Object Oriented Design

Part 2

Program Design

- Analysis
- Design
- Implementation

Analysis Phase

- Functional Specification
  - Completely defines tasks to be solved
  - Free from internal contradictions
  - Readable both by domain experts and software developers
  - Reviewable by diverse interested parties
  - Testable against reality

Design Phase

- Goals
  - Identify classes
  - Identify behavior of classes
  - Identify relationships among classes
- Artifacts
  - Textual description of classes and key methods
  - Diagrams of class relationships
  - Diagrams of important usage scenarios
  - State diagrams for objects with rich state

Implementation Phase

- Implement and test classes
- Combine classes into program
- Avoid "big bang" integration
- Prototypes can be very useful

Problem 1:

- Design a voicemail system for use in your
- typical cellphone.
- How would the requirements look like?
- What would be a typical session?
- What modules are involved?

Identifying Classes in design

- Rule of thumb: Look for nouns in problem description

- Mailbox
- Message
- User
- Passcode
- Extension
- Menu

When defining classes

- Focus on concepts, not implementation
- ????? stores messages
  - Lets say a messageQueue
- Don't worry yet how the queue is implemented

Categories

- Tangible Things
- Agents
- Events and Transactions
- Users and Roles
- Systems
- System interfaces and devices
- Foundational Classes

Identifying Responsibilities

- Rule of thumb: Look for verbs in problem description
- Behavior of MessageQueue:
- Add message to tail
- Remove message from head
- Test whether queue is empty

OO Design

- OO Principle: Every operation is the responsibility of a single class
- Example:
  - Add message to mailbox
- Who is responsible:
  - Message or Mailbox?

Relationship

- Dependency ("uses")
- Aggregation ("has")
- Inheritance ("is")

Dependency

- C depends on D: Method of C manipulates objects of D
- Example: Mailbox depends on Message
- If C doesn't use D, then C can be developed without knowing about D

Java Definitions

- When class X extends Y
  - X is a subclass
  - Y is a superclass
- When interface A extends Interface B
  - A is a sub-interface
  - B is a super-interface
- When G implements interface B
  - G is an implementation of B
  - B is an interface of class G

Independent operations

- Minimize dependency:
  - reduce having to relay on anything set in stone
- Example: Replace

void print() // prints to System.out

- with

String getText() // can print anywhere

- Removes dependence on System, PrintStream

Aggregation

- Object of a class contains objects of another class
- Example: MessageQueue aggregates Messages
- Example: Mailbox aggregates MessageQueue
- Implemented through instance fields

Relationships

- 1 : 1 or 1 : 0...1 relationship:

```
public class Mailbox
{
. . .
private Greeting myGreeting;
}
```

- 1 : n relationship:

```
public class MessageQueue
{
. . .
private ArrayList<Message> elements;
}
```

Inheritance

- More general class = superclass
- More specialized class = subclass
- Subclass supports all method interfaces of superclass (but implementations may differ)
- Subclass may have added methods, added state
- Subclass inherits from superclass
- Example:
  - ForwardedMessage inherits from Message
  - Greeting does not inherit from Message (Can't store greetings in mailbox)

Use Cases

- Analysis technique
- Each use case focuses on a specific scenario
- Use case = sequence of actions
- Action = interaction between actor and computer system
- Each action yields a result
- Each result has a value to one of the actors
- Use variations for exceptional situations

Use case: Leave a Message

1. Caller dials main number of voice mail system
2. System speaks prompt
   - Enter mailbox number followed by #
3. User types extension number
4. System speaks
   - You have reached mailbox xxxx. Please leave a message now
5. Caller speaks message
6. Caller hangs up
7. System places message in mailbox

Variations

- user enters invalid extension number
  - What do you do?
  - Who does it?
- What if user hangs up instead of using message?
- How many attempts at password?

