Object Oriented Methodologies

Dr. Ranjana Vyas

Indian Institute of Information Technology, Allahabad

OOM

- **OOM is…..**
  - ✓ Think in Object Oriented Manner !
  - ✓ Design with Object Oriented Modeling !!
  - ✓ Develop using Object Oriented Technology !!
  - ✓ Maintain/Reuse in Object Oriented Approach !!!

Design with Object Oriented Modeling !!

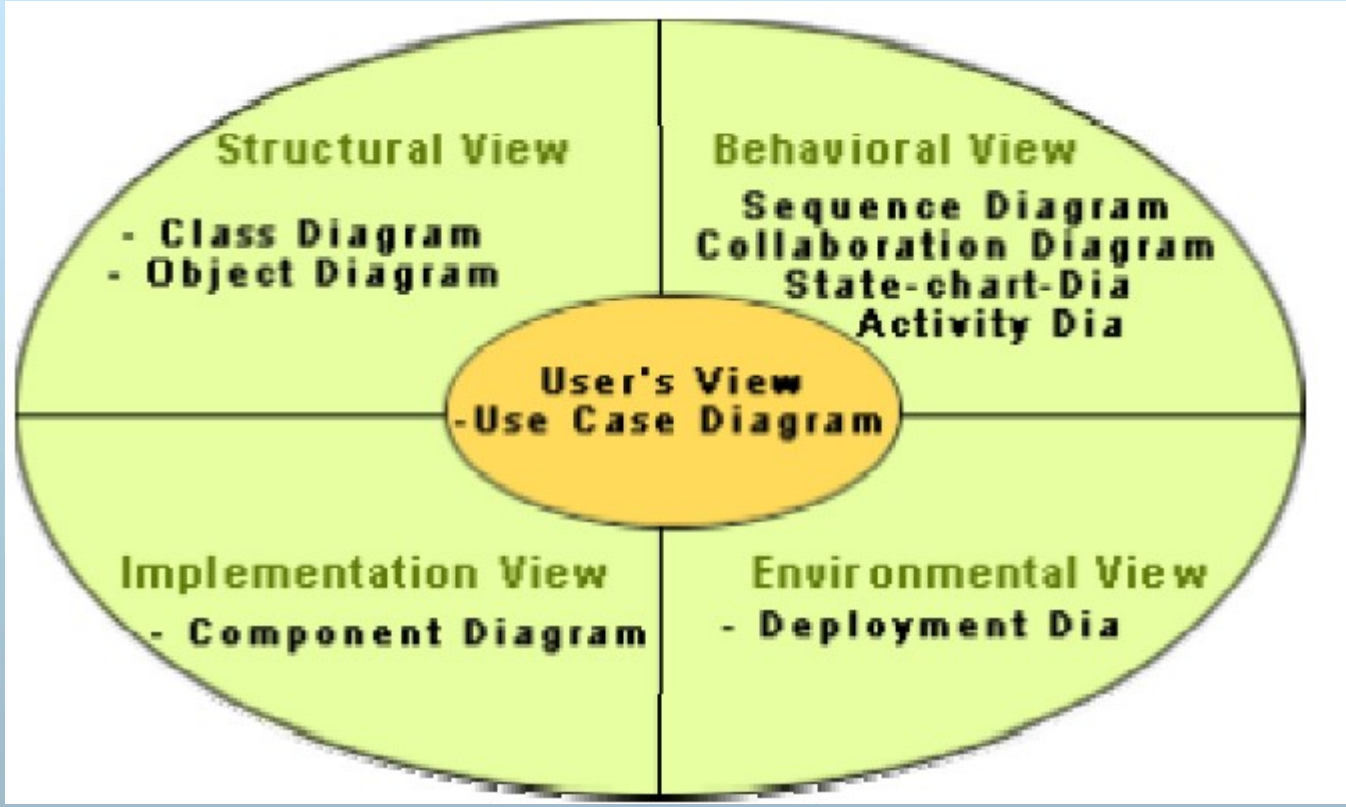
Using UML

UML

UML provides different view of the same system from different viewpoint so that we may know it in totality;

- **User's view of the System**
- **Structural view of the System.**
- **Behavioral View of System**
- **Implementation View of System**
- **Deployment View of System**

UML: Different View diagrams



UML

At the center of the UML are its nine kinds of modeling diagrams;.

- Use case diagrams
- Class diagrams
- Object diagrams
- Sequence diagrams
- Collaboration diagrams
- Statechart diagrams
- Activity diagrams
- Component diagram
- Deployment diagram

UML : Use Case & Class diagram

Use case diagrams describe what a system does from the standpoint of an external observer.

- **The emphasis is on *what* a system does rather than *how.***
- **Use case diagrams are closely connected to scenarios.**
- **A scenario is an example of what happens when someone interacts with the system**

Class Diagrams:

- A class diagram describes the **static structure** of a system. It shows how a system is **structured** rather than how it **behaves**.
- The static structure of a system comprises of a number of **class diagrams** and their dependencies.

Use case diagram or Class Diagram ?

- Many Software Engineers consider that ;
  - **Class Diagram** is the most significant diagram and we should begin with drawing Class Diagram.
  - **Based on Class diagram**, the **developers can write the software** in any OO Programming Language.
  - So in practice many **try to avoid starting with Use Case** Diagram..
  - Is there any problem in this approach ?

What is your opinion ?

Use Case Diagram First ??

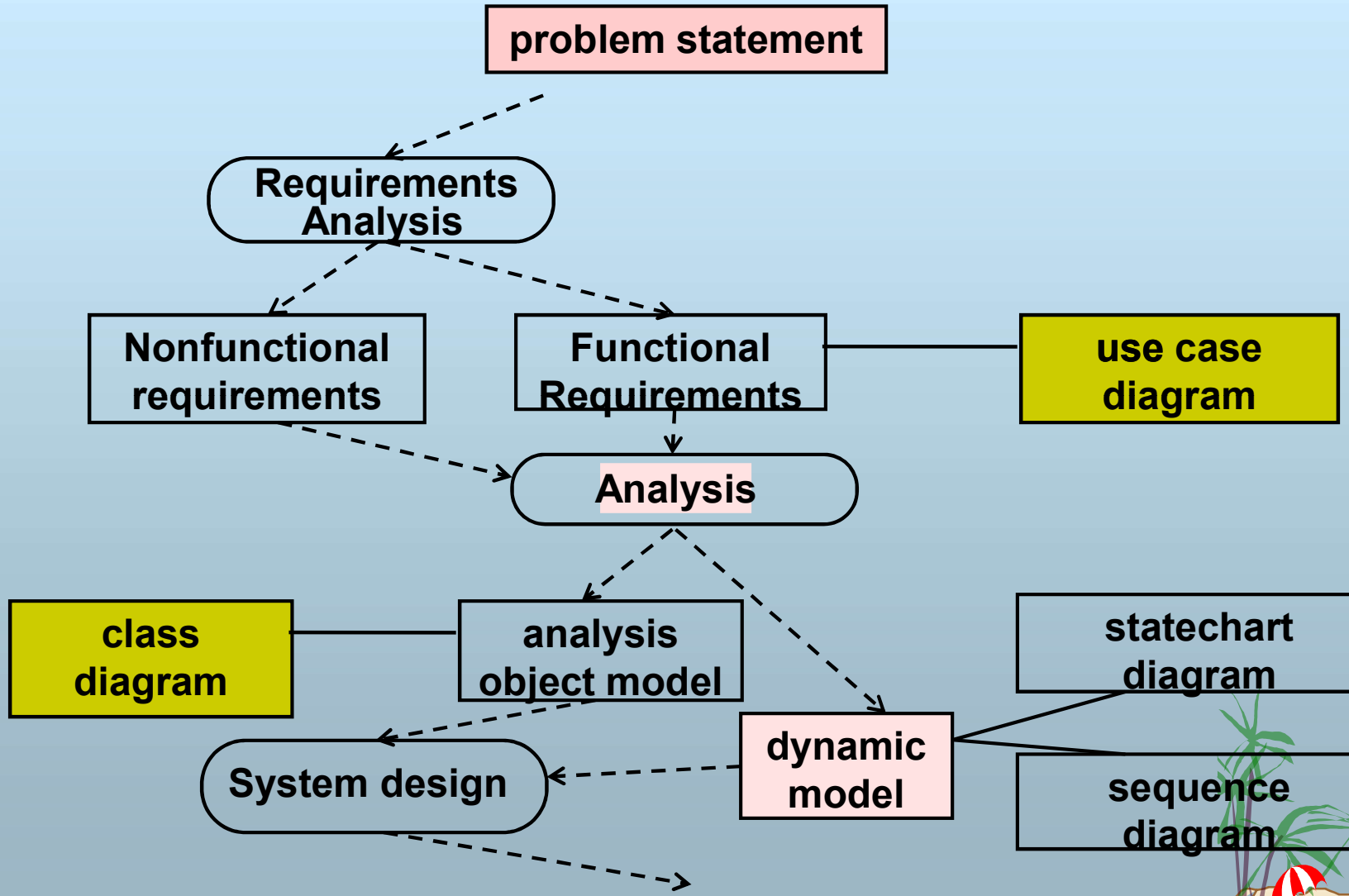
or

Class Diagram can be taken up directly..

Use case diagram or Class Diagram ?

- The matter regarding use case First or Class Diagram was though very discussed in early days of UML development.
  - **The Objectivity method (1992) by Jacobson has major contribution in the UML and they believed that with the idea that analysis should start with use cases rather than with a class model.**
  - **The classes were then to be derived from the use cases.**
  - **The technique marked an important step forward in object-oriented analysis and has been widely adopted.**

Software Project development and OOM



The approach of not Starting with Use Case diagram and directly starting with Class Diagram will have serious consequences…

Because without understanding User's views and without its' Analysis, the Design of the Software so developed based on Class Diagram may seriously *lack in user's perspective* with software design…and may be the reason for its failure after deployment…

OO Design & Class Diagram

Creating Classes based on the analysis of Use-Case diagrams will be significant step towards successful project design.

- Class Diagrams should be thus derived from study of Use case diagrams…

- Creating Class diagram requires thinking at appropriate level of abstraction !
- Making sure that our design is good means our Software developed thus fulfills objectives and avoids characteristics of a bad design.

How to avoid Bad Software Design..

- Our Software should avoid Bad Design !!!
- The characteristics of a **bad design** :
  - **Rigidity** - It is hard to change because every change affects too many other parts of the system.
  - **Fragility** - When you make a change, unexpected parts of the system break.
  - **Immobility** - It is hard to reuse in another application because it cannot be disentangled from the current application.
- We will study in detail regarding **How to avoid bad Software Design**

Design phase

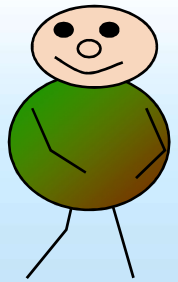
- **Design** : specifying the structure of how a software system will be written and function, without actually writing the complete implementation
  - A transition from "**what**" the system must do, to "**how**" the system will do it –
- What classes will we need to implement a system that meets our requirements?
- What fields and methods will each class have?
- How will the classes interact with each other?

Class Diagram represents the program structure in brief…

And does not depend on the programming language..

