Modelling with Classes

Hari Prasad Pokhrel (hpokhrel24@gmail.com)

What constitutes a good model?

- A model should
  - use a standard notation
  - be understandable by clients and users
  - lead software engineers to have insights about the system
  - provide abstraction

- Models are used:
  - to help create designs
  - to permit analysis and review of those designs.
  - as the core documentation describing the system.

Static: Class Diagram (Rumbaugh/Booch)

- Utilized for Static Structure of Conceptual Model
- Class Diagram Describes
  - Types of Objects in Application
  - Static Relationships Among Objects
  - Temporal Information Not Supported
- Class Diagrams Contain
  - Classes: Objects, Attributes, and Operations
  - Packages: Groupings of Classes
  - Subsystems: Grouping of Classes/Packages
- Main Concepts: Class, Association, Generalization, Dependency, Realization, Interface
- Granularity Level of Use-Cases is Variable

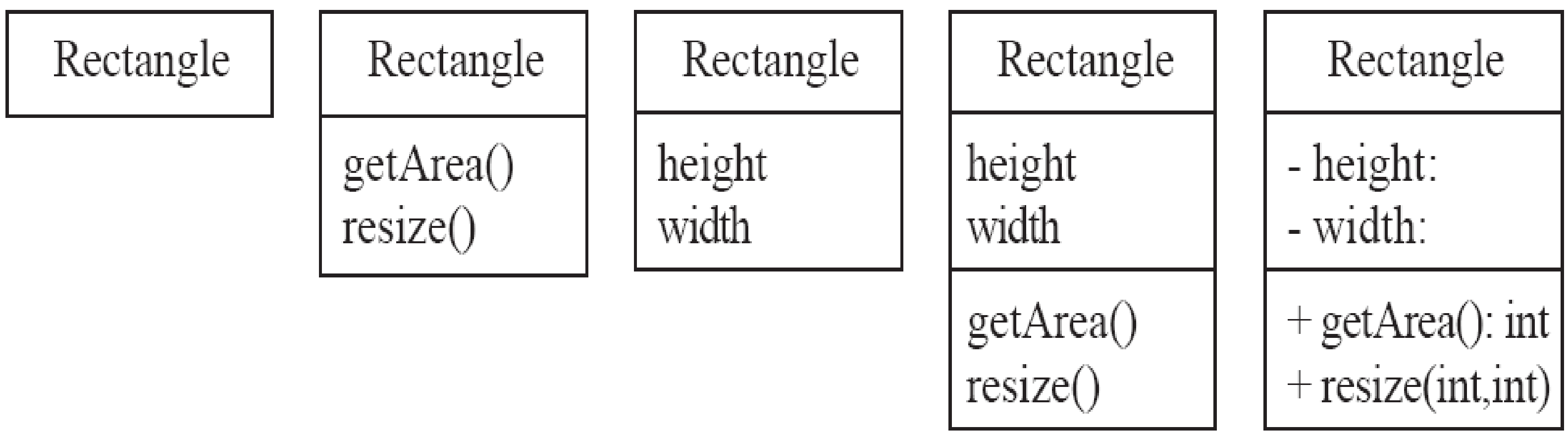
Essentials of UML Class Diagrams

- *The main symbols shown on class diagrams are:*
  - *Classes*
    - represent the types of data themselves
  - *Associations*
    - represent linkages between instances of classes
  - *Attributes*
    - are simple data found in classes and their instances
  - *Operations*
    - represent the functions performed by the classes and their instances
  - *Generalizations*
    - group classes into inheritance hierarchies

Classes

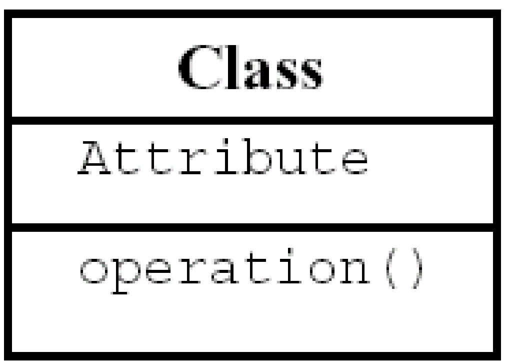
- A class is simply represented as a box with the name of the class inside
  - The diagram may also show the attributes and operations
  - The complete signature of an operation is:

    operationName(parameterName: parameterType …): returnType

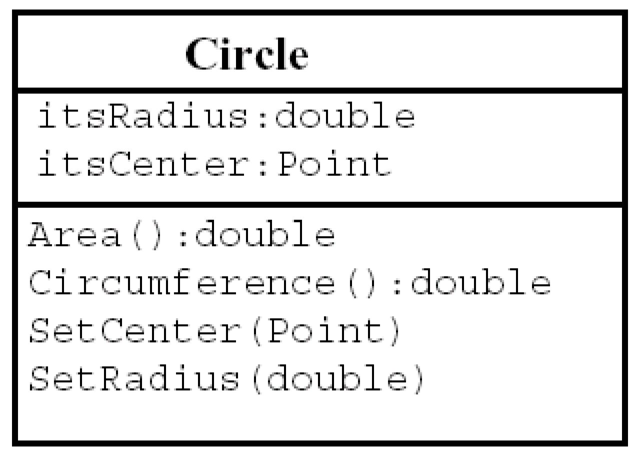


The Class Diagram Notation

- Identify classes, attributes of each class, and operations of each class
- Classes, their attributes and methods are specified based on the objects needed to realized use case and interfaces to external entities



Detailed
Attributes,
Data types,
And operations
Are defined/
refined
During design



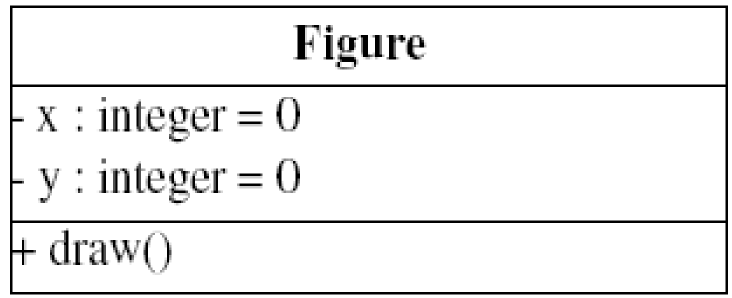
UML Class-to-Java Example

```
Public class UNIXaccount
{
  public string username;
  public string groupname = "csai";
  public int filesystem_size;
  public date creation_date;
  private string password;
  static private integer no_of_accounts = 0
  public UNIXaccount()
  {
    //Other initialisation
    no_of_accounts++;
  }
  //Methods go here
};
```

UNIXaccount
+ username : string + groupname : string = “staff" + filesystem_size : integer + creation_date : date - password : string - no_of_accounts : integer = 0

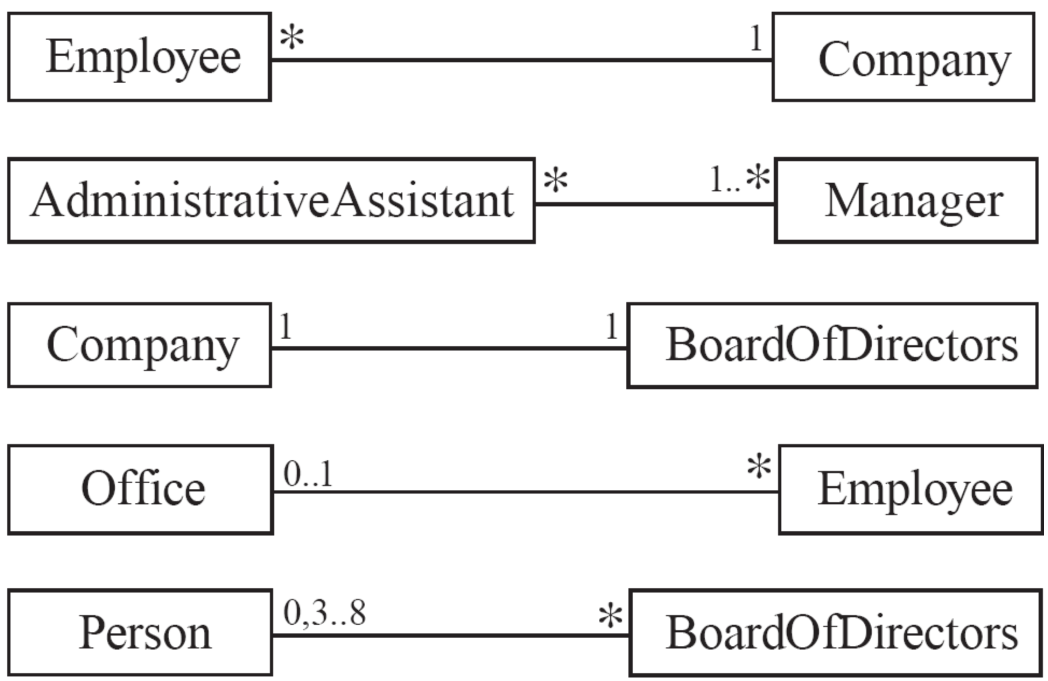
Operations (Methods)

```
Public class Figure
{
  private int x = 0;
  private int y = 0;
  public void draw()
  {
    //Java code for drawing figure
  }
};

Figure fig1 = new Figure();
Figure fig2 = new Figure();
fig1.draw();
fig2.draw();
```



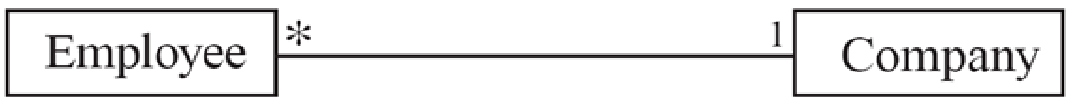
Associations and Multiplicity

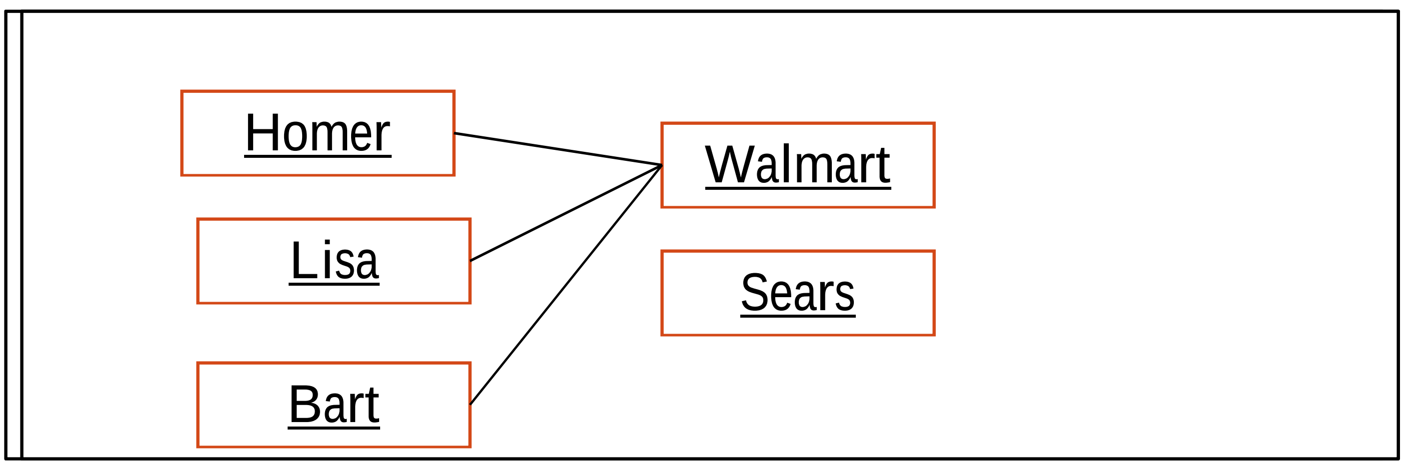
- An *association* is used to show how two classes are related to each other
  - Symbols indicating *multiplicity* are shown at each end of the association



Object Diagrams

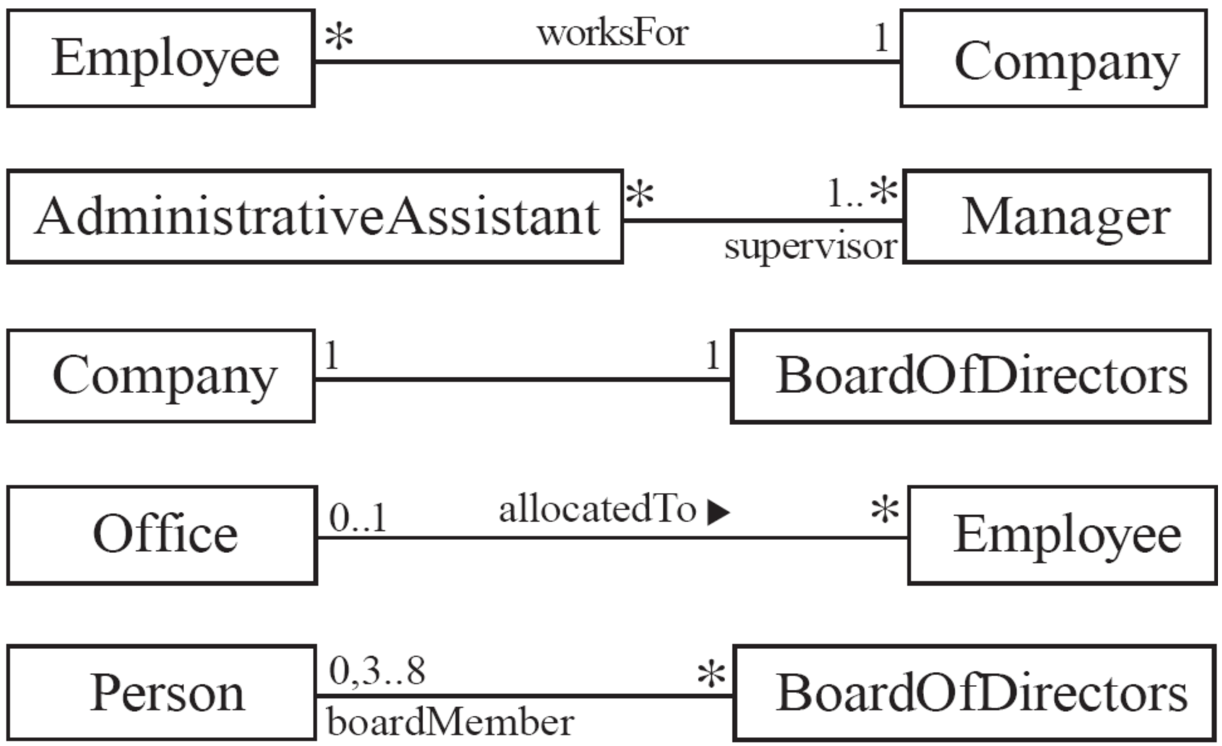
- Object (Instant) Diagrams give a representation of a class diagram using actual objects in the system. For example if this is our class diagram:
- Which of the following object diagrams are valid?



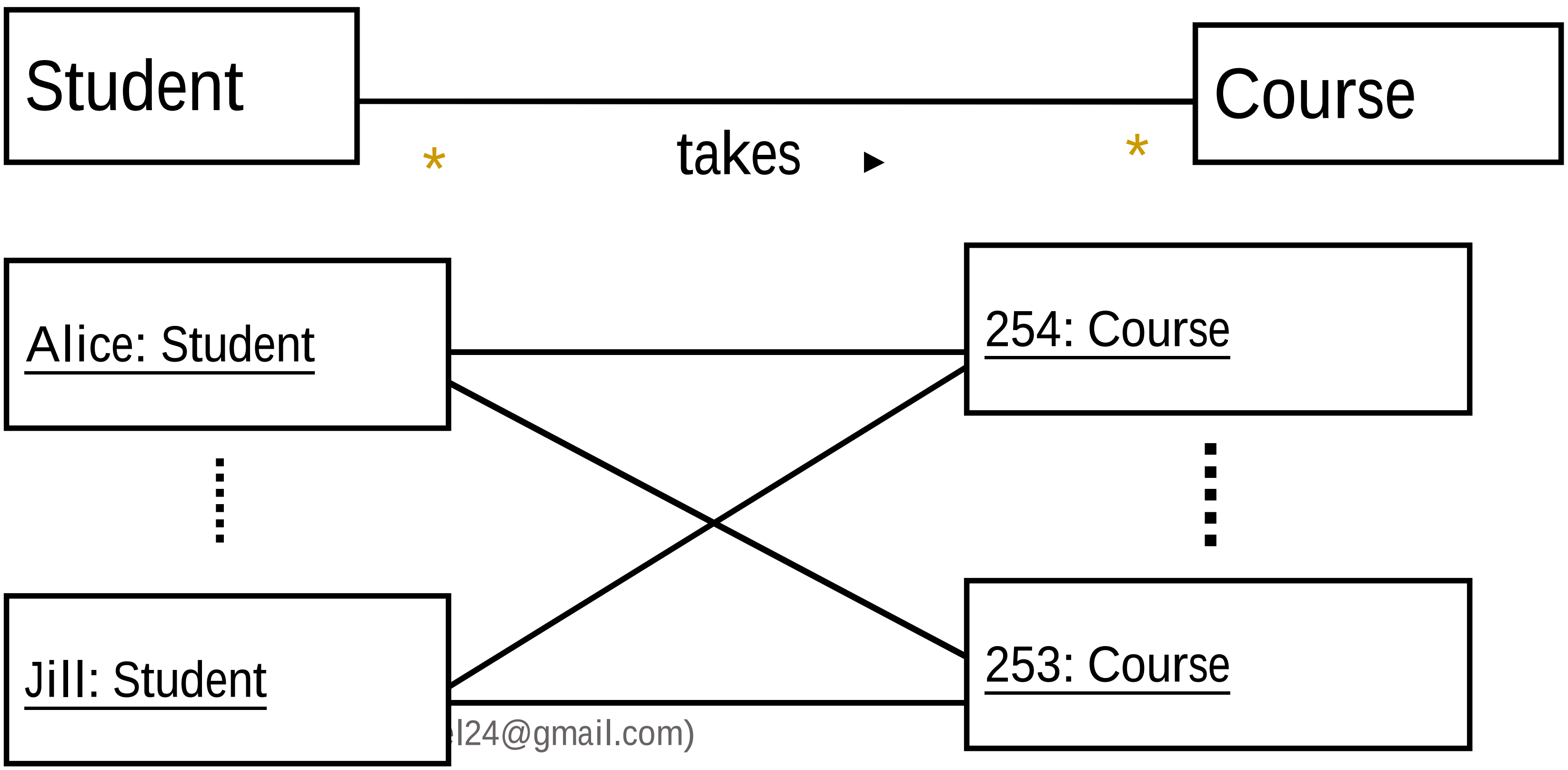


Labelling associations

- Each association can be labelled, to make explicit the nature of the association

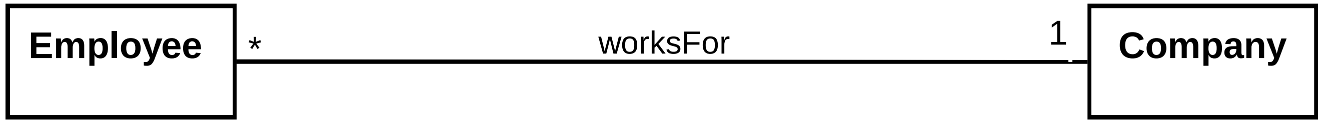


- A Student can take many Courses and many Students can be enrolled in one Course.



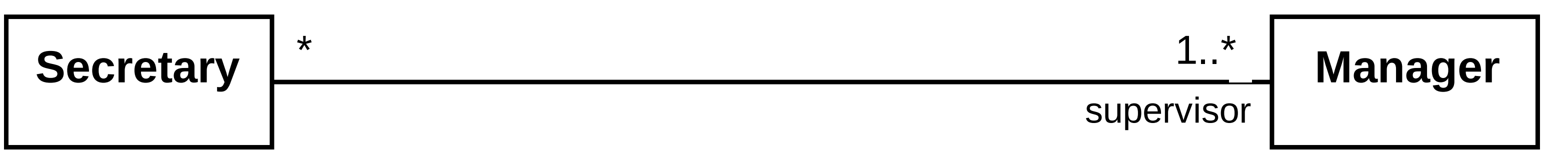
Analyzing and validating associations

- **Many-to-one**
  - A company has many employees,
  - An employee can only work for one company.
    - This company will not store data about the moonlighting activities of employees!
  - A company can have zero employees
    - E.g. a ‘shell’ company
  - It is not possible to be an employee unless you work for a company



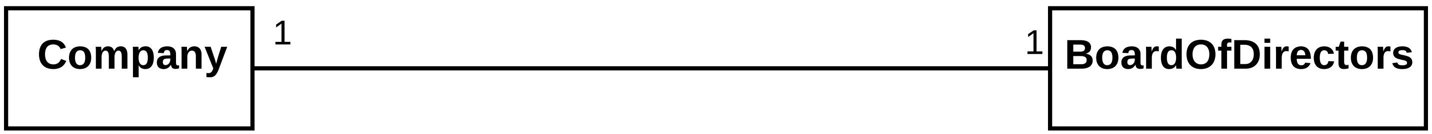
Analyzing and validating associations

- **Many-to-many**
  - A secretary can work for many managers
  - A manager can have many secretaries
  - Secretaries can work in pools
  - Managers can have a group of secretaries
  - Some managers might have zero secretaries.
  - Is it possible for a secretary to have, perhaps temporarily, zero managers?



Analyzing and validating associations

- **One-to-one**
  - For each company, there is exactly one board of directors
  - A board is the board of only one company
  - A company must always have a board
  - A board must always be of some company

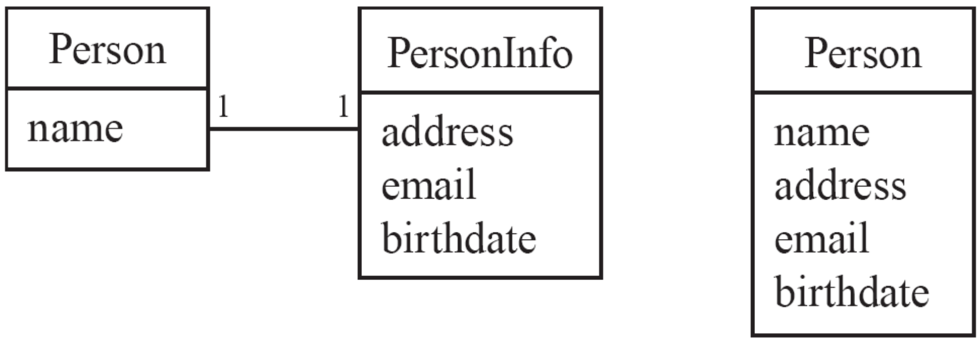


Multiplicity

- Multiplicity can be expressed as,
  - Exactly one - 1
  - Zero or one - 0..1
  - Many - 0..* or *
  - One or more - 1..*
  - Exact Number - e.g. 3..4 or 6
  - Or a complex relationship – e.g. 0..1, 3..4, 6..* would mean any number of objects other than 2 or 5

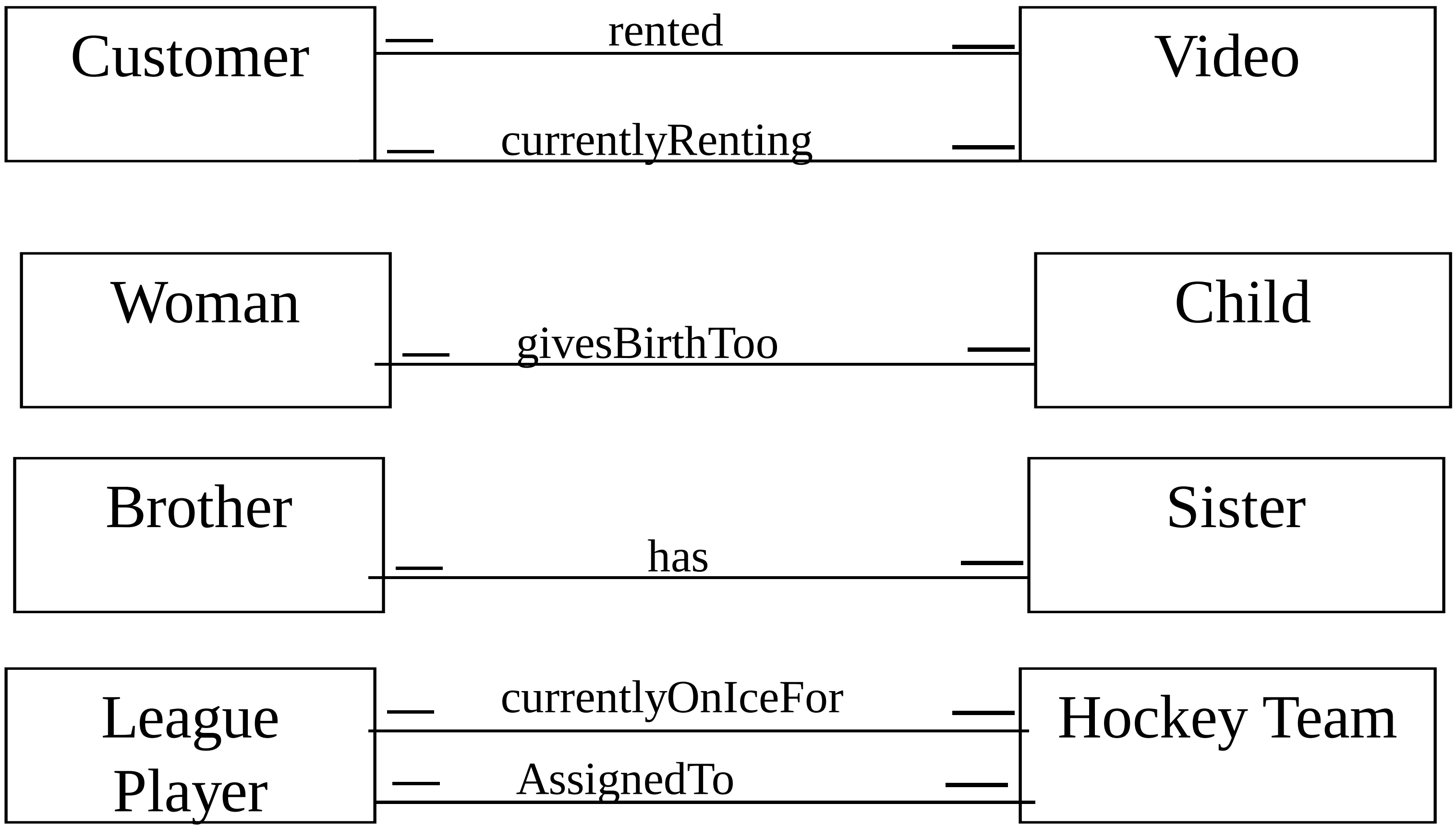
Analyzing and validating associations

- Avoid unnecessary one-to-one associations
- Avoid this do this



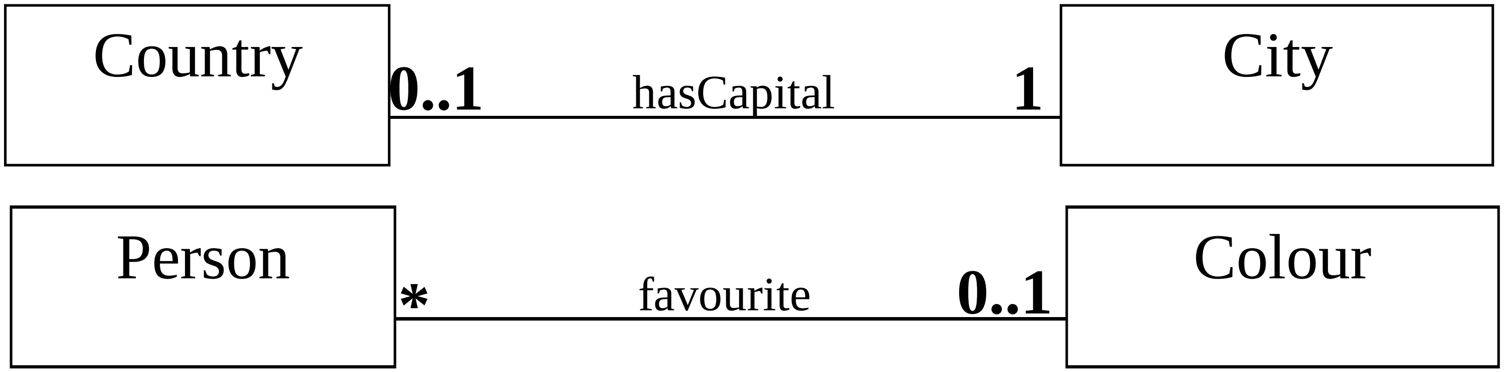
Question

- Label the multiplicities for the following examples:



Question

- In words, what do these diagrams mean?

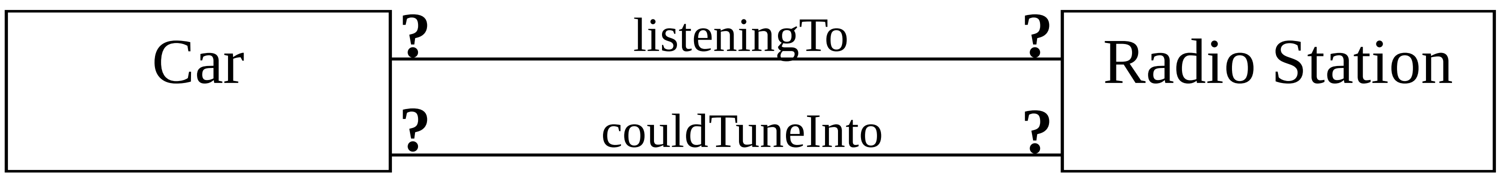


- A Country has ***one and only one city as its capital***
- A City ______________________

- A Colour __________________________
- A Person ___________________________

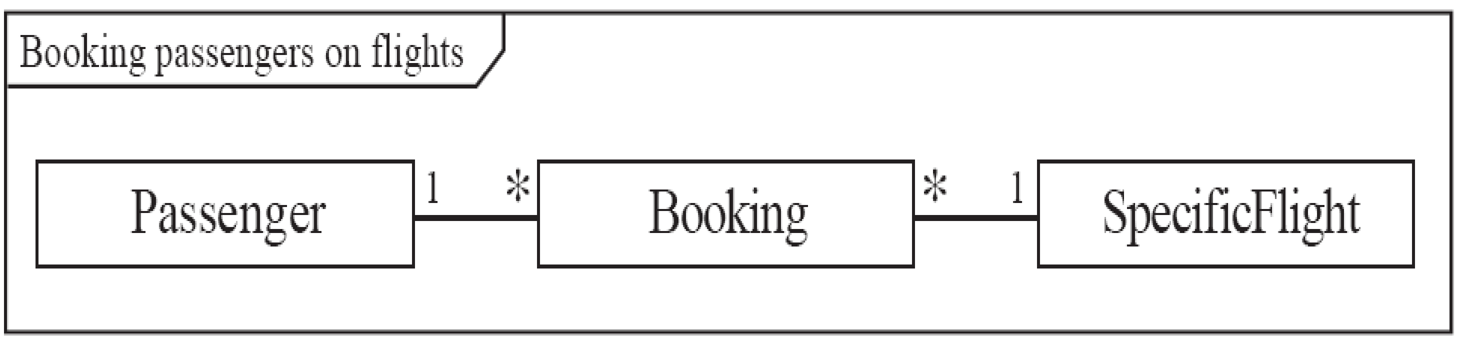
Another Question:

- Correctly label this diagrams multiplicity:



A more complex example

- A booking is always for exactly one passenger
  - no booking with zero passengers
  - a booking could *never* involve more than one passenger.
- A Passenger can have any number of Bookings
  - a passenger could have no bookings at all
  - a passenger could have more than one booking



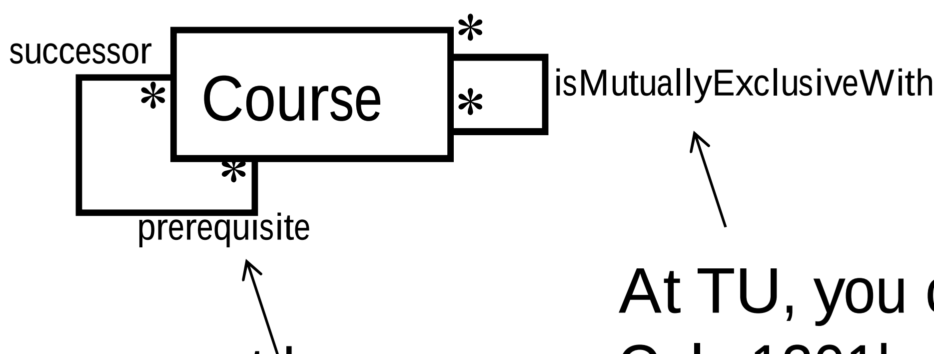
Question

- Create two or three classes linked by associations to represent the following situations:
- *A landlord renting apartments to tenant*
- *An author writing books distributed by publishers*
- Label the multiplicities (justify why you picked them)
- Give each class you choose at least 1 attribute

Your Answer

Reflexive associations/ self association.

- It is possible for an association to connect a class to itself
- An association that connects a class to itself is called a self association.

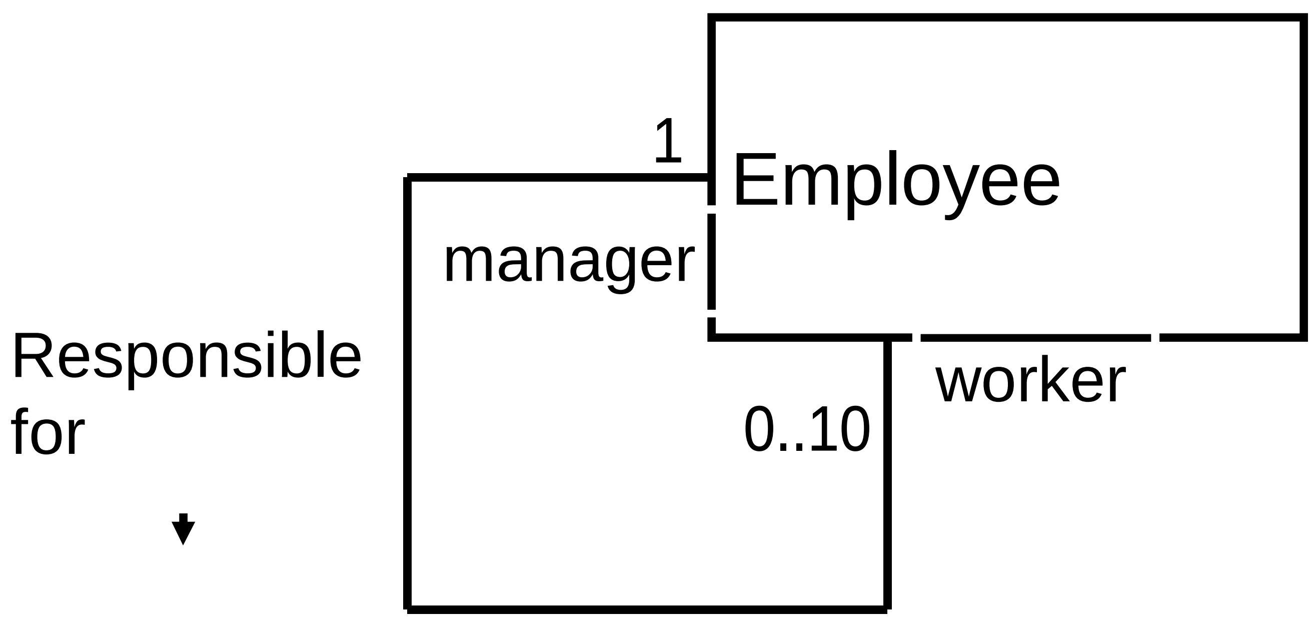


At TU, you must have CS2210 and CS2211 to take CS2212

At TU, you can’t take Calc 1301b and Calc1501b

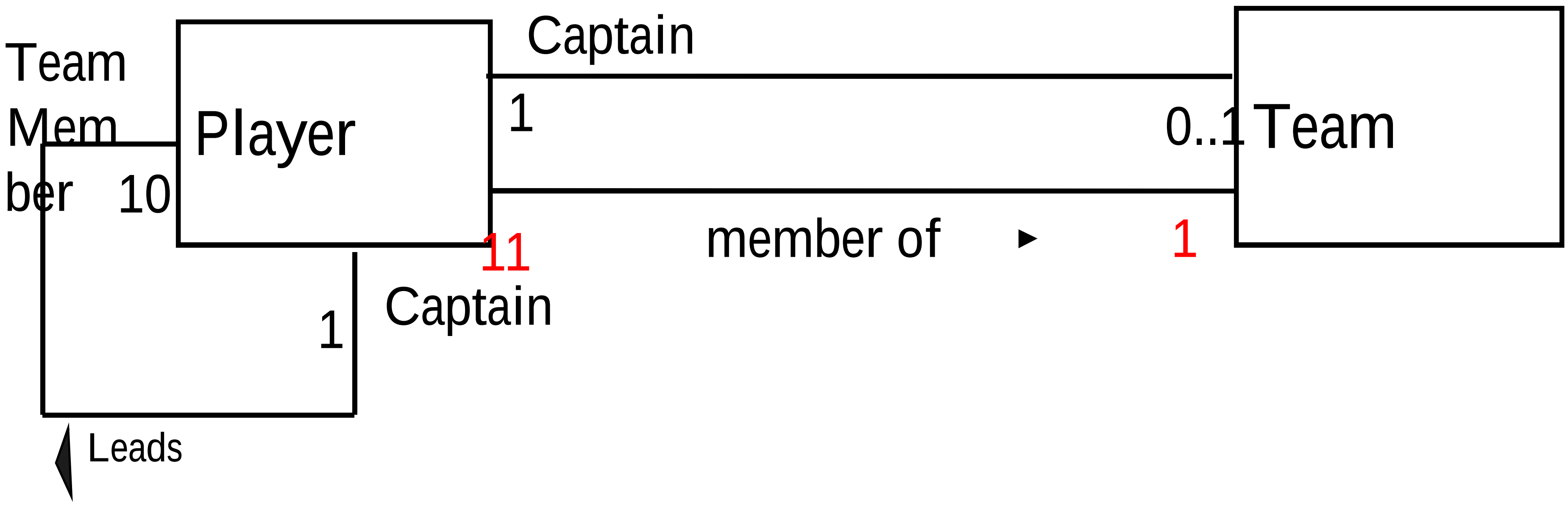
Association - Self

- A Company has Employees.
- A single manager is responsible for up to 10 workers.



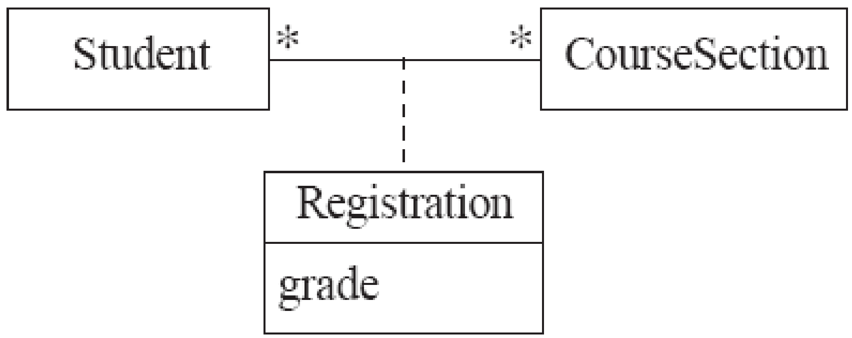
Association - Multiplicity

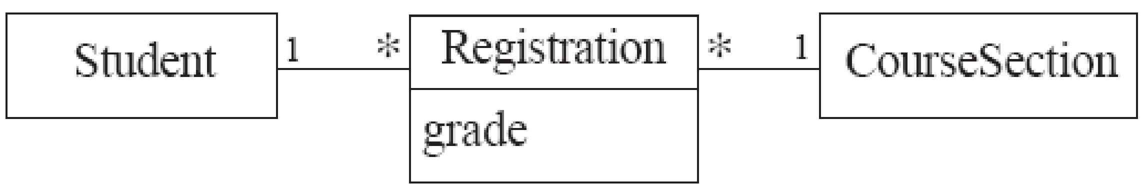
- A cricket team has 11 players. One of them is the captain.
- A player can play only for one Team.
- The captain leads the team members.



Association classes

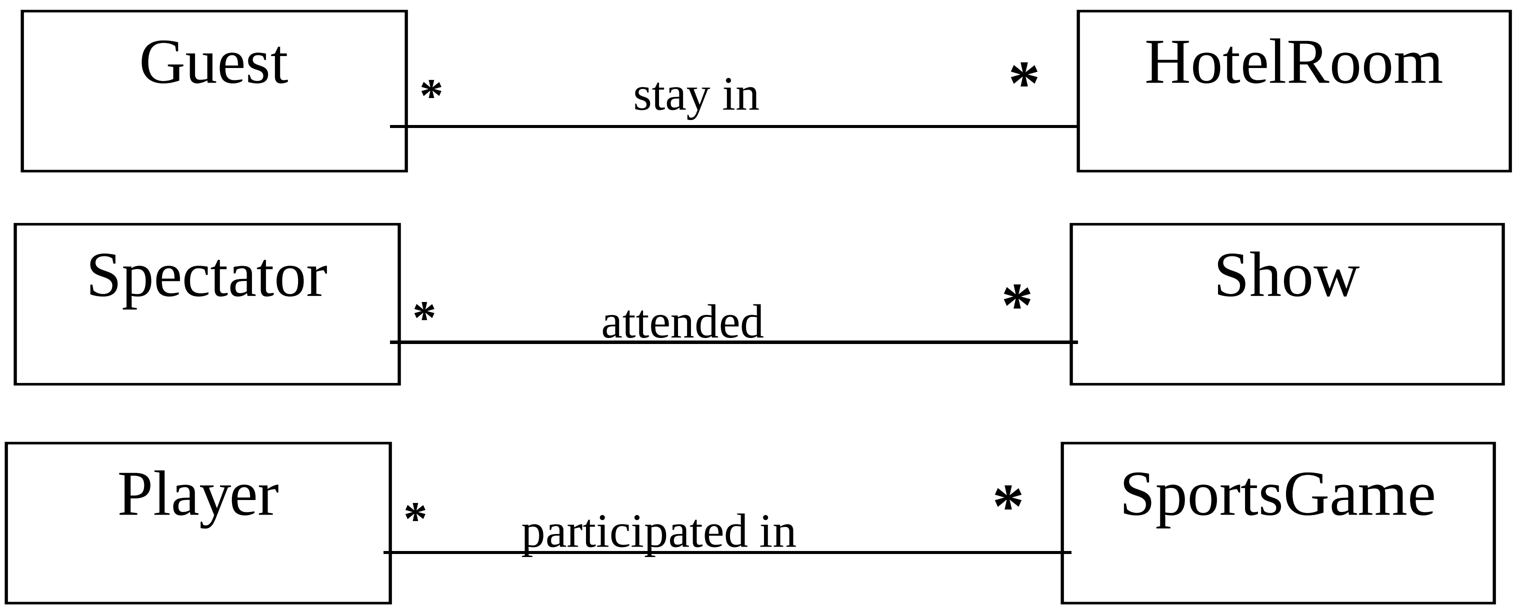
- Sometimes, an attribute that concerns two associated classes cannot be placed in either of the classes
- The following are equivalent



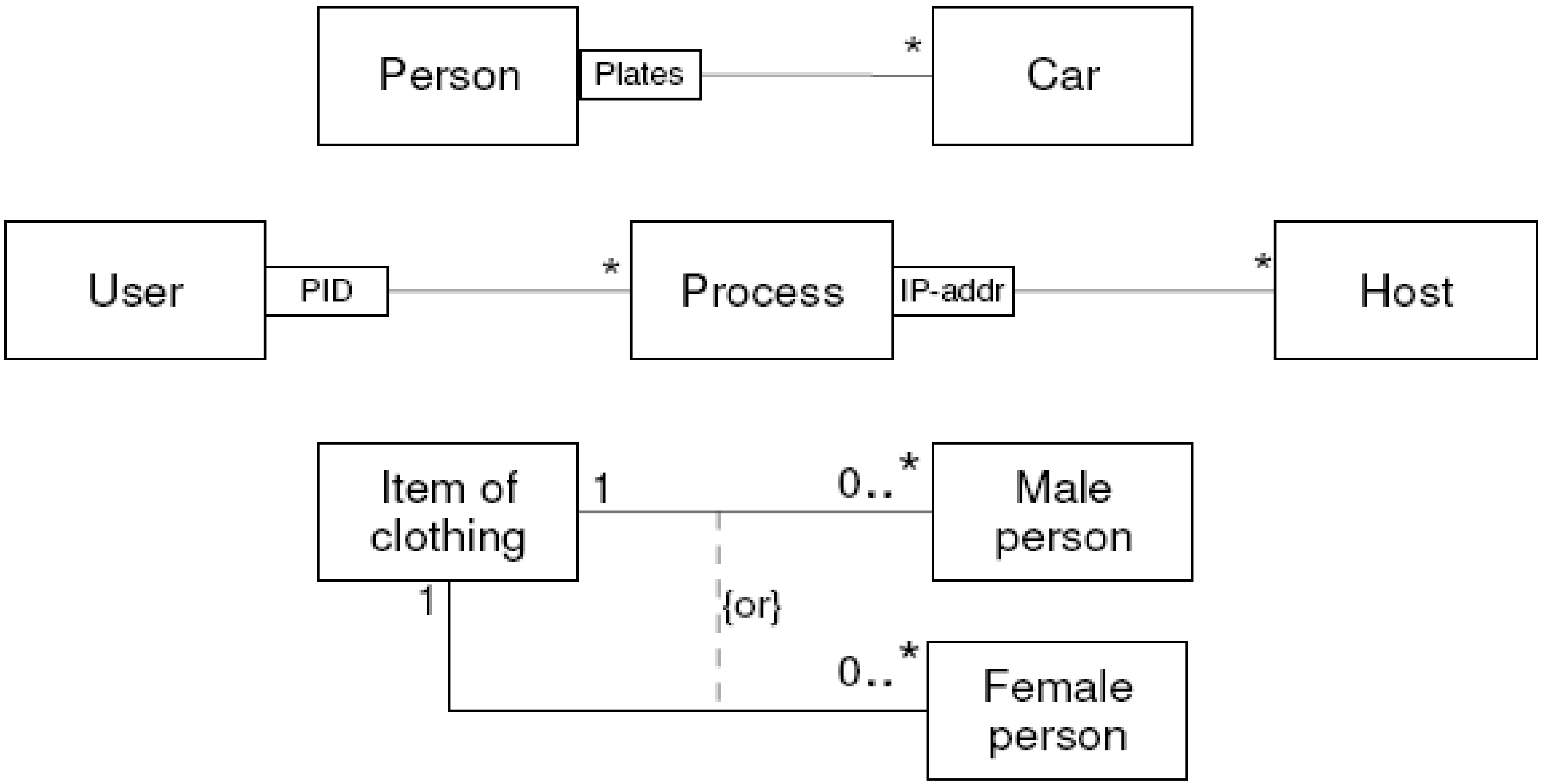


Question

- Add association classes to the following many to many associations and come up with at least one attribute for the new association class:



Qualified and "Or" Associations

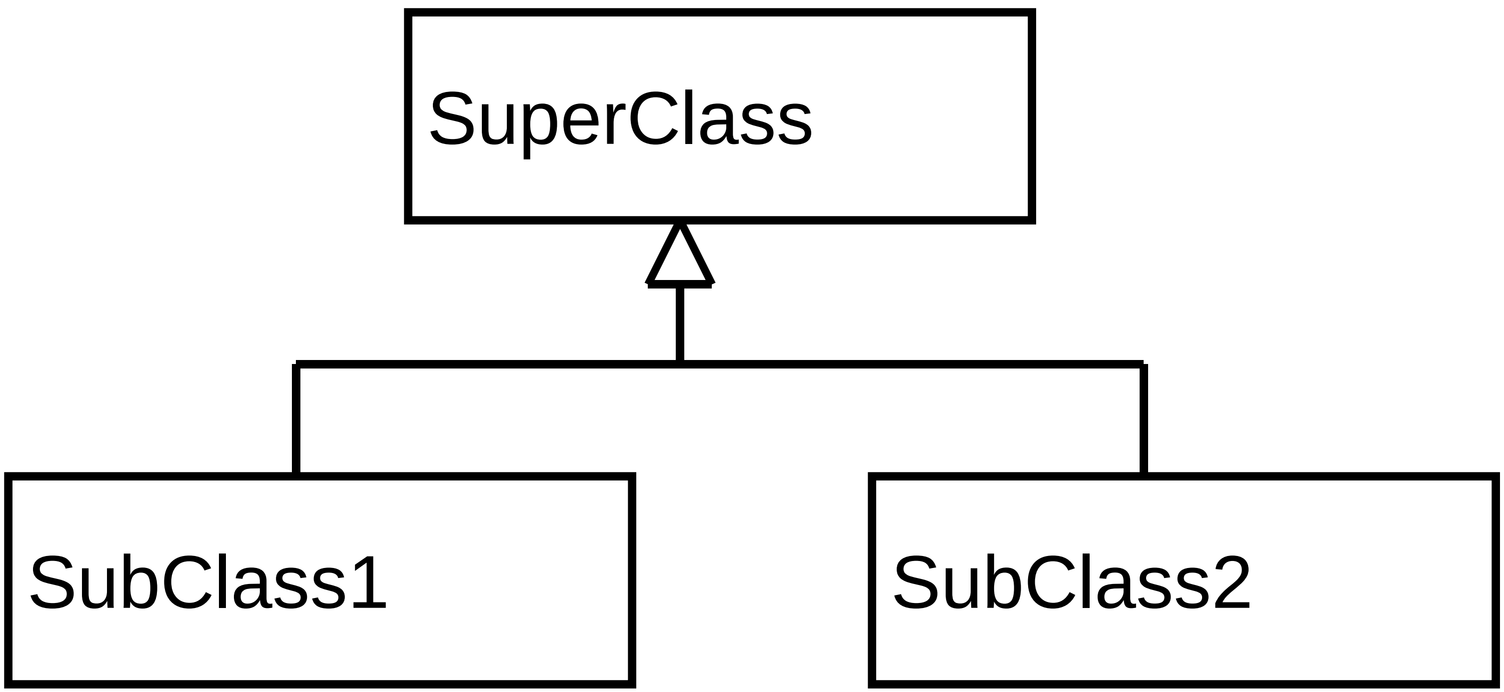


Class Relationships

- Association
- Aggregation
- Composition
- *Generalization*
- Realization
- Dependency

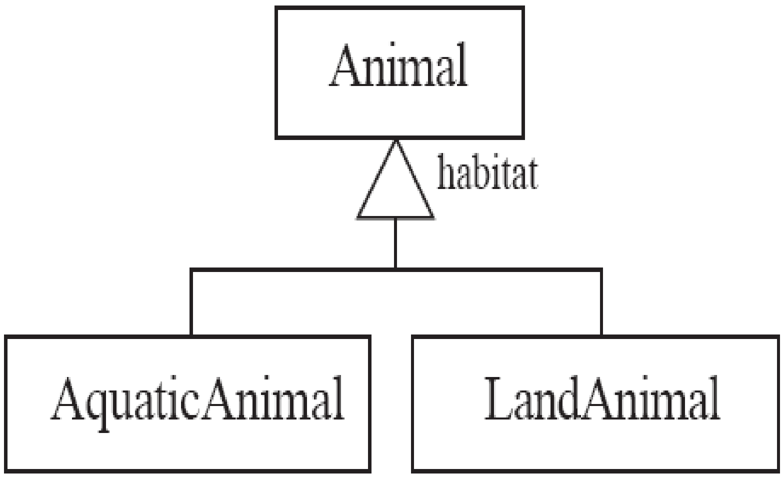
Generalization (Inheritance)

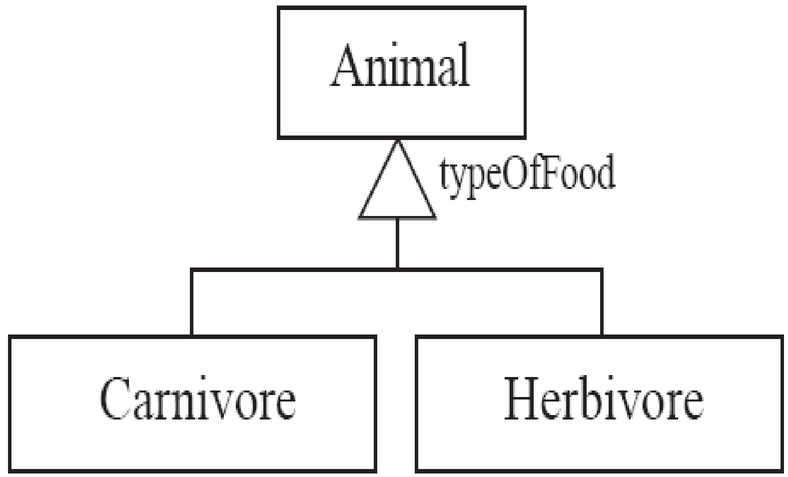
- Child class is a special case of the parent class



Generalization/Specialization Relation

- Specializing a superclass into two or more subclasses
  - The *discriminator* is a label that describes the criteria used in the specialization

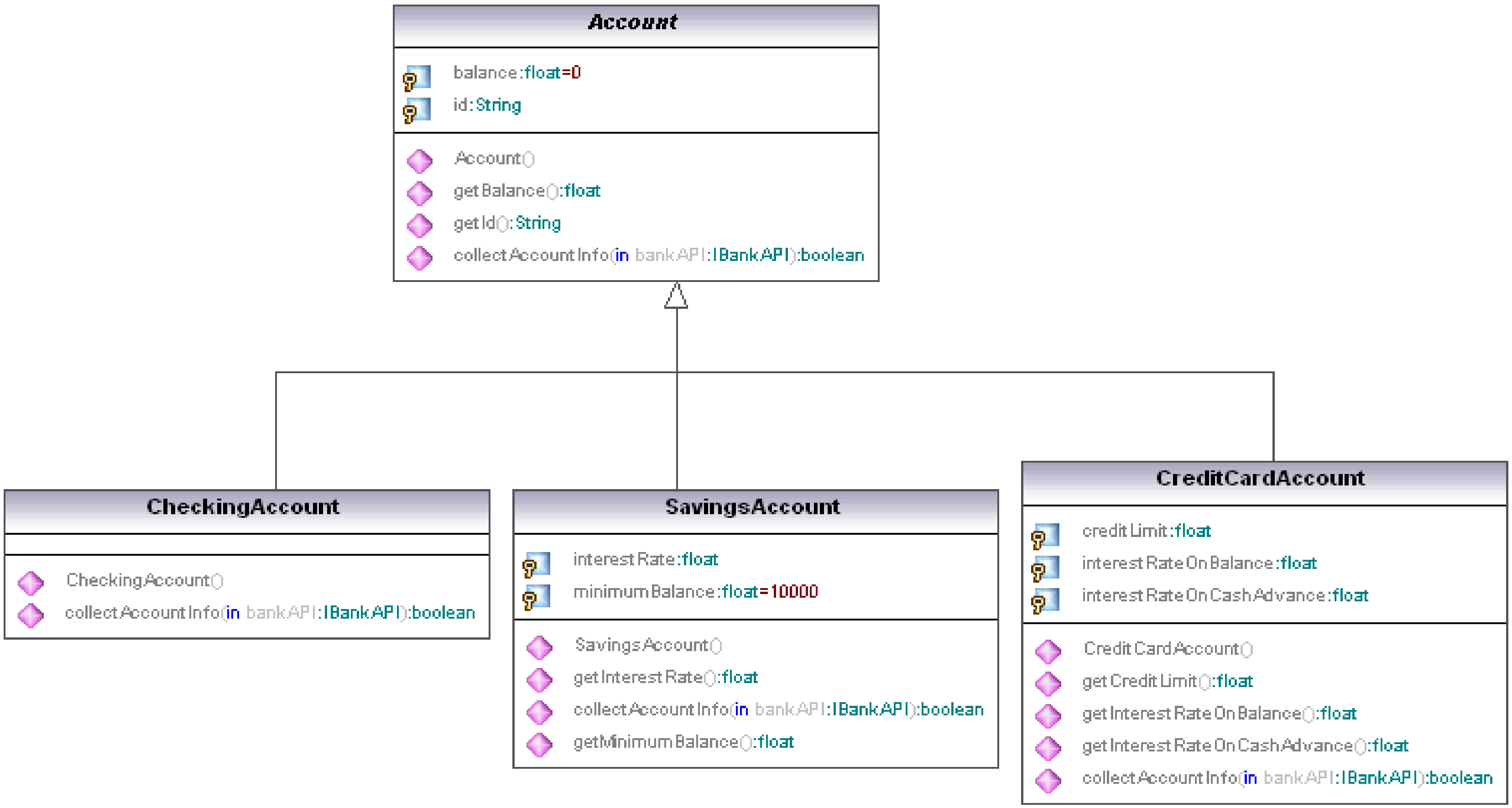




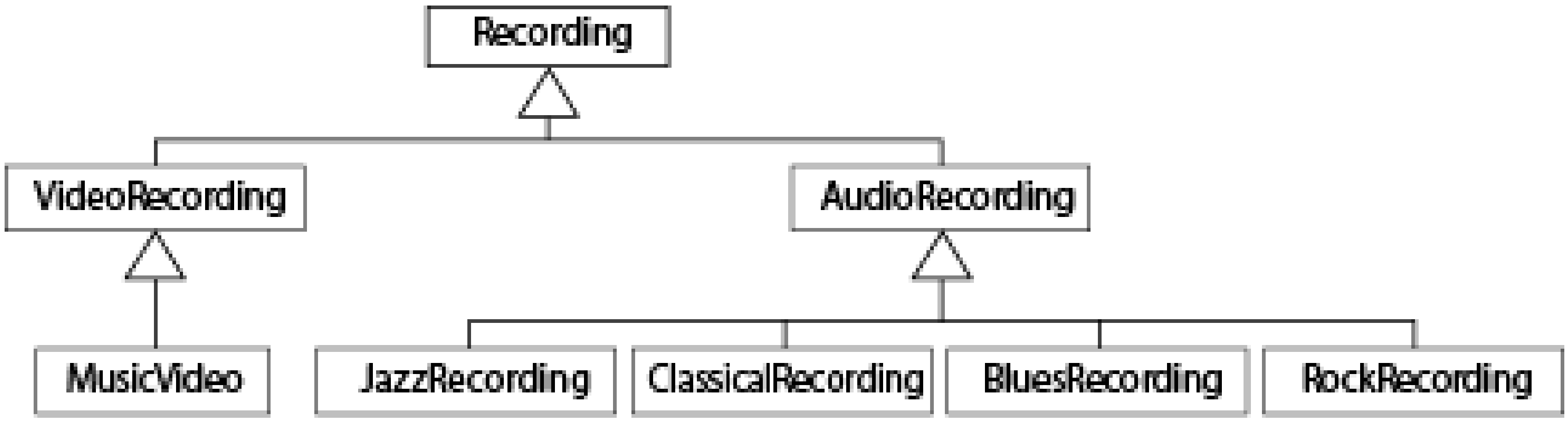
Generalization/Specialization Relation

- Generalization is shown as a solid-line arrow from the child (the more specific element) to the parent (the more general element) this type of relationship is also called inheritance.
- Should be used to define class hierarchies based on abstraction

Generalization/Specialization Relation

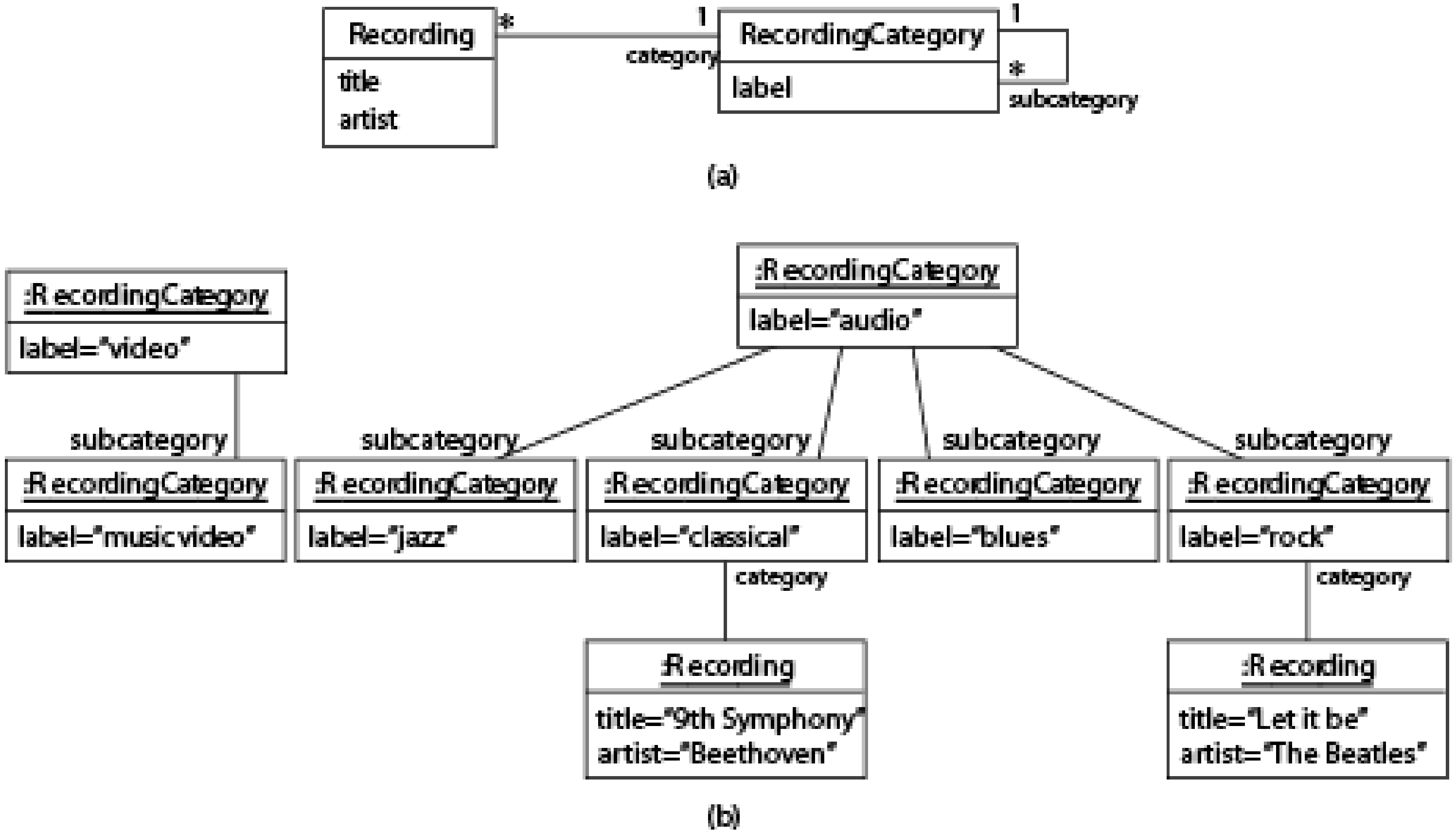


Avoiding unnecessary generalizations



Inappropriate hierarchy of classes, which should be instances
Ask yourself: Does this class require any operations that will be done differently than the other classes? If answer is no, don't make it a class!

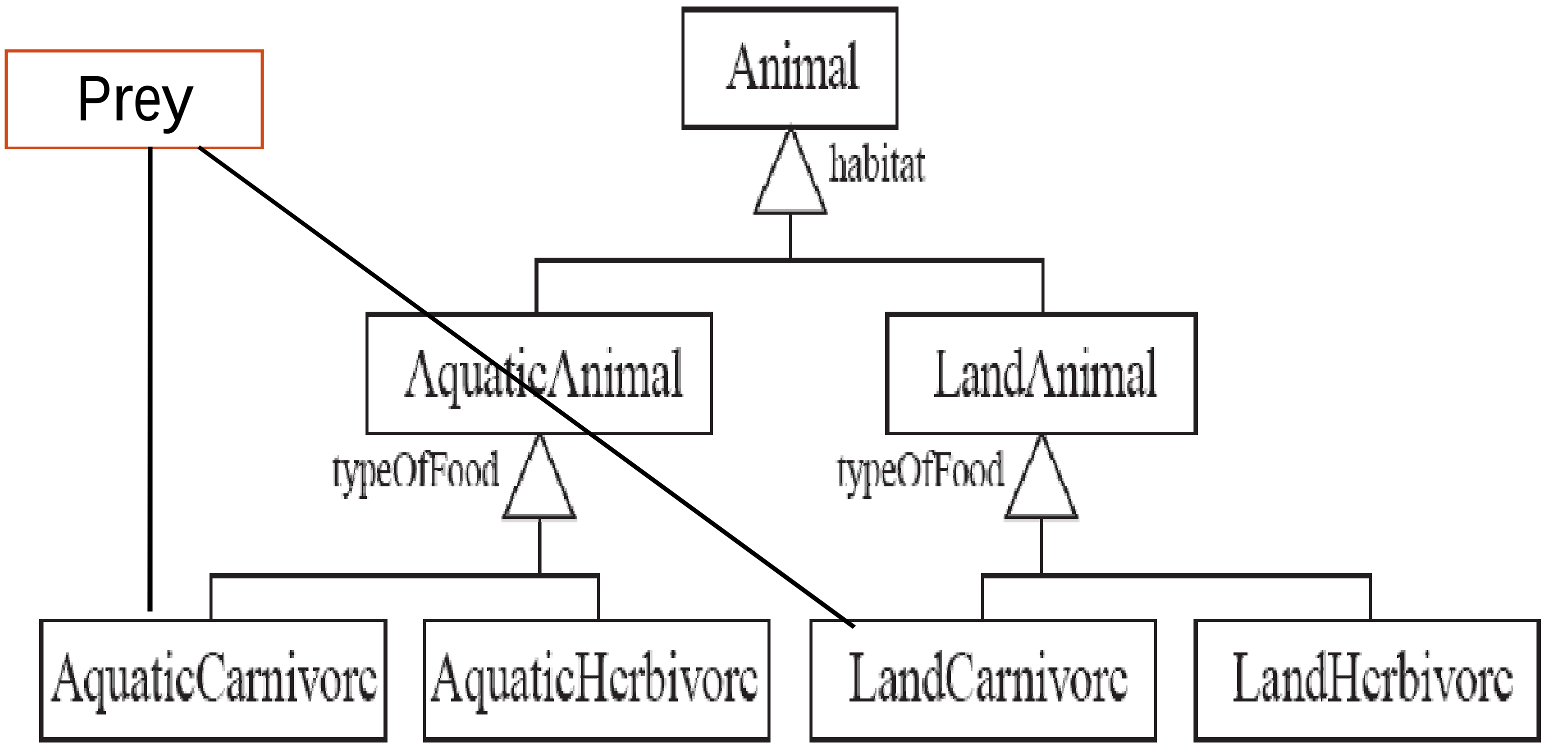
Avoiding unnecessary generalizations (cont)



Improved class diagram, with its corresponding instance diagram

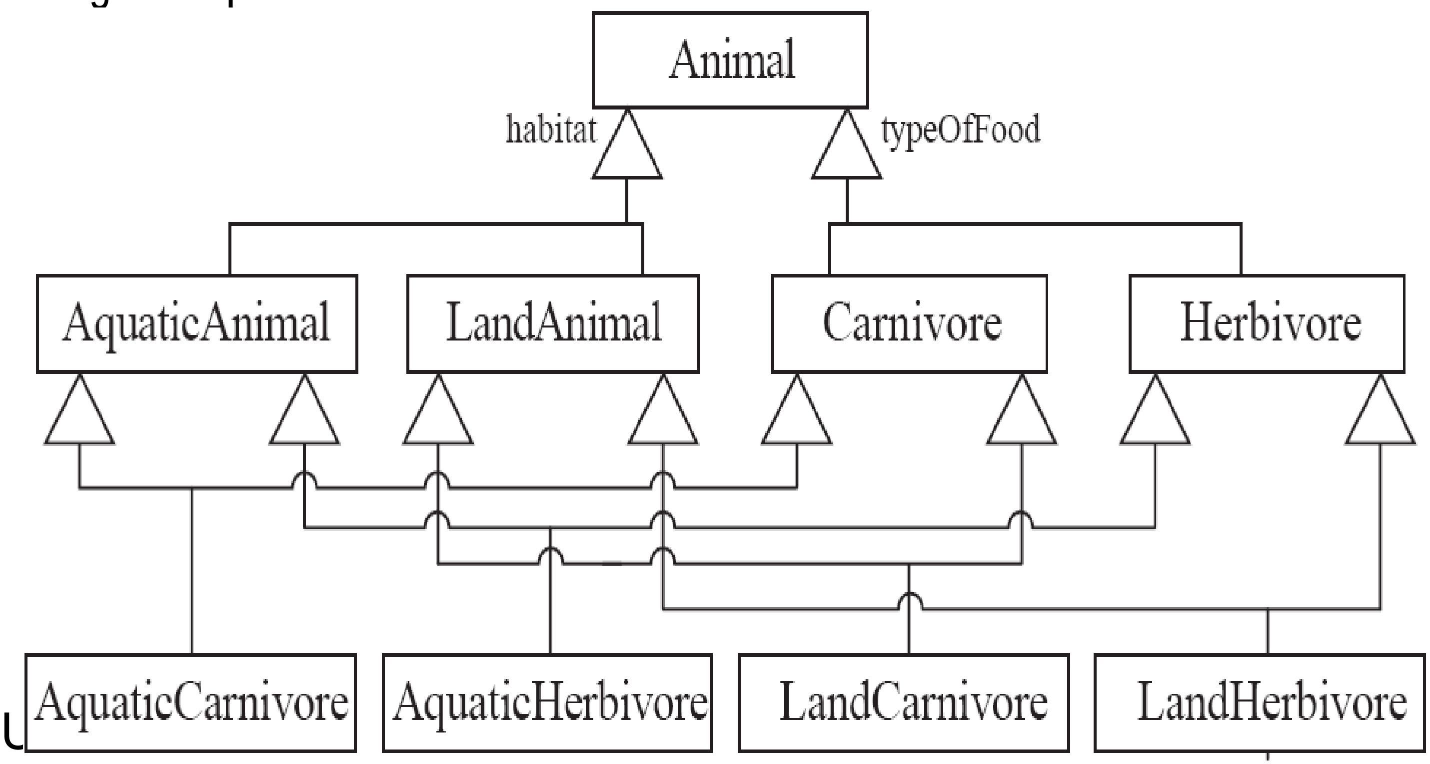
Handling multiple discriminators

- Creating higher-level generalization
- Say we had a Prey class, we would need TWO associations instead of just one.