CRC Cards

- CRC = Classes, Responsibilities, Collaborators
- Use an index card for each class
- Class name on top of card
- Responsibilities on left
- Collaborators on right

CRC

- Responsibilities should be high level
- 1 - 3 responsibilities per card
- Collaborators are for the class, not for each responsibility

Example

- Use case: "Leave a message"
- Caller connects to voice mail system
- Caller dials extension number
- "Someone" must locate mailbox
- Neither Mailbox nor Message can do this
- New class: MailSystem
- Responsibility: manage mailboxes

UML

- UML = Unified Modeling Language
- Many diagram types
- We'll use three types:
  - Class Diagrams
  - Sequence Diagrams
  - State Diagrams

UML

- Why do we model?
  - – Provide structure for problem solving
  - – Experiment to explore multiple solutions
  - – Furnish abstractions to manage complexity
  - – Decrease development costs
  - – Manage the risk of mistakes
- Graphical Approach
  - – Picture is worth 1000 words

UML Building Blocks

- model elements (classes, interfaces, components, use cases, etc.)
- relationships (associations, generalization, dependencies, etc.)
- diagrams (class diagrams, use case diagrams,
- interaction diagrams, etc.)
- Simple building blocks are used to create large, complex
- structures
- – elements, bonds and molecules in chemistry
- – components, connectors and circuit boards in hardware

Definition: UML

- It is a language
  - – Syntax & Semantics
- When we model a concept there rules on how things can be put together and what it means when they are organized in a specific way

Applications

- Software design
- System requirements
- Documenting system process

View

- UML provides a view
  - – Many views of system
  - – Don’t stuff everything into one huge diagram
  - – Specific diagram types can express specific concept
  - – Sometimes multiple diagrams can apply
- Pick best which you think can express the idea
- We will be covering a simple UML

Modeling

- What is modeling?
- Means to capture idea, relationship, decision, and requirements in a well defined notation.

Diagrams

- Visual representation of concepts and relationships
- Structural Diagrams
- Behavior Diagrams

UML

Based on

Practical UML™: A Hands-On Introduction for Developers - by Randy Miller

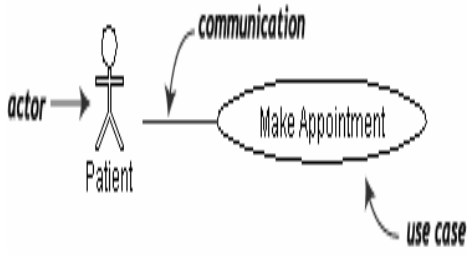
Use case diagrams

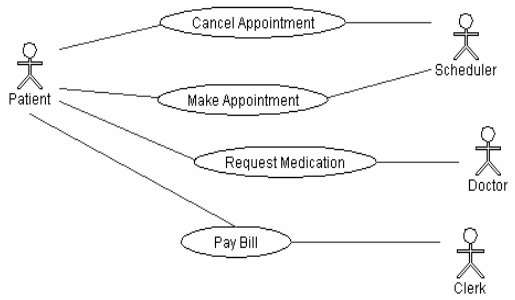
- Describe what a system does from the standpoint of an external observer.
- Emphasis - *what* a system does not *how.*
- A **scenario** is an example of what happens when someone interacts with the system.
- Scenario: a medical clinic.
  - "A patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot. "

…

- A **use case** is a summary of scenarios for a single task or goal.
- An **actor** initiates the events in that task.
- Actors = roles
- A use case diagram is a collection of actors, use cases, and their communications.
- **Make Appointment** as part of a diagram with four actors and four use cases.

Make an Appointment



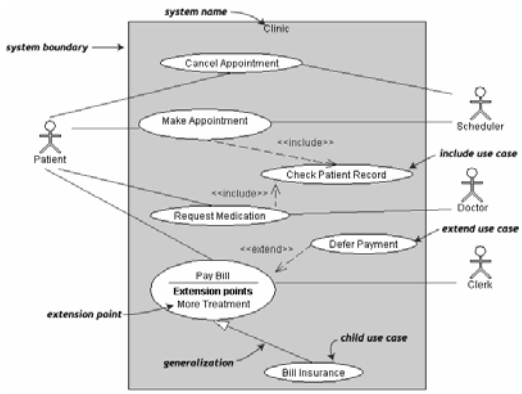


Usage

- Use case diagrams are helpful in three areas.
  - **determining features (requirements)**.
  - **communicating with clients**.
  - **generating test cases**.