Class Diagrams: Ignoring detail to get the high level structure right

C++ code of the Person class



```
#include <iostream>
class Person{
  private:
  char name[20];
  int yearOfBirth;
  public:
Person(char *theName, int theYear) {
    yearOfBirth = theYear;
    sprintf(name, theName);
    }
    void jump() {
    std::cout << name <<
      " jumps" << std::endl;
    }
  int getAge(int thisYear) {
  return thisYear - yearOfBirth;
    }
};
```

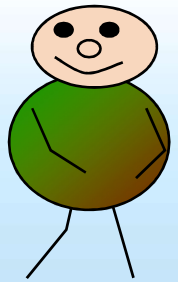
Name

Private data (attributes)

Initialisation (constructor)

Public behaviour (methods)

Java code of the Person class



```
public class Person {

    private String name;
    private int yearOfBirth;

    public Person(String theName, int theYear)
    {
        name = theName;
        yearOfBirth = theYear;
    }

    public void jump() {
        System.out.println(name + " jumps");
    }

    public int getAge(int thisYear) {
        return thisYear - yearOfBirth;
    }
}
```

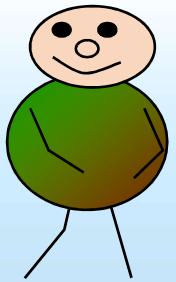
Name

Private data (attributes)

Initialisation (constructor)

Public behaviour (methods)

UML diagram of the Person class



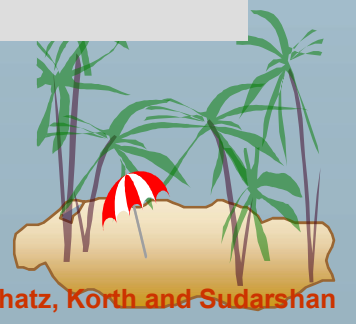
Person
-name: string -yearOfBirth: int
+create(theName: string ,theYear: int) +jump() +getAge(this year: int):int

Name

Private data (attributes)

Initialisation (constructor)

Public behaviour (methods)



Class Diagrams

- Classes
  - Basic Class Components
  - Attributes and Operations
- Class Relationships
  - Associations
  - Generalizations
  - Aggregations and Compositions

Basic Class Compartments

- Name
- Attributes
  - represent the state of an object of the class
  - are descriptions of the structural or static features of a class
- Operations
  - define the way in which objects may interact
  - are descriptions of behavioural or dynamic features of a class

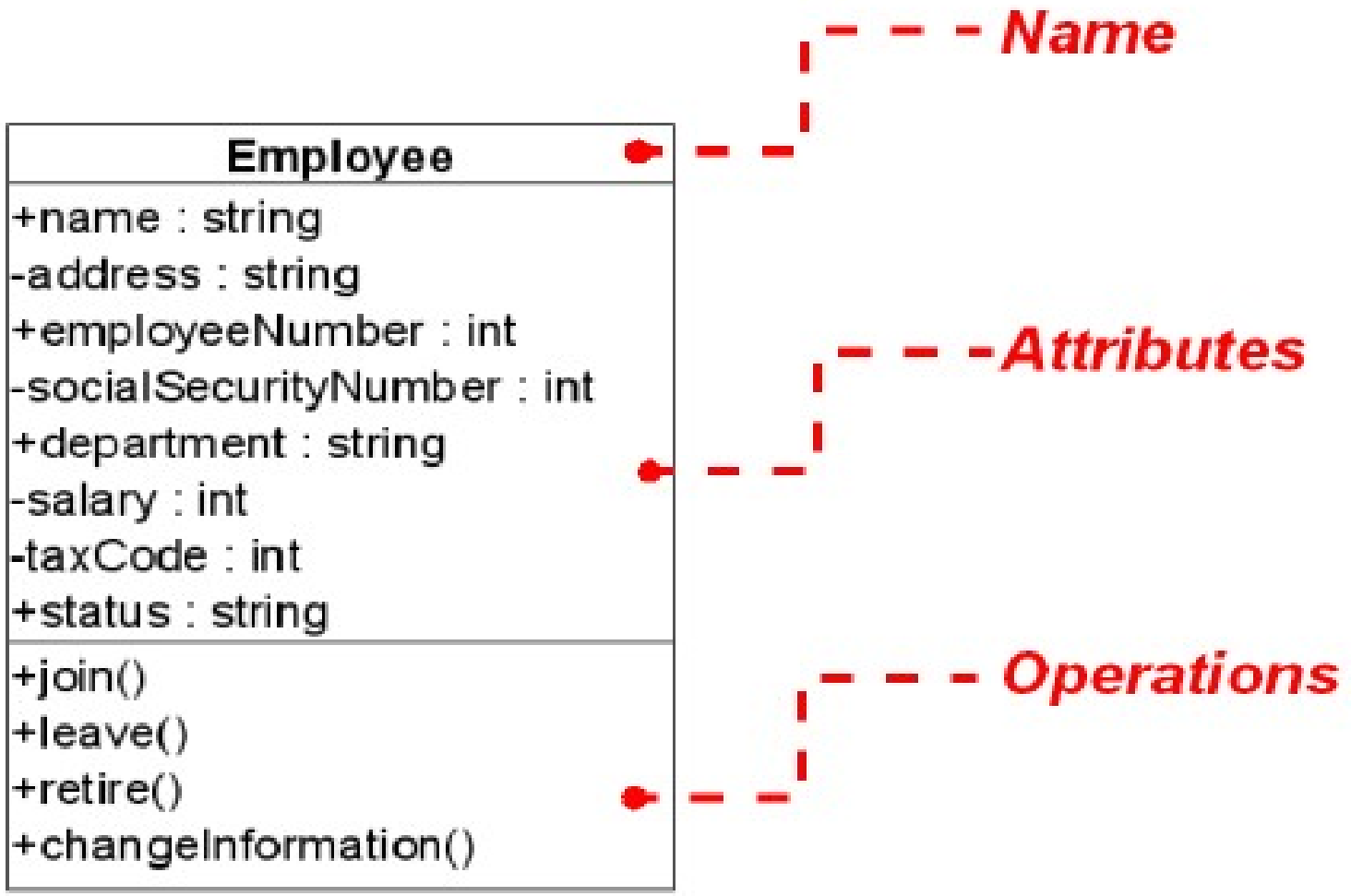
Class diagrams

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

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

Basic Class Compartments

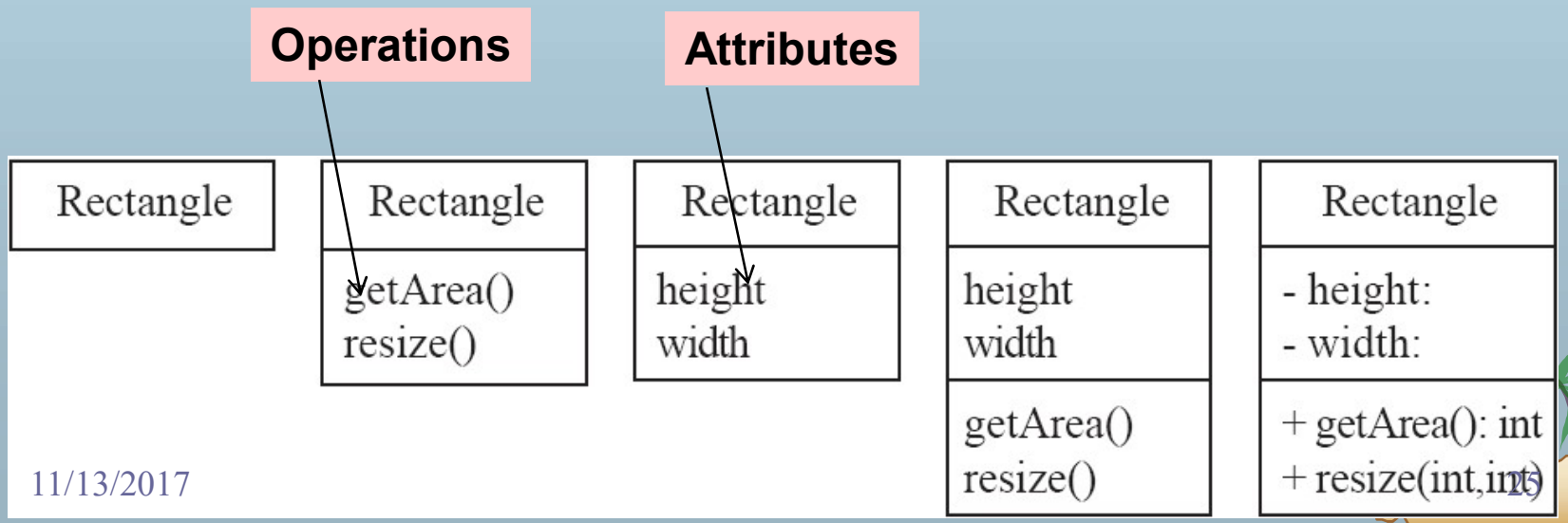


Classes

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



Class diagram

UML class diagram is a rectangle divided into three parts: class name, attributes, and operations.

Names of abstract classes, such as *Payment*, are in italics.

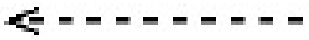
Relationships between classes are the connecting links.

association -- a relationship between instances of the two classes.

There is an association between two classes if an instance of one class must know about the other in order to perform its work.

In a diagram, an association is a link connecting two classes.

Class Relationships

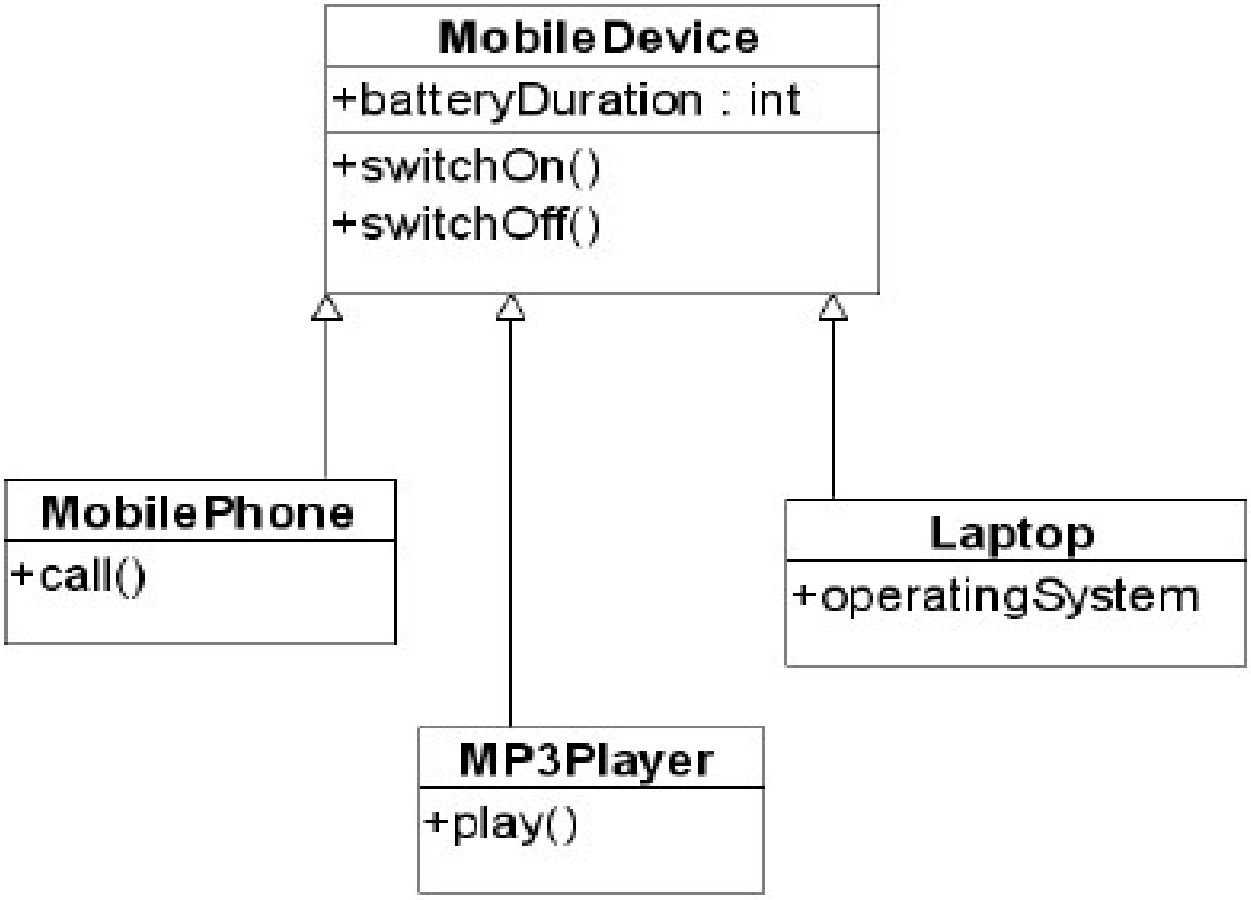
Relationship	Description
	**Dependency:** objects of one class work briefly with objects of another class
	Association: objects of one class work with objects of another class for some prolonged amount of time
	**Aggregation:** one class owns but share a reference to objects of other class
	**Composition:** one class contains objects of another class
	**Generalization (Inheritance):** one class is a type of another class