Handling multiple discriminators

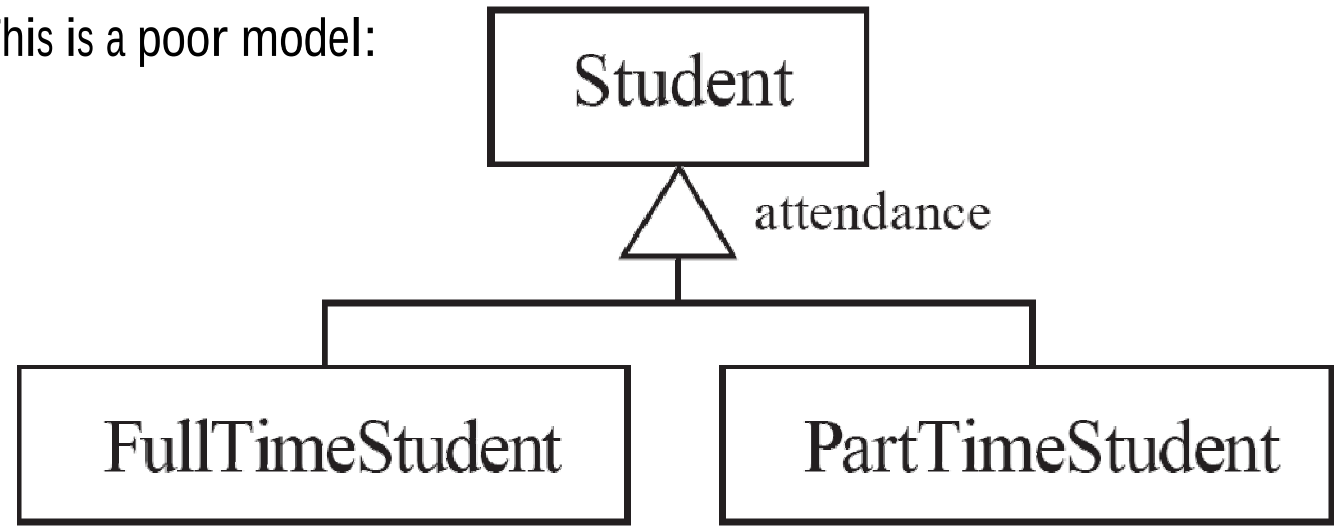
- Using multiple inheritance



- U

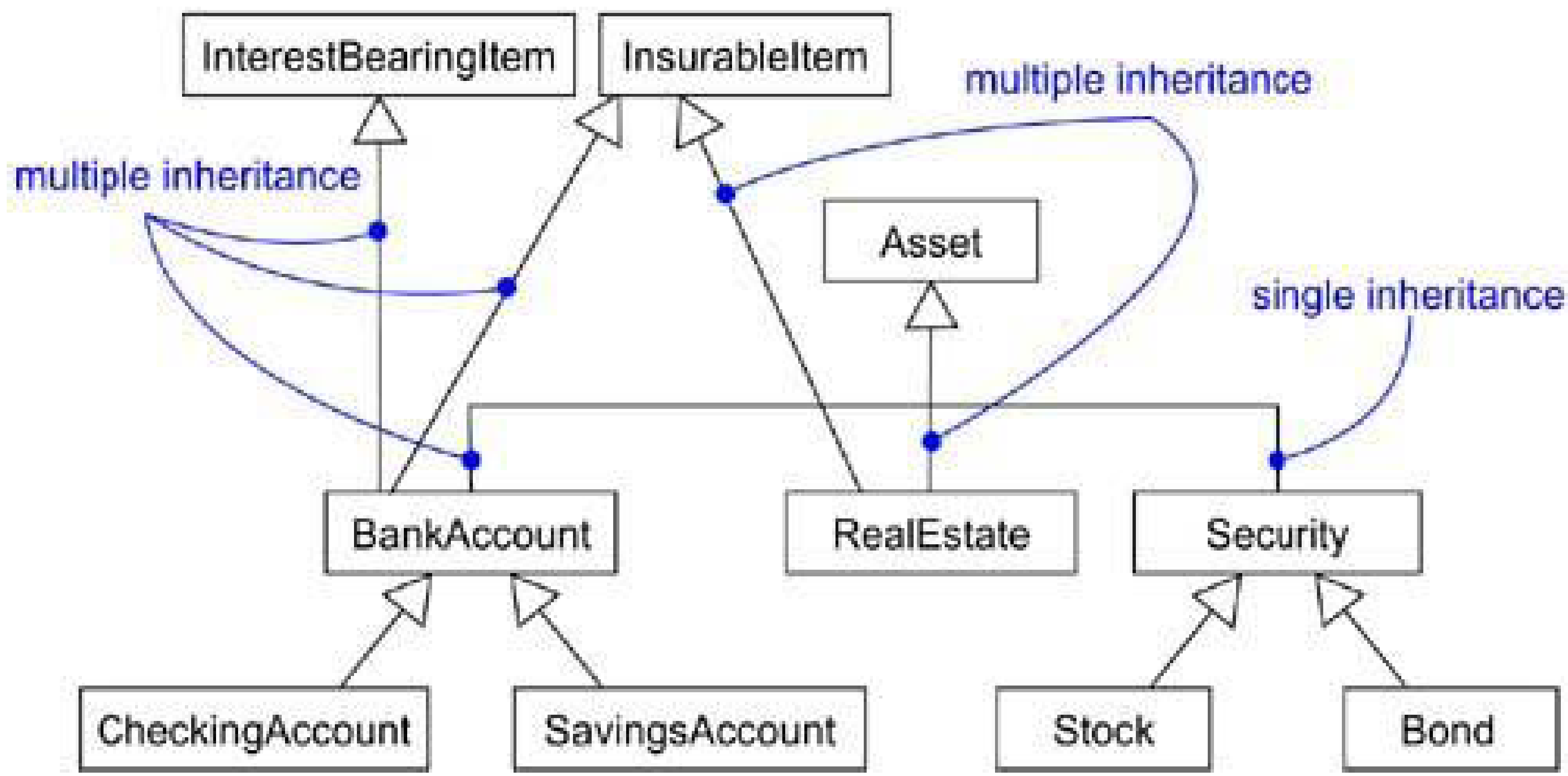
Avoiding having instances change class

- An instance should never need to change class
- This is a poor model:

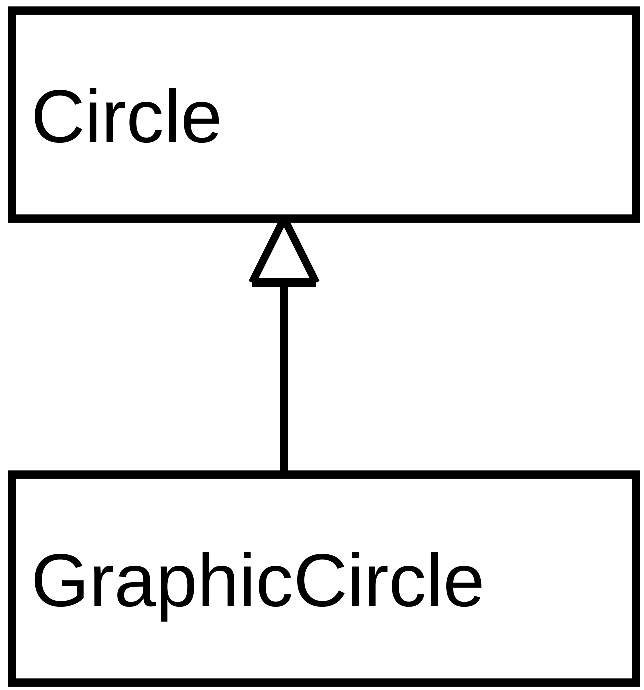


- A bit better solution, but then we lose the polymorphism advantage for any operations that differ between FullTimeStudent and PartTimeStudent:

Multiple inheritance



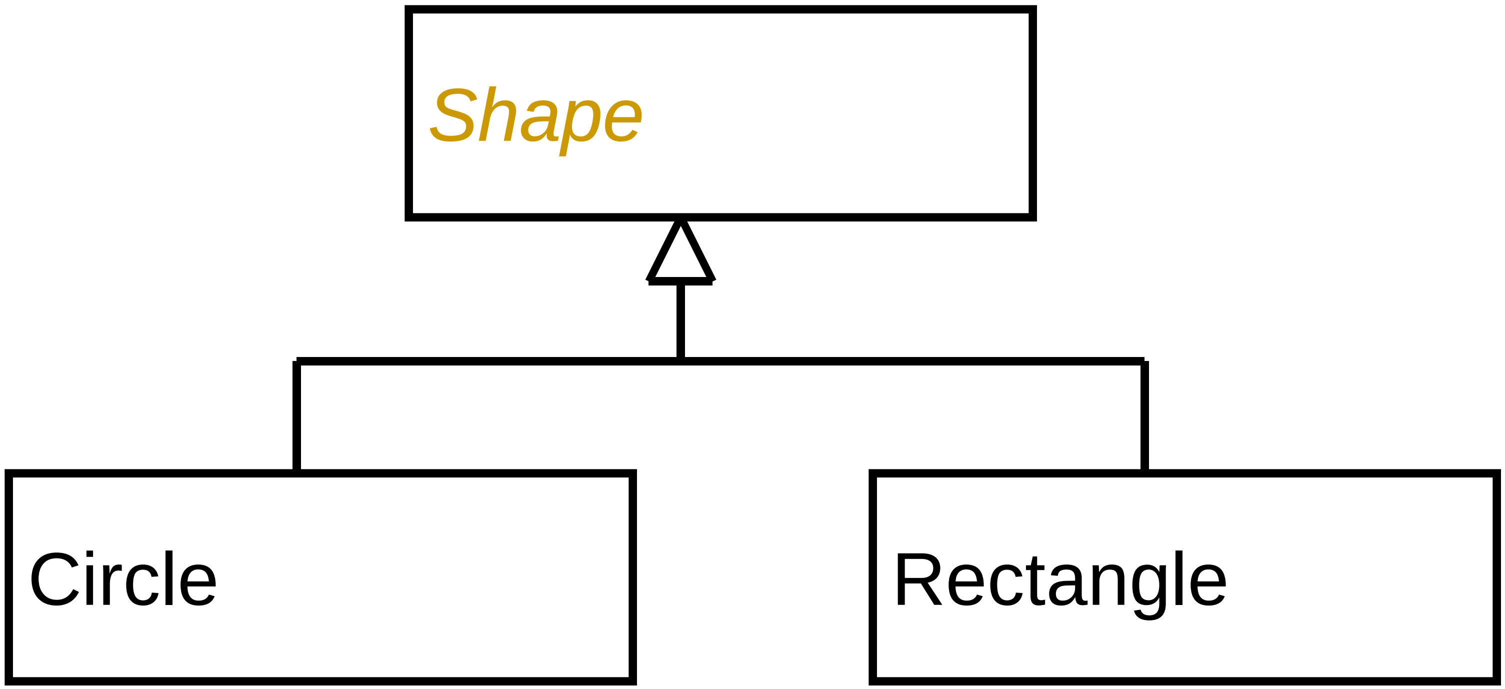
Generalization (Inheritance) e.g.



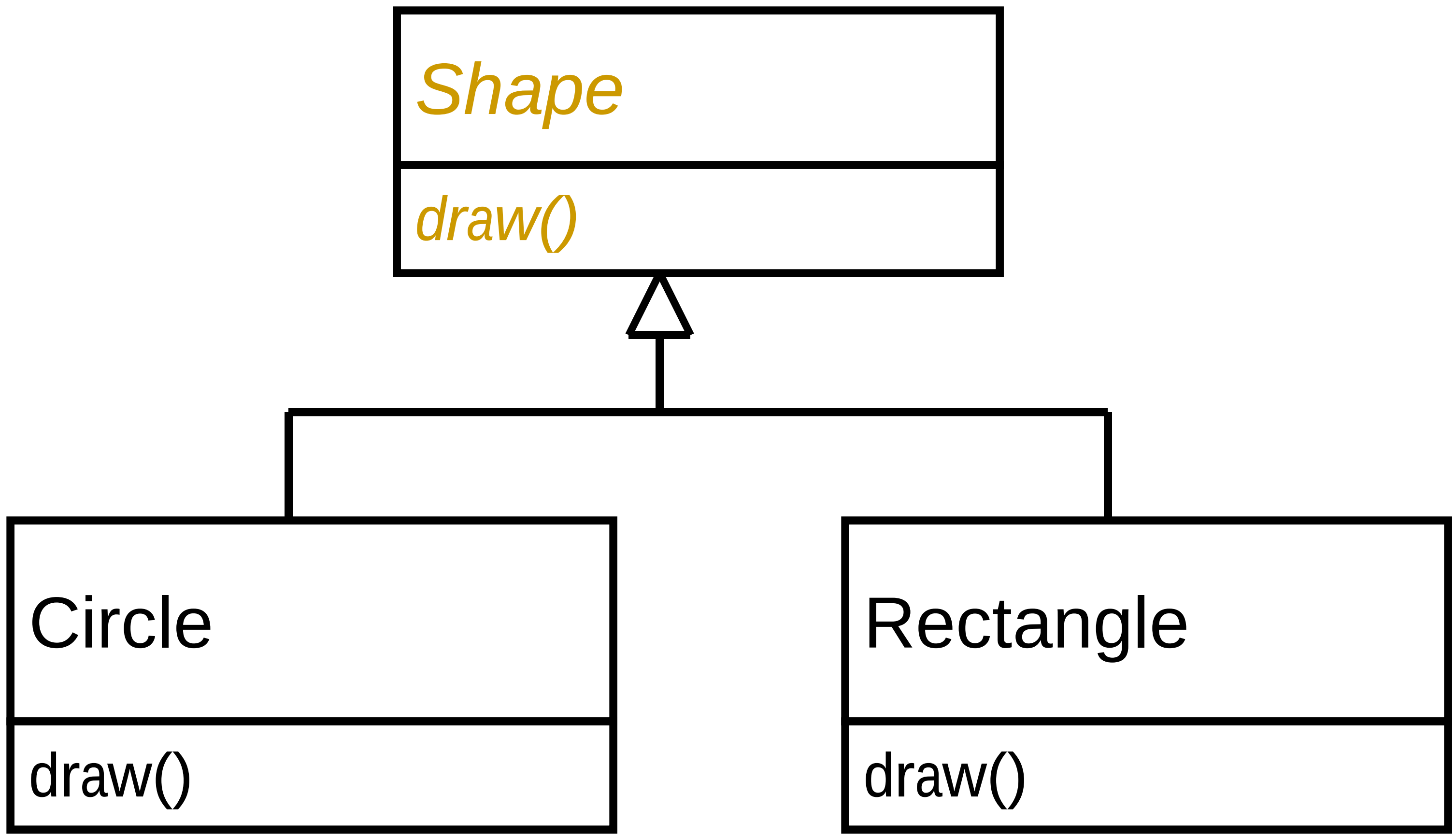
Inheritance - Implementation

```
public class Circle {

}



public class GraphicCircle extends Circle {

}
```

Abstract Class



Abstract Methods (Operations)



Abstract class and method Implementation

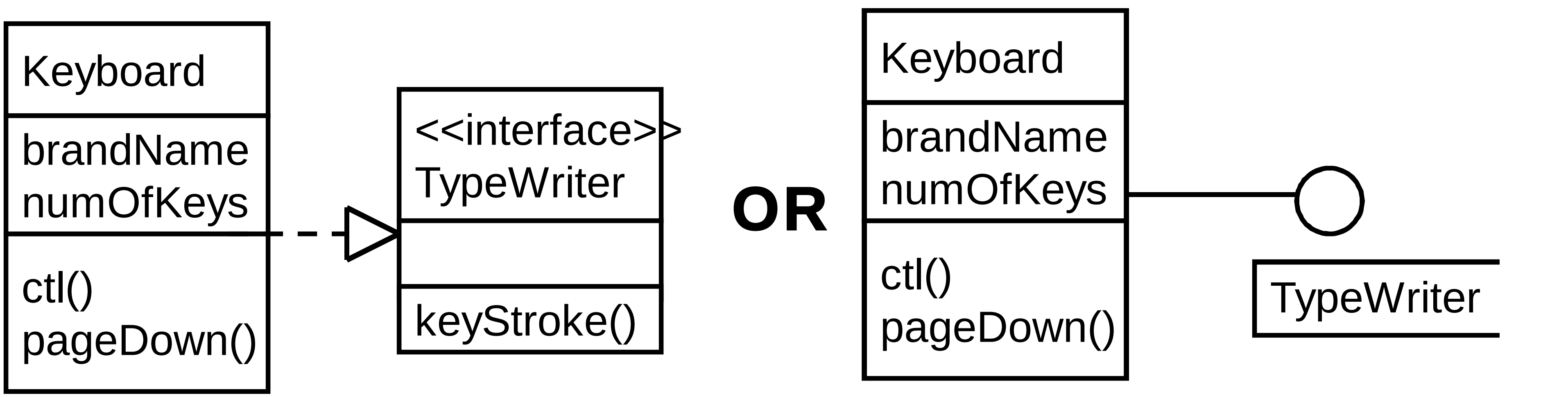
```
public abstract class Shape {
   public abstract draw(); //declare
without implementation
    ………
}

public class Circle {
    public draw(){
    …….
    }
    …..
}
```

Class Relationships

- Association
- Generalization
- *Realization*
- Dependency

Realization- Interface

- Interface is a set of operation the class carries out



Realization - Implementation

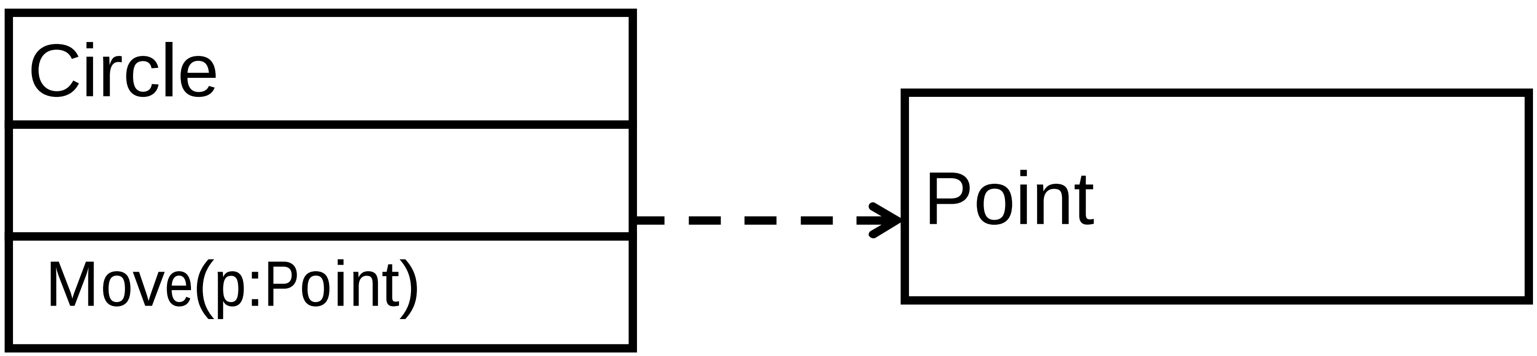
```
public interface TypeWriter {
    void keyStroke()

}

public class KeyBoard implements TypeWriter {
    public void  keyStroke(){
    ………

   }
}
```

Class Relationships

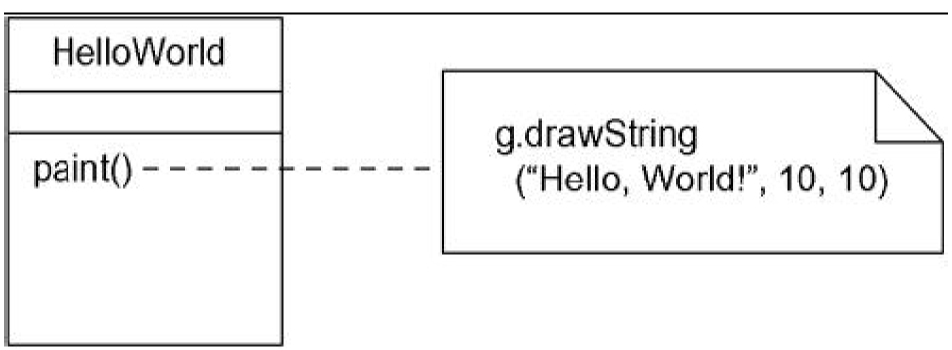
- Association
- Generalization
- Realization
- *Dependency*

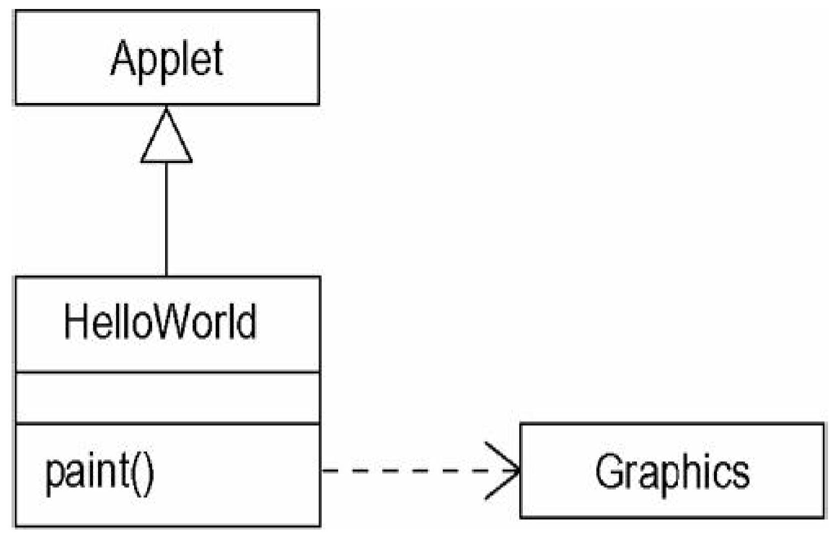
Dependency : A Special Case of Association

- Change in specification of one class can change the other class. This can happen when one class is using another class.



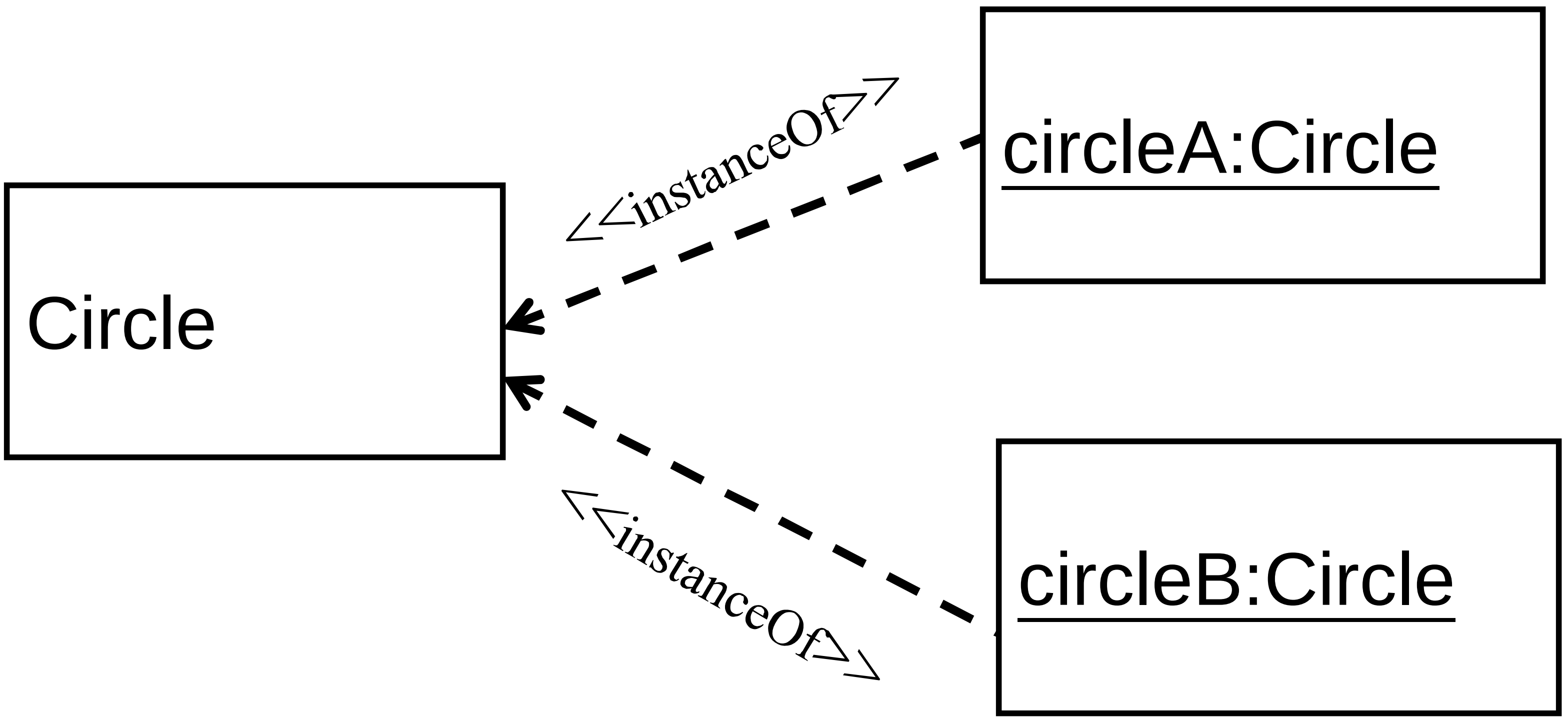
```
import java.awt.Graphics;
class HelloWorld extends java.applet.Applet {
public void paint (Graphics g) {
g.drawString("Hello, World!", 10, 10);
}
}
```





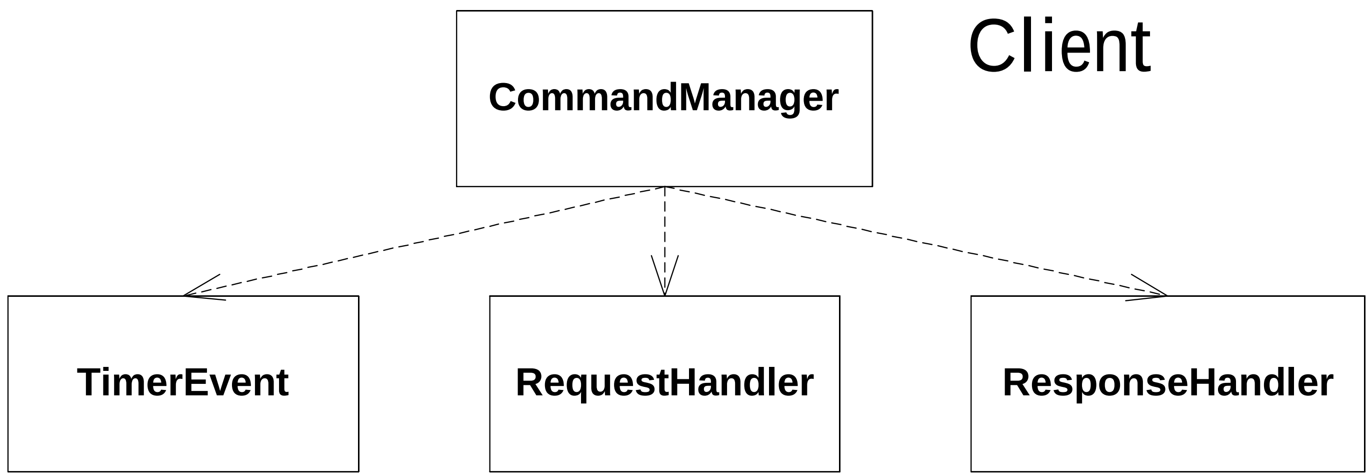
Dependency cont

- Dependency relationship can be used to show relationships between classes and objects.



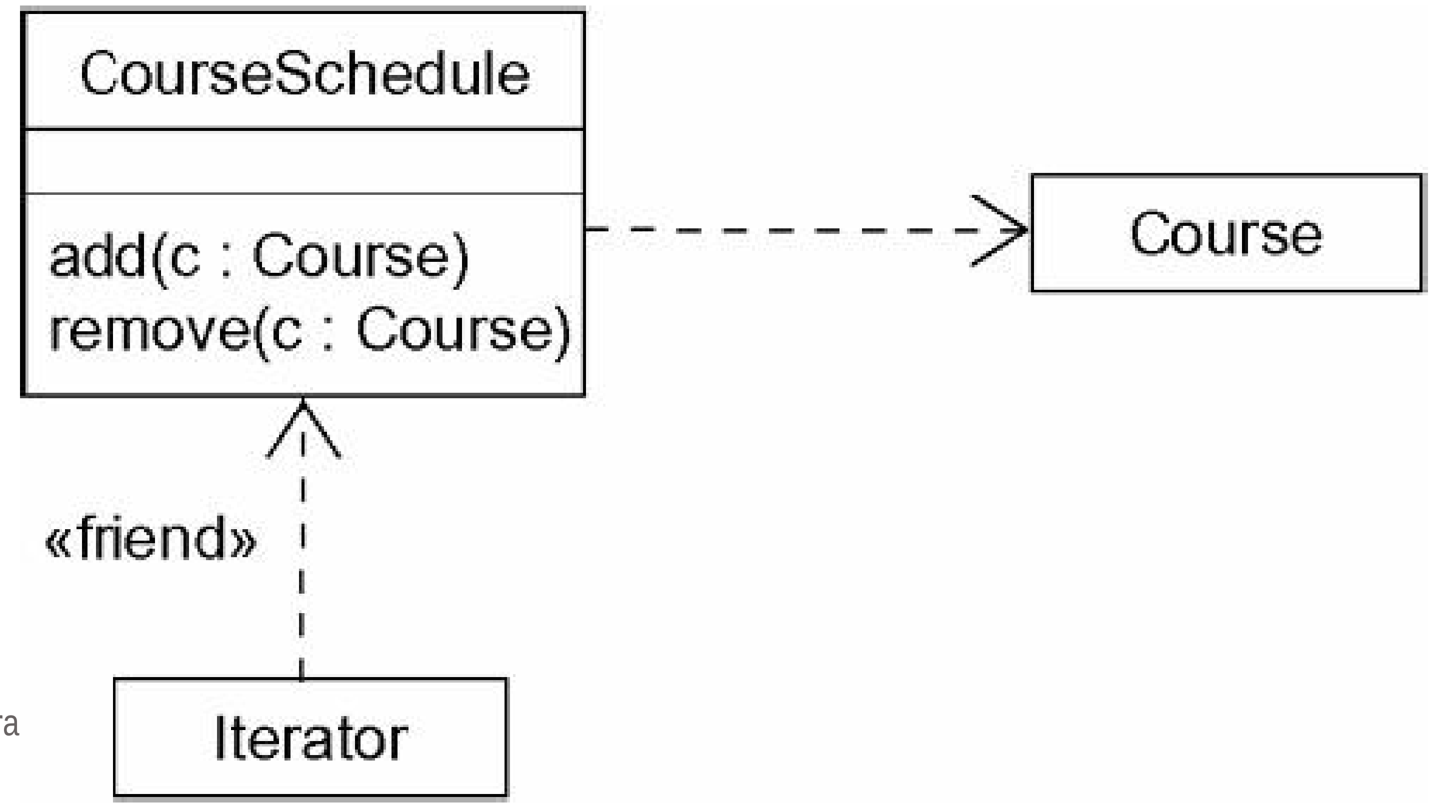
Dependency

Dependency



CommandManager (Client class) depends on services provided by the other three server classes

This figure shows a dependency from CourseSchedule to Course, because Course is used in both the add and remove operations of CourseSchedule.



Class Diagram - Example

- Draw a class diagram for a information modeling system for a school.
  - School has one or more Departments.
  - Department offers one or more Subjects.
  - A particular subject will be offered by only one department.
  - Department has instructors and instructors can work for one or more departments.
  - Student can enrol in upto 5 subjects in a School.
  - Instructors can teach upto 3 subjects.
  - The same subject can be taught by different instructors.
  - Students can be enrolled in more than one school.

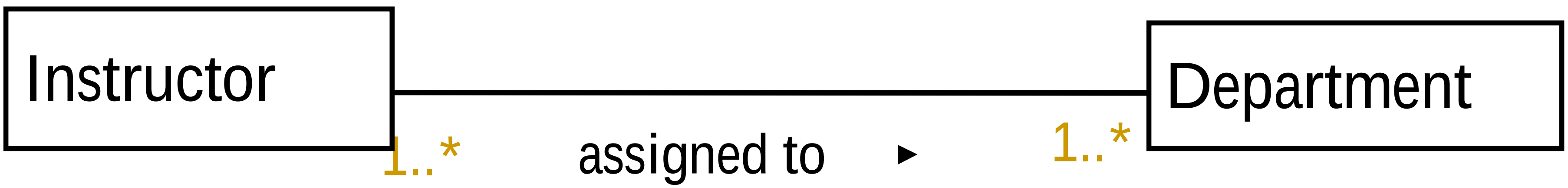
Class Diagram - Example

- School has one or more Departments.

- Department offers one or more Subjects.
- A particular subject will be offered by only one department.

Class Diagram - Example

- Department has Instructors and instructors can work for one or more departments.



- Student can enrol in upto 5 Subjects.

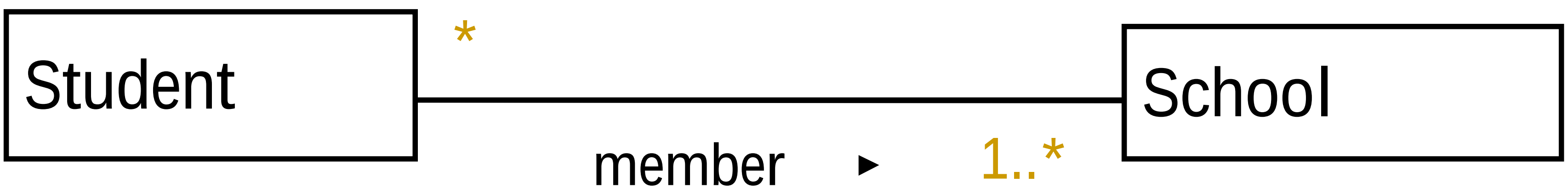
Class Diagram - Example

- Instructors can teach up to 3 subjects.
- The same subject can be taught by different instructors.

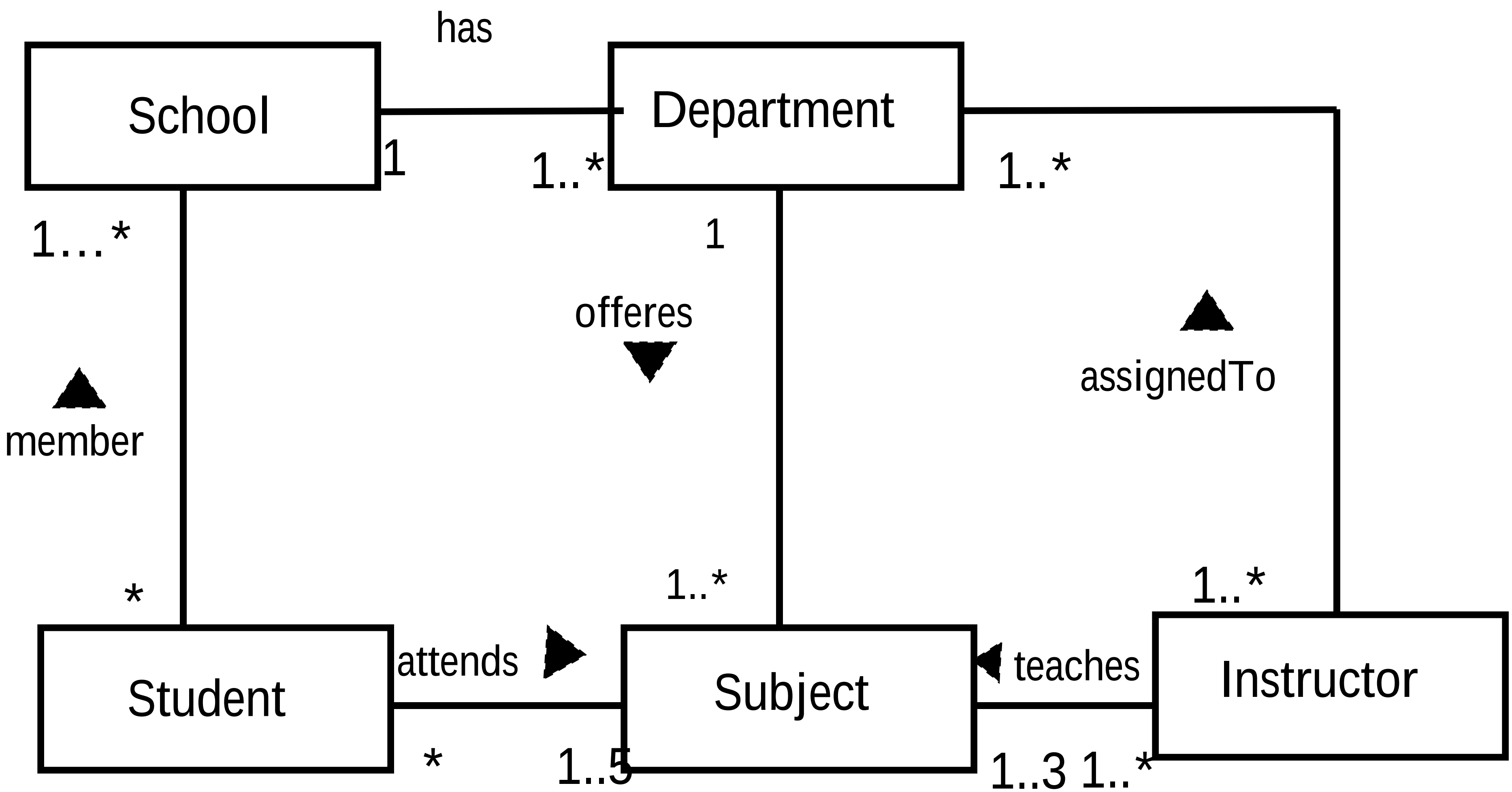
Hari Prasad Pokhrel (hpokhrel24@gmail.com)

Class Diagram - Example

- Students can be enrolled in more than one school.



Class Diagram Example



Hari Prasad Pokhrel (hpokhrel24@gmail.com)

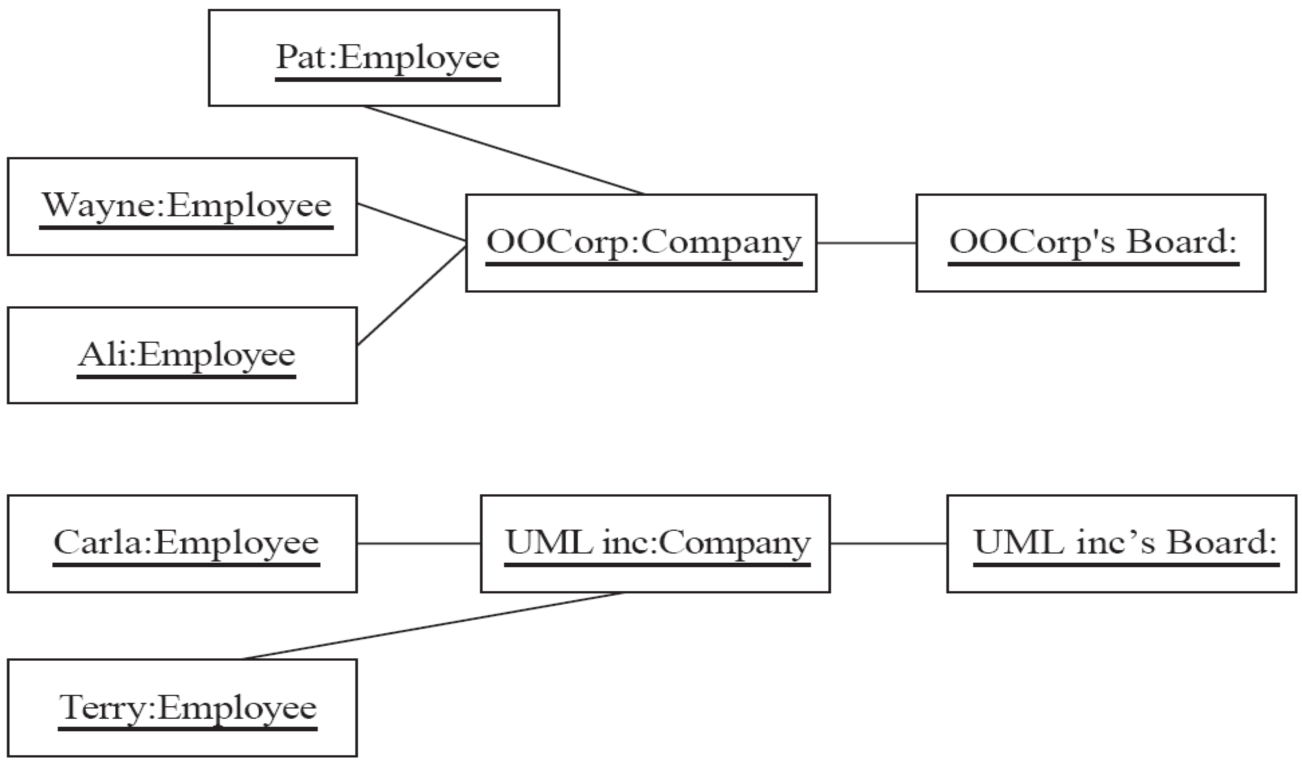
Object Diagram

- Object Diagram shows the relationship between objects.

- Unlike classes objects have a state.

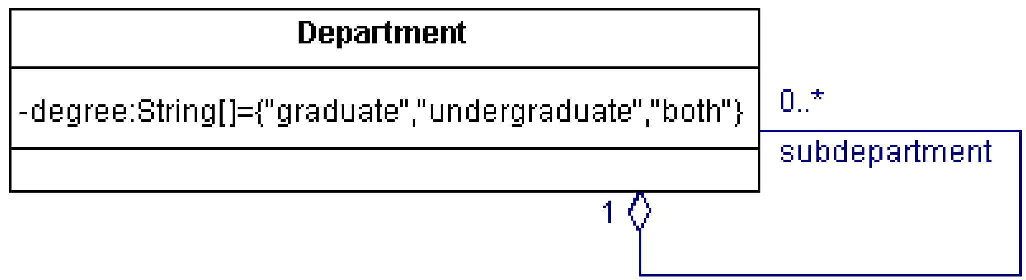
Object Diagrams

- A *link* is an instance of an association
  - In the same way that we say an object is an instance of a class

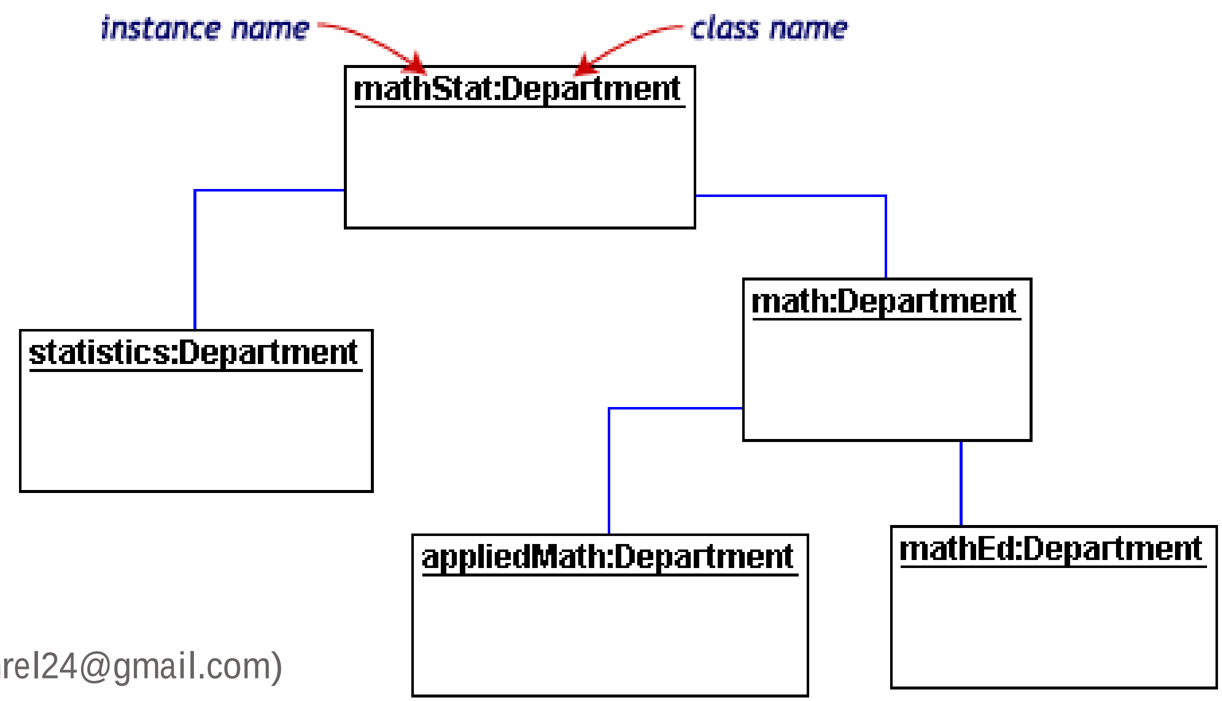


Object Diagram
Track Instance Behavior

- Class Diagram

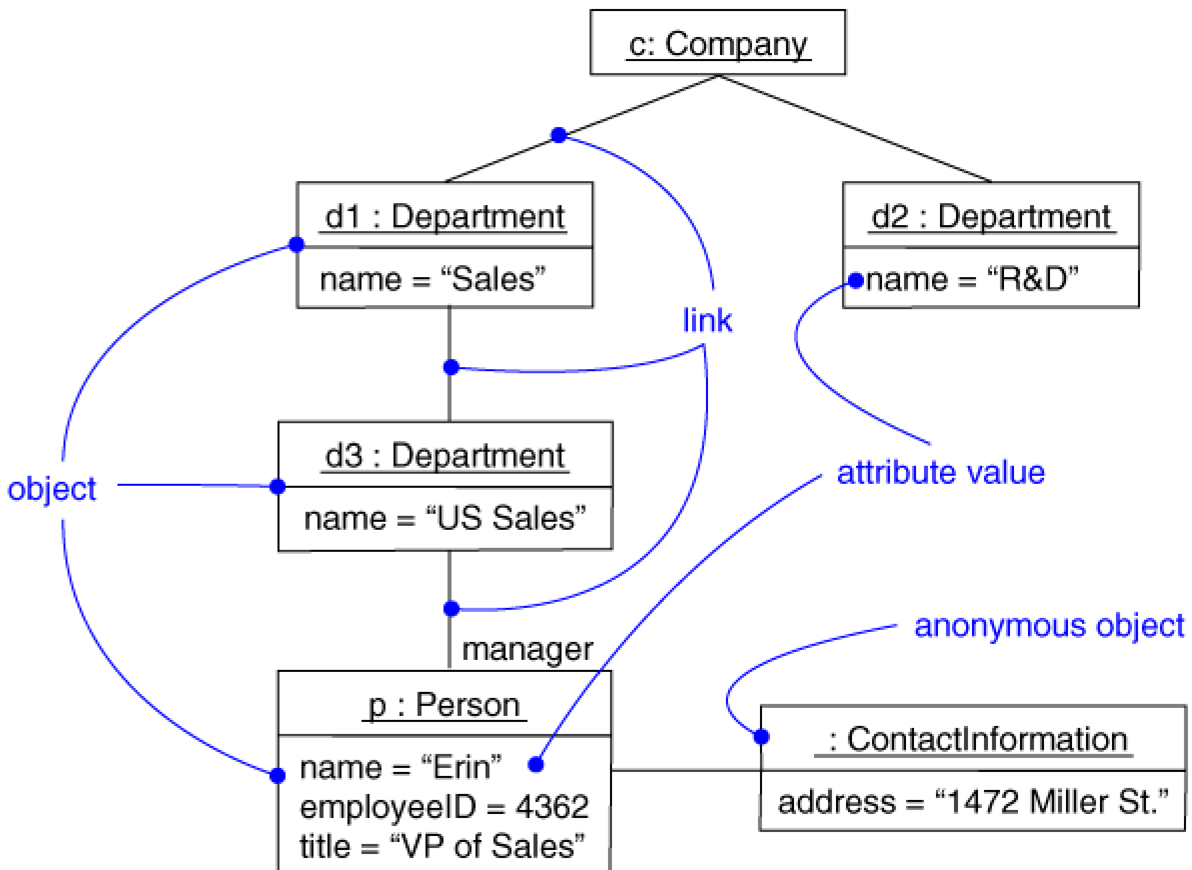


- Instance Diagram



Object Diagram

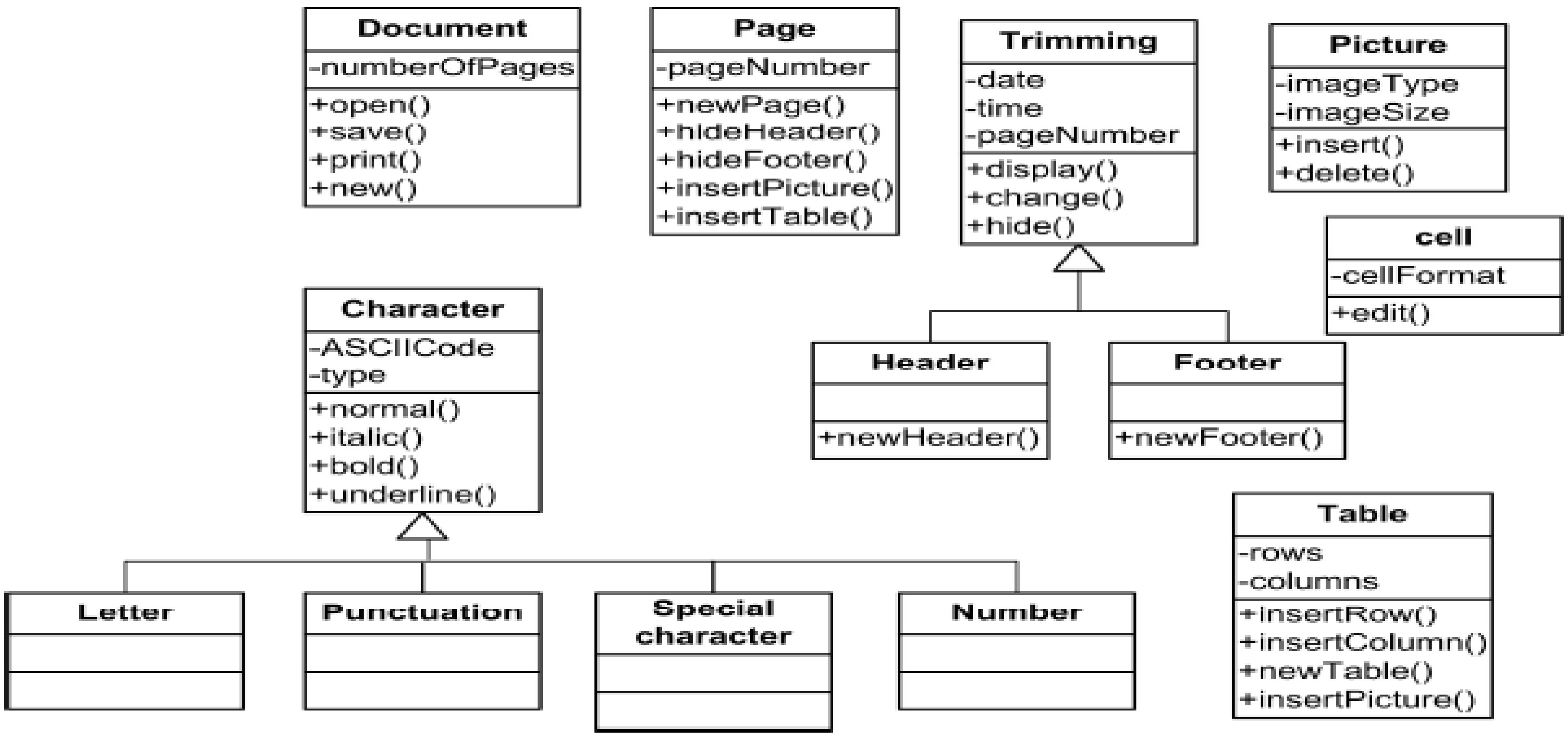
- Captures Instances and Links

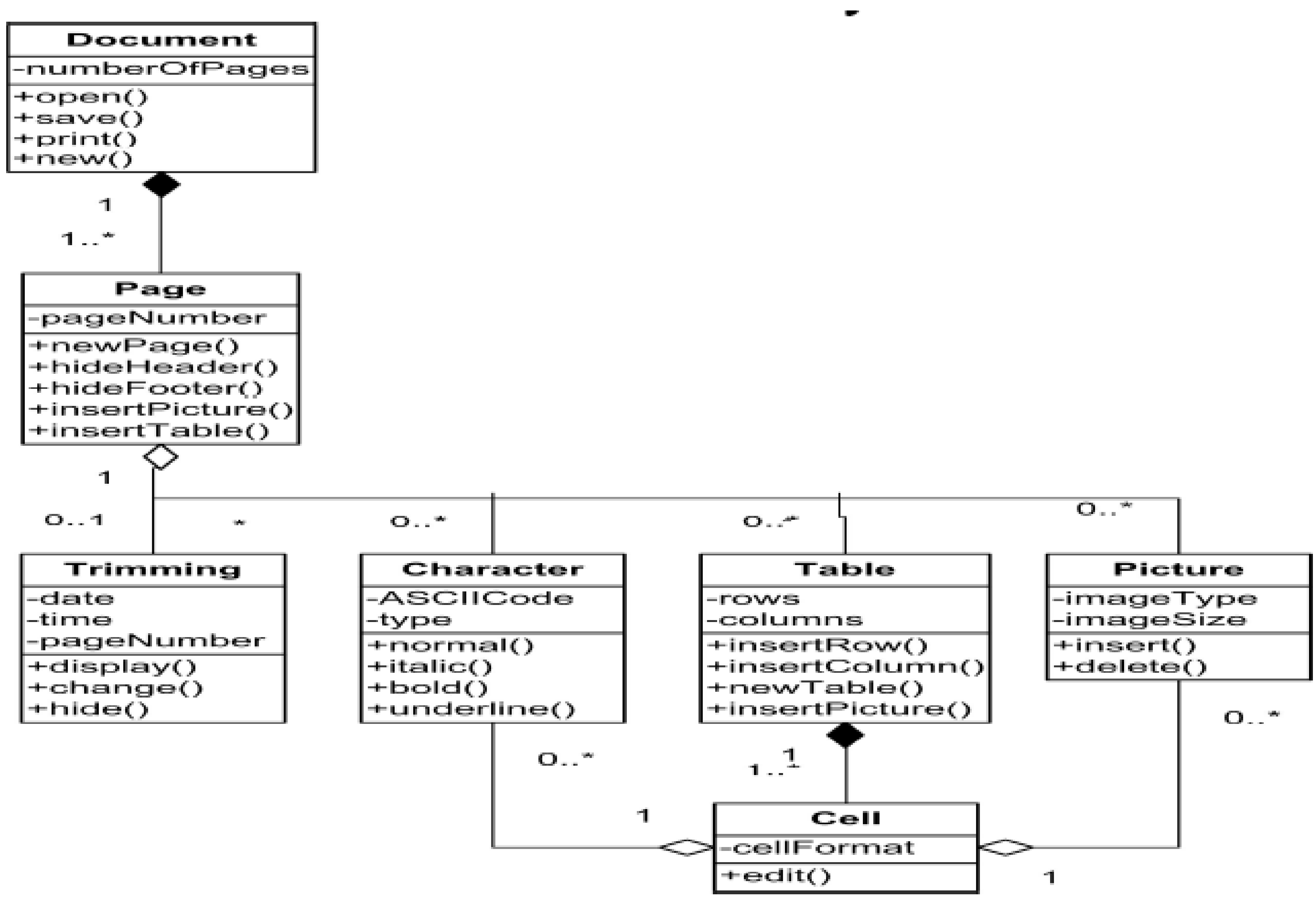


Describing the use of a word processor

A user can *open* a new or existing document. Text i
entered through a keyboard. A document is made u
of several pages and each page is made up of a
header, body and footer. Date, time and page numb
may *be added* to header or footer. Document body i
made up of sentences, which are themselves made
up of words and punctuation characters. Words are
made up of letters, digits and/or special characters.
Pictures and tables may *be inserted* into the
document body. Tables are made up of rows and
columns and every cell in a table can contain both
text and pictures. Users can *save* or *print* document

- Nouns (underlined in previous) are either classes or their attributes
- Verbs (italicised in previous) are class operations
- Main handled entity: document




Document
-numberOfPages
+open()
+save()
+print()
+new()
1
1..*
Page
-pageNumber
+newPage()
+hideHeader()
+hideFooter()
+insertPicture()
+insertTable()
1
0..1
*
0..*
0..*
0..*
Trimming
-date
-time
-pageNumber
+display()
+change()
+hide()
Character
-ASCIICode
-type
+normal()
+italic()
+bold()
+underline()
Table
-rows
-columns
+insertRow()
+insertColumn()
+newTable()
+insertPicture()
Picture
-imageType
-imageSize
+insert()
+delete()
0..*
0..*
1
1..*
1
Cell
-cellFormat
+edit()
1

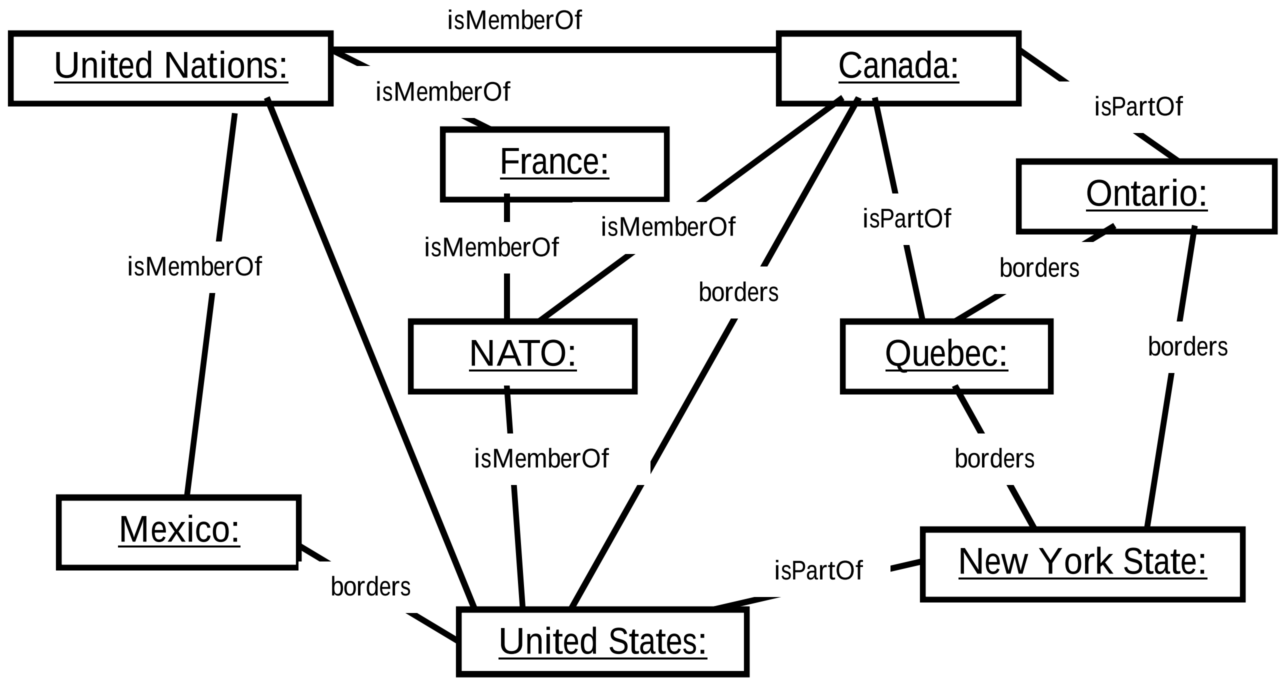
Question:

- Draw a class diagram corresponding to the following situation:
  - A media player (Most software media players support an array of media formats, like Quicktime for Macs or Windows Media Player for Windows)that can handle sound, images and sequences of images. Each type of medium requires a “plug-in” (plug-in is a set of software components that adds specific capabilities to a larger software application), although some plug-ins can handle more than one type of medium.
  - An organization has three categories of employee: professional staff, technical staff and support staff. The organization also has departments and divisions. Each employee belongs to either a department or a division. Assume that people will never need to change from one category to another.

Your Answer:

Question:

- Draw a class diagram that could generate the object diagram shown below:



Hari Prasad Pokhrel (hpokhrel24@gmail.com)

Your answer:

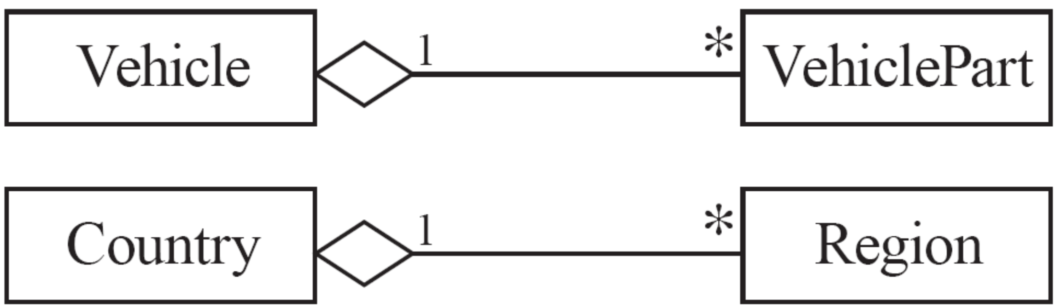
Associations versus generalizations in object diagrams

- Associations describe the relationships that will exist between *instances* at run time.
  - When you show an instance diagram generated from a class diagram, there will be an instance of *both* classes joined by an association

- Generalizations describe relationships between *classes* in class diagrams.
  - They do not appear in instance diagrams at all.
  - An instance of any class should also be considered to be an instance of each of that class's superclasses

More Advanced Features: Aggregation

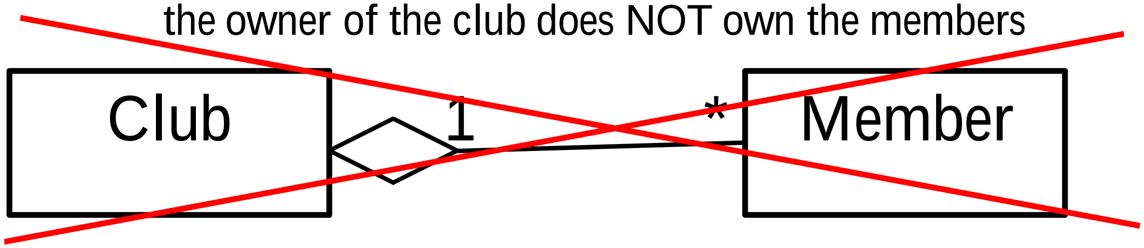
- Aggregations are special associations that represent 'part-whole' relationships.
  - The 'whole' side is often called the *assembly* or the *aggregate*
  - This symbol is a shorthand notation association named `isPartOf`



When to use an aggregation

- As a general rule, you can mark an association as an aggregation if the following are true:
  - You can state that
    - the parts ‘are part of’ the aggregate
    - or the aggregate ‘is composed of’ the parts
  - When something owns or controls the aggregate, then they also own or control the parts

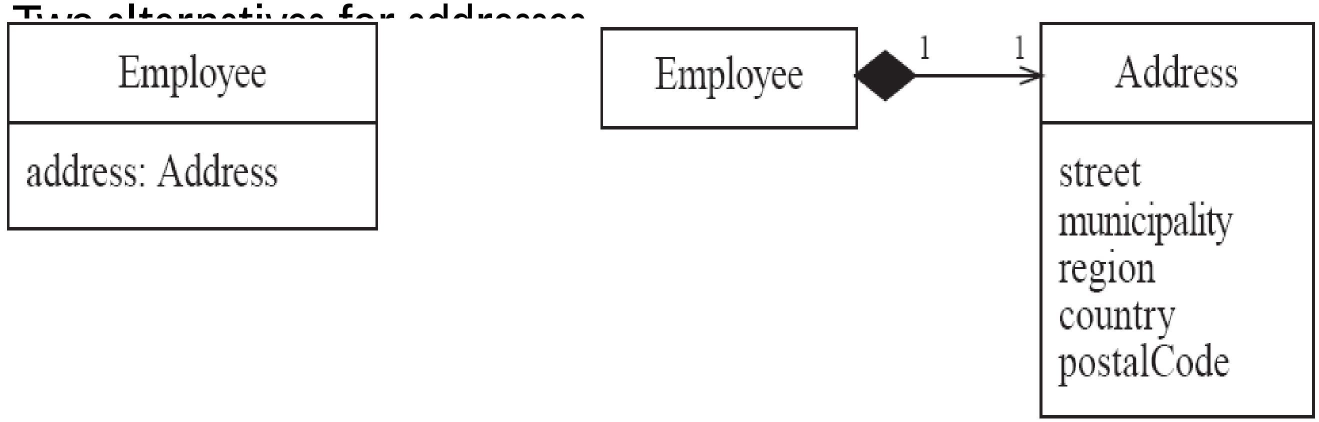
NOTE: Might be able to say a person is part of a club BUT the owner of the club does NOT own the members



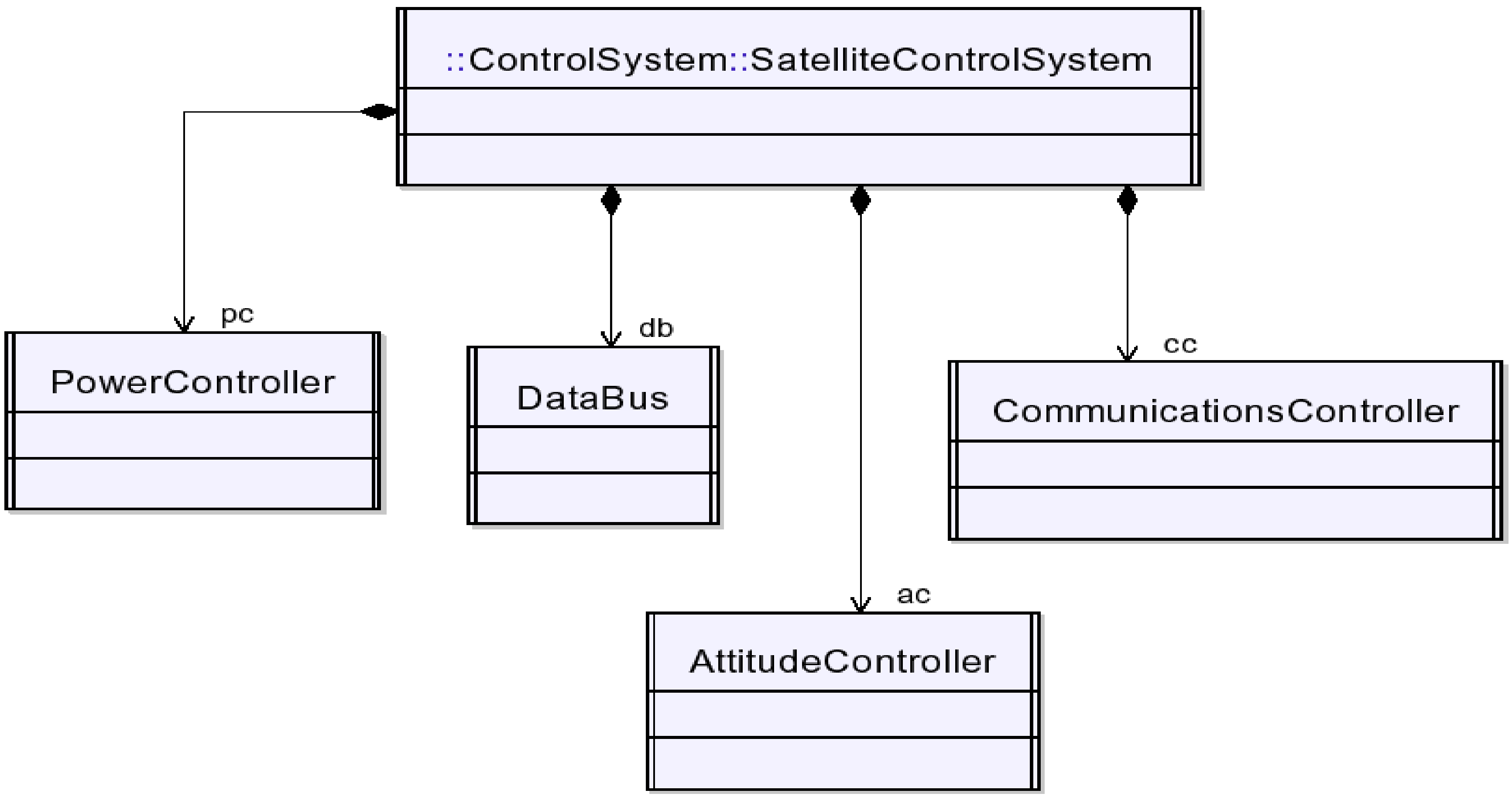
Composition: A Special case of Aggregation

- A *composition* is a strong kind of aggregation
- Composition is shown as a solid filled diamond, with the diamond attached to the class that is the composite. Composition is a form of aggregation that requires coincident lifetime of the part with the whole and singular ownership; i.e. the part is owned by only one whole and is deleted when the whole is deleted
  - if the aggregate is destroyed, then the parts are destroyed as well

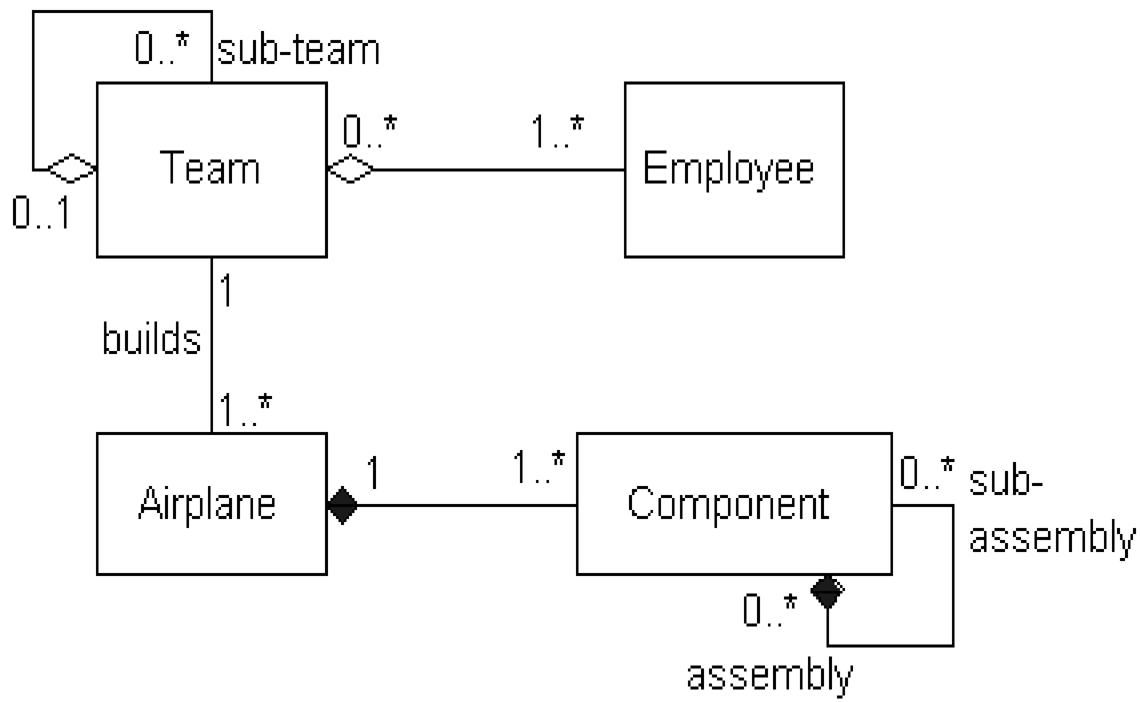
- Two alternatives for addresses



Composition example



Aggregation vs Composition



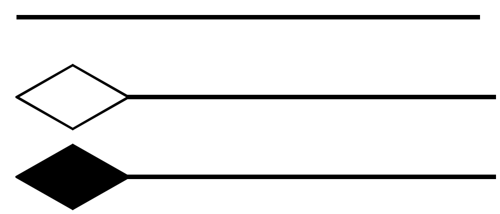
Propagation

- A mechanism where an operation in an aggregate is implemented by having the aggregate perform that operation on its parts
- At the same time, properties of the parts are often propagated back to the aggregate
- Propagation is to aggregation as inheritance is to generalization.
  - The major difference is:
    - inheritance is an implicit mechanism
    - propagation has to be programmed when required
  - Eg. Deleting a polygon means deleting the line segments

- Marking a part-whole association as an aggregation using the diamond symbol is optional. Leaving it as an ordinary association is not an error, whereas marking a non-aggregation with a diamond is an error, therefore, **when in doubt, leave it out!**

Question

- For each of the following associations, indicate whether it should be
  an *ordinary association*
  a *standard aggregation*
  a *composition*



a) A telephone and its handset

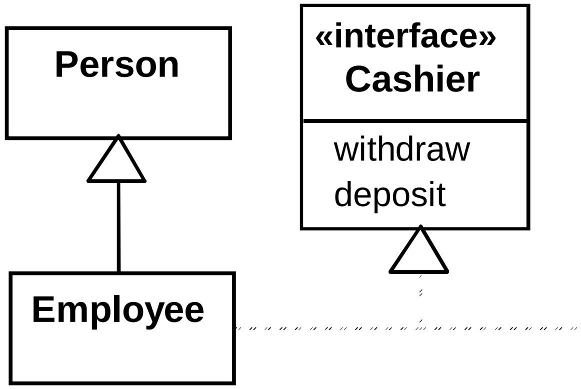
b) A school and its teachers

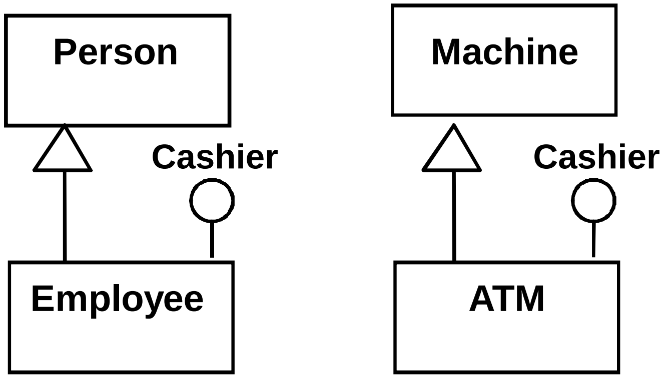
c) A book and its chapters

Hari Prasad Pokhrel (hpokhrel24@gmail.com)

Interfaces

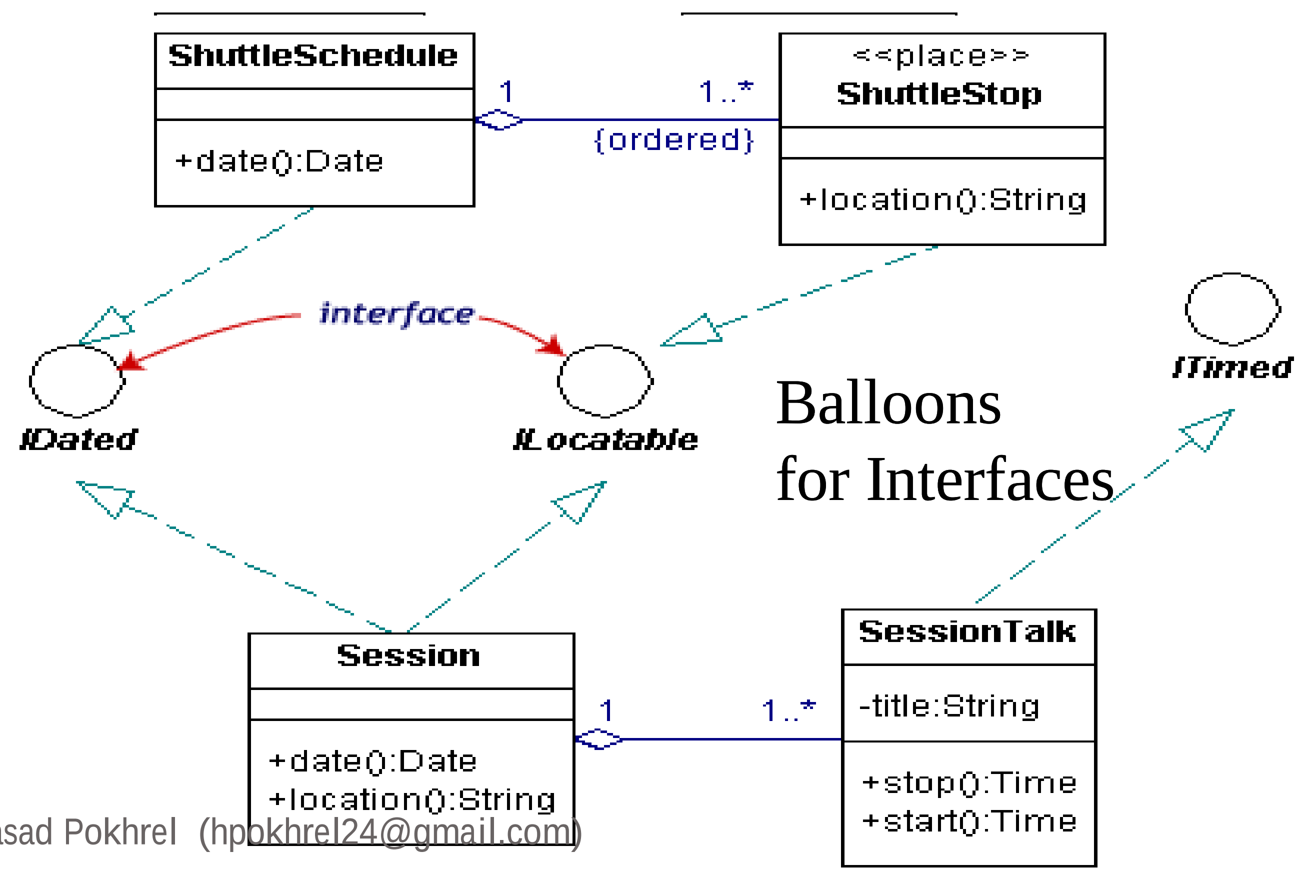
- An interface describes a *portion of the visible behaviour* of a set of objects.
  - An *interface* is similar to a class, except it lacks instance variables and implemented methods
  - Although Employee and ATM share common operations they have different superclasses. This means they cannot be put in the same inheritance hierarchy; therefore the interface called Cashier is used
  - A key advantage of using interfaces is that they can reduce what is called *coupling* between classes.
  - Inheritance indicates an ***isa*** relationship, interfaces indicate a ***can-be-seen-as*** relationship





Interfaces and Stereotypes

- Interface – Operation Signatures (Abstract Class)
- Stereotype – Extend UML with New Modeling Items Created from Existing Kinds (Classes)



The Process of Developing Class Diagrams

- You can create UML models at different stages and with different purposes and levels of details
  - **Exploratory domain model**:
    - Developed in domain analysis to learn about the domain
  - **System domain model**:
    - Models aspects of the domain represented by the system
  - **System model**:
    - Includes also classes used to build the user interface and system architecture

System domain model vs System model

- The *system domain model* omits many classes that are needed to build a complete system
  - Can contain less than half the classes of the system.
  - Should be developed to be used independently of particular sets of
    - user interface classes
    - architectural classes
- The complete *system model* includes
  - The system domain model
  - User interface classes
  - Architectural classes such as the database, files, servers, clients
  - Utility classes

Suggested sequence of activities

- Identify a first set of candidate **classes**
- Add **associations** and **attributes**
- Find **generalizations**
- List the main **responsibilities** of each class
- Decide on specific **operations**
- **Iterate** over the entire process until the model is satisfactory
  - Add or delete classes, associations, attributes, generalizations, responsibilities or operations
  - Identify interfaces
  - Apply design patterns
- *Don't be too disorganized. Don't be too rigid either.*

Identifying classes

- When developing a domain model you tend to *discover* classes
- When you work on the user interface or the system architecture, you tend to *invent* classes
  - Needed to solve a particular design problem
  - (Inventing may also occur when creating a domain model)
- Reuse should always be a concern
  - Frameworks
  - System extensions
  - Similar systems

A simple technique for discovering domain classes

- Look at a source material such as a description of requirements
- Extract the *nouns* and *noun phrases*
- Eliminate nouns that:
  - are redundant
  - represent instances
  - are vague or highly general
  - not needed in the application. For example in a domain model, you would eliminate classes that represent command or menus in the UI. As a rule of thumb, a class is only needed in a domain model if you have to store or manipulate instances of it in order to implement a requirement
- Pay attention to classes in a domain model that represent *types of users* or other actors

Identifying associations and attributes

- Start with classes you think are most **central** and important
- Decide on the clear and obvious data it must contain and its relationships to other classes.
- Work outwards towards the classes that are less important.
- Avoid adding many associations and attributes to a class
  - A system is simpler if it manipulates less information

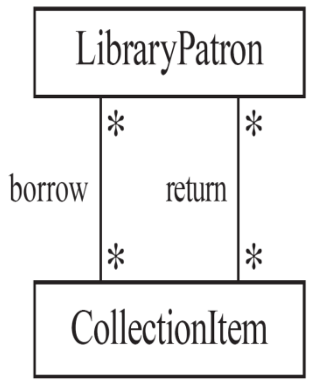
Tips about identifying and specifying valid associations

- An association should exist if a class
  - *possesses*
  - *controls*
  - *is connected to*
  - *is related to*
  - *is a part of*
  - *has as parts*
  - *is a member of*, or
  - *has as members*

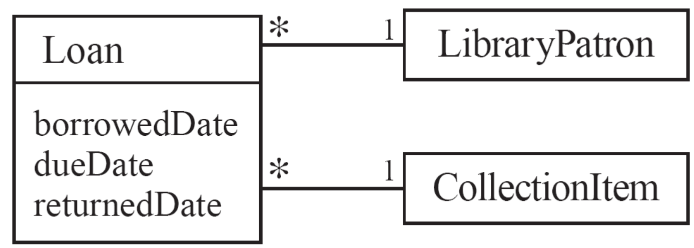
  some other class in your model
- Specify the multiplicity at both ends
- Label it clearly.

