a row, the system cancels the entire transaction, preventing the Customer from interacting with the ATM for 60 seconds.
- **Collaborations**
  - A use case captures the intended behavior of the system (or subsystem, class, or interface) we are developing, without having to specify how that behavior is implemented.
  - That's an important separation because the analysis of a system should, as much as possible, not be influenced by implementation issues.
  - Ultimately, however, we have to implement our use cases, and we do so by creating a society of classes and other elements that work together to implement the behavior of this use case.
  - This society of elements, including both its static and dynamic structure, is modelled in the UML as collaboration.
  - As figure shows, you can explicitly specify the realization of a use case by collaboration.

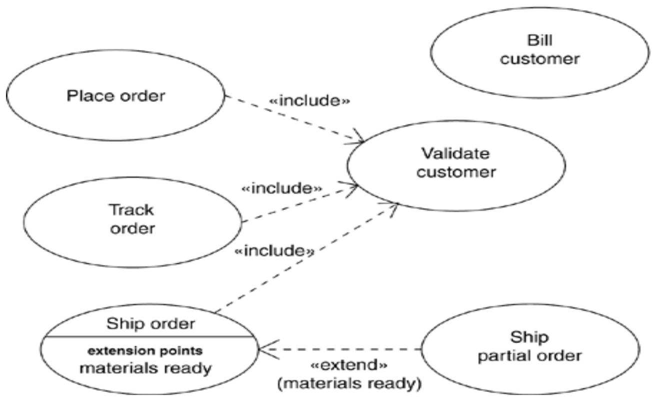


- **Organizing Use Cases**
  - We can also organize use cases by specifying generalization, include, and extend relationships among them.
  - Generalization among use cases is just like generalization among classes. Here it means that the child use case inherits the behavior and meaning of the parent use case; the child may add to or override the behavior of its parent; and the child may be substituted any place the parent appears.
  - For example, in a banking system, we might have the use case Validate User, which is responsible for verifying the identity of the user. You might then have two specialized children of this use case (Check password and Retinal scan).
  - As shown in Figure generalization among use cases is rendered as a solid directed line with a large triangular arrowhead, just like generalization among classes.
  - <<**include**>> Specifies that the source use case explicitly incorporates the behaviour of another use case at a location specified by the source
  - <<**extend**>> Specifies that the target use case extends the behaviour of the source.



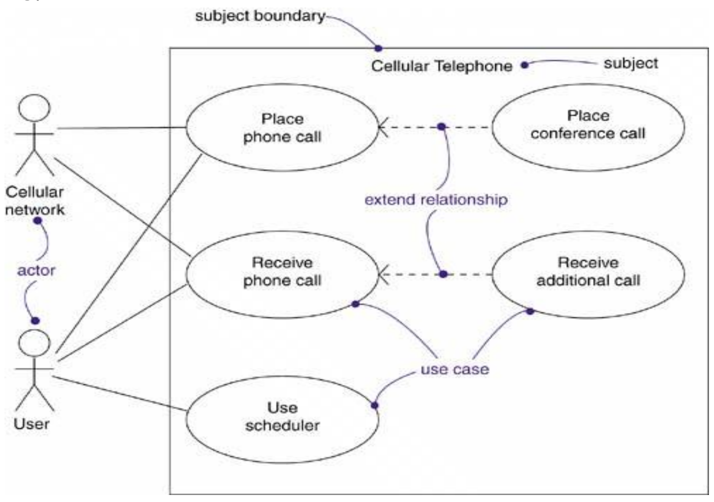
✓ **Common Modeling Techniques**

- **Modeling the Behavior of an Element**
  - Identify the actors that interact with the element. Candidate actors include groups that require certain behavior to perform their tasks or that are needed directly or indirectly to perform the element's functions.
  - Organize actors by identifying general and more specialized roles.
  - For each actor, consider the primary ways in which that actor interacts with the element. Consider also interactions that change the state of the element or its environment or that involve a response to some event.
  - Consider also the exceptional ways in which each actor interacts with the element.
  - Organize these behaviors as use cases, applying include and extend relationships to factor common behavior and distinguish exceptional behavior.



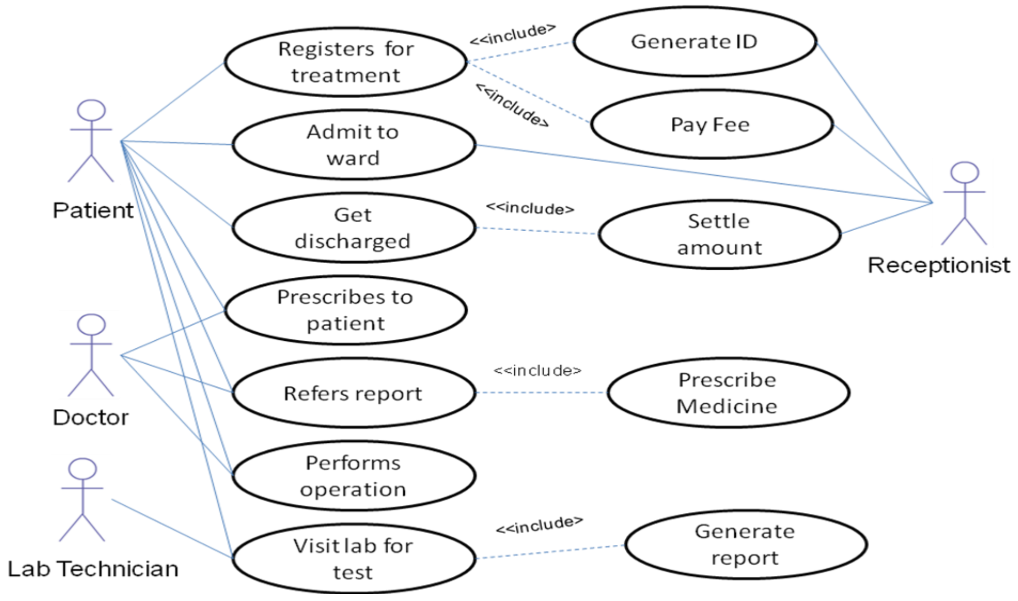
Use case diagrams

- ✓ A use case diagram is a diagram that shows a set of use cases, actors and their relationships.
- ✓ Use case diagrams to visualize the behavior of a system, subsystem, or class so that users can comprehend how to use that element, and so that developers can implement that element.
- ✓ As Figure shows, we can provide a use case diagram to model the behavior of a cellular phone.



- ✓ **Terms and Concepts**
  - **Common Properties**
    - ▪ A use case diagram is just a special kind of diagram and shares the same common properties as do all other diagrams a name and graphical contents that are a projection into a model.
  - **Contents**
    - ▪ Use case diagrams commonly contain
      - ✓ Subject
      - ✓ Use cases
      - ✓ Actors
      - ✓ Dependency, generalization, and association relationships
  - **Common Uses**
    - ▪ To model the context of a subject
    - ▪ To model the requirements of a subject

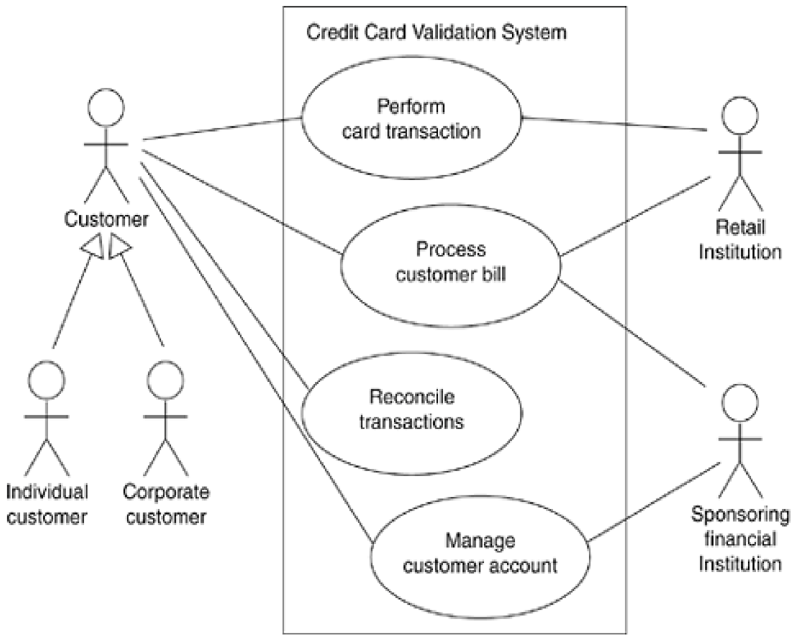
✓ **Example for Use case diagram for Hospital Management System**



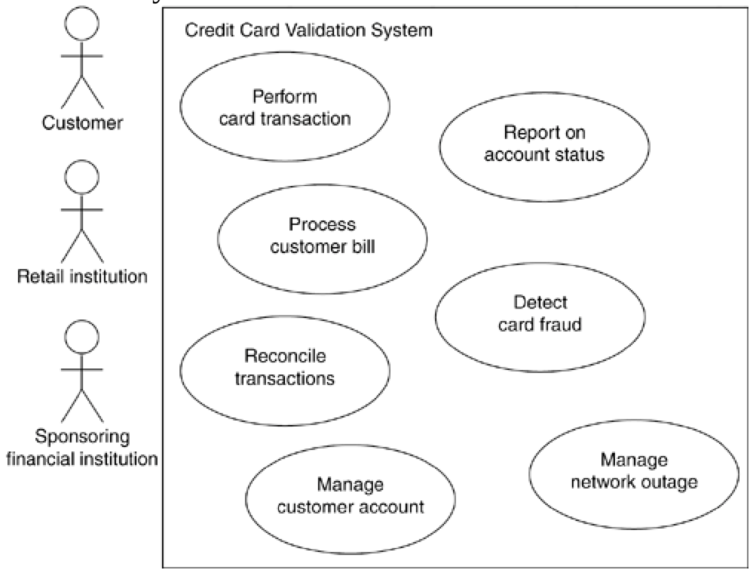
✓ **Common Modeling Techniques**

- **Modeling the Context of a System**
  - Identify the boundaries of the system by deciding which behaviors is part of it and which are performed by external entities. This defines the subject.
  - Identify the actors that surround the system by considering which groups require help from the system to perform their tasks, which groups are needed to execute the system's functions, which groups interact with external hardware or other software systems, and which groups perform secondary functions for administration and maintenance.
  - Organize actors that are similar to one another in a generalization-specialization hierarchy.
  - Where it aids understand ability, provide a stereotype for each such actor.

  - For example, Figure shows the context of a credit card validation system, with an emphasis on the actors that surround the system. We will find Customers, of which there are two kinds (Individual customer and corporate customer). These actors are the roles that humans play when interacting with the system. In this context, there are also actors that represent other institutions, such as Retail institution (with which a Customer performs a card transaction to buy an item or a service) and sponsoring financial institution (which serves as the clearinghouse for the credit card account). In the real world, these latter two actors are likely software-intensive systems themselves.



- **Modeling the Requirements of a System**
    - Establish the context of the system by identifying the actors that surround it.
    - For each actor, consider the behavior that each expects or requires the system to provide.
    - Name these common behaviors as use cases.
    - Factor common behavior into new use cases that are used by others; factor variant behavior into new use cases that extend more main line flows.
    - Model these use cases, actors, and their relationships in a use case diagram.
    - Adorn these use cases with notes or constraints that assert nonfunctional requirements; you may have to attach some of these to the whole system.

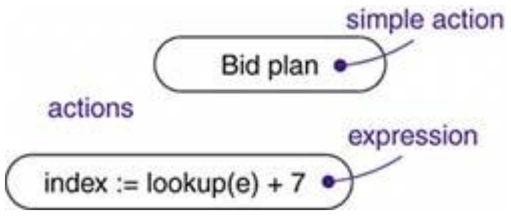


- **Forward Engineering**
    - Identify the objects that interact with the system. Try to identify the various roles that each external object may play.
    - Make up an actor to represent each distinct interaction role.
    - For each use case in the diagram, identify its flow of events and its exceptional flow of events.
    - Depending on how deeply you choose to test, generate a test script for each flow, using the flow's preconditions as the test's initial state and its postconditions as its success criteria.
    - As necessary, generate test scaffolding to represent each actor that interacts with the use case. Actors that push information to the element or are acted on by the element may either be simulated or substituted by its real-world equivalent.
    - Use tools to run these tests each time you release the element to which the use case diagram applies.
- **Reverse Engineering**
    - Identify each actor that interacts with the system.
    - For each actor, consider the manner in which that actor interacts with the system, changes the state of the system or its environment, or responds to some event.
    - Trace the flow of events in the executable system relative to each actor. Start with primary flows and only later consider alternative paths.
    - Cluster related flows by declaring a corresponding use case. Consider modeling variants using extend relationships, and consider modeling common flows by applying include relationships.
    - Render these actors and use cases in a use case diagram, and establish their relationships.

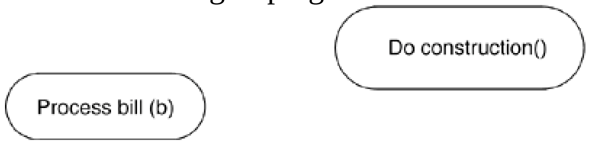
Syllabus: Unit – IV: Advanced Behavioral And Architectural Modeling
Activity diagrams - Events and Signals – State chart diagrams - Components and Component diagrams - Deployment and Deployment diagrams.

Activity diagrams

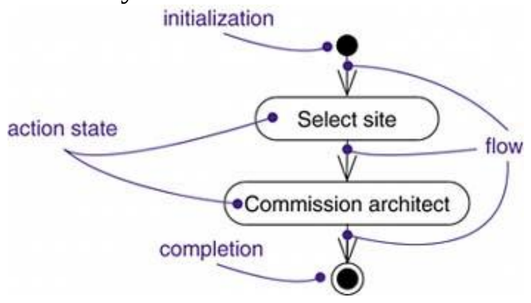
- ✓ An activity diagram shows the flow from activity to activity.
- ✓ An activity is an ongoing non-atomic execution within a state machine.
- ✓ The execution of an activity ultimately expands into the execution of individual actions, each of which may change the state of the system or communicate messages.
- ✓ Actions encompass calling another operation, sending a signal, creating or destroying an object, or some pure computation such as evaluating an expression.
- ✓ Graphically, an activity diagram is a collection of nodes and arcs.
- ✓ **Contents**
    - Activity diagrams commonly contain
        - Actions
        - Activity nodes
        - Flows
        - Object values
- ✓ **Actions**
    - We might evaluate some expression that sets the value of an attribute or that returns some value.



- ✓ **Activity nodes**
    - An activity node is an organizational unit within an activity. In general, activity nodes are nested groupings of actions or other nested activity nodes.

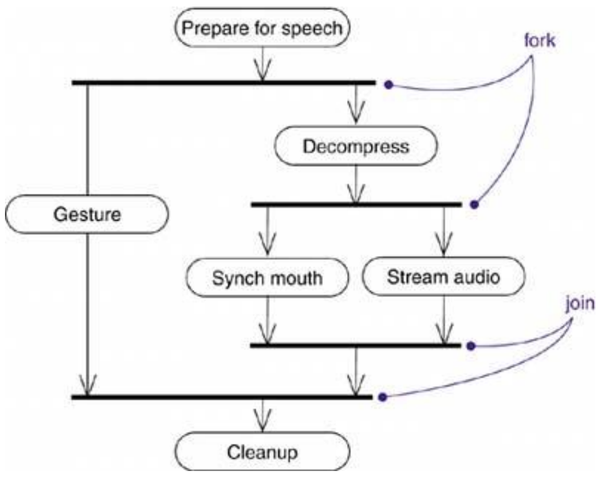


- ✓ **Control Flows**
    - When an action or activity node completes execution, flow of control passes immediately to the next action or activity node. You specify this flow by using flow arrows to show the path of control from one action or activity node to the next action or activity node.



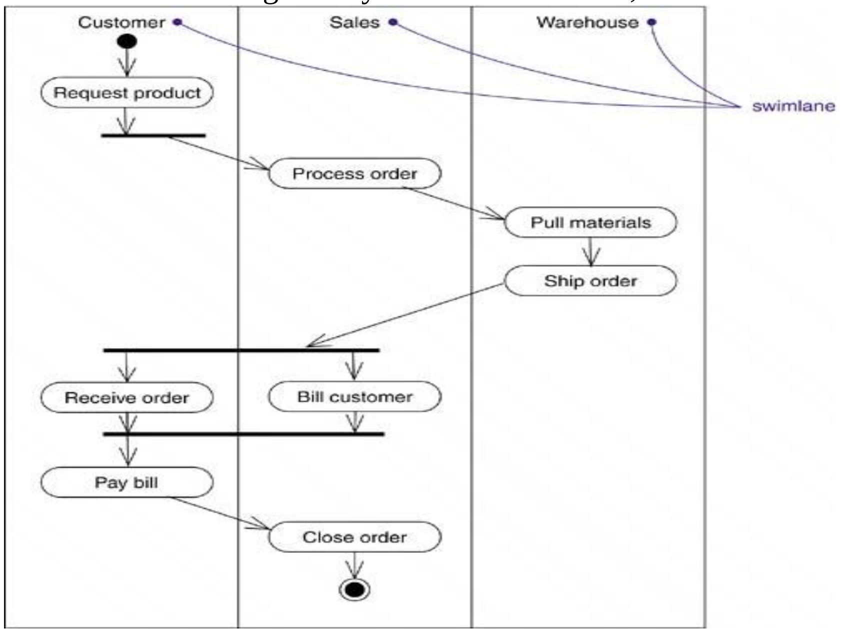
✓ **Forking and Joining**

- In the UML, we use a synchronization bar to specify the forking and joining of these parallel flows of control.
- A synchronization bar is rendered as a thick horizontal or vertical line.