Steps in Generating Class Diagrams

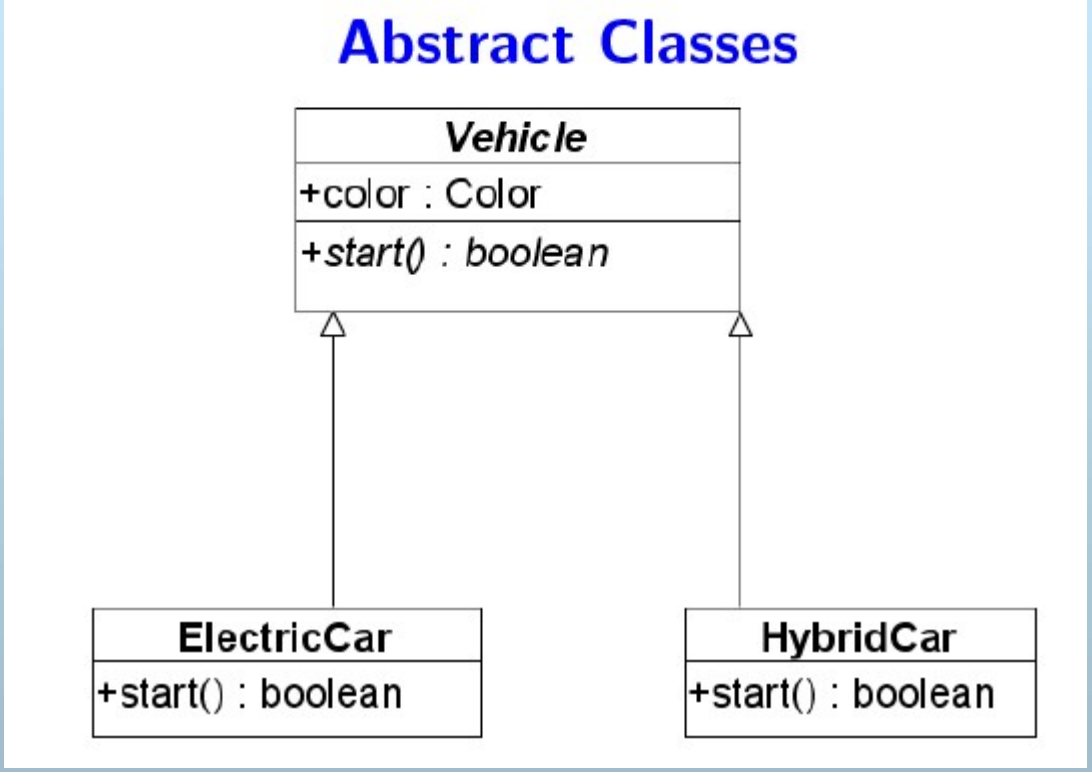
1. **Class identification** (textual analysis, domain expert) **based on Use Case diagram..**
2. **Identification of attributes and operations** (sometimes before the classes are found!)
3. **Identification of associations between classes**
4. **Identification of multiplicities & roles**
5. **Identification of inheritance**

What does following Class Diagram say ?

Generalization (Inheritance)



How to show Abstract Class

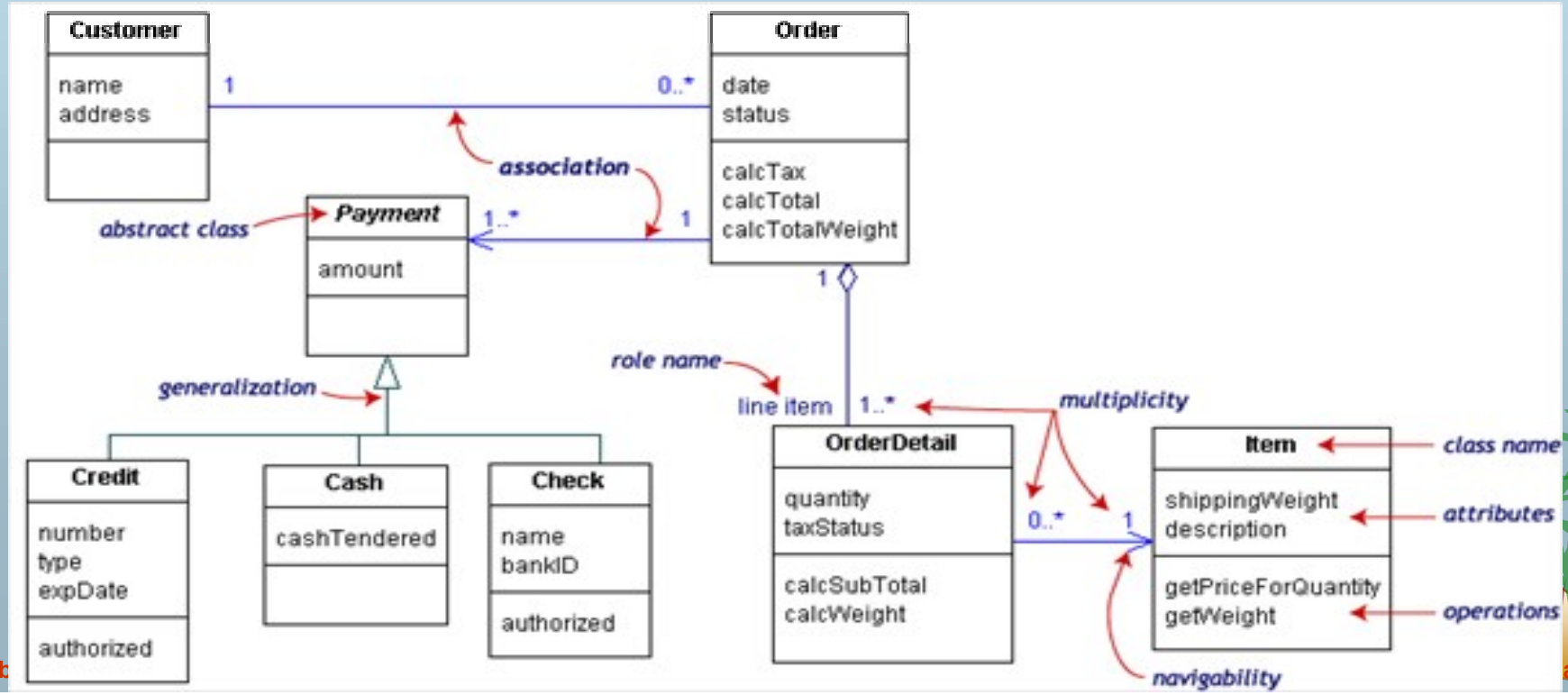


Vehicle is here shown as ABSTRACT class (shown in Italics) with child class as ElectricCar & HybridCar

Class diagram

The class diagram showed below **models a customer order from a retail catalog.**

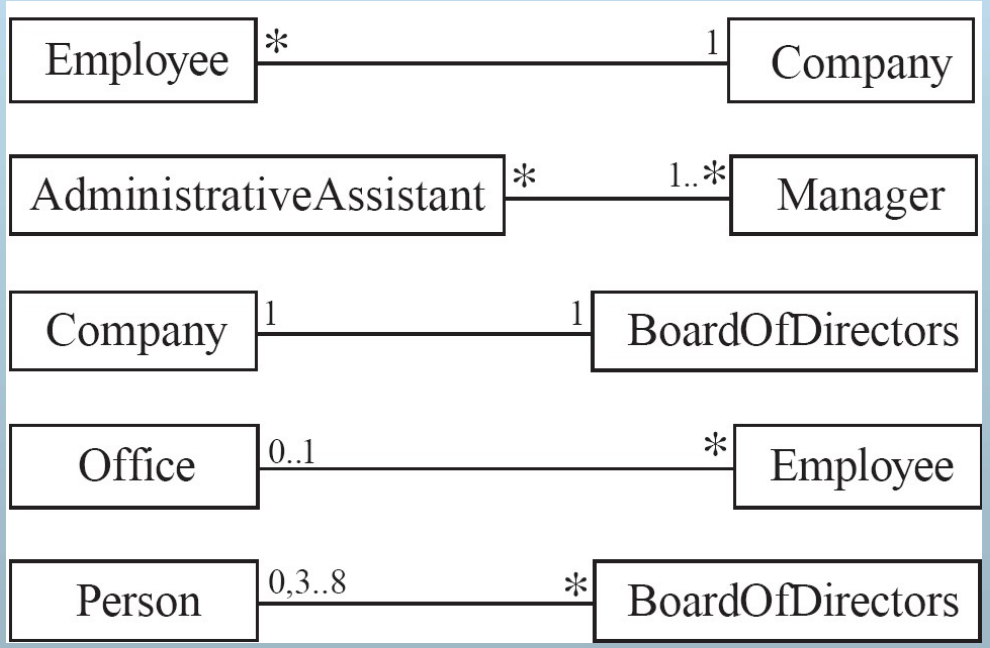
- The central class is the **Order**. Associated with it are the **Customer** making the purchase and the **Payment**. A **Payment** is Abstract Class, and one of three kinds: **Cash**, **Check**, or **Credit**. The order contains **OrderDetails** (line items), each with its associated **Item**.



Associations and Multiplicity

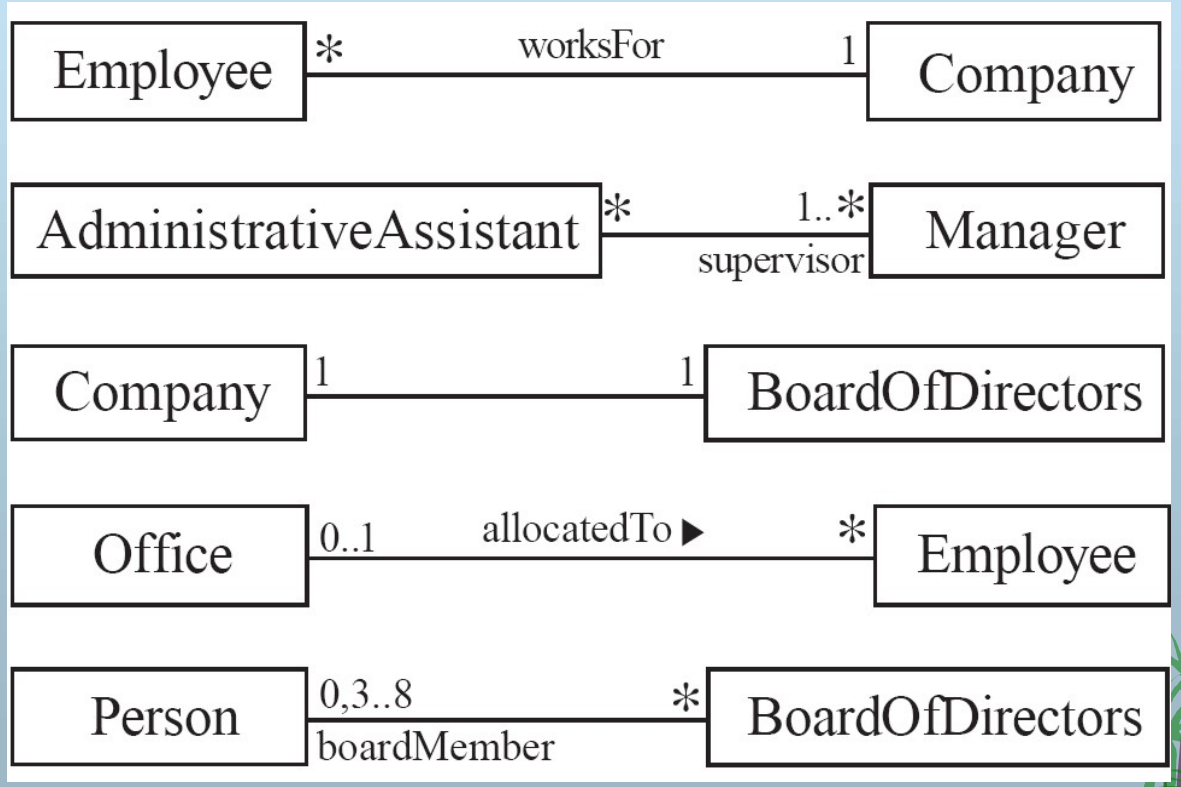
An ***association*** is used to show how two classes are related to each other

- Symbols indicating *multiplicity* are shown at each end of the association



Labelling associations

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



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**
  - An assistant can work for many managers
  - A manager can have many assistants
  - Assistants can work in pools
  - Managers can have a group of assistants
  - Some managers might have zero assistants.
  - Is it possible for an assistant 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

Who uses Class Diagrams?