Actions versus associations

- A common mistake is to represent *actions* as if they were associations



Bad, due to the use of associations that are actions



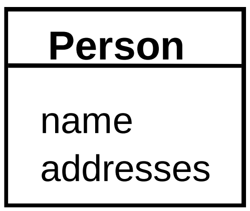
Better: The **borrow** operation creates a **Loan** object and the return operation set the **returnedDate** attribute

Identifying attributes

- Look for information that must be maintained about each class
- Several nouns rejected as classes, may now become attributes
- An attribute should generally contain a simple value
  - E.g. string, number

Tips about identifying and specifying valid attributes

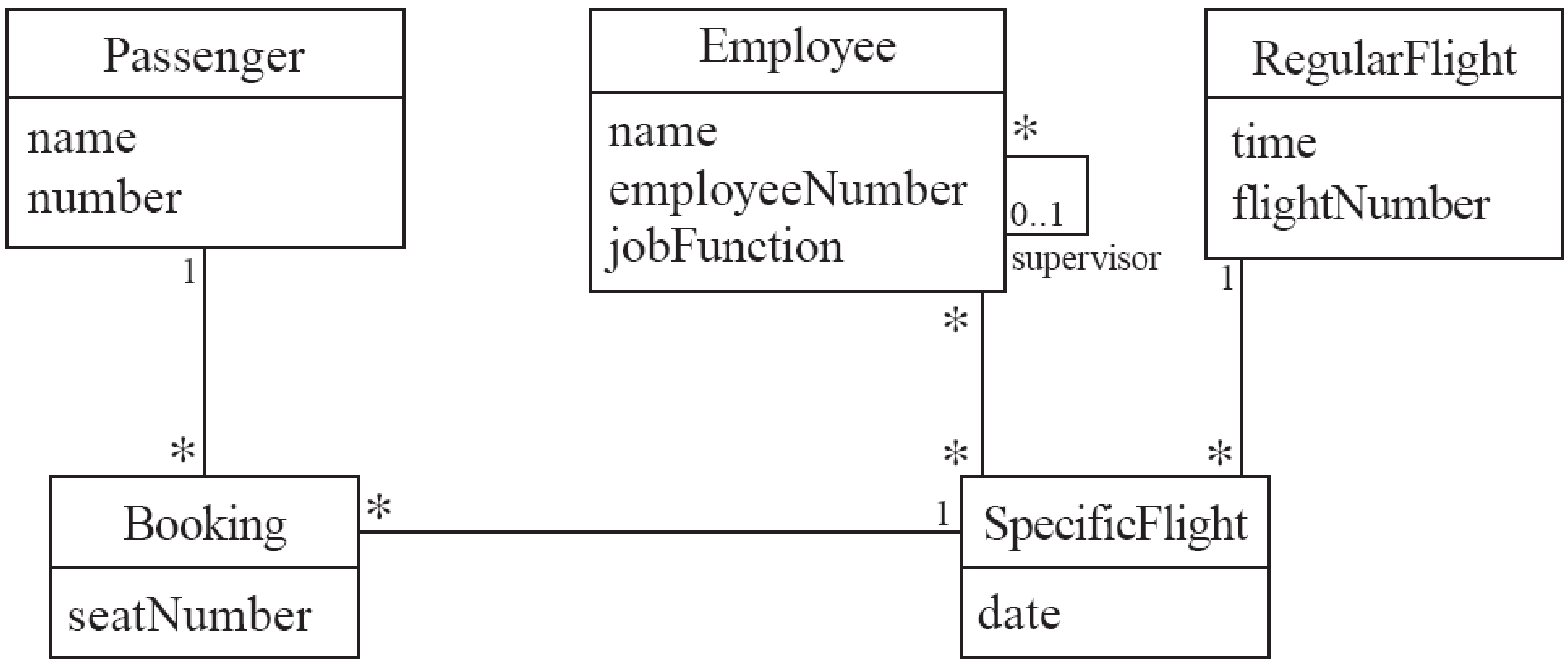
- It is not good to have many duplicate attributes
- If a subset of a class's attributes form a coherent group, then create a distinct class containing these attributes



Bad due to a plural attribute

Hari Prasad Pokhrel (hpokhrel24@gmail.com)

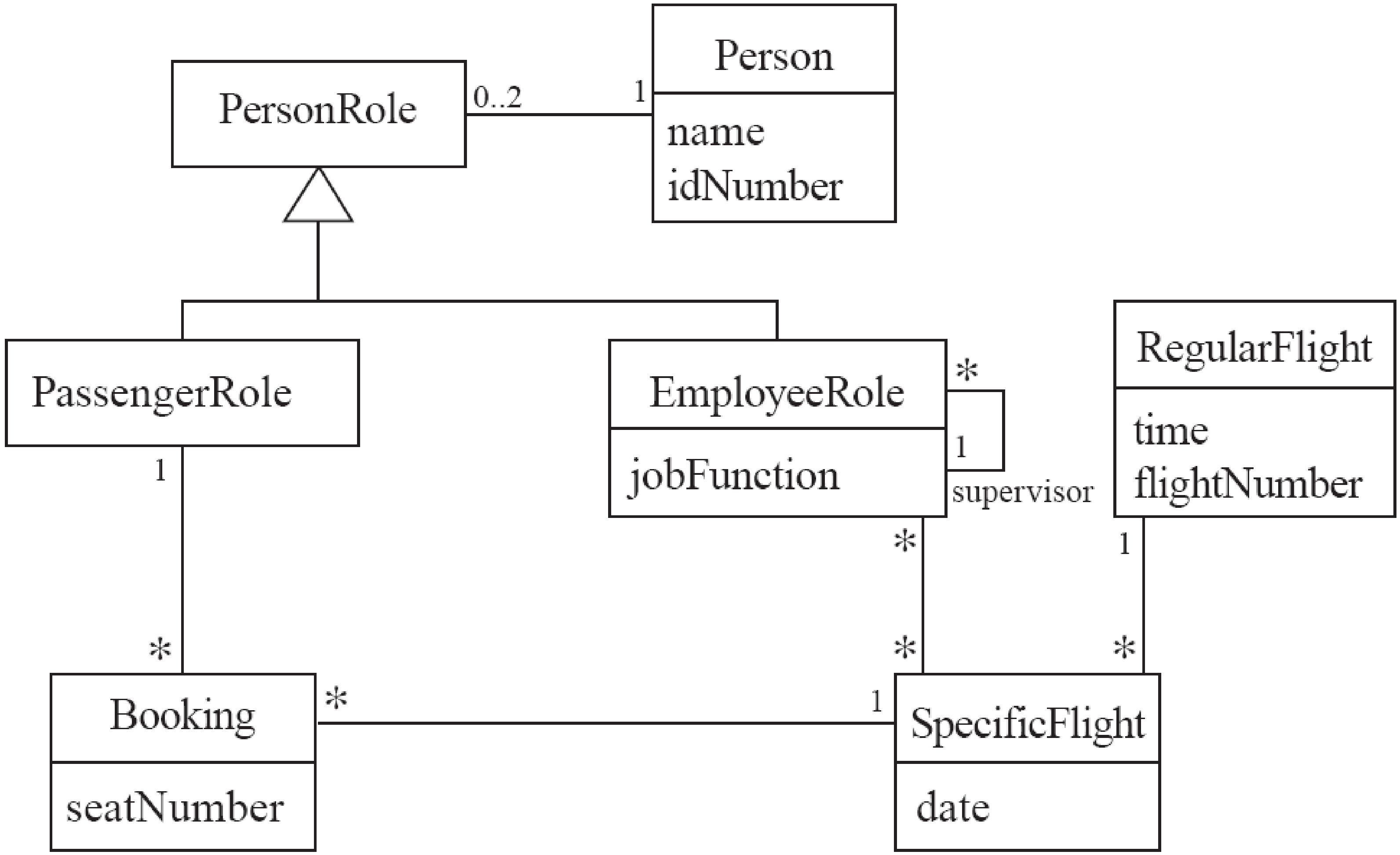
An example (attributes and associations)



Identifying generalizations and interfaces

- There are two ways to identify generalizations:
  - bottom-up
    - Group together similar classes creating a new superclass
  - top-down
    - Look for more general classes first, specialize them if needed
- Create an *interface*, instead of a superclass if
  - The classes are very dissimilar except for having a few operations in common
  - One or more of the classes already have their own superclasses
  - Different implementations of the same class might be available

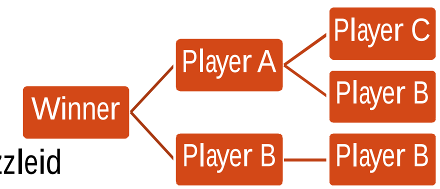
An example (generalization)



Case Study

- Implement the game of Boggle
- http://www.hasbro.com/scrabble/en_US/boggleGame.cfm
- Some Specs:

- There is an admin mode, tournament mode, fun play mode
  - To get into admin mode type: *java UWOBoggle -admin*
  - To get into tournament mode or fun play mode type: *java UWOBoggle*
    - Then select Tournament or Fun Play
- In Admin Mode Must Be able to
  - Manage players:
    - Add, delete and modify players. Players have a first name, last name, userid and password.
    - Sort players by first and last name
    - Reset password
  - Manage puzzles (puzzles are made up of 16 letters and puzzleid)
    - Add, delete, bulkload, solve (using a stored dictionary of valid words) and list by puzzleid



  - Manage tournaments (2-8 players per tournament, tournament has a unique tournament id and a name. A tournaments run in 1-3 battles of 2 per round, winning player moves to the next round)
    - Add tournaments
      - Add players to a tournament, add puzzles (all players play the same puzzle on each round but in pairs, higher scorer for each pair moves to the next round, in odd numbers, the highest scorer get a buy to the next round) to a tournament (must keep track of the score for each player for each round and the winner of each pairing)
    - Delete tournaments
    - List tournaments by tournament id or by tournament name
    - Print tournaments

More Specs

- In Tournament or Fun Play Mode a user must be able to:
  - Log on (gets 3 attempts and then kicked out)
  - Play puzzles as follows
    - Start the puzzle
    - Given 3 minutes to find words
    - Given a score at the end
    - If the puzzle is not a tournament puzzle, the player can see the solution for the puzzle.
- In just Tournament Mode
  - Player sees a list of tournaments that he/she is participating in
  - Picks an ongoing tournament
  - Plays the puzzles
  - Sees if he/she moves to the next round

- In just Fun Mode
  - Pick a puzzle from the list of puzzles
  - Play the puzzle
  - View the top 3 scores for that puzzle
  - See the top 3 players (who have the highest scores for any games)

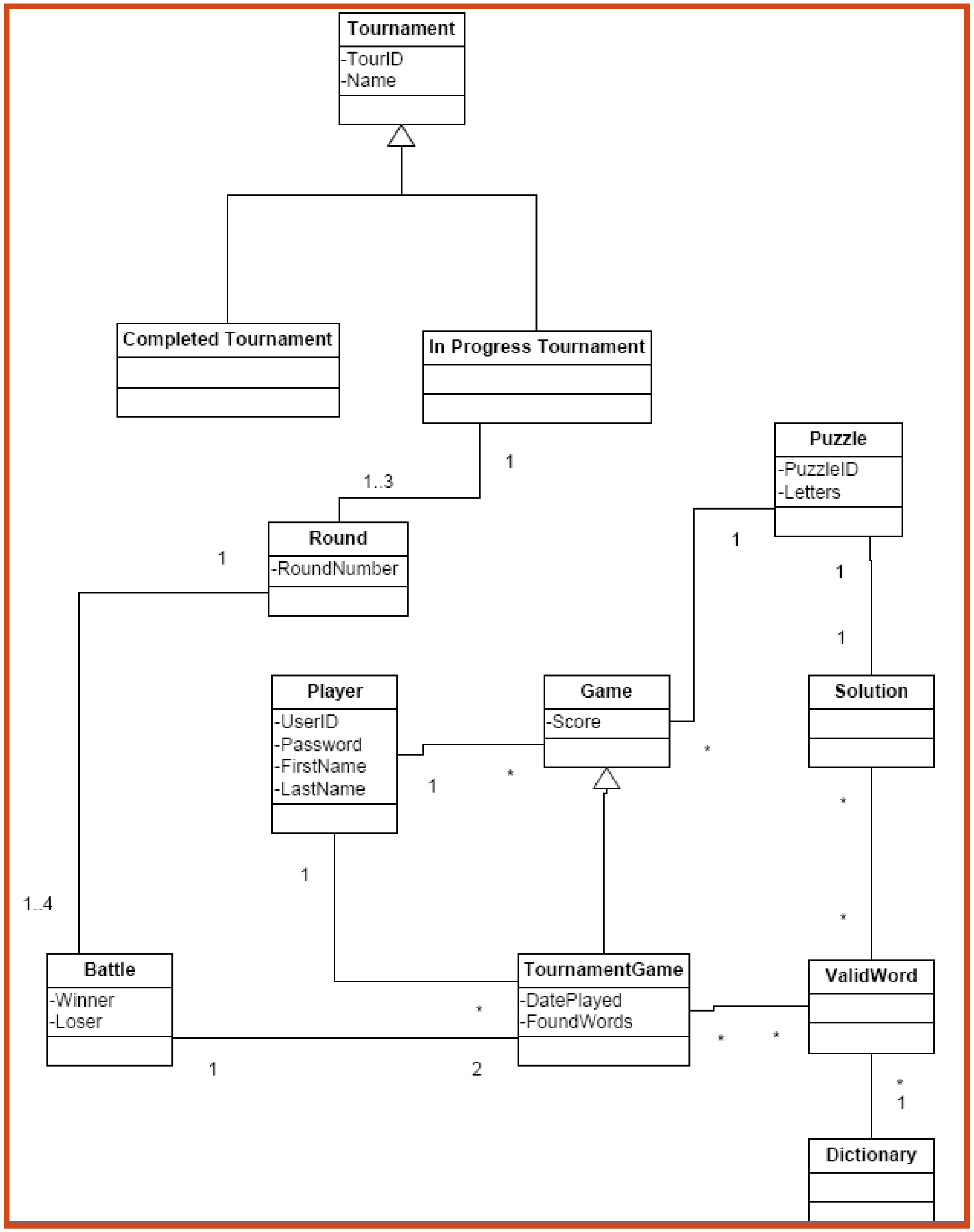
Working in pairs, determine the nouns and noun phrases that might, in the end, become potential classes. Add the attributes and associations. While making your list, choose good names for each of the potential classes
REMEMBER you only need classes in the domain model for things that need to have data stored about them!

Hari Prasad Pokhrel (hpokhrel24@gmail.com)

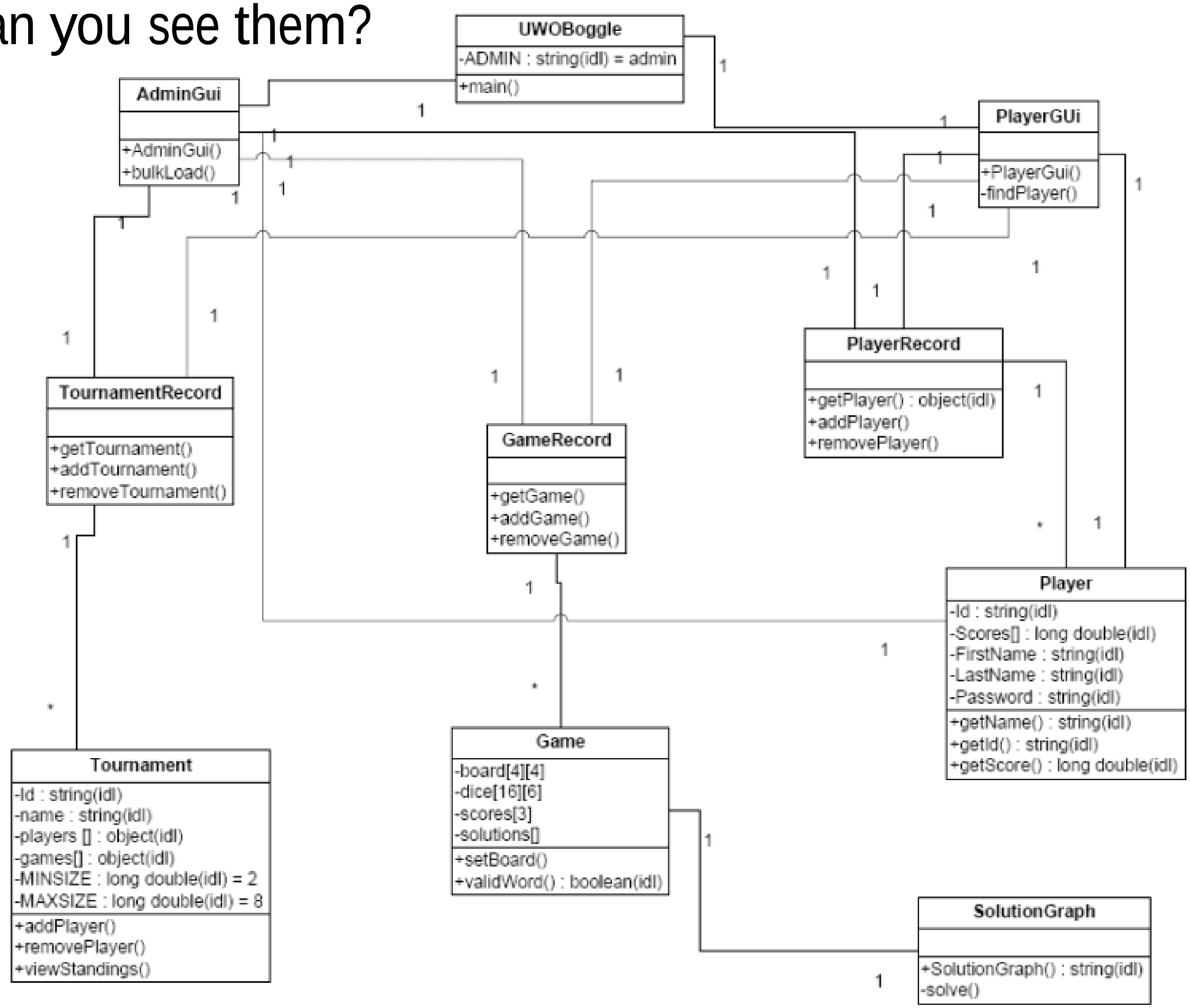
Your Answer

Laura’s Use Case Diagram for the Project →

Laura’s Class Diagram for the Project →

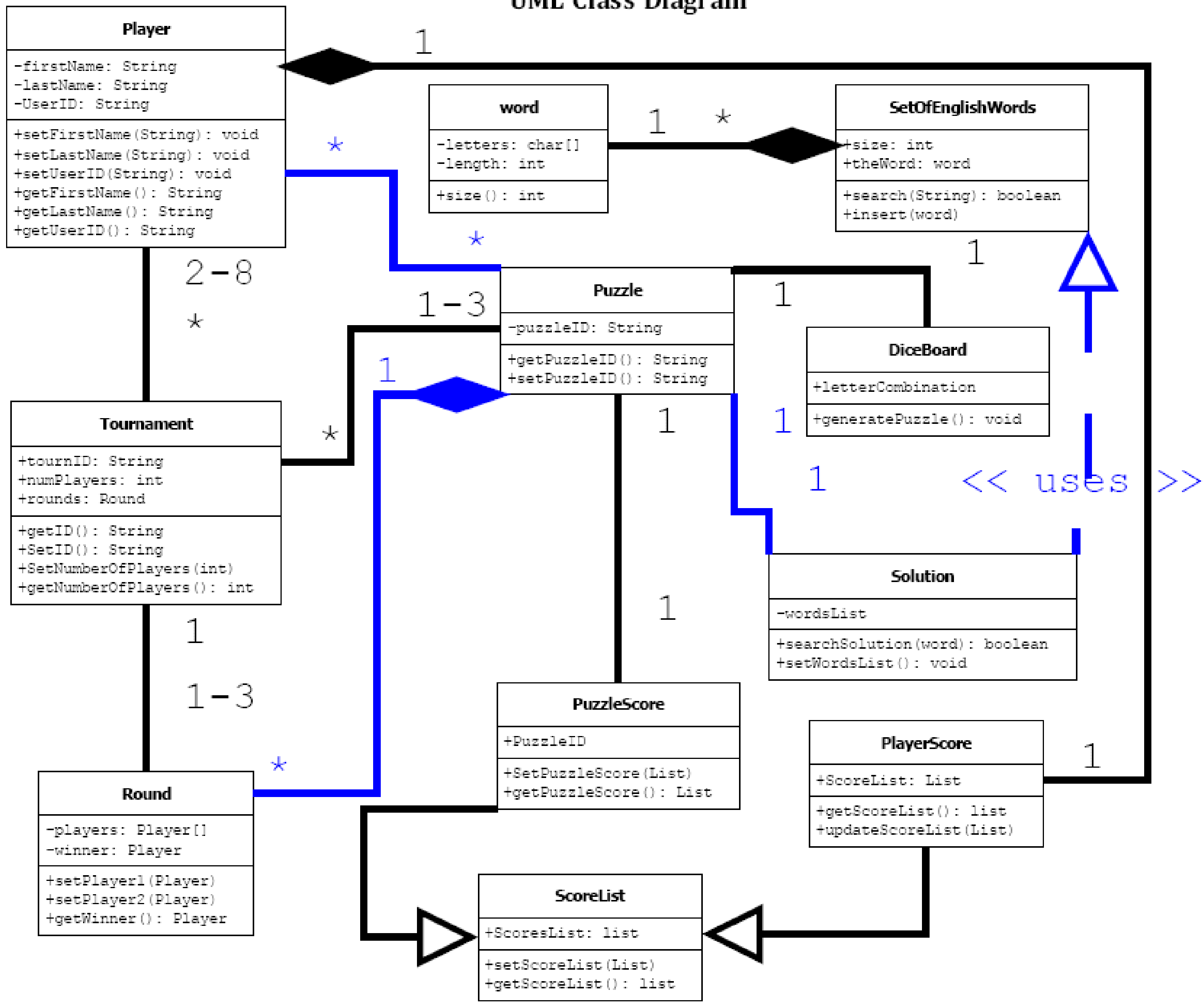


One group's class diagram for the project. This one has problems…Can you see them?

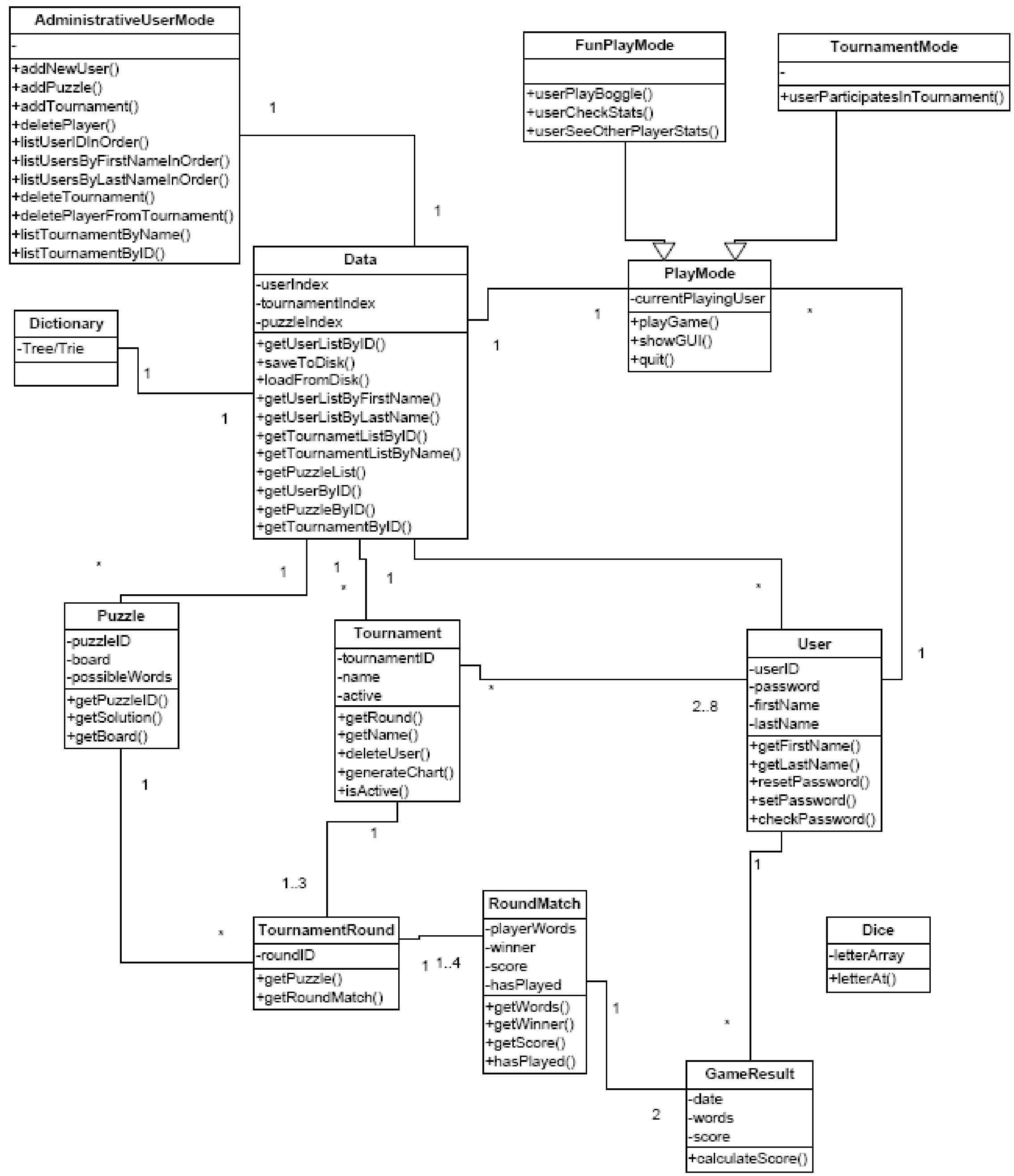


Hari Prasad Pokhrel (hpokhrel24@gmail.com)

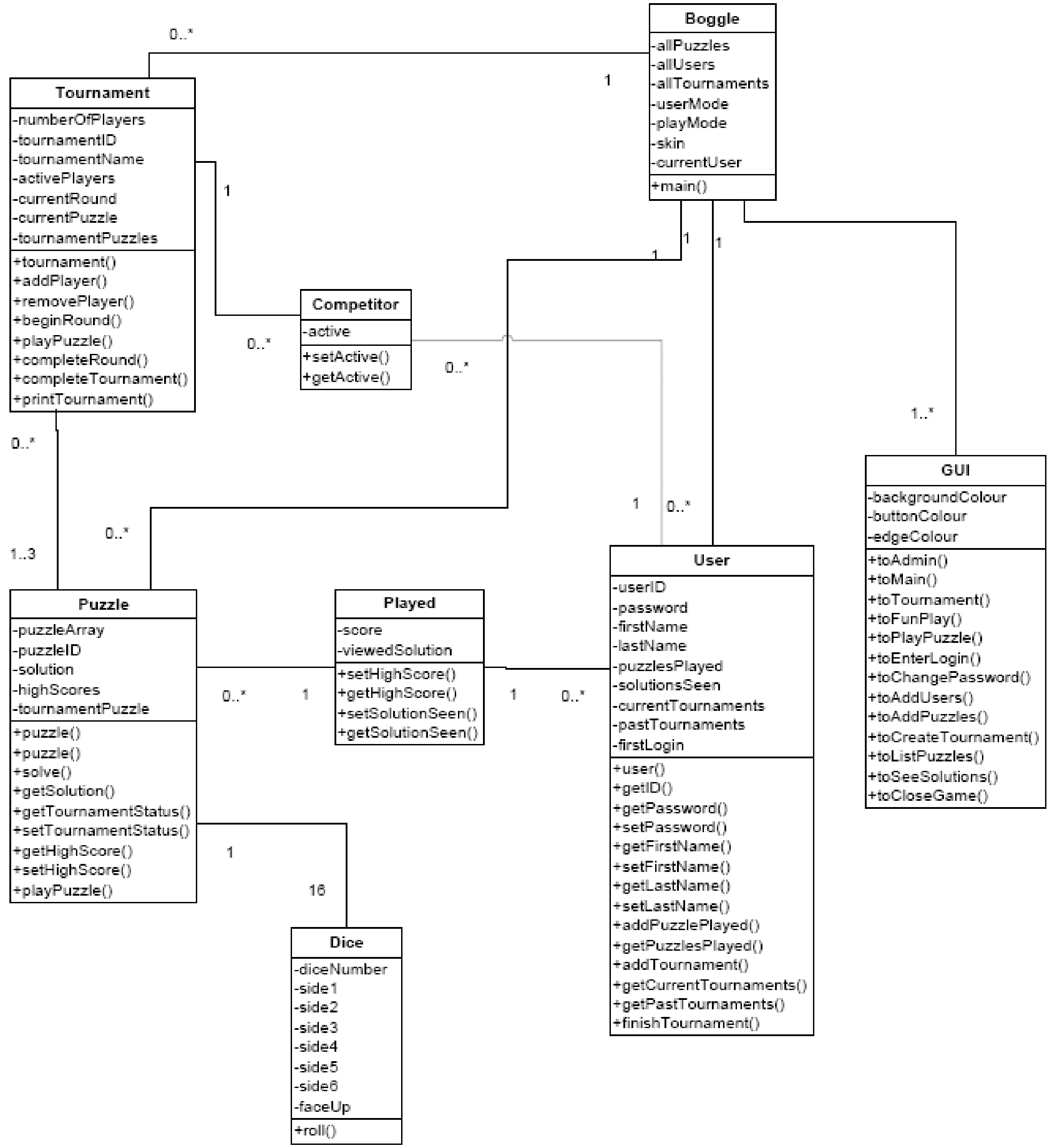
UML Class Diagram



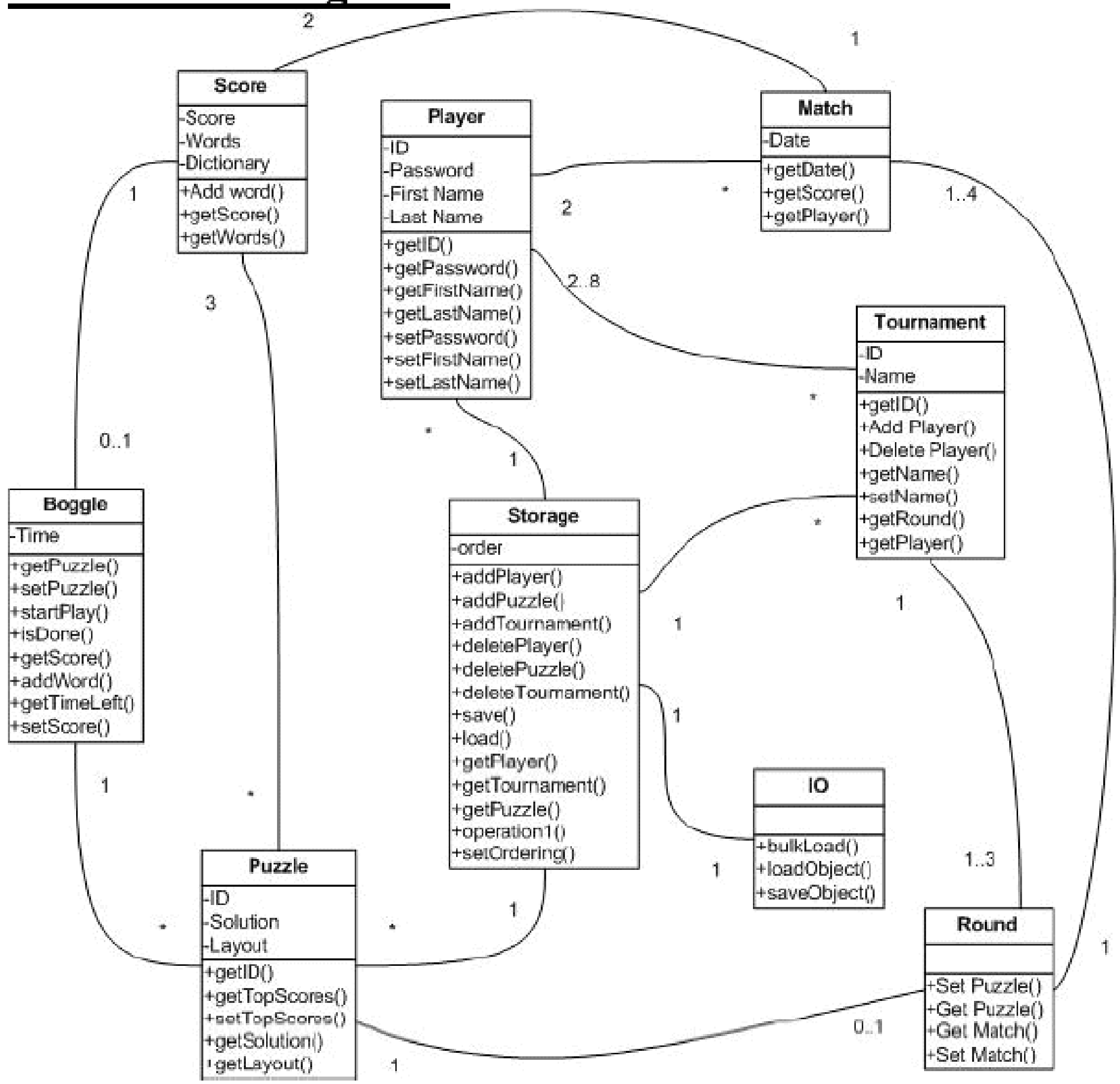
Hari Pr



Hari Prasad Pokhrel (hpokhrel


Boggle
-allPuzzles
-allUsers
-allTournaments
-userMode
-playMode
-skin
-currentUser
+main()
Tournament
-numberOfPlayers
-tournamentID
-tournamentName
-activePlayers
-currentRound
-currentPuzzle
-tournamentPuzzles
+tournament()
+addPlayer()
+removePlayer()
+beginRound()
+playPuzzle()
+completeRound()
+completeTournament()
+printTournament()
Competitor
-active
+setActive()
+getActive()
GUI
-backgroundColour
-buttonColour
-edgeColour
+toAdmin()
+toMain()
+toTournament()
+toFunPlay()
+toPlayPuzzle()
+toEnterLogin()
+toChangePassword()
+toAddUsers()
+toAddPuzzles()
+toCreateTournament()
+toListPuzzles()
+toSeeSolutions()
+toCloseGame()
Puzzle
-puzzleArray
-puzzleID
-solution
-highScores
-tournamentPuzzle
+puzzle()
+puzzle()
+solve()
+getSolution()
+getTournamentStatus()
+setTournamentStatus()
+getHighScore()
+setHighScore()
+playPuzzle()
Played
-score
-viewedSolution
+setHighScore()
+getHighScore()
+setSolutionSeen()
+getSolutionSeen()
User
-userID
-password
-firstName
-lastName
-puzzlesPlayed
-solutionsSeen
-currentTournaments
-pastTournaments
-firstLogin
+user()
+getID()
+getPassword()
+setPassword()
+getFirstName()
+setFirstName()
+getLastName()
+setLastName()
+addPuzzlePlayed()
+getPuzzlesPlayed()
+addTournament()
+getCurrentTournaments()
+getPastTournaments()
+finishTournament()
Dice
-diceNumber
-side1
-side2
-side3
-side4
-side5
-side6
-faceUp
+roll()
0..*
1
1
0..*
0..*
1
1
1
1..*
0..*
1..3
0..*
1
0..*
0..*
1
1
0..*
1
16

UML Class Diagram:



Hari Prasad Pokhrel (hpokhrel24@gmail.com)

Question

- Identify any generalizations or interfaces. This may lead you to add or delete classes, associations and attributes. Modify your class diagrams accordingly.

Allocating responsibilities to classes

- A *responsibility* is something that the system is required to do.
  - Each functional requirement must be attributed to one of the classes
    - All the responsibilities of a given class should be *clearly related*.
    - If a class has too many responsibilities, consider *splitting* it into distinct classes
    - If a class has no responsibilities attached to it, then it is probably *useless*
    - When a responsibility cannot be attributed to any of the existing classes, then a *new class* should be created

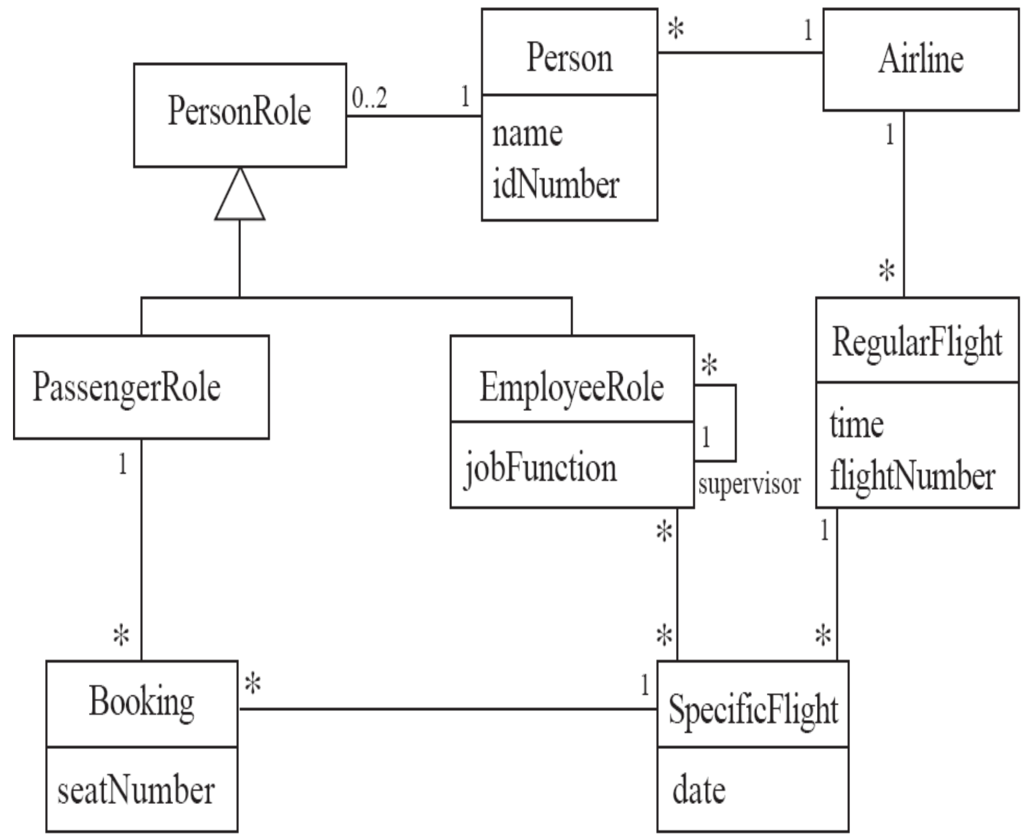
  - To determine responsibilities
    - Perform use case analysis
    - Look for verbs and nouns describing *actions* in the system description

Categories of responsibilities

- Setting and getting the values of attributes
- Creating and initializing new instances
- Loading to and saving from persistent storage
- Destroying instances
- Adding and deleting links of associations
- Copying, converting, transforming, transmitting or outputting
- Computing numerical results
- Navigating and searching
- Other specialized work

An example (responsibilities)

- Creating a new regular flight
- Searching for a flight
- Modifying attributes of a flight
- Creating a specific flight
- Booking a passenger
- Canceling a booking



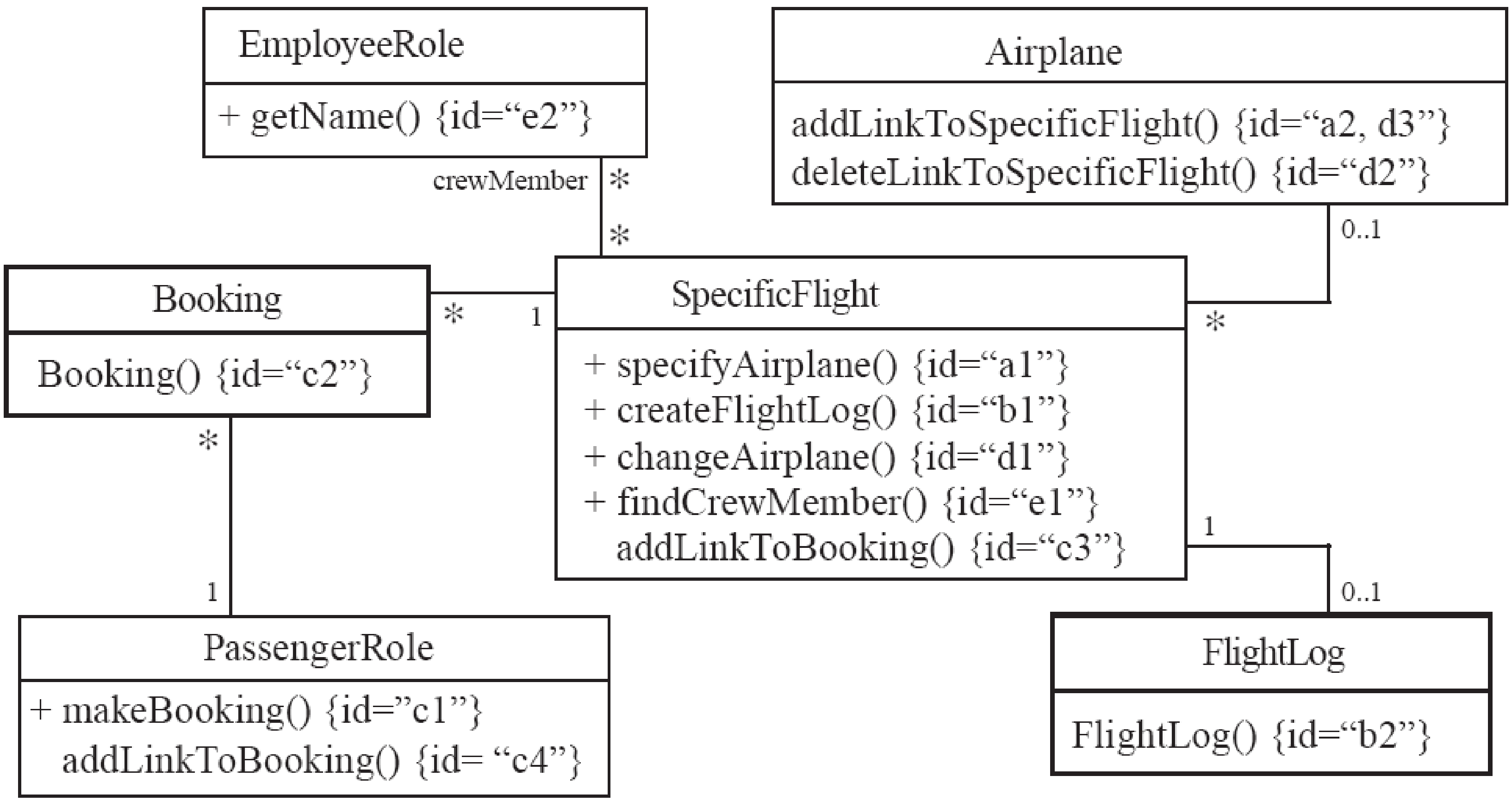
Prototyping a class diagram on paper

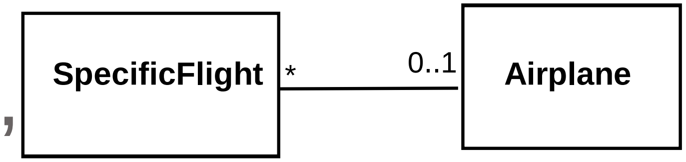
- As you identify classes, you write their names on small cards
- As you identify attributes and responsibilities, you list them on the cards
  - If you cannot fit all the responsibilities on one card:
    - this suggests you should split the class into two related classes.
- Move the cards around on a whiteboard to arrange them into a class diagram.
- Draw lines among the cards to represent associations and generalizations.

Identifying operations

- Operations are needed to realize the responsibilities of each class
  - There may be several operations per responsibility
  - The main operations that implement a responsibility are normally declared public
  - Other methods that collaborate to perform the responsibility must be as private as possible

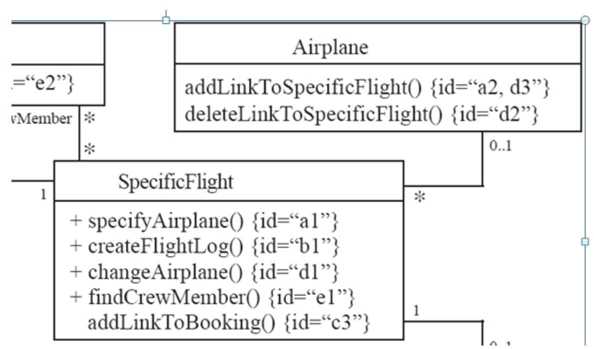
An example (class collaboration)



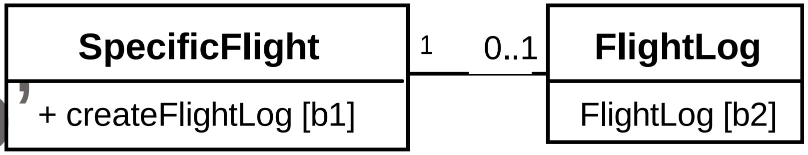


Class collaboration ‘a’

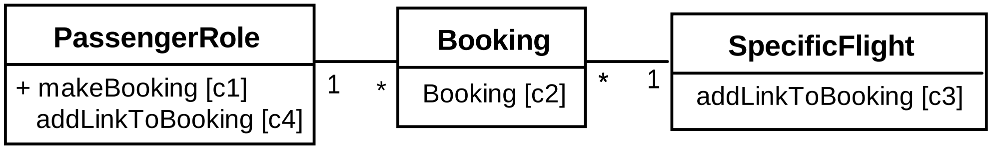
- *Making a bi-directional link between two existing objects*;
- e.g. adding a link between an instance of SpecificFlight and an instance of Airplane.
-
- 1. (public) The instance of SpecificFlight
  - **makes a one-directional link to the instance of** Airplane
  - **then calls operation 2.**
- 2. (non-public) The instance of Airplane
  - **makes a one-directional link back to the instance of** SpecificFlight



Class collaboration ‘b’



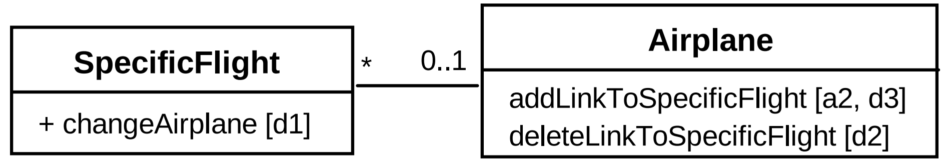
- *Creating an object and linking it to an existing object*
- e.g. creating a `FlightLog`, and linking it to a `SpecificFlight`.
-
- 1. (public) The instance of `SpecificFlight`
    - **calls the constructor of** `FlightLog` **(operation 2)**
    - **then makes a one-directional link to the new instance of** `FlightLog`**.**
- 2. (non-public) Class `FlightLog`’s constructor
    - **makes a one-directional link back to the instance of** `SpecificFlight`**.**



Class collaboration 'c'

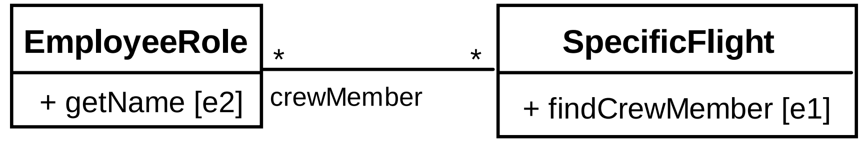
- *Creating an association class, given two existing objects*
- e.g. creating an instance of Booking, which will link a SpecificFlight to a PassengerRole.
- 1. (public) The instance of PassengerRole
  - calls the constructor of **Booking** (operation 2).
- 2. (non-public) Class Booking's constructor, among its other actions
  - makes a one-directional link back to the instance of **PassengerRole**
  - makes a one-directional link to the instance of **SpecificFlight**
  - calls operations 3 and 4.
- 3. (non-public) The instance of SpecificFlight
  - makes a one-directional link to the instance of **Booking**.
- 4. (non-public) The instance of PassengerRole
  - makes a one-directional link to the instance of **Booking**.

Class collaboration ‘d’

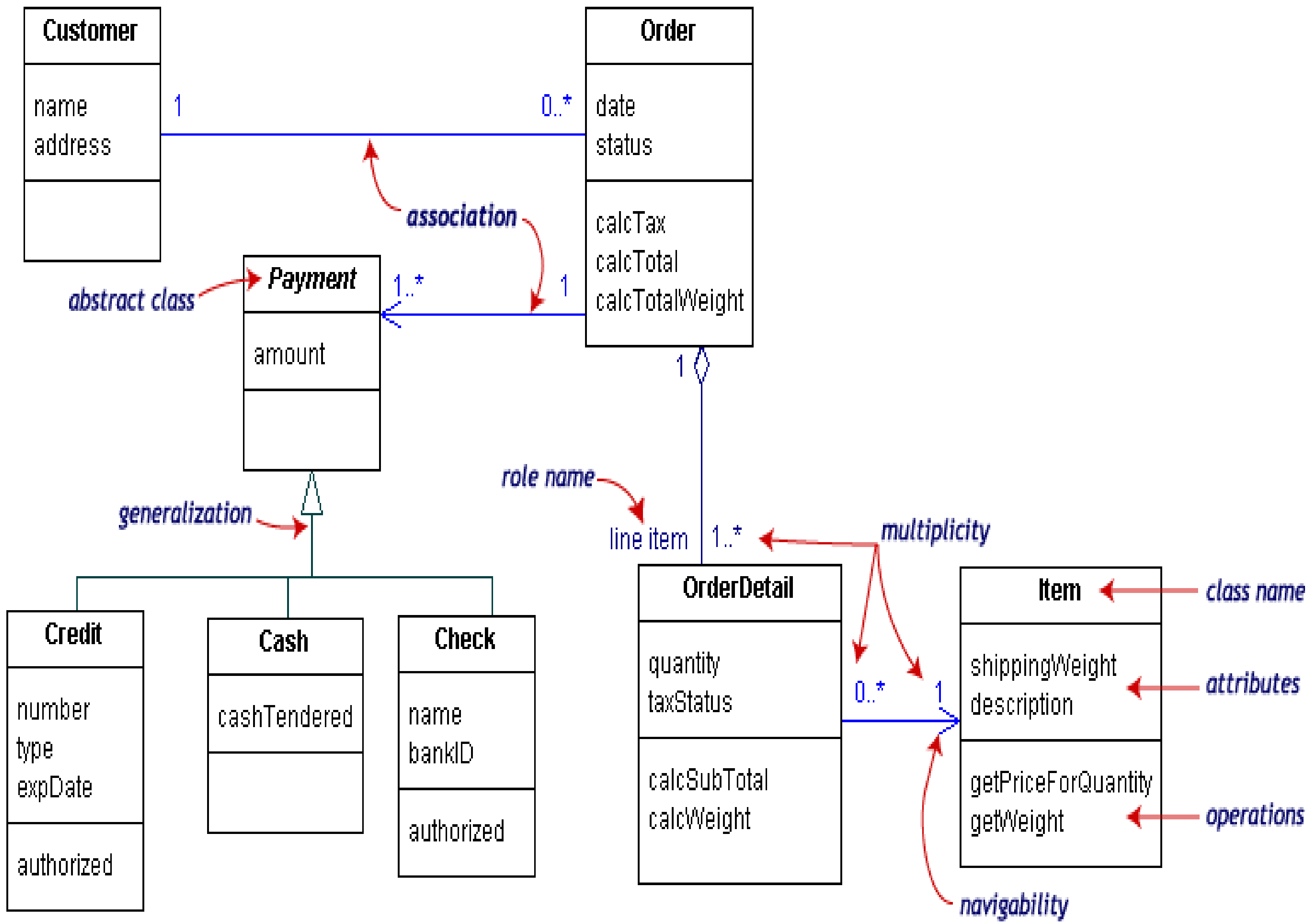


- *Changing the destination of a link*
- e.g. changing the `Airplane` of to a `SpecificFlight`, from `airplane1` to `airplane2`
- 1. (public) The instance of `SpecificFlight`
  - deletes the link to **`airplane1`**
  - makes a one-directional link to **`airplane2`**
  - calls operation 2
  - then calls operation 3.
- 2. (non-public) `airplane1`
  - deletes its one-directional link to the instance of **`SpecificFlight`**.
- 3. (non-public) `airplane2`
  - makes a one-directional link to the instance of **`SpecificFlight`**.

Class collaboration ‘e’



- *Searching for an associated instance*
- e.g. searching for a crew member associated with a SpecificFlight that has a certain name.
-
- 1. (public) The instance of SpecificFlight
  - creates an Iterator over all the crewMember links of the SpecificFlight\
  - for each of them call operation 2, until it finds a match.
- 2. (may be public) The instance of EmployeeRole returns its name.



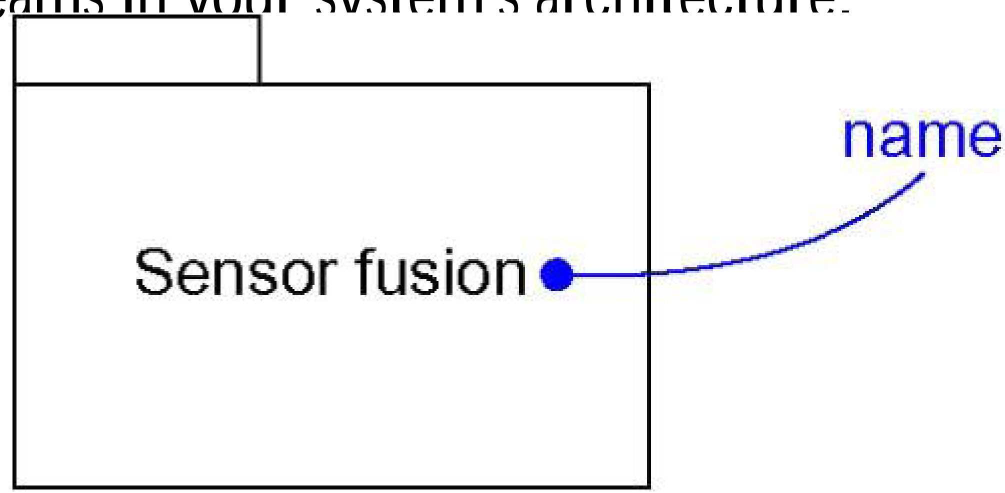
Hari Prasad Pokhrel (hpokhrel24@gmail.com)

Packages in Class Diagrams

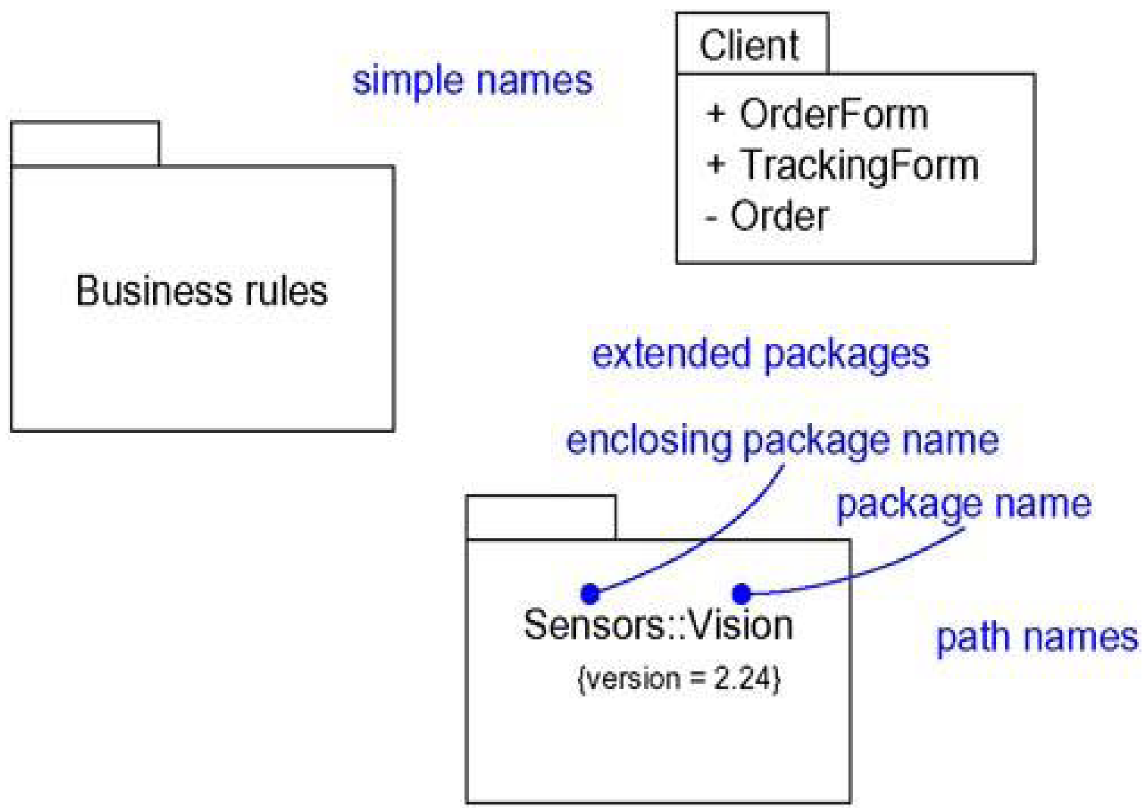
- Complex Class Diagrams are Abstracted
- Packages Contain Multiple Classes and are Associated and Linked to One Another
  - Dependency Arrow is Dashed
  - Indicates that One Package Depends on Another
  - Means that Changes in Destination (Dependee - Arrow Head) Can Possible Force Changes in the Source (Dependent – Arrow Tail) \
- Supports Rudimentary SW Architecture Concepts
- However, no Checking / Enforcement of Dependencies in Subsequent Diagrams

Package

- A package is a general purpose mechanism for organizing elements into groups.
- Packages help you organize the elements in your models so that you can more easily understand them.
- Packages also let you control access to their contents so that you can control the seams in your system's architecture.



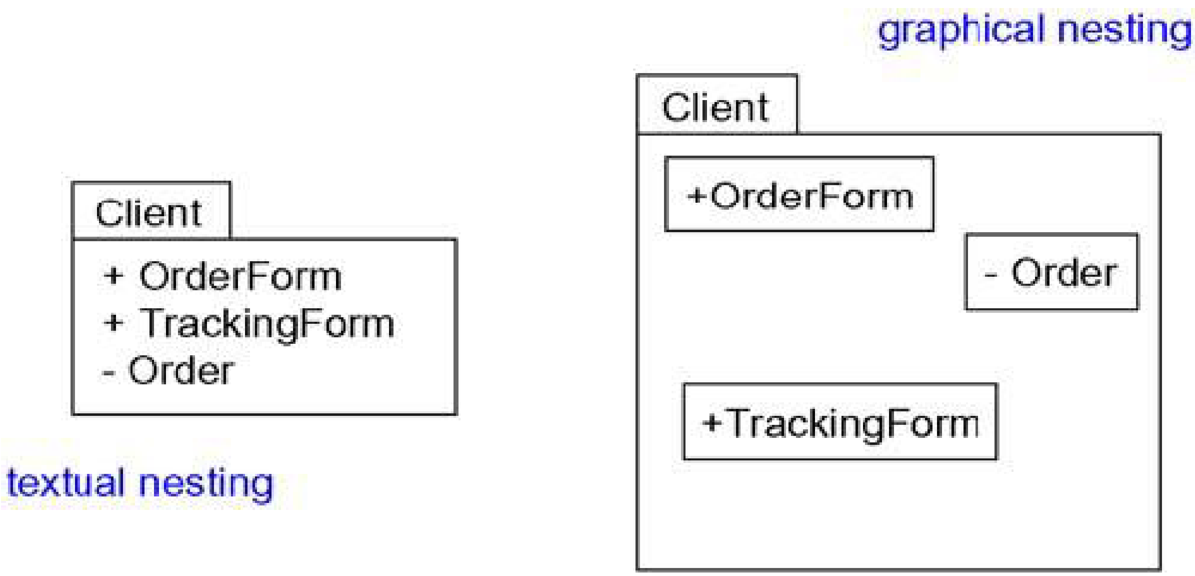
Simple and Extended Package



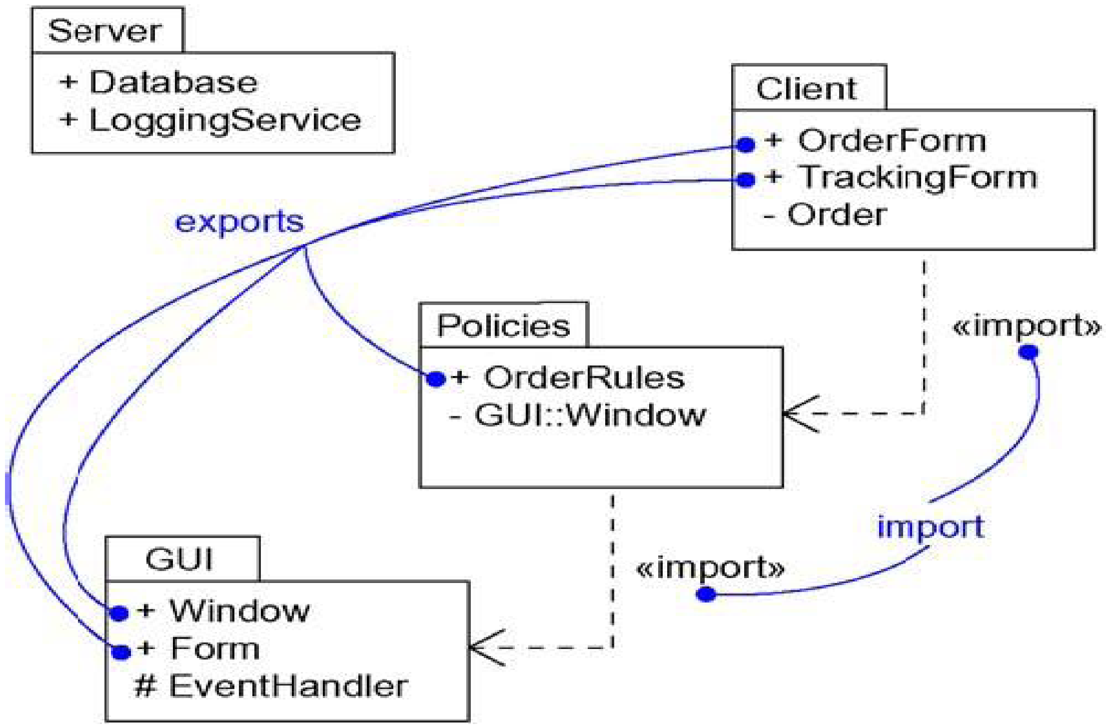
Package

- A package may own other elements, including classes, interfaces, components, nodes, collaborations, use cases, diagrams, and even other packages.
- A package forms a namespace, which means that elements of the same kind must be named uniquely within the context of its enclosing package.
  - For example, you can't have two classes named Queue owned by the same package, but you can have a class named Queue in package P1 and another (and different) class named Queue in package P2.
  - The classes P1::Queue and P2::Queue are, in fact, different classes and can be distinguished by their path names.
  - Different kinds of elements may have the same name.
- Packages may own other packages

Packages may own other packages



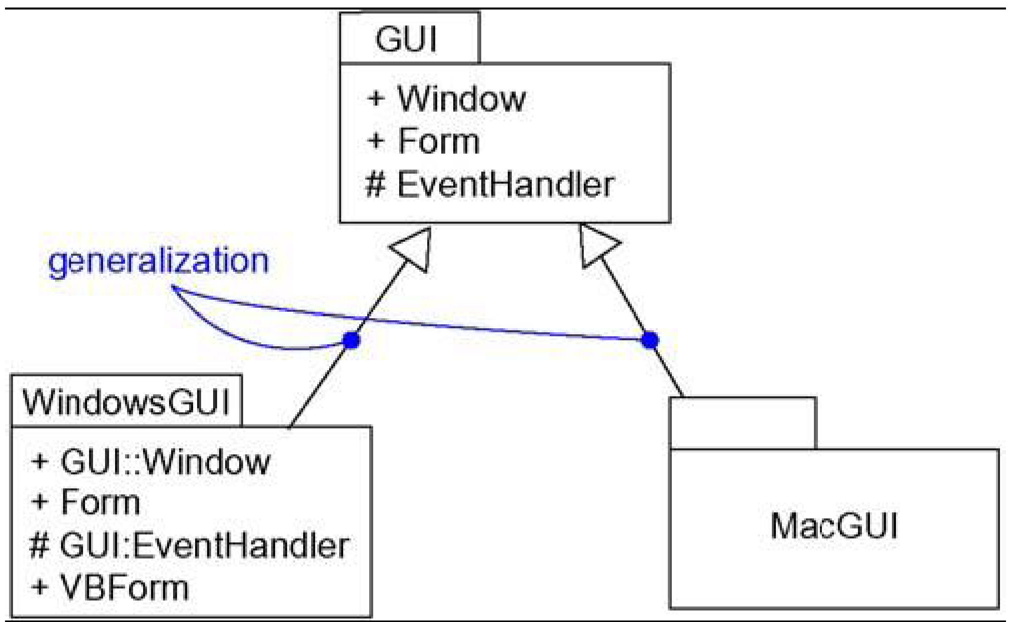
Importing and Exporting



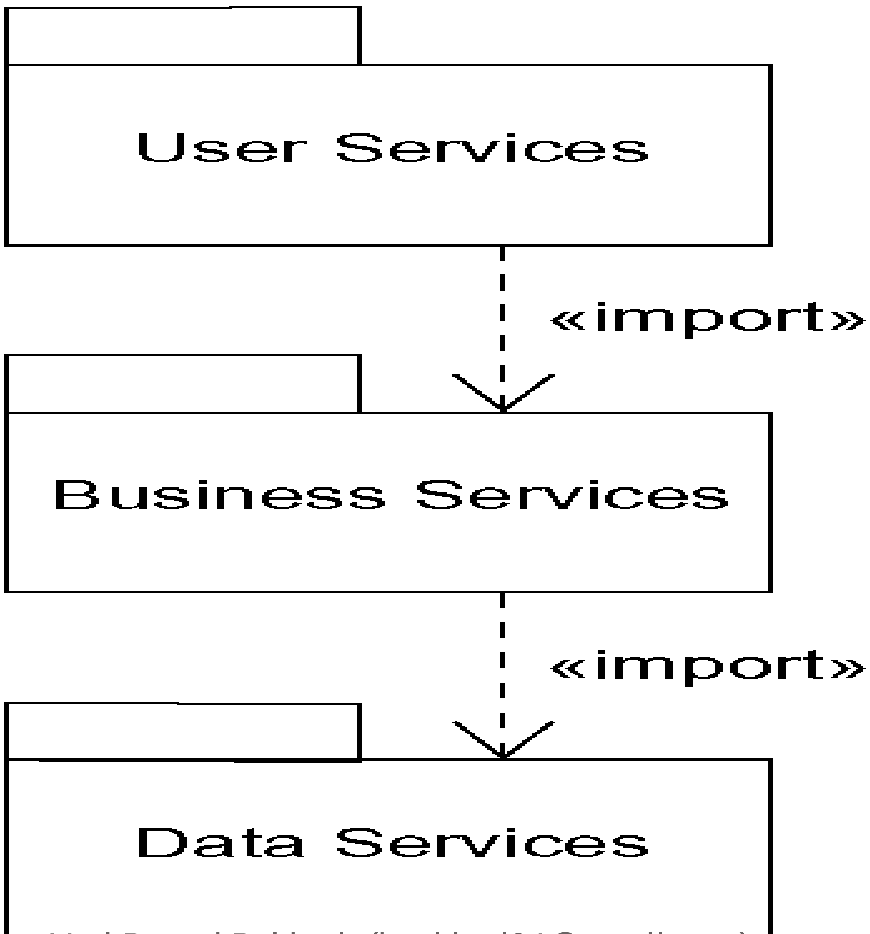
The public parts of a package are called its exports.
The package *GUI* exports two classes, *Window* and *Form*.
EventHandler is not exported by GUI;
EventHandler is a protected part of the package.