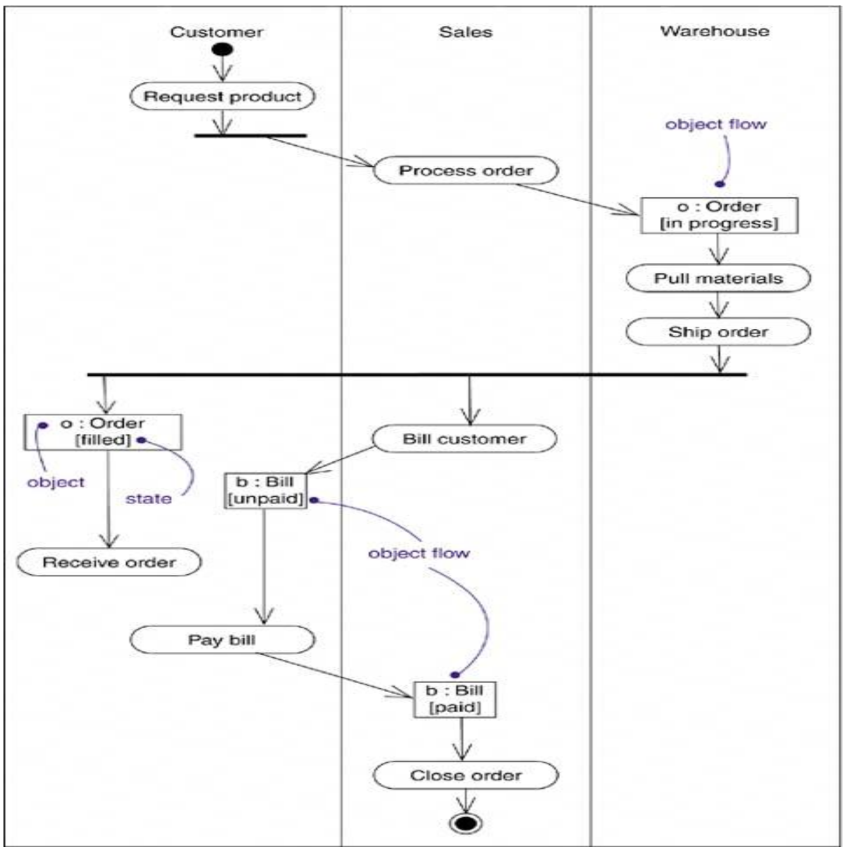
✓ **Swimlanes**

- To partition the activity states on an activity diagram into groups, each group representing the business organization responsible for those activities.
- In the UML, each group is called a swimlane because, visually, each group is divided from its neighbor by a vertical solid line,



✓ **Object Flow**

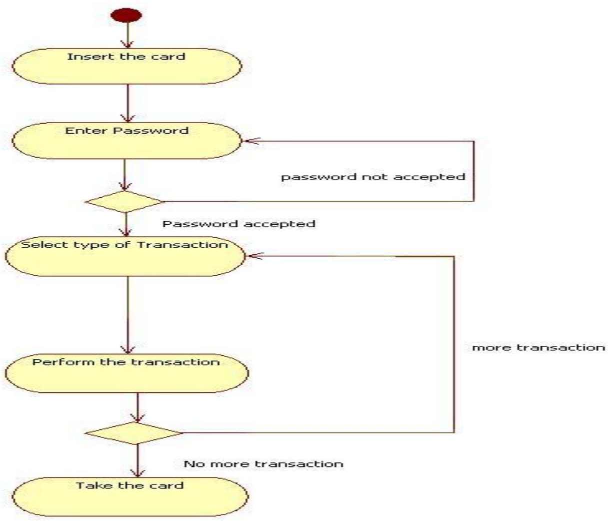
- Objects may be involved in the flow of control associated with an activity diagram.



✓ **Common Uses**

1. To model a workflow
2. To model an operation

✓ **Example for ATM Machine**



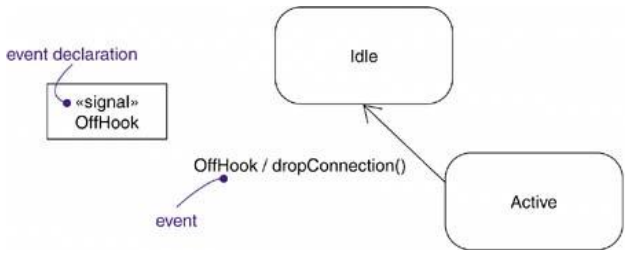
✓ **Common Modeling Techniques**

- **Modeling a Workflow**
    - Establish a focus for the workflow. For nontrivial systems, it's impossible to show all interesting workflows in one diagram.
    - Select the business objects that have the high-level responsibilities for parts of the overall workflow. These may be real things from the vocabulary of the system, or they may be more abstract. In either case, create a swimlane for each important business object or organization.
    - Identify the preconditions of the workflow's initial state and the post conditions of the workflow's final state. This is important in helping you model the boundaries of the workflow.
    - Beginning at the workflow's initial state, specify the actions that take place over time and render them in the activity diagram.
    - For complicated actions or for sets of actions that appear multiple times, collapse these into calls to a separate activity diagram.
    - Render the flows that connect these actions and activity nodes. Start with the sequential flows in the workflow first, next consider branching, and only then consider forking and joining.
    - If there are important object values that are involved in the workflow, render them in the activity diagram as well. Show their changing values and state as necessary to communicate the intent of the object flow.
- **Modeling an Operation**
    - Collect the abstractions that are involved in this operation. This includes the operation's parameters (including its return type, if any), the attributes of the enclosing class, and certain neighboring classes.
    - Identify the preconditions at the operation's initial state and the post conditions at the operation's final state. Also identify any invariants of the enclosing class that must hold during the execution of the operation.
    - Beginning at the operation's initial state, specify the activities and actions that take place over time and render them in the activity diagram as either activity states or action states.
    - Use branching as necessary to specify conditional paths and iteration.
    - Only if this operation is owned by an active class, use forking and joining as necessary to specify parallel flows of control.

Events and signals

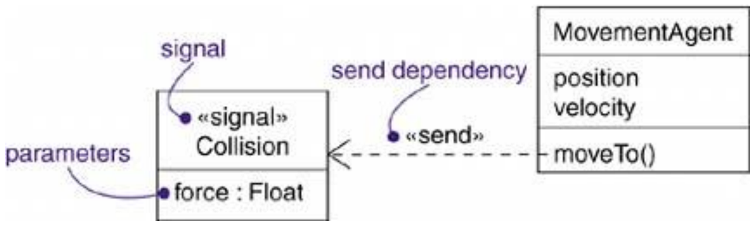
✓ **Events**

- An event is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition.
- A signal is a kind of event that represents the specification of an asynchronous message communicated between instances.
- **Events may be external or internal:** External events are those that pass between the system and its actors.
- For example, the pushing of a button and an interrupt from a collision sensor are both examples of external events. Internal events are those that pass among the objects that live inside the system. An overflow exception is an example of an internal event.



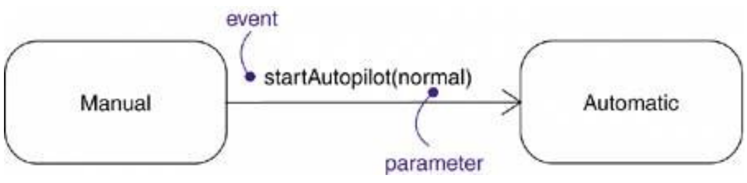
✓ **Signals**

- A message is a named object that is sent asynchronously by one object and then received by another. A signal is a classifier for messages; it is a message type.



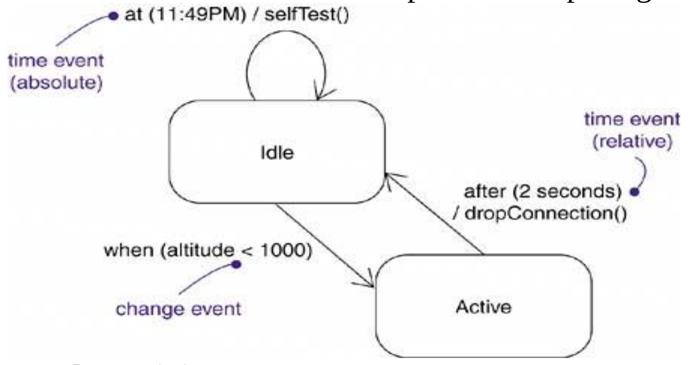
✓ **Call Events**

- Just as a signal event represents the occurrence of a signal, a call event represents the receipt by an object of a call request for an operation on the object. A call event may trigger a state transition in a state machine or it may invoke a method on the target object. The choice is specified in the class definition for the operation.



✓ **Time and Change Events**

- A time event is an event that represents the passage of time.

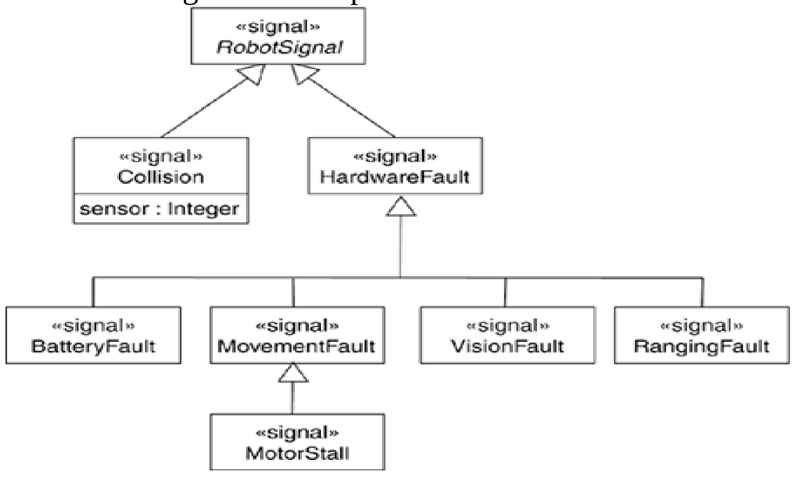


✓ **Sending and Receiving Events**

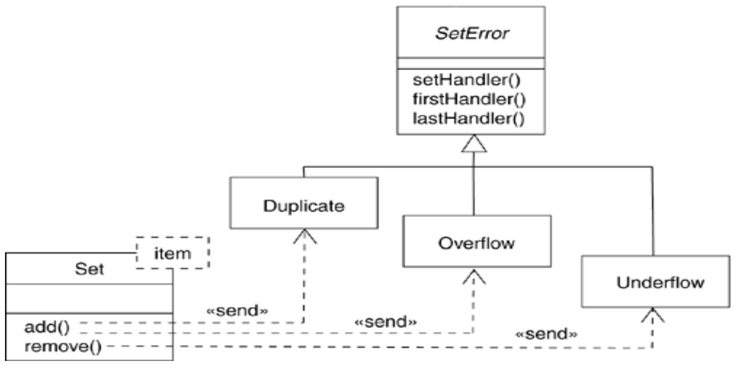
- Signal events and call events involve at least two objects: the object that sends the signal or invokes the operation and the object to which the event is directed. Because signals are asynchronous, and because asynchronous calls are themselves signals, the semantics of events interact with the semantics of active objects and passive objects.

✓ **Common Modeling Techniques**

- **Modeling a Family of Signals**
  - Consider all the different kinds of signals to which a given set of active objects may respond.
  - Look for the common kinds of signals and place them in a generalization/specialization hierarchy using inheritance. Elevate more general ones and lower more specialized ones.
  - Look for the opportunity for polymorphism in the state machines of these active objects. Where you find polymorphism, adjust the hierarchy as necessary by introducing intermediate abstract signals.
  - Figure models a family of signals that may be handled by an autonomous robot. Note that the root signal (RobotSignal) is abstract, which means that there may be no direct instances. This signal has two immediate concrete specializations (Collision and HardwareFault), one of which (HardwareFault) is further specialized. Note that the Collision signal has one parameter.

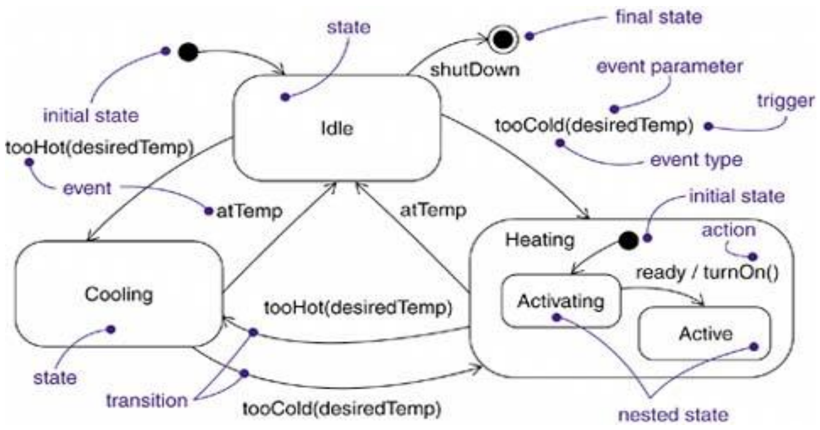


- **Modeling Abnormal Occurrences**
  - For each class and interface, and for each operation of such elements, consider the normal things that happen. Then think of things that can go wrong and model them as signals among objects.
  - Arrange the signals in a hierarchy. Elevate general ones, lower specialized ones, and introduce intermediate exceptions as necessary.
  - For each operation, specify the abnormal occurrence signals that it may raise. You can do so explicitly (by showing send dependencies from an operation to its signals) or you can use sequence diagrams illustrating various scenarios.

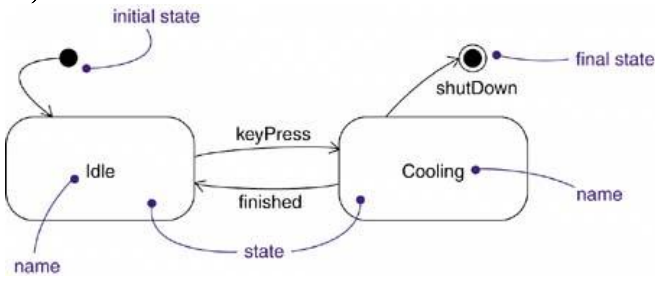


State machines

- ✓ A state machine is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.
- ✓ A state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for events.
- ✓ An event is the specification of a significant occurrence that has a location in time and space.
- ✓ In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition.
- ✓ A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.
- ✓ An activity is ongoing non-atomic execution within a state machine.
- ✓ An action is an executable computation that results in a change in state of the model or the return of a value.
- ✓ Graphically, a state is rendered as a rectangle with rounded corners. and a transition is rendered as a solid directed line or path from the original state to the new state.
- ✓ The UML provides a graphical representation of states, transitions, events, and effects, as Figure shows.



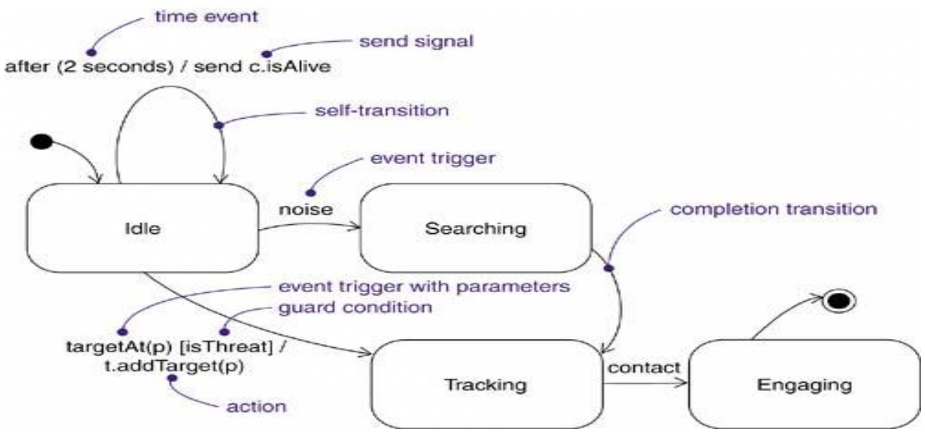
- ✓ **States**
  - A state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
  - An object remains in a state for a finite amount of time.
  - For example, a Heater in a home might be in any of four states: Idle (waiting for a command to start heating the house), Activating (its gas is on, but it's waiting to come up to temperature), Active (its gas and blower are both on), and ShuttingDown (its gas is off but its blower is on, flushing residual heat from the system).



  - A state has several parts:
    - 1. Name
      - ✓ A textual string that distinguishes the state from other states; a state may be anonymous, meaning that it has no name.
    - 2. Entry/exit effects
      - ✓ Actions executed on entering and exiting the state, respectively.
    - 3. Internal transitions
      - ✓ Transitions that are handled without causing a change in state.
    - 4. Substates – may sequential or concurrent
      - ✓ The nested structure of a state, involving non-orthogonal (sequentially active) or orthogonal (concurrently active) sub-states.
    - 5. Deferred events - (infrequently used)
      - ✓ A list of events that are not handled in that state but, rather, are postponed and queued for handling by the object in another state.
    - Special categories of states
      - ✓ Initial state - indicates the initial starting state for the state machine or a substate.
      - ✓ Final state - indicates the state machine’s execution has completed.

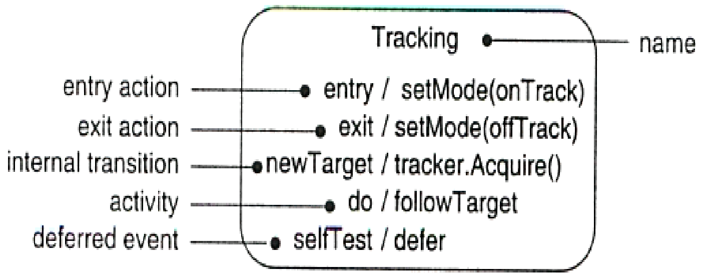
✓ **Transition**

- A directed relationship between two states.
- A flow of control through a statechart diagram.
- A Transition has five parts:
    1. Source state - current state before transition fires.
    2. Event trigger - external stimulus that has the potential to cause a transition to fire.
    3. Guard condition - a condition that must be satisfied before a transition can fire.
    4. Action - an executable atomic computation.
    5. Target state - new state after transition fires.