Expansion

- A simple use case diagram can be expanded with additional features to display more information.
- Use case features.
  - system boundaries
  - generalizations
  - includes
  - extensions



Features

- A **system boundary** rectangle separates the clinic system from the external actors.
- A **generalization** shows that one use case is simply a special kind of another.
- **Pay Bill** - parent; **Bill Insurance** - child
- Generalization - a line with a triangular arrow head toward the parent use case

Include

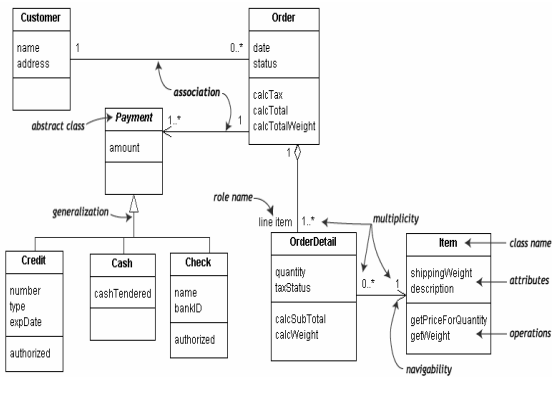
- **Include** relationships factor use cases into additional ones.
- Helpful when the same use case can be factored out of two different use cases.
- Both **Make Appointment** and **Request Medication** include **Check Patient Record** as a subtask.
- Include - a dotted line beginning at base use case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

Extend

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

Class diagrams

- Overview of a system showing classes and the relationships among them.
- Class diagrams are static -- they display what interacts but not what happens when they do interact.
- Next: a customer order from a retail catalog.
- The central class is the **Order**.
- Associated with it are the **Customer** making the purchase and the **Payment**.
- A **Payment** is one of three kinds: **Cash**, **Check**, or **Credit**.
- The order contains **OrderDetails** (line items), each with its associated **Item**.



Class Notation

- Class notation - rectangle divided into three parts:
  - class name,
  - attributes, and
  - operations.
- Names of abstract classes, such as ***Payment,*** are in italics.
- Relationships between classes are the connecting links.
- **association** -- a relationship between instances of the two classes.
  - There is an association between two classes if an instance of one class must know about the other in order to perform its work.
- **aggregation** -- an association in which one class belongs to a collection.
  - An aggregation has a diamond end pointing to the part containing the whole. In our diagram, **Order** has a collection of **OrderDetails**.
- **generalization** -- an inheritance link indicating one class is a superclass of the other.
  - A generalization has a triangle pointing to the superclass. ***Payment*** is a superclass of **Cash**, **Check**, and **Credit**.

Association

- An **association** has two ends.
  - An end may have a **role name** to clarify the nature of the association.
  - For example, an **OrderDetail** is a line item of each **Order**.
- A **navigability** arrow on an association shows which direction the association can be traversed or queried.
- An **OrderDetail** can be queried about its **Item**, but not the other way around. The arrow also lets you know who "owns" the association's implementation; in this case, **OrderDetail** has an **Item**.
- Associations with no navigability arrows are bi-directional.
- The **multiplicity** of an association end is the number of possible instances of the class associated with a single instance of the other end.
- Multiplicities are single numbers or ranges of numbers.
- In our example, there can be only one **Customer** for each **Order**, but a **Customer** can have any number of **Orders**.