In this example, *Policies* explicitly *imports* the package GUI. *GUI::Window* and *GUI::Form* are therefore made visible to the contents of the package Policies. However, *GUI::EventHandler* is not visible because it is protected.
Because the package *Server* doesn't import GUI, the contents of *Server* don't have permission to access any of the contents of GUI.
Similarly, the contents of GUI don't have permission to access any of the contents of Server.

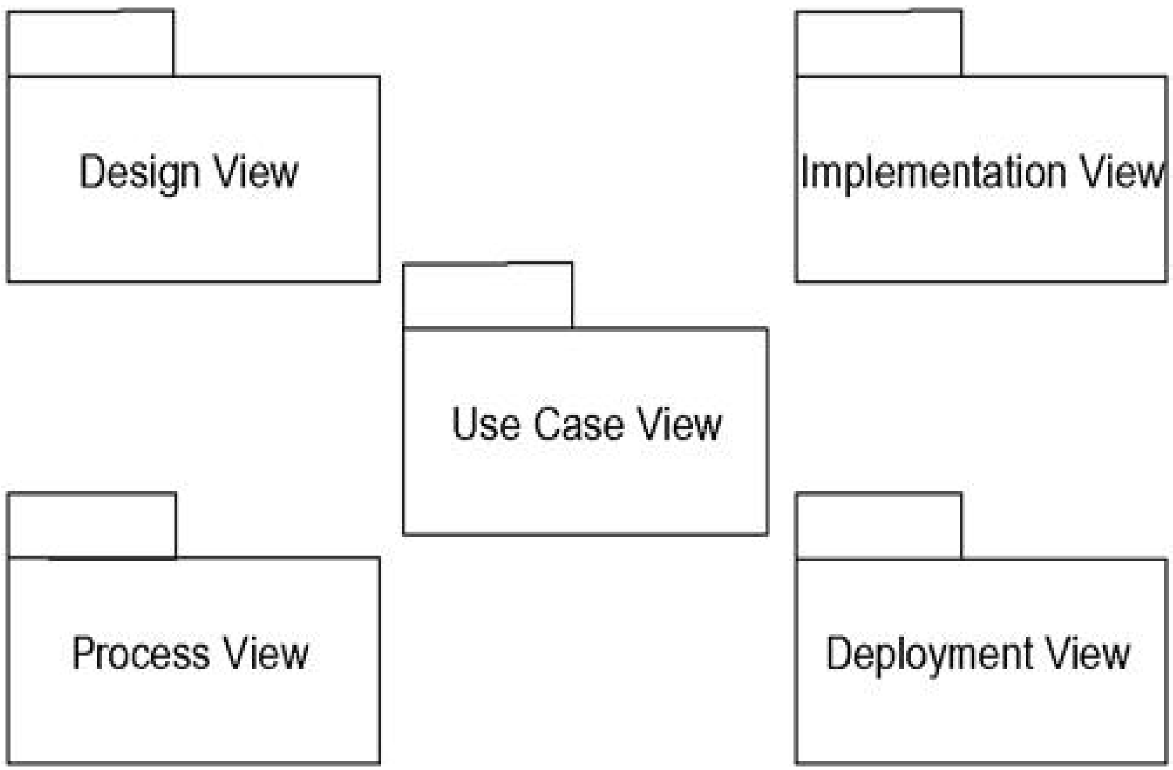
Generalization Among Packages



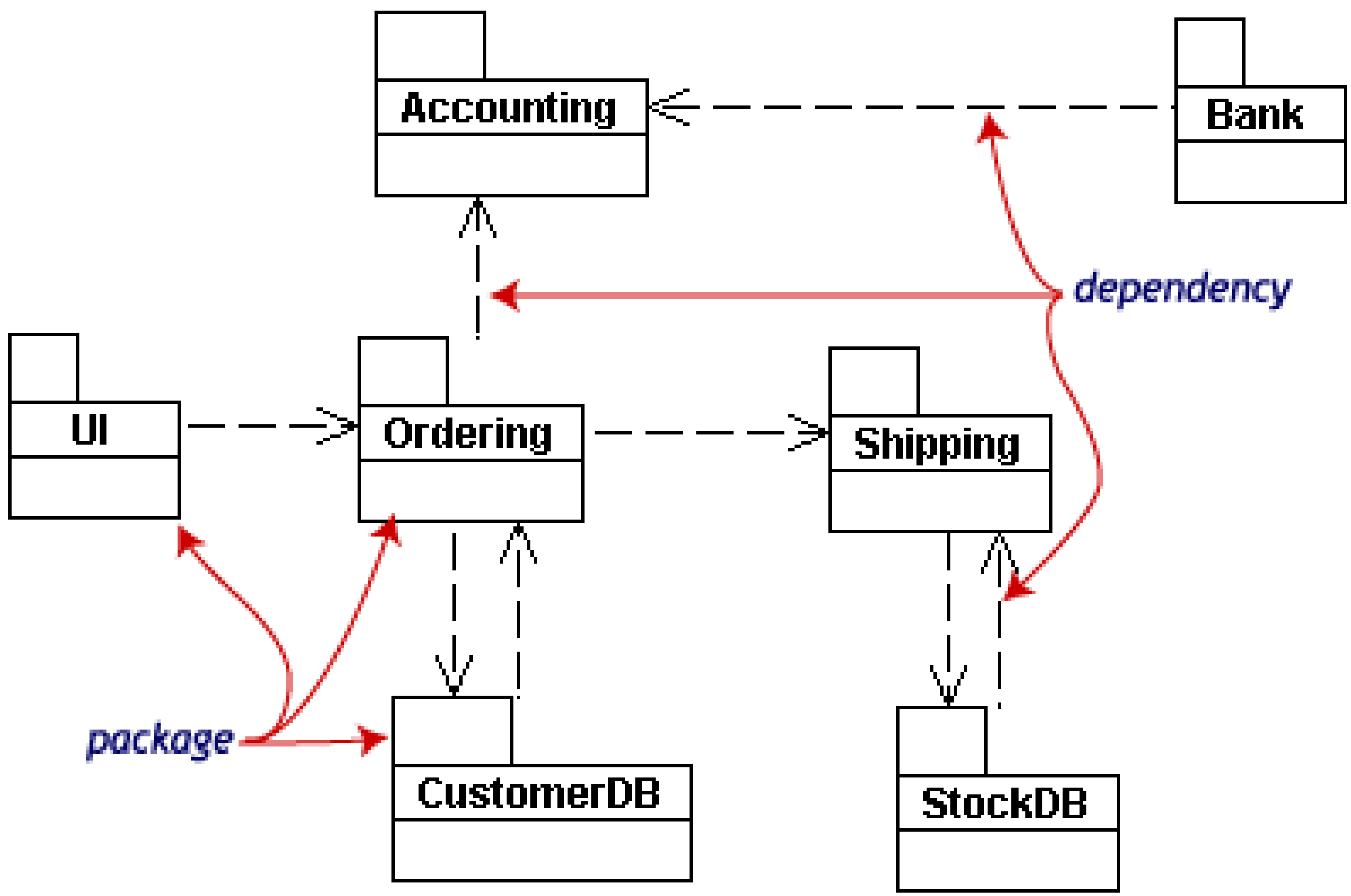
Modeling Groups of Elements



Modeling Architectural Views



Example Package

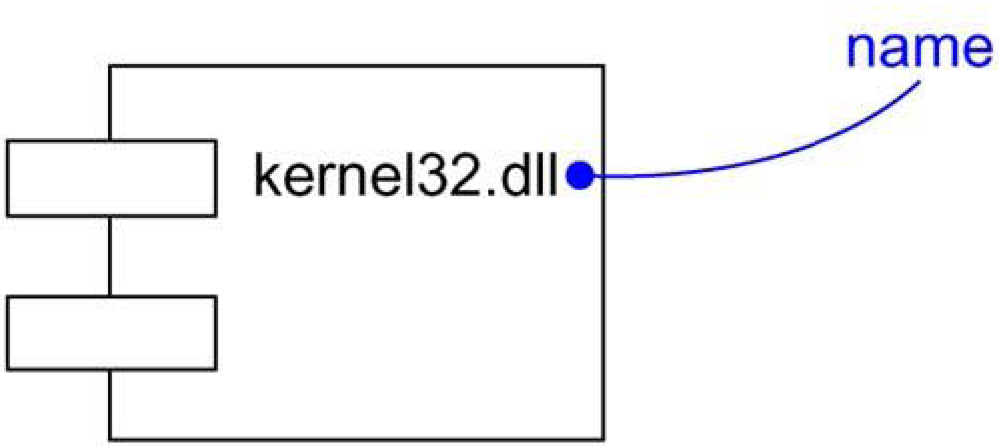


Static: Component Diagram

- Component Diagram: High-Level Interaction and Dependencies Among Software Components
- Captures the Physical Structure of the Implementation
- Built As Part of Architectural Specification
- Purposes:
  - Organize Source Code
  - Construct an Executable Release
  - Specify a Physical Database
- Main Concepts: Component, Interface, Dependency, Realization
- Developed by Architects and Programmers

Component

- A *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.

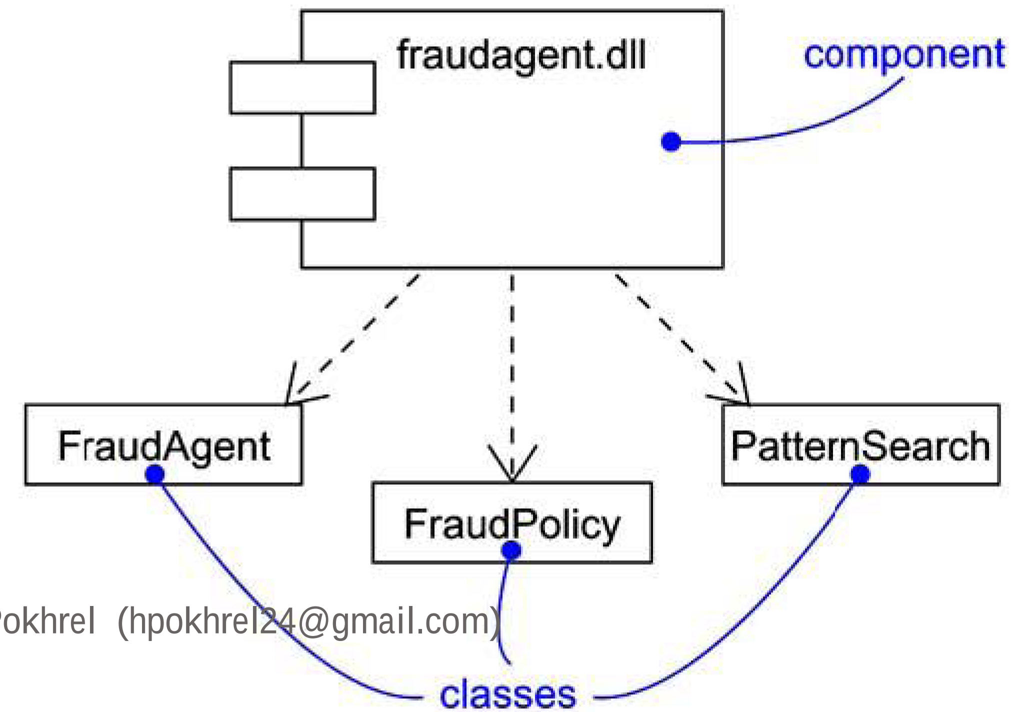


Components and Classes

- 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; both may have instances; both may be participants in interactions. 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 other 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.

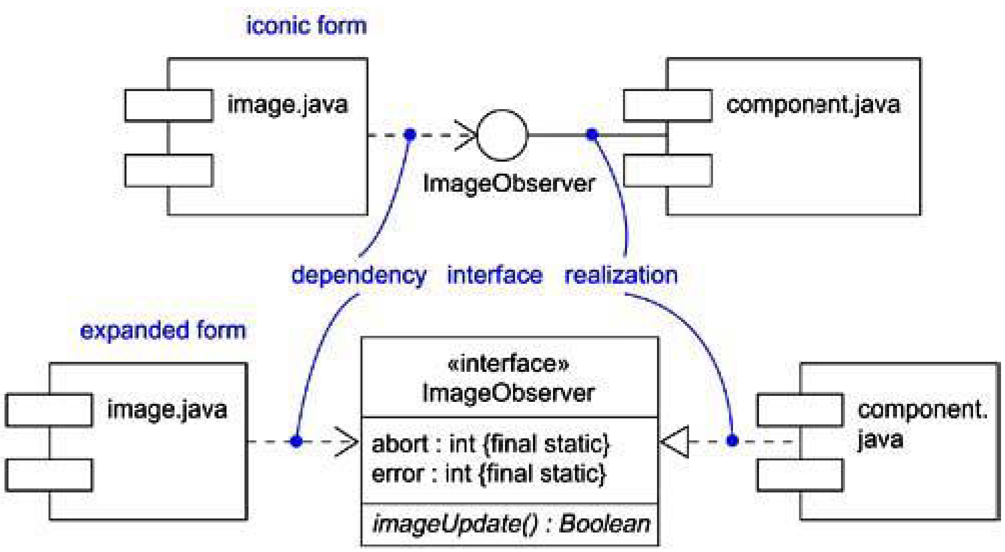
Components and Classes

- A component is the physical implementation of a set of other logical elements, such as classes and collaborations
- shows, the relationship between a component and the classes it implements can be shown explicitly by using a dependency relationship.



Components and Interfaces

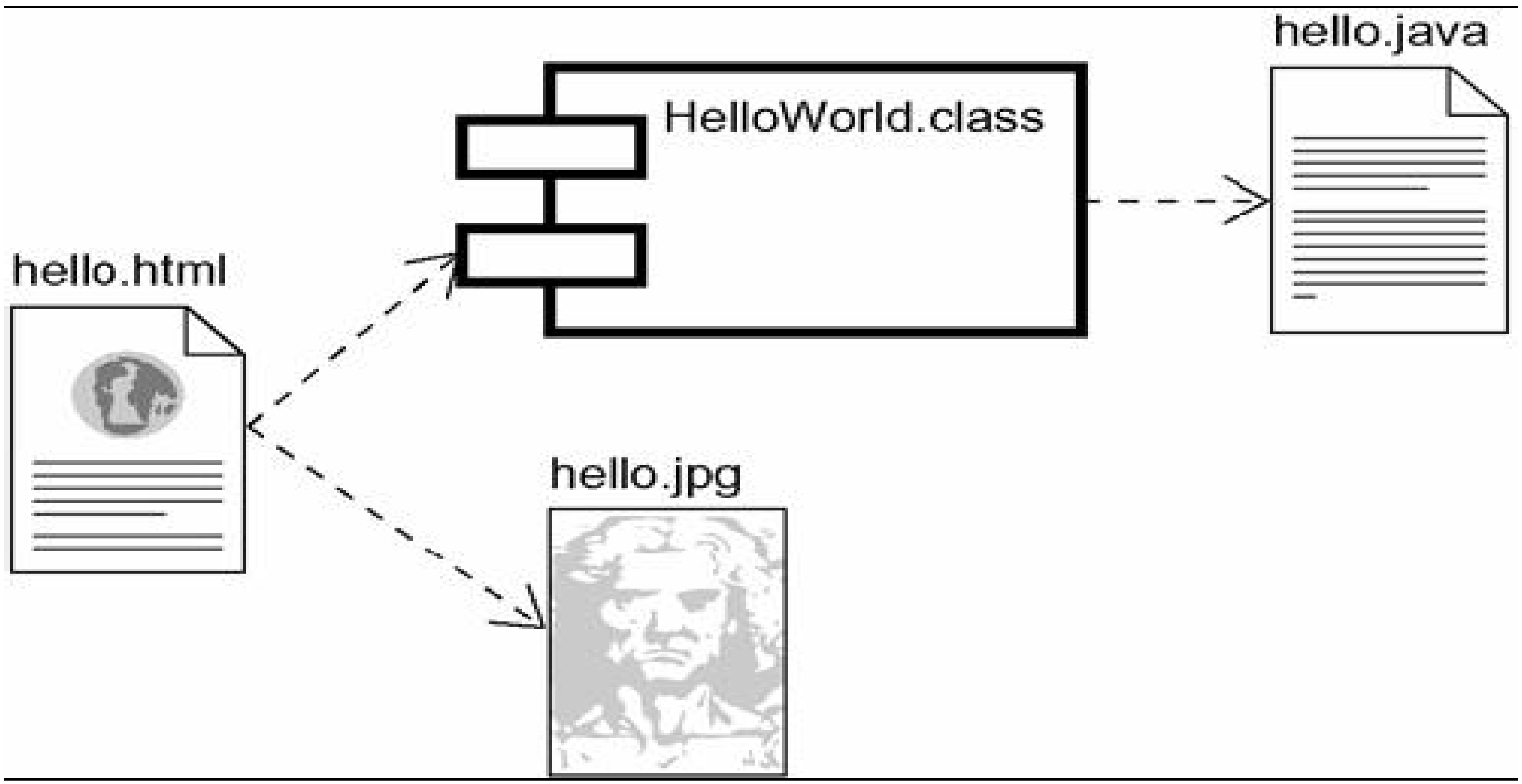
An interface is a collection of operations that are used to specify a service of a class or a component. The relationship between component and interface is important.



All the most common component-based operating system facilities (such as COM+, CORBA, and Enterprise Java Beans) use interfaces as the glue that binds components together.

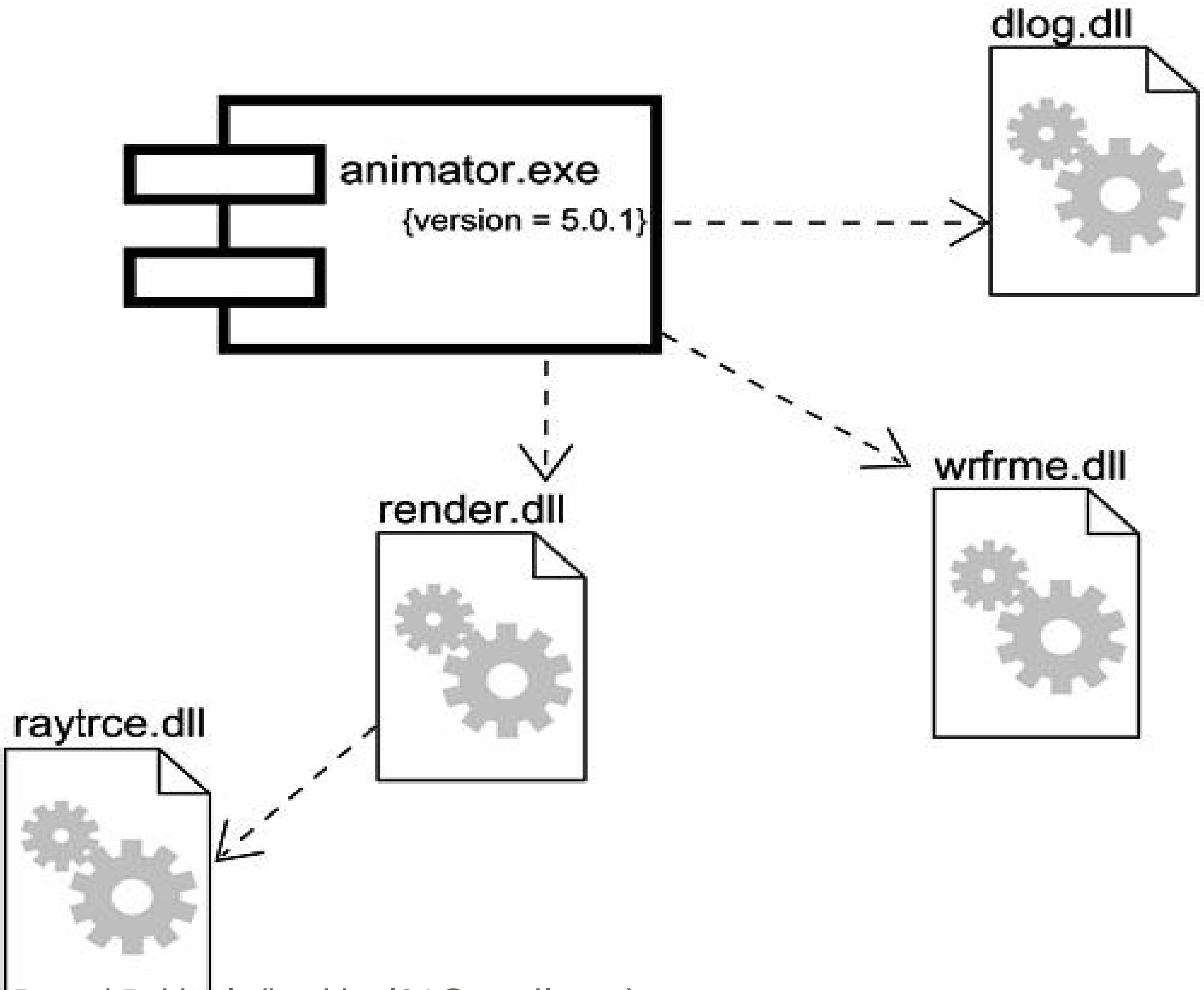
Kinds of Components

- *Deployment components.*
  - These are the components necessary and sufficient to form an executable system, such as dynamic libraries (DLLs) and executables (EXEs).
- *Work product components.*
  - These components are essentially the residue of the development process, consisting of things such as source code files and data files from which deployment components are created. These components do not directly participate in an executable system but are the work products of development that are used to create the executable system.
- *Execution components.*
  - These components are created as a consequence of an executing system, such as a COM+ object, which is instantiated from a DLL.


hello.java
HelloWorld.class
hello.html
hello.jpg

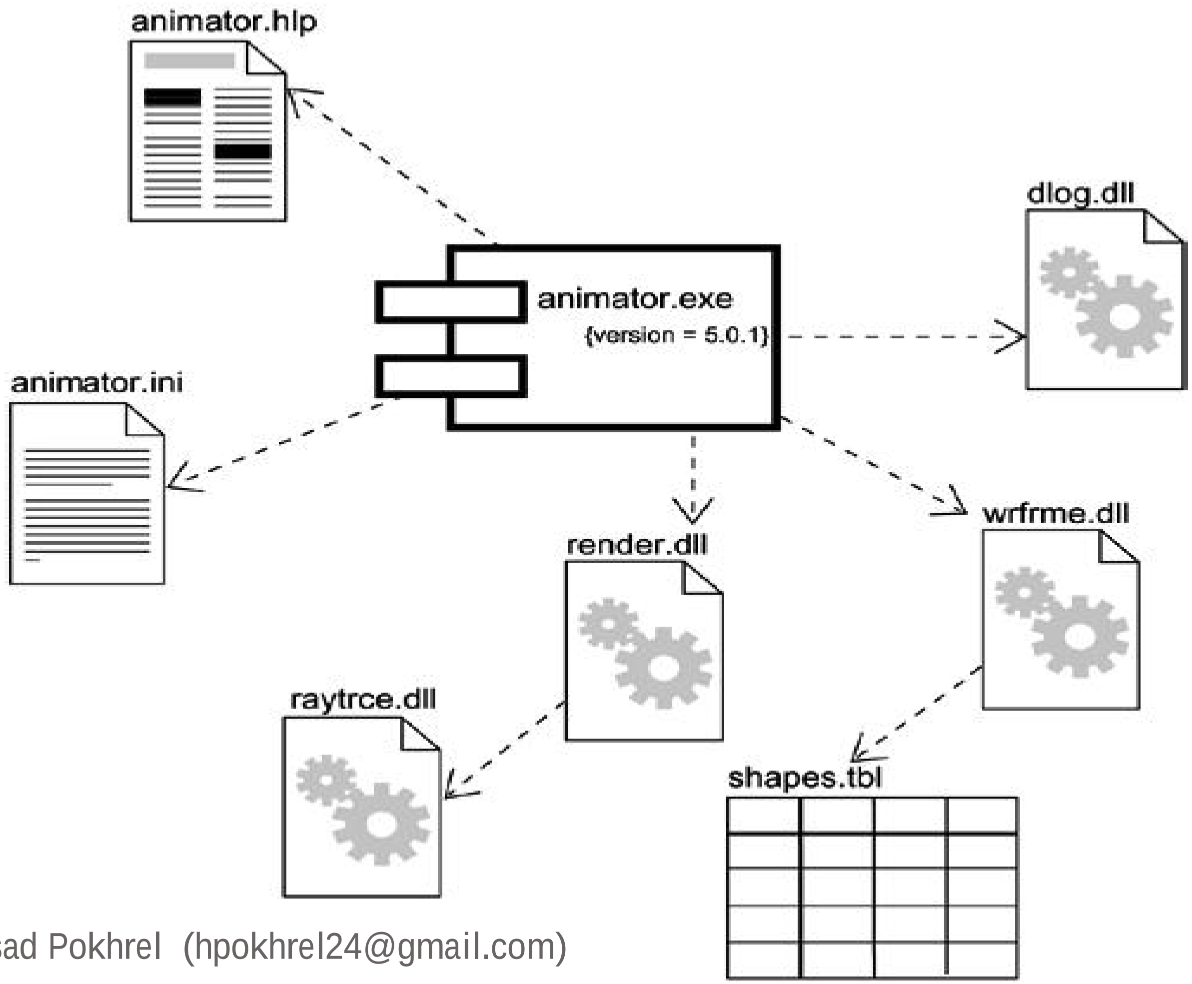
Modeling Executables and Libraries

- To model executables and libraries,
- Identify the partitioning of your physical system.
- Model any executables and libraries as components, using the appropriate standard elements
- If it's important for you to manage the seams in your system, 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.



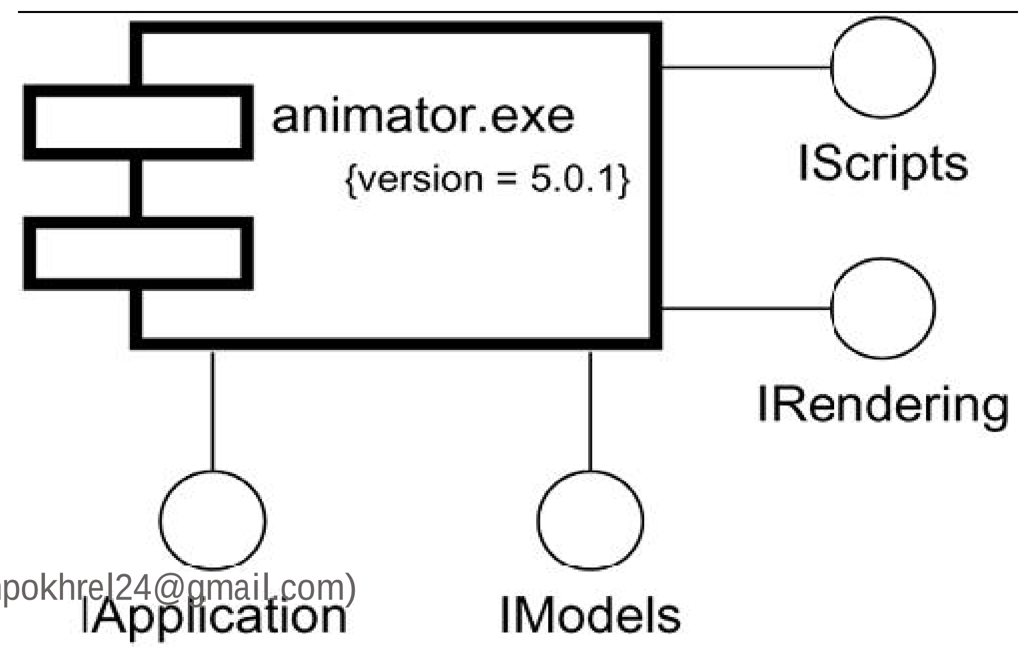
Hari Prasad Pokhrel (hpokhrel24@gmail.com)

Modeling Tables, Files, and Documents

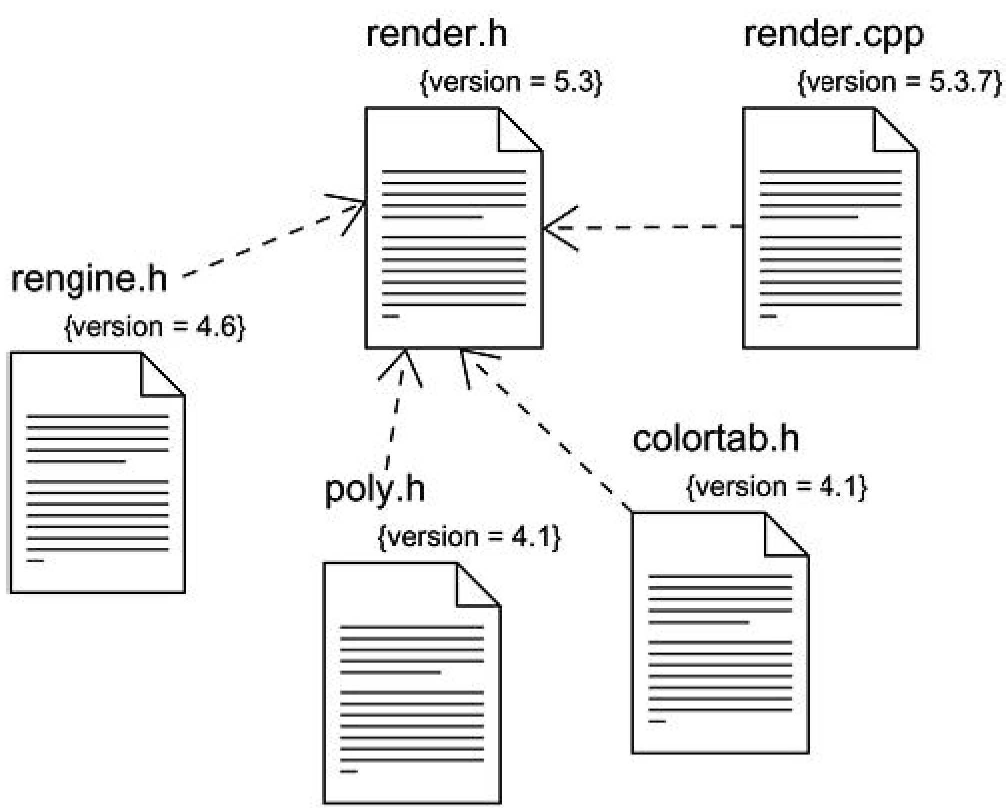


Modeling an API

- An API is essentially an interface that is realized by one or more components. As a developer, you'll really care only about the interface itself; which component realizes an interface's operations is not relevant as long as *some component realizes it.*

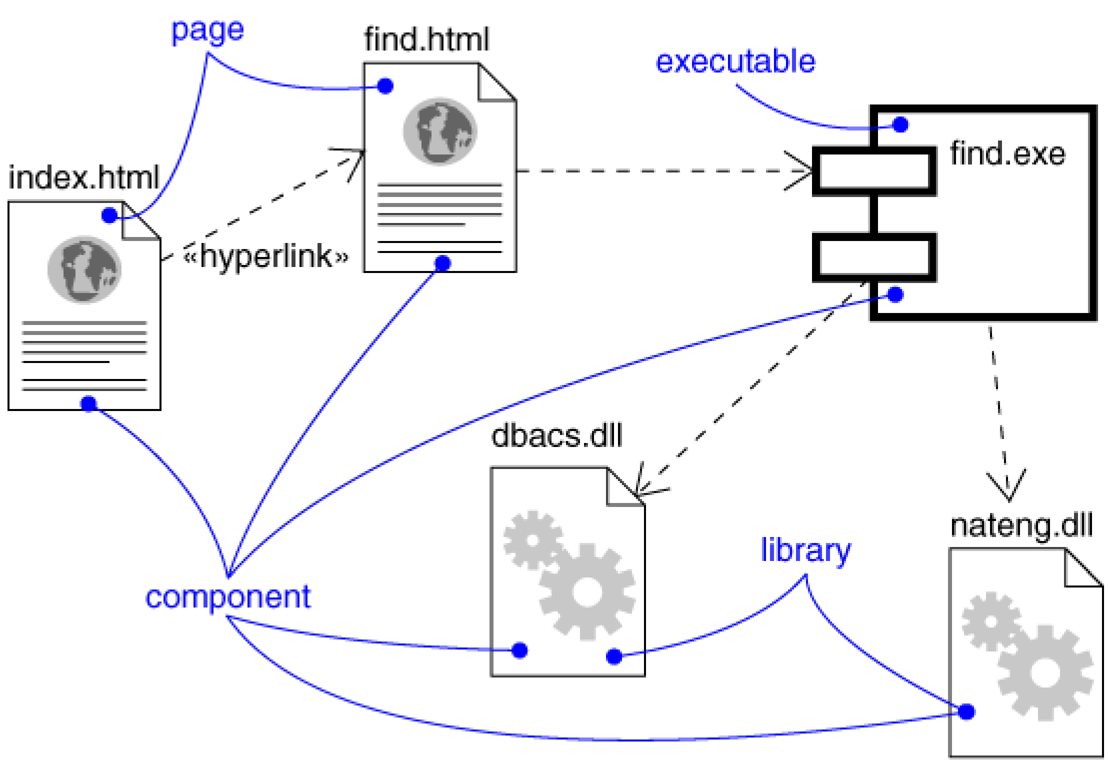


Modeling Source Code



Component Diagram

- Captures the Physical Structure of the Implementation

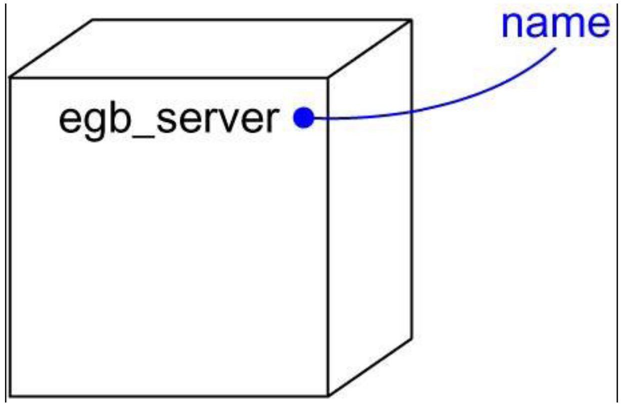


Static: Deployment Diagram

- Deployment Diagram: Focus on the Placement and Configuration of Components at Runtime
- Captures the Topology of a System's Hardware
- Built As Part of Architectural Specification
- Purposes:
  - Specify the Distribution of Components
  - Identify Performance Bottlenecks
- Main Concepts: Node, Component, Dependency, Location
- Developed by Architects, Networking Engineers, and System Engineers
- Focus is on physical aspects of a system.

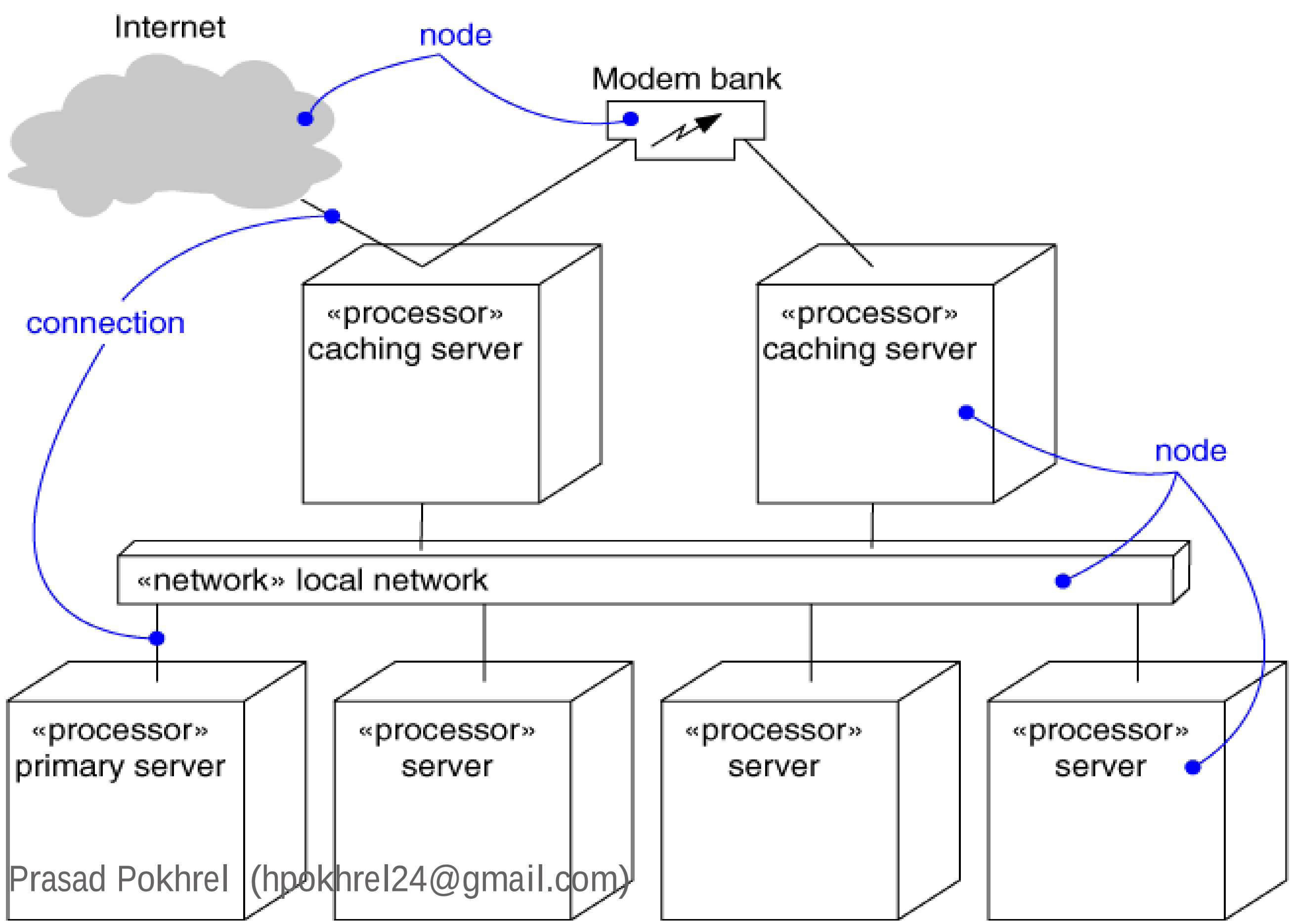
Node

- A **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.
- A node typically represents a processor or a device on which components may be deployed



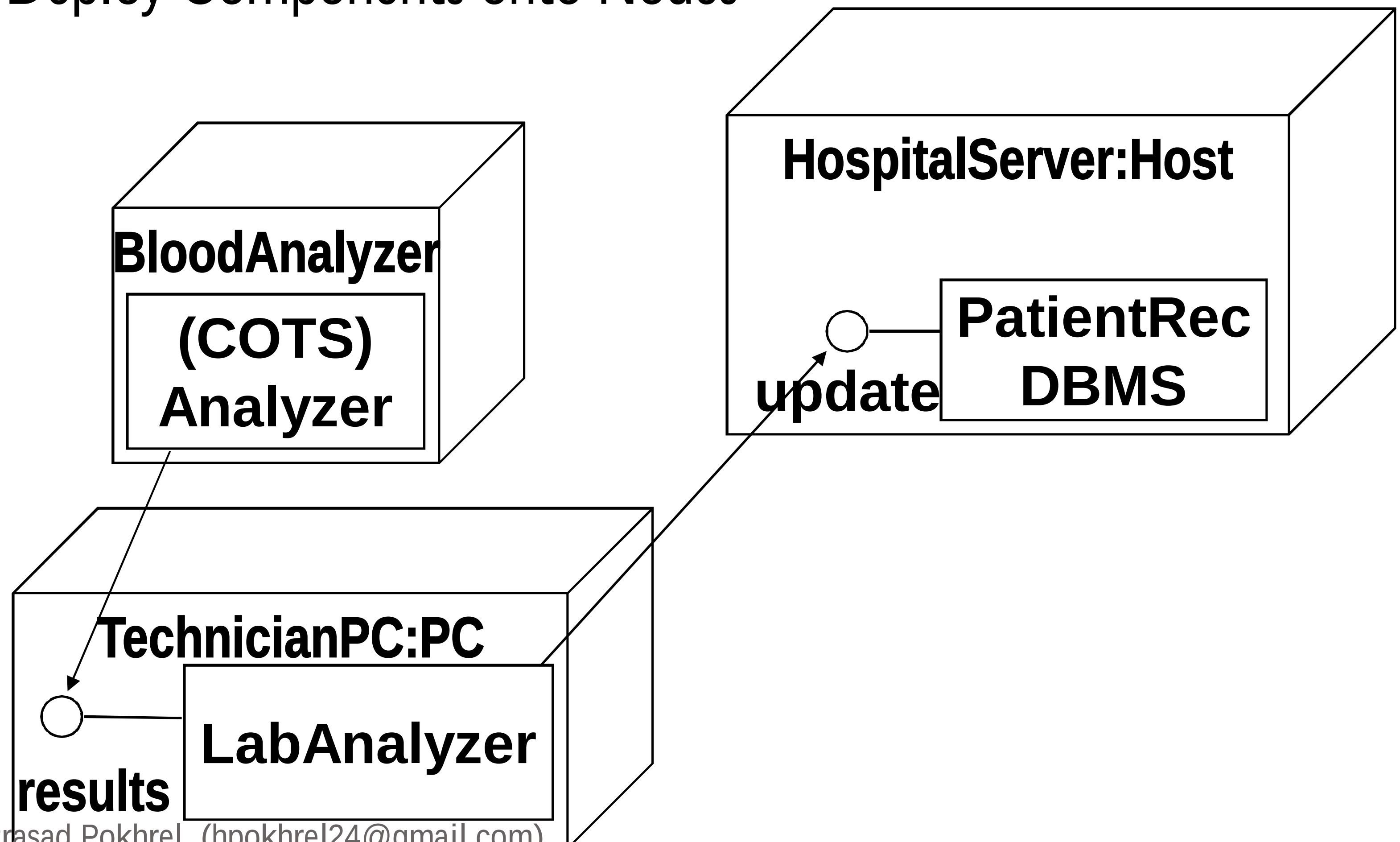
Deployment Diagram

- Captures the Topology of a System's Hardware

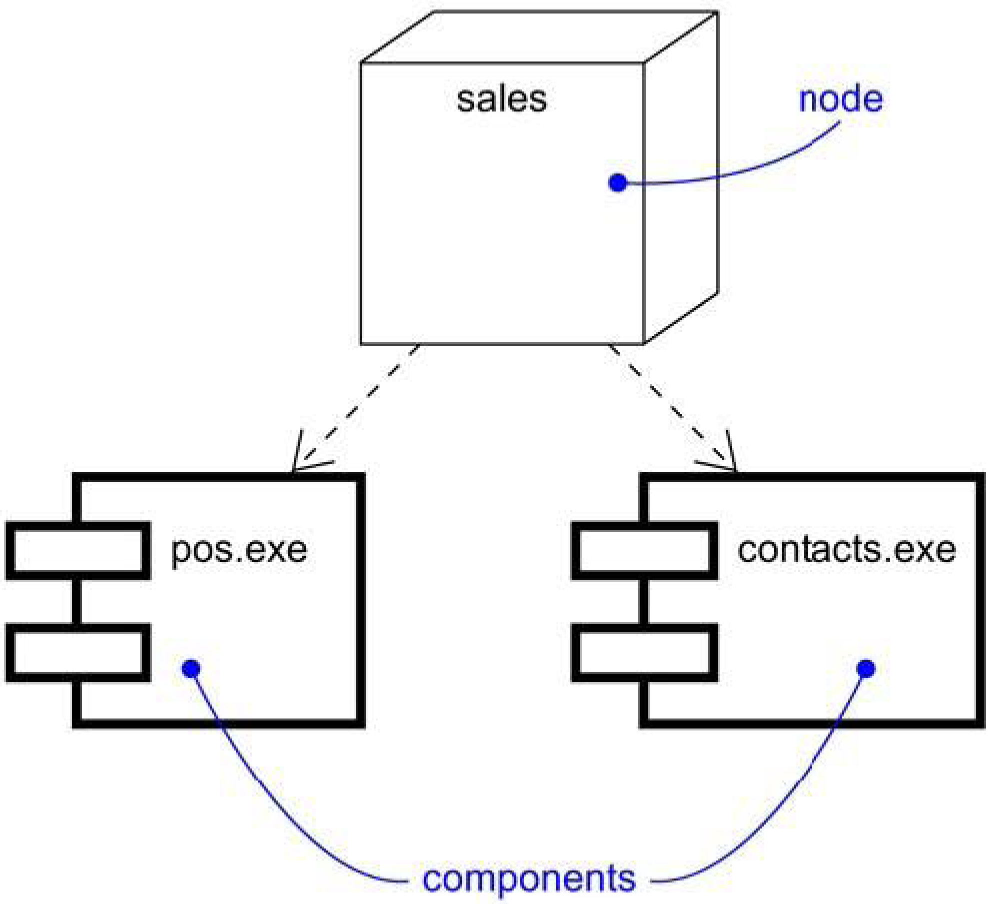


Deployment Diagram

- Deploy Components onto Nodes

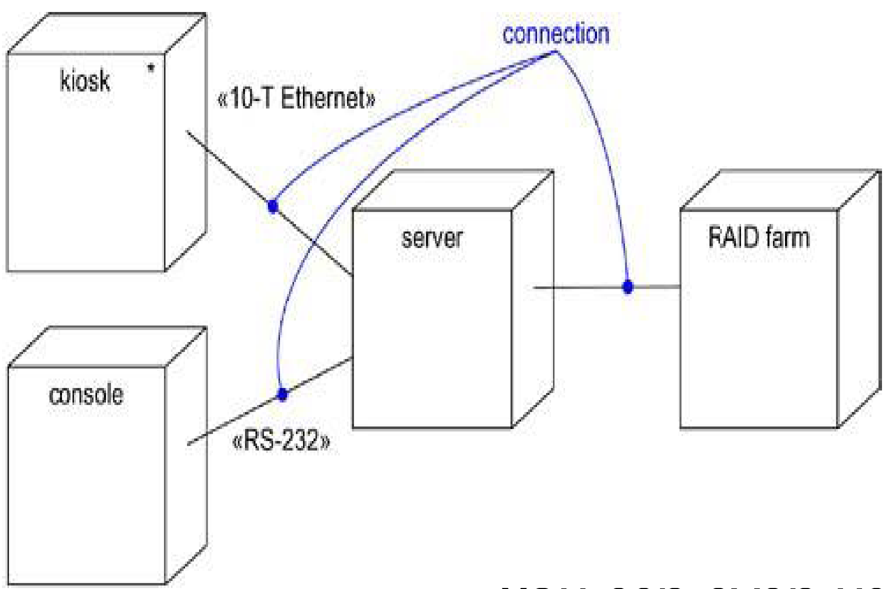


Nodes and Components



- Components are things that participate in the execution of a system; nodes are things that execute components.
- Components represent the physical packaging of otherwise logical elements; nodes represent the physical deployment of components.

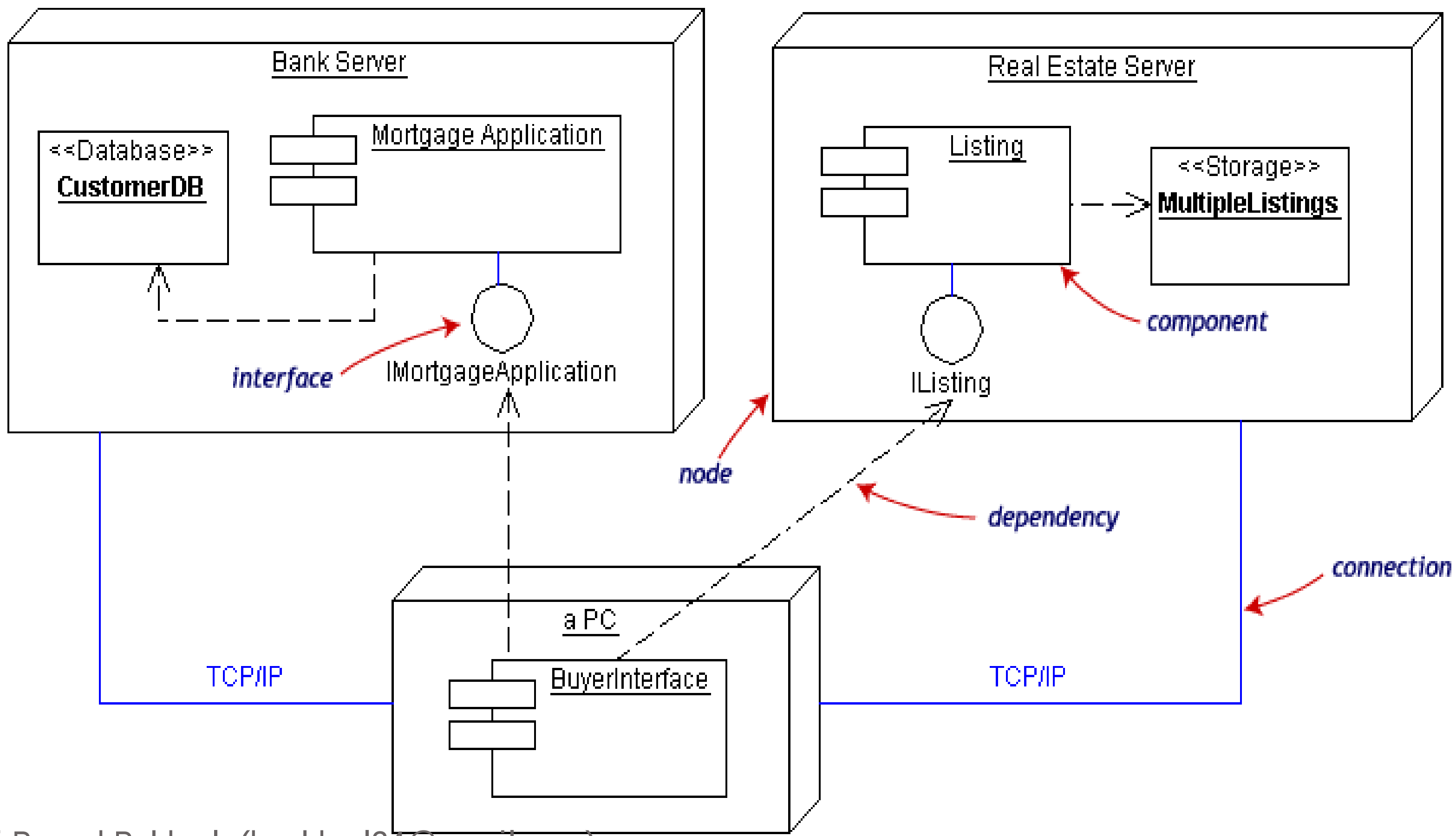
Connections



- The most common kind of relationship you'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 Figure shows

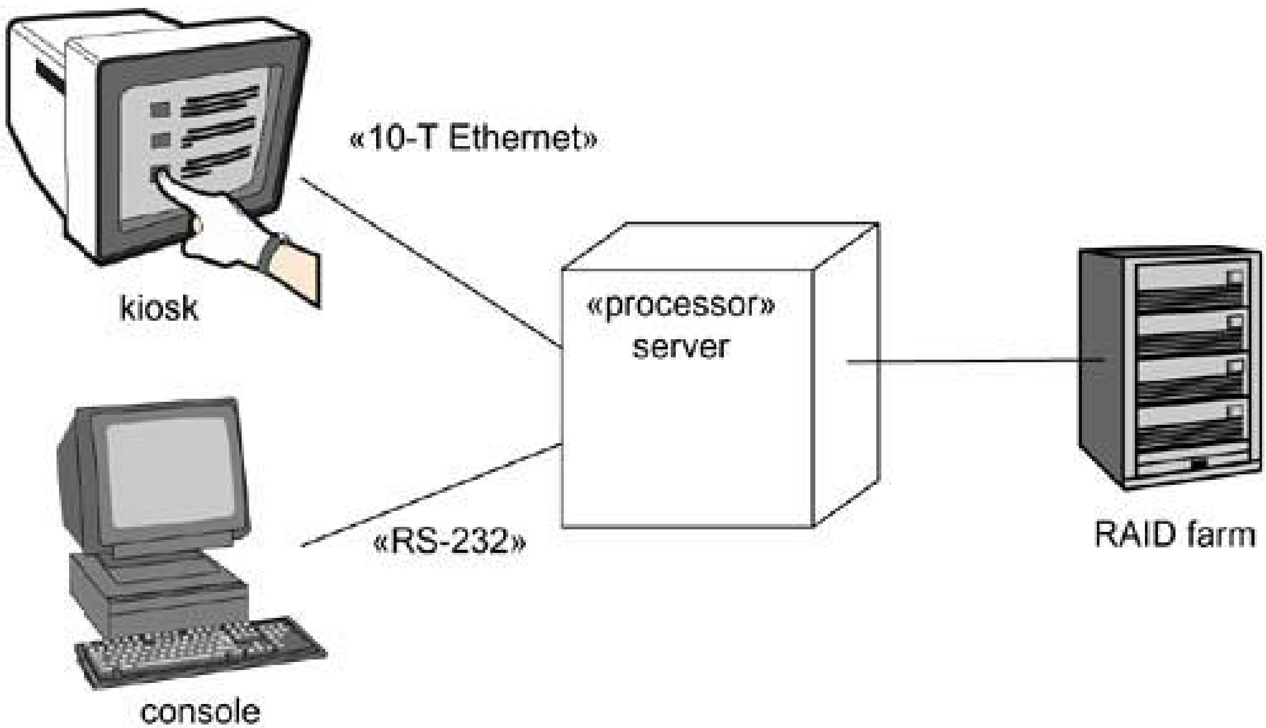
- You can even use associations to model indirect connections, such as a satellite link between distant processors.

Combining
Component and Deployment Diagrams

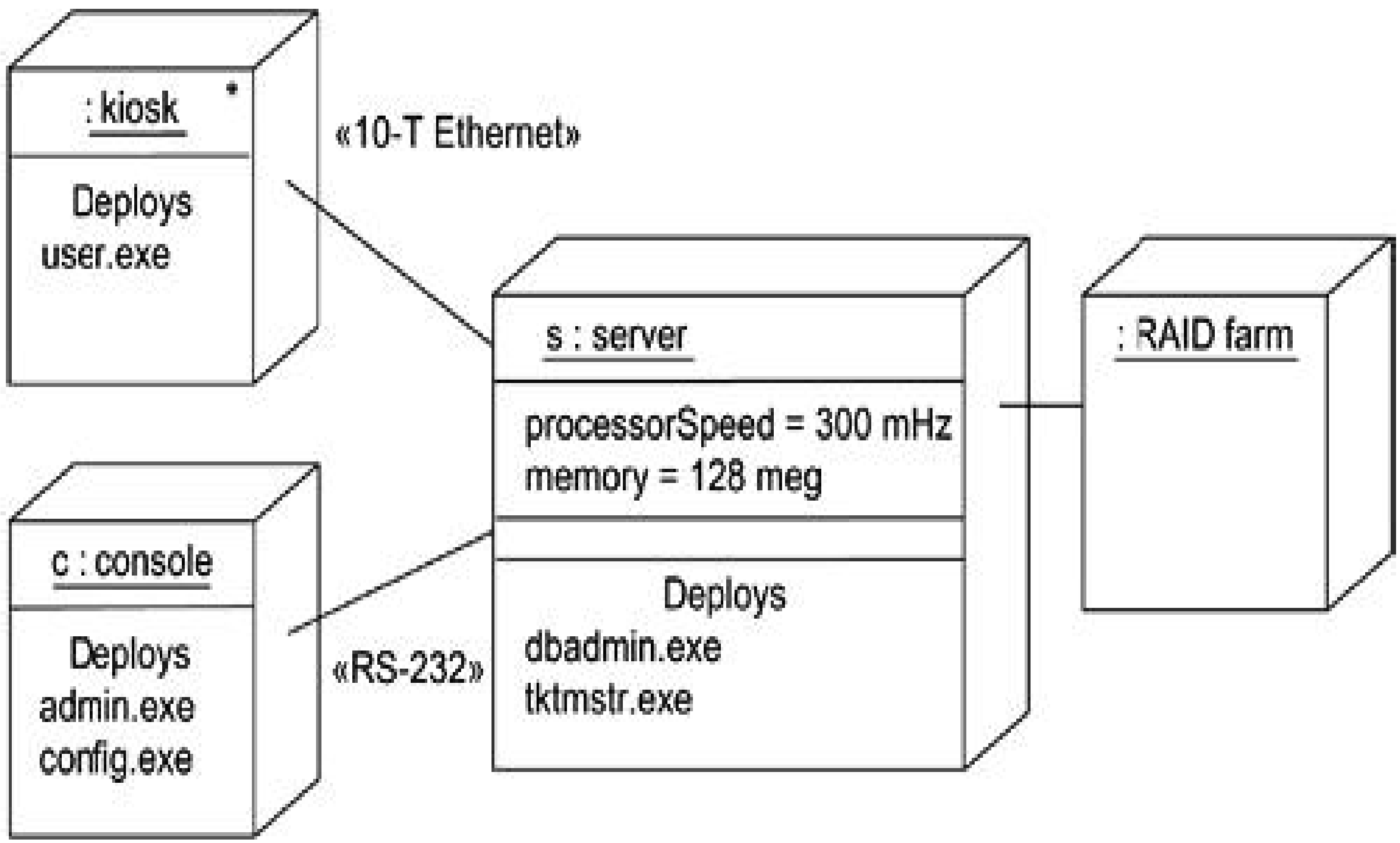


Modeling Processors and Devices

- Because all of the UML's extensibility mechanisms apply to nodes, you will often use stereotypes to specify new kinds of nodes that you can use to represent specific kinds of processors and devices.
- A *processor is a node that has processing capability, meaning that it can execute a* component.
- A *device is a node that has no processing capability (at least, none that are modeled* at this level of abstraction) and, in general, represents something that interfaces to the real world.


«10-T Ethernet»
kiosk
«processor»
server
RAID farm
console
«RS-232»

Modeling the Distribution of Components.



Dynamic: Interaction Diagrams

- A series of diagrams describing the *dynamic behavior* of an object-oriented system.
  - A set of messages exchanged among a set of objects within a context to accomplish a purpose.
- Often used to model the way a use case is realized through a sequence of messages between objects.
- Interaction diagrams are used for capturing dynamic nature of a system

Dynamic: Interaction Diagrams (Cont.)

- The purpose of Interaction diagrams is to:
  - Model interactions between objects
  - Assist in understanding how a system (a use case) actually works
  - Verify that a use case description can be supported by the existing classes
  - Identify responsibilities/operations and assign them to classes

Interaction Diagrams (Cont.)

- UML
  - Collaboration Diagrams
    - Emphasizes structural relations between objects
  - Sequence Diagram
    - Sequence diagrams are used to capture time ordering of message flow

Generally a set of sequence and collaboration diagrams are used to model an entire system

Two kinds of UML Interaction Diagrams

- **Sequence** Diagrams: show object interactions arranged in time sequence, *vertically*
- **Communication** Diagrams: show object interactions arranged as a flow of objects and their links to each other, *numerically*
- Semantically equivalent, structurally different
  - Sequence diagram emphasize time ordering
  - Communication diagrams make object linkages explicit

Interaction and Message

An interaction is a behavior that comprises a set of messages, exchanged among a set of objects, to accomplish a specific purpose.

A message is the specification of a communication between objects that conveys information, with the expectation that some kind of activity will ensue (follow).

From the name *Interaction* it is clear that the diagram is used to describe some type of interactions among the different elements in the model

Interaction..

- This interactive behavior is represented in UML by two diagrams known as *Sequence diagram* and *Collaboration diagram.*

- Sequence diagram emphasizes on time sequence of messages and collaboration diagram emphasizes on the structural organization of the objects that send and receive messages

- The purposes of interaction diagrams are to visualize the interactive behavior of the system

Interaction..

- So the purposes of interaction diagram can be describes as:
  - To capture dynamic behavior of a system.
  - To describe the message flow in the system.
  - To describe structural organization of the objects.
  - To describe interaction among objects.

Dynamic: Sequence Diagram

- Sequence Diagram: For a Task, Indicates the Object Interactions Over Time that are Needed
- Captures Dynamic Behavior (Time-oriented)
- Purposes:
  - Model Flow Of Control
  - Illustrate Typical Scenarios
  - Provide Perspective on Usage an Flow
- Main Concepts: Interaction, Object, Message, Activation
- Notes:
  - Dynamic Diagrams are Complementary
  - Provide Contrasting Perspectives of "Similar" Information and Behavior

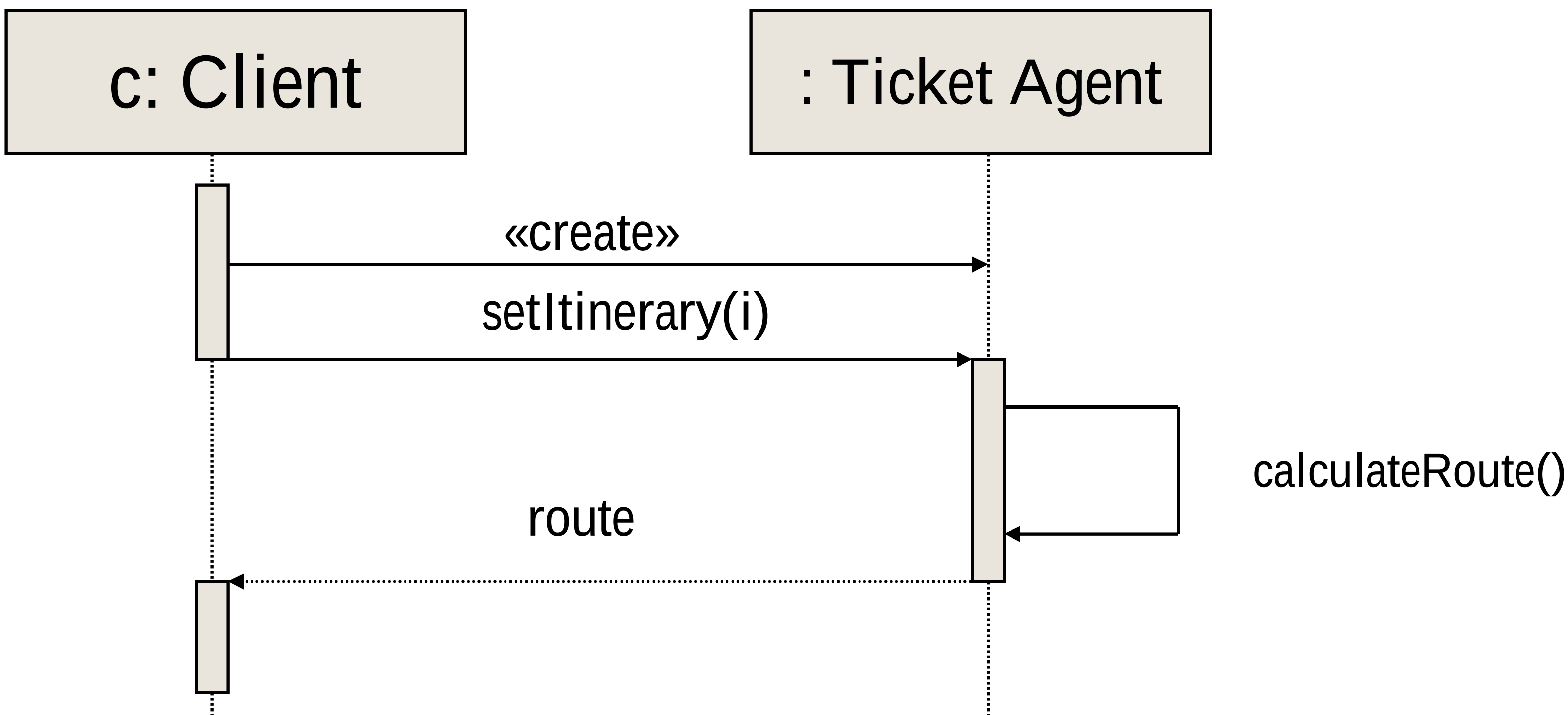
Sequence Diagram

A sequence diagram is an interaction diagram that emphasizes the time ordering of messages.

A lifeline is a vertical dashed line that represents the lifetime of an object.

A focus of control is a tall, thin rectangle that shows the period of time during which an object is performing an action.

Sequence Diagram Notation



Types of Messages

- Synchronous (flow interrupt until the message has completed.
- Asynchronous (don't wait for response)
- Flat – no distinction between sysn / async
- Return – control flow has returned to the caller.

- **Create message**
  - A create message represents the creation of an instance in an interaction. The create message is represented by the keyword «create». The target lifeline begins at the point of the create message.
- **Destroy message**
  - A destroy message represents the destruction of an instance in an interaction. The destroy message is represented by the keyword «destroy». The target lifeline ends at the point of the destroy message, and is denoted by an X.
- **Synchronous call message**
  - Synchronous calls, which are associated with an operation, have a send and receive message. A message is sent from the source lifeline to the target lifeline. The source lifeline is blocked from other operations until it receives a response from the target lifeline.
- **Asynchronous call message**
  - Asynchronous calls, which are associated with an operation, typically have only a send message, but can also have a reply message. In contrast to a synchronous message, the source lifeline is not blocked from receiving or sending other messages. You can also move the send and receive points individually to delay the time between the send and receive events. You might choose to do this if a response is not time-sensitive or order- sensitive.
- **Asynchronous signal message**
  - Asynchronous signal messages, are associated with a signal. A signal differs from a message in that there is no operation associated with the signal. A signal can represent an interrupt or error condition. To specify a signal, you create an asynchronous call message and change the type in the message properties view.

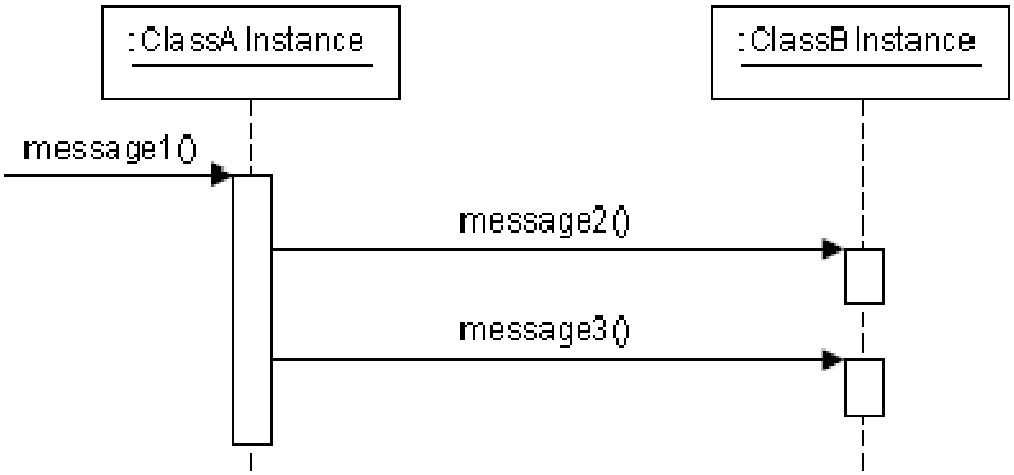
Synchronous and Asynchronous Calls

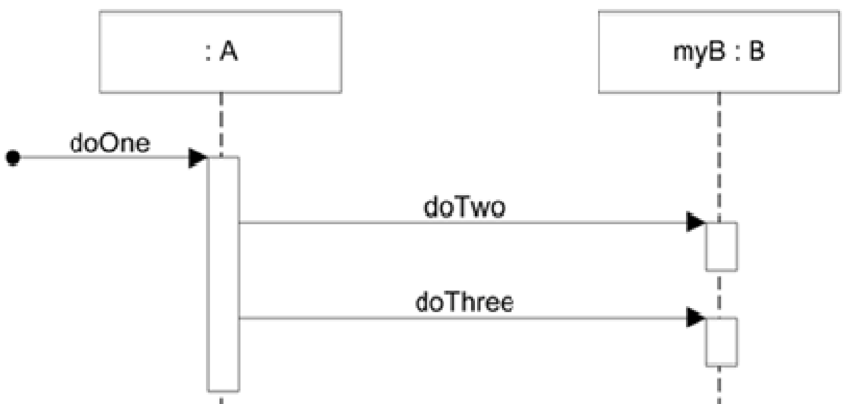
- If a caller sends a **synchronous message**, it must wait until the message is done, such as invoking a subroutine.
- If a caller sends an **asynchronous message**, it can continue processing and doesn't have to wait for a response.
- You see asynchronous calls in multithreaded applications and in message-oriented middleware.
- Asynchrony gives better responsiveness and reduces the temporal coupling but is harder to debug.