- Purpose of class diagrams
  - The description of the **static** properties of a system.
  - Study and evaluate the Structure of the Software.
  - Evaluate the Software Design that whether it can be improved.

Who uses Class Diagrams?

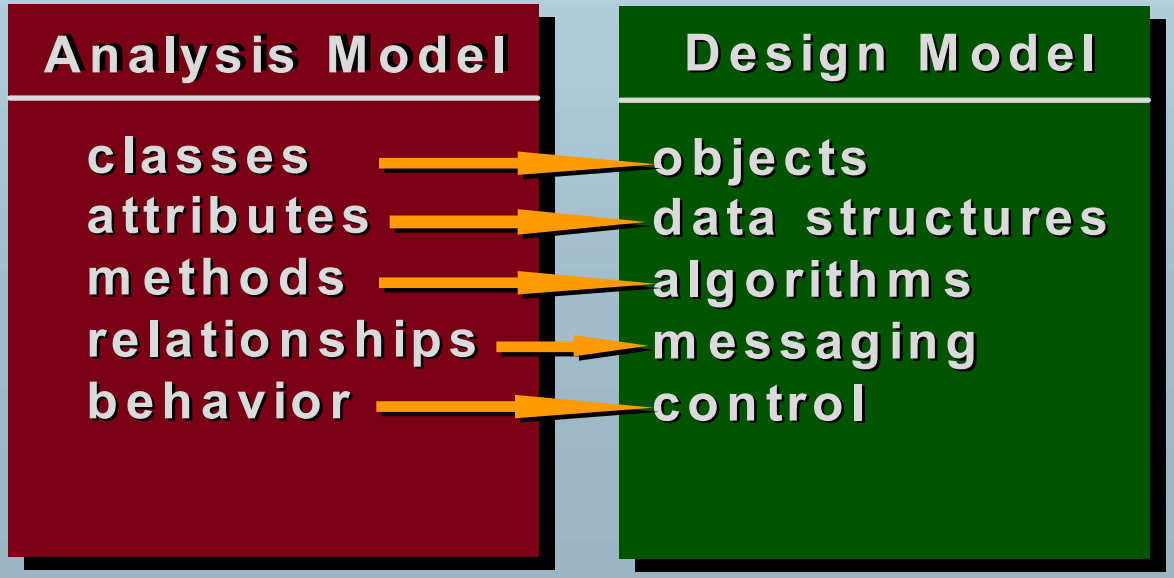
- The main users of class diagrams:
  - The application domain expert
    - uses class diagrams to model the application domain (including taxonomies)
      - during requirements elicitation and analysis
  - The developer
    - uses class diagrams during the development of a system
      - during analysis, system design, object design and implementation.

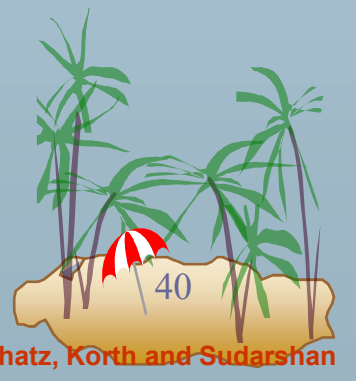
Who does not use Class Diagrams?

- The client and the end user are usually not interested in class diagrams
  - Clients focus more on project management issues
  - End users are more interested in the functionality of the system.

OOSE

- OO Software Engineering will involve mapping from Analysis Model to Design Models as well.
- OO Analysis will adapt different approach and techniques.
- OO Design will be done through various OO modeling standards.
- Analysis Model needs to be appropriately mapped with Design model for smooth transition of OOSE phases.





Class diagram pros/cons

- Class diagrams are great for:
  - discovering related data and attributes
  - getting a quick picture of the important entities in a system
  - seeing whether you have too few/many classes
  - seeing whether the relationships between objects are too complex, too many in number, simple enough, etc.
  - spotting dependencies between one class/object and another

- Not so great for:
  - discovering algorithmic (not data-driven) behavior
  - finding the flow of steps for objects to solve a given problem
  - understanding the app's overall control flow (event-driven? web-based? sequential? etc.)

What are OO Design Criterion…?

How Class Diagrams can enable creation of good quality design?

OOM for Software Project: ATM

- Creating Quality Classes
- Class Diagrams: Difficulties & Risks
- Class Responsibility Collaboration
- Requirement Analysis & UI
- Class Analysis
  - CRC & Class Diagram

OOM Design: Creating Quality Classes

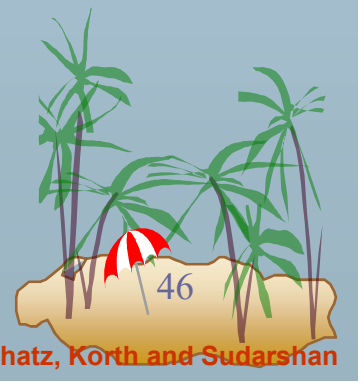
- Creating **Classes based on the analysis** of Use-Case diagrams will be significant step towards successful project design.
- Creating **Class diagram** requires thinking at appropriate level of abstraction !
- Assigning **appropriate responsibility** to each Class requires analytical thinking.
- The **Quality of Classes** should be kept high from the Design perspective.
  - **Functional Independence**

Functional Independence

- The main goal (almost not evident) of design (from a low-level perspective) is to minimize **coupling** and maximize **cohesion.**
  - **Coupling** is the level of **interdependency** between a **method and the environment**, (other methods, objects and classes), while **cohesion** is the level of uniformity of the method goal.
- **While coupling needs a low-level perspective, cohesion needs an higher point of view.**

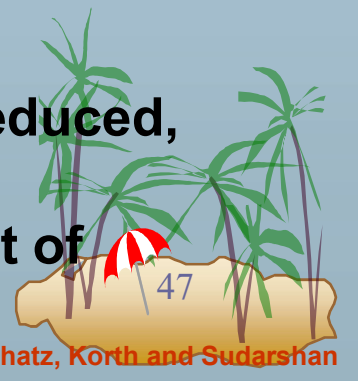
OO Design

- If two modules interchange large amounts of data, then they are highly interdependent.
- The degree of coupling between two modules depends on their interface complexity.
- A module having high cohesion and low coupling is said to be functionally independent of other modules.
- By the term functional independence, we mean that a cohesive module performs a single task or function.
- A functionally independent module has minimal interaction with other modules.
- The primary characteristics of neat module decomposition are high cohesion and low coupling.



Need for functional independence

- Functional independence is a key to any good design due to the following reasons:
  - **• Error isolation: Functional independence reduces error propagation. The reason behind this is that if a module is functionally independent, its degree of interaction with the other modules is less. Therefore, any error existing in a module would not directly effect the other modules.**
  - **• Scope of reuse: Reuse of a module becomes possible. Because each module does some well-defined and precise function, and the interaction of the module with the other modules is simple and minimal. Therefore, a cohesive module can be easily taken out and reused in a different program.**
  - **• Understandability: Complexity of the design is reduced, because different modules can be understood in isolation as modules are more or less independent of each other.**



Cohesion

- Classes contain certain data and exhibit certain behavior.
- Cohesive means that a certain class performs a set of *closely related actions.*
- A lack of cohesion means that a class is performing several unrelated tasks.
- Lack of cohesion may never have an impact on the overall functionality of a particular class—or of the application itself
- *However, the application software will eventually become unmanageable.*

Cohesion

- Classes contain certain data and exhibit certain behavior.
- Cohesive means that a certain class performs a set of *closely related actions.*
- A lack of cohesion means that a class is performing several unrelated tasks.
- Lack of cohesion may never have an impact on the overall functionality of a particular class—or of the application itself
- *However, the application software will eventually become unmanageable.*

Classification of Coupling

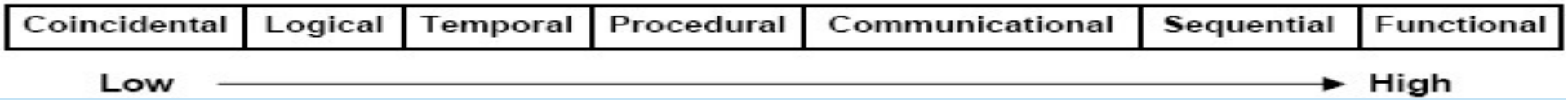
Data coupling: Two modules are data coupled, if they communicate through a parameter. An example is an elementary data item passed as a parameter between two modules, e.g. an integer, a float, a character, etc. This data item should be problem related and not used for other purpose.

Stamp coupling: Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.

Control coupling: Control coupling exists between two modules, if data from one module is used to direct the order of instructions execution in another. An example of control coupling is a flag set in one module and tested in another module. Common coupling: Two modules are common coupled, if they share data through some global data items.

Content coupling: Content coupling exists between two modules, if they share code, e.g. a branch from one module into another module..

Classification of Cohesion



➢**Coincidental cohesion**: A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all.

➢ In this case, the module contains a random collection of functions. It is likely that the functions have been put in the module out of pure coincidence without any thought or design.

➢**Functional cohesion**: Functional cohesion is said to exist, if different elements of a module cooperate to achieve a single function. For example, a module containing all the functions required to manage employees' pay-roll exhibits functional cohesion. Suppose a module exhibits functional cohesion and we are asked to describe what the module does, then we would be able to describe it using a single sentence.

OOM : design metrics

Main objectives :

- to build highly cohesive classes
- to maintain loose coupling between those classes.

- High-cohesion results in well-structured classes.
- Loose coupling means more flexible, extensible software.
- Applying this to the design and code helps determine whether project goals have been achieved.

Design Metrics: Cohesion & Coupling in OO

- Most researchers and engineers agree that a good software design implies clean decomposition of the problem into modules, and the neat arrangement of these modules in a hierarchy.
- It is desirable to have a "good" design finalized before we proceed for developing the program .
- For comparing various design options, we need to have good design metrics.

How to ensure Cohesion & Coupling

CRC
(Class Responsibility Collaboration)

Class-responsibility-collaboration (CRC) cards are a brainstorming tool used in the design of object-oriented software.

They were originally proposed by Ward Cunningham and Kent Beck **but are also popular among expert designers and recommended by extreme programming supporters.**

Martin Fowler has described CRC cards as a viable alternative to UML diagram for design

Class Responsibility Collaboration

- CRC cards are a brainstorming tool used in the design of object-oriented software:
- They are typically used when first determining which classes are needed and how they will interact.
- CRC cards are usually created from index cards on which are written:
  - The class name
  - Its Super and Sub classes (if applicable)
  - The responsibilities of the class.
  - The names of other classes with which the class will collaborate to fulfill its responsibilities.
  - Author

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

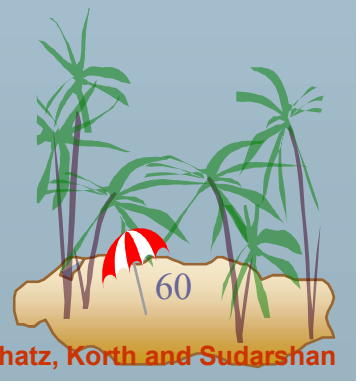
- Creating and initializing new instances
- Loading to and saving from persistent storage
- Destroying instances
- Setting and getting the values of attributes
- Computing numerical results
- Adding and deleting links of associations
- Copying, converting, transforming, transmitting or outputting
- Navigating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

Categories of responsibilities

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

OOPS & Software Re-Use

- OOPs characteristics like Inheritance, Abstraction and polymorphism can be very useful in Software Reuse.
- Java provides many features in terms of Core / Foundation classes and various APIs which can ease and encourage the Software Re-use.
- Creation of user friendly GUIs, Java Applets and many other new generation Software development applications / paradigm are based on effective Re-use while incorporating OOPS characteristics like Inheritance, Abstraction and polymorphism