Multiplicities

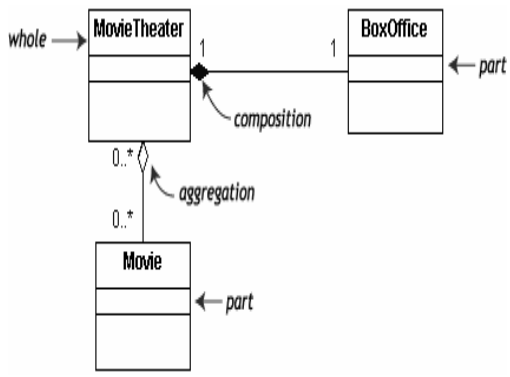
Multiplicities	**Meaning**
0..1	zero or one instance. The notation ***n . . m*** indicates ***n*** to ***m*** instances
0..* *or* ****	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance

Additional Features

- All class diagrams have classes, links, and multiplicities.
- Additional items:
  - compositions
  - class member visibility and scope
  - dependencies and constraints
  - interfaces

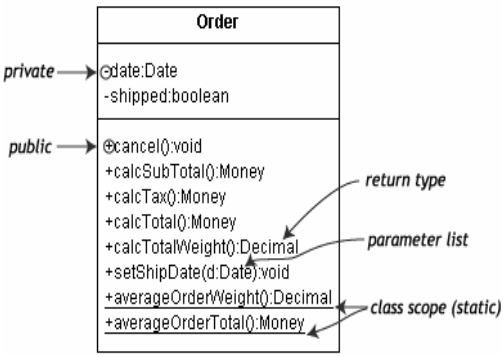
Composition and aggregation

- Associations in which an object is part of a whole are aggregations.
- **Composition** is a strong association in which the part can belong to only one whole -- the part cannot exist without the whole.
  - Composition is denoted by a filled diamond at the whole end



Class information: visibility and scope

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

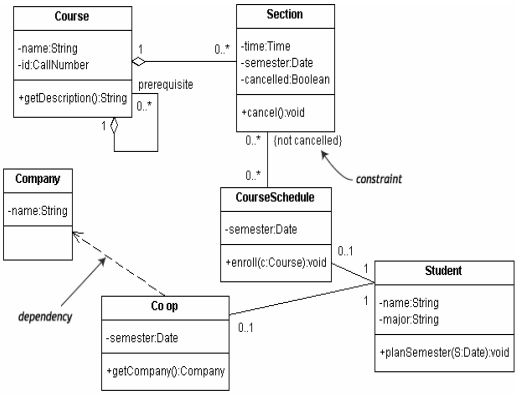


Conventions

- The illustration uses the following conventions.
  - Static members are underlined. Instance members are not.
  - The operations follow this form: <*access specifier*> <*name*> (<*parameter list*>) : <*return type*>
  - The parameter list shows each parameter type preceded by a colon.
  - Access specifiers appear in front of each member.
    - + public
    - - private
    - # protected

Dependencies and constraints

- A **dependency** is a relation between two classes in which a change in one may force changes in the other.
  - Dependencies are drawn as dotted lines. In the class diagram below, **Co_op** depends on **Company**. If you decide to modify **Company**, you may have to change **Co_op** too.
- A **constraint** is a condition that every implementation of the design must satisfy. Constraints are written in curly braces { }.
  - The constraint on our diagram indicates that a **Section** can be part of a **CourseSchedule** only if it is not canceled.