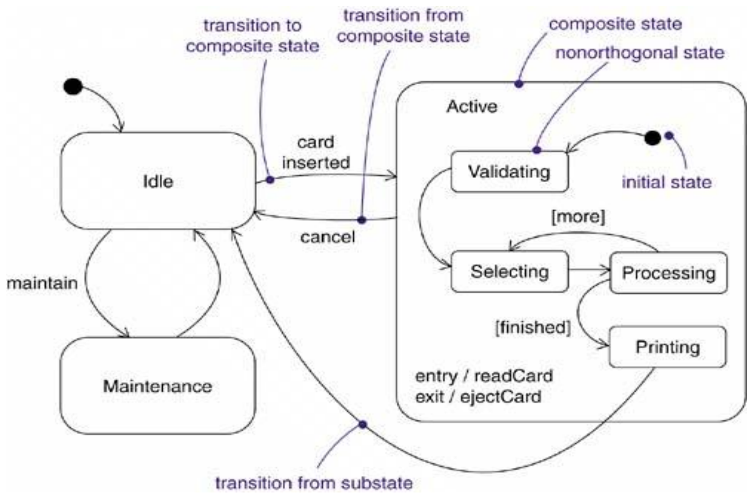


✓ **Advanced States and Transitions**

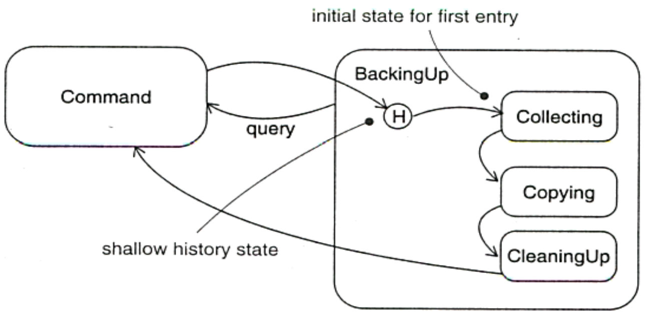
- Entry action - Upon each entry to a state, a specified action is automatically executed.
  entry / action
- Exit action - Just prior to leaving a state, a specified action is automatically executed.
  exit / action
- Internal Transitions - The handling of an event without leaving the current state.
  event / action
- Activities - Ongoing work that an object performs while in a particular state. The work automatically terminates when the state is exited.
  do / activity
- Deferred Event - An event whose occurrence is responded to at a later time.
  event / defer



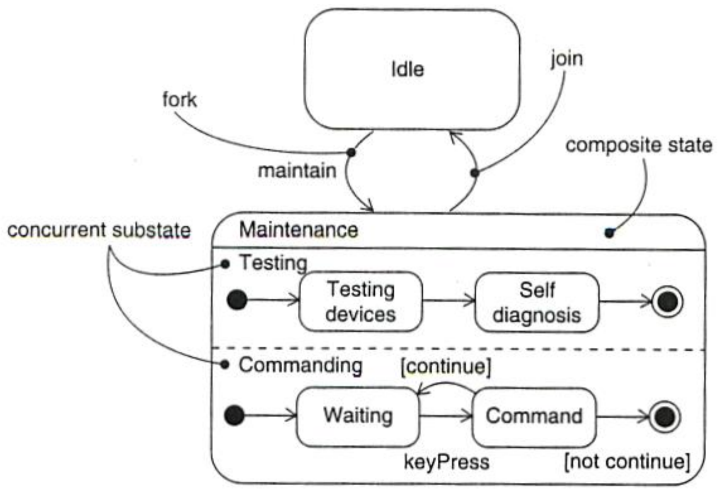
- Simple state - A state that contains no substates.
- Composite state - A state that contains substates.
- Substate - A state that is nested inside another state.
  - Substates allow state diagrams to show different levels of abstraction.
  - Substates may be sequential or concurrent.
  - Substates may be nested to any level.
- Sequential Substates - The most common type of substate. Essentially, a state diagram within a single state.
  - The "containing" state becomes an abstract state.
  - The use of substates simplifies state diagrams by reducing the number of transition lines.
  - A nested sequential state diagram may have at most one initial state and one final state.



- History States - Allows an object to remember which substate was last active when the containing state was exited.

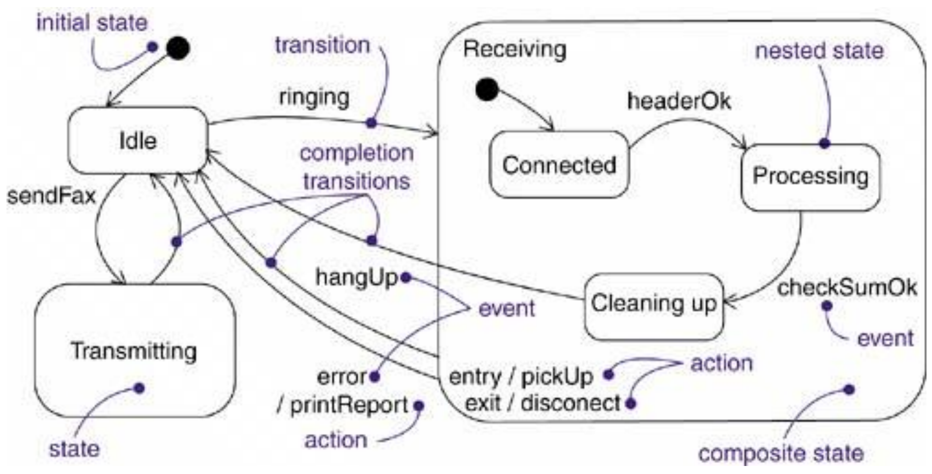


- Concurrent Substates (Fork and Join) - Used when two or more state diagrams are executing concurrently within a single object.
  - Allows an object to be in multiple states simultaneously.
  - The concurrent state diagrams within a "containing" state must begin and end execution simultaneously.
  - If one concurrent state diagram finishes first, it must wait for the others to complete before exiting the containing state.



State chart diagrams

- ✓ A state diagram shows a state machine, emphasizing the flow of control from state to state.
- ✓ A state machine is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.
- ✓ A state is a condition or situation in the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
- ✓ An event is the specification of a significant occurrence that has a location in time and space.
- ✓ In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition.
- ✓ A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.
- ✓ An activity specifies an ongoing execution within a state machine. An action specifies a primitive executable computation that results in a change in state of the model or the return of a value.
- ✓ Graphically, a state diagram is a collection of nodes and arcs.



✓ **Common Properties**

- A state diagram is just a special kind of diagram and shares the same common properties as do all other diagrams that is, a name and graphical contents that are a projection into a model. What distinguishes a state diagram from all other kinds of diagrams is its content.

✓ **Contents**

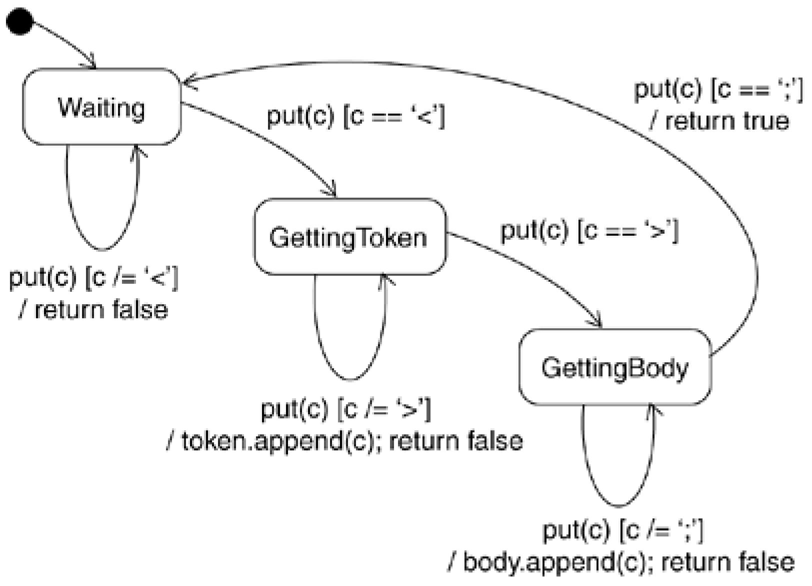
- State diagrams commonly contain
    - Simple states and composite states
    - Transitions, events, and actions

✓ **Common Uses**

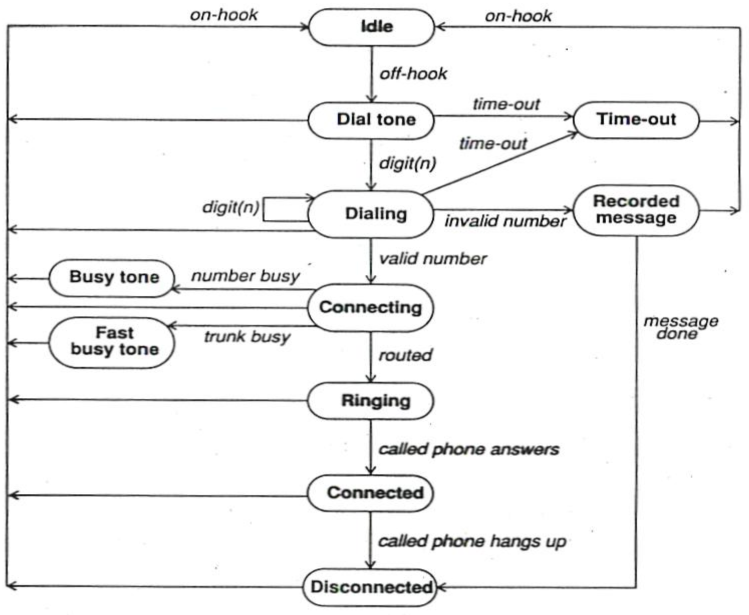
- To model the dynamic aspects of a system

✓ **Common Modeling Techniques**

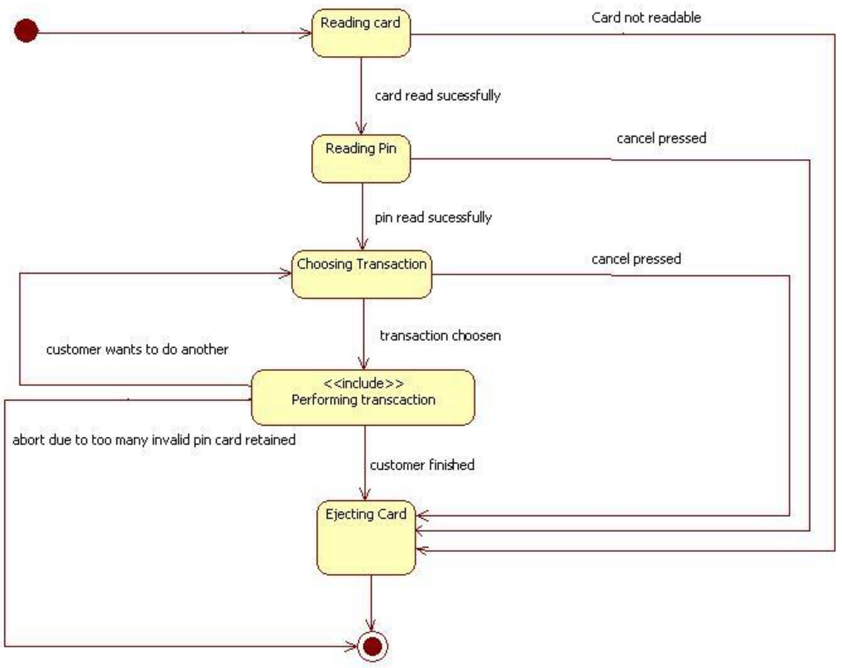
- **Modeling Reactive Objects**
    - Choose the context for the state machine, whether it is a class, a use case, or the system as a whole.
    - Choose the initial and final states for the object. To guide the rest of your model, possibly state the pre-and post conditions of the initial and final states, respectively.
    - Decide on the stable states of the object by considering the conditions in which the object may exist for some identifiable period of time. Start with the high-level states of the object and only then consider its possible substates.
    - Decide on the meaningful partial ordering of stable states over the lifetime of the object.
    - Decide on the events that may trigger a transition from state to state. Model these events as triggers to transitions that move from one legal ordering of states to another.
    - Attach actions to these transitions (as in a Mealy machine) and/or to these states (as in a Moore machine).
    - Consider ways to simplify your machine by using substates, branches, forks, joins, and history states.
    - Check that all states are reachable under some combination of events.
    - Check that no state is a dead end from which no combination of events will transition the object out of that state.
    - Trace through the state machine, either manually or by using tools, to check it against expected sequences of events and their responses.


Waiting
put(c) [c == '<']
put(c) [c == ';'] / return true
put(c) [c /= '<'] / return false
GettingToken
put(c) [c == '>']
put(c) [c /= '>'] / token.append(c); return false
GettingBody
put(c) [c /= ';'] / body.append(c); return false

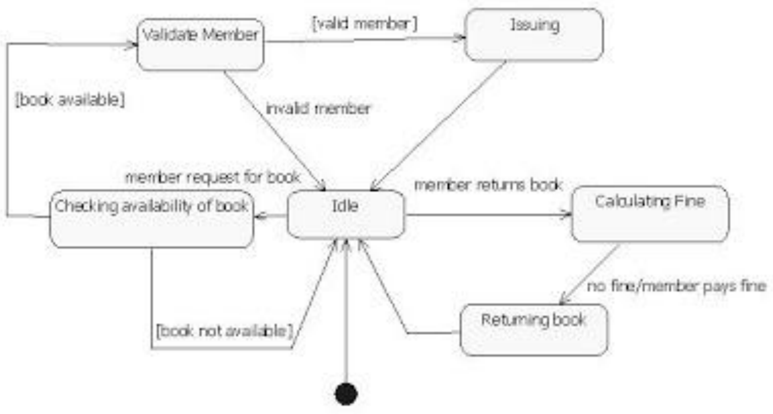
- ✓ **Example for phone call**



- ✓ **Example for ATM**

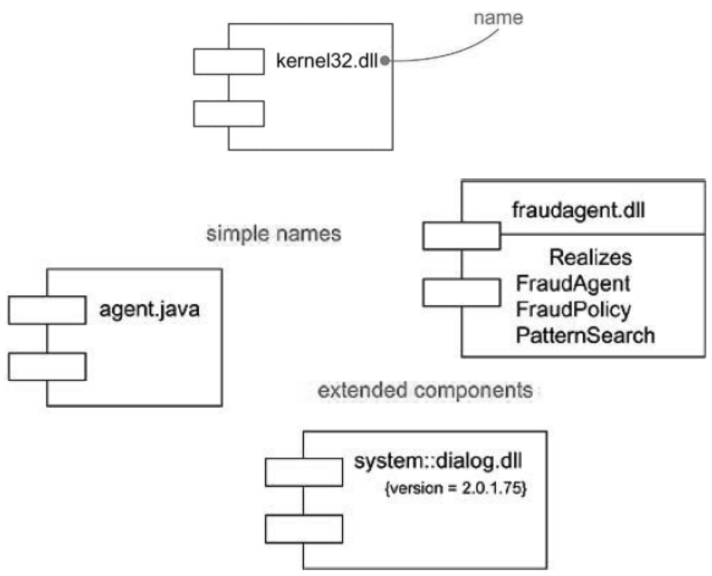


- ✓ **Example for Library Management System**

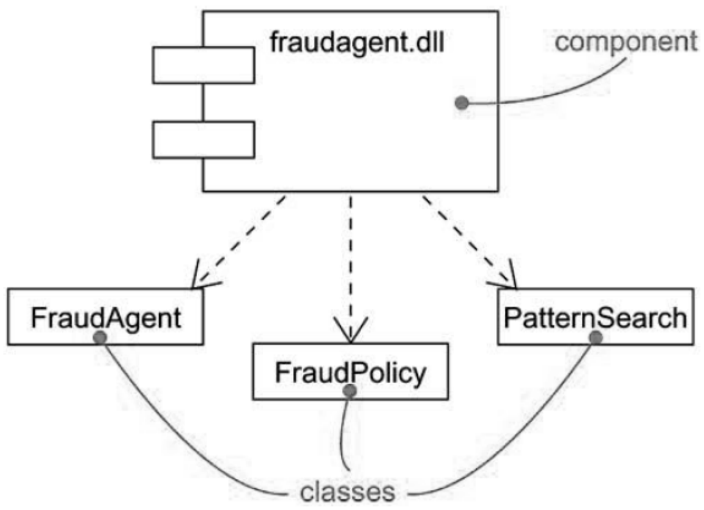


Component

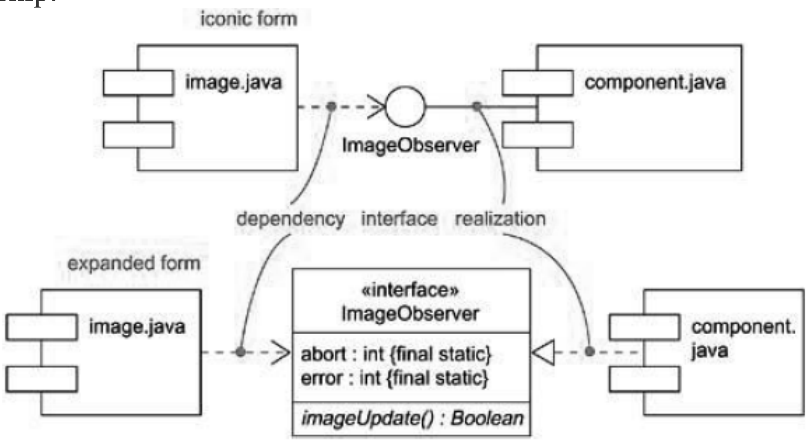
- ✓ A component is a physical replaceable part of a system that complies with and provides the realization of a set of interfaces.
- ✓ We use components to model the physical things that may reside on a node, such as executables, libraries, tables, files and documents.
- ✓ A component typically represents the physical packaging of otherwise logical elements such as classes, interfaces and collaborations.
- ✓ We do logical modeling to visualize, specify, and document our decisions about the vocabulary of our domain and the structural and behavioral way those things collaborate.
- ✓ We do physical modeling to construct the executable system. Object libraries, executables, COM+ components and Enterprise Java Beans are all examples of components.



- ✓ **Components and Classes**
  - In many ways, components are like classes. Both have names, both may realize a set of interfaces, both may participate in dependency, generalization and association relationships, both may be nested, and both may have instances. However, there are some significant differences between components and classes:
    - Classes represent logical abstractions, components represent physical things that live in the world of bits. In short, components may live on nodes, classes may not.
    - Components represent the physical packaging of otherwise logical components and are at a different level of abstraction.
    - Classes may have attributes and operations directly. In general, components only have operations that are reachable only through their interfaces.
  - The relationship between a component and the classes it implements can be shown explicitly by using a dependency relationship as shown below:



- ✓ **Components and Interfaces**
  - o An interface is a collection of operations that are used to specify a service of a class or a component.
  - o We can show the relationship between a component and its interfaces in one of the two ways.
    - ▪ The first style renders the interfaces in its elided, iconic form. The component that realizes the interfaces is connected to the interface using an elided realization relationship.
    - ▪ The second style renders the interface in its expanded form, perhaps revealing its operations. The component that realizes the interface is connected to the interface using a full realization relationship.
  - o In both cases, the component that accesses the services of the other component through the interfaces is connected to the interface using a dependency relationship.