Sequence diagrams

- *Some control information can also be included.*
- *Two types of control information are particularly valuable:*
  - ***A condition*** *(e.g. [invalid] or [OK]) indicates that a message is sent, only if the condition is true.*
  - ***An iteration (*)*** *marker shows the message is sent many times to multiple receiver objects as would happen when a collection or the elements of an array are being iterated.The basis of the iteration can also be indicated e.g. [for every book object].*

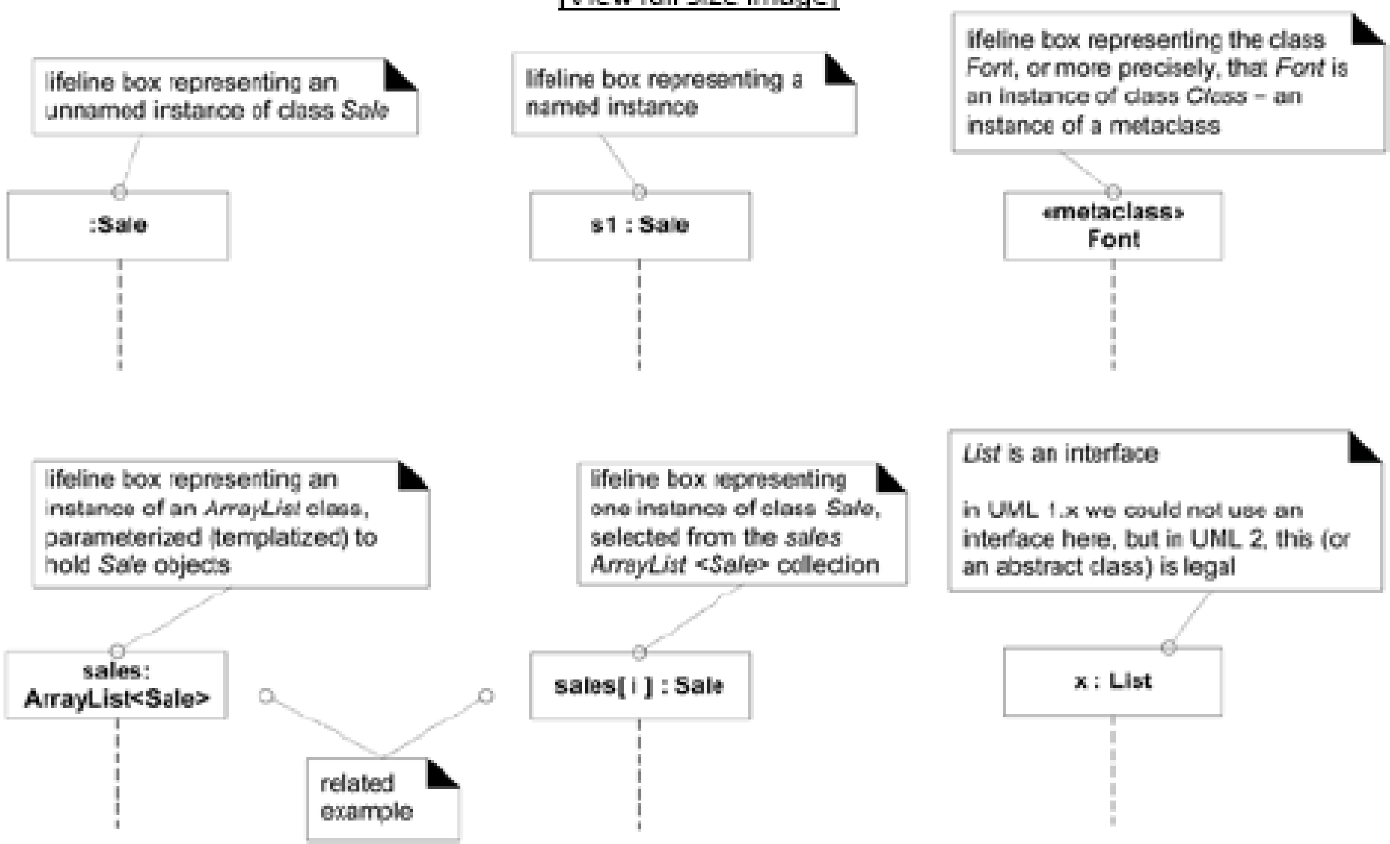
Sequence diagrams



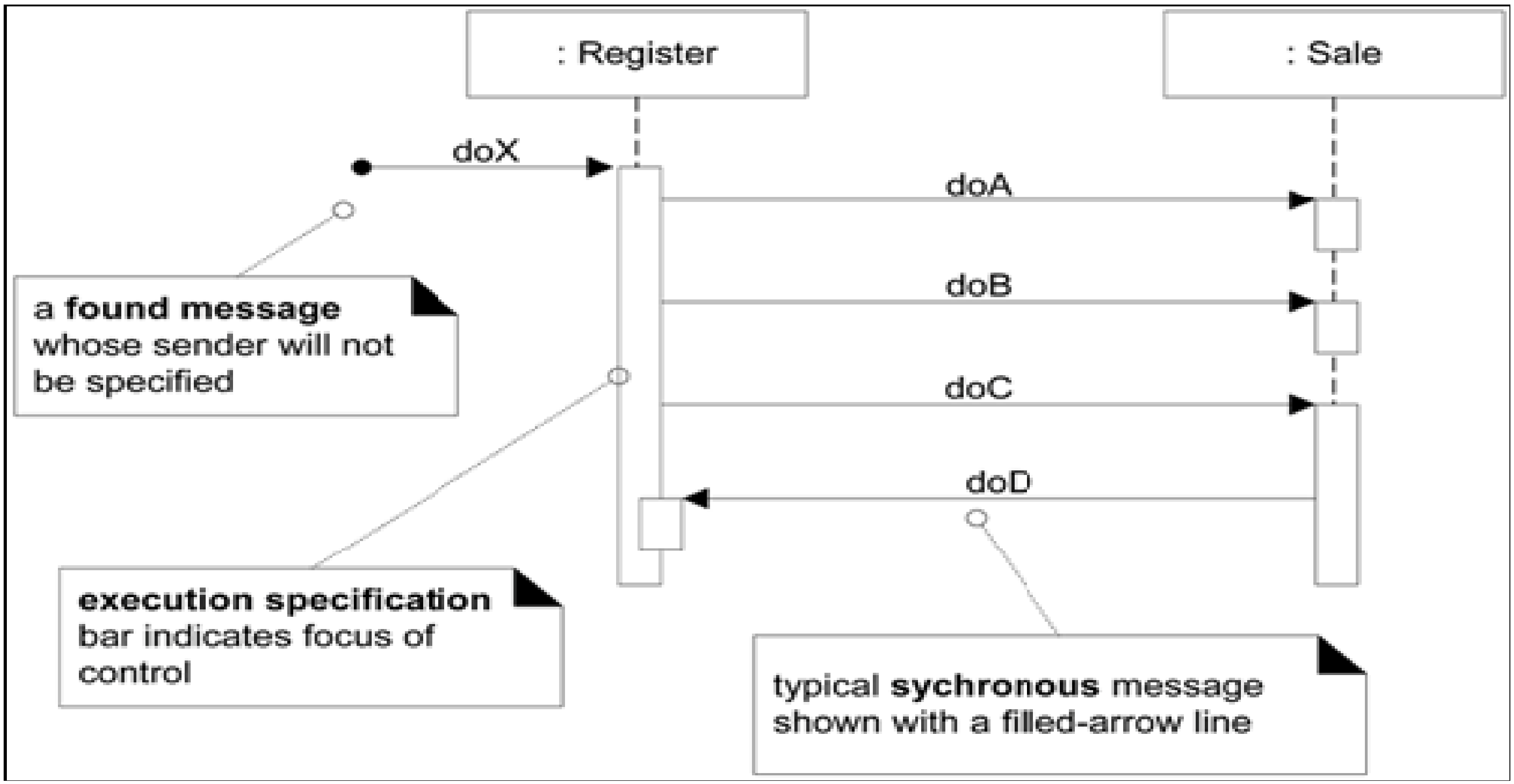


Lifeline box

[View full size image]

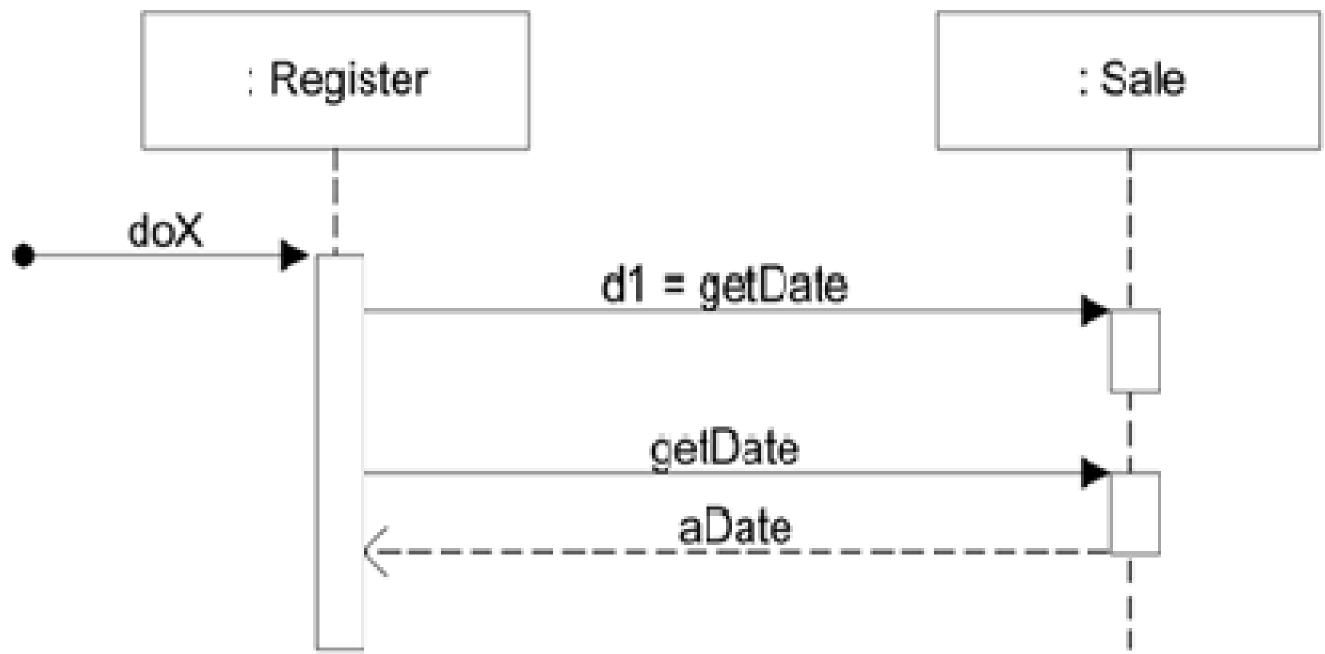


Messages



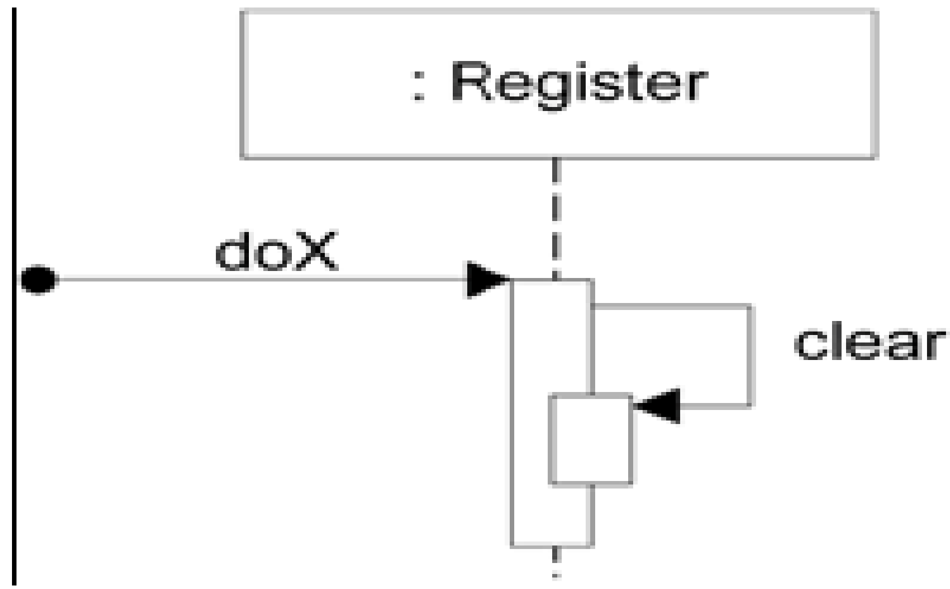
The first message doesn’t have a participant that sent it, as it comes from an undetermined source. It’s called a **found message**.

Reply or Returns

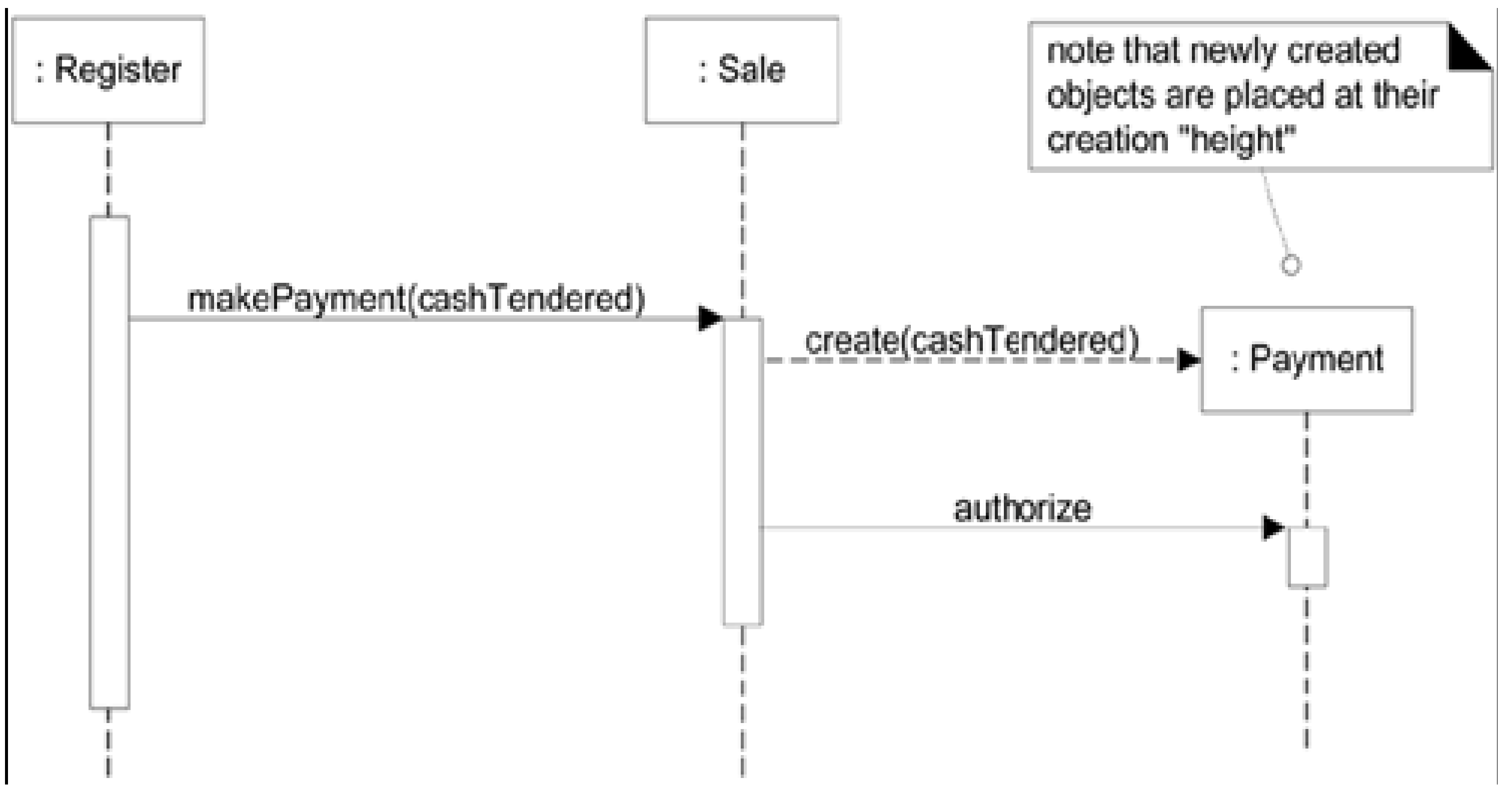


•**Using the message syntax returnVar = message(parameter).**
•**Using a reply (or return) message line at the end of an activation bar.**

Messages to "self" or "this"



Creation of Instances



Object Lifelines and Object Destruction

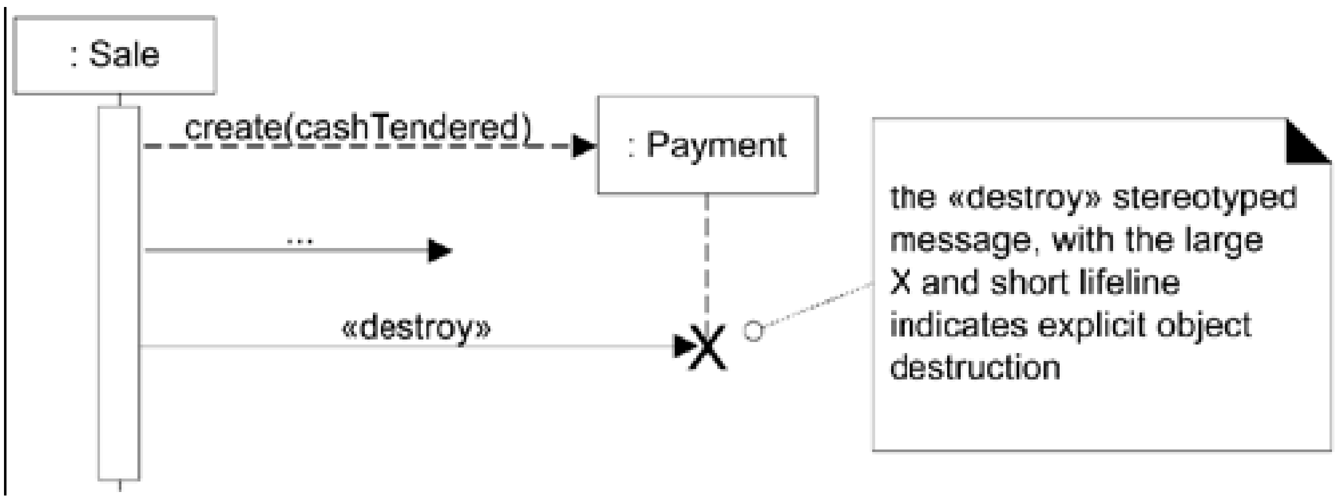
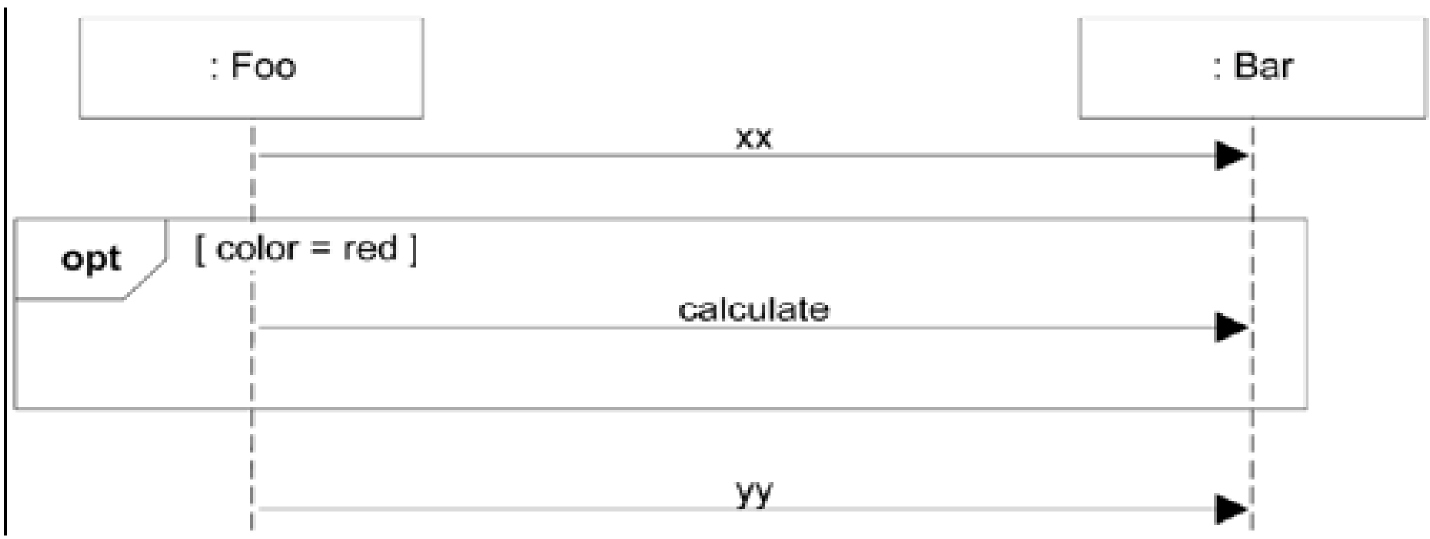


Diagram Frames

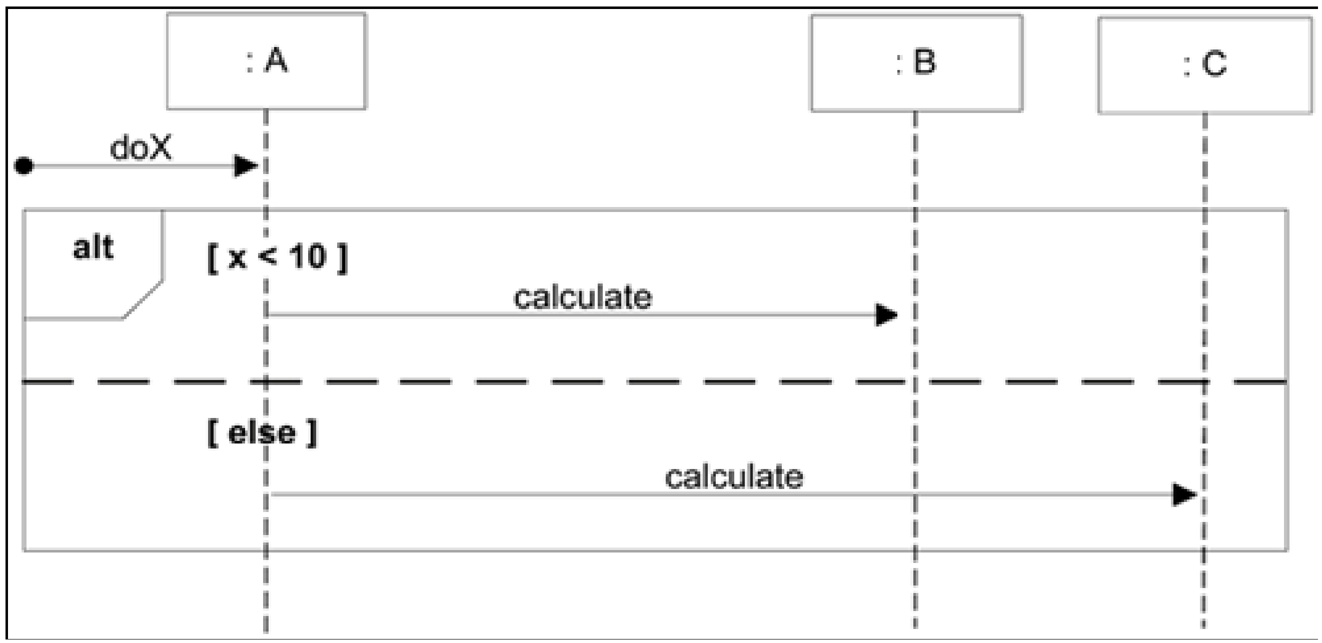
The following table summarizes some common frame operators:

Frame Operator	Meaning
alt	Alternative fragment for mutual exclusion conditional logic expressed in the guards.
loop	Loop fragment while guard is true. Can also write *loop(n)* to indicate looping n times. There is discussion that the specification will be enhanced to define a *FOR* loop, such as *loop(i, 1, 10)*
opt	Optional fragment that executes if guard is true.
par	Parallel fragments that execute in parallel.
region	Critical region within which only one thread can run.

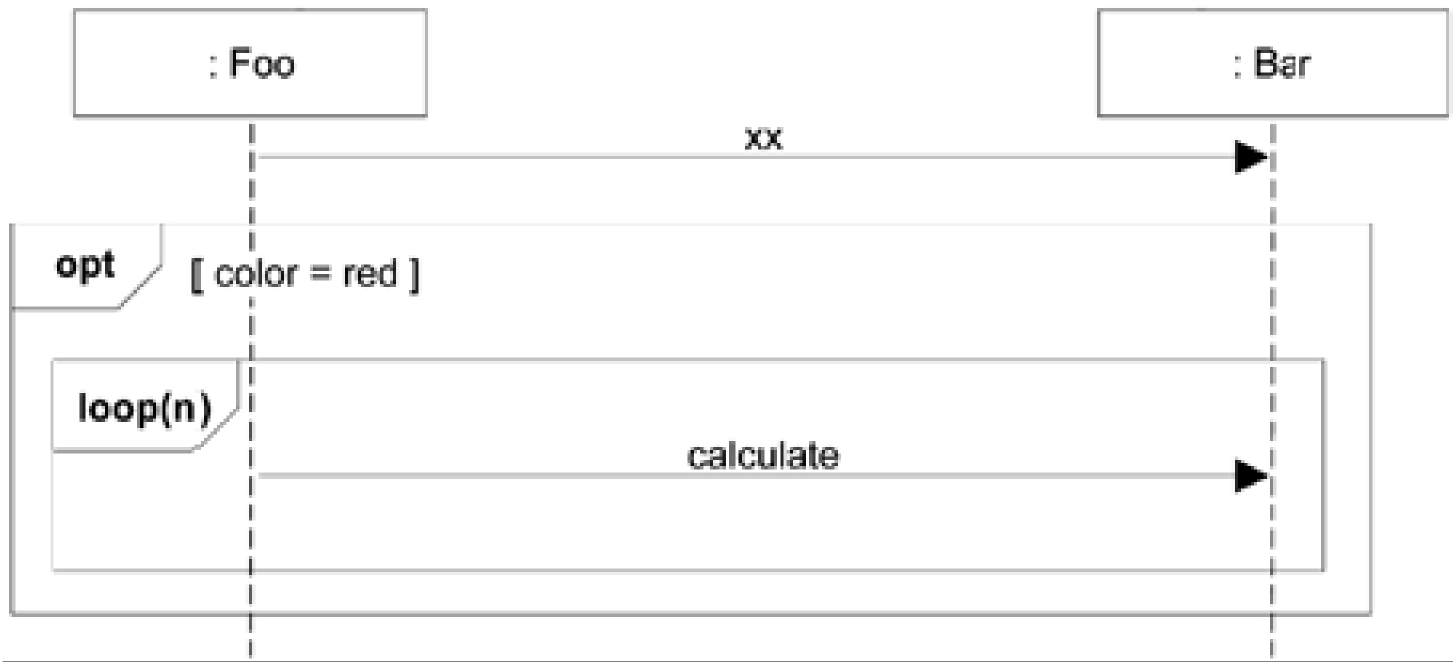
Conditional Messages

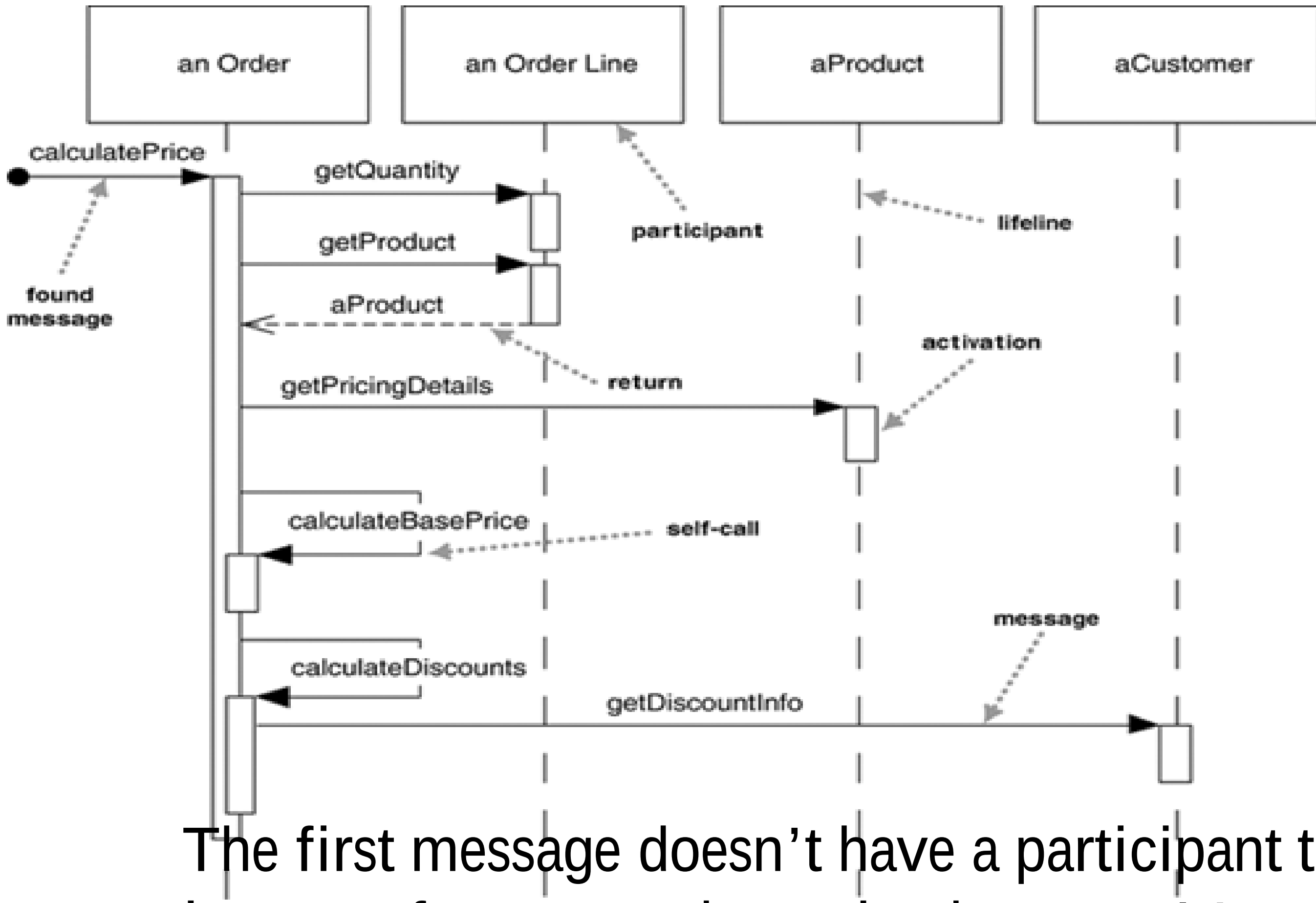


Mutually Exclusive Conditional Messages



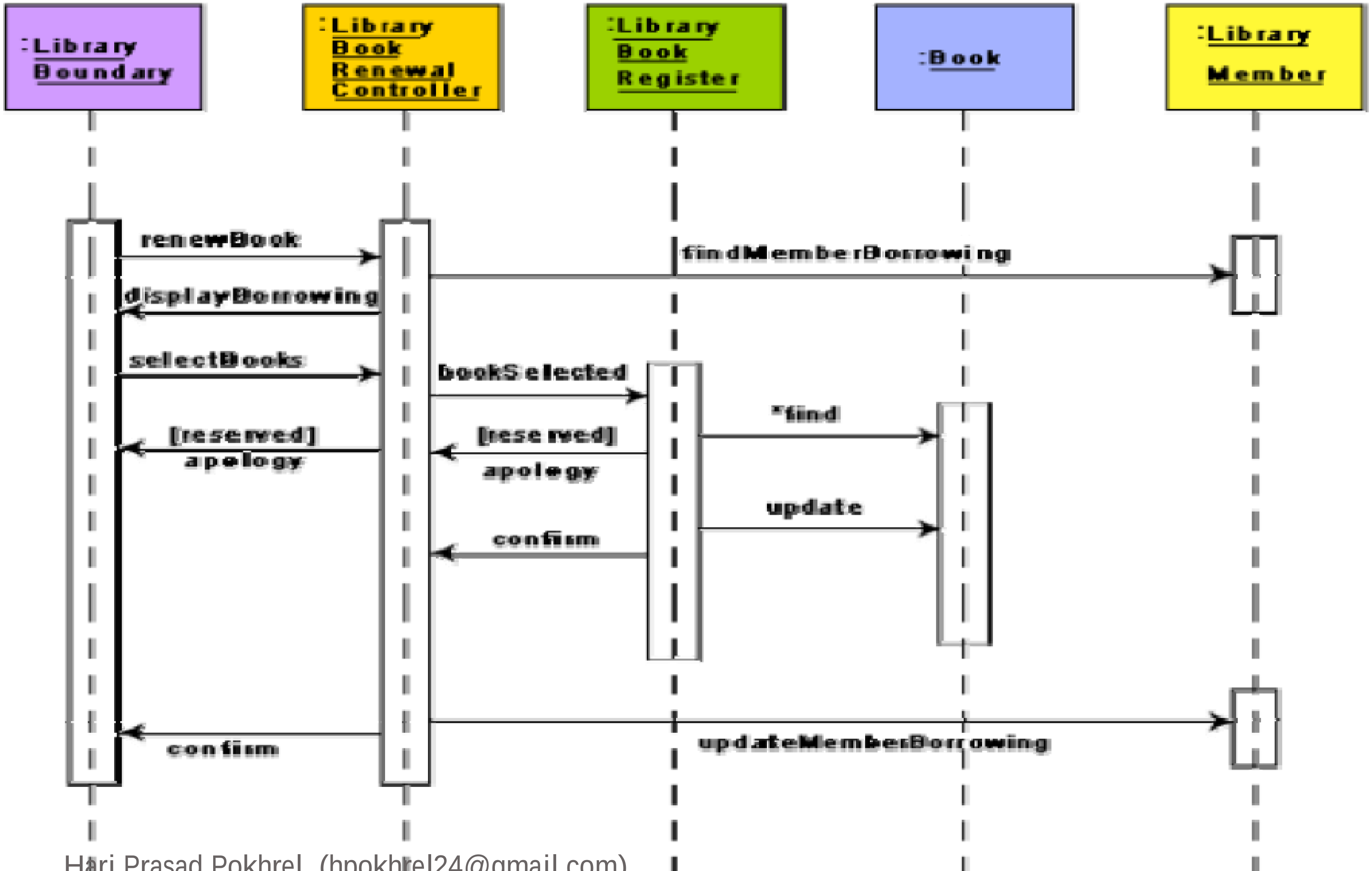
Nesting of frames.



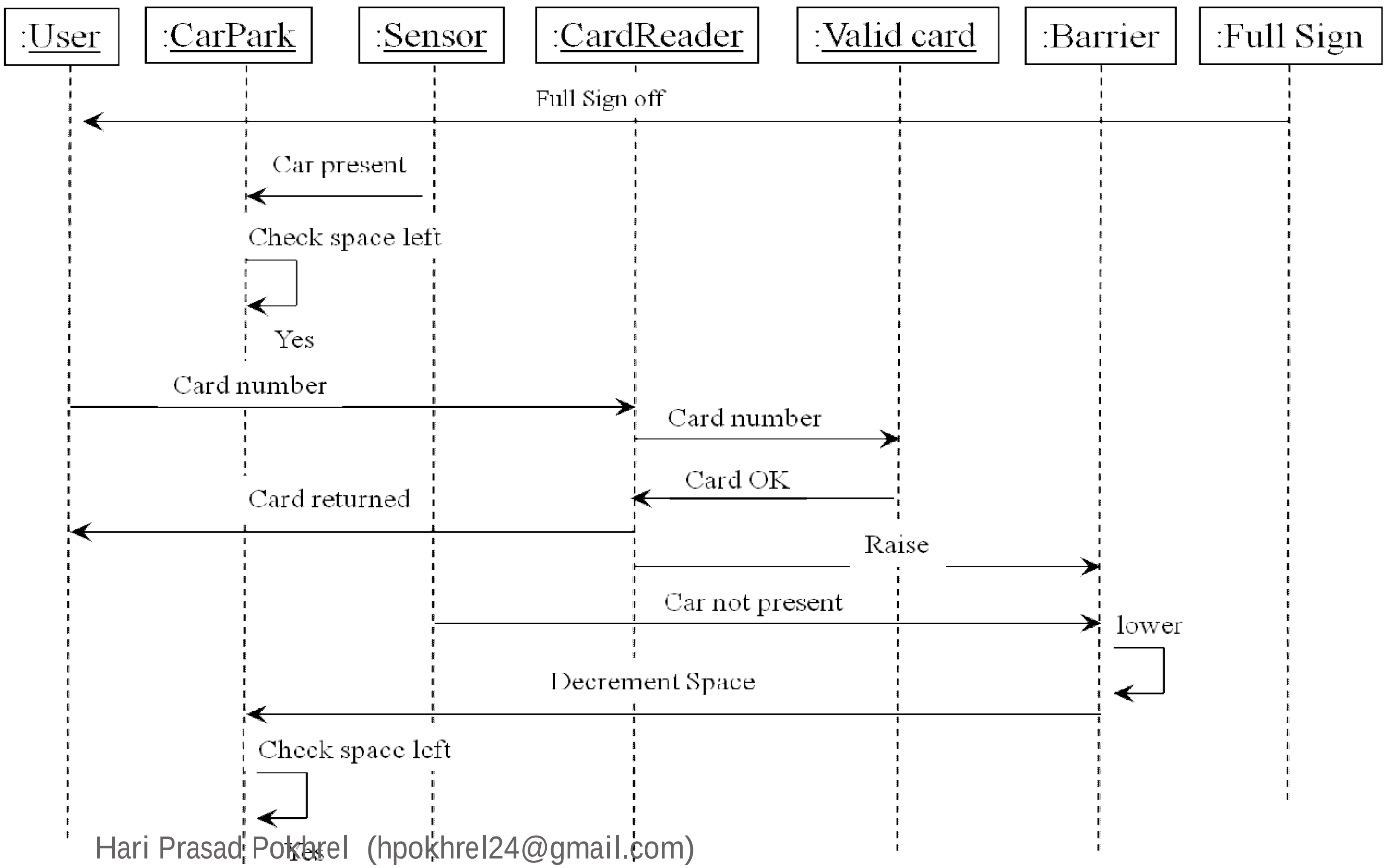


The first message doesn't have a participant that sent it, as it comes from an undetermined source. It's called a **found message**.

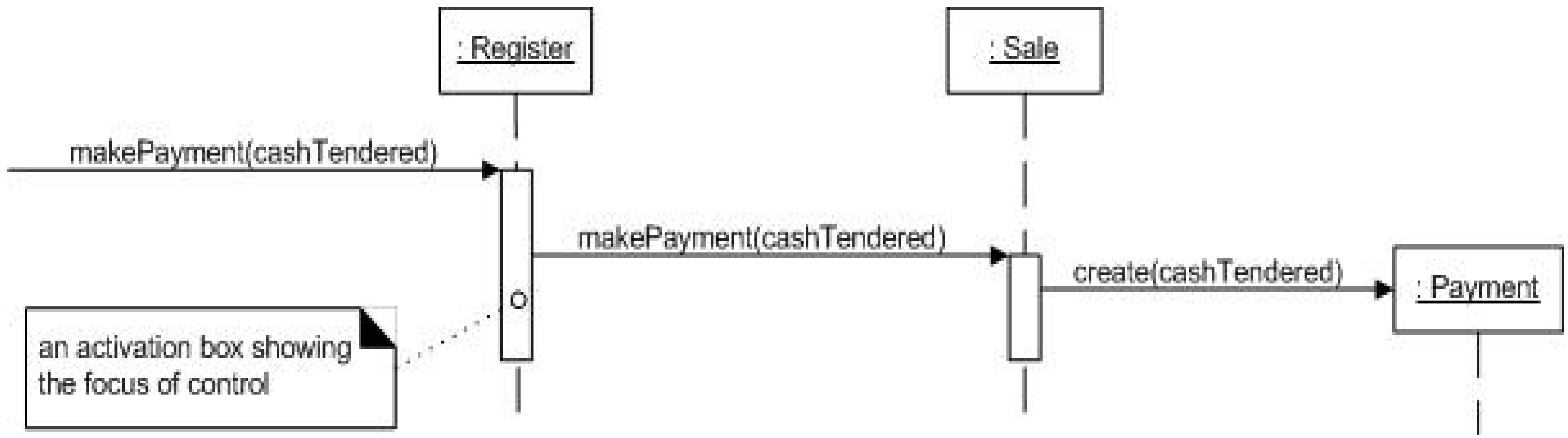
Sequence diagram for "book renew use case.



Sequence diagram for car parking



Sequence diagram of makePayment use case



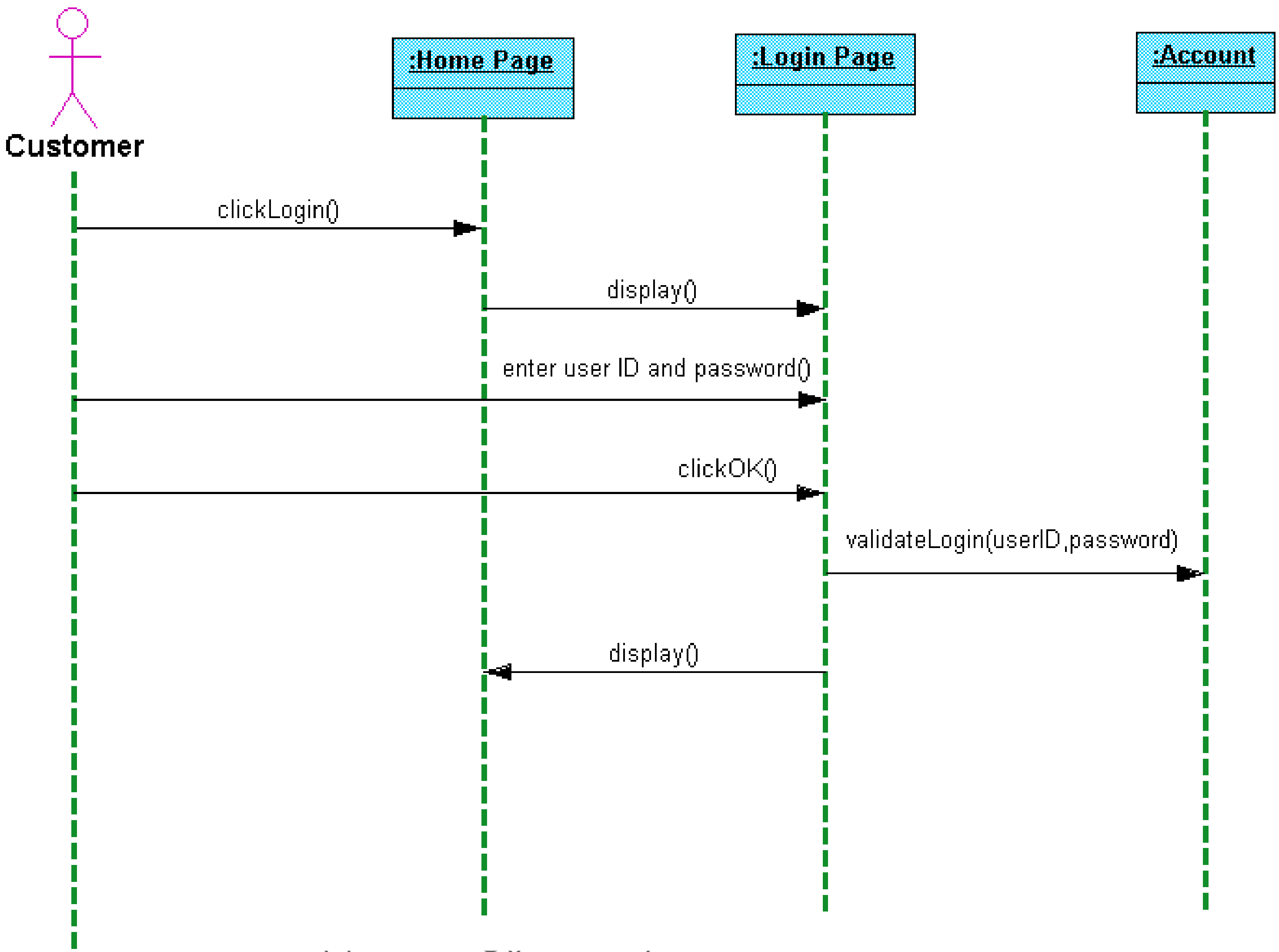
The sequence diagram shown in Figure *makePayment* is read as follows:

1. The message *makePayment* is sent to an instance of a *Register*. The sender is not identifie

2. The *Register* instance sends the *makePayment* message to a *Sale* instance.

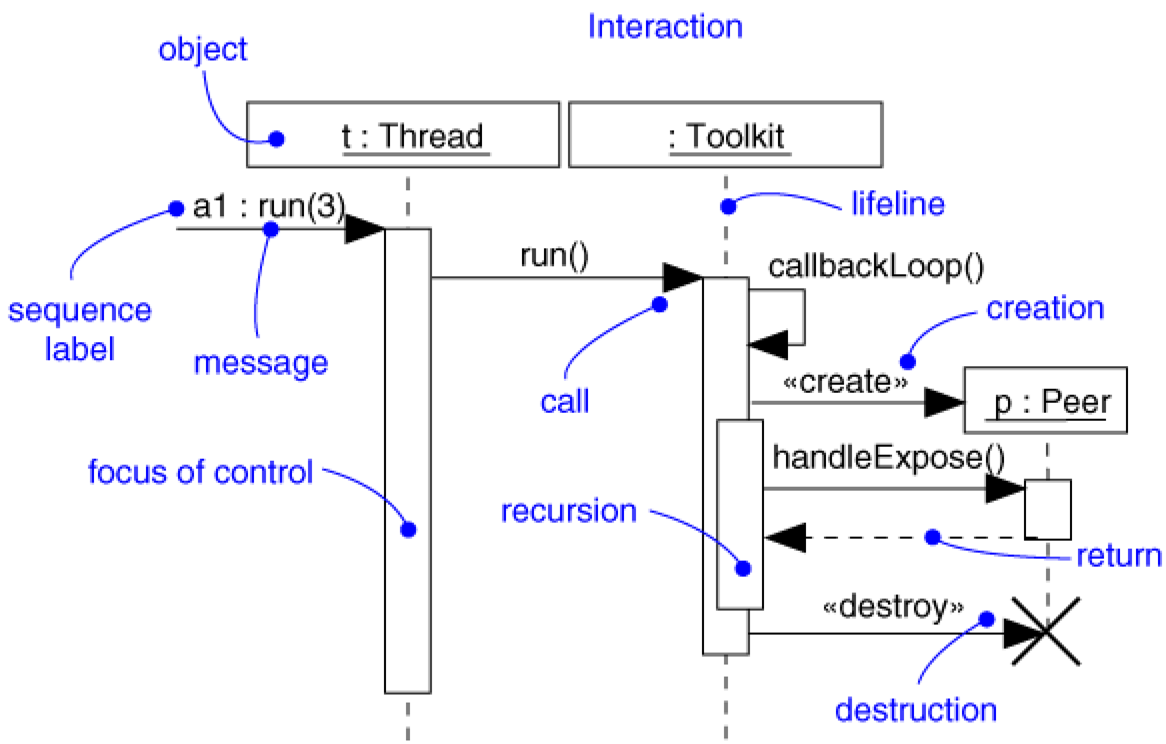
3. The *Sale* instance creates an instance of a *Payment*.

Sequence

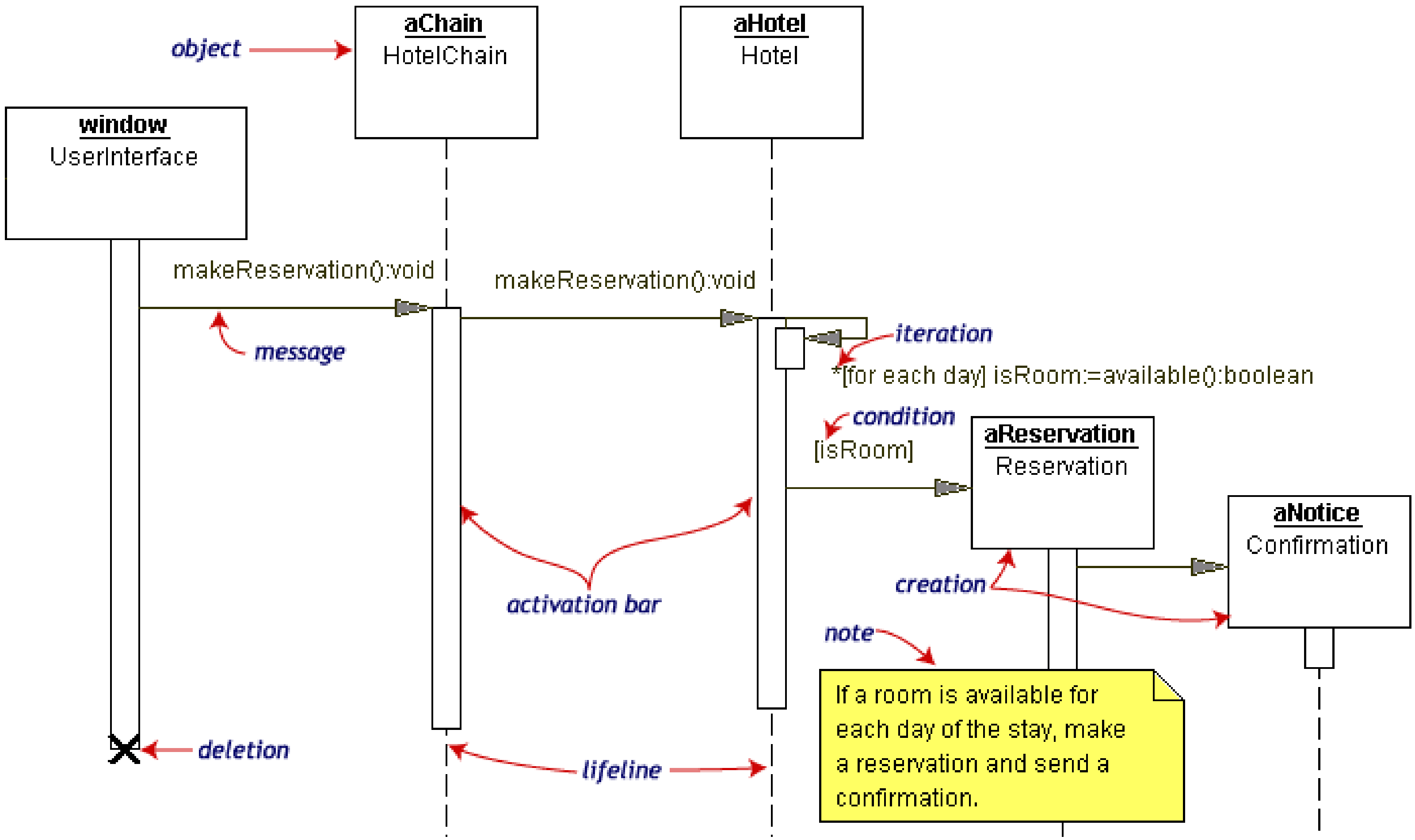

Customer
:Home Page
:Login Page
:Account
clickLogin()
display()
enter user ID and password()
clickOK()
validateLogin(userID,password)
display()

Sequence Diagram

- Captures Dynamic Behavior (Time-Oriented)

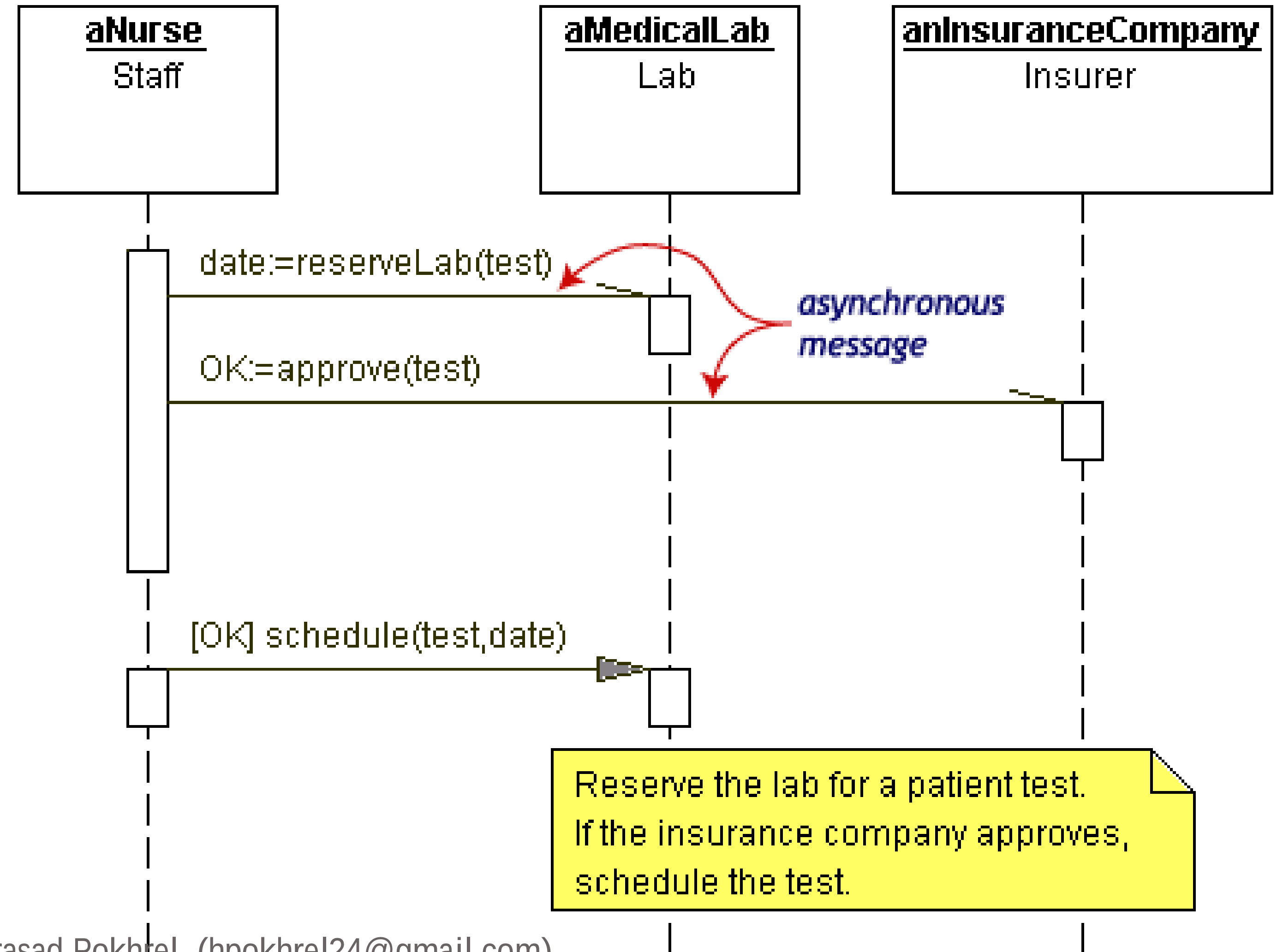


Sequence Diagram



Hari Prasad Pokhrel (hpokhrel24@gmail.com)

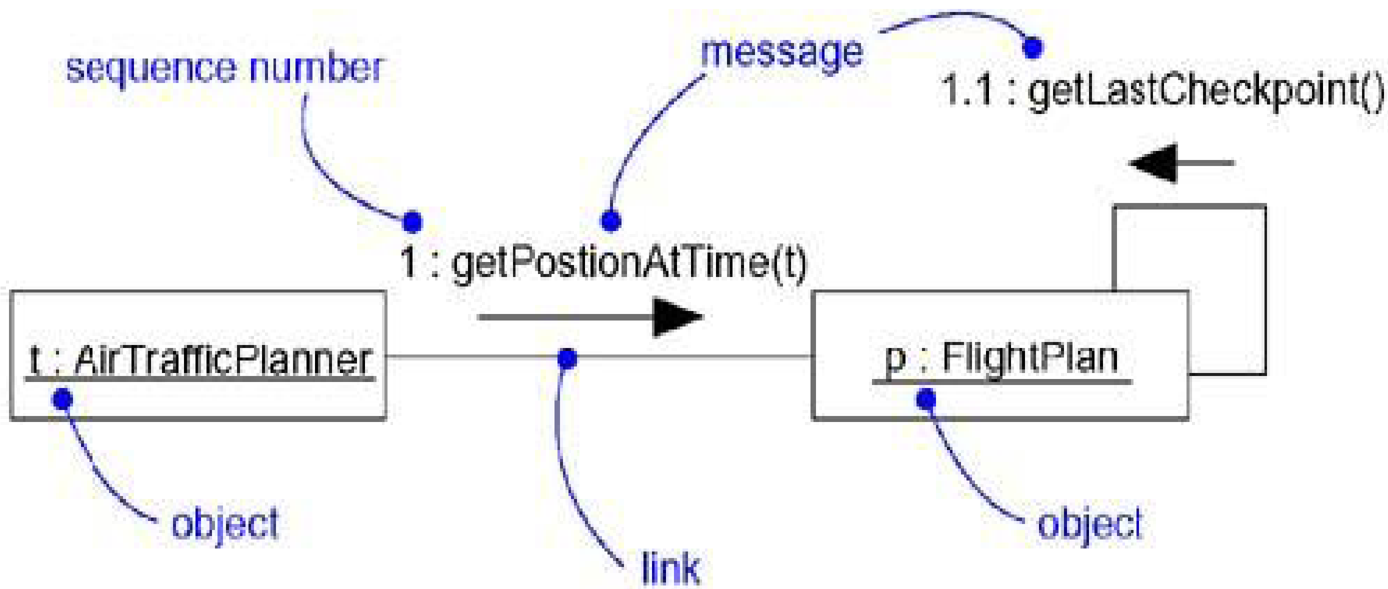
Sequence Diagram
HCA



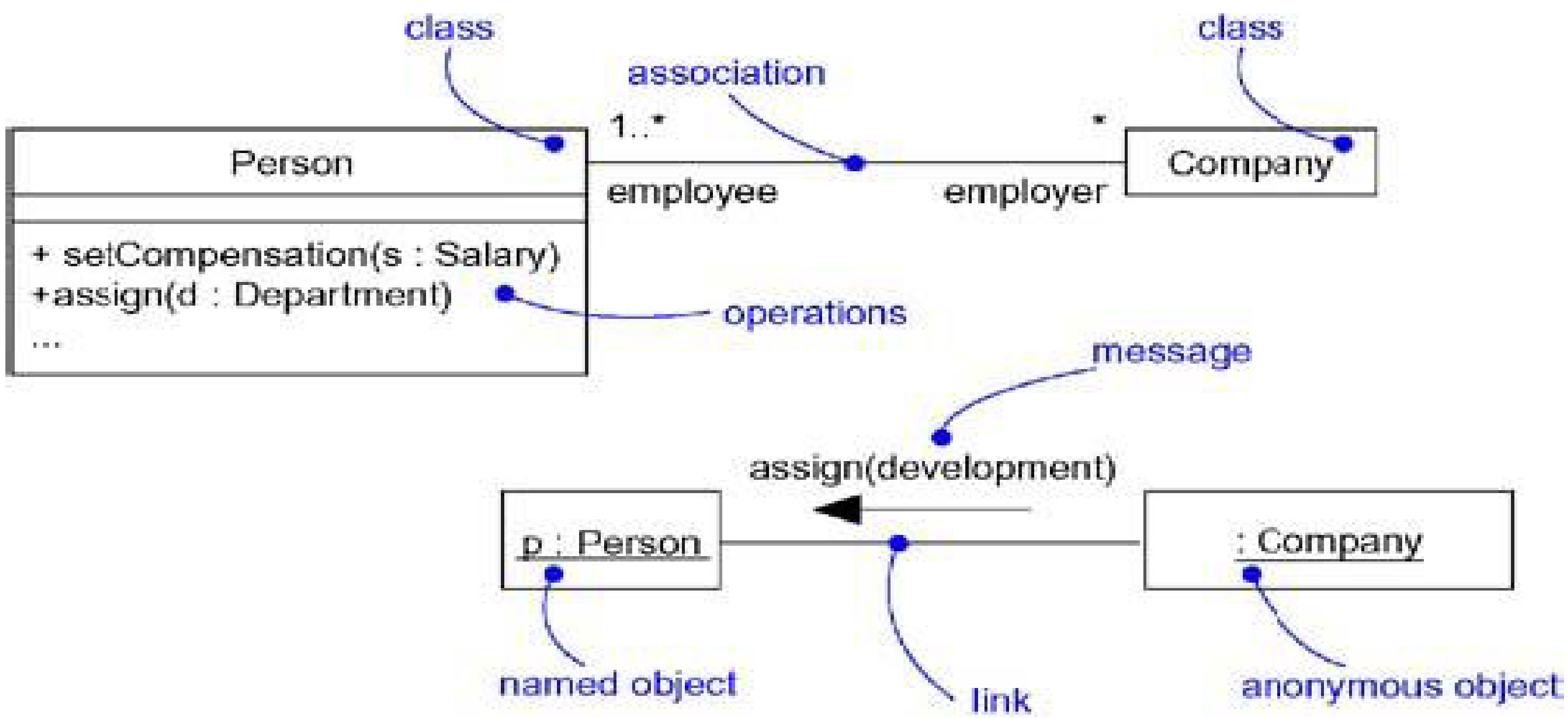
Dynamic: Collaboration Diagram

- Collaboration Diagram: Structured from the Perspective of Interactions Among Objects
- Captures Dynamic Behavior (Message-oriented)
- Purposes:
  - Model Flow of Control
  - Illustrate Coordination of Object Structure and Control
  - Objects that Interact with Other Objects
  - Are Collaboration Diagrams Really FSMs?
  - Sequence::Time vs. Collaboration::Message
- Main Concepts: Collaboration, Interaction, Collaboration Role, Message

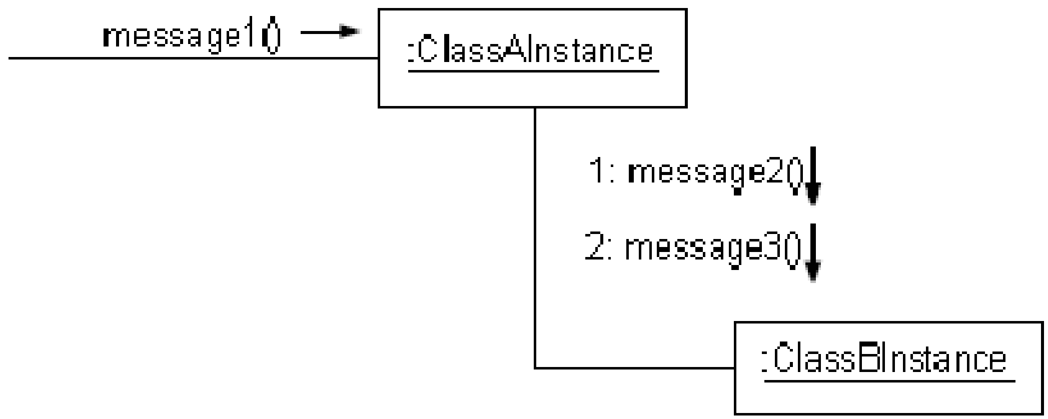
Messages, Links, and Sequencing

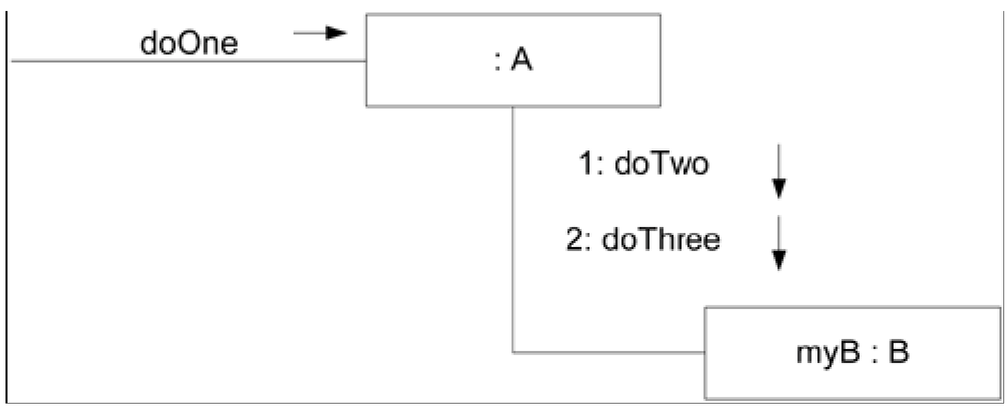


Links and Associations

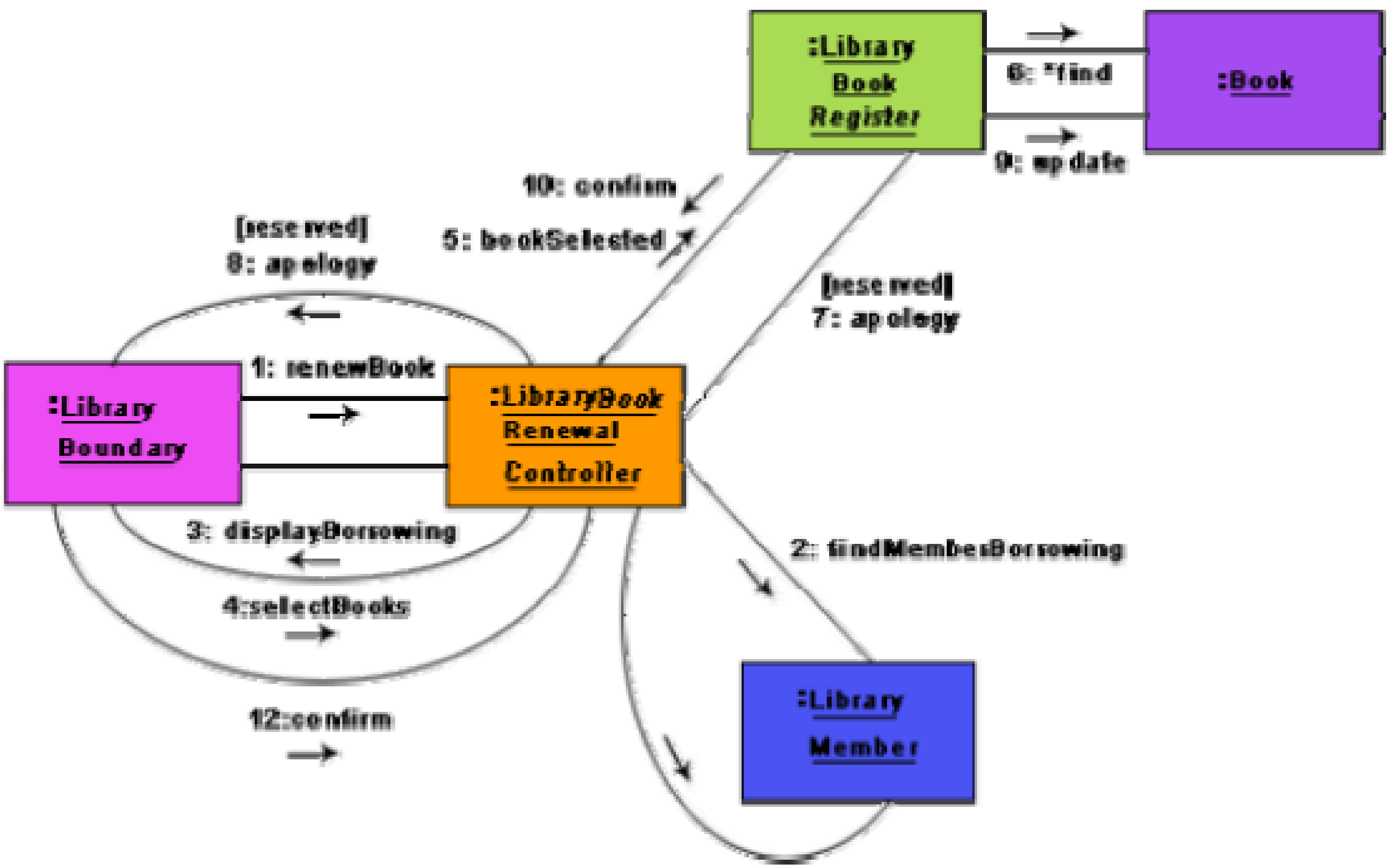


A link is a semantic connection among objects.
In general, a link is an instance of an association.
As Figure shows, wherever a class has an association to another class, there may be a link between the instances of the two classes; wherever there is a link between two objects, one object can send a message to the other object.


message1()
:ClassAInstance
1: message2()
2: message3()
:ClassBInstance


doOne
: A
1: doTwo
2: doThree
myB : B

Collaboration Diagram for book renew use case



Hari Prasad Pokhrel (hpokhrel24@gmail.com)