Relationships

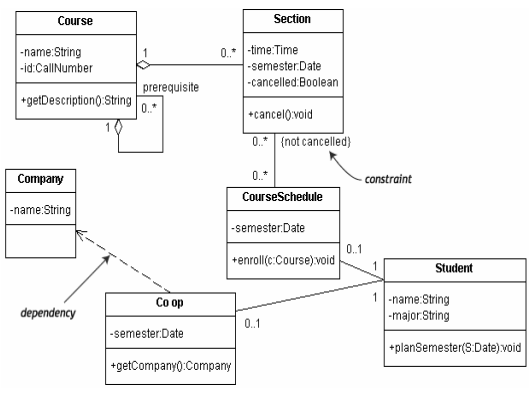
- Class diagrams have three kinds of relationships.
  - **association** -- a relationship between instances of the two classes. There is an association between two classes if an instance of one class must know about the other in order to perform its work.
  - **aggregation** -- an association in which one class belongs to a collection. An aggregation has a diamond end pointing to the part containing the whole. In our diagram, **Order** has a collection of **OrderDetails**.
  - **generalization** -- an inheritance link indicating one class is a superclass of the other. A generalization has a triangle pointing to the superclass. ***Payment*** is a superclass of **Cash**, **Check**, and **Credit**.

…

- An association has two ends. An end may have a **role name** to clarify the nature of the association. For example, an **OrderDetail** is a line item of each **Order**.
- A **navigability** arrow on an association shows which direction the association can be traversed or queried. An **OrderDetail** can be queried about its **Item**, but not the other way around. The arrow also lets you know who "owns" the association's implementation; in this case, **OrderDetail** has an **Item**. Associations with no navigability arrows are bi-directional.
- The **multiplicity** of an association end is the number of possible instances of the class associated with a single instance of the other end. Multiplicities are single numbers or ranges of numbers. In our example, there can be only one **Customer** for each **Order**, but a **Customer** can have any number of **Orders**.

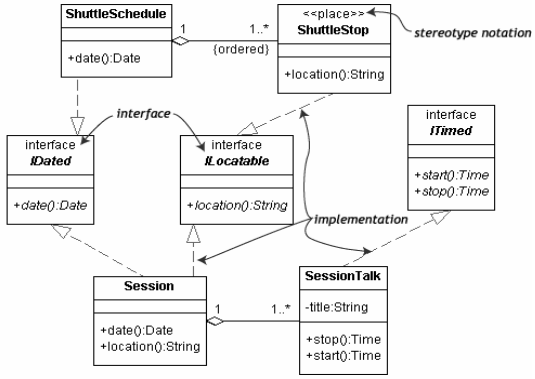
Dependencies and constraints

- D**ependency** - relation between two classes in which a change in one may force changes in the other.
  - Dependencies are drawn as dotted lines. In the class next diagram, **Co_op** depends on **Company**.
- A **constraint** is a condition that every implementation of the design must satisfy.
- Constraints are written in curly braces { }.
  - **CourseSchedule** only if it is not canceled.



Interfaces and stereotypes

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

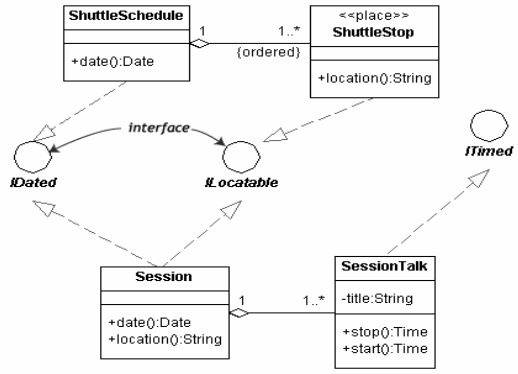


Interfaces

- There are three interfaces in the diagram: ***IDated***, ***ILocatable***, and ***ITimed***.
- Interfaces typically begin with the letter **I** and written in italics.
- A class such as **ShuttleStop**, with operations matching those in an interface, such as **ILocatable**, is an **implementation** (or **realization**) of the interface.