A CRC card lists

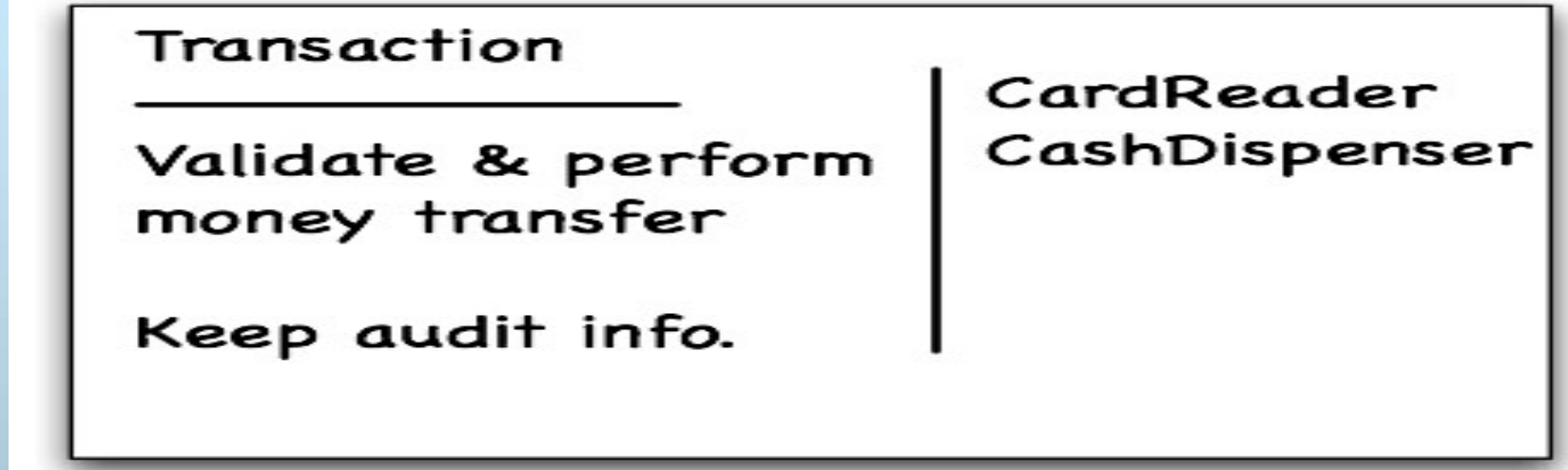
- a Class,the Responsibilities of the class, andthe Collaborators of the class.
- We will use index cards for this.
- **The class name appears underlined in the upper-left hand corner,**
  - **a list of responsibilities appears under it**
  - **in the left two-thirds of the card,**
  - **and the list of collaborators appears in the right third.**
- Using a small card keeps the complexity of the design at a minimum.

- **The *class name* of an object creates a vocabulary for discussing a design.**
  - The *responsibilities* of an object are expressed by a handful of short verb phrases, each containing an active verb. *Collaborators* are objects which will send or be sent messages in the course of satisfying responsibilities.

Example :

CRC cards using the scenario of a banking

ATM



ATM : Analysis for Classes

- As initial in the Use Case diagram, now Classes needs to be analyzed;
- A controller object representing the ATM itself (managing the boundary objects as listed)
  - Controller objects corresponding to use cases.

1) Session
2) Transaction

- Boundary objects representing the individual component parts of the ATM:
  - Operator panel.
  - Card reader.
  - Customer console, consisting of a display and keyboard.
  - Network connection to the bank.
  - Cash dispenser.
  - Envelope acceptor. & Receipt printer.

Class Analysis

- **Controller objects; class ATM can handle the Startup and Shutdown use cases itself, so these do not give rise to separate objects here.**
  - Session
  - Transaction (*abstract generalization, responsible for common features, with concrete specializations responsible for type-specific portions*)
- **An entity object representing the information encoded on the ATM card inserted by customer.**
- **An entity object representing the log of transactions maintained by the machine.**

CRC

- Using CRC cards to assign responsibilities to various classes for the tasks required by the various use cases leads to the creation of the following cards.
  - Class ATM
  - Controller objects corresponding to the various use cases
    - Class Session
    - Class Transaction
    - Class Withdrawal
    - Class Deposit
    - Class Transfer
    - Class Inquiry
- What about relations among these Classes?
- Entity objects found necessary when assigning responsibilities to other object

CRC for Class ATM

Responsibilities	Collaborators
Start up when switch is turned on	OperatorPanel CashDispenser NetworkToBank
Shut down when switch is turned off	NetworkToBank
Start a new session when card is inserted by customer	CustomerConsoleSession
Provide access to component parts for sessions and transactions	CustomerConsoleSession

Class Session

- Responsibilities
  Perform session use case

  Collaborators
  ATM
  CardReader
  Card
  CustomerConsole
  Transaction

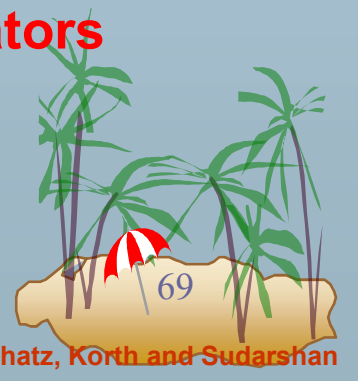
  Update PIN value if customer has to re-enter it

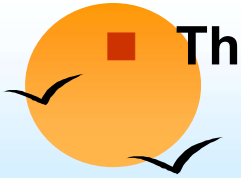
CRC: Class CardReader

Responsibilities	Collaborators
Tell ATM when card is inserted	**ATM**
Read information from card	**Card**
Eject card	
Retain card	

- **Class Card**

Responsibilities	Collaborators
Represent information encoded on customer's ATM card	





The following links can be used to go directly to the CRC cards for the various classes:

Class ATM

Responsibilities	Collaborators
Start up when switch is turned on	OperatorPanel CashDispenser NetworkToBank
Shut down when switch is turned off	NetworkToBank
Start a new session when card is inserted by customer	CustomerConsoleSession
Provide access to component parts for sessions and transactions	

Class CardReader

Responsibilities	Collaborators
Tell ATM when card is inserted	ATM
Read information from card	Card
Eject card	
Retain card	

Class CashDispenser

Responsibilities	Collaborators
Keep track of cash on hand, starting with	Log
initial amount Report whether enough	
cash is available Dispense cash	

Class Customer Console

Responsibilities	Collaborators
Display a message	
Display a prompt, accept a PIN from keyboard	
Display a prompt and menu, accept a choice from keyboard	
Display a prompt, accept a dollar amount from keyboard	
Respond to cancel key being pressed by customer	

Class EnvelopeAcceptor

-
- Responsibilities — Collaborators
- Accept envelope from customer; — Log
- report if timed out or cancelled

Responsibilities	Collaborators
Log messages sent to bank	
Log responses from bank	
Log dispensing of cash	
Log receiving an envelope	

Class NetworkToBank

Responsibilities	Collaborators
Initiate connection to bank at startup	Message
Send message to bank and wait for response	Log
Terminate connection to bank at shutdown	Balances Status

Class OperatorPanel

Responsibilities	Collaborators
Inform ATM of changes to state of switch	**ATM**
Allow operator to specify amount of initial cash	

Class Session

- **Responsibilities**
- Perform session use case
  Update PIN value if customer has to
- re-enter it

Collaborators

ATM
CardReader
Card
CustomerConsole
Transaction

Abstract Class Transaction

Responsibilities	Collaborators
Allow customer to choose a type of **transaction**	ATM CustomerConsole Withdrawal Deposit Transfer Inquiry
Perform Transaction Use Case	ATM CustomerConsole Withdrawal Deposit Transfer Inquiry Message NetworkToBank Receipt ReceiptPrinter
Perform invalid PIN extension	CustomerConsole Session CardReader

Class Diagram

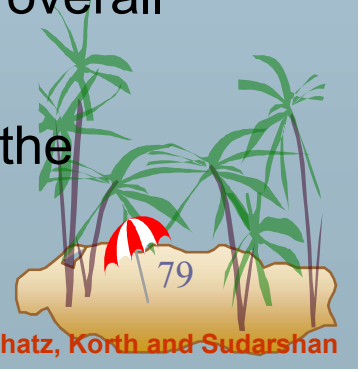
- In the ATM system, one of the responsibilities of the ATM is to provide access to its component parts for Session and Transaction objects; thus, Session and Transaction have associations to ATM, which in turn has associations to the classes representing the individual component parts.
- (Explicit "uses" links between Session and Transaction, on the one hand, and the component parts of the ATM, on the other hand, have been omitted from the diagram to avoid making it excessively cluttered.)
- The need for the various classes in the diagram was discovered at various points in the design process.

Class diagram

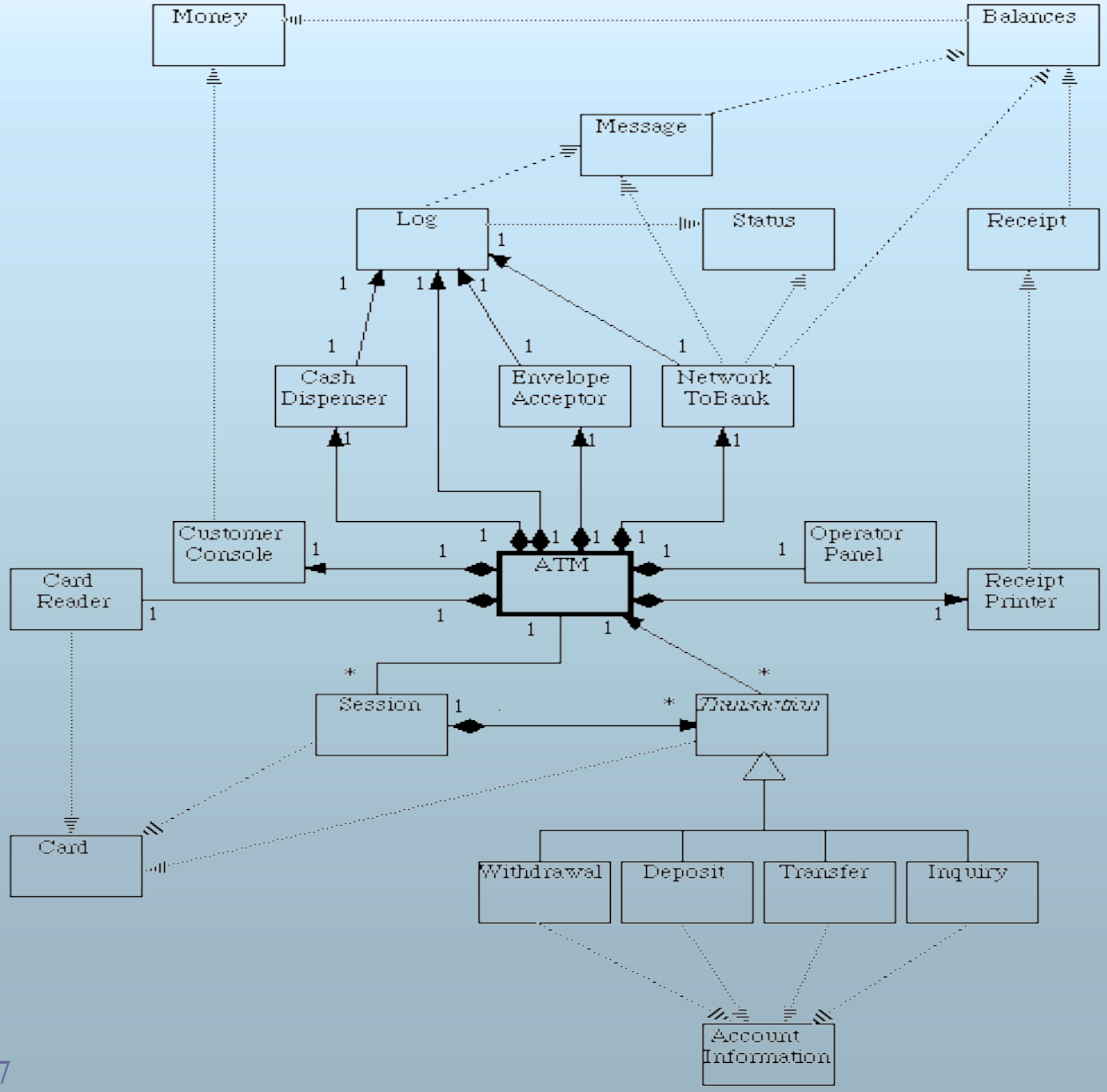
- Some classes were discovered when doing analysis
- Some classes were discovered when doing CRC cards
  - Message - used to represent a message to the bank.
  - Receipt - used to encapsulate information to be printed on a receipt.
  - Status - used to represent return value from message to the bank.
  - Balances - used to record balance information returned by the bank

Class diagram

- Some Classes were discovered when doing detailed design or writing code
  - Money - used to represent money amounts, in numerous places.
  - AccountInformation - contains names of various types of accounts customer can choose from
- That is, OO design is not a "waterfall" process - discoveries made when doing detailed design and coding can impact overall system design.
- To prevent the diagram from becoming overly large, only the name of each class is shown - the attribute and behavior "compartments" are shown in the detailed design, but are omitted here.