Procedural Sequence

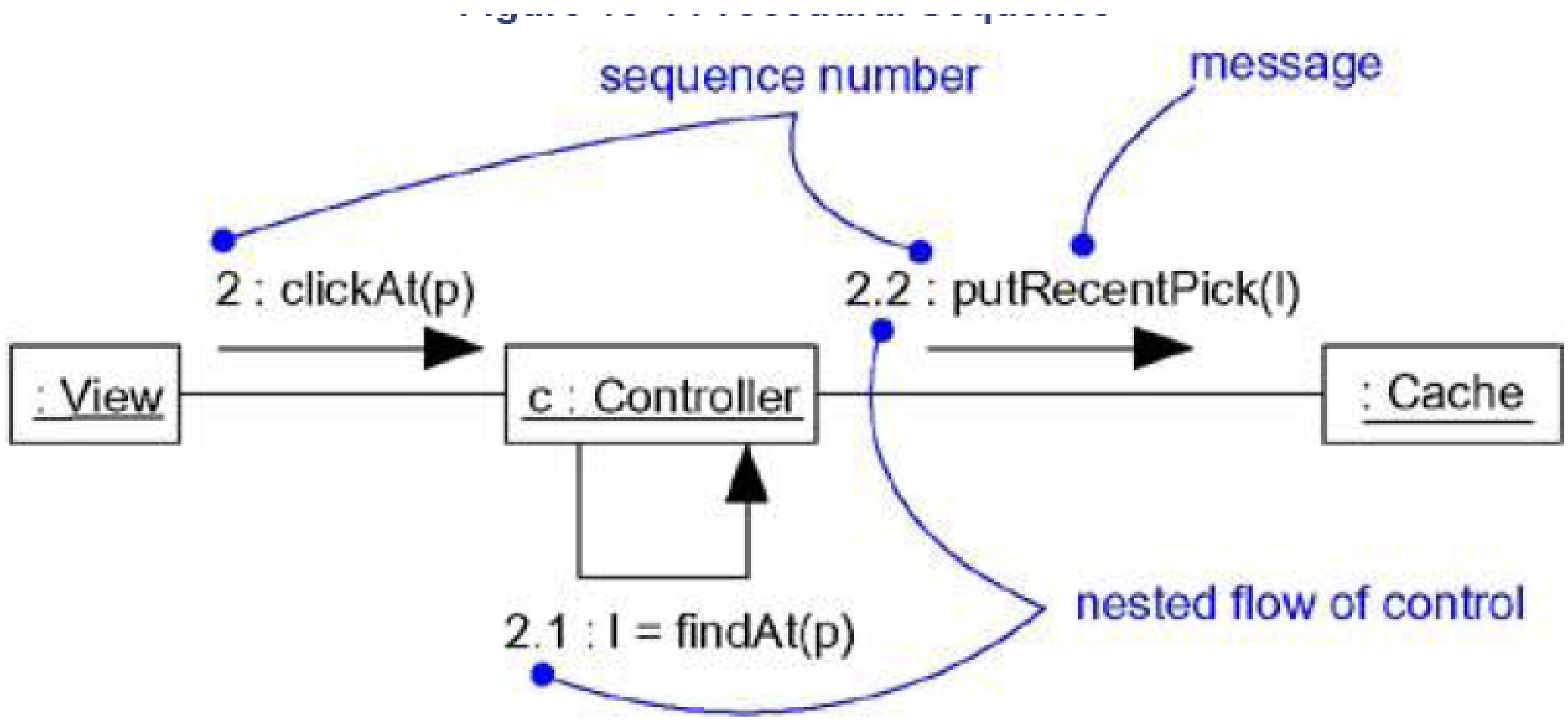
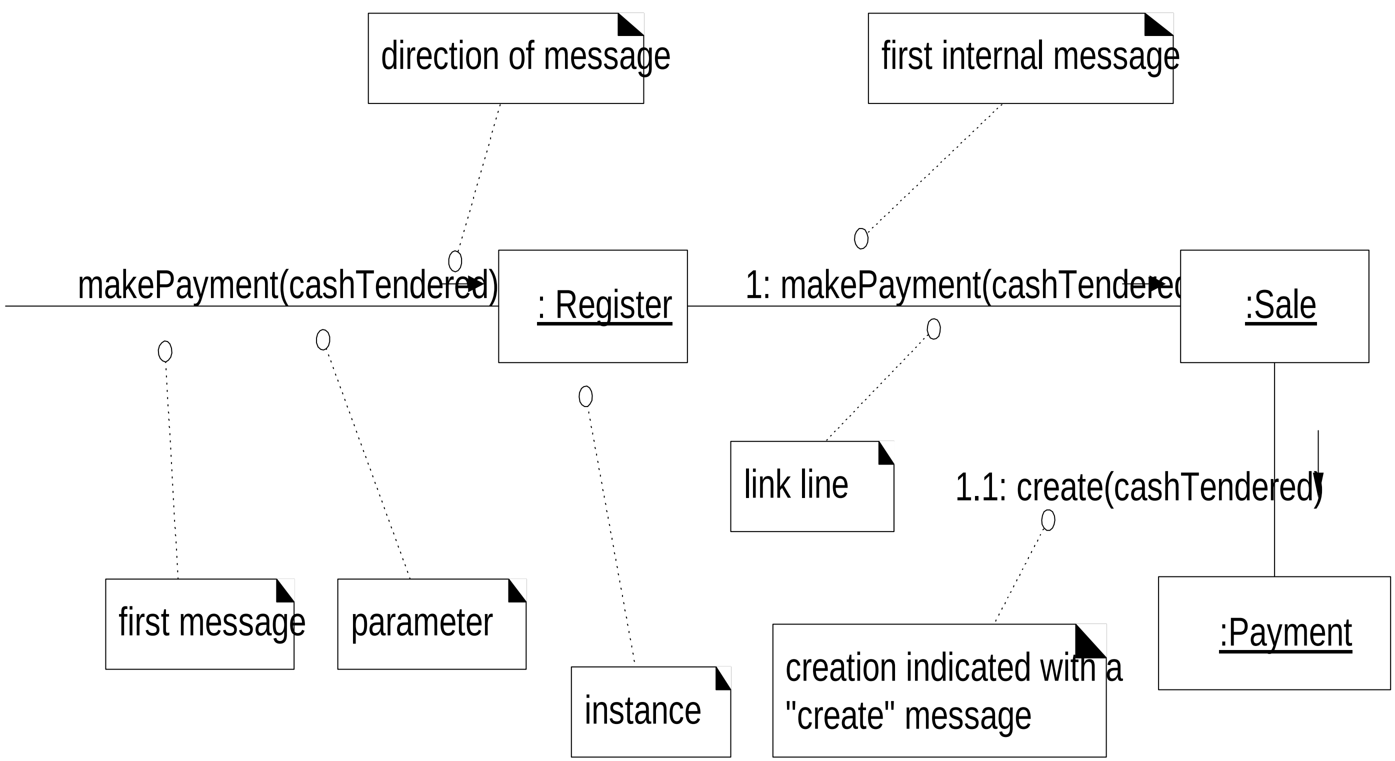
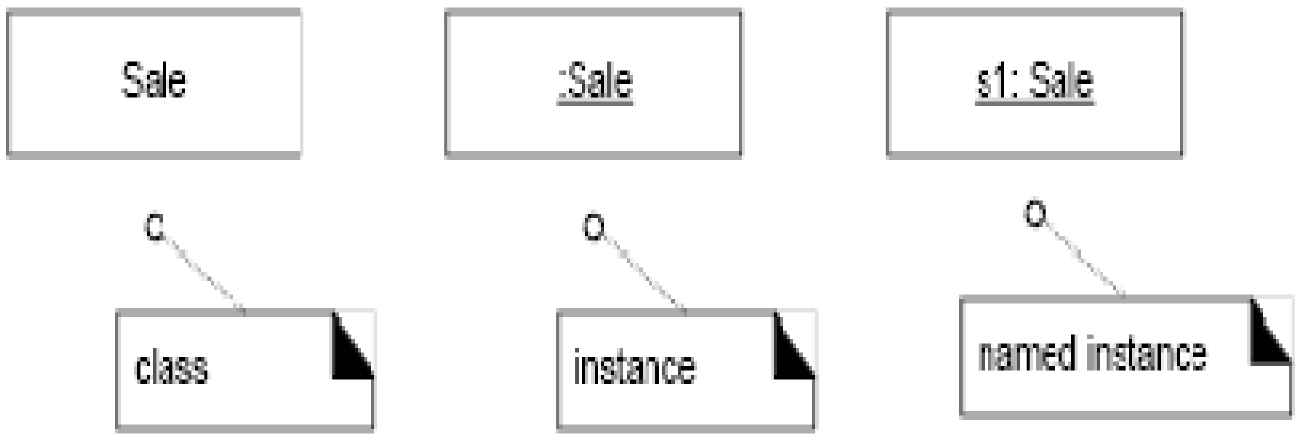


Fig: Example of collaboration diagram for makePayment



Hari Prasad Pokhrel (hpokhrel24@gmail.com)

Common Interaction Diagram Notation

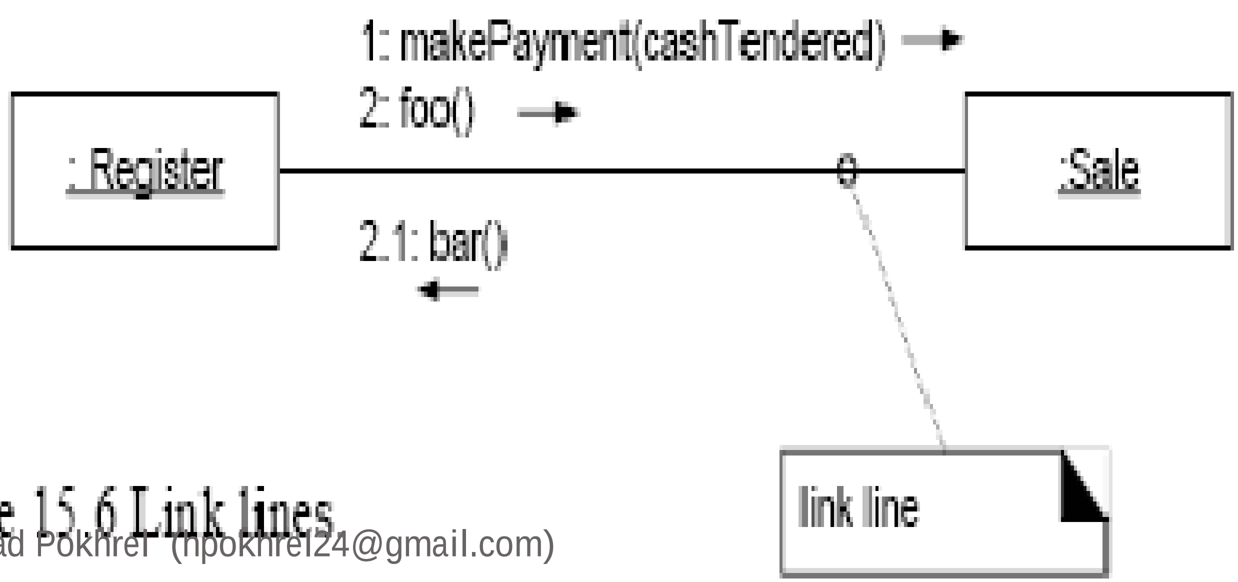


Basic Message Expression Syntax

- The UML has a standard syntax for message expressions:
- *return := message(parameter : parameterType) : returnType*
- Type information may be excluded if obvious or unimportant. For example:
- spec := getProductSpect(id)
- spec := getProductSpect(id:ItemID)
- spec := getProductSpect(id:ItemID) ProductSpecification

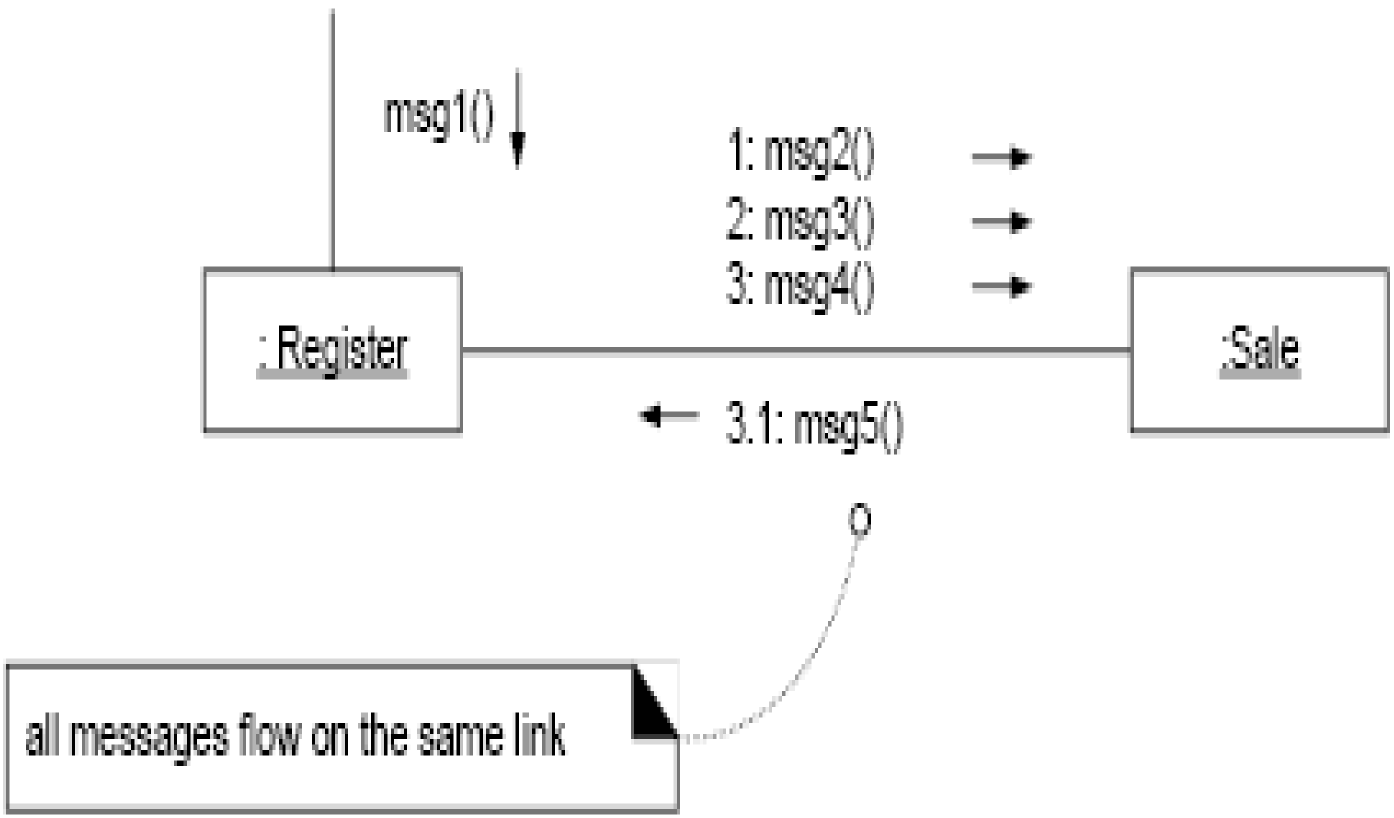
Basic Collaboration Diagram Notation

- ***Links***
  - **A link** is a connection path between two objects; it indicates some form of navigation and visibility between the objects is possible . More formally, a link is an instance of an association. For example, there is a link.or path of navigation.from a *Register* to a *Sale,* along which messages may flow, such as the *makePayment* message

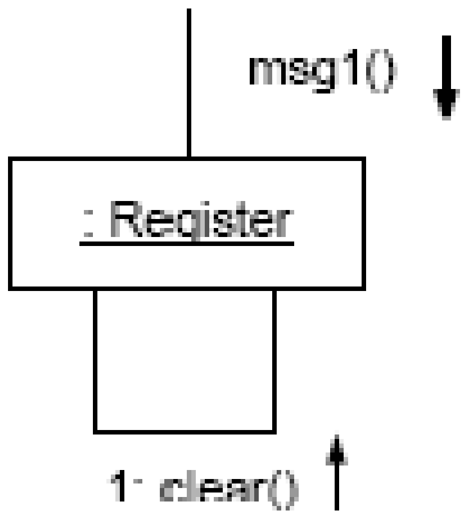


gure 15.6 Link lines

Messages

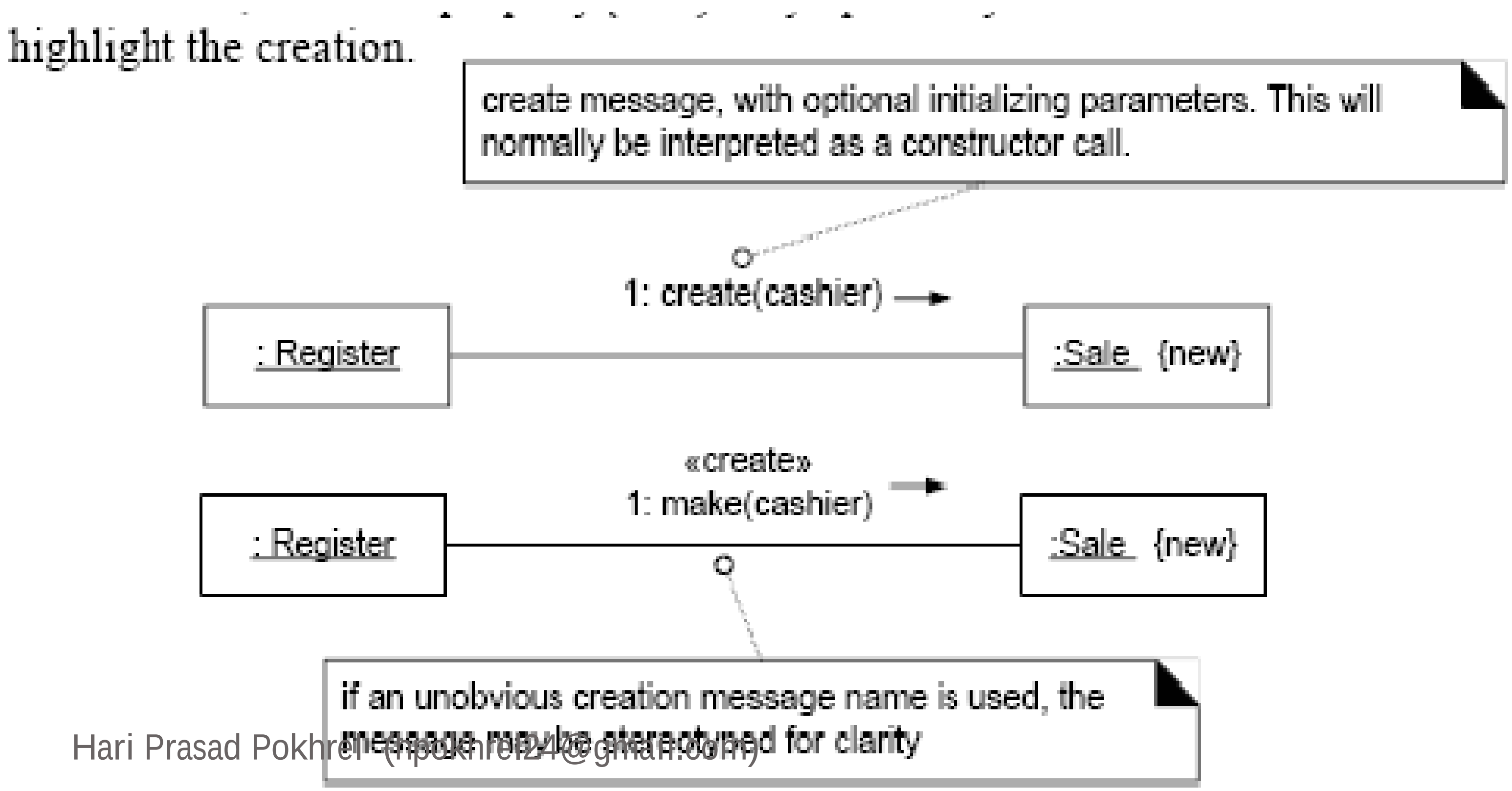

msg1()
1: msg2()
2: msg3()
3: msg4()
: Register
:Sale
3.1: msg5()
all messages flow on the same link

Messages to "self" or "this"



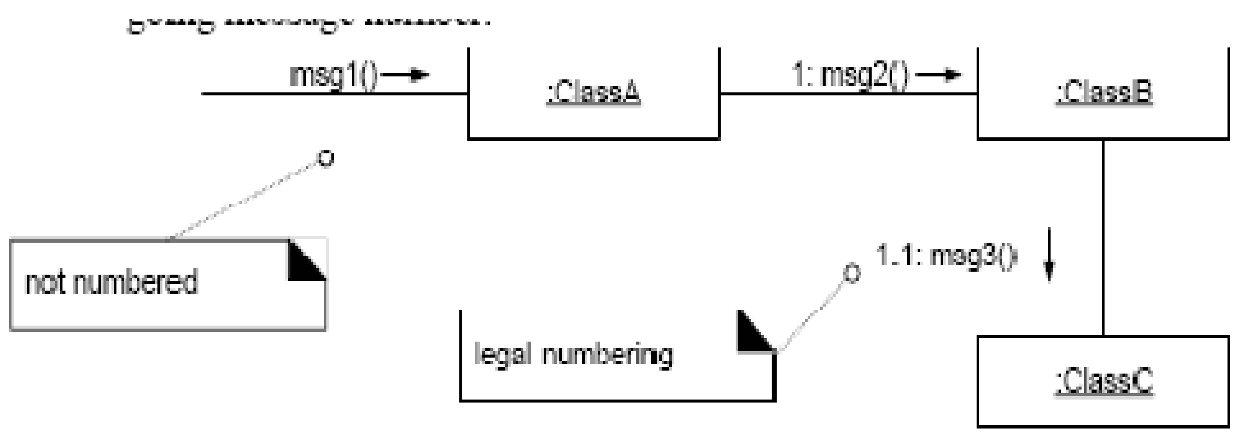
Creation of Instances

- Any message can be used to create an instance, but there is a convention in the UML to use a message named *create* for this purpose. If another message name is used, the message may be annotated with a special feature called a UML stereotype, like so: *«create».*
- The *create* message may include parameters, indicating the passing of initial values. Furthermore, the UML property *{new}* may optionally be added to the instance box to highlight the creation.



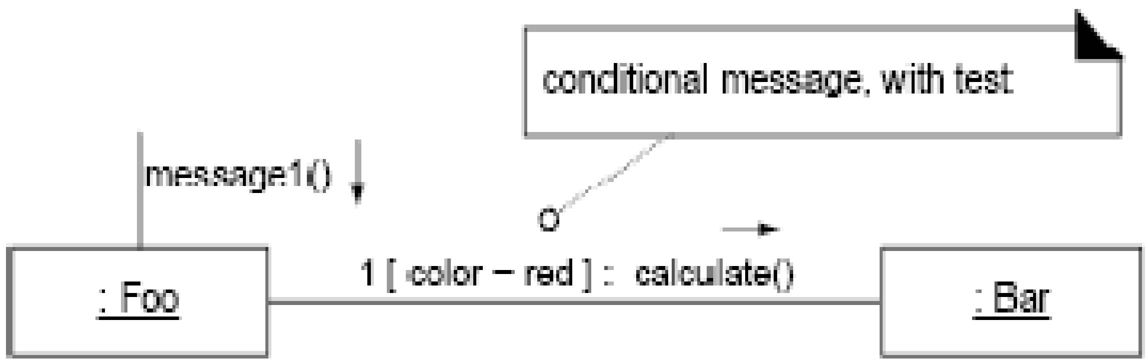
Message Number Sequencing

- The order of messages is illustrated with **sequence numbers**
  1. The first message is not numbered. Thus, *msg1()* is unnumbered.
  2. The order and nesting of subsequent messages is shown with a legal numbering scheme in which nested messages have a number appended to them.
- Nesting is denoted by prepending the incoming message number to the out going message number.



Conditional Messages:

- A conditional message is shown by following a sequence number with a conditional clause in square brackets, similar to an iteration clause. The message is only sent if the clause evaluates to *true.*



Mutually Exclusive Conditional Paths

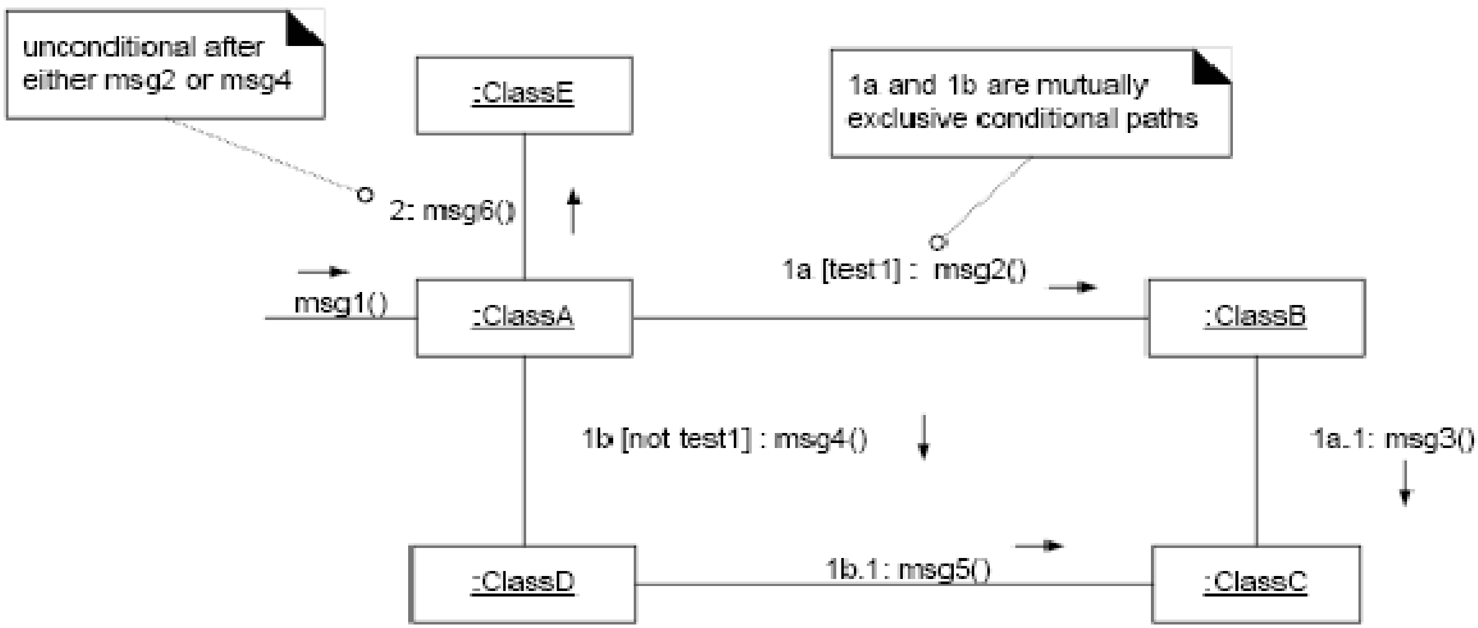


Figure 15.13 Mutually exclusive messages.

Iteration or Looping

- Iteration notation is shown in Figure below. If the details of the iteration clause
- are not important to the modeler, a simple '*' can be used.

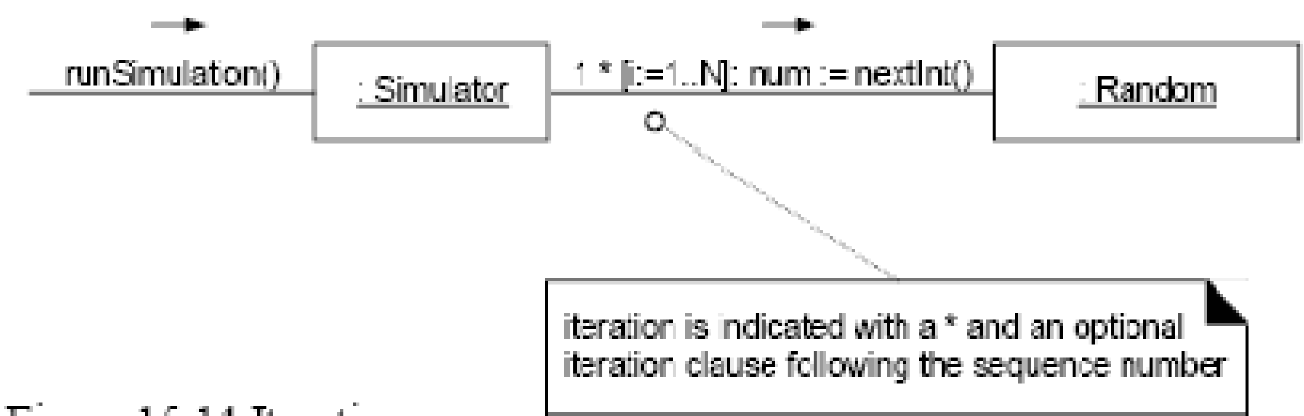
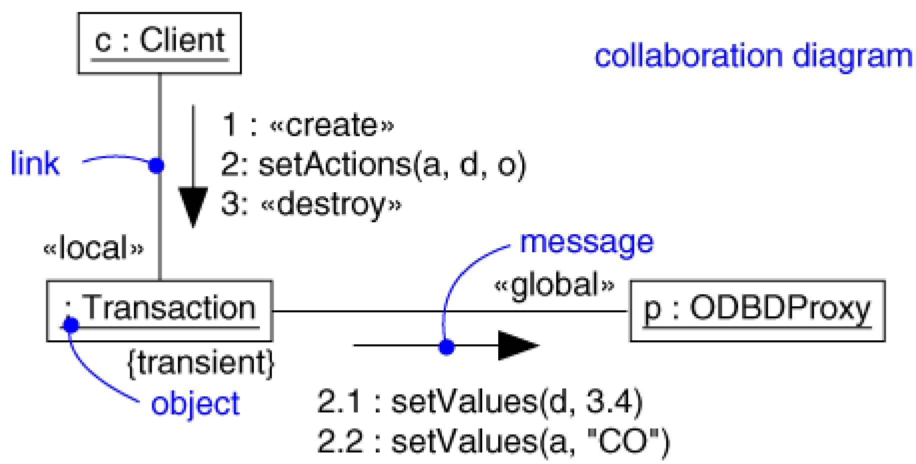


Figure 15.14 Iteration.

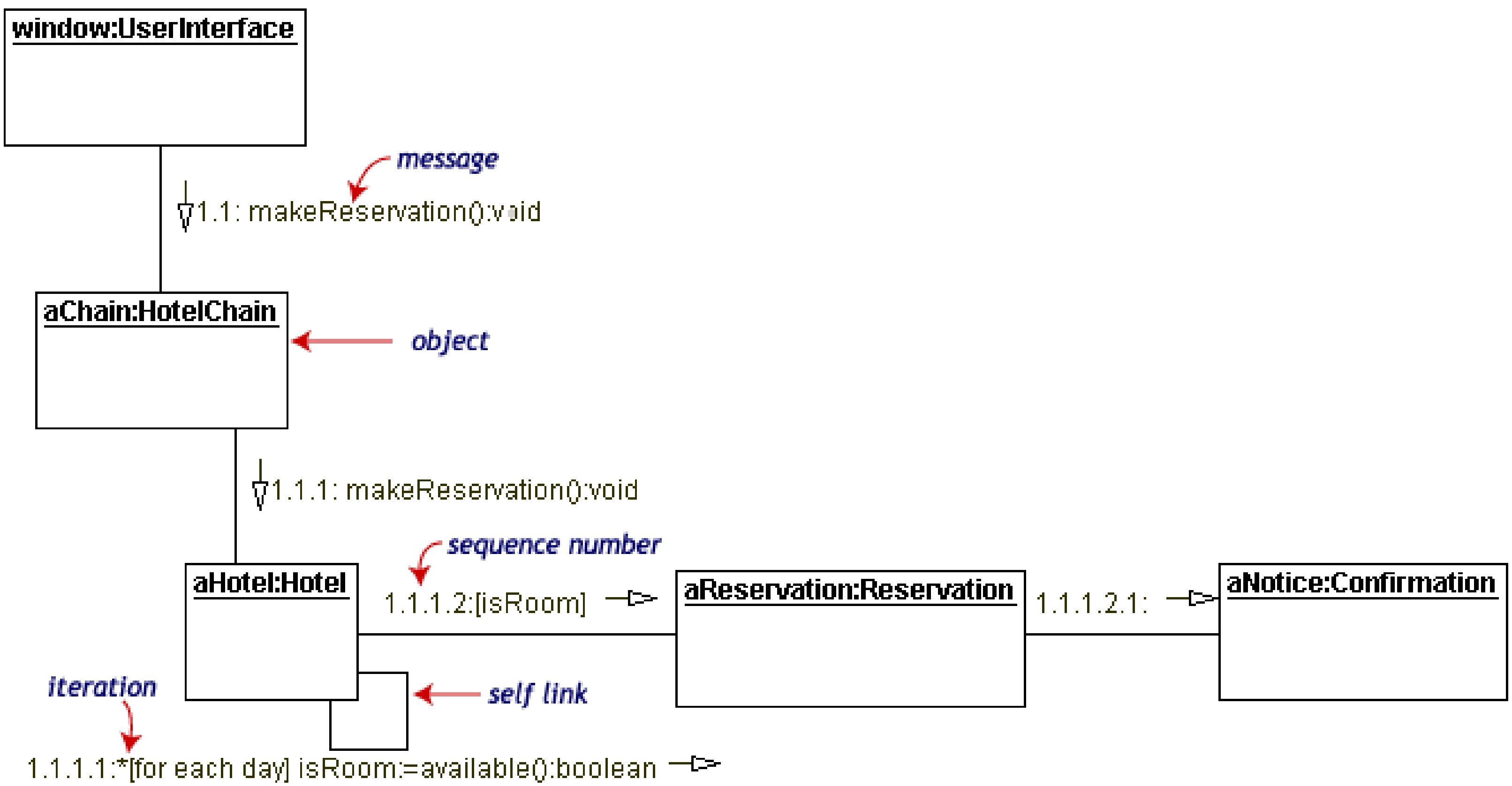
Collaboration Diagram

- Captures Dynamic Behavior (Message-Oriented)

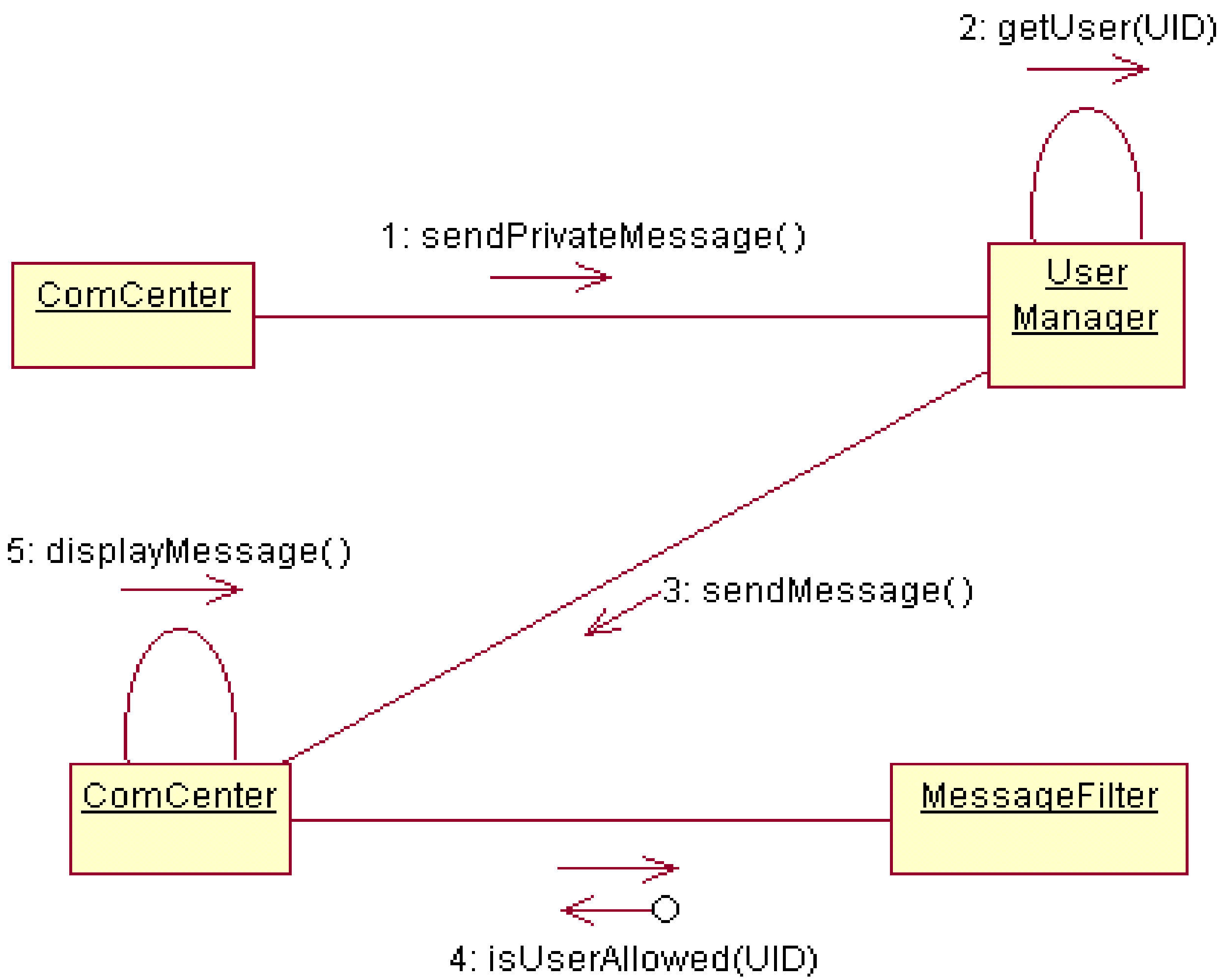


Collaboration Diagram

- Convey Same Info as Sequence Diagrams but Focus on Object Roles



Collaboration Diagram



Dynamic: Statechart Diagram

- Statechart Diagrams: Tracks the States that an Object Goes Through
- Captures Dynamic Behavior (Event-Oriented)
- Purposes:
  - Model Object Lifecycle
  - Model Reactive Objects (User Interfaces, Devices, etc.)
  - Are Statecharts Complex FSMs?
  - Sequence::Time vs. Collaboration::Message vs. Statechart::Event
- Main Concepts: State, Event, Transition, Action

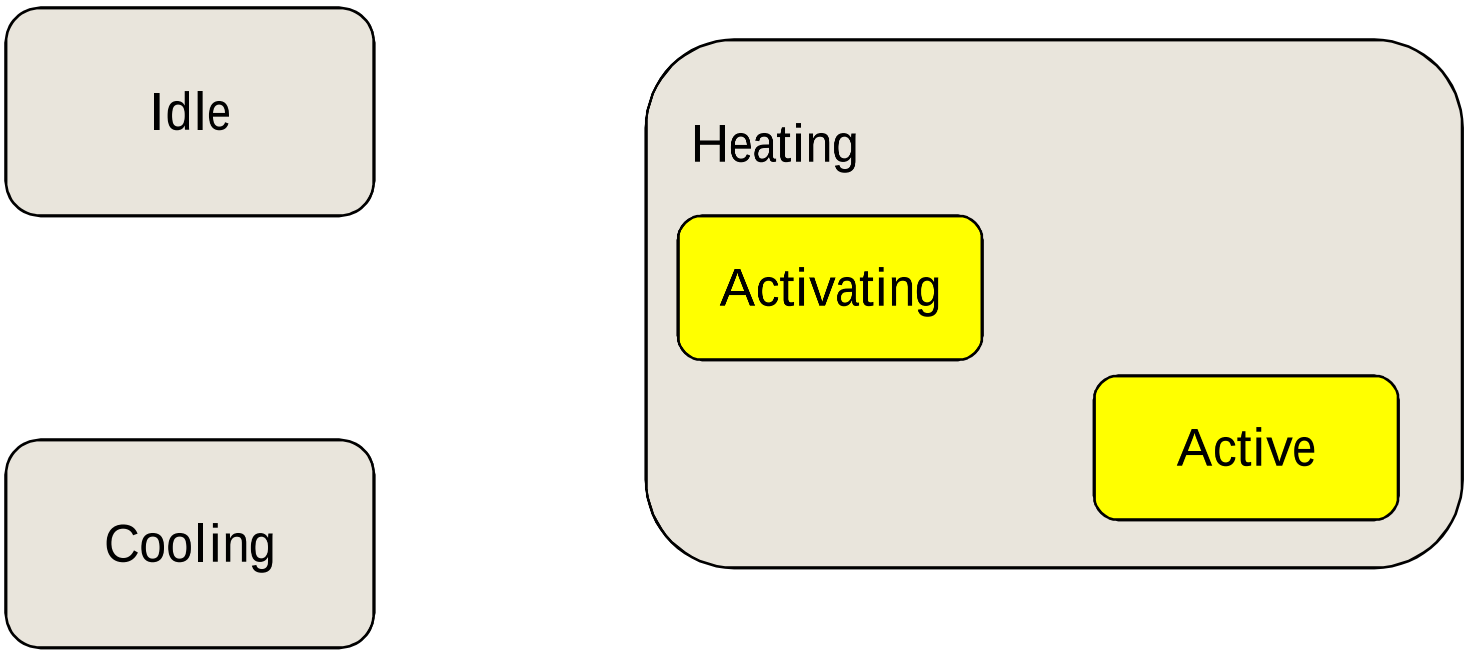
State, Event, and Signal

A state is a condition in which an object can reside during its lifetime while it satisfies some condition, performs an activity, or waits for an event.

An event is a significant occurrence that has a location in time and space.

A signal is an asynchronous communication from one object to another.

State Notation



State Machine and Transition

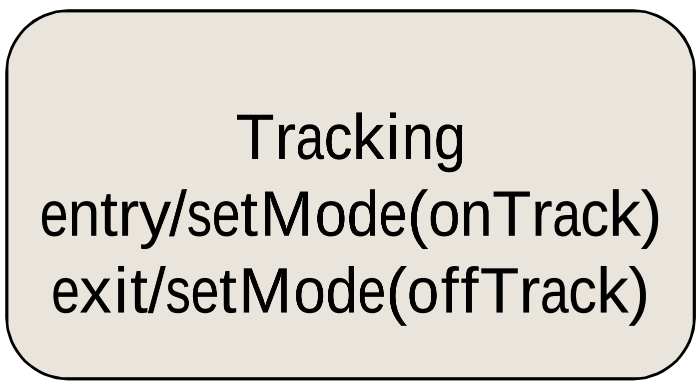
A state machine is a behavior that specifies the sequences of states that an object goes through in its lifetime, in response to events, and also its responses to those events.

A transition is a relationship between two states; it indicates that an object in the first state will perform certain actions, then enter the second state when a given event occurs.

Entry and Exit Actions

An entry action is the first thing that occurs each time an object enters a particular state.

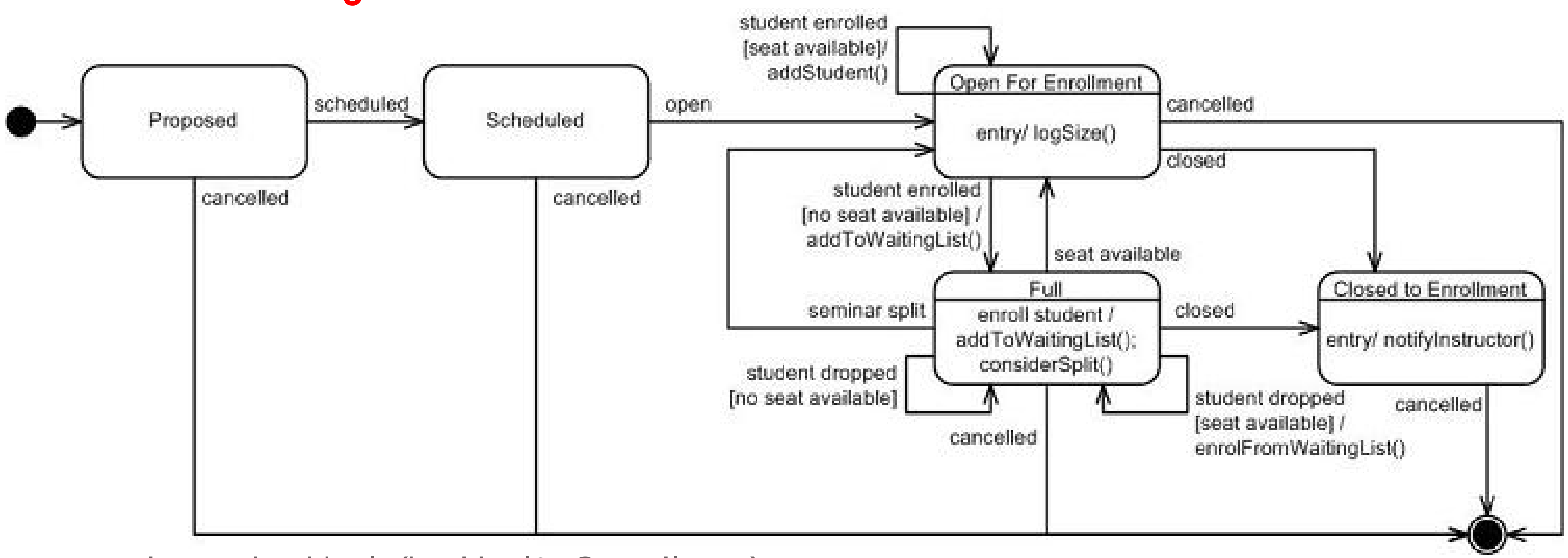
An exit action is the last thing that occurs each time an object leaves a particular state.



Statechart

- Use state diagrams to demonstrate the behavior of an object through many use cases of the system. Only use state diagrams for classes where it is necessary to understand the behavior of the object through the entire system.

- Seminar Registration

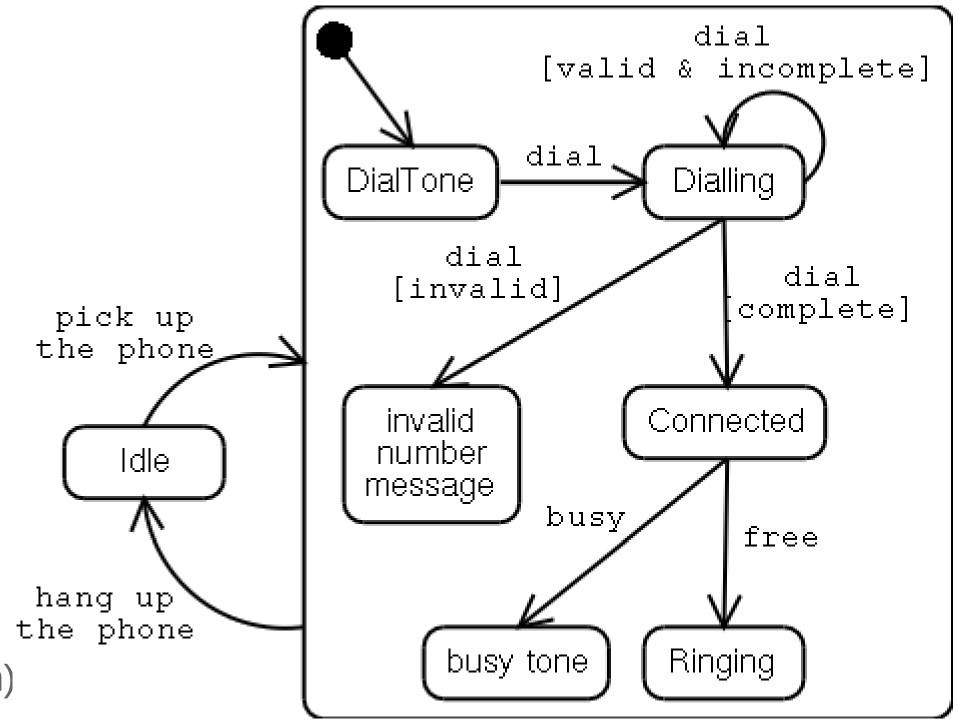


Statechart

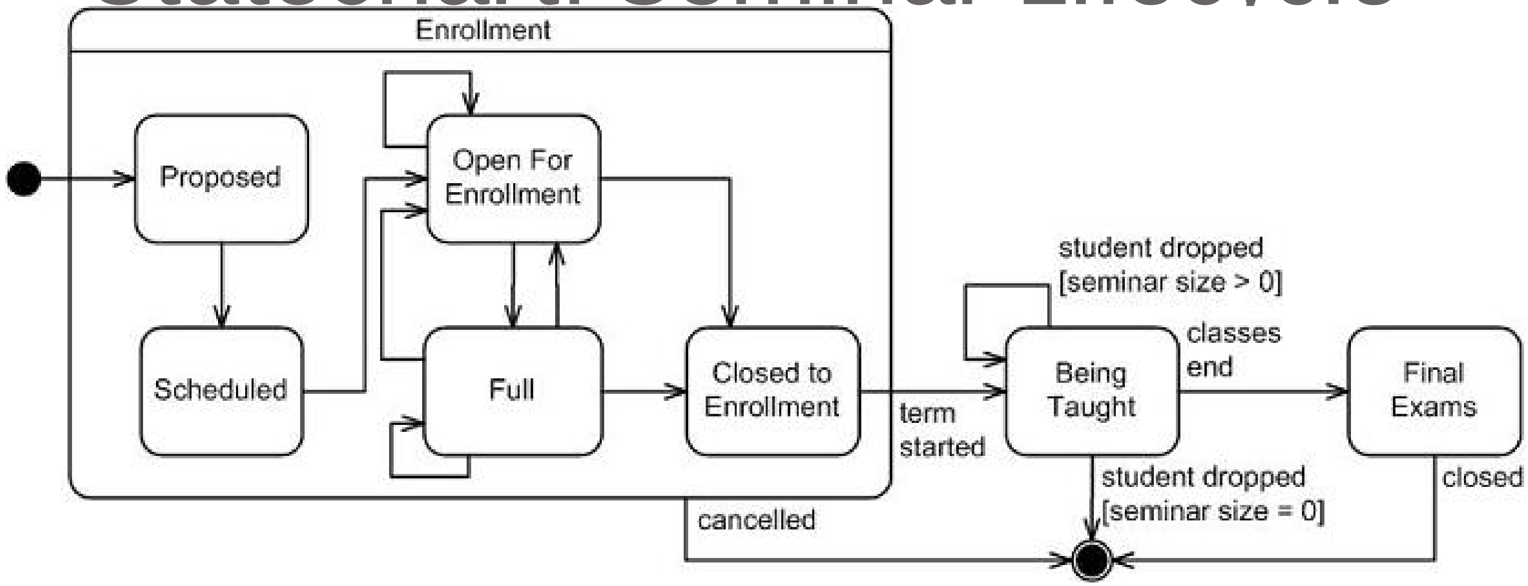
- Following are the main purposes of using Statechart diagrams.
  - To model dynamic aspect of a system.
  - To model life time of a reactive system.
  - To describe different states of an object during its life time.
  - Define a state machine to model states of an object.

Statechart

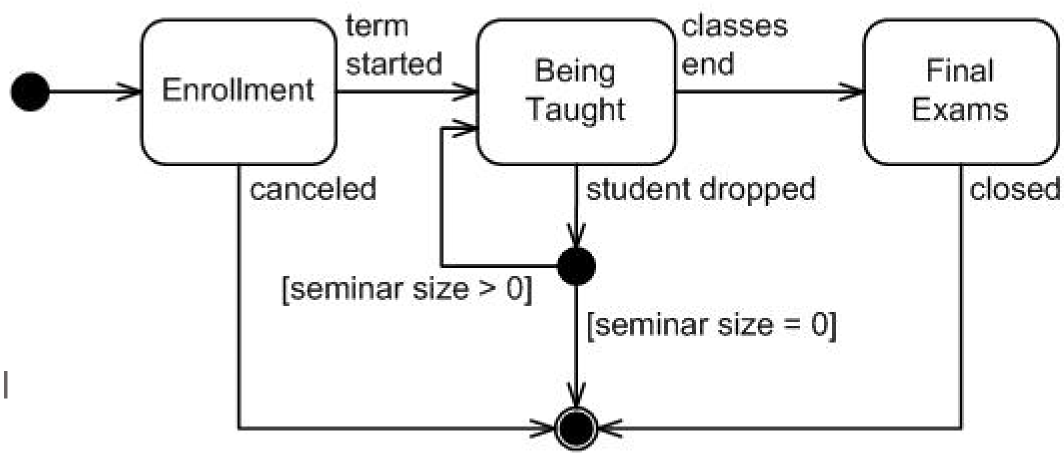
- Before drawing a Statechart diagram we must have clarified the following points:
  - Identify important objects to be analyzed.
  - Identify the states.
  - Identify the events.



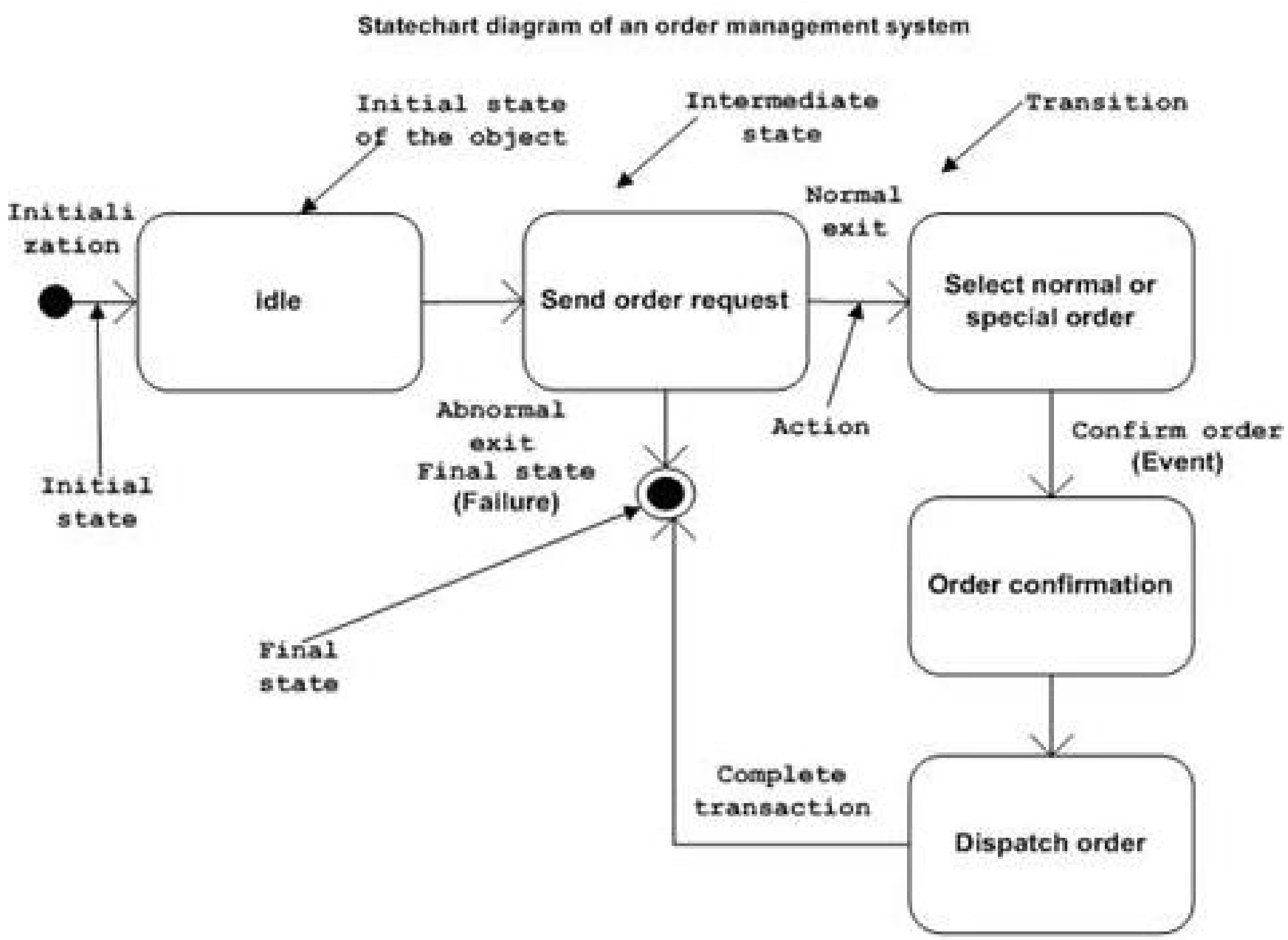
Statechart: Seminar Lifecvcle



- Top Level State Machine of Seminar



Statechart Diagram



Hari Prasad Pokhrel (hpokhrel24@gmail.com)

Activities

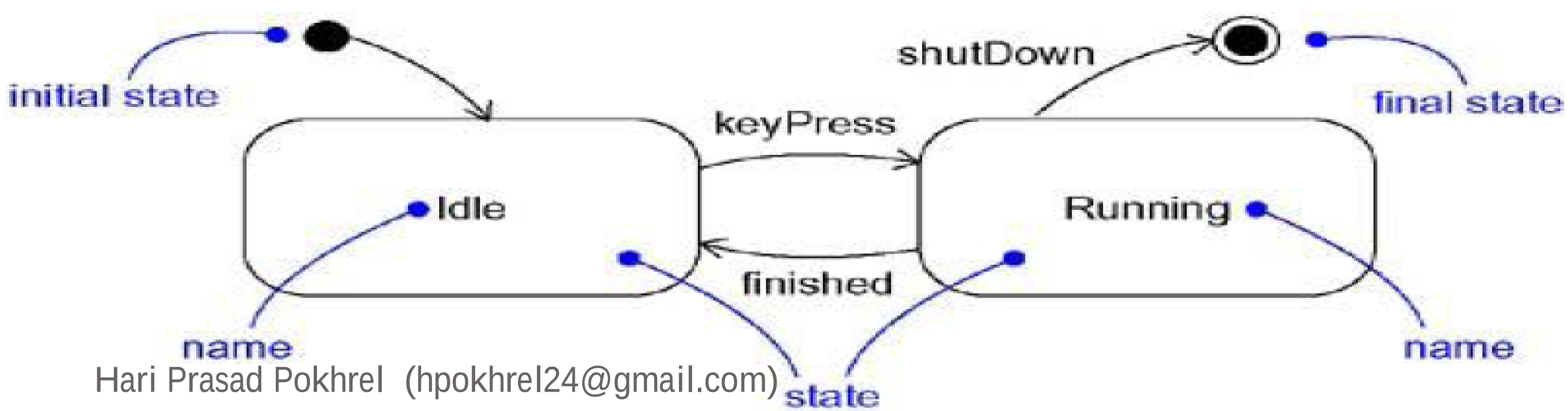
An activity is an interruptible sequence of actions that an object can perform while it resides in a given state.

State

- 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).

State

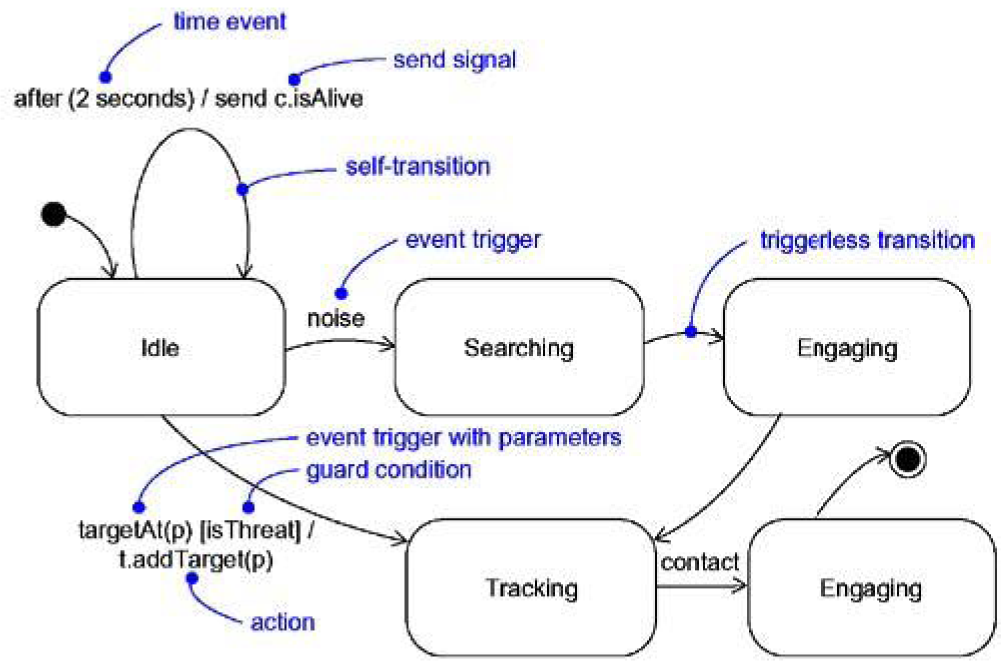
1. Name	A textual string that distinguishes the state from other states; a state may be anonymous, meaning that it has no name
2. Entry/exit actions	Actions executed on entering and exiting the state, respectively
3. Internal transitions	Transitions that are handled without causing a change in state
4. Substates	The nested structure of a state, involving disjoint (sequentially active) or concurrent (concurrently active) substates
5. Deferred events	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



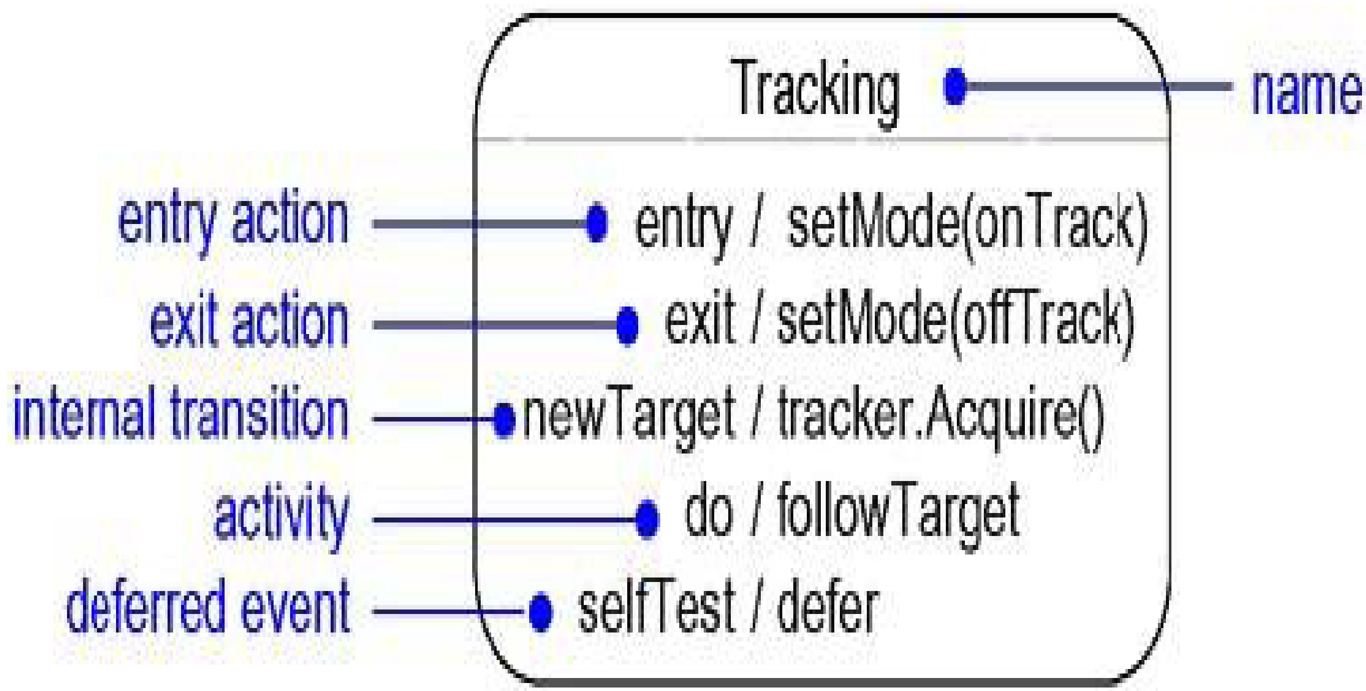
Transitions

- 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.
- On such a change of state, the transition is said to fire. Until the transition fires, the object is said to be in the source state; after it fires, it is said to be in the target state.
- For example, a Heater might transition from the Idle to the Activating state when an event such as tooCold (with the parameter desiredTemp) occurs.

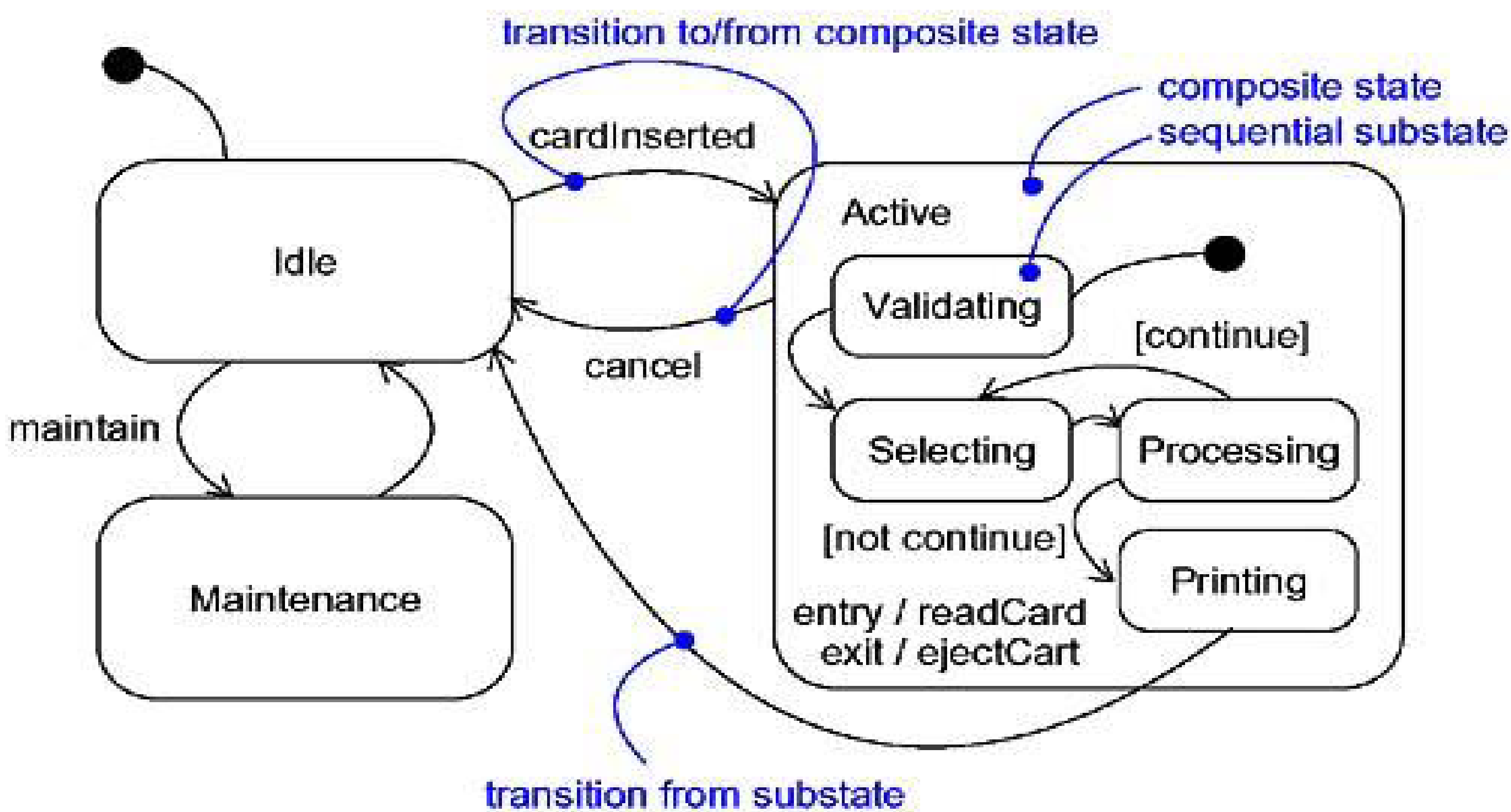
Transitions



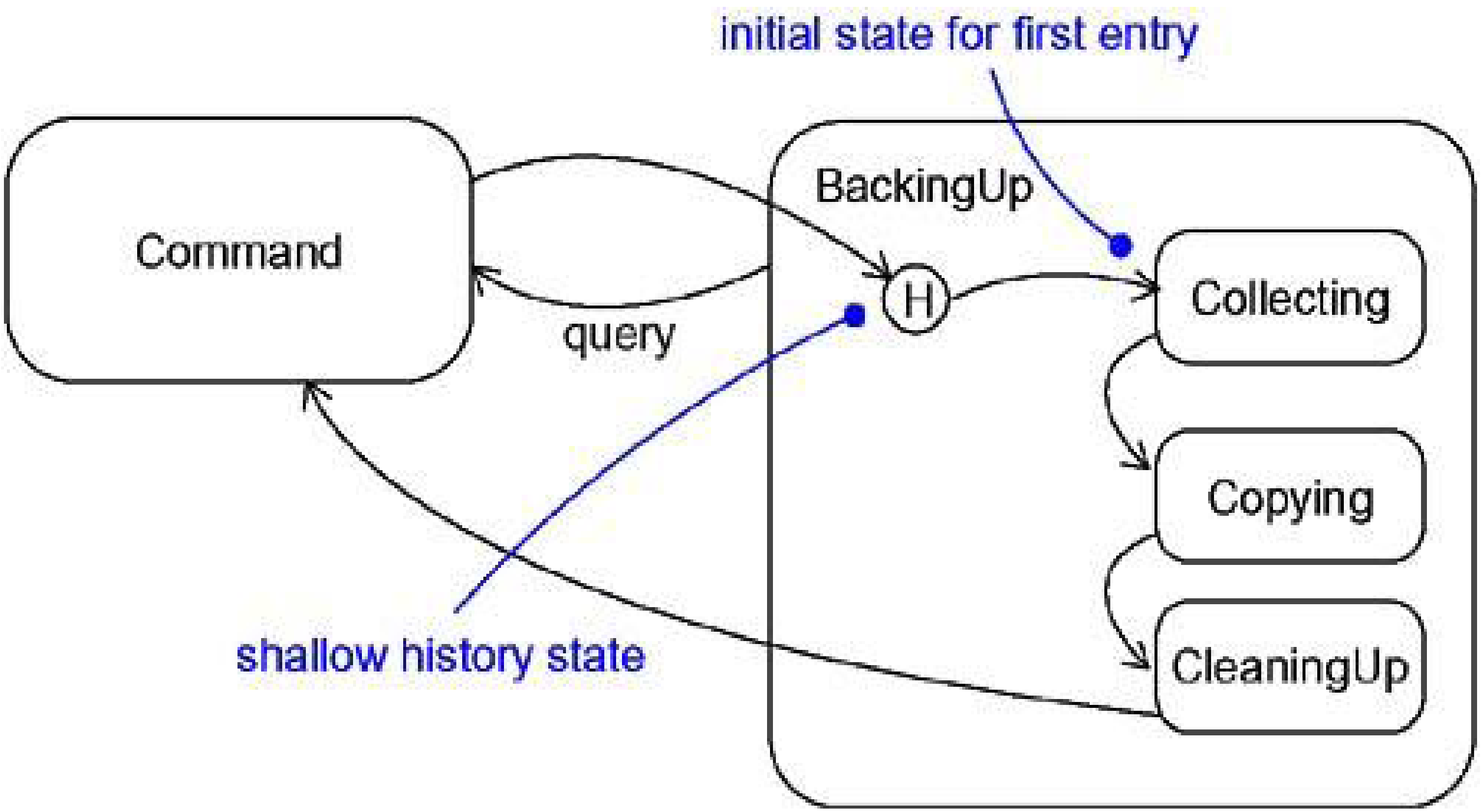
Advanced States and Transitions



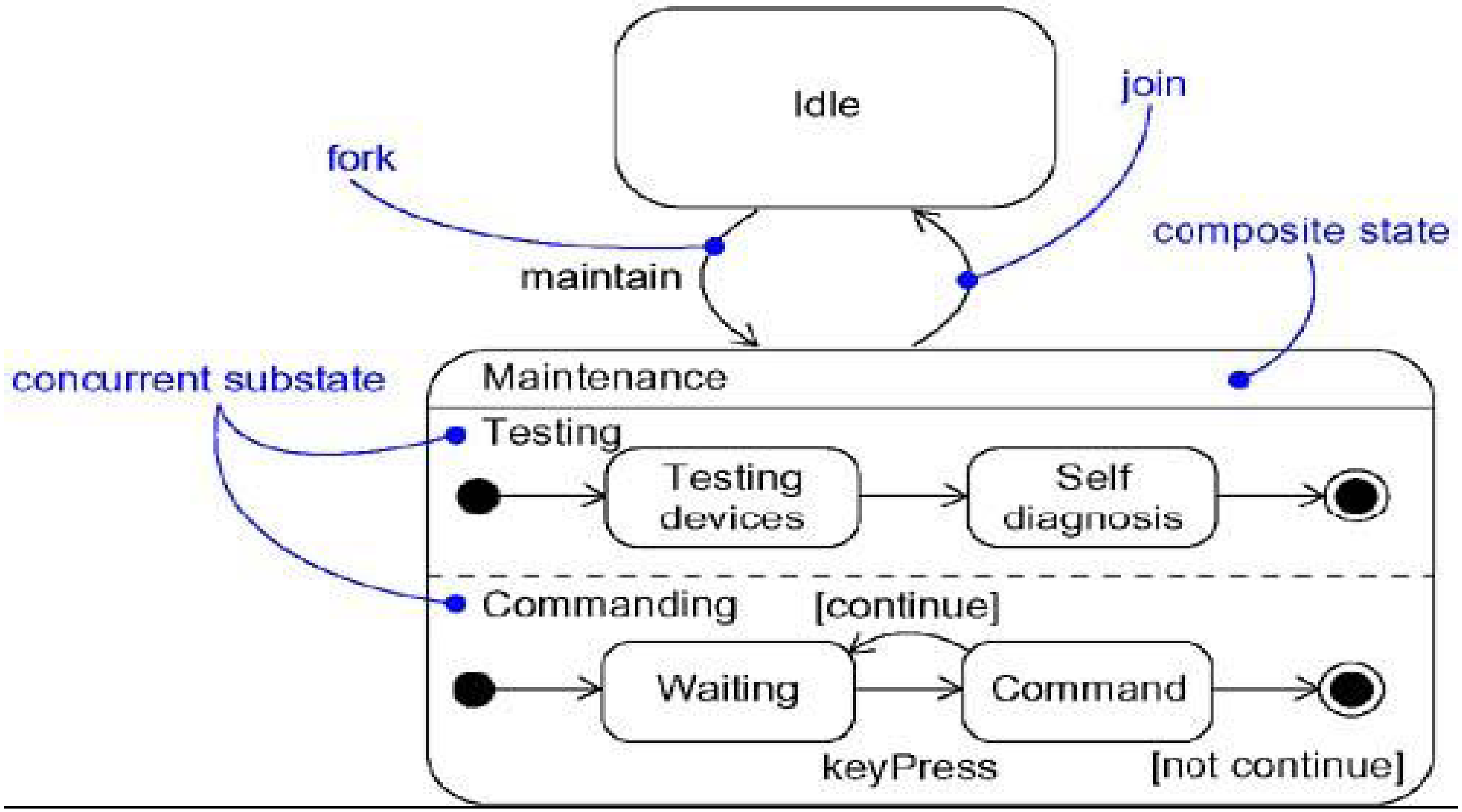
Substates


transition to/from composite state
composite state
sequential substate
cardInserted
Idle
Active
Validating
[continue]
cancel
maintain
Selecting
Processing
Maintenance
[not continue]
Printing
entry / readCard
exit / ejectCart
transition from substate

History States



Concurrent Sub states



State Transition Diagrams

State transition diagrams are a useful tool for constructing the individual classes. Specifically, they aid in two important ways in "fleshing out" the structure of the class:

1. method development -- State transition diagrams provide the "blueprints" for developing the algorithms that implement methods in the class

2. attribute identification – Attributes contain the state information needed for regulating the behaviors of the instances of the class

When constructing state transition diagrams, take care to ensure that the post-conditions stipulated in the contracts are enforced.