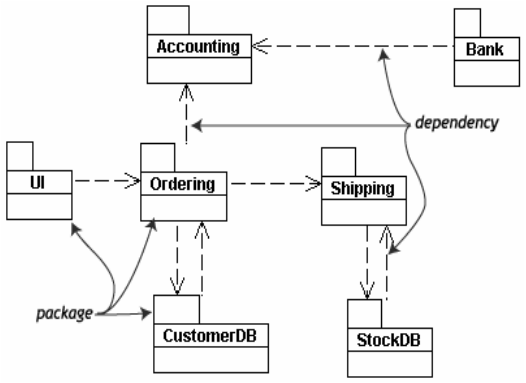
Stereotype

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



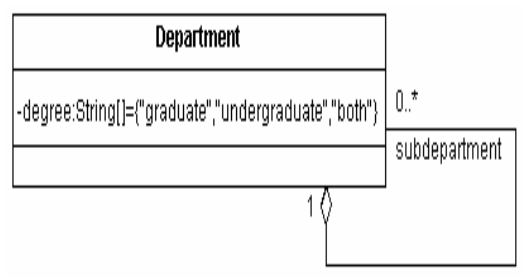
Packages and object diagrams

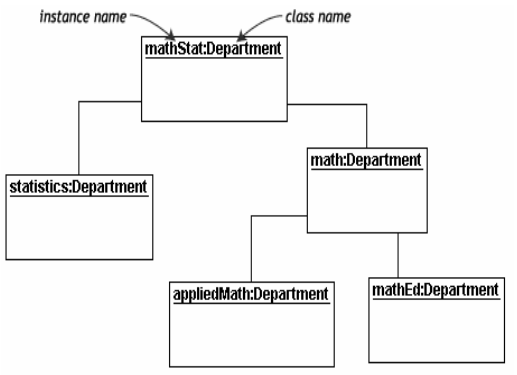
- To simplify complex class diagrams, you can group classes into **packages**.
- A package is a collection of logically related UML elements.
- The next diagram is a business model in which the classes are grouped into packages.
- Packages appear as rectangles with small tabs at the top. .



Object diagrams

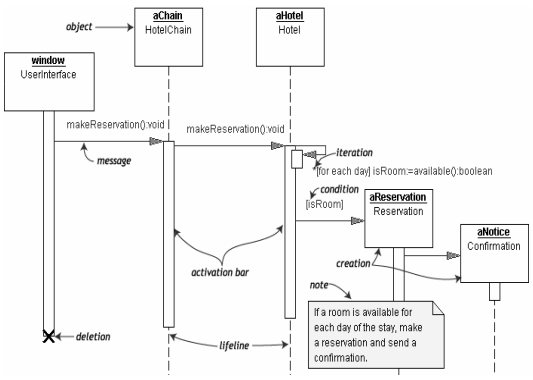
- Show instances instead of classes.
- They are useful for explaining small pieces with complicated relationships, especially recursive relationships.
- This small class diagram shows that a university **Department** can contain lots of other **Departments**





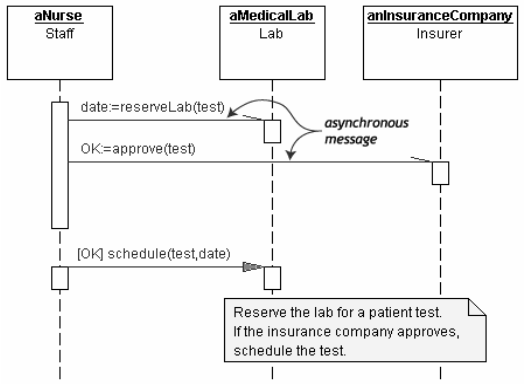
Sequence Diagrams

- A **sequence diagram** is an interaction diagram that details how operations are carried out -- what messages are sent and when.
- Sequence diagrams are progress with time down the page.
- The objects involved in the operation are listed from left to right according to when they take part in the message sequence.



Sequence diagrams with asynchronous messages

- A message is **asynchronous** the sender is not blocked until the message is delivered.
- The following sequence diagram illustrates the action of a nurse requesting a diagnostic test at a medical lab.
- There are two asynchronous messages from the **Nurse**:
  1. ask the **MedicalLab** to reserve a date for the test and
  2. ask the **InsuranceCompany** to approve the test.
- The order in which these messages are sent or completed is irrelevant.