ATM Class diagram



Class Diagrams are

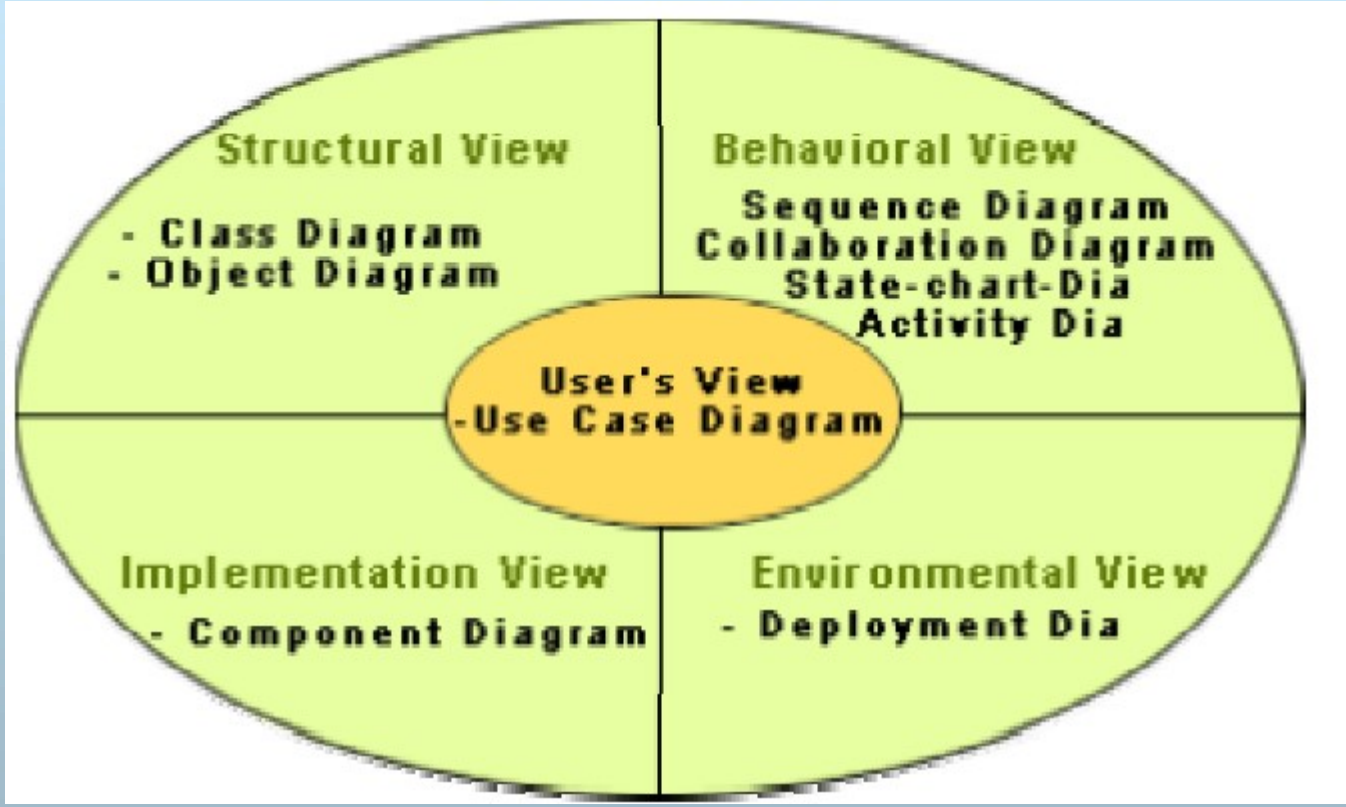
Static Diagrams…

but

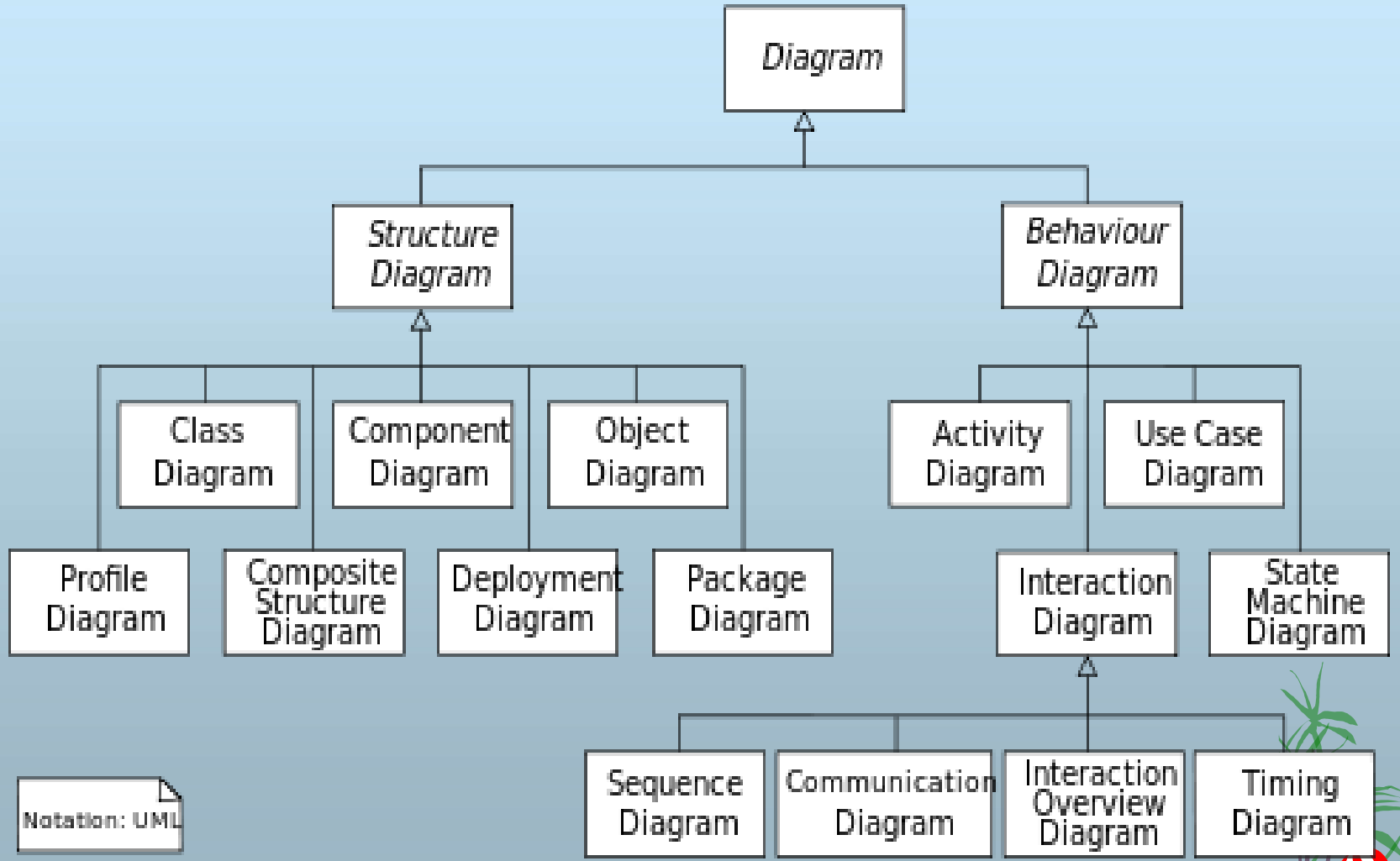
there are

Dynamic diagrams

Modeling as UML :Static & Dynamic

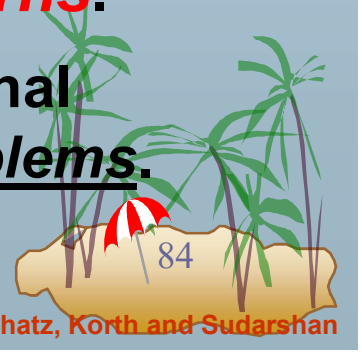


UML Diagrams



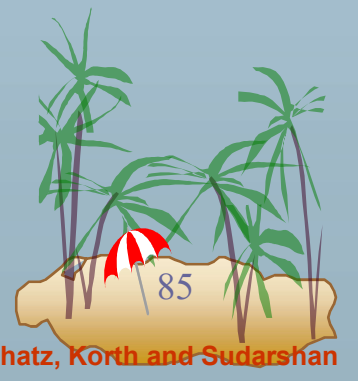
Software Design Vs Design Patterns

- Class diagram forms the significant basis of Software design.
  - **Class diagram gives an overview of a system by showing its classes and the relationships among them.**
- Software design researches are also influenced by the development: Design Patterns are **general reusable solution to a commonly occurring problem in software design. It is a description or template for how to solve a problem that can be used in many different situations.**
- **Design patterns provide tested, proven development paradigms.**
  - ***All software patterns are not design patterns*.**
    - **For instance, algorithms solve computational problems rather than *software design problems*.**



Design Patterns

- GoF proposed some significant concepts;
  - *"Program to an '**interface**', not an '**implementation**'." (Gang of Four 1995:18)*
  - *"Favor '**object composition**' over '**class inheritance**'." (Gang of Four 1995:20)*
- The GoF refer to inheritance as ***white-box reuse***, with white-box referring to visibility, because the internals of parent classes are often visible to subclasses.
- In contrast, the authors refer to object composition as ***black-box reuse*** because no internal details of composed objects need be visible in the code using them.



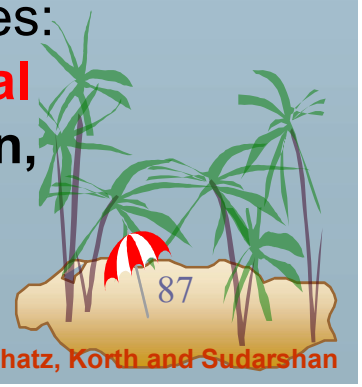
GoF & Design Patterns

- **Design Patterns proposed in the popular Book**
  - **Authors are Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides with a foreword by Grady Booch. They are often referred to as the *Gang of Four*, or *GoF.***
  - **Describing 23 classic software design patterns.**
  - **Regarded as an important source for object-oriented design theory and practice.** *More than 500,000 copies have been sold in English and in 13 other languages*

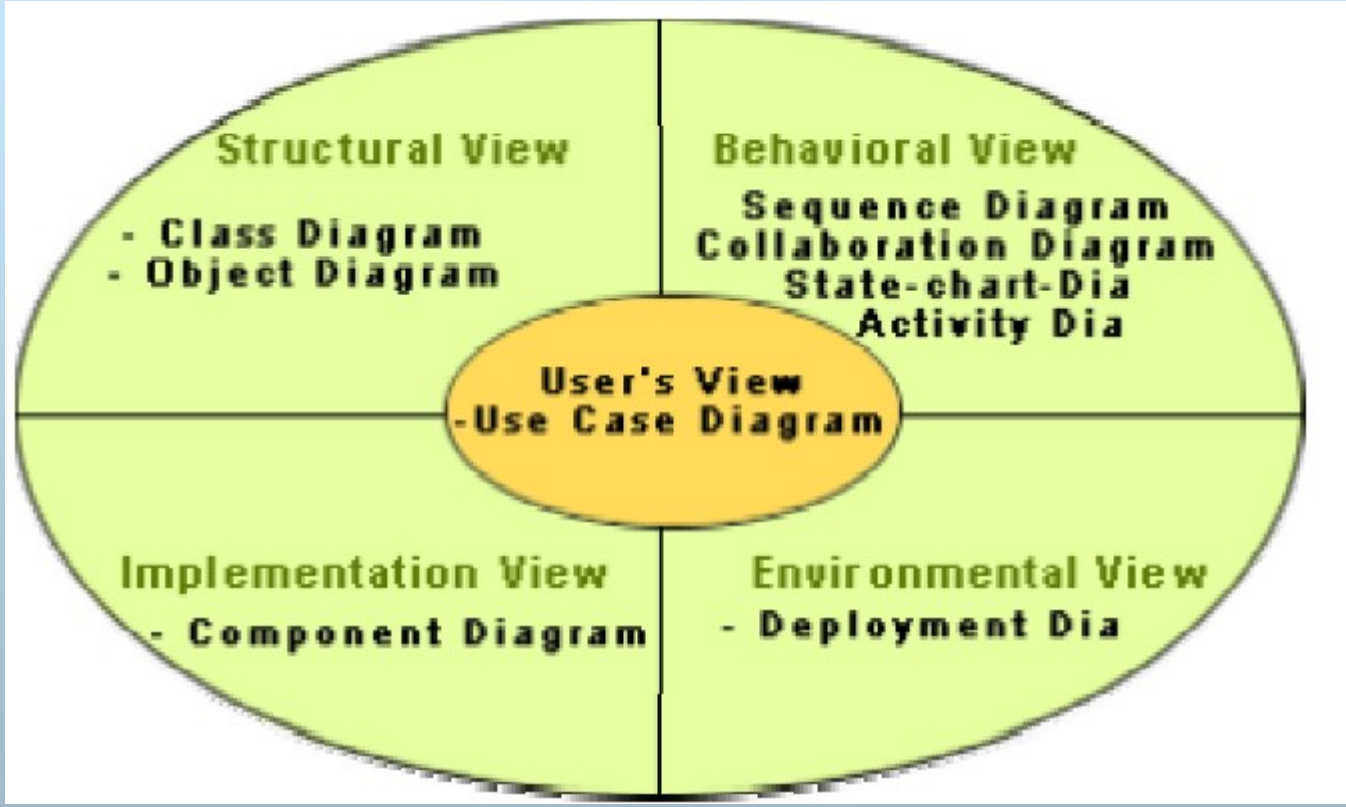
- **Effective software design requires considering issues that *may not become visible* until later in the implementation and thus Design Patterns provides a tested Solution paradigm to many often occurring situations.**