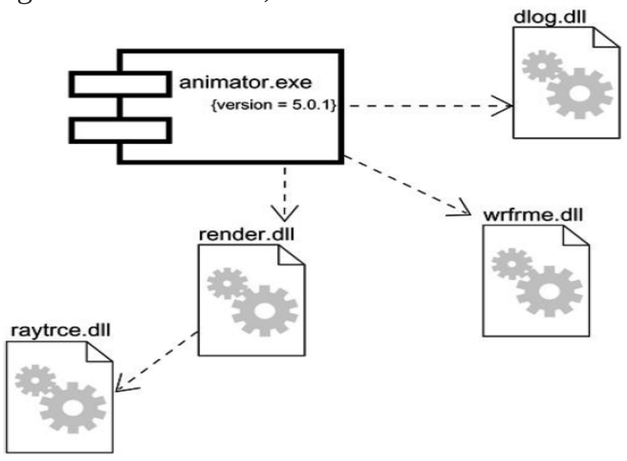
- ✓ **Binary Replaceability**
  - o The basic intent of every component-based operating system facility is to permit the assembly of systems from binary replaceable parts.
  - o This means that we can create a system out of components and then evolve the system by adding new components and replacing the old ones, without rebuilding the system.
    - ▪ First, a component is physical. It lives in the world of bits, not concepts.
    - ▪ Second, a component is replaceable. A component is substitutable means it is possible to replace a component with another that conforms to the sane interfaces.
    - ▪ Third, a component is part of a system. A component rarely stands alone. Rather, a given component collaborates with other components and in so doing exists in the architectural or technology context in which it is intended to be used.
    - ▪ Fourth, a component conforms to and provides the realization of a set of interfaces.

- ✓ **Kinds of Components**
  - Three kinds of components may be distinguished.
    - First, there are deployment components. These are the components necessary and sufficient to form an executable system, such as dynamic libraries (DLLs) and executables (EXEs).
    - Second, there are work product components. These components are generally the residue of the development process, consisting of things such as the source code files and data files from which deployment components are created.
    - Third are execution components. These components are created as a consequence of an executing system, such as COM+ object, which is instantiated from a DLL.

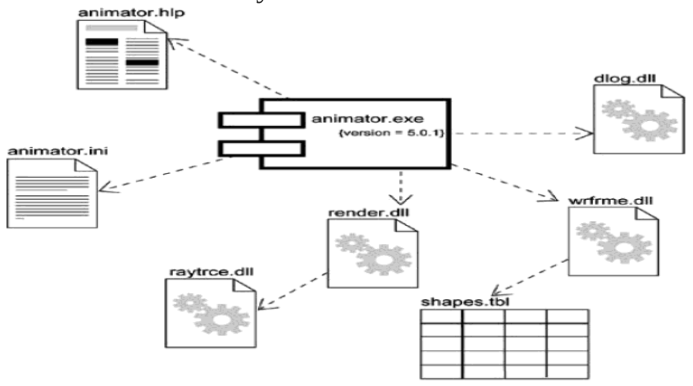
- ✓ **Standard Elements**
  - All the UML's extensibility mechanisms apply to components. Most often, we'll use tagged values to extend the component properties and stereotypes to specify new kind of components.
  - The UML defines five standard stereotypes that apply to components:

executable	Specifies a component that may be executed on a node
library	Specifies a static or dynamic object library
table	Specifies a component that represents a database table
file	Specifies a component that represents a document containing code or data
document	Specifies a component that represents a document

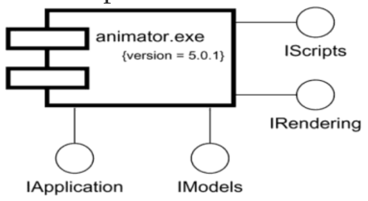
- ✓ <u>**Common Modeling Techniques**</u>
  - **Modeling Executables and Libraries**
    - Identify the partitioning of the physical system. Consider the impact of the technical, configuration management and reuse issues.
    - Model any executables and libraries as components, using the appropriate standard elements.
    - Model the significant interfaces that some components use and others realize.
    - As necessary to communicate your intent, model the relationships among these executables, libraries and interfaces.



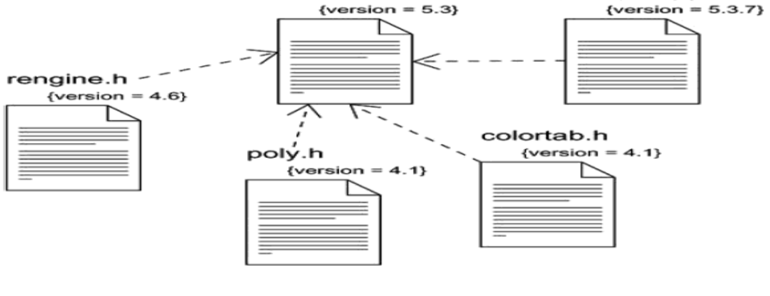
- **Modeling Tables, Files and Documents**
  - Identify the components that are part of the physical implementation of your system.
  - Model these things as components.
  - As necessary to communicate your intent, model the relationships among these components and other executables, libraries and interfaces in the system



- **Modeling an API**
  - Identify the programmatic seems in the system and model each seem as an interface.
  - Expose only those properties of the interface that are important to visualize the given context. Otherwise, hide these properties, keeping them in the interface's specification for reference, as necessary.
  - Model the realization of each API only as it is important to show the configuration of a specific implementation.

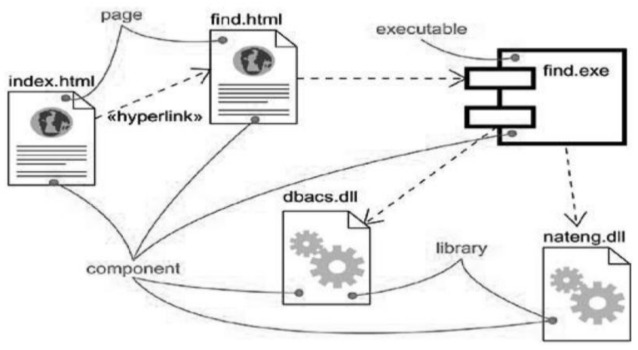


- **Modeling Source Code**
  - Depending on the constraints imposed by your development tools, model the files used to store the details of all your logical elements, along with their compilation dependencies.
  - Use tagged values if you want to use configuration management and version control tools.
  - As far as possible, let your development tools manage the relationships among these files, and use the UML only to visualize and document these relationships.



Component diagrams

- ✓ Component diagrams are one of the two kinds of diagrams for modeling the physical aspects of object-oriented software systems.
- ✓ A component diagram shows the organization and dependencies among a set of components.
- ✓ We use component diagrams to model the static implementation view of a software system.

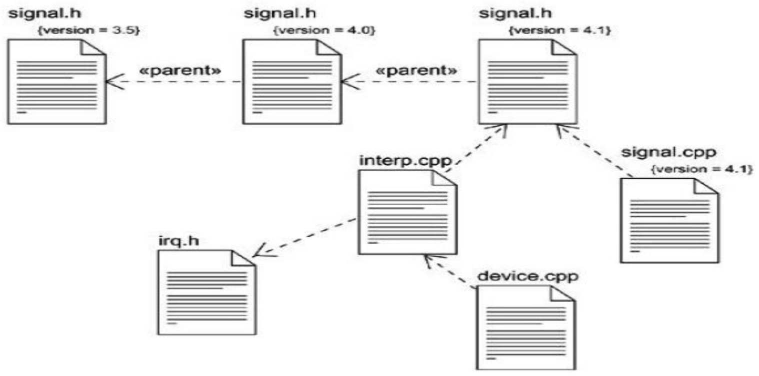


- ✓ **Common Properties**
  - o A component is just a special kind of a diagram and shares the same common properties as the other diagrams like: a name and graphical contents. What distinguishes a component diagram from the rest of the diagrams is its content.
- ✓ **Content**
  - o Component diagram commonly contain:
    - ▪ Components
    - ▪ Interfaces
    - ▪ Dependency, generalization, association and realization relationships.
- ✓ **Common Uses**
  - o When modeling the static implementation view of a system, we will typically use component diagrams in one of four ways:
    1. To model source code.
    2. To model executable releases.
    3. To model physical databases.
    4. To model adaptable systems.

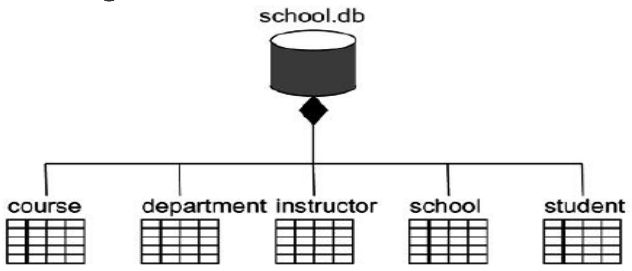
- ✓ **Common Modeling Techniques**
  - o **Modeling source code**
    - ▪ Either by forward or reverse engineering identifies the set of source code files of interest and model them as components stereotypes as files.
    - ▪ For larger systems, use packages to show groups of source code files.
    - ▪ Consider using tagged values indicating such information as the version number of the source code file, its author, and the date it was last changed.
    - ▪ Model the compilation dependencies among these files using dependencies.

  - o **Modeling an executable release**
    - ▪ Identify the set of components you'd like to model.
    - ▪ Consider the stereotype of each component in this set.

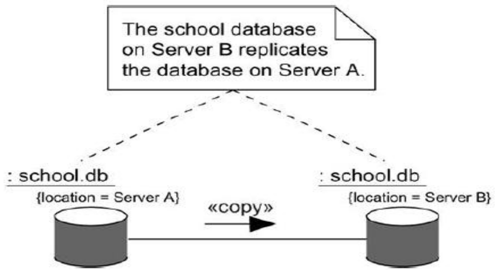
- For each component in this set, consider its relationship to its neighbors. Most, often this will involve interfaces that are realized by certain components and then imported by others.



- **Modeling a physical database**
  - Identify the classes in your model that represent your logical database schema.
  - Select a strategy for mapping these classes to tables. You have to also consider the physical distribution of your databases.
  - To visualize, specify, construct and document your mapping, create a component diagram that contains components stereotyped as tables.
  - Where possible, use tools to help you transform your logical design into a physical design.

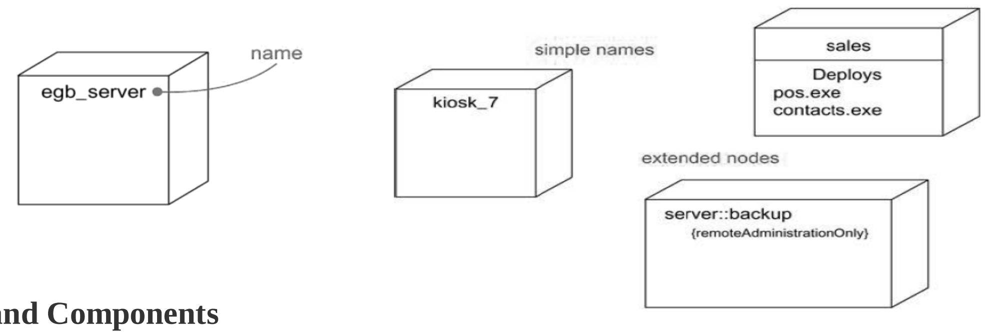


- **Modeling adaptable systems**
  - Consider the physical distribution of the components that may migrate from node to node. We can specify the location of a component instance by marking it with a location tagged value.
  - If you want to model the actions that cause a component to migrate, create a corresponding interaction diagram that contains component instances. We can illustrate a change of location by drawing the same instance more than once, but with different values for its location tagged value.

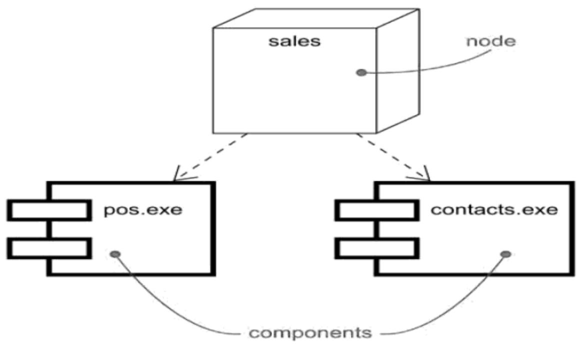


Deployment

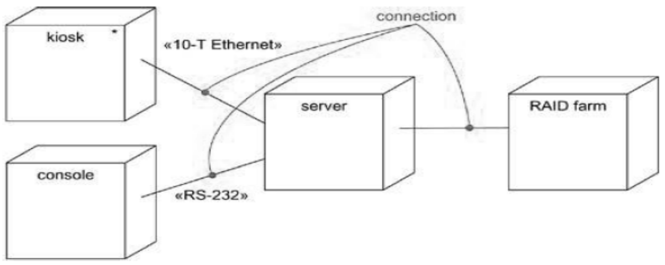
- ✓ A node is a physical element that exists at runtime and represents a computational resource, generally having atleast some memory and, often, processing capability. We use nodes to model the topology of the hardware on which our system executes. A node typically represents a processor or a device on which components may be deployed.
- ✓ When we architect a software-intensive system, we have to consider both its logical and physical dimensions. On the logical side, you'll find things such as classes, interfaces, collaborations, interactions and state machines. On the physical side you'll find components and nodes.
- ✓ In UML, a node is represented as a cube as shown below. Using stereotypes we can tailor this notation to represent specific kinds of processors and devices.



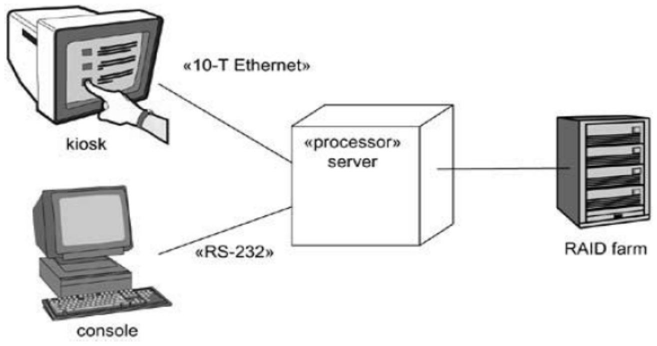
- ✓ **Nodes and Components**
  - In many ways, nodes are like components: Both have names, both may participate in dependency, generalization and association relationships. Both may be nested, both may have instances, both may be participants in interactions. However, there are significant differences between nodes and components:
    - Components are things that participate in the execution of a system. Nodes are things that execute components.
    - Components represent the physical packaging of logical elements, nodes represent the physical deployment of components.
  - This first difference is the most important. Simply put, nodes execute components; components are things that are executed by nodes.
  - A set of objects or components that are allocated to a node as a group is called a distribution unit.



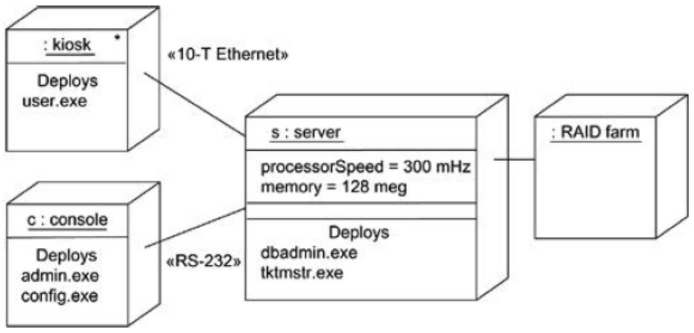
- ✓ **Connections**
  - The most common kind of relationship we'll use among nodes is an association. In this context, an association represents a physical connection among nodes, such as an Ethernet connection, a serial line, or a shared bus as shown below. We can even use associations to model indirect connections, such as a satellite link between distant processors.



✓ **Common Modeling Techniques**
- Modeling processors and devices
  - Identify the computational elements of your system's deployment view and model each as a node.
  - If these elements represent generic processors and devices, then stereotype them as such. If they are kinds of processors and devices that are part of the vocabulary of your domain, then specify an appropriate stereotype with an icon for each.
  - As with class modeling, consider the attributes and operations that might apply to each node.

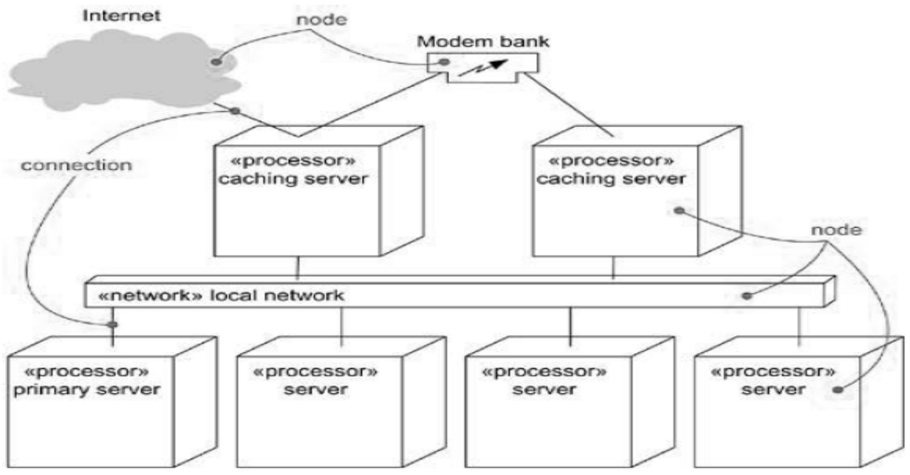


- Modeling the distribution of components
  - For each significant component in your system, allocate it to a given code.
  - Consider duplicate locations for components.
  - Render this allocation in one of the three ways:
    1. Don't make the allocation visible, but leave it as part of the backplane of your model that is, in each node's specification.
    2. Using dependency relationships, connect each node with the components it deploys.
    3. List the components deployed on a node in an additional compartment.



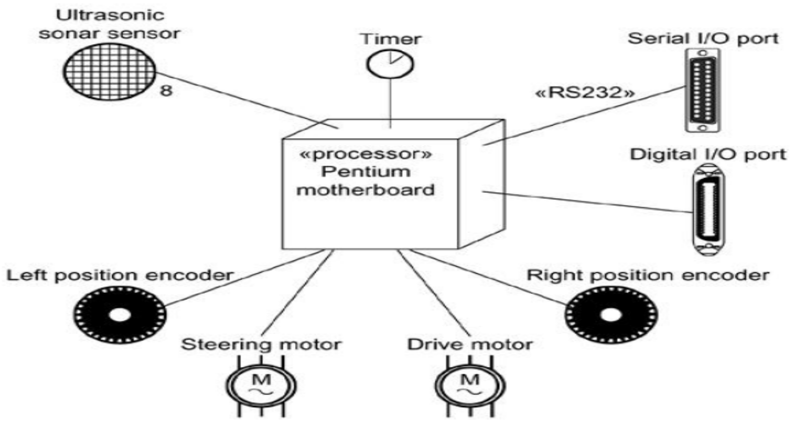
Deployment diagrams

- ✓ Deployment diagrams are one of the two kinds of diagrams used in modeling the physical aspects of an object-oriented system. A deployment diagram shows the configuration of run time processing nodes and the components that live on them.
- ✓ We use deployment diagrams to model the static deployment view of a system. A deployment diagram is a diagram that shows the configuration of run time processing nodes and the components that live on them.