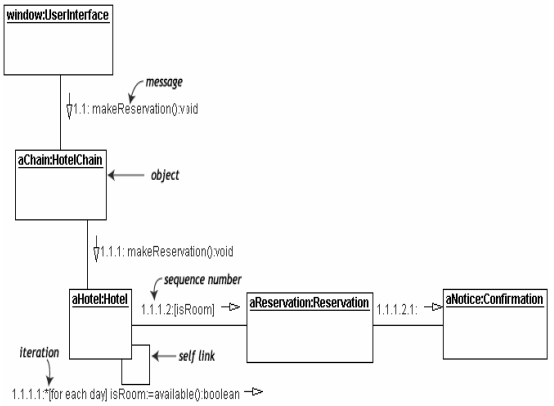


Notations

Symbol simple	**Meaning**
	message which may be synchronous or asynchronous
	simple message return (optional)
	a synchronous message
	an asynchronous message

Collaboration Diagrams

- **Collaboration diagrams** are also interaction diagrams.
- Same information as sequence diagrams, but focus on object roles instead of the times that messages are sent.
- In a sequence diagram, object roles are the vertices and messages are the connecting links.

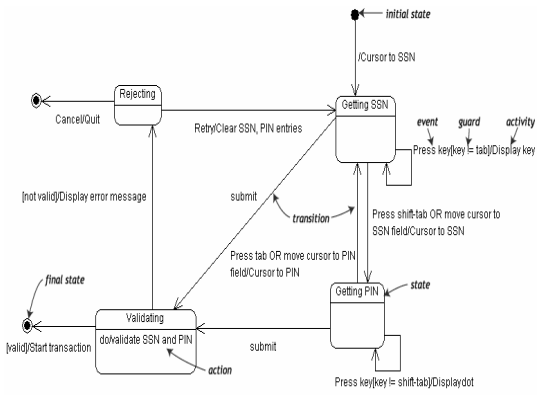


…

- The object-role rectangles are labeled with either class or object names (or both).
- Class names are preceded by colons (:).
- Each message in a collaboration diagram has a **sequence number**.
- The top-level message is numbered 1.
- Messages at the same level (sent during the same call) have the same decimal prefix but suffixes of 1, 2, etc. according to when they occur.

Statechart Diagrams

- Objects have behaviors and state.
- The state of an object depends on activity or condition.
- A **statechart diagram** - the possible states of the object and the transitions that change state.
- Example
  - The login part of an online banking system.
  - Logging - entering a valid social security number and personal id number, then submitting the information for validation.
- Logging - states: **Getting SSN**, **Getting PIN**, **Validating**, and **Rejecting**.
- Each state comes a complete set of **transitions** that determine the subsequent state.

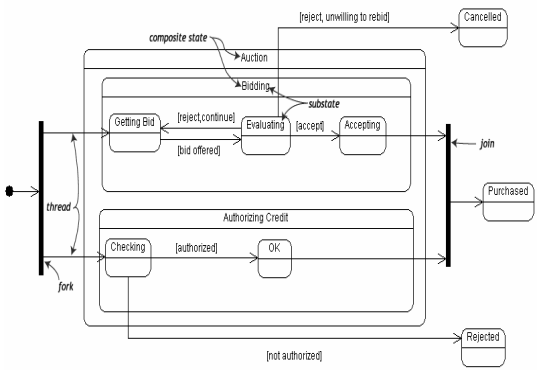


States, Transitions

- States - rounded rectangles.
- Transitions - arrows from one state to another.
- Events or conditions that trigger transitions - beside arrows.
- The initial state (black circle) and final states
- The action that occurs as a result of an event or condition is expressed as /action.
- While in **Validating** state, the object does not wait for an outside event.
  - It performs an activity.
- The result of that activity determines its subsequent state.