Design Patterns

- **Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.**
- **Design patterns reside in the domain of modules and interconnections.**
- ***Not all software patterns are design patterns*. For instance, algorithms solve computational problems rather than software design problems.**
- **At a higher level there are architectural patterns that are larger in scope, usually describing an *overall pattern followed by an entire system.***
- Design patterns were originally grouped into the categories: **creational patterns, structural patterns, and behavioral patterns**, and described using the concepts of **delegation, aggregation,** and **consultation.**

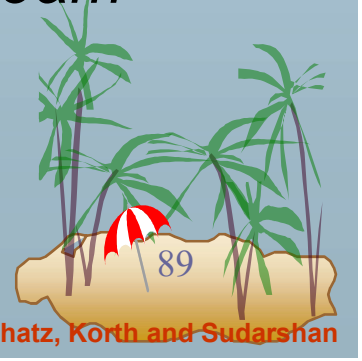


Modeling as UML :Static & Dynamic



Difficulties and Risks when creating class diagrams

- Modeling is particularly difficult skill
  - *Even excellent programmers have difficulty thinking at the appropriate level of abstraction*
  - *Education traditionally focus more on programming than modeling*
- Resolution:
  - *Ensure that tem members have adequate training*
  - *Have experienced modeler as part of the team*
  - *Review all models thoroughly*

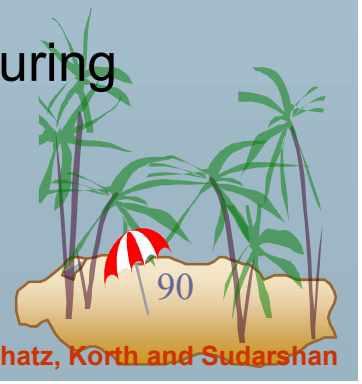


Why dynamic modeling

As the software system continuously changes (in *states, activities, and collaborations*) during life time, therefore it is needful to model it and following are grouped in the dynamic modeling diagrams of UML;

- **Activities**
- **States**
- **Sequences**
- **Collaborations**

These diagram shows how **objects interact dynamically**, during execution time of the system.



Dynamic modeling diagrams

- There are following four diagrams, who represent the behavioral view of the system and as behavior is a dynamic phenomenon, the diagrams are also categorized as Dynamic modeling diagrams

1 Activity diagrams – another way to show interaction but the focus is on work.

2 State diagrams – describes different states an object can have during its life cycle, behavior in the states, events that can change states

3 Sequence diagrams – describes how objects interact and communicate with each other with the focus on time

4. Collaboration diagrams – also describes how objects interact, but the focus is on space

Activity Diagrams

- Activity diagram is another important diagram in UML to describe the dynamic aspects of the system.
- Activity diagram is basically a flowchart to represent the flow from one activity to another activity. The activity can be described as an operation of the system.
- The control flow is drawn from one operation to another. This flow can be sequential, branched, or concurrent. Activity diagrams deal with all type of flow control by using different elements such as fork, join, etc

Purpose of Activity Diagrams

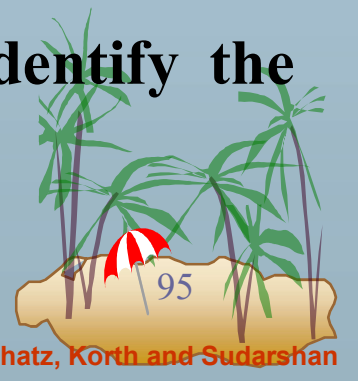
- The basic purposes of activity diagrams is similar to other four diagrams. It captures the dynamic behavior of the system. Other four diagrams are used to show the message flow from one object to another but activity diagram is used to show message flow from one activity to another.
- Activity is a particular operation of the system. Activity diagrams are not only used for visualizing the dynamic nature of a system, but they are also used to construct the executable system by using forward and reverse engineering techniques. The only missing thing in the activity diagram is the message part.
- It does not show any message flow from one activity to another. Activity diagram is sometimes considered as the flowchart. Although the diagrams look like a flowchart, they are not. It shows different flows such as parallel, branched, concurrent, and single

The purpose of an activity diagram 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.

How to Draw an Activity Diagram?

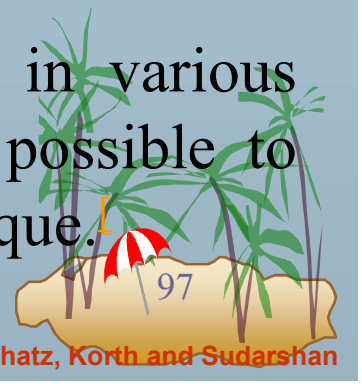
- Activity diagrams are mainly used as a flowchart that consists of activities performed by the system. Activity diagrams are not exactly flowcharts as they have some additional capabilities. These additional capabilities include branching, parallel flow, swimlane, etc.
- Before drawing an activity diagram, we must have a clear understanding about the elements used in activity diagram. The main element of an activity diagram is the activity itself. An activity is a function performed by the system. After identifying the activities, we need to understand how they are associated with constraints and conditions.
- Before drawing an activity diagram, we should identify the following elements −



Activity diagrams

- **Activity diagrams** are graphical representations of workflows of stepwise activities and actions with support for choice, **iteration and concurrency.**
- **Activity diagrams** are constructed from a limited number of **shapes, connected with arrows**. The most important shape types:
- *rounded rectangles* represent *actions*;
- *diamonds* represent *decisions*;
- *bars* represent the start (*split*) or end (*join*) of concurrent activities;
- a *black circle* represents the start (*initial node*) of the workflow;
- an *encircled black circle* represents the end (*final node*).

- Arrows run from the start towards the end and represent the order in which activities happen.
- Activity diagrams may be regarded as a form of flowchart. Typical flowchart techniques lack constructs for expressing concurrency. However, the join and split symbols in activity diagrams only resolve this for simple cases; the meaning of the model is not clear when they are arbitrarily combined with decisions or loops.
- While in UML 1.x, activity diagrams were a specialized form of state diagrams, in UML 2.x, the activity diagrams were reformalized to be based on Petri net-like semantics, increasing the scope of situations that can be modeled using activity diagrams These changes cause many UML 1.x activity diagrams to be interpreted differently in UML 2.x.
- UML activity diagrams in version 2.x can be used in various domains, e.g. in design of embedded systems. It is possible to verify such a specification using model checking technique.

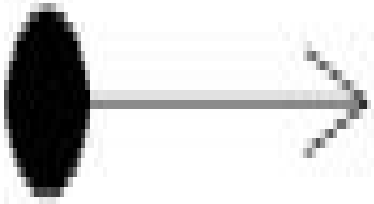


Fork vertices

- A swim lane (or **swimlane diagram**) is a visual element used in process flow **diagrams**, or flowcharts, that visually distinguishes job sharing and responsibilities for sub-processes of a business process.
- **Swim lanes** may be arranged either horizontally or vertically
- **Fork** vertices in the UML State chart **Diagram** serve to split an incoming transition into two or more transitions terminating on orthogonal target vertices.
- A **Fork** notation in a UML **Activity Diagram** is a control node that splits a flow into multiple concurrent flows.

Basic Activity Diagram Notations and Symbols

- **Initial State or Start Point**
- A small filled circle followed by an arrow represents the initial action state or the start point for any activity diagram. For activity diagram using swimlanes, make sure the start point is placed in the top left corner of the first column.



Start Point/Initial State

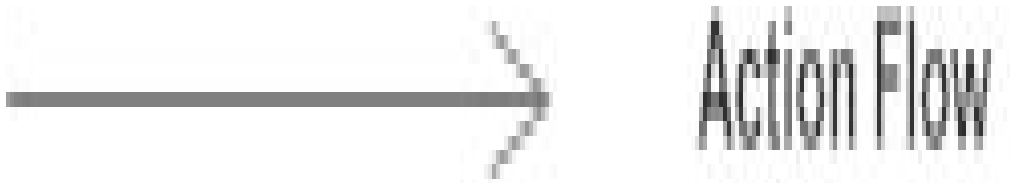
Activity or Action State

- An action state represents the non-interruptible action of objects. You can draw an action state in SmartDraw using a rectangle with rounded corners.

Activity

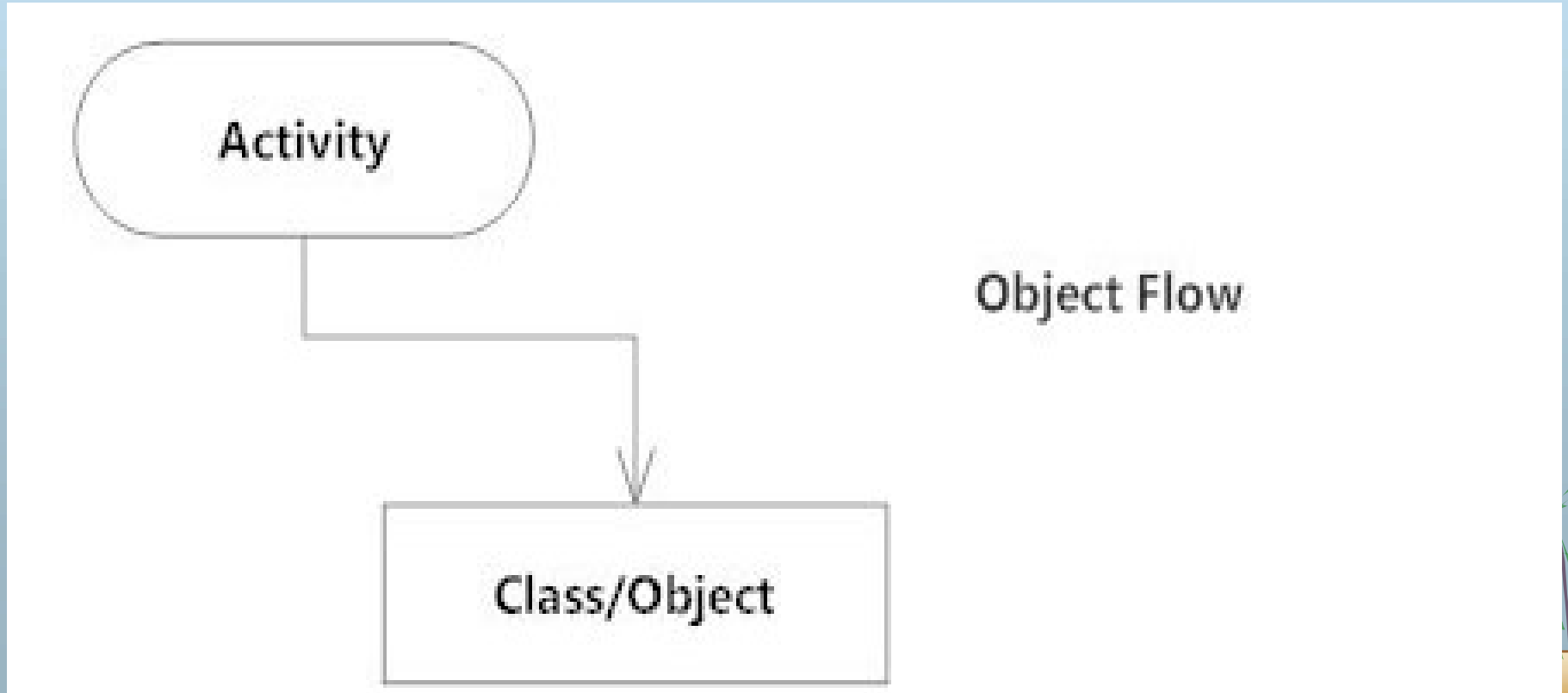
Action Flow

- Action flows, also called edges and paths, illustrate the transitions from one action state to another. They are usually drawn with an arrowed line.