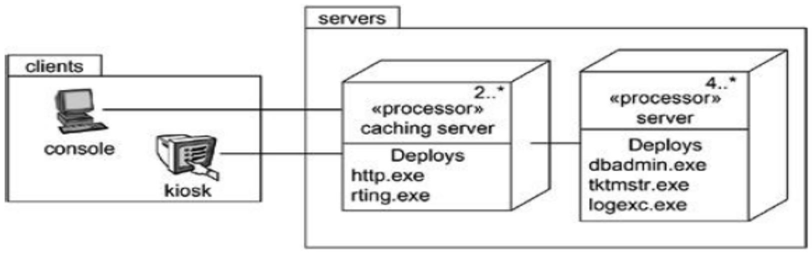


- ✓ **Common Properties**
  - A deployment is just a special kind of diagram that shares the same properties as all other diagrams like: a name and graphical contents. What distinguishes a deployment diagram from the rest of the diagrams is its content.
- ✓ **Contents**
  - A deployment diagram commonly contains:
    - Nodes
    - Dependency and association relationships
- ✓ **Common Uses**
  - When modeling the static deployment view of a system, we'll typically use deployment diagrams in one of the three ways:
    1. To model embedded systems.
    2. To model client/server systems.
    3. To model fully distributed systems.

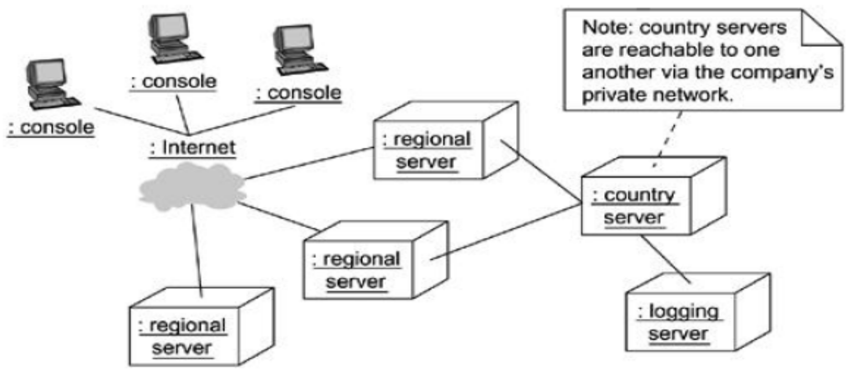
- ✓ **Common Modeling Techniques**
  - **Modeling an embedded system**
    - Identify the devices and nodes that are unique to your system.
    - Provide visual cues, especially for unusual devices, by using stereotypes.
    - Model the relationships among these processors and devices in a deployment diagram. Similarly, specify the relationship between components and nodes.
    - As necessary, expand on the intelligent devices by modeling their structure with a more detailed deployment diagram.



- **Modeling a client/server system**
  - Identify the nodes that represent your system's client and server processors.
  - Highlight those devices that are relevant to the behavior of your system.
  - Provide visual cues for these processors and devices via stereotyping.
  - Model the topology of these nodes in a deployment diagram.



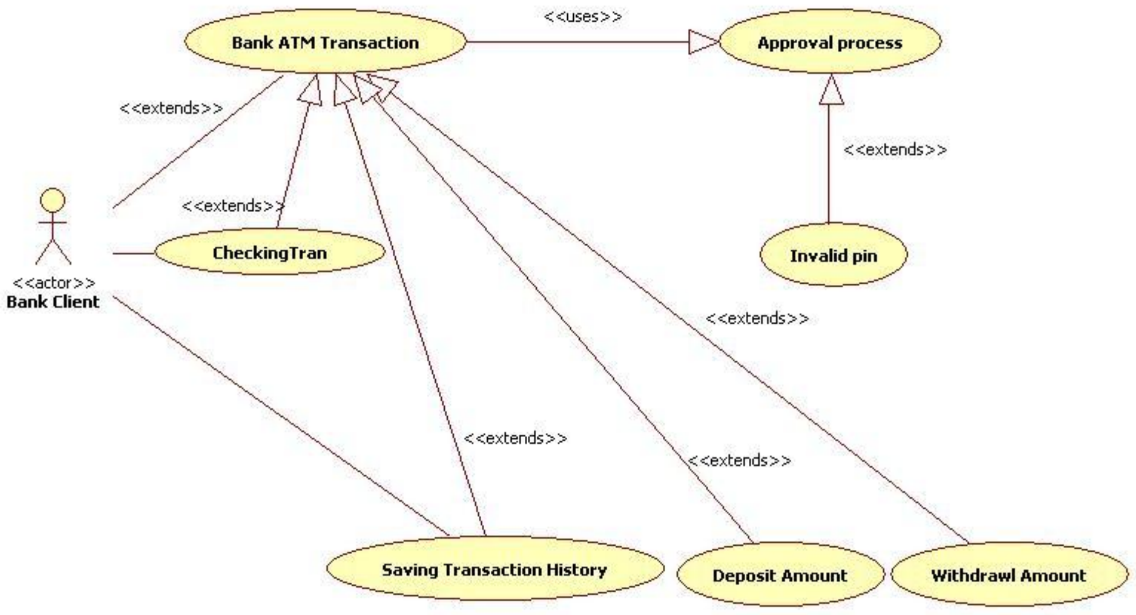
- **Modeling a fully distributed system**
  - Identify the system's devices and processors as for simpler client/server systems.
  - If you need to reason about the performance of the system's network or the impact of changes to the network, be sure to model these communication devices to the level of detail sufficient to make assessments.
  - Pay close attention to logical groupings of nodes, which you can specify by using packages.
  - Model these devices and processors using deployment diagrams.
  - If you need to focus on the dynamics of the system, introduce use case diagrams to specify the kind of behavior you are interested in, and expand on these use cases with interaction diagrams.



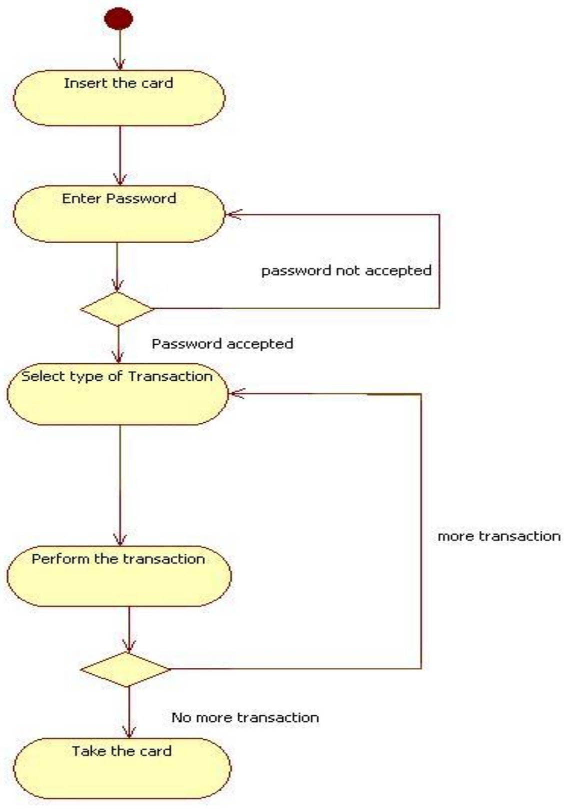
Syllabus: Unit – V: Case Studies
Analysis and Design of ATM system using object oriented approach - Analysis and Design of Library management system using object oriented approach - Analysis and Design of Online Railway reservation system using object oriented approach

ATM (Automatic Teller Machine)

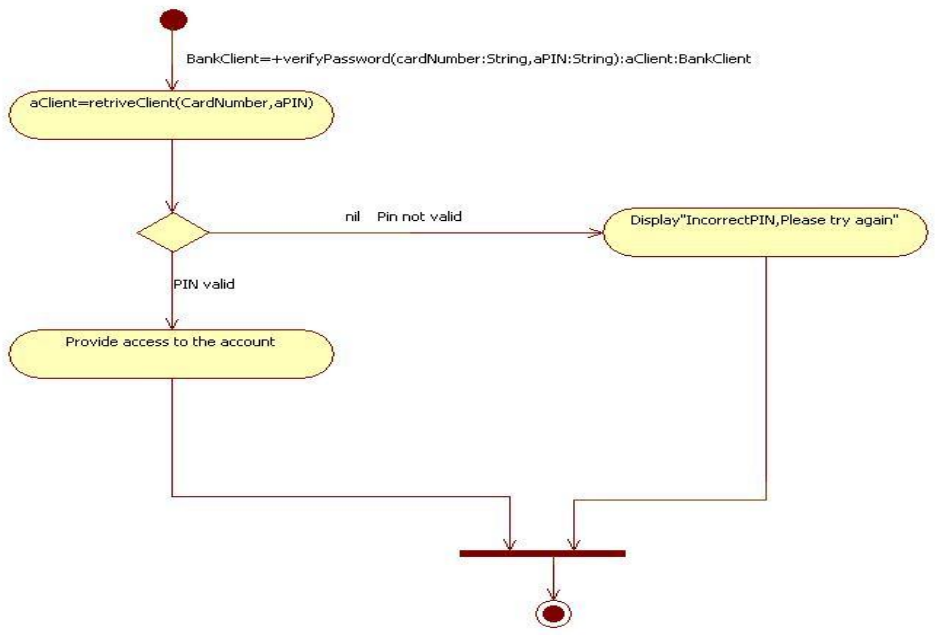
Use Case Diagram:



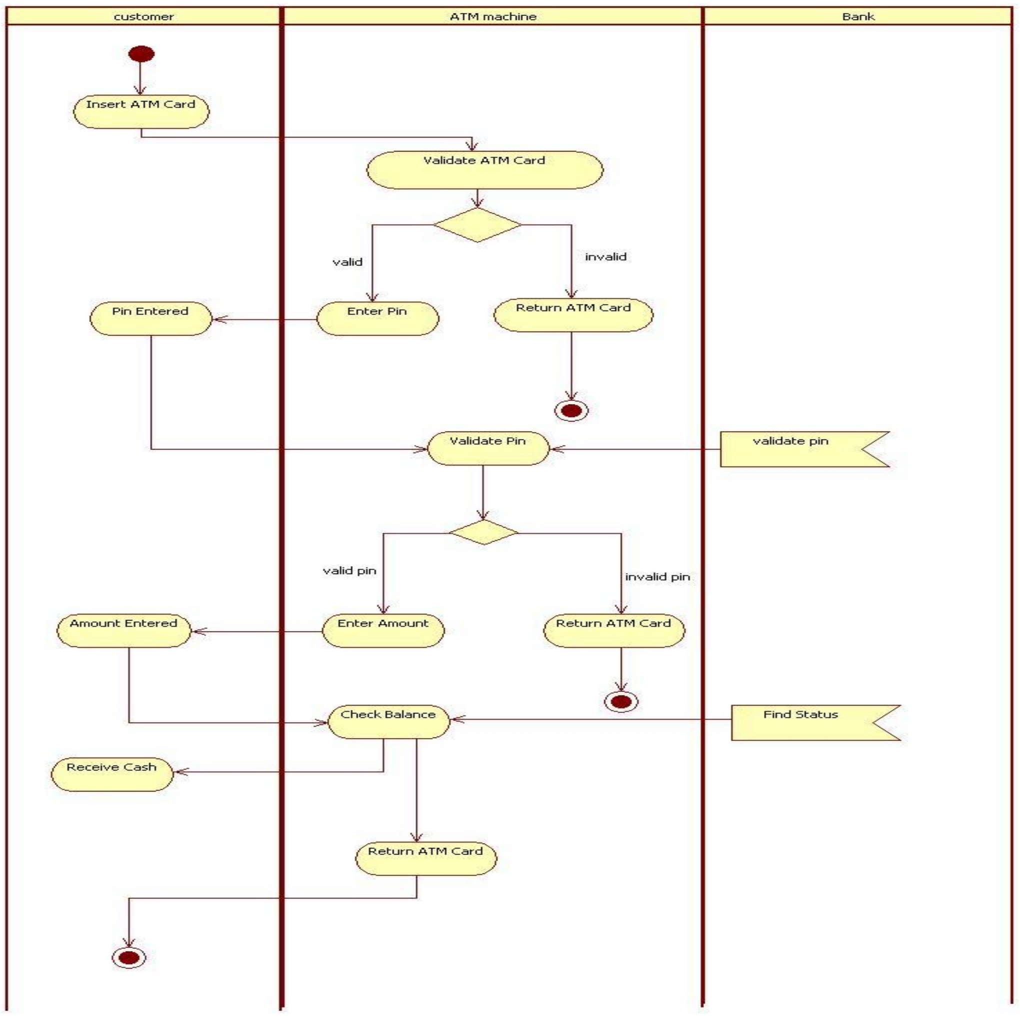
Activity Diagram for Overall ATM Machine



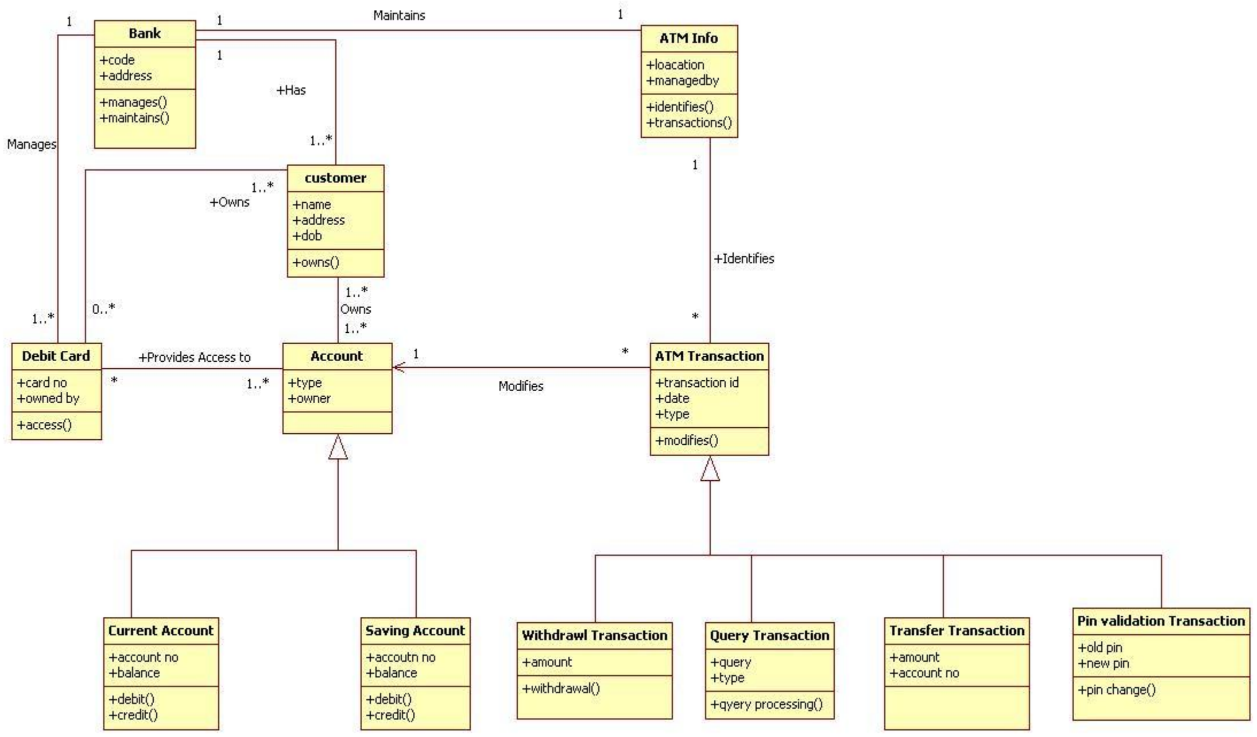
Activity Diagram for Verify Password ATM Machine


BankClient=+verifyPassword(cardNumber:String,aPIN:String):aClient:BankClient
aClient=retriveClient(CardNumber,aPIN)
nil Pin not valid
Display"IncorrectPIN,Please try again"
PIN valid
Provide access to the account