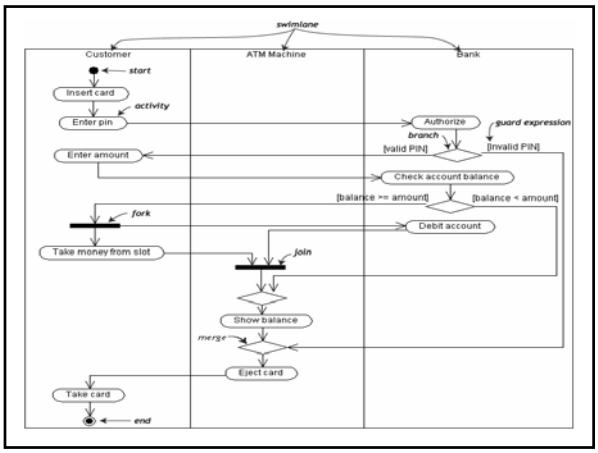
Concurrency and asynchronization in statechart diagrams

- States in statechart diagrams can be nested.
- Related states can be grouped together into a single **composite state**.
- Nesting states - an activity involves concurrent or asynchronous subactivities.
- An auction with two concurrent threads leading into two substates of the composite state **Auction**:
  - **Bidding** itself is a composite state with three substates.
  - **Authorizing Credit** has two substates.
- Entering the **Auction** requires a fork at the start into two separate threads.
- Unless there is an abnormal exit (**Cancelled** or **Rejected**), the exit from the **Auction** composite state occurs when both substates have exited.



Activity Diagrams

- A fancy flowchart.
- Activity diagrams and statechart diagrams are related.
- Statechart diagram focuses on an object undergoing a process
- Activity diagram focuses on flow of activities involved in a single process.
- Example:
  - "Withdraw money from a bank account through an ATM."
- The classes of the activity are **Customer**, **ATM**, and **Bank**.
- Process begins at the black start circle.
- The activities are rounded rectangles.

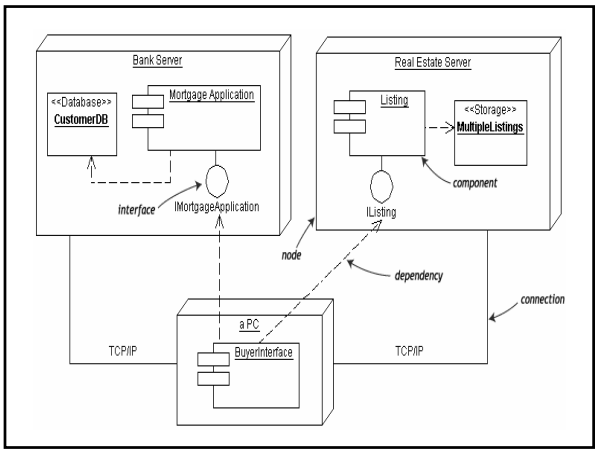


…

- Activity diagrams can be divided into object **swimlanes** that determine which object is responsible for which activity.
- A single **transition** comes out of each activity, connecting it to the next activity.
- A transition may **branch** into two or more mutually exclusive transitions.
- **Guard expressions** (inside [ ]) label the transitions coming out of a branch.
- A branch and its subsequent **merge** marking the end of the branch appear in the diagram as hollow diamonds.
- A transition may **fork** into two or more parallel activities.
- The fork and the subsequent **join** of the threads coming out of the fork appear in the diagram as solid bars.

Component and deployment diagrams

- Component diagrams are physical analogs of class diagram.
- **Deployment diagrams** show the physical configurations of software and hardware.
- The next deployment diagram shows the relationships among software and hardware components involved in real estate transactions
- The physical hardware is made up of **nodes**.
- Each component belongs on a node.
- Components are shown as rectangles with two tabs at the upper left.


Bank Server
<<Database>>
CustomerDB
Mortgage Application
interface
IMortgageApplication
Real Estate Server
Listing
<<Storage>>
MultipleListings
component
IListing
node
dependency
connection
a PC
BuyerInterface
TCP/IP
TCP/IP