State Transition Diagrams

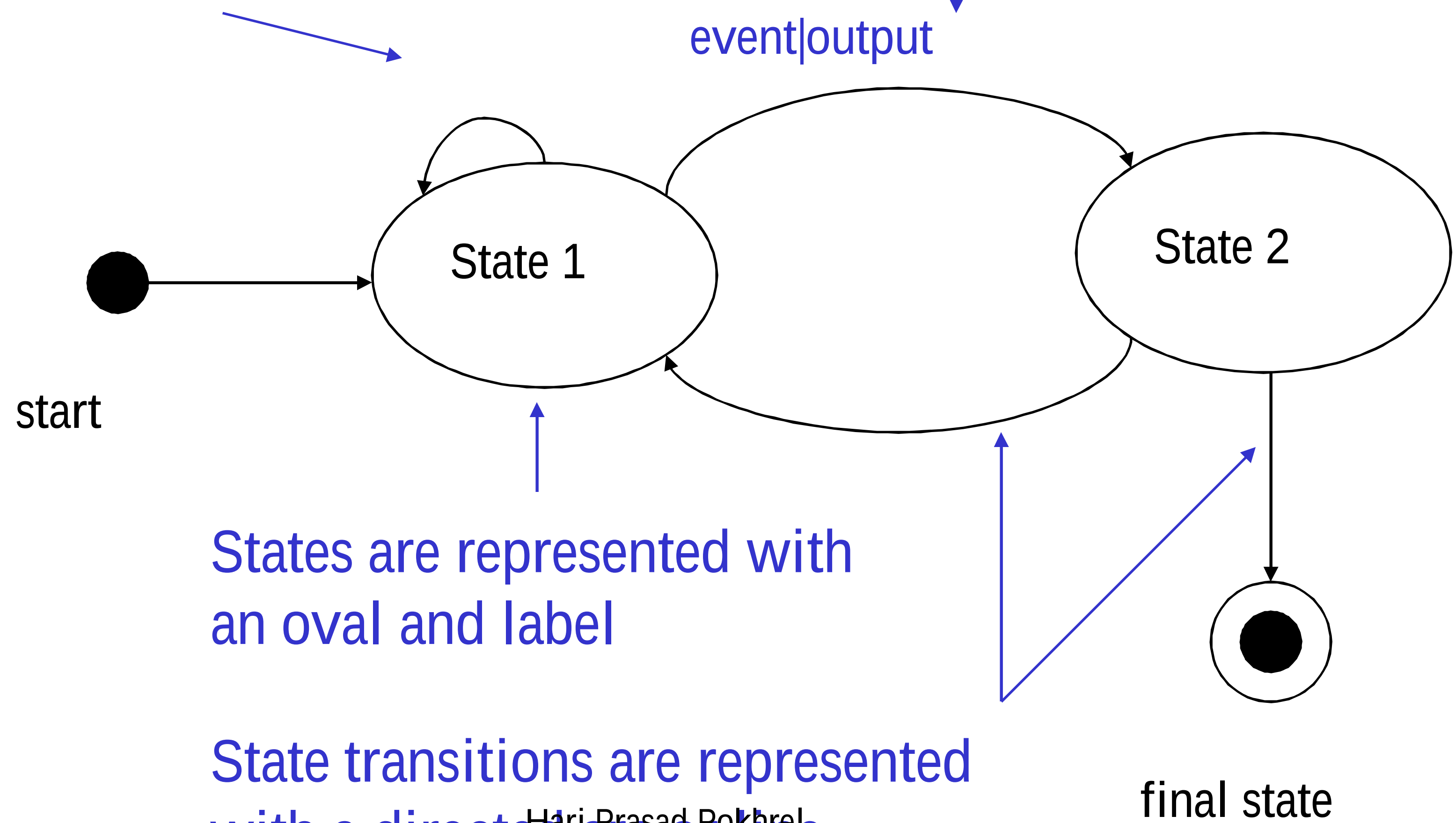
- These diagrams are used to model the entire life cycle of an object.
- State of an object is defined as the condition where an object resides for a particular time and the object again moves to other states when some events occur.
- State chart diagrams are also used for forward and reverse engineering.

State Transition Diagrams

Notation

Some events do not trigger a change in state

Transitions are labeled with the triggering event and the output if any



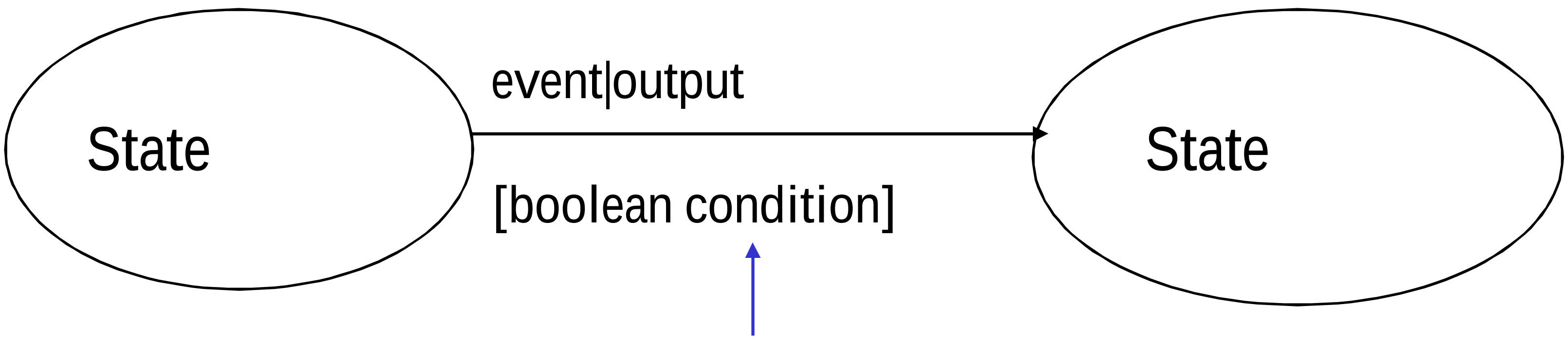
States are represented with an oval and label

State transitions are represented with a directed arc or line

Hari Prasad Pokhrel
(hpokhrel24@gmail.com)

State Transition Diagrams

Additional Notation

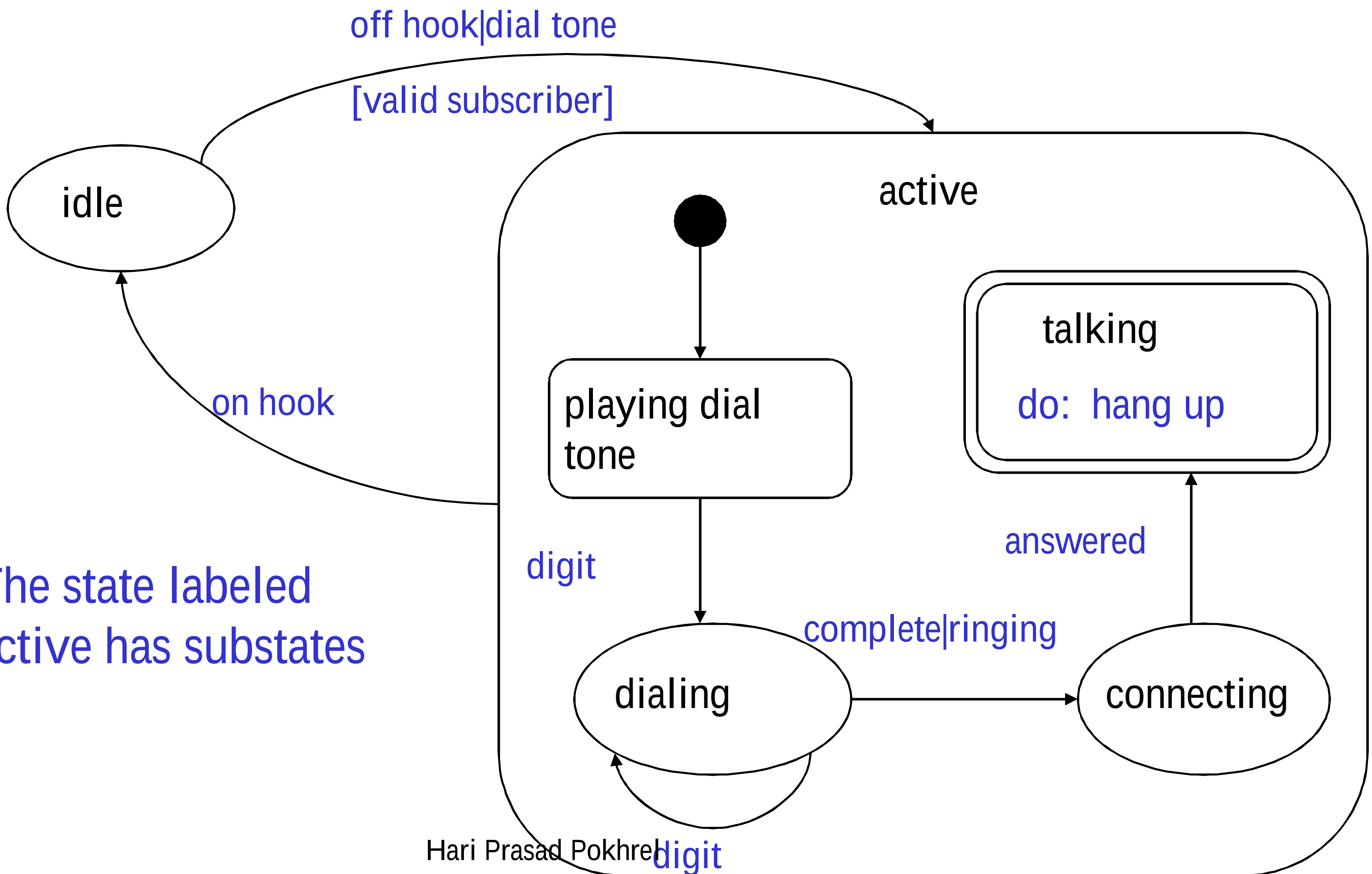


Guard condition – transition occurs only if condition is true

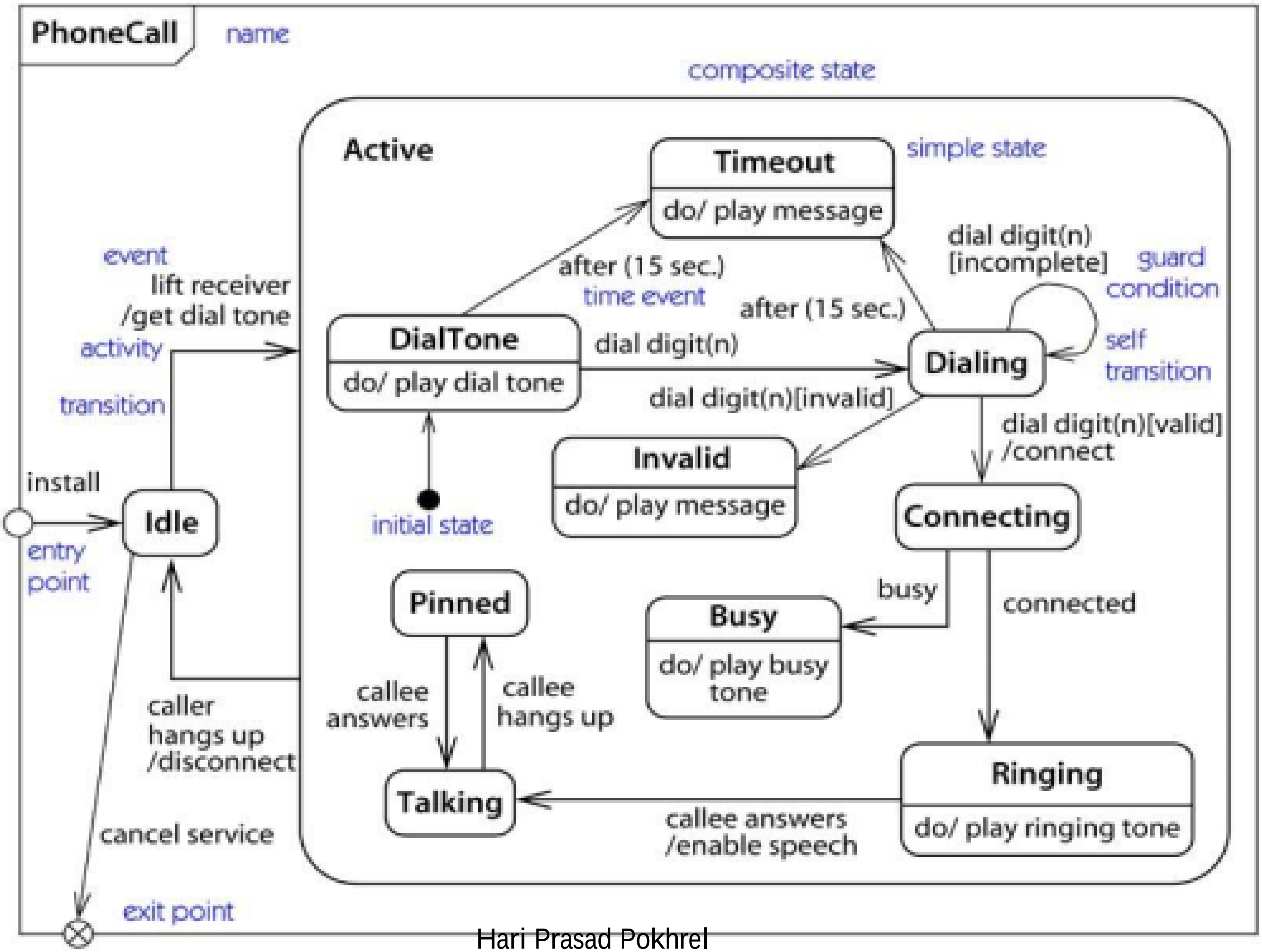
Hari Prasad Pokhrel
(hpokhrel24@gmail.com)

State Transition Diagrams

Example – Nested States in Telephone Call



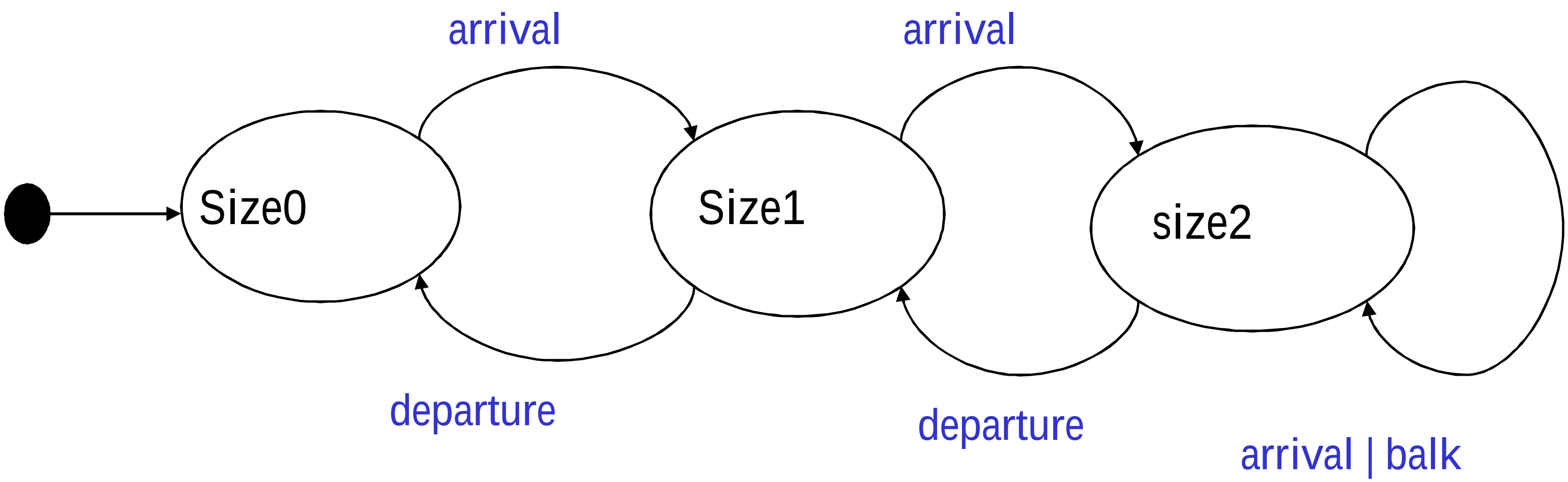
The state labeled active has substates



Hari Prasad Pokhrel
(hpokhrel24@gmail.com)

State Transition Diagrams

Second Example – A Queue of Capacity Two

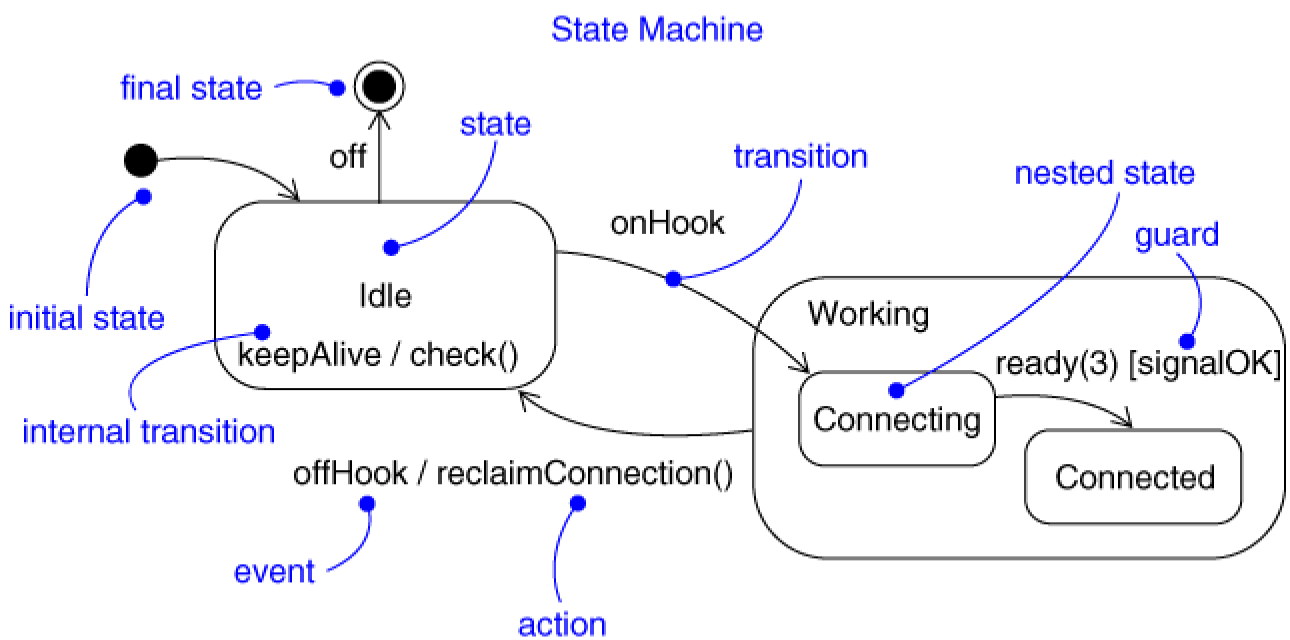


The queue has three states that indicate its number of occupants. When the queue is full, new arrivals cannot enter, and must leave the system.

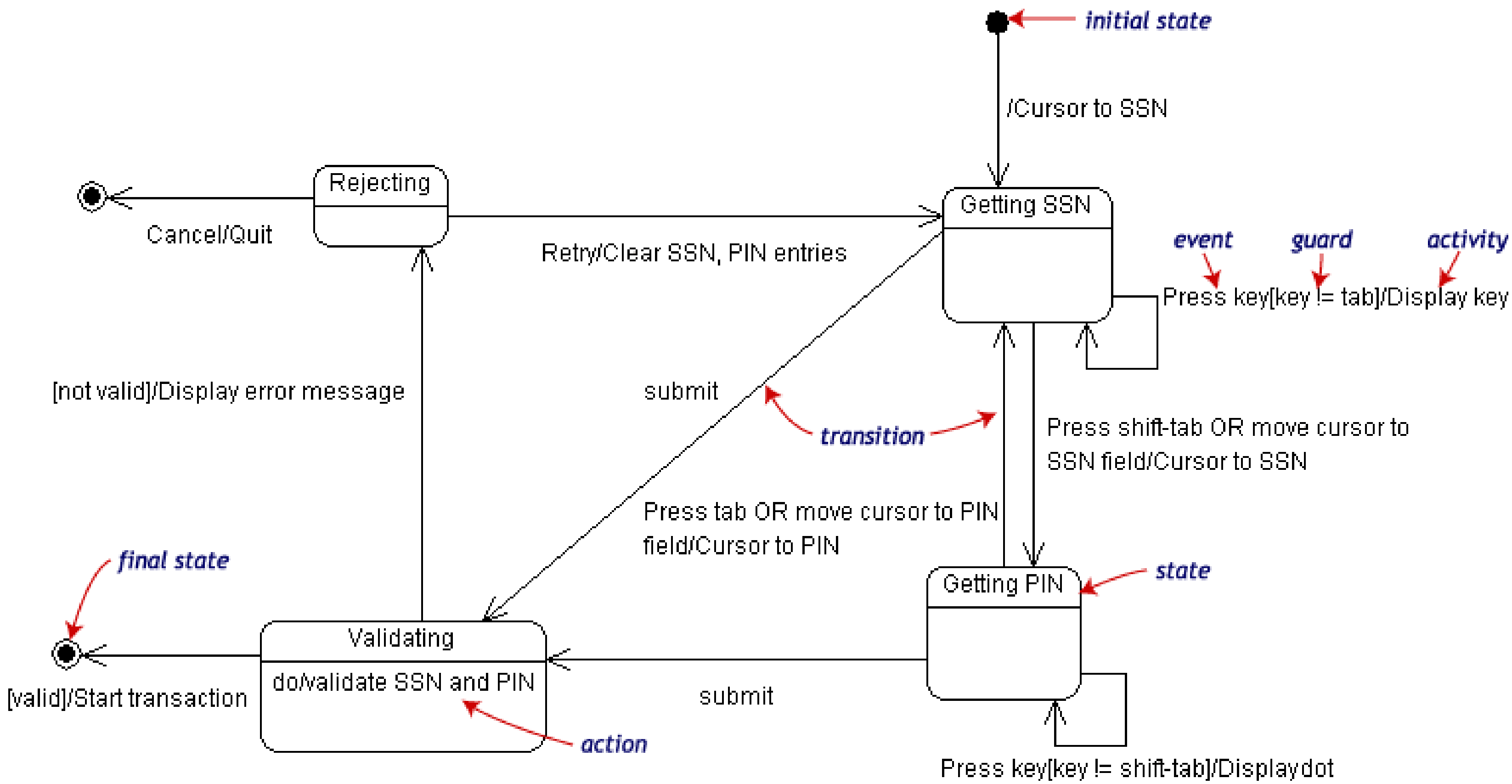
Hari Prasad Pokhrel
(hpokhrel24@gmail.com)

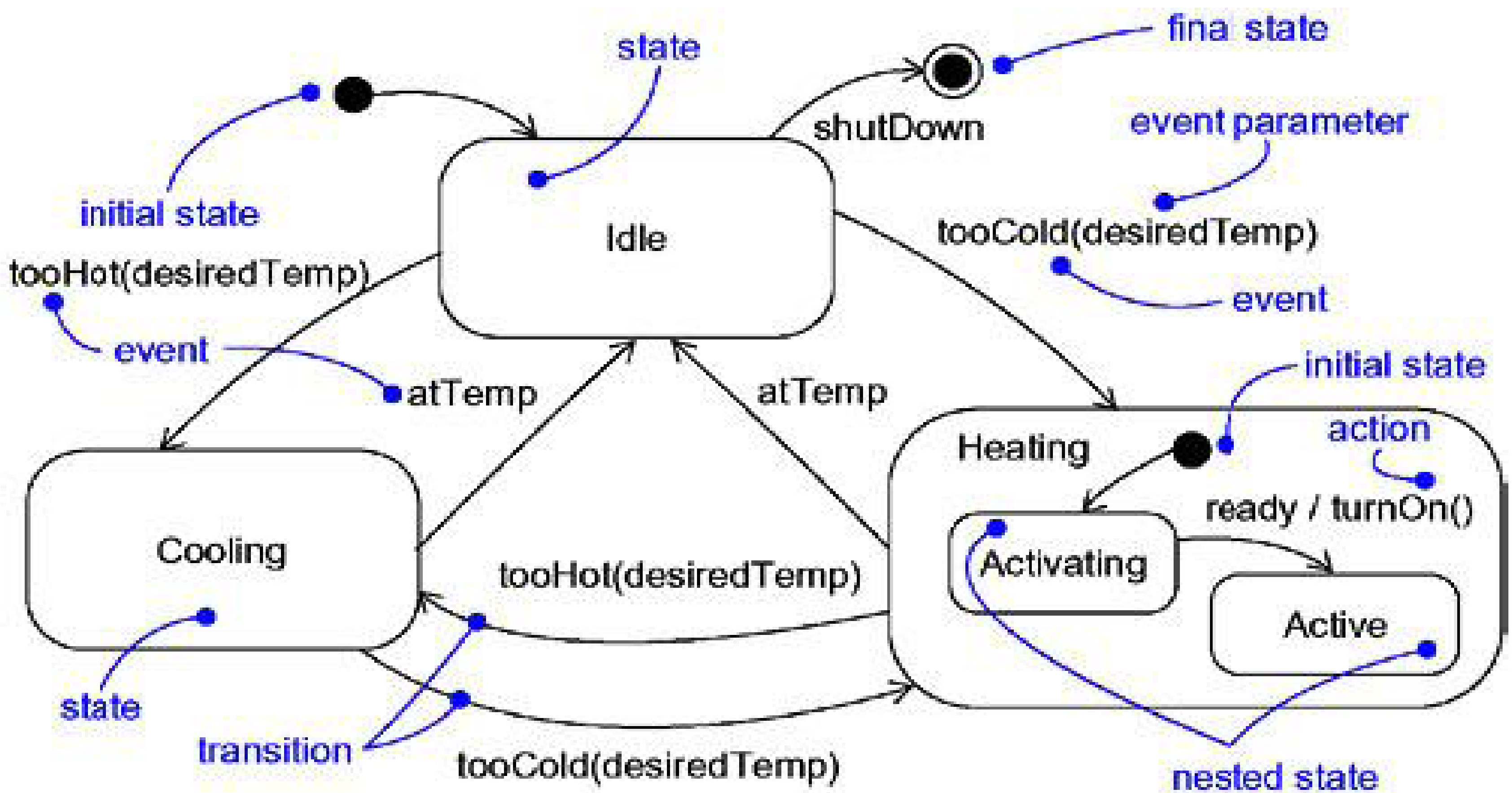
Statechart Diagram

- Captures Dynamic Behavior (Event-Oriented)



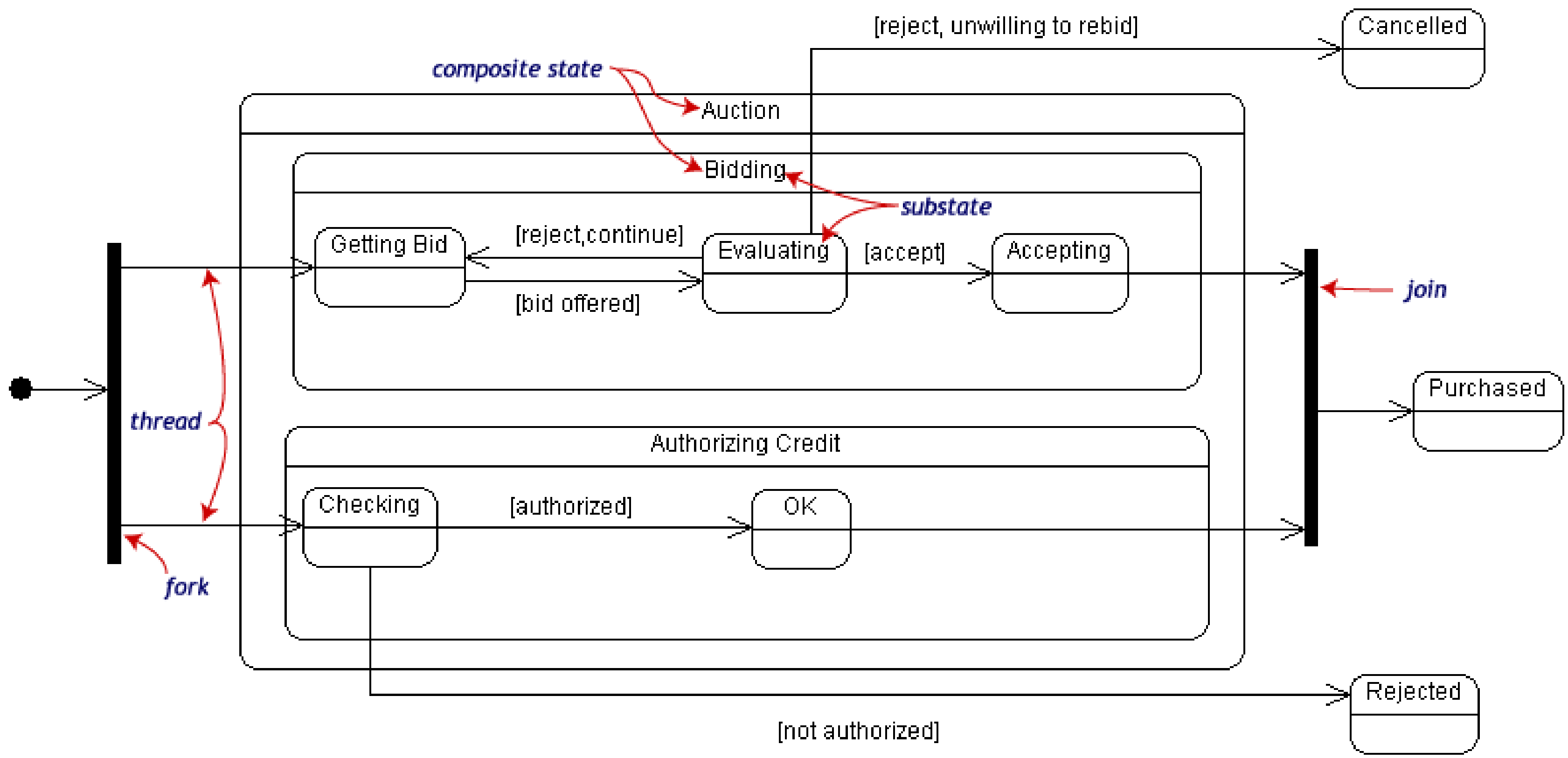
Statechart Diagram



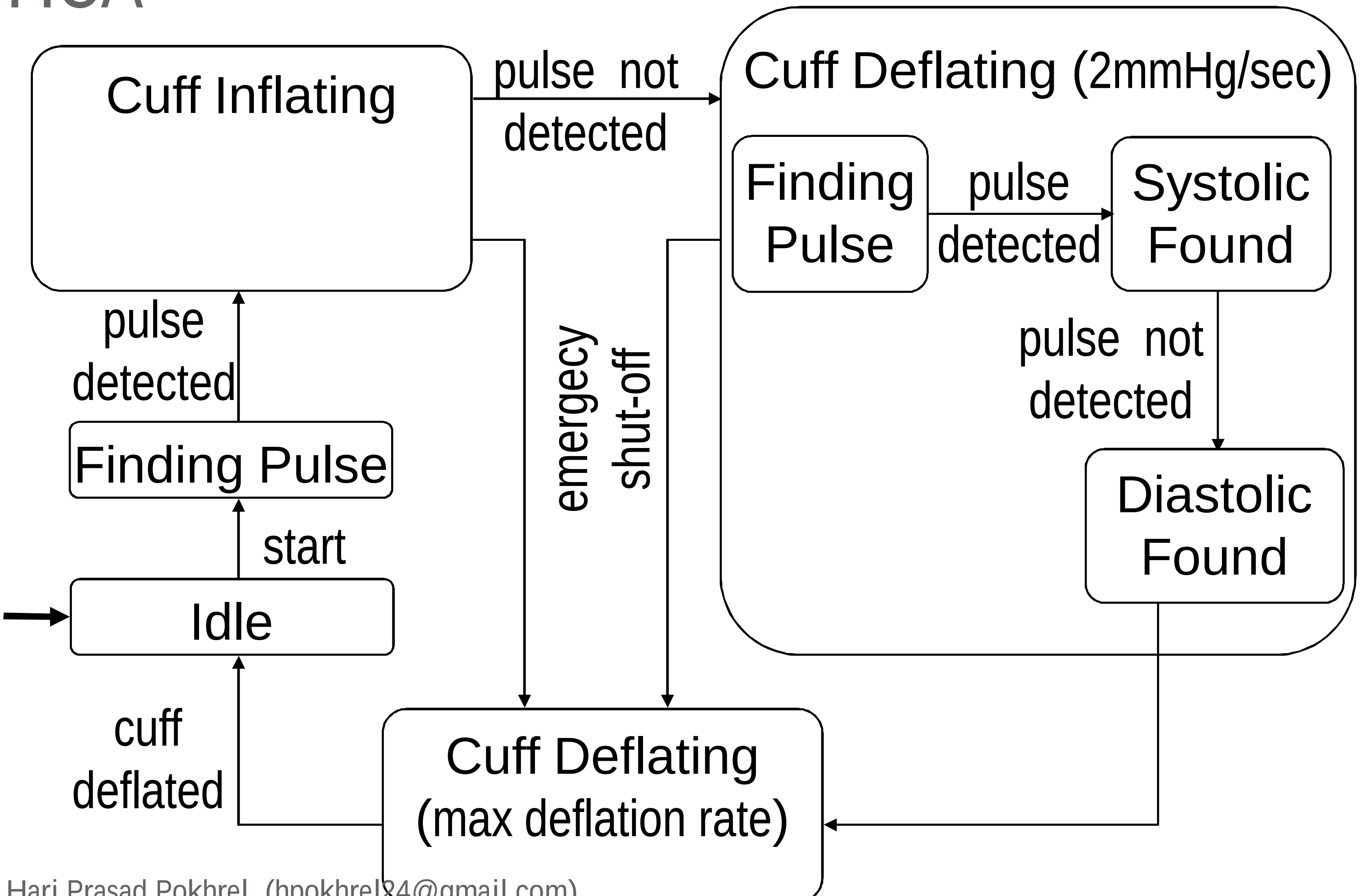

fina state
state
initial state
shutDown
event parameter
Idle
tooCold(desiredTemp)
tooHot(desiredTemp)
event
event
atTemp
atTemp
initial state
Heating
action
ready / turnOn()
Cooling
Activating
tooHot(desiredTemp)
Active
state
transition
tooCold(desiredTemp)
nested state

Statechart Diagram

- Composite States Illustrated
- Fork and Join Possible

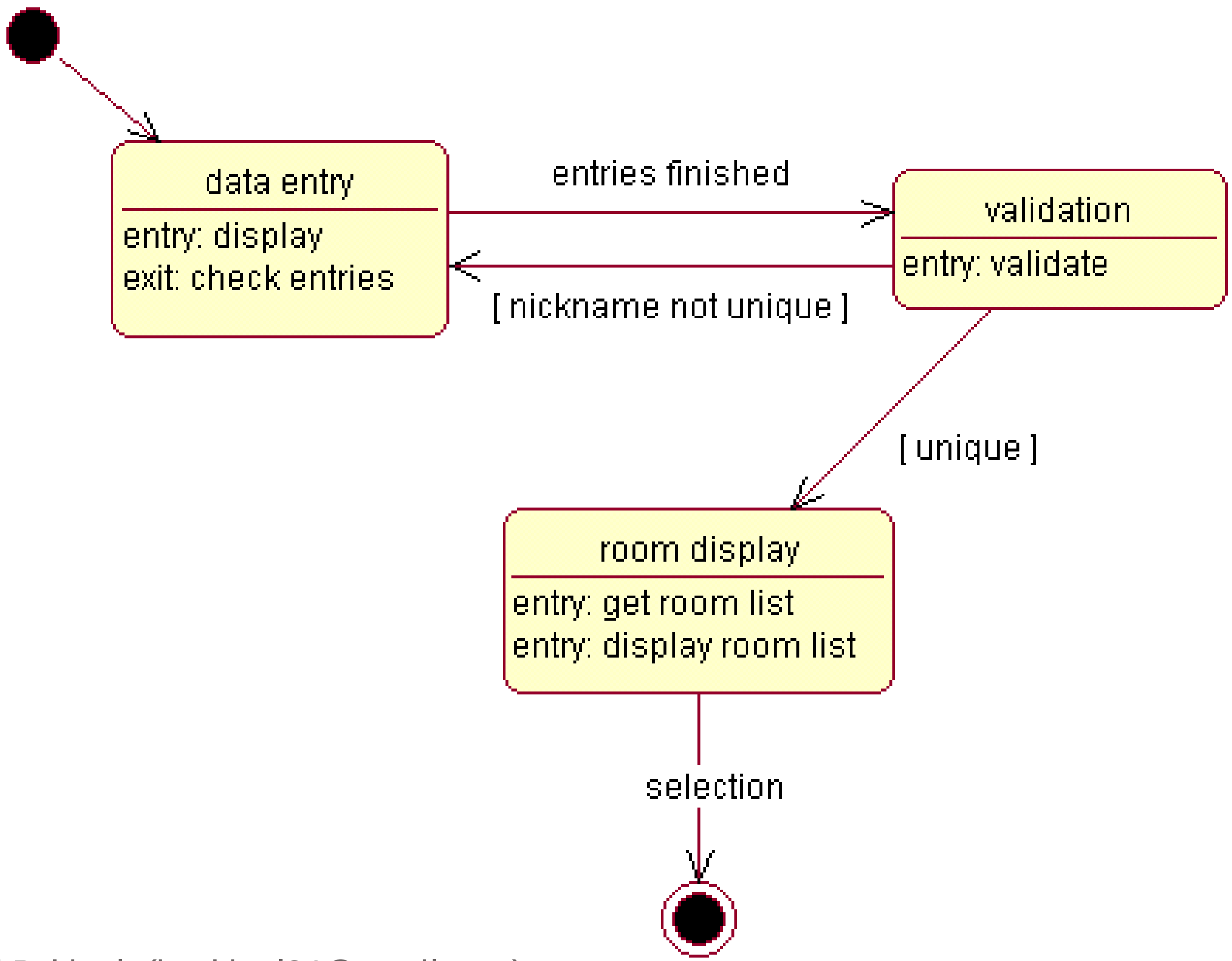


Statechart Diagram
HCA



Hari Prasad Pokhrel (hpokhrel24@gmail.com)

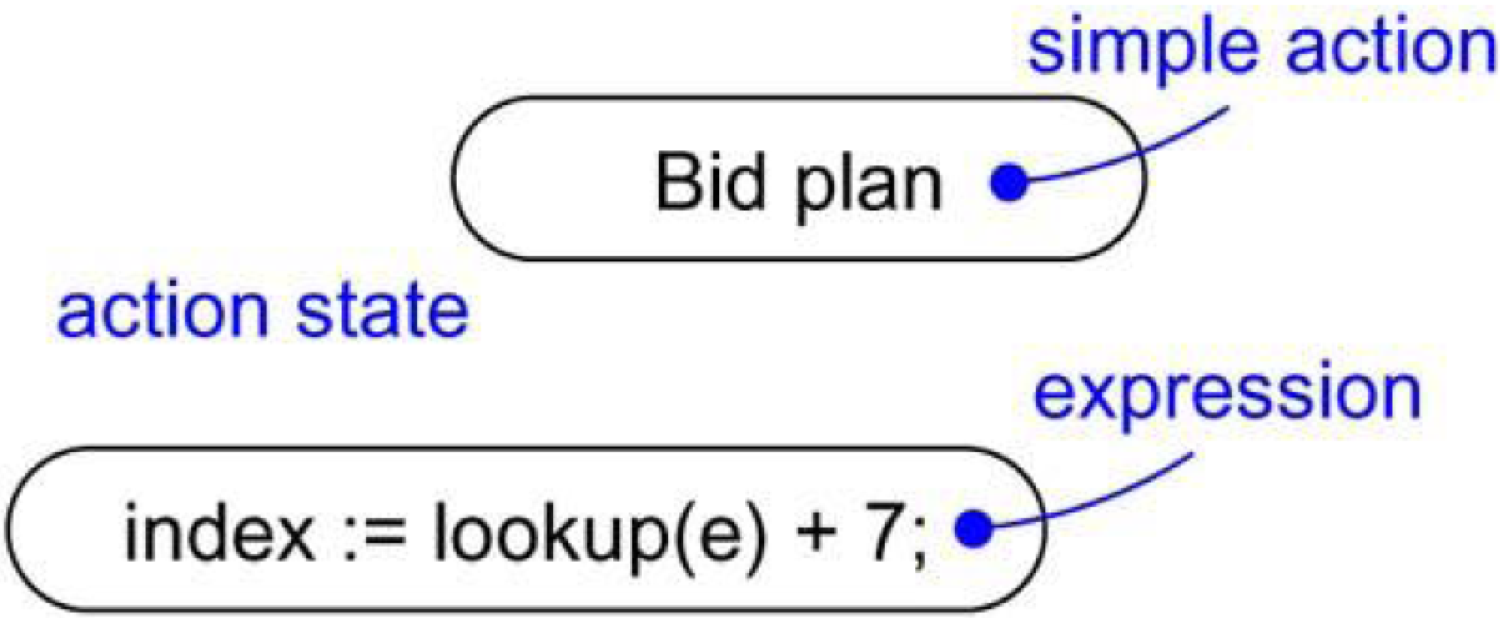
Statechart Diagram



Dynamic: Activity Diagram

- Activity Diagrams: Represent the Performance of Operations and Transitions that are Triggered
- Captures Dynamic Behavior (Activity-Oriented)
- Purposes:
  - Model Business Workflows
  - Model Operations
  - Merging of FSMs and Petri-Net Concepts?
  - Sequence::Time vs. Collaboration::Message vs. Statechart::Event vs. Activity::Actions
- Main Concepts: State, Activity, Completion Transition, Fork, Join
- Swimlanes Allow Relevant Classes to be Used

Action States



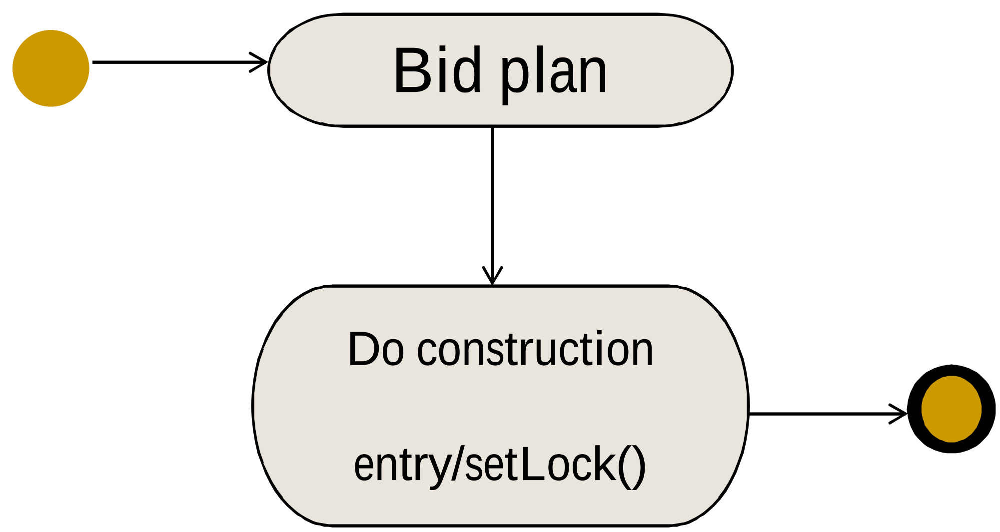
Activity: purpose

- So the purposes can be described as:
  - Draw the activity flow of a system.
  - Describe the sequence from one activity to another.
  - Describe the parallel, branched and concurrent flow of the system
- before drawing an activity diagram we should identify the following elements:
  - Activities
  - Association
  - Conditions
  - Constraints

State Diagram Carryovers

The following items are common to state diagrams and activity diagrams:

- activities
- actions
- transitions
- initial / final states



Breaking Up Flows

alternate paths:

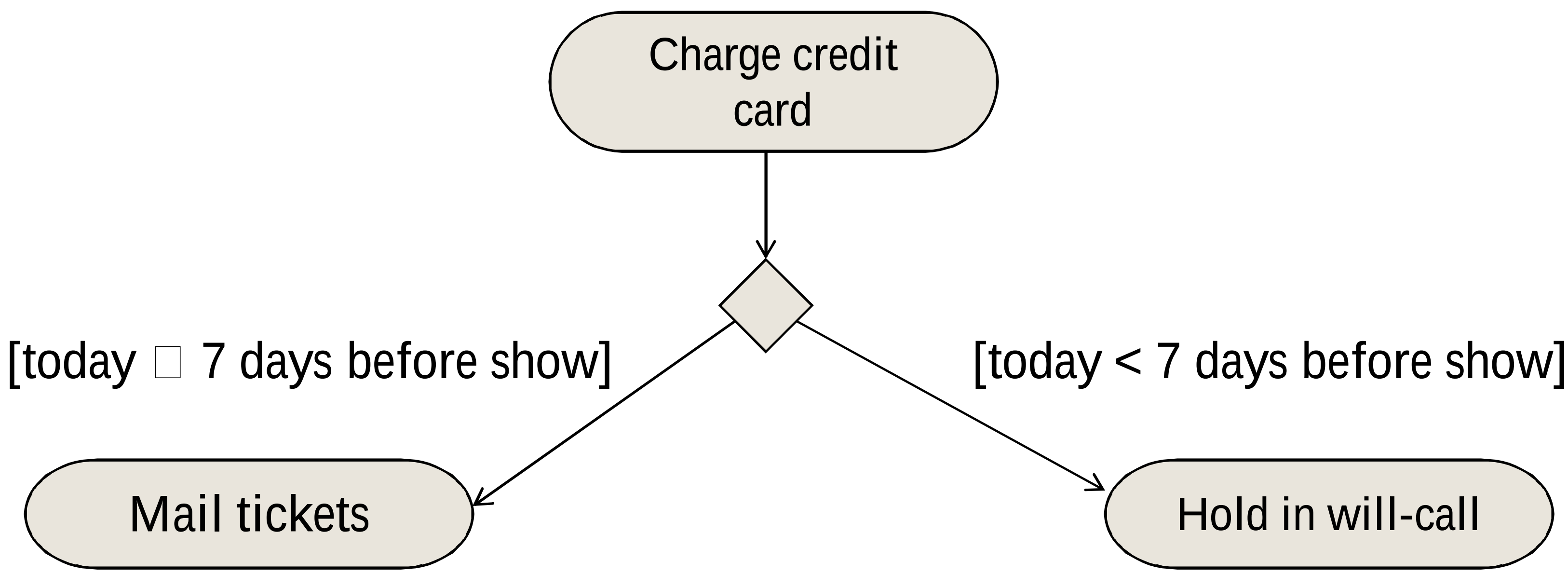
- branch
- merge

parallel flows:

- fork
- join

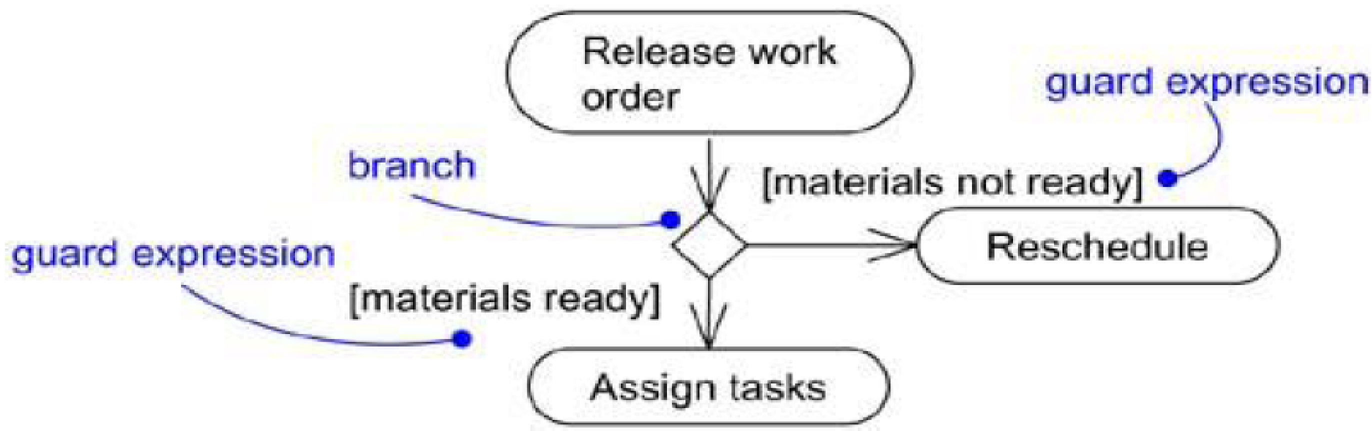
Branching

A branch has one incoming transition and two or more outgoing transitions:



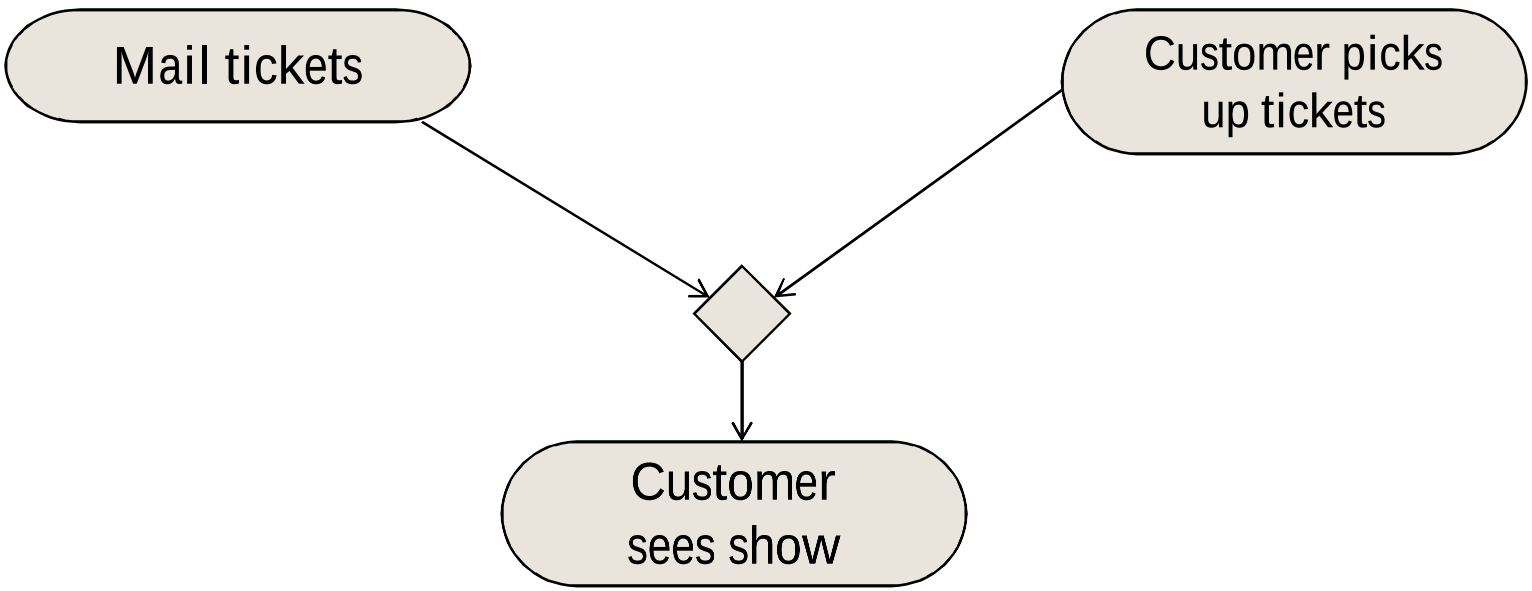
Hari Prasad Pokhrel (hpokhrel24@gmail.com)

A branch may have one incoming transition and two or more outgoing ones.
On each outgoing transition, you place a Boolean expression, which is evaluated only once on entering the branch.



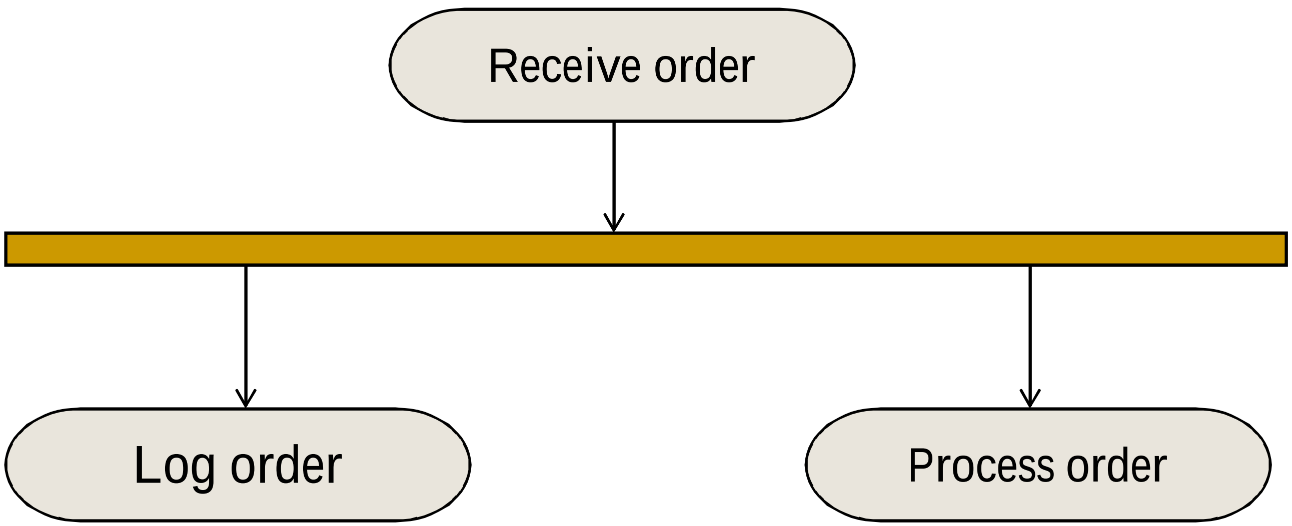
Merging

A merge has two or more incoming transitions and one outgoing transition:



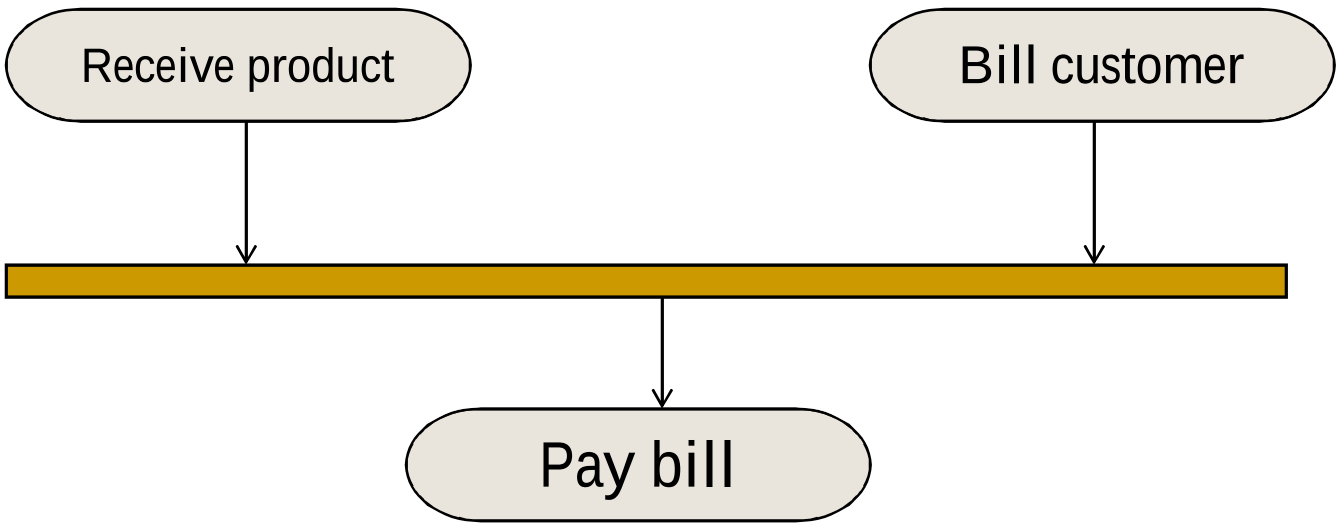
Forking

A fork represents the splitting of a single flow of control into two or more concurrent flows of control:



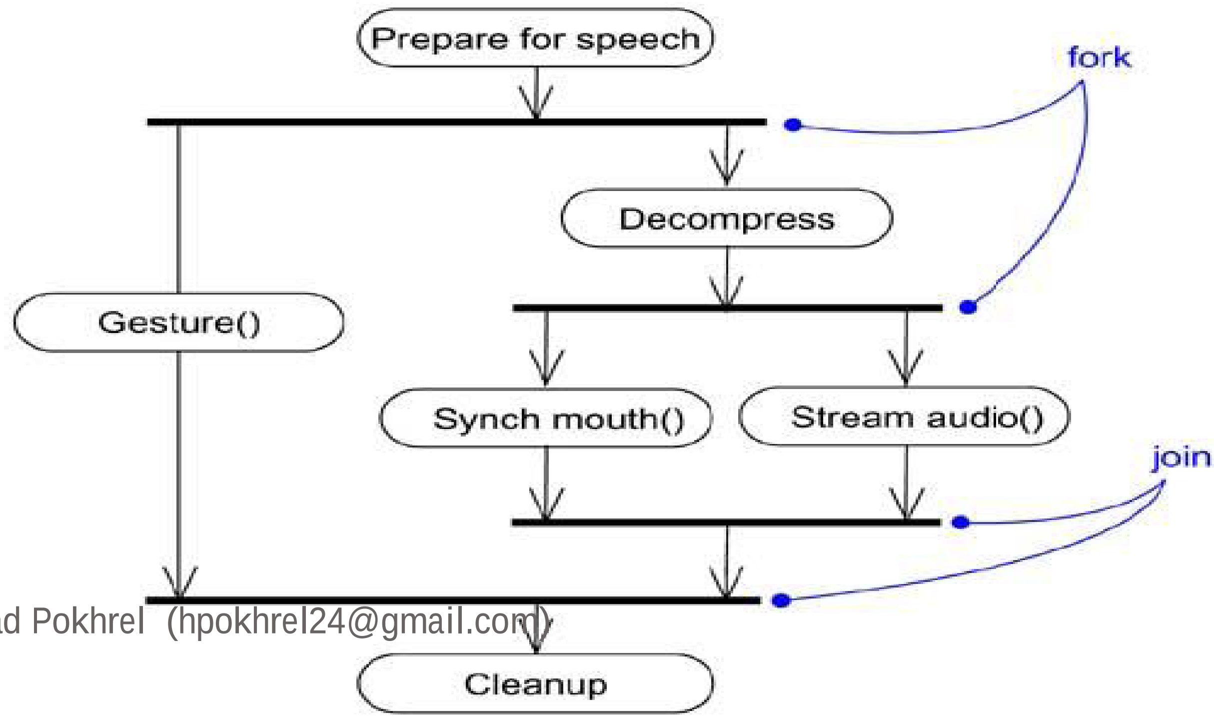
Joining

A join represents the synchronization of two or more flows of control into one sequential flow of control:



Joins and forks should balance, meaning that the number of flows that leave a fork should match the number of flows that enter its corresponding join.

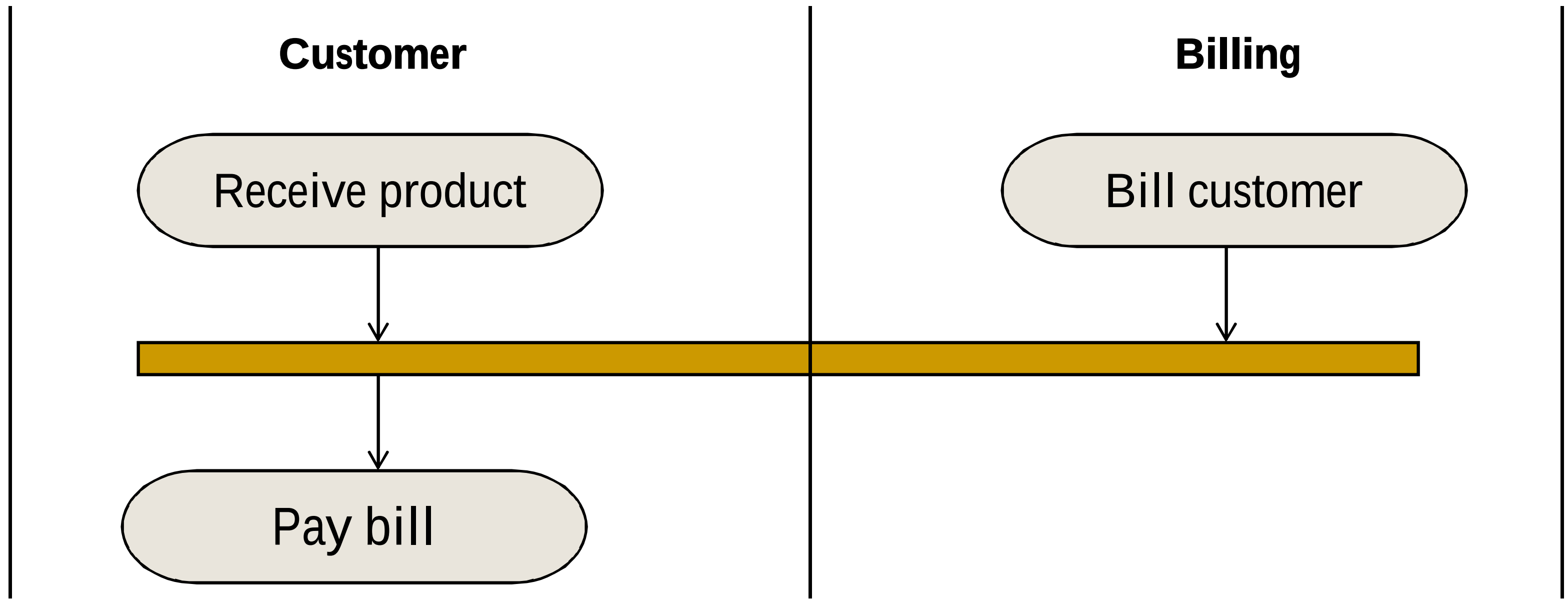
Also, activities that are in parallel flows of control may communicate with one another by sending signals. This style of communicating sequential processes is called a coroutine. Most of the time, you model this style of communication using active objects.



Swimlanes

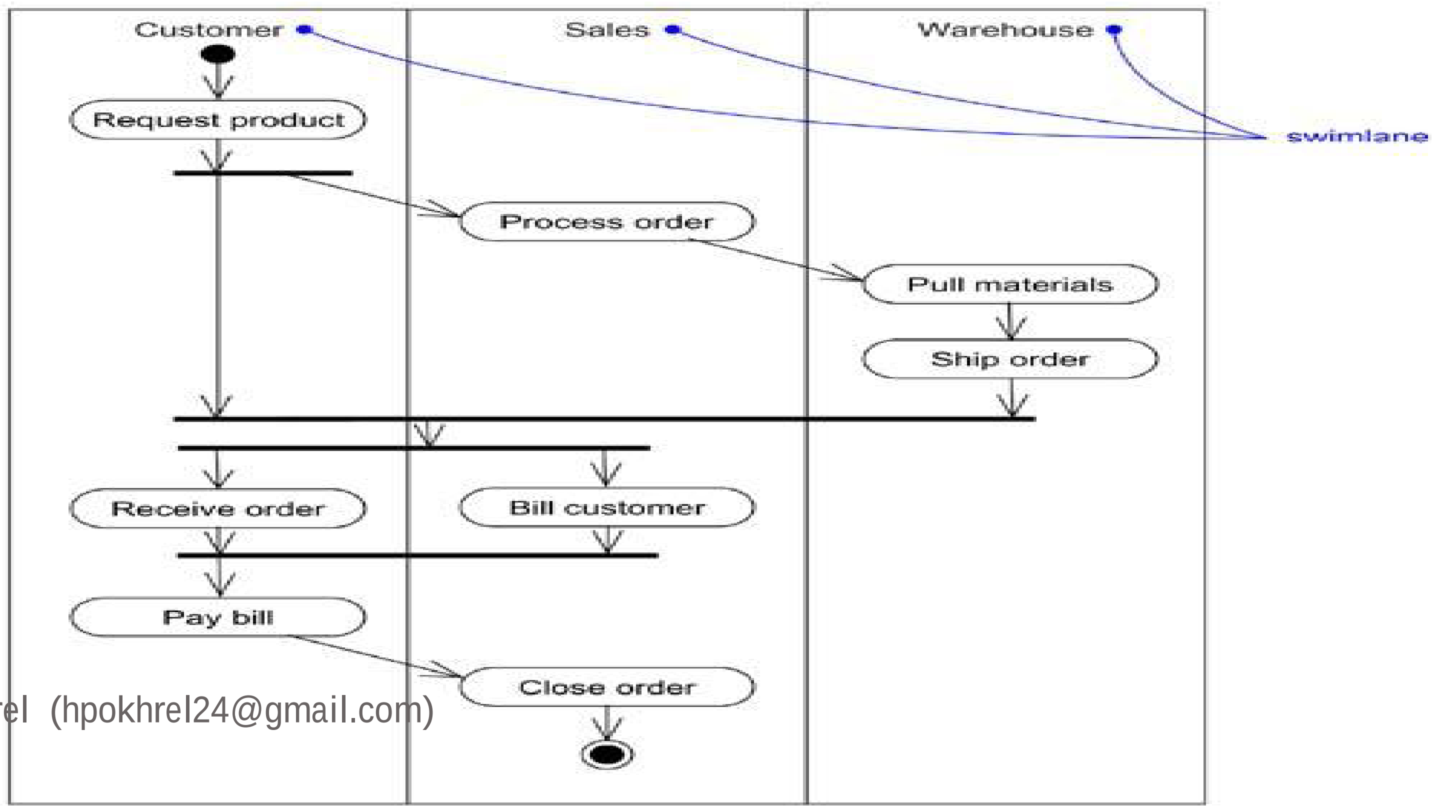
You'll find it useful, especially when you are modeling workflows of business processes, to partition the activity states on an activity diagram into groups, each group representing the business organization responsible for those activities.