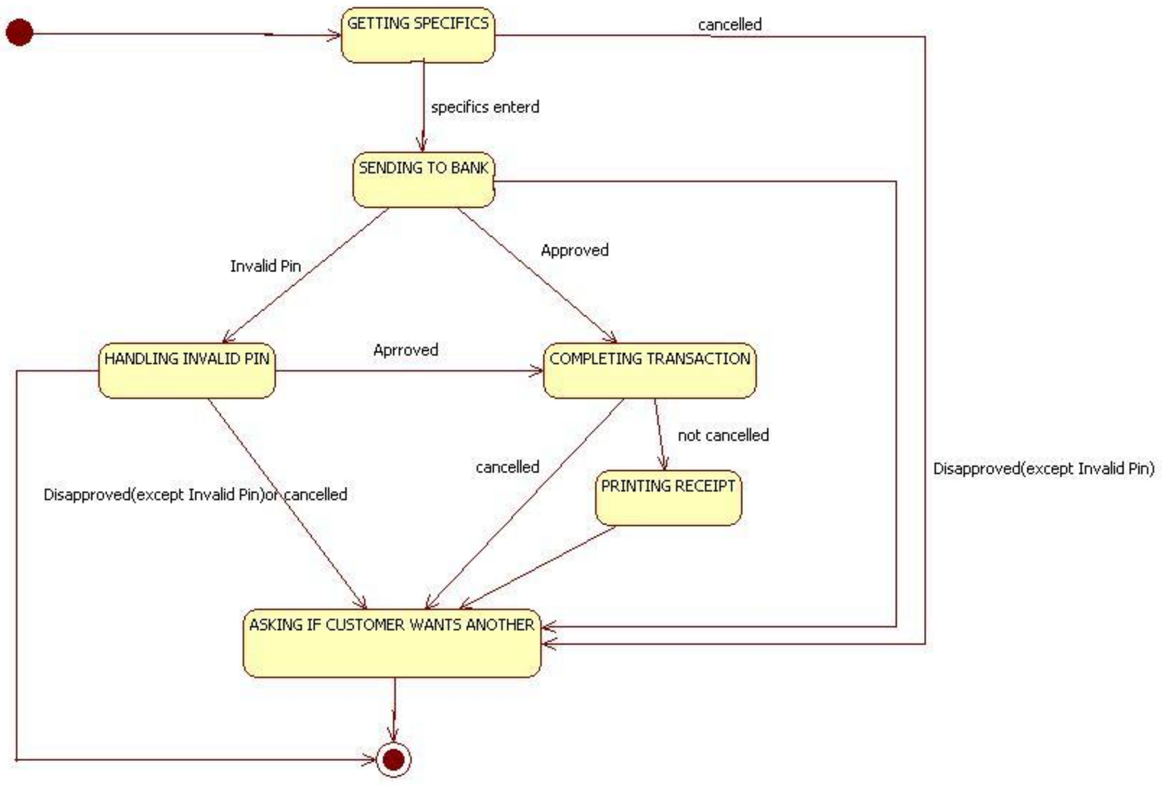
Activity Diagram for ATM Machine



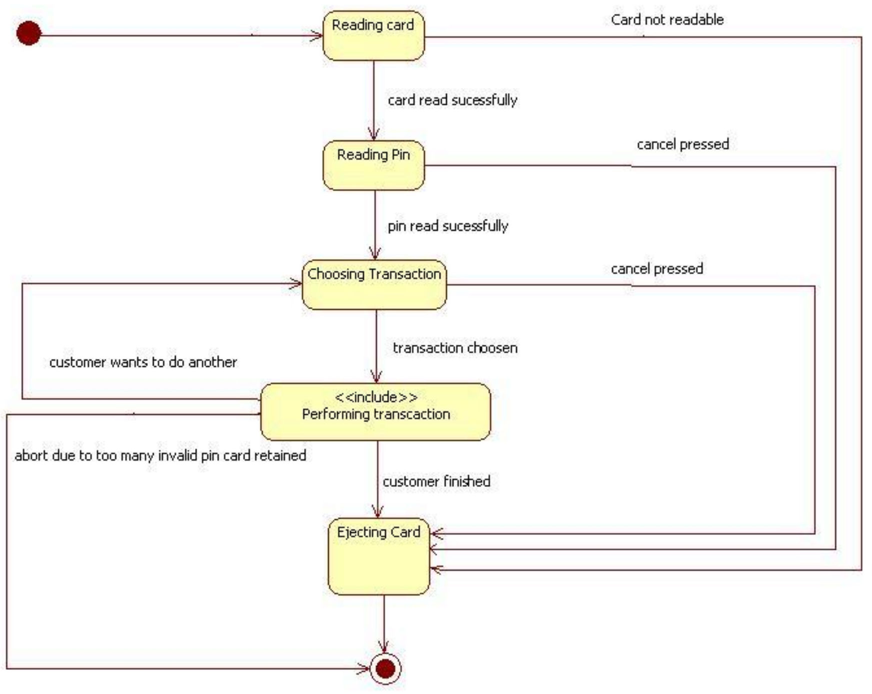
Class Diagram



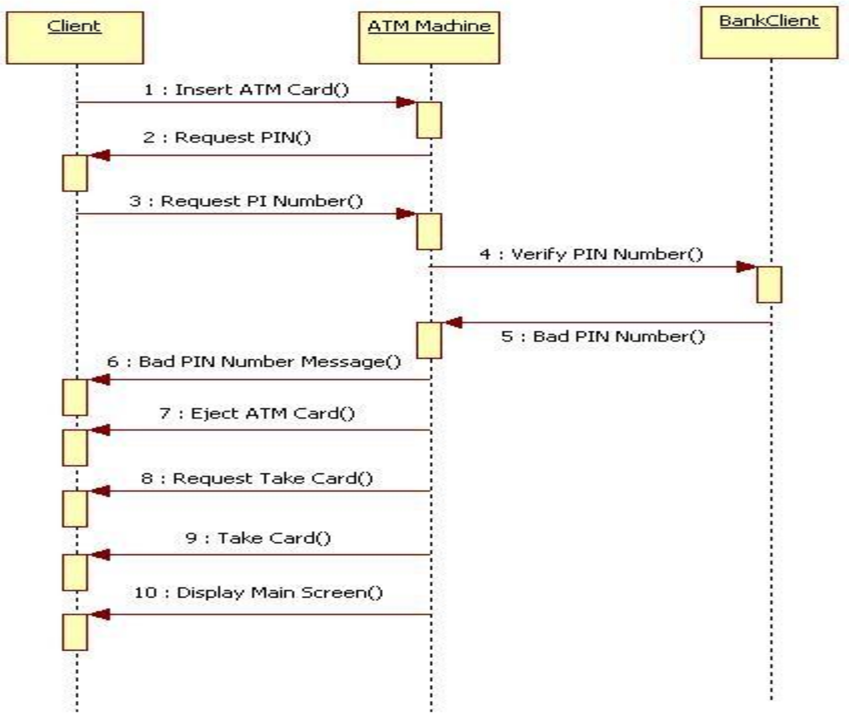
State diagram for One Transaction ATM machine



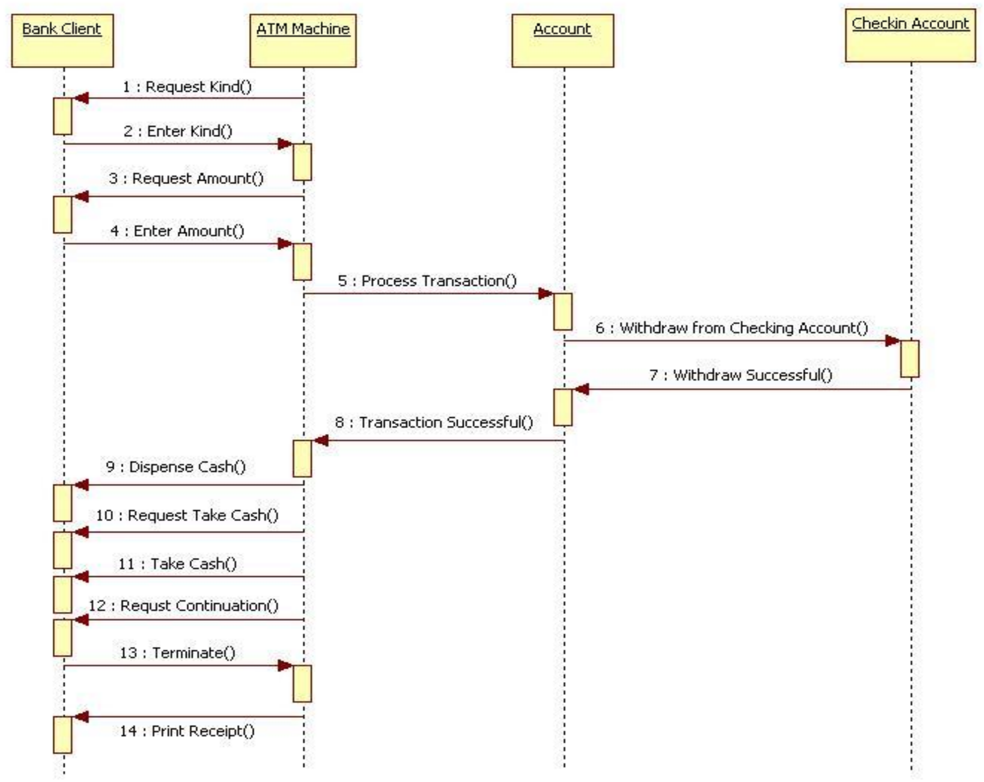
State diagram for one Session ATM Machine



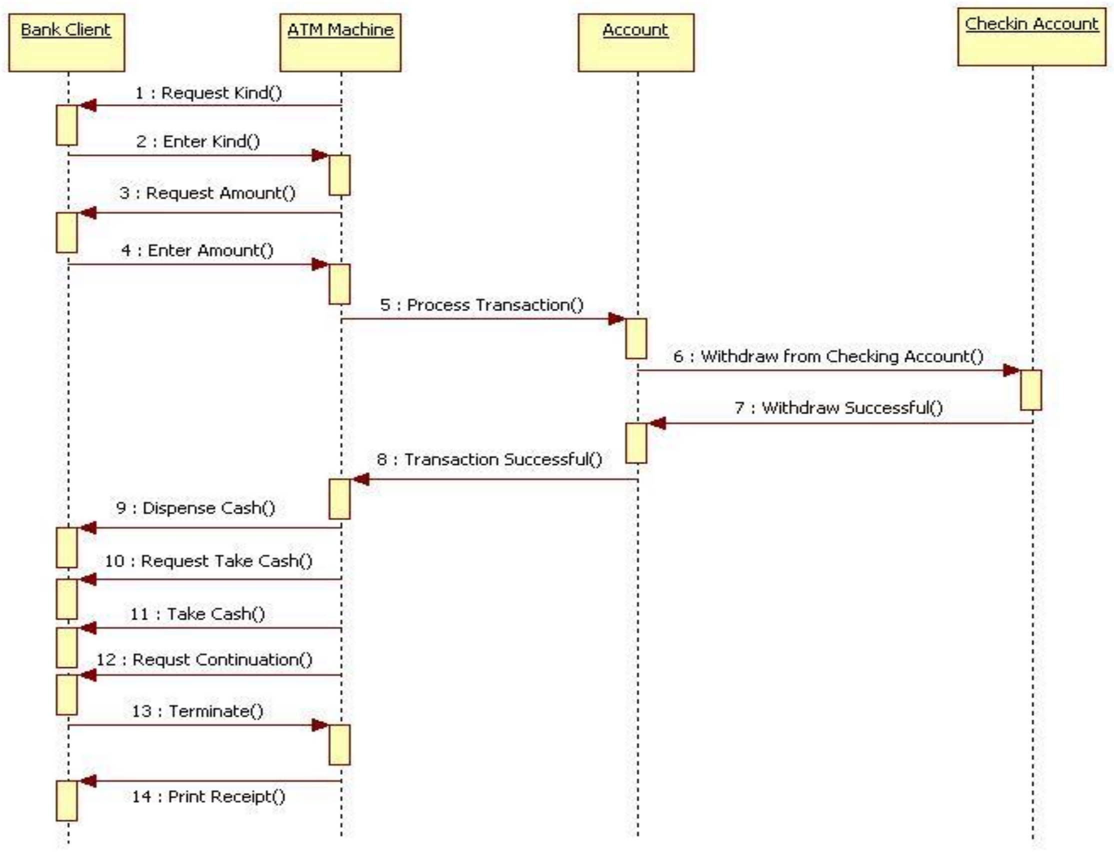
Sequence diagram for ATM Machine



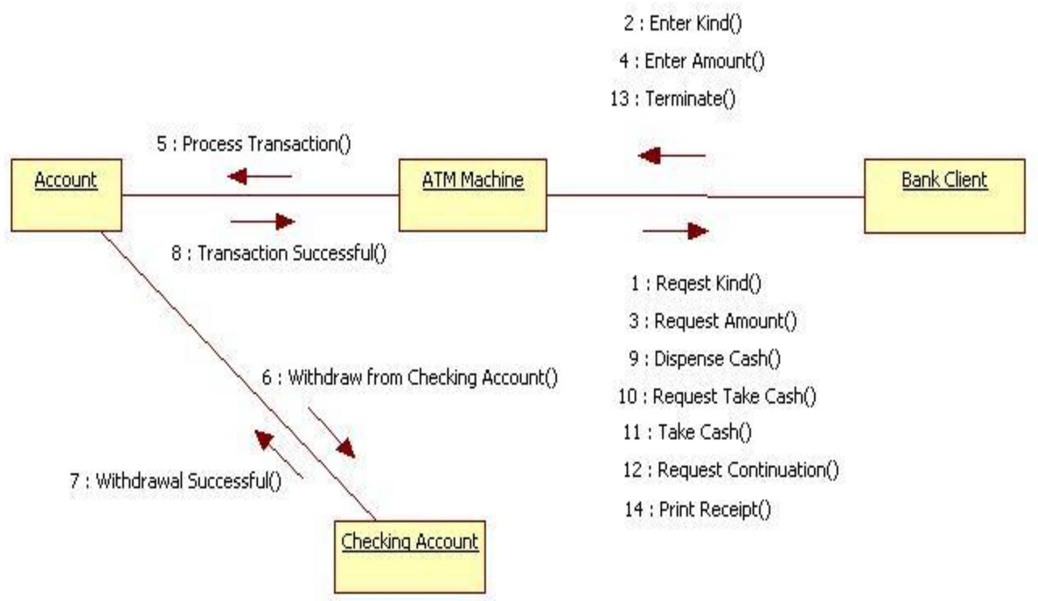
Sequence Diagram for Invalid Pin ATM Machine



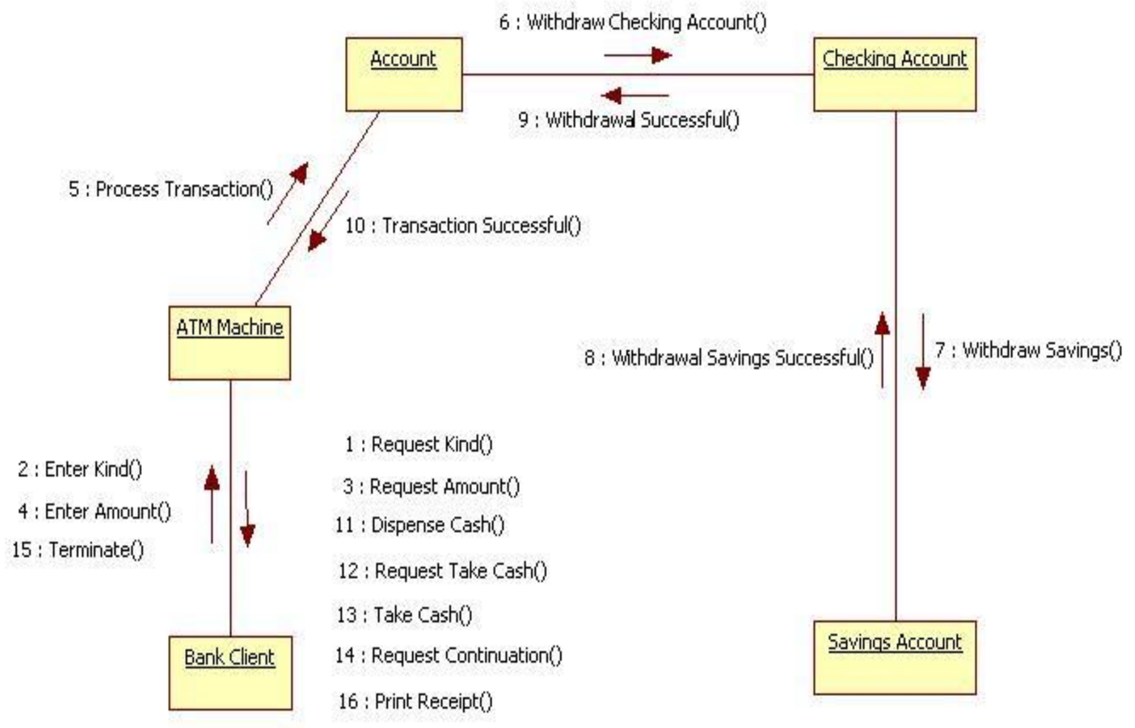
Sequence Diagram ATM withdrawal



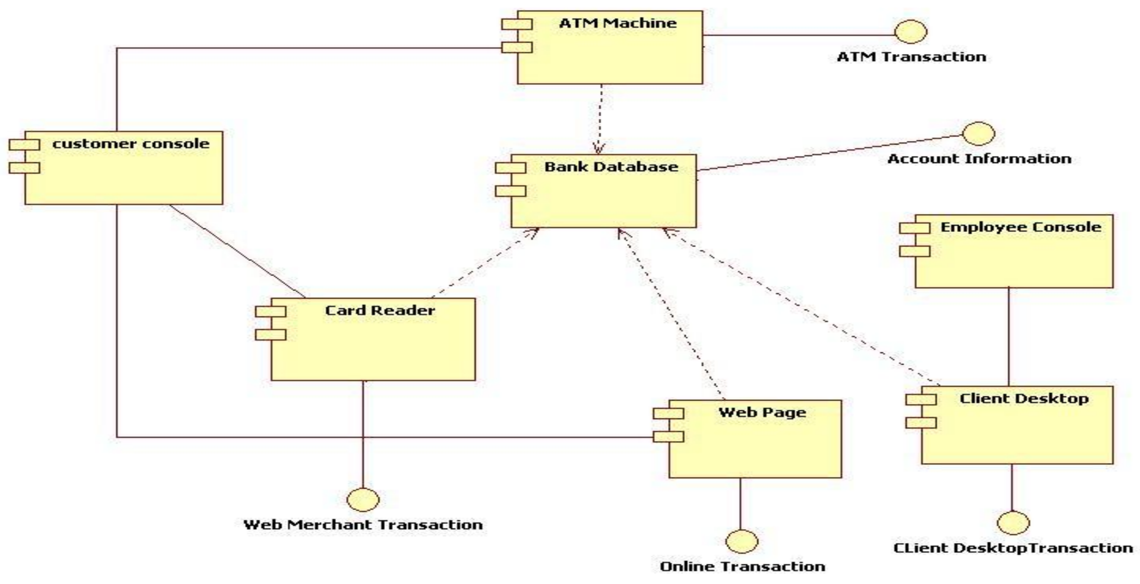
Collaboration Diagram for ATM machine



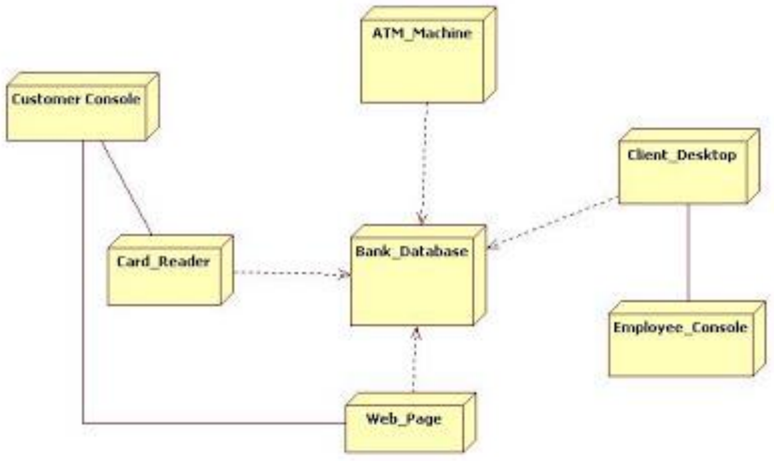
Collaboration Diagram for ATM withdrawal



Component diagram

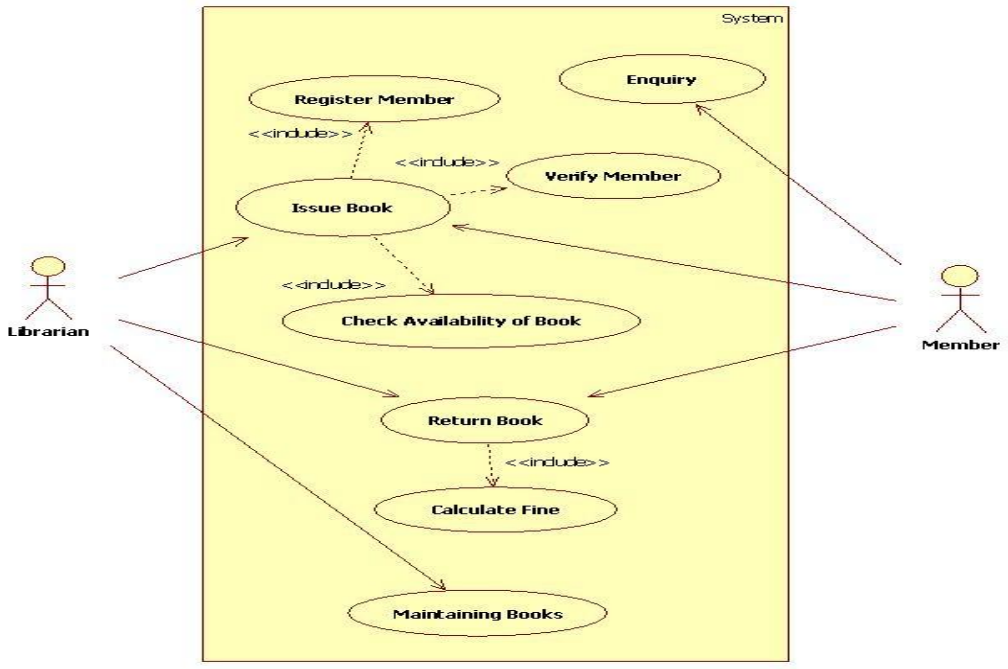


Deployment Diagram

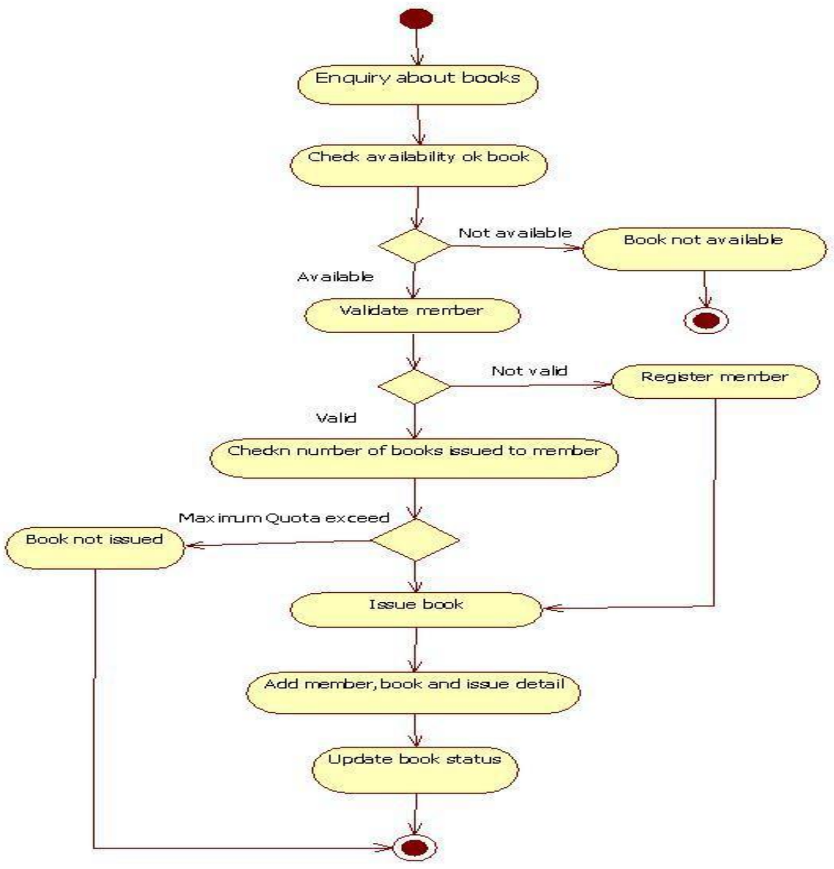


Library Management System

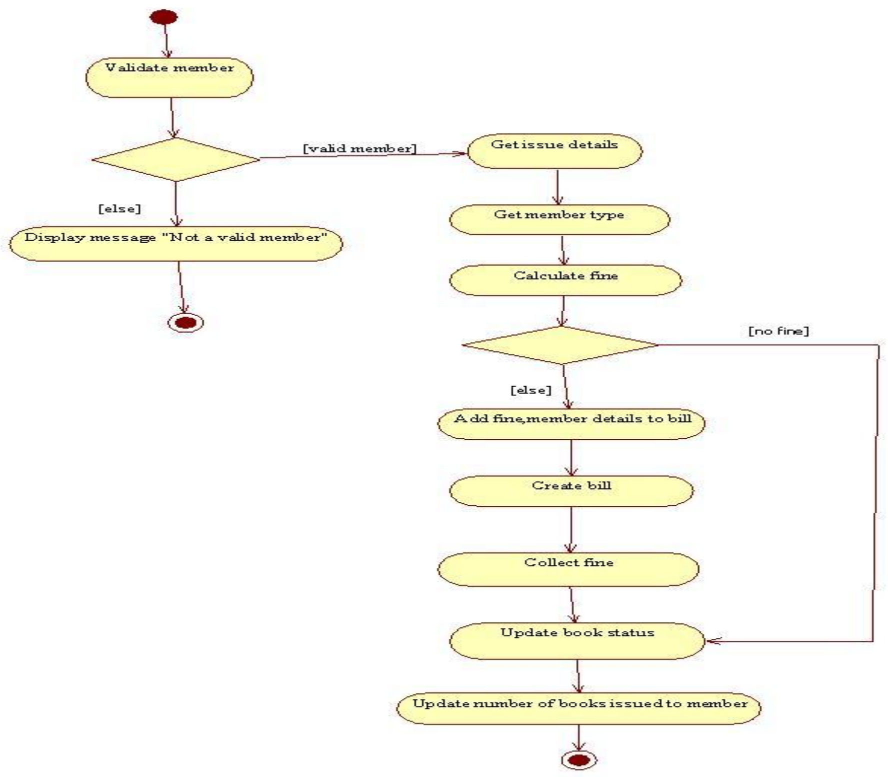
Use Case Diagram:



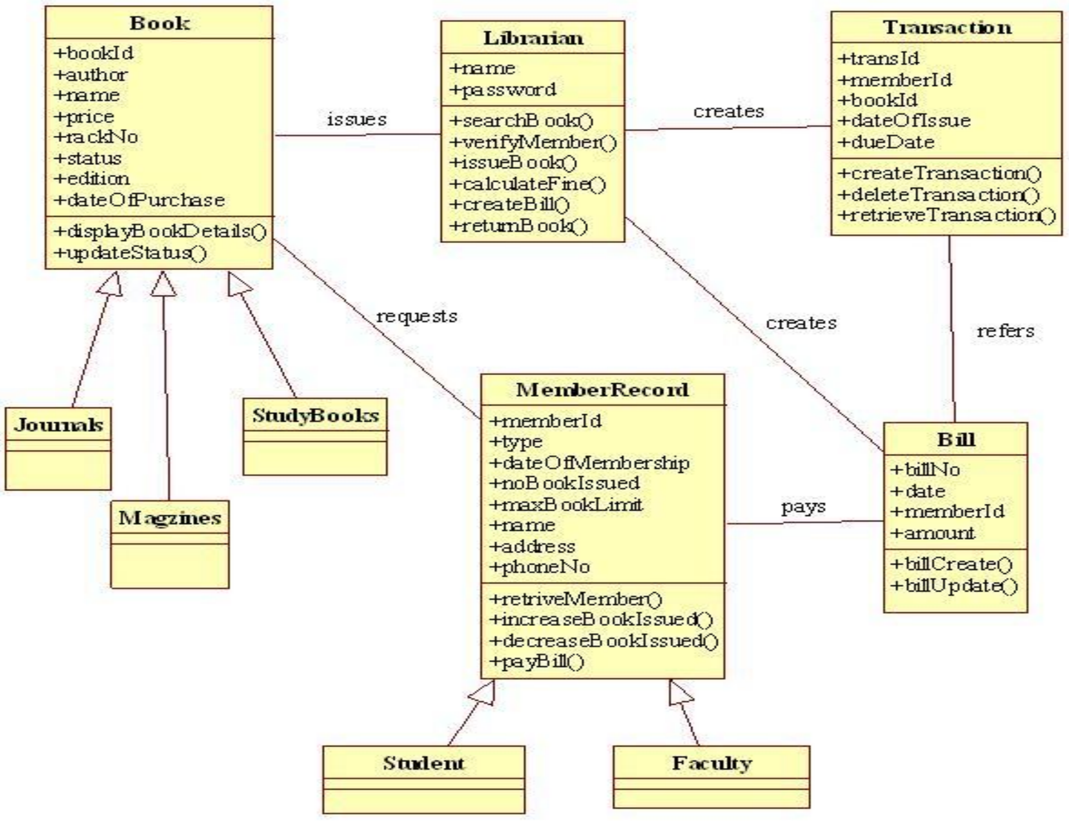
Activity Diagram for Issue Book



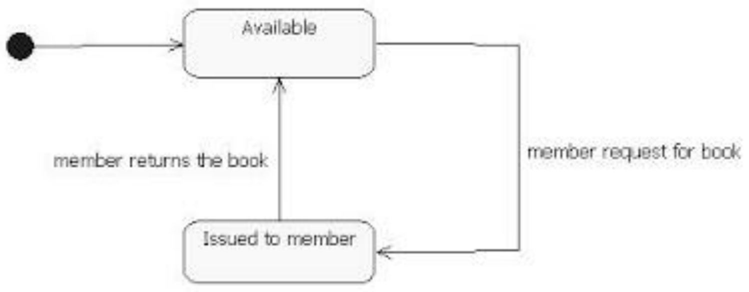
Activity Diagram for Return Book



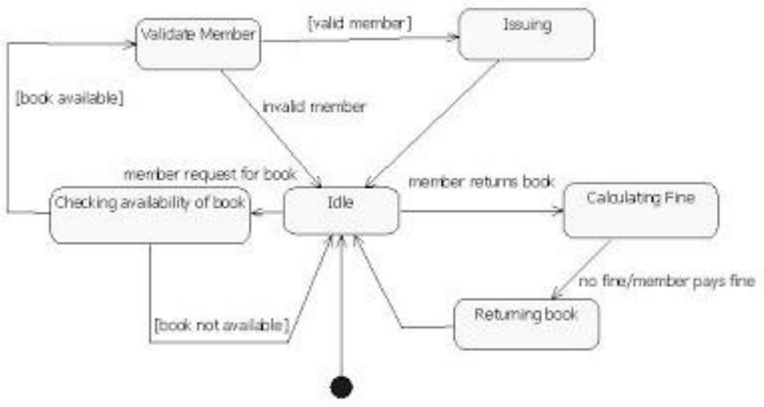
Class Diagram



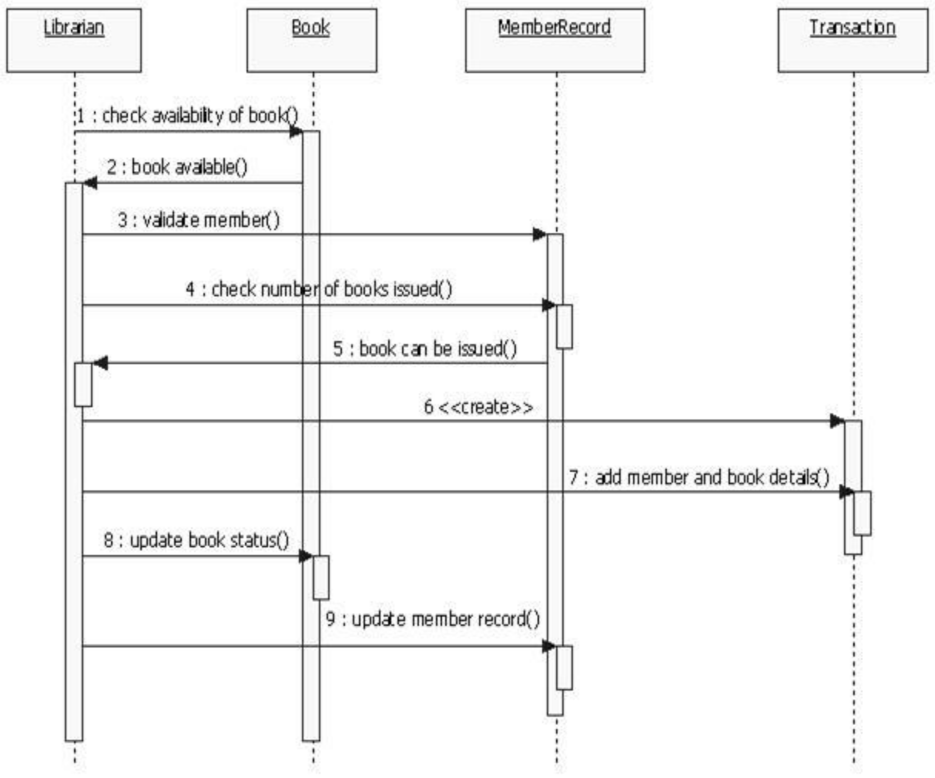
State diagram for Book



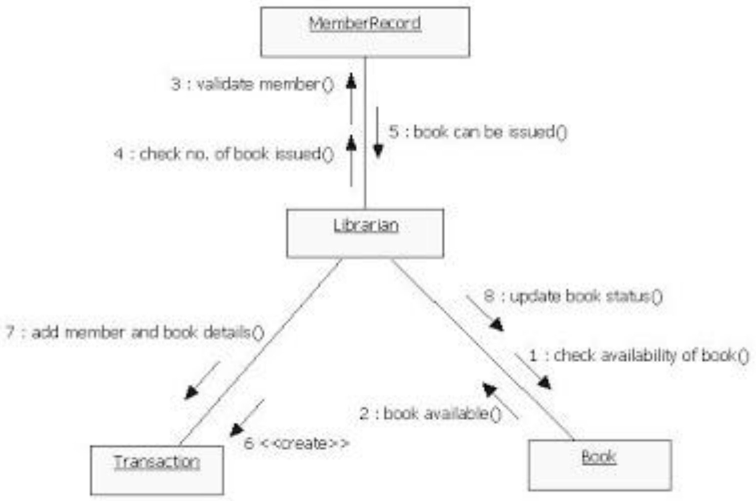
State diagram for Librarian



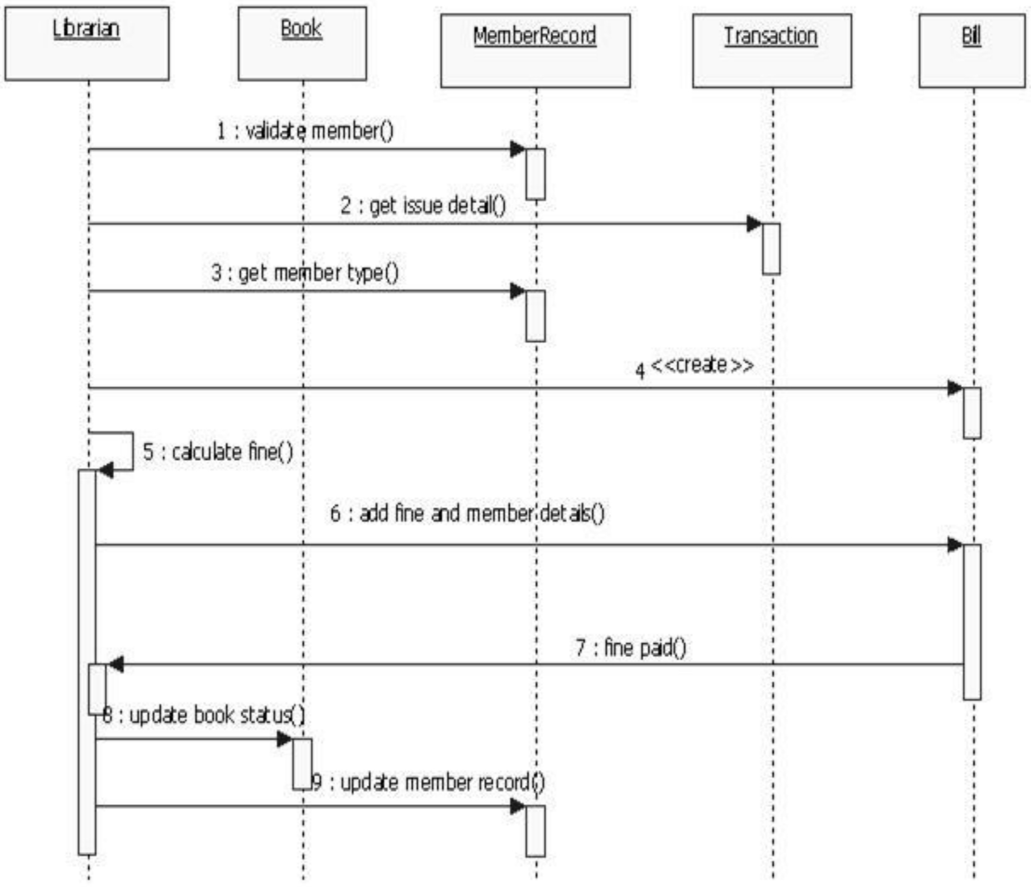
Sequence diagram for issuing book



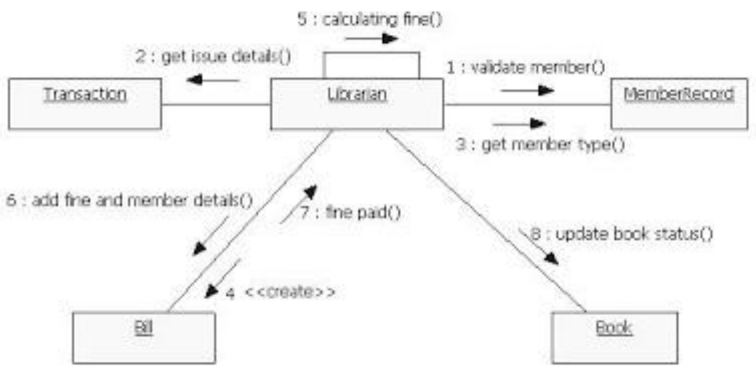
Collaboration diagram for issuing Book



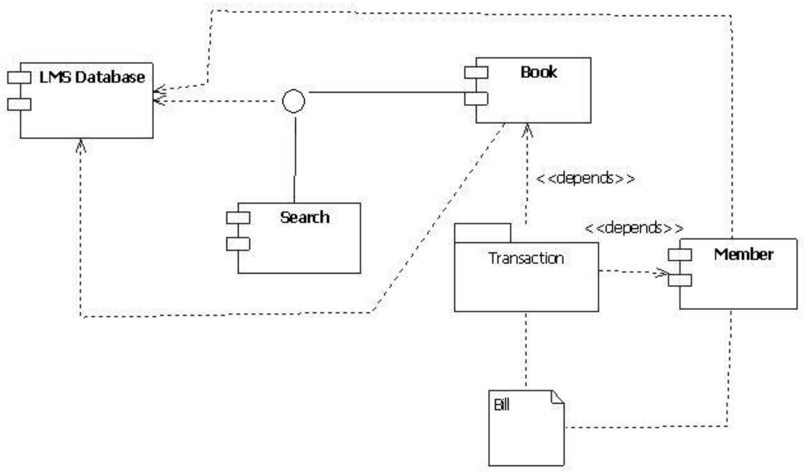
Sequence diagram for returning book



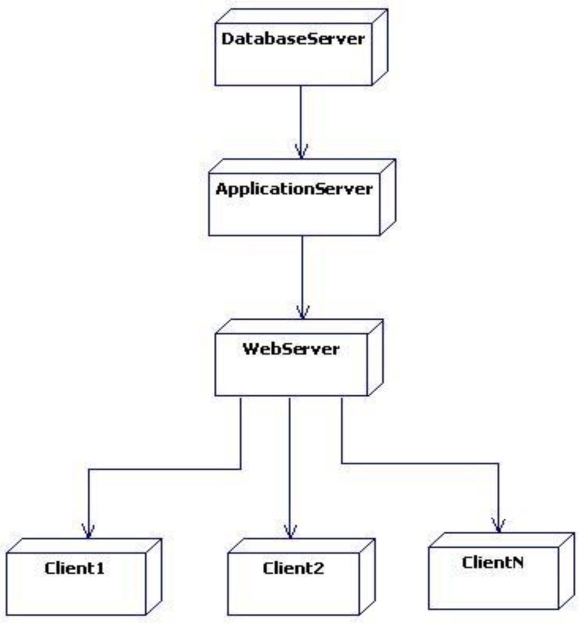
Collaboration diagram for returning Book