Swimlanes partition groups of activities based on, for instance, business organizations:

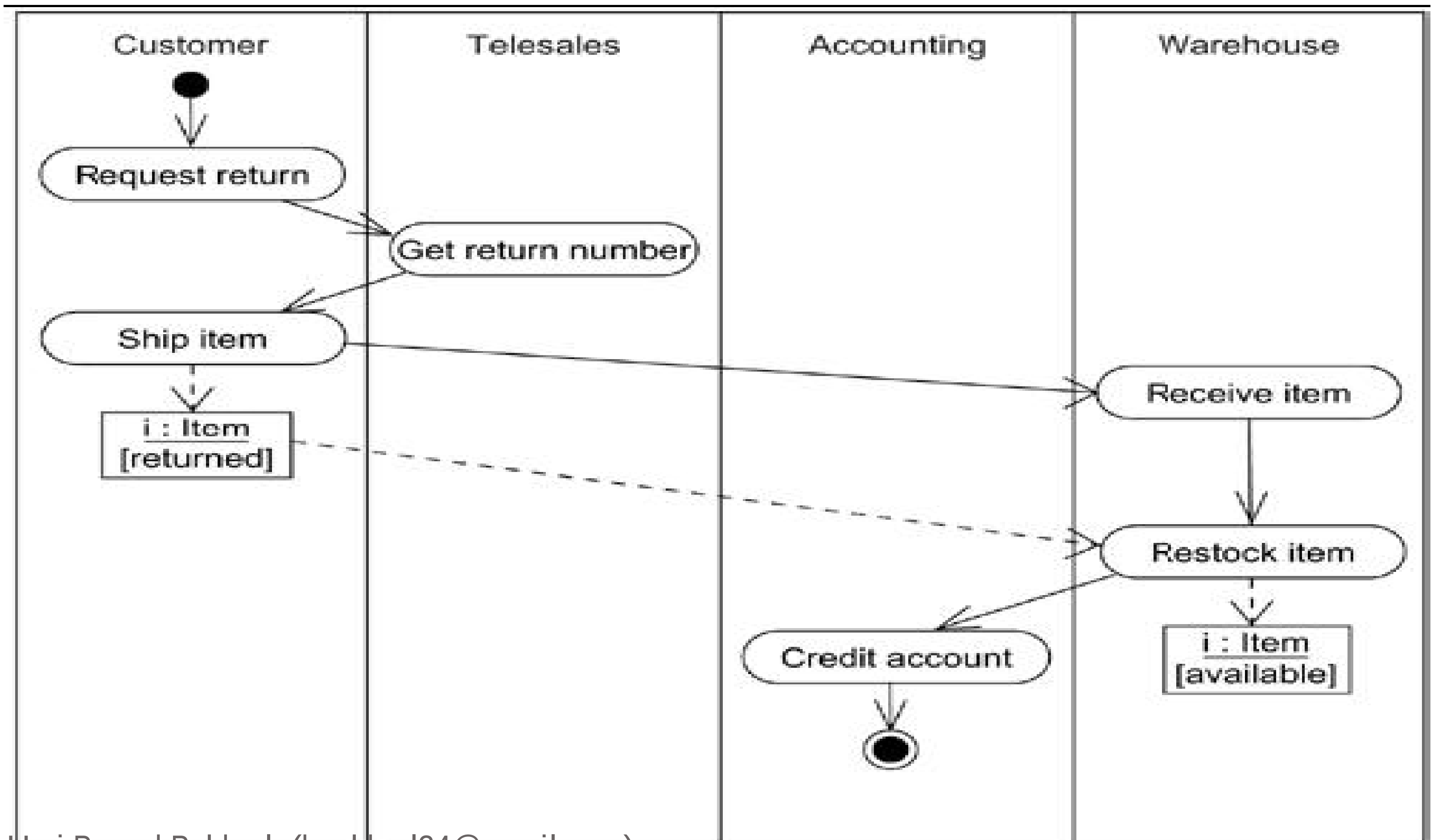


A swimlane is a kind of package;

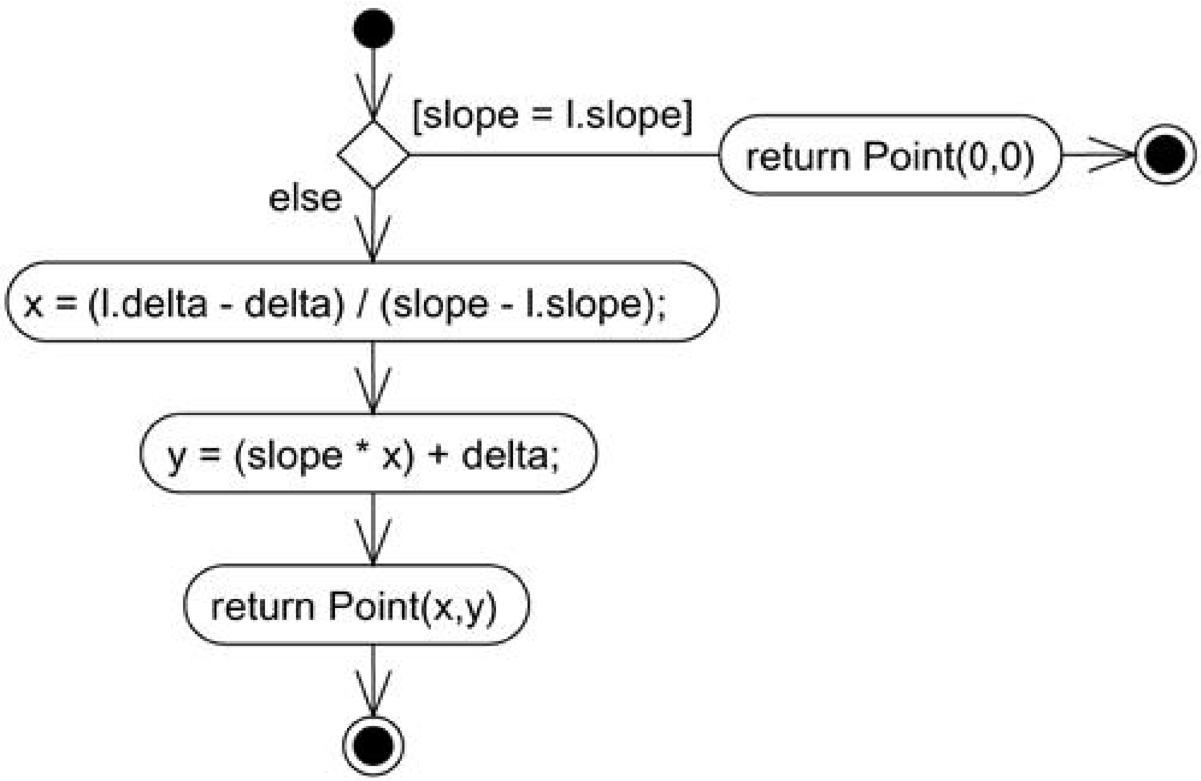
A swimlane really has no deep semantics, except that it may represent some real-world entity. Each swimlane represents a high-level responsibility for part of the overall activity of an activity diagram, and each swimlane may eventually be implemented by one or more classes



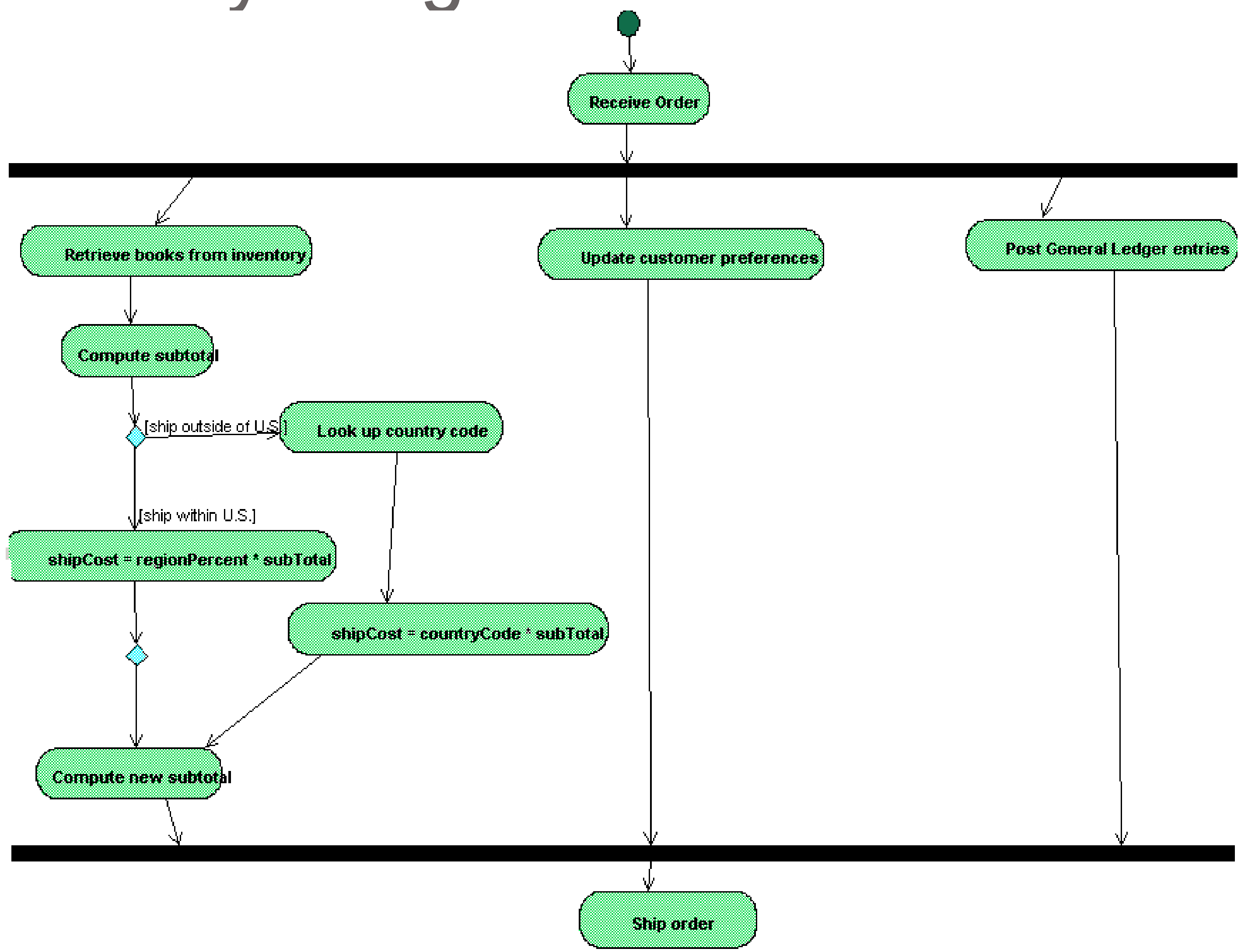
Modeling a Workflow



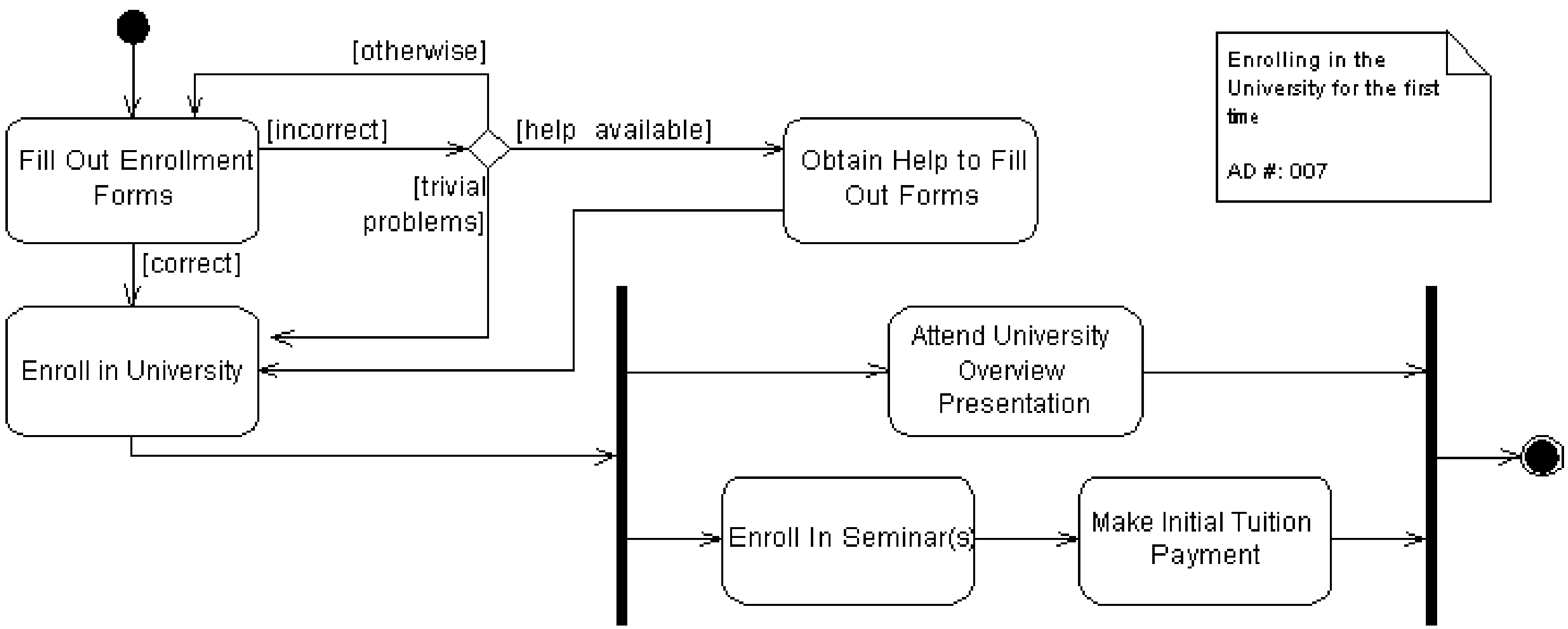
Modeling an Operation



Activity Diagram

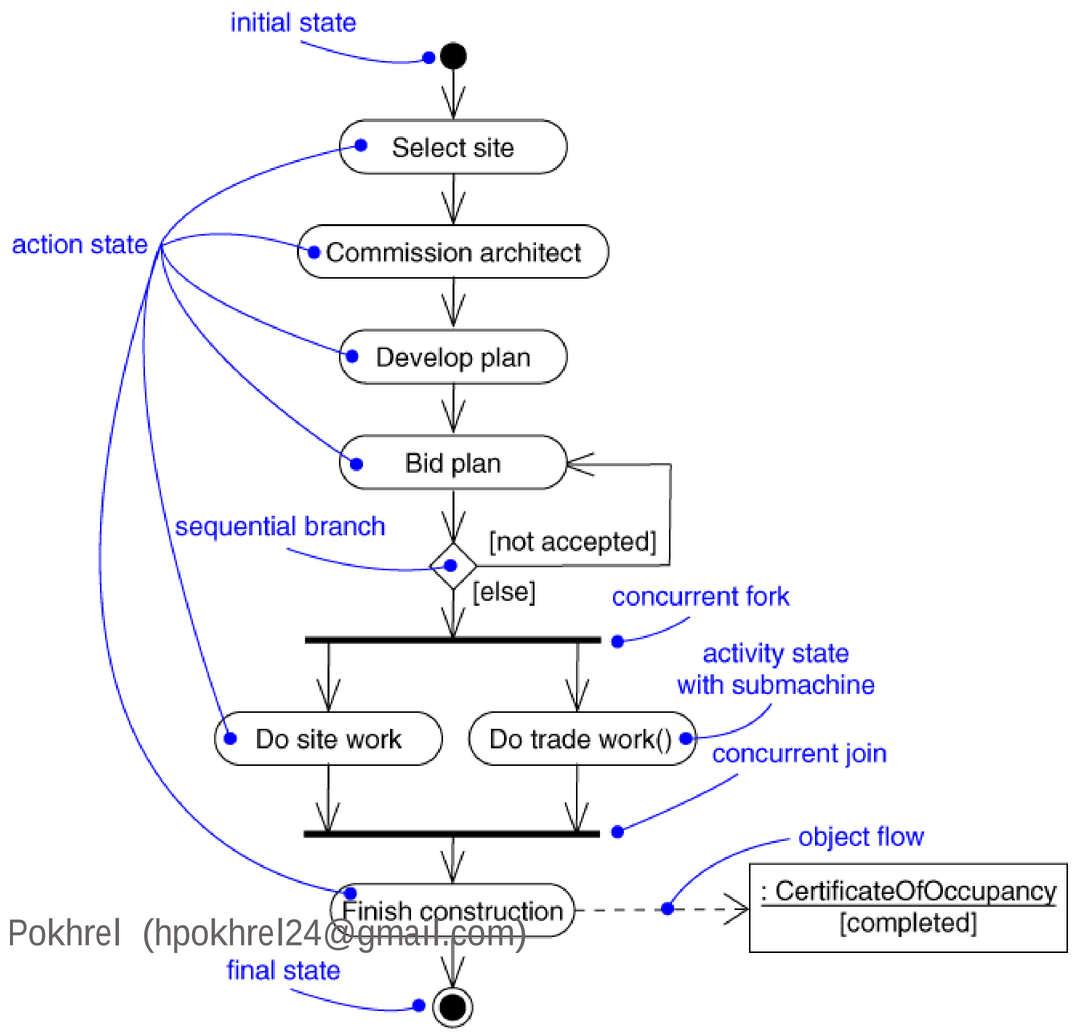


Activity Diagram

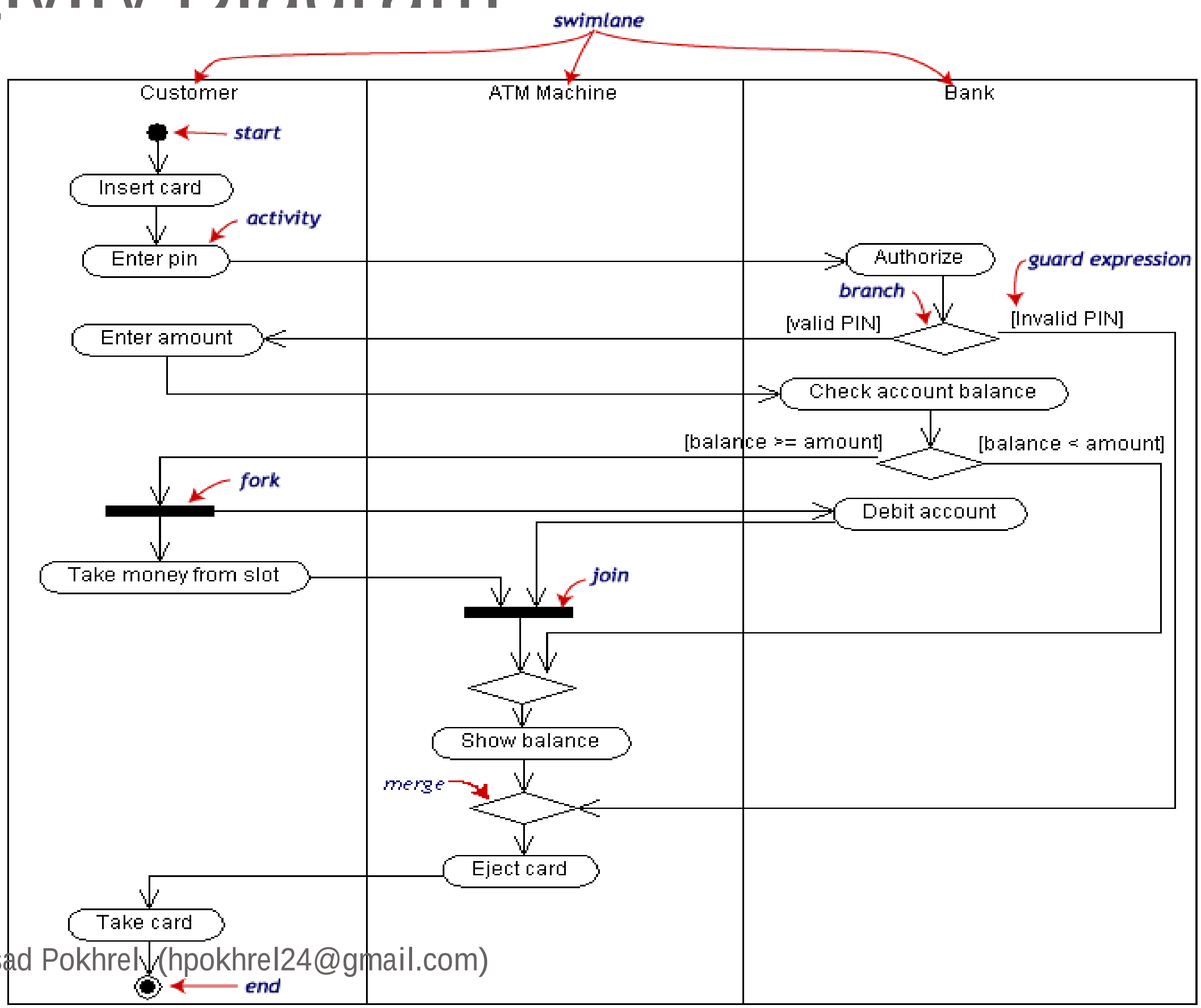


Activity Diagram

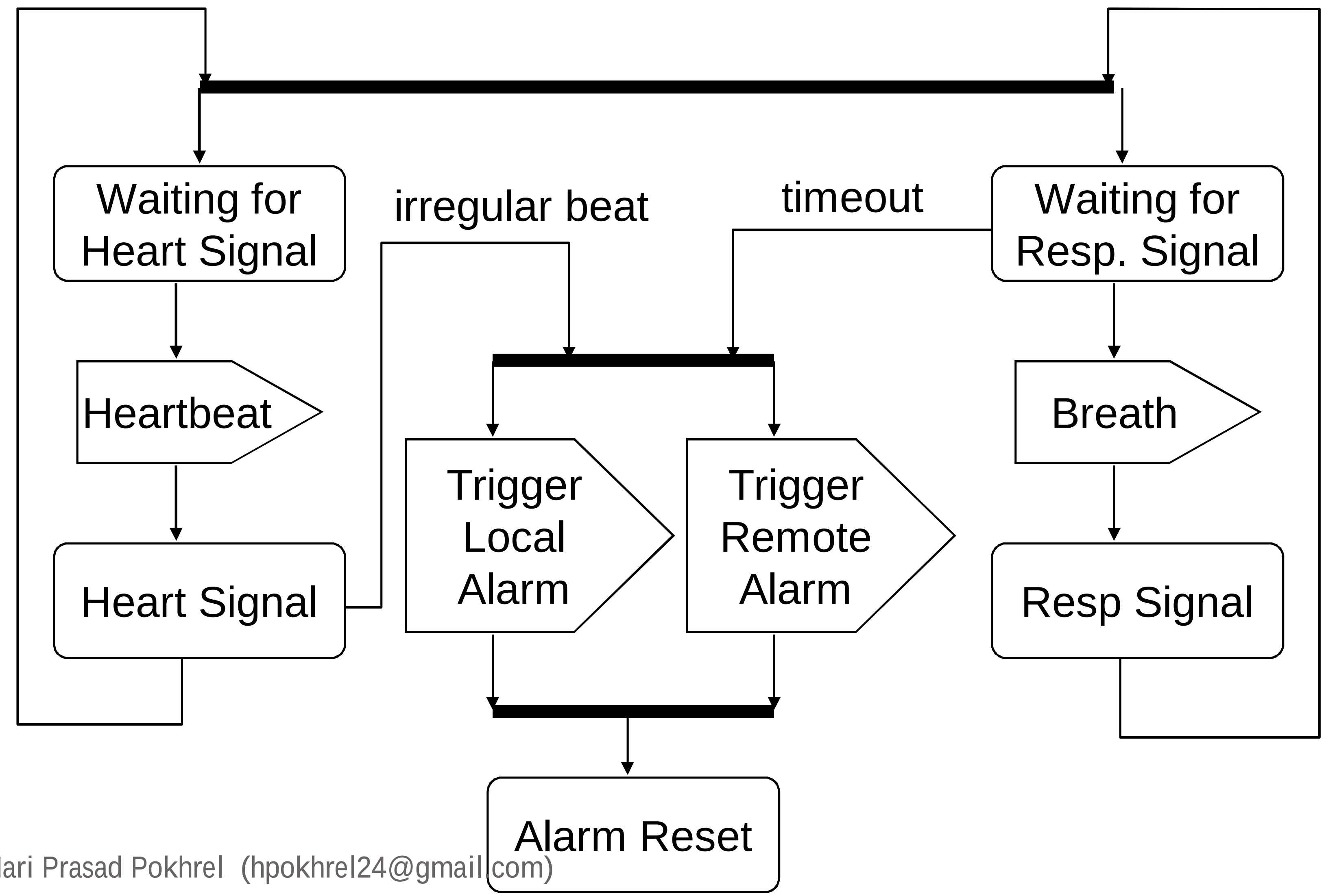
- Captures Dynamic Behavior (Activity-Oriented)



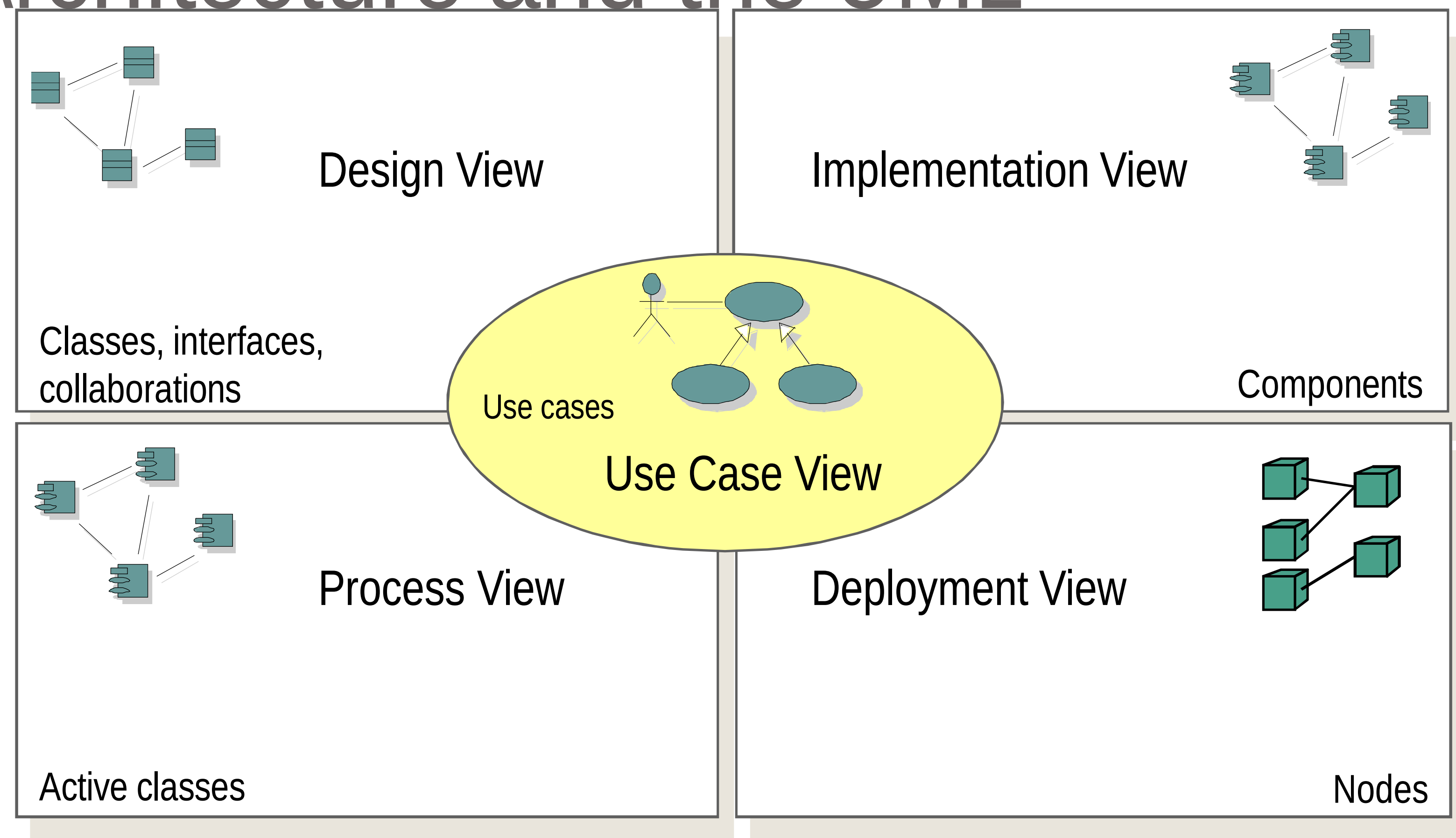
Activity Diagram



Activity Diagram
HCA



Architecture and the UML



Organization
Package, subsystem

Dynamics
Interaction
State machine

From UML to the Unified Process

- UML as a Model Can't Work in Isolation
- Large Scale System Design/Development Involves
  - Team-Oriented Efforts
  - Software Architectural Design
  - System Design, Implementation, Integration
- The Unified Process by Rational is
  - Iterative and Incremental
  - Use Case Driven
  - Architecture-Centric

Creating the Unified Process

Rational Unified Process 5.0
1998

Rational Objectory Process 4.1
1996-1997

Objectory Process 1.0-3.8
1987-1995

Functional testing
Performance testing
Requirements mgmt
Conf. and change mgmt
Business engineering
Data engineering
UI design

The Rational Approach

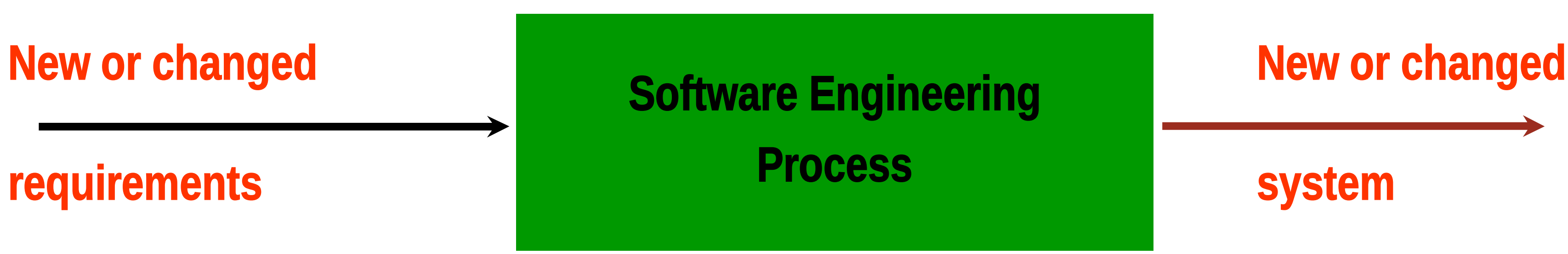
UML

The Ericsson Approach

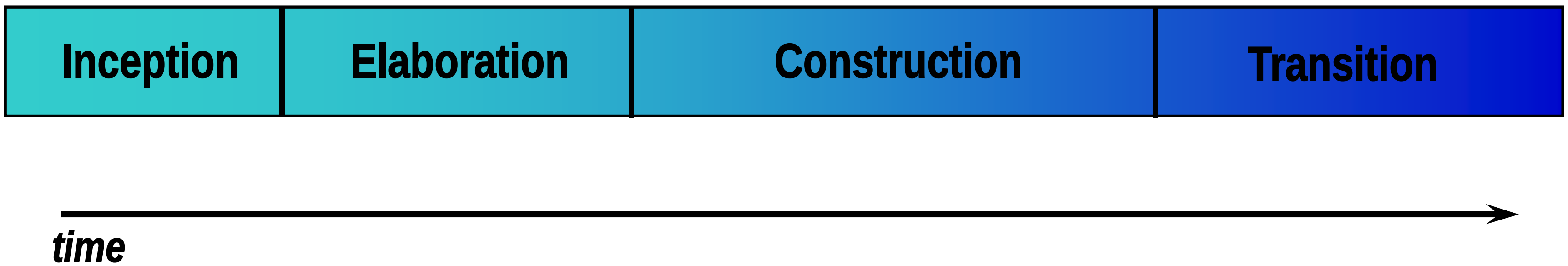
Hari Prasad Pokhrel (hpokhrel24@gmail.com)

What Is a Process?

- Defines Who is doing What, When to do it, and How to reach a certain goal.

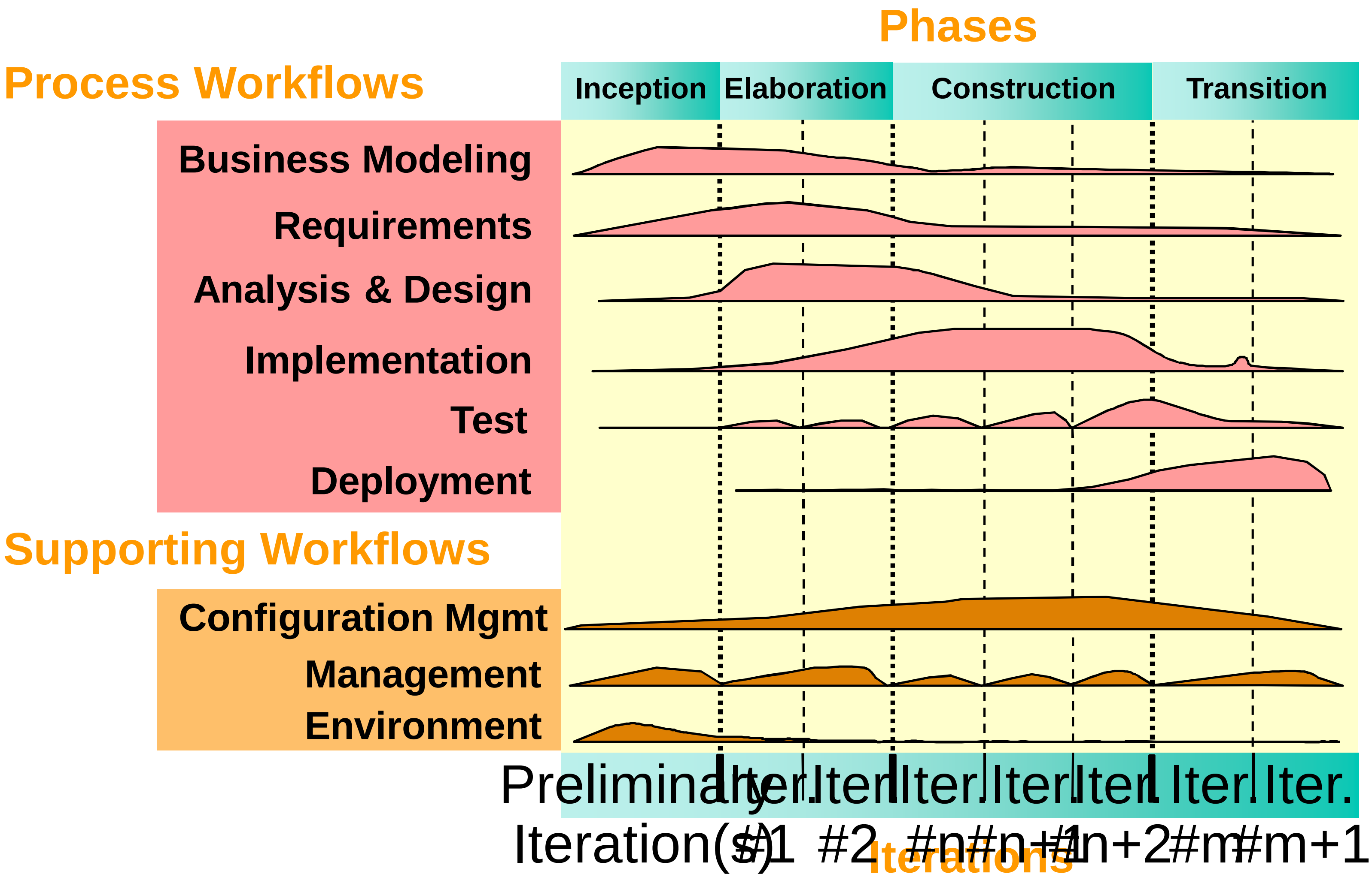


Lifecycle Phases

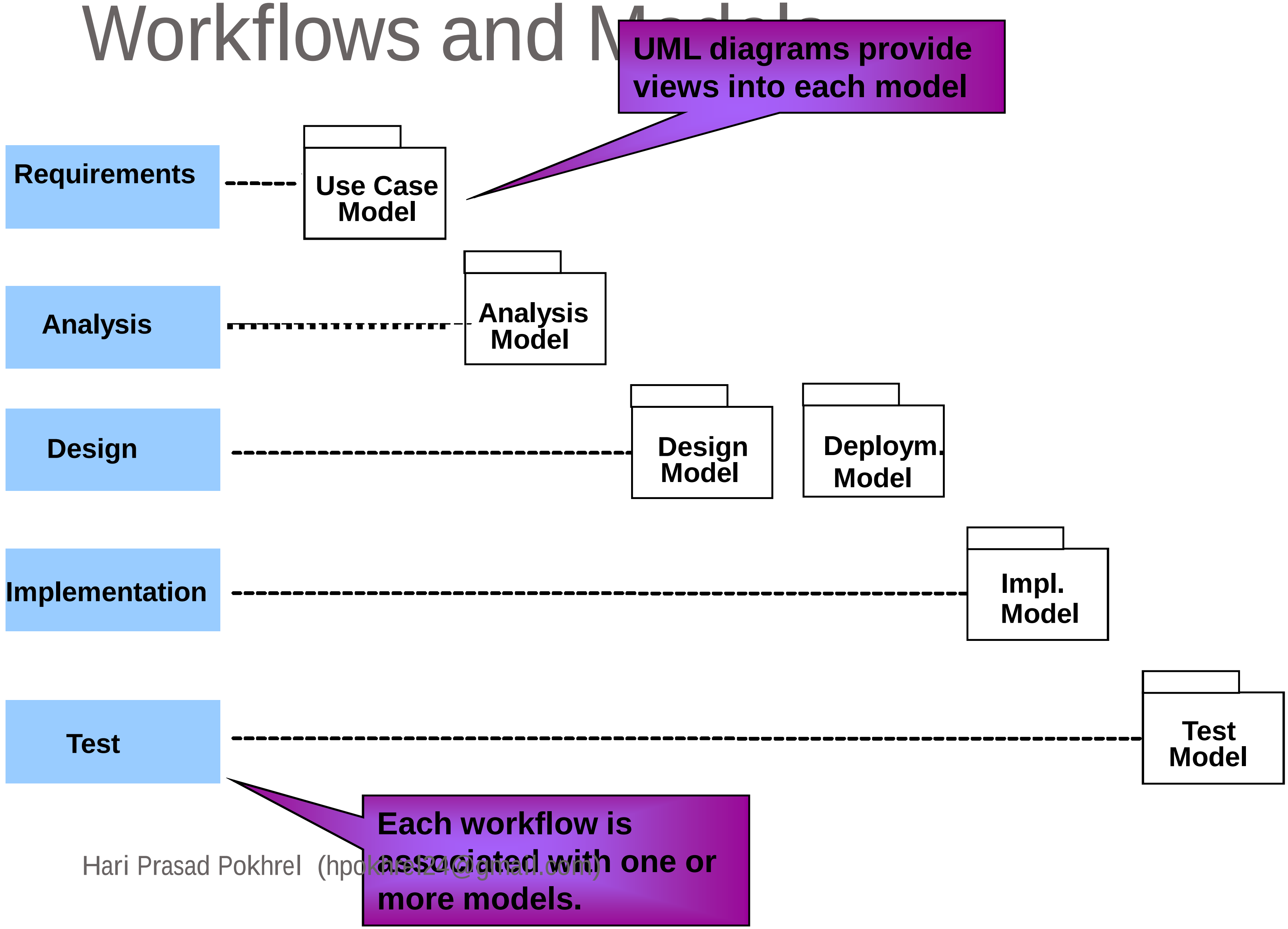


- ☐ Inception — Define the scope of the project / develop business case
- ☐ Elaboration — Plan project, specify features, and baseline the architecture
- ☐ Construction — Build the product
- ☐ Transition — Transition the product to its users

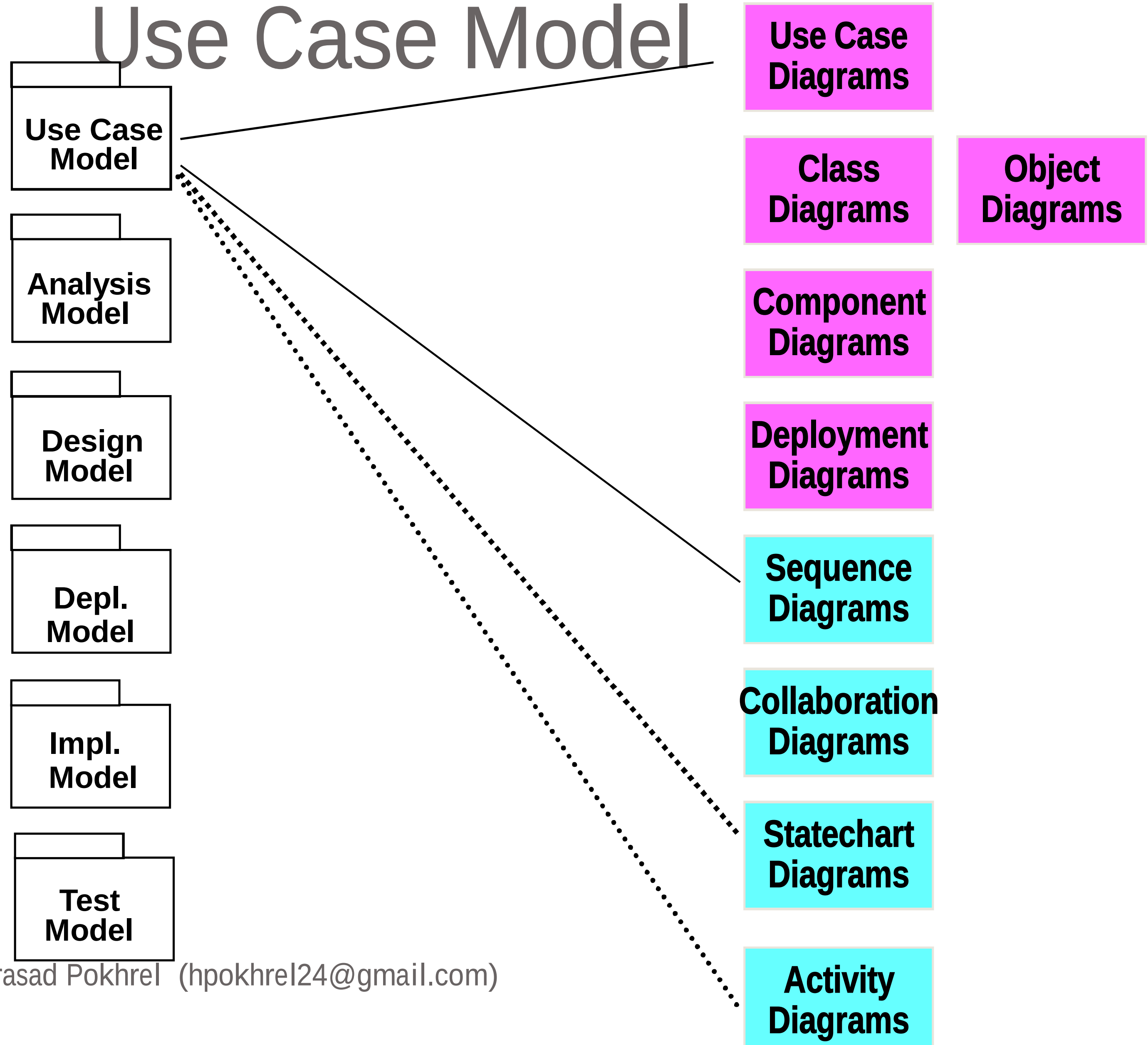
Unified Process Structure Iterations and Workflow



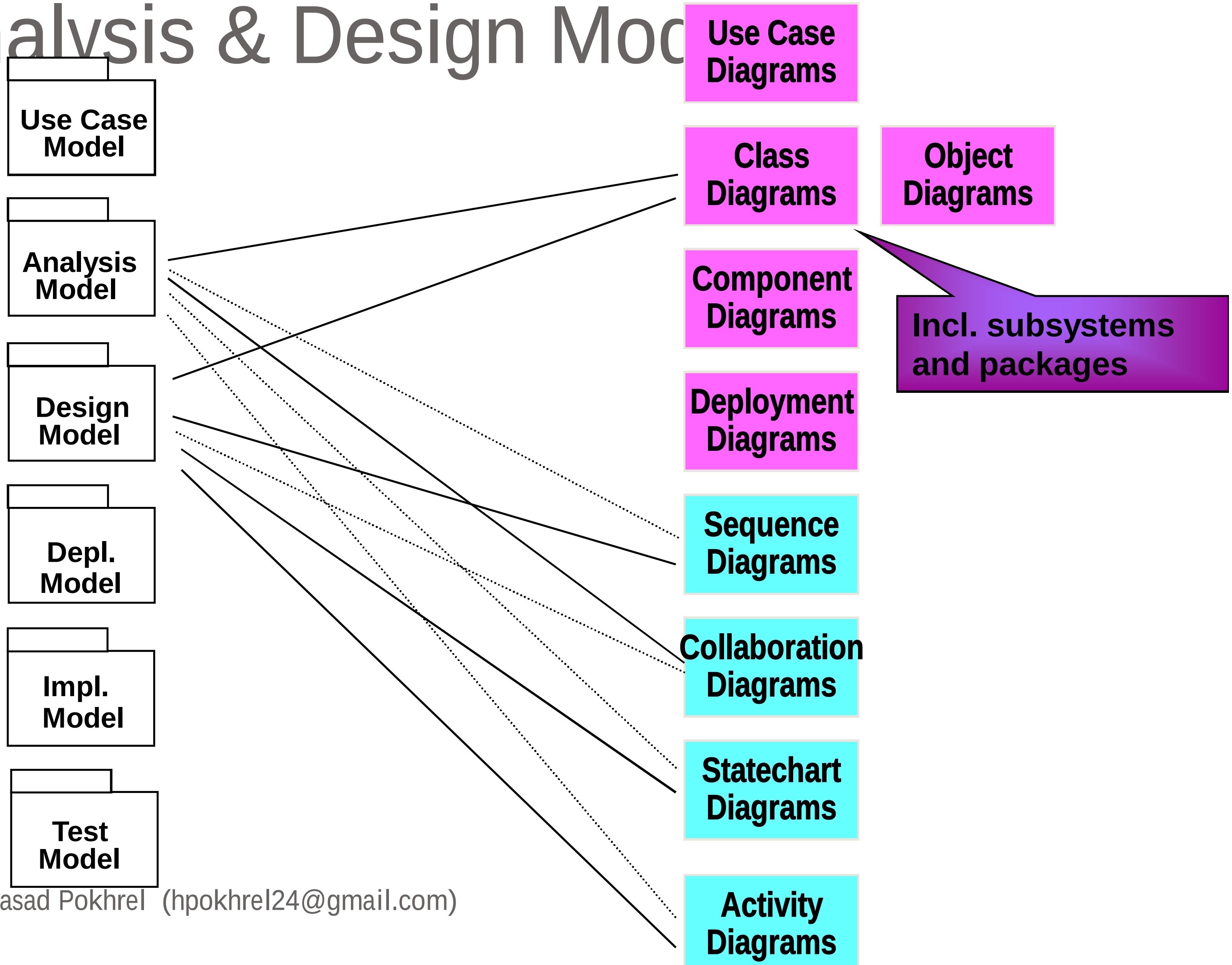
Workflows and Models



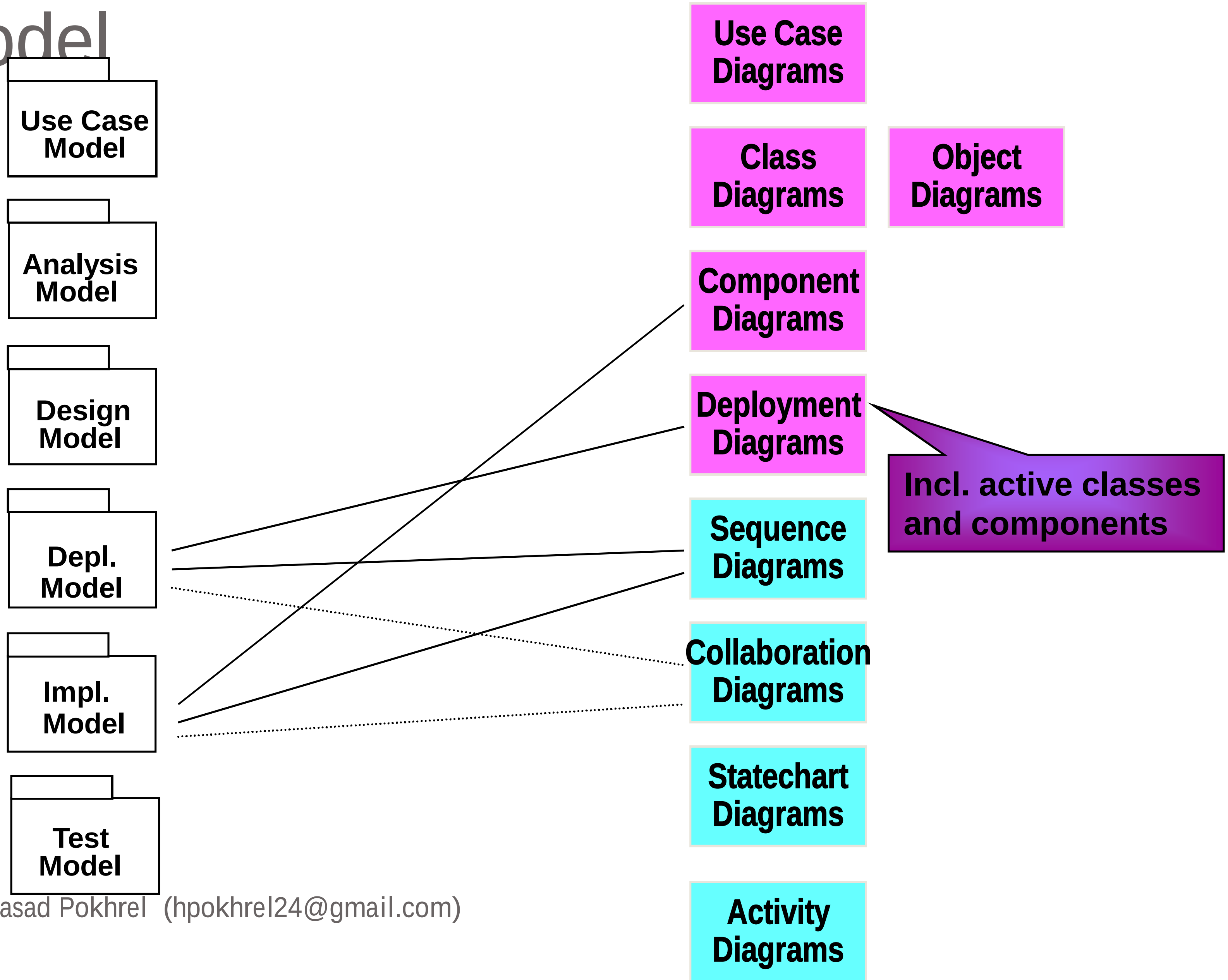
Use Case Model



Analysis & Design Moc

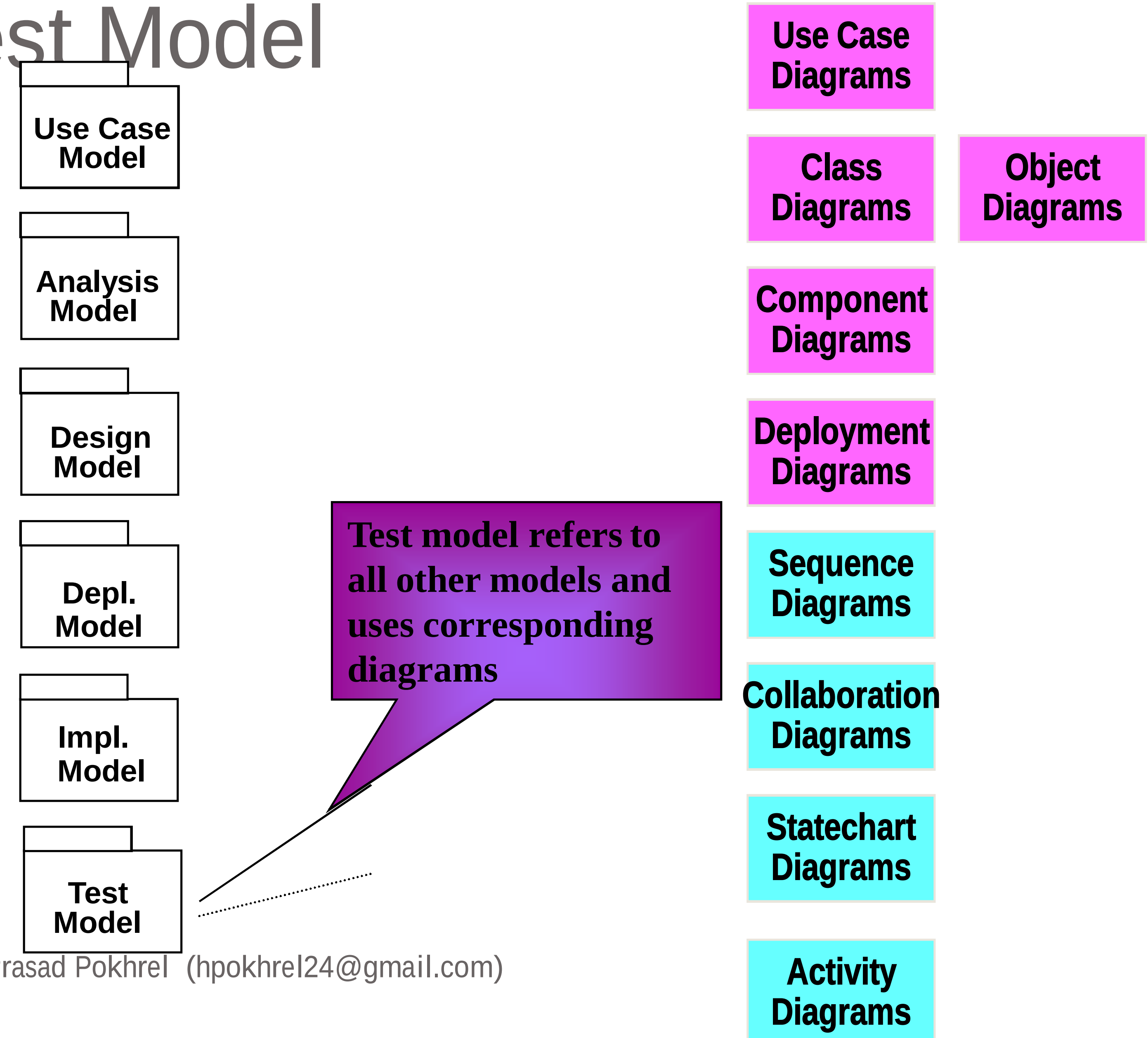


Deployment and Implementation Model

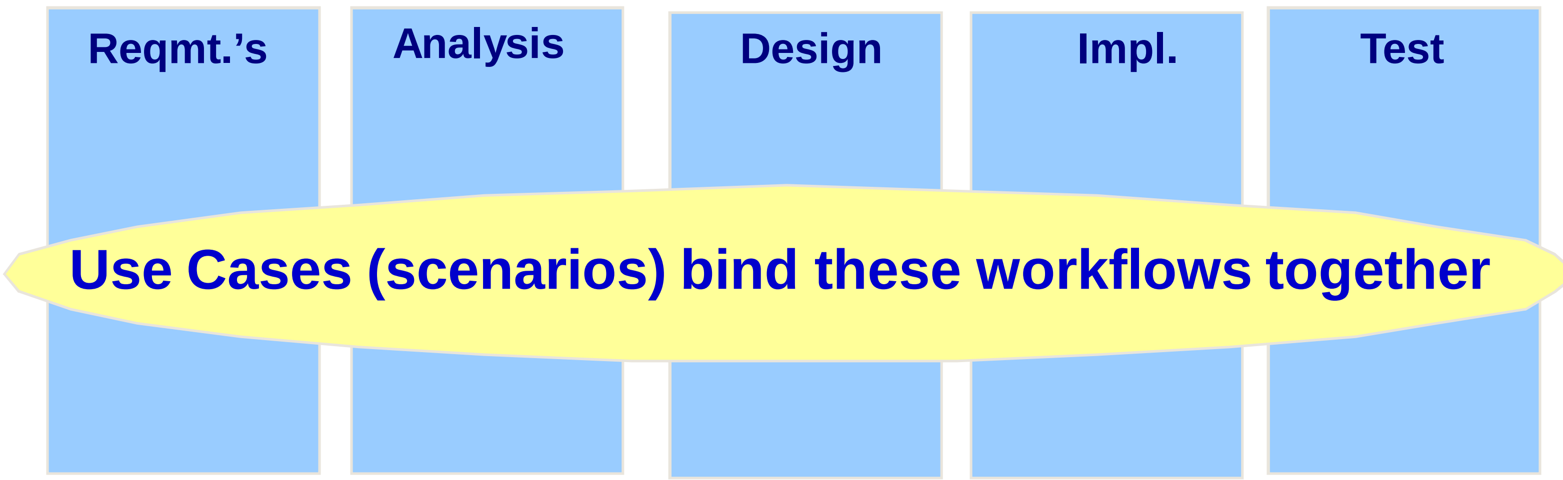


Hari Prasad Pokhrel (hpokhrel24@gmail.com)

Test Model


Use Case Model
Analysis Model
Design Model
Depl. Model
Impl. Model
Test Model
Test model refers to all other models and uses corresponding diagrams
Use Case Diagrams
Class Diagrams
Object Diagrams
Component Diagrams
Deployment Diagrams
Sequence Diagrams
Collaboration Diagrams
Statechart Diagrams
Activity Diagrams

Use Case Driven

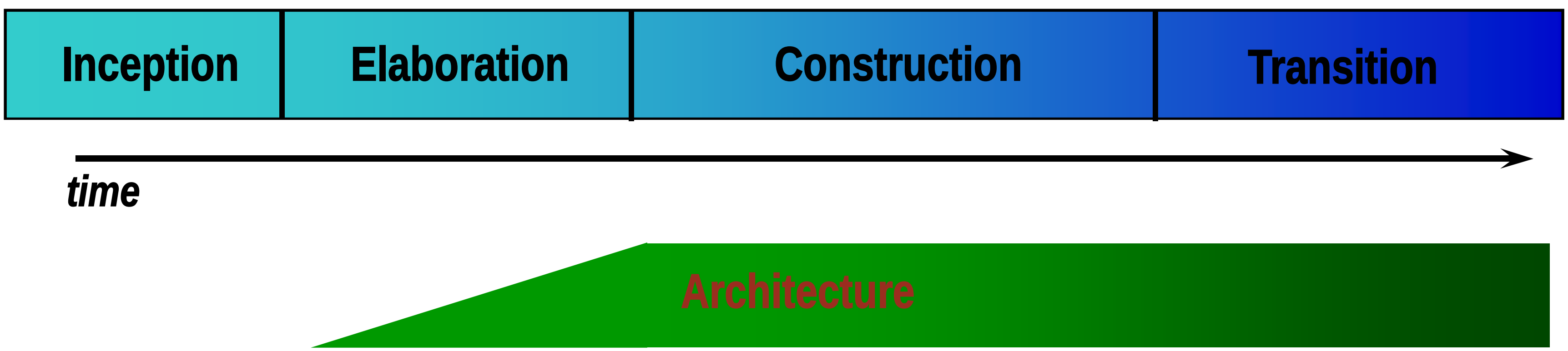


Use Cases Drive Iterations

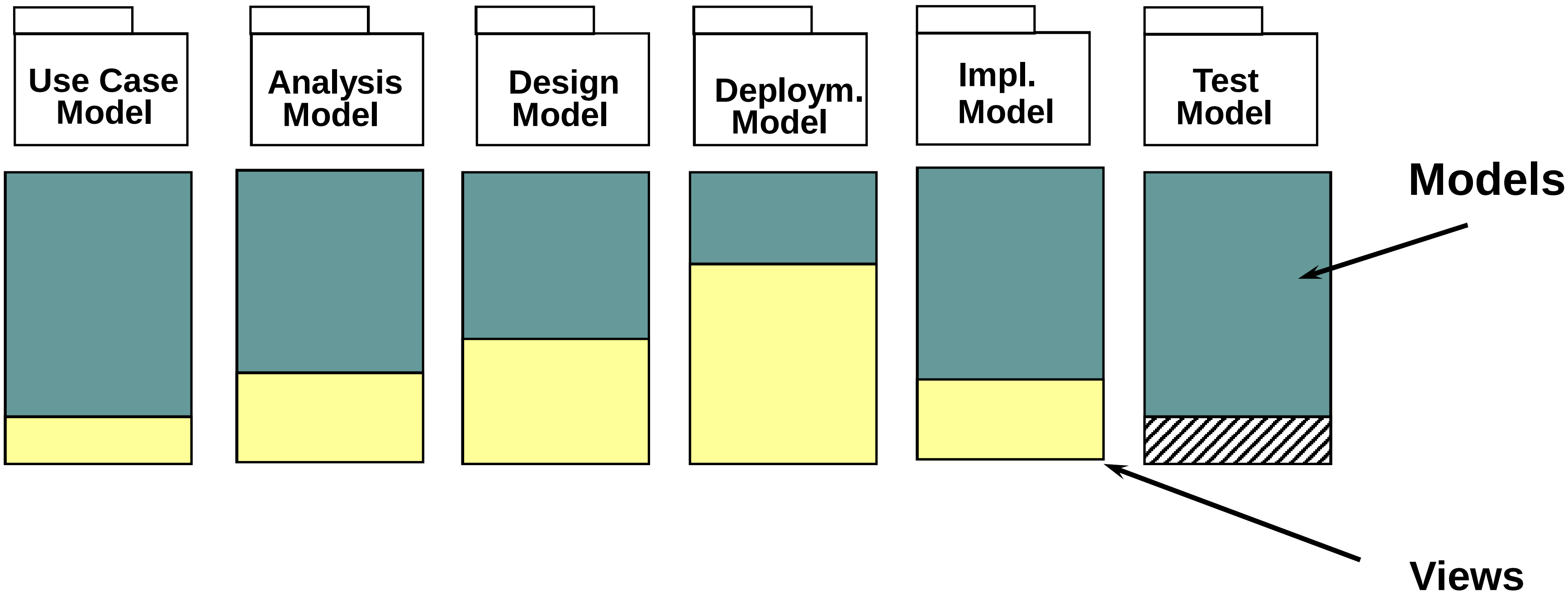
- Drive a Number of Development Activities
  - Creation and Validation of the System's Architecture
  - Definition of Test Cases and Procedures
  - Planning of Iterations
  - Creation of User Documentation
  - Deployment of System
- Synchronize the Content of Different Models

Architecture-Centric

- Models Are Vehicles for Visualizing, Specifying, Constructing, and Documenting Architecture
- The Unified Process Prescribes the Successive Refinement of an Executable Architecture

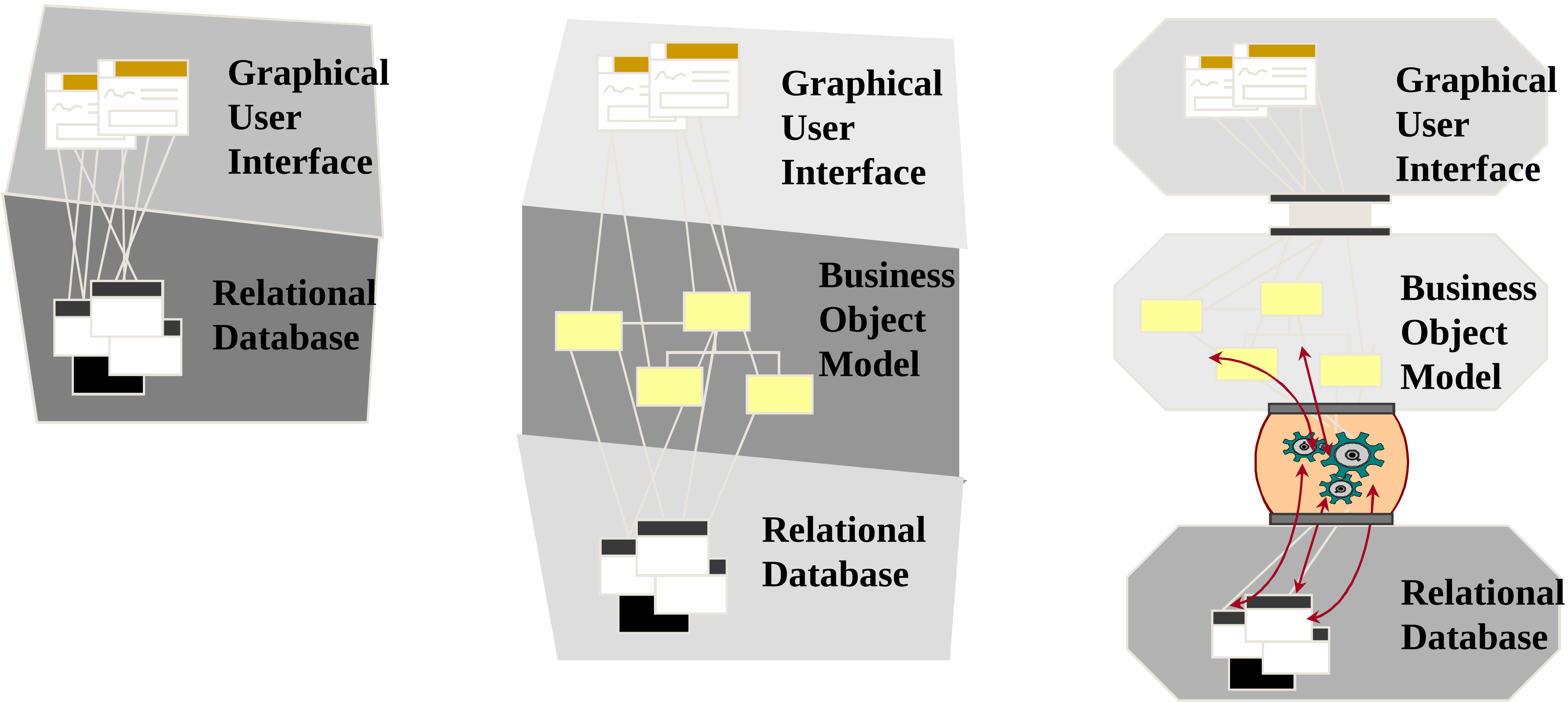


Architecture and Models



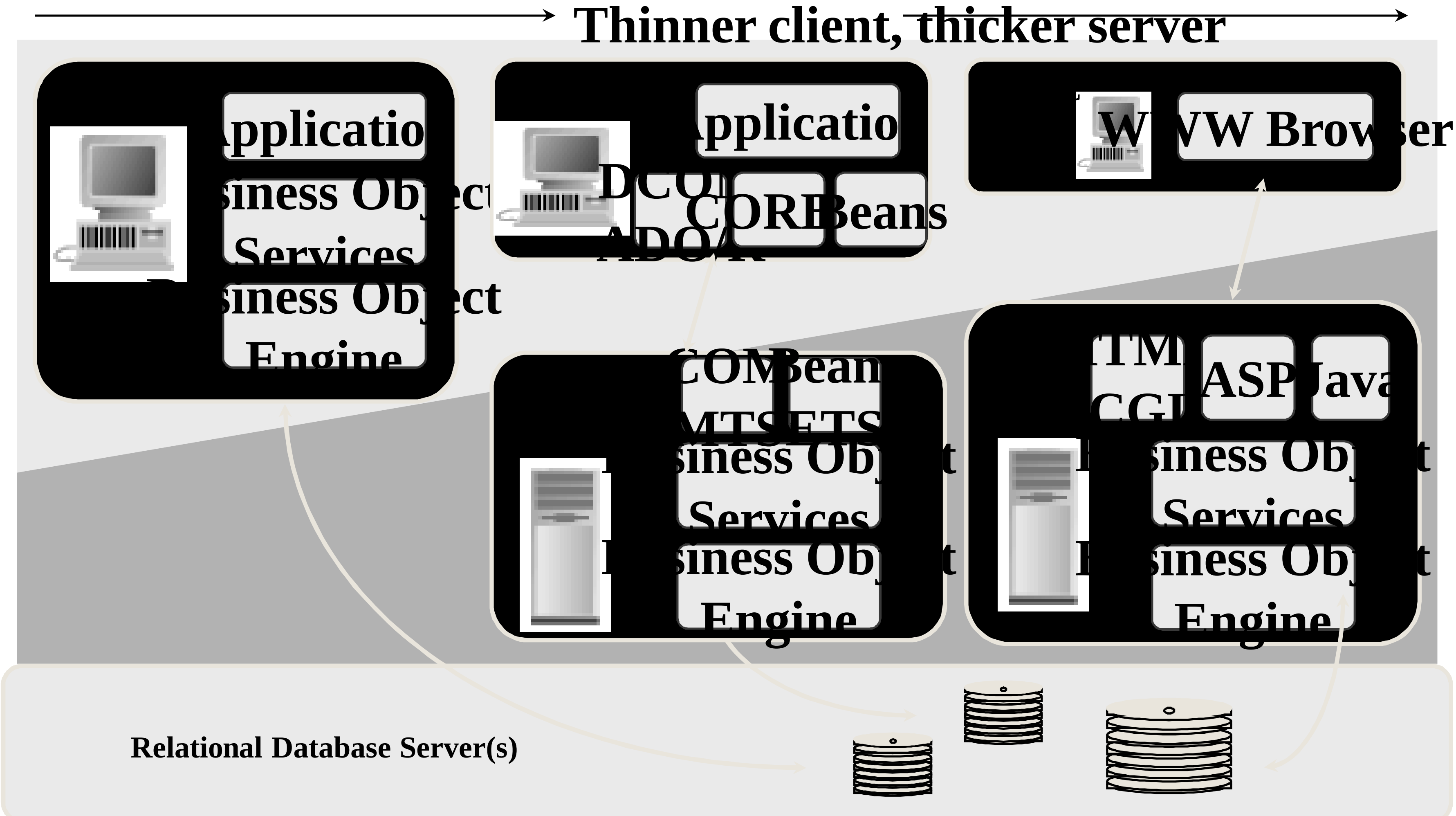
Architecture embodies a collection of views of the models

Logical Application Architecture

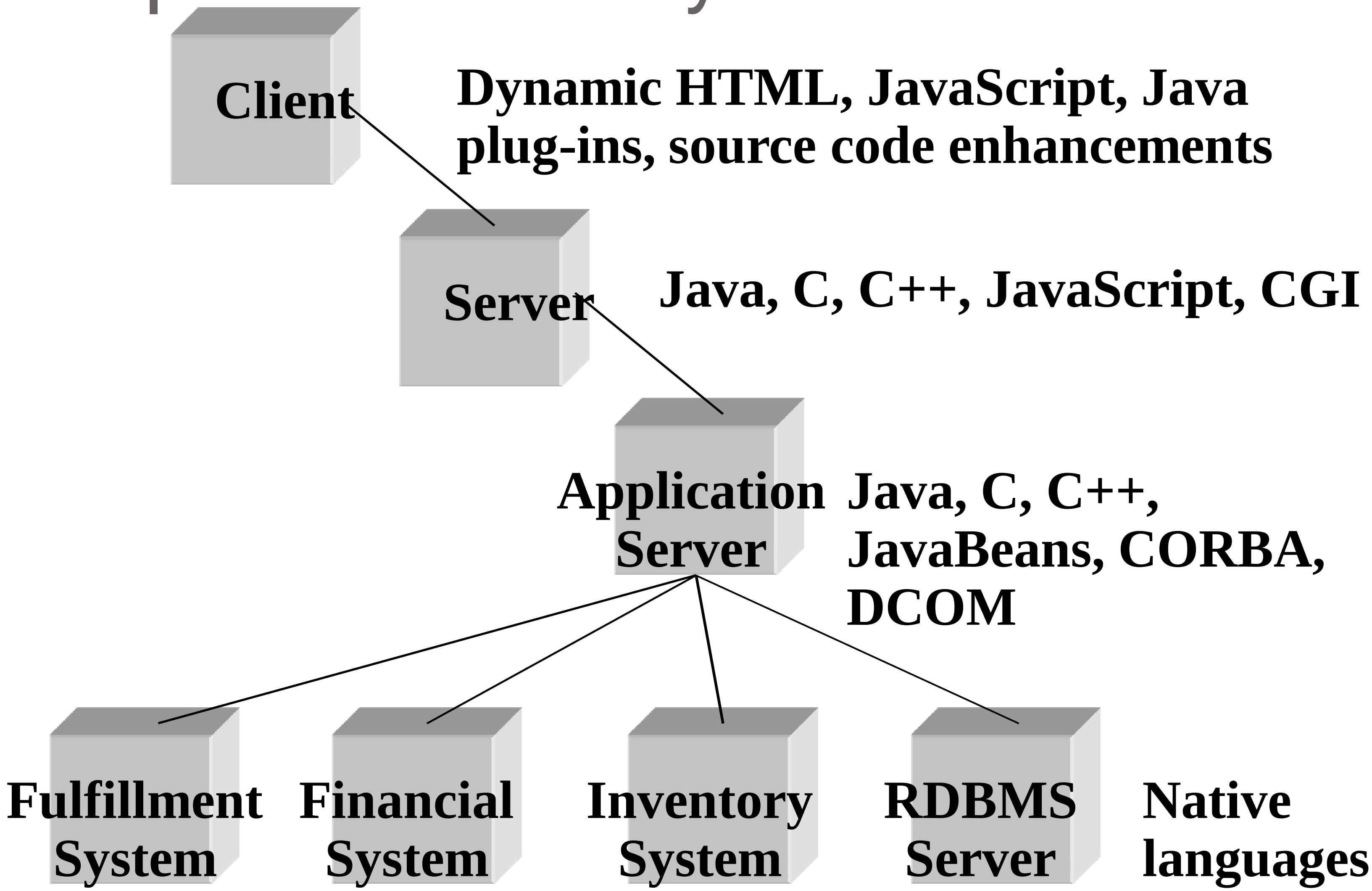


Hari Prasad Pokhrel (hpokhrel24@gmail.com)

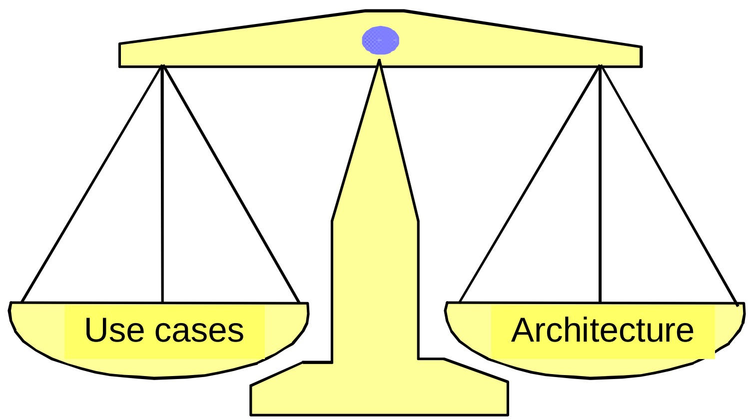
Physical Application Architecture



Complex Internet System



Function versus Form



- Use Case Specify Function; Architecture Specifies Form
- Use Cases and Architecture Must Be Balanced

The Unified Process is Engineered