Component diagram

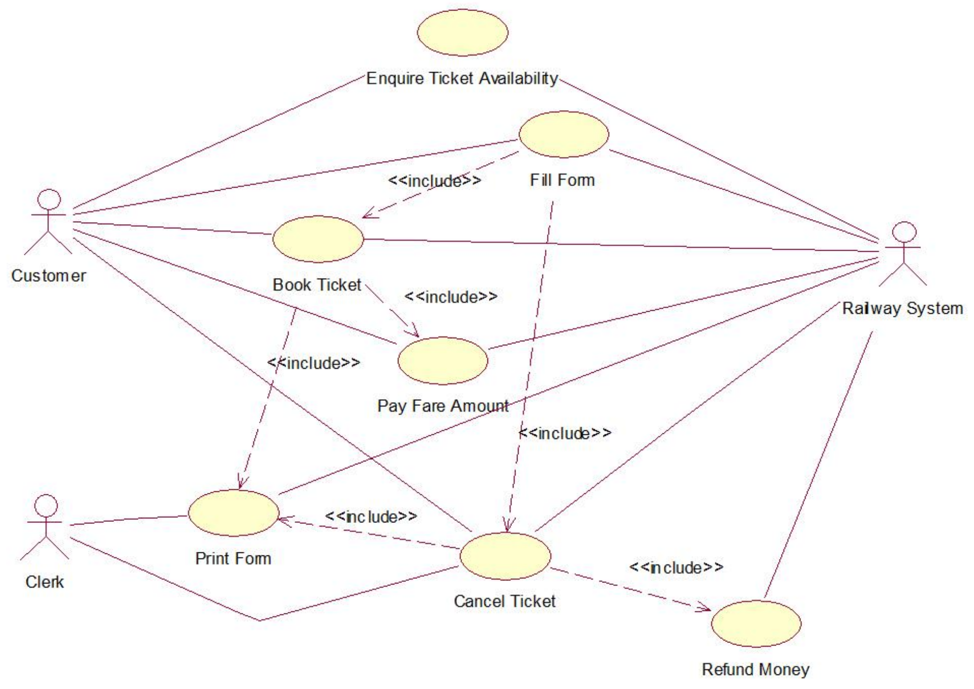


Deployment Diagram

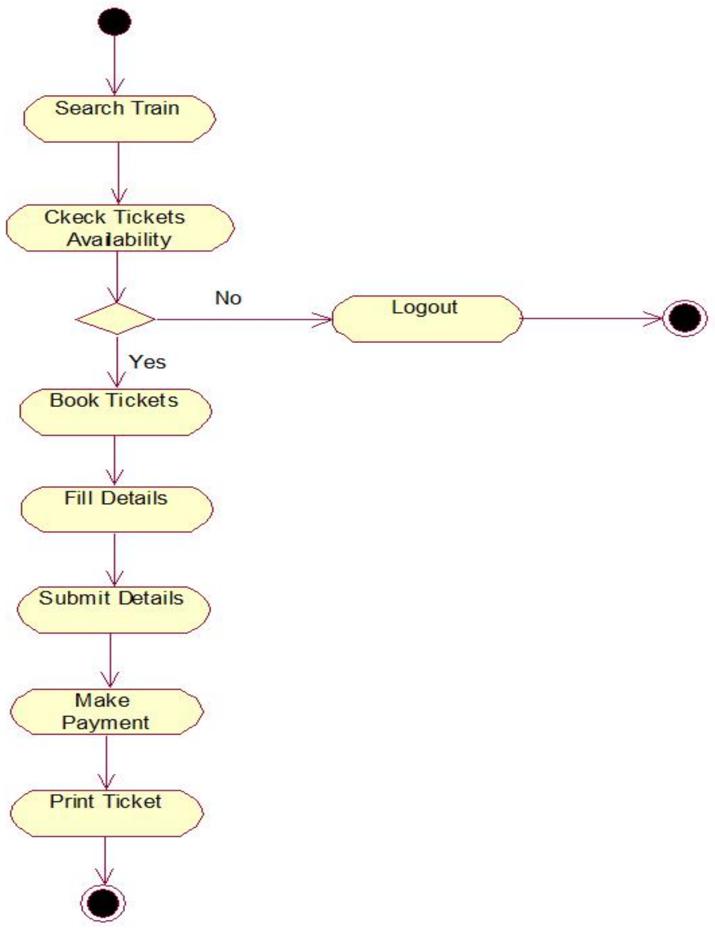


Online Railway reservation system

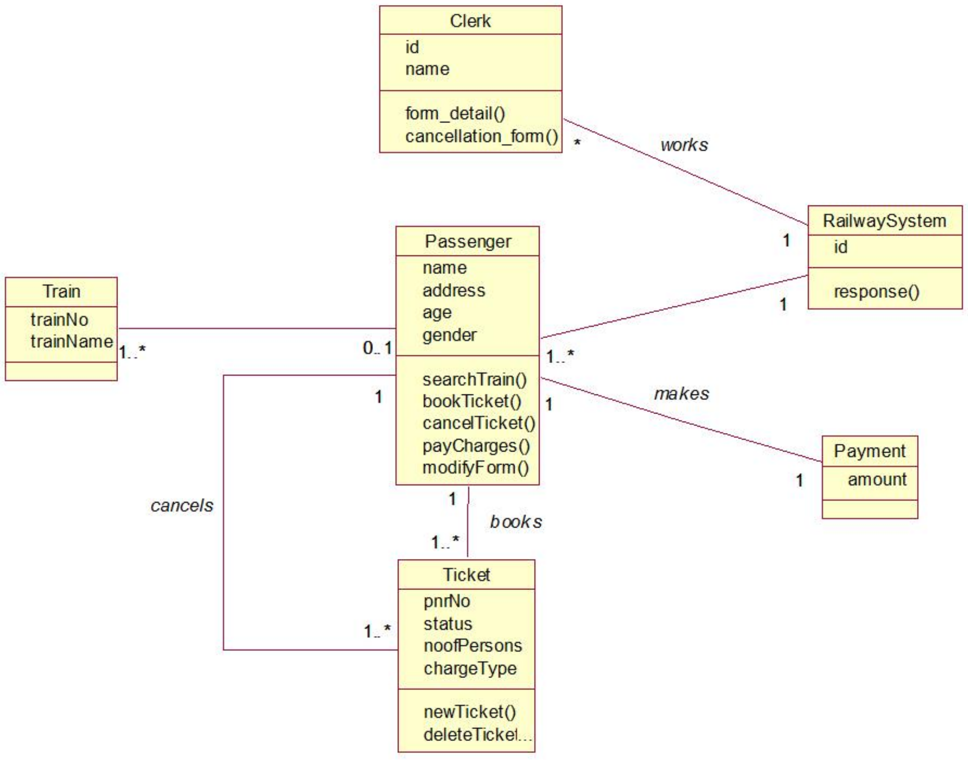
Use Case Diagram:



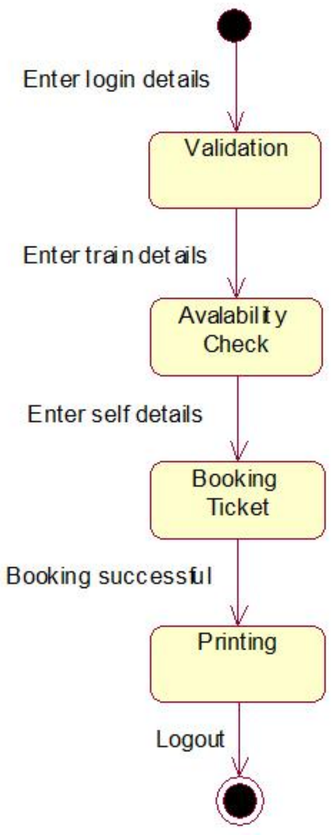
Activity Diagram:



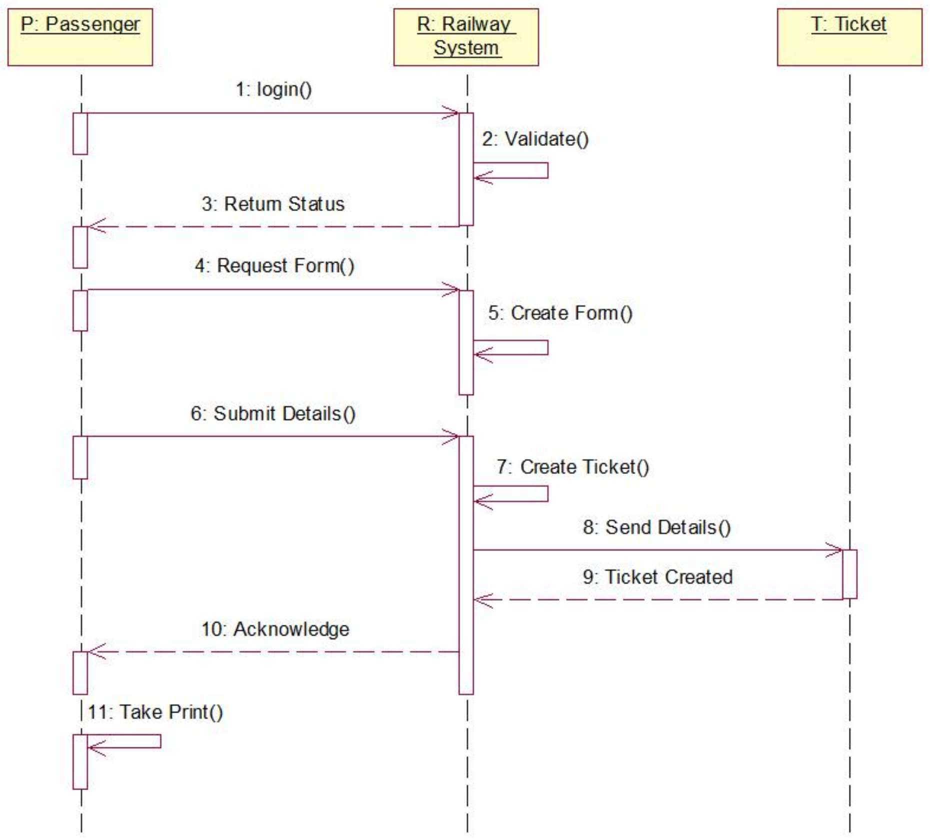
Class Diagram:



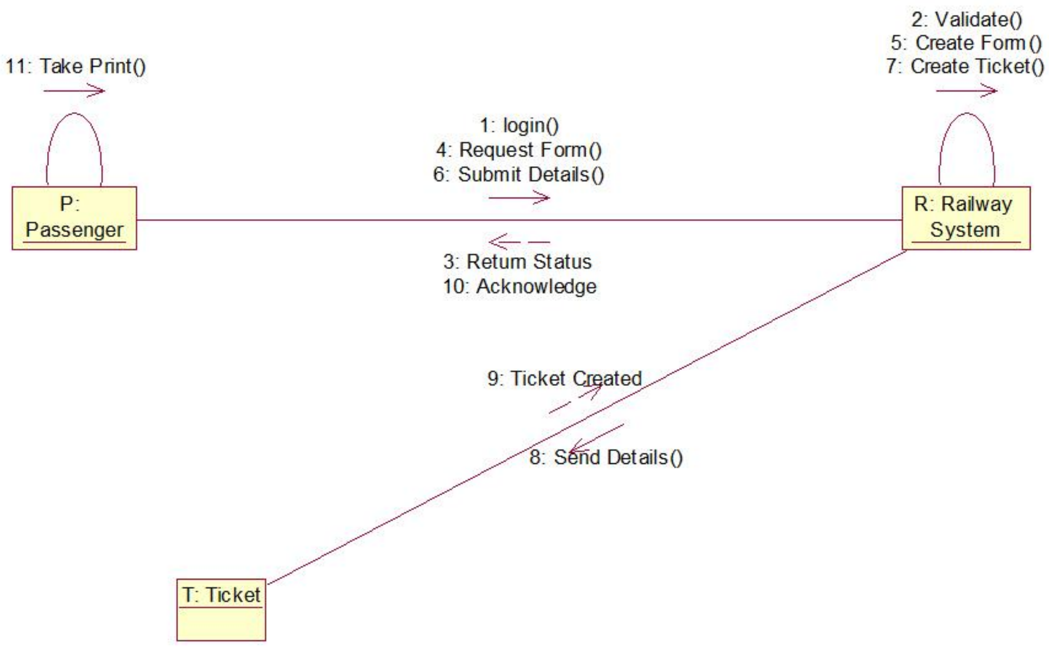
State diagram



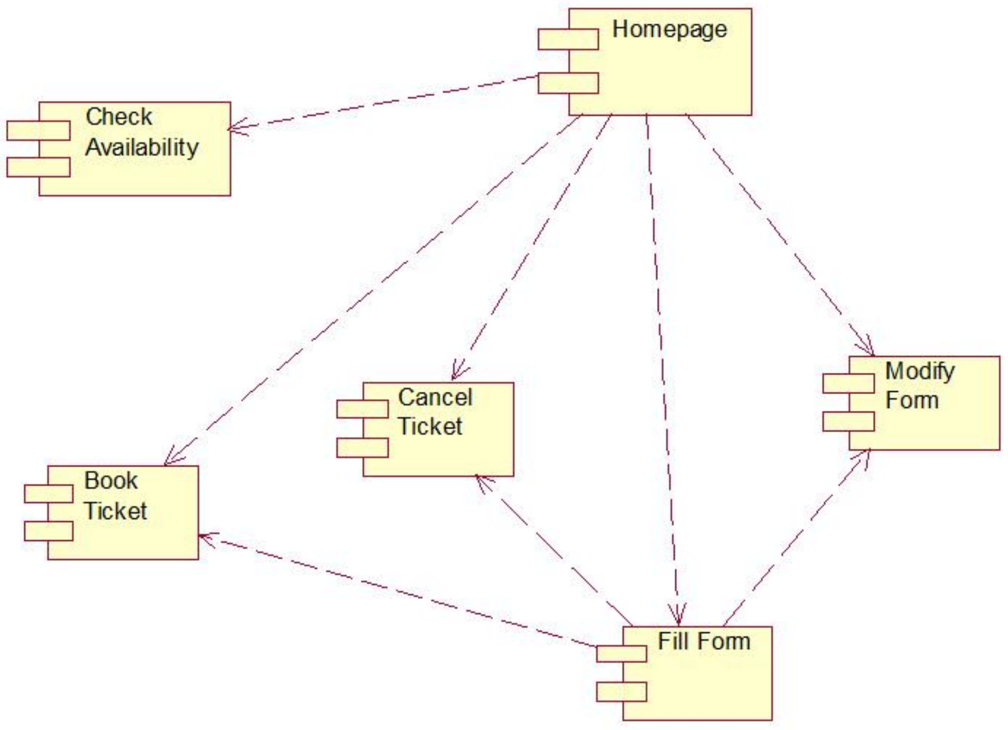
Sequence diagram:



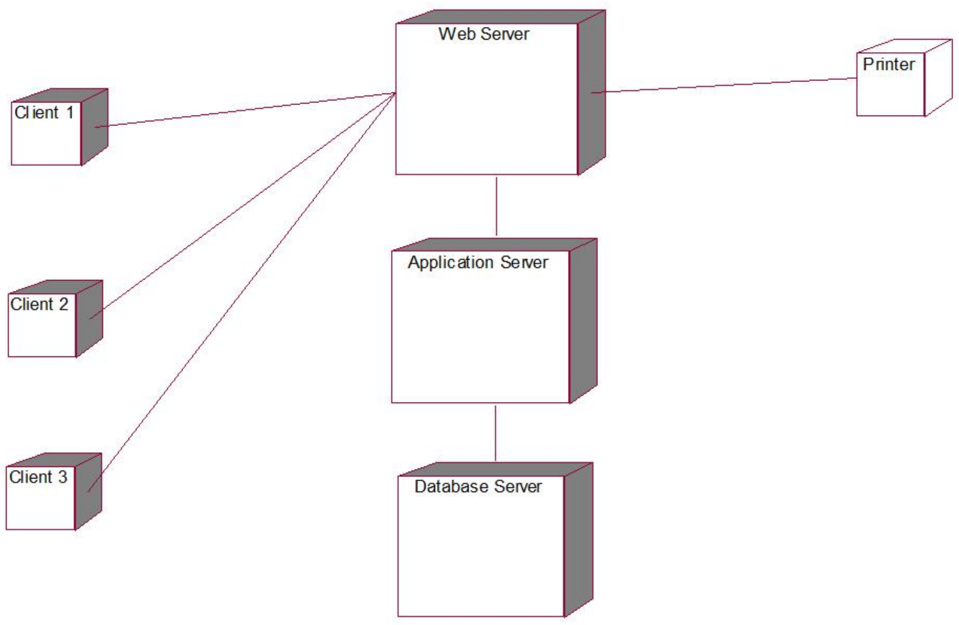
Collaboration Diagram:



Component diagram:



Deployment Diagram:


Web Server
Printer
Client 1
Client 2
Client 3
Application Server
Database Server