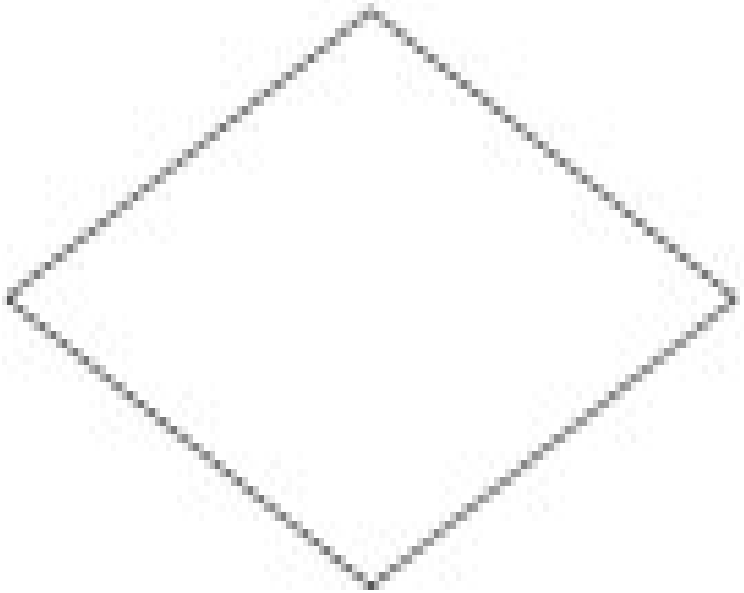
Object Flow

- Object flow refers to the creation and modification of objects by activities. An object flow arrow from an action to an object means that the action creates or influences the object. An object flow arrow from an object to an action indicates that the action state uses the object.



Decisions and Branching

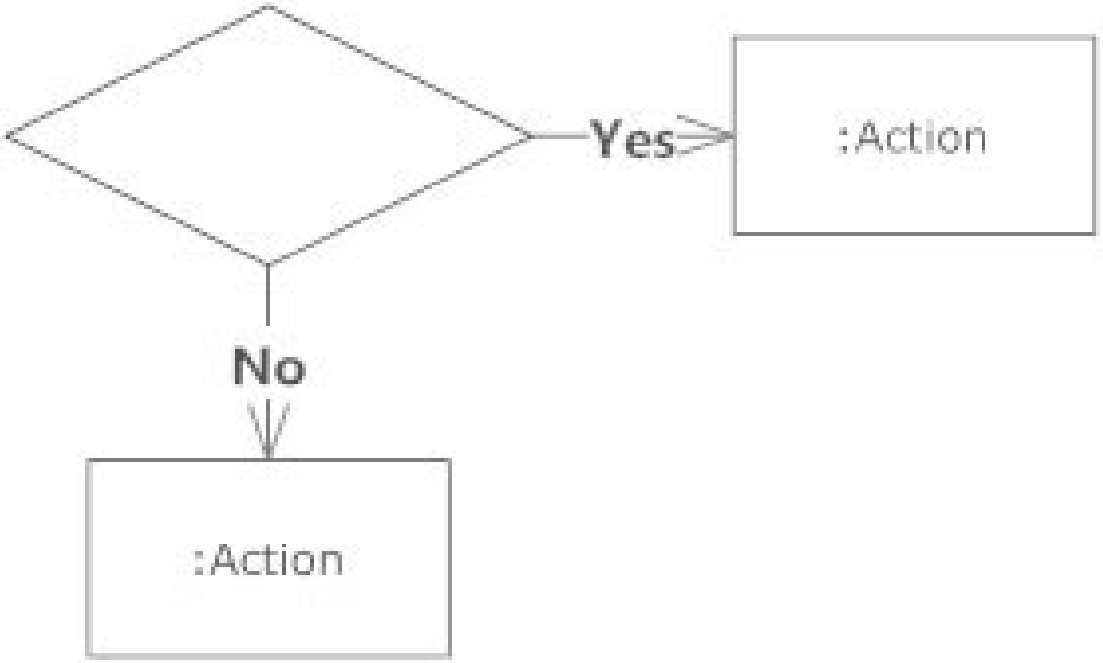
- A diamond represents a decision with alternate paths. When an activity requires a decision prior to moving on to the next activity, add a diamond between the two activities. The outgoing alternates should be labeled with a condition or guard expression. You can also label one of the paths "else."



Decision Symbol

Guards

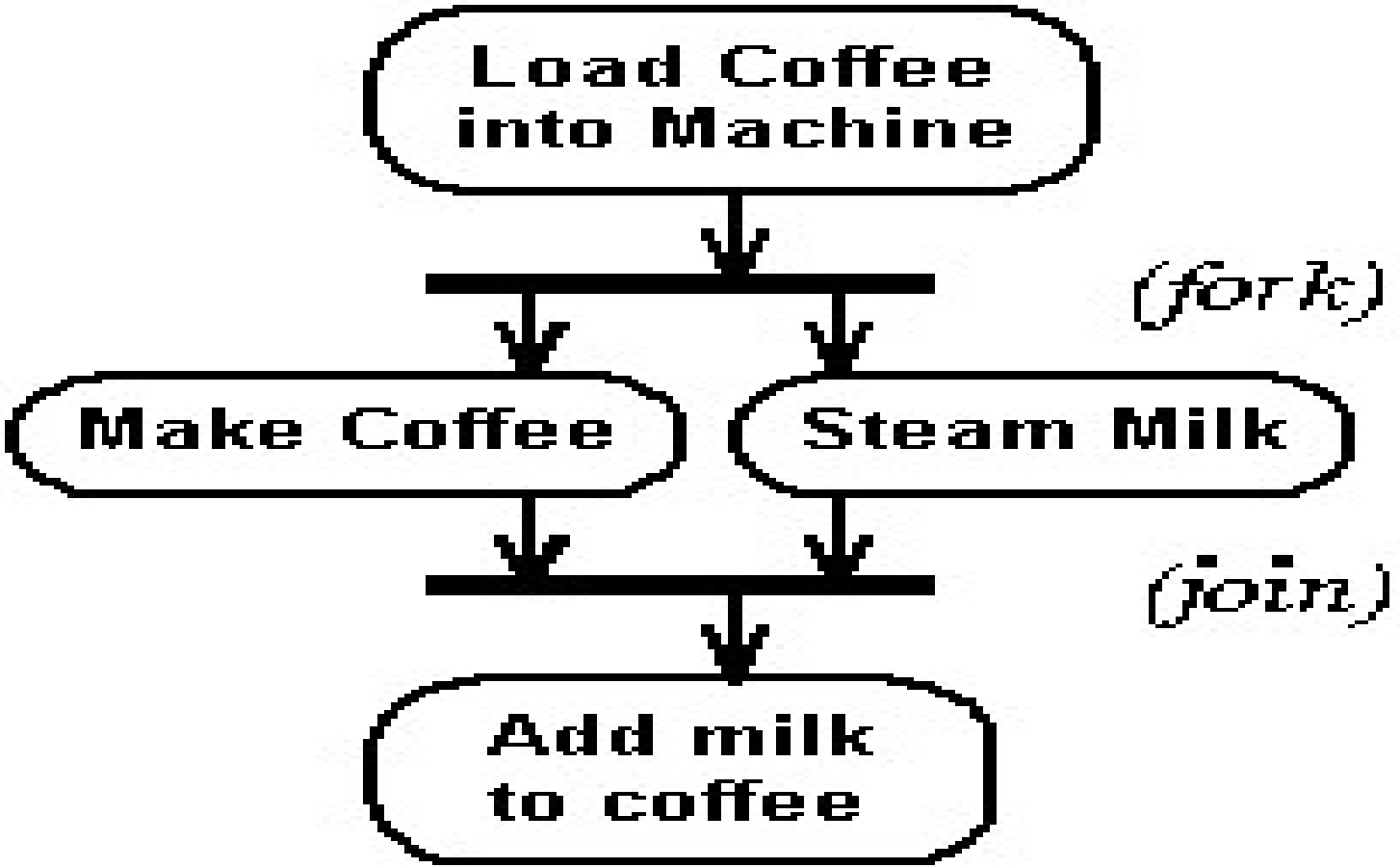
- In UML, guards are a statement written next to a decision diamond that must be true before moving next to the next activity. These are not essential, but are useful when a specific answer, such as "Yes, three labels are printed," is needed before moving forward.



Guard Symbols

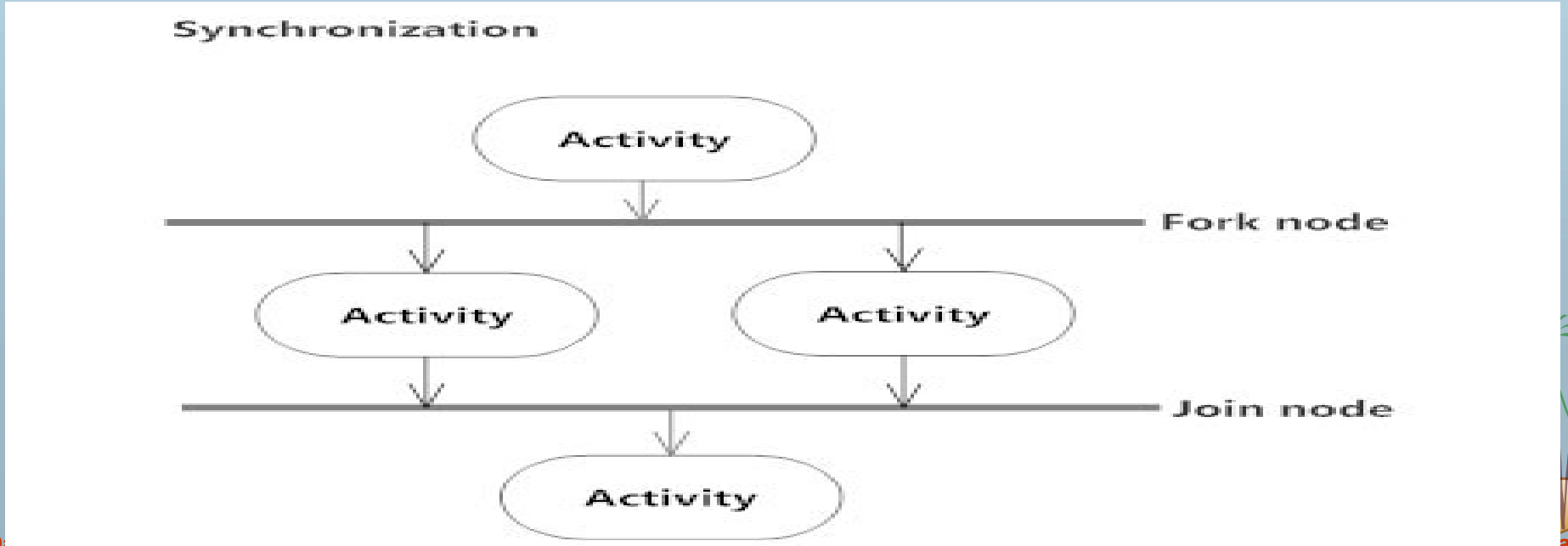
Synchronization (Fork/Join

- Used either when several activities can go on in parallel or when the order in which a set of activities execute is immaterial.
- The heavy bar at the top is a *fork*.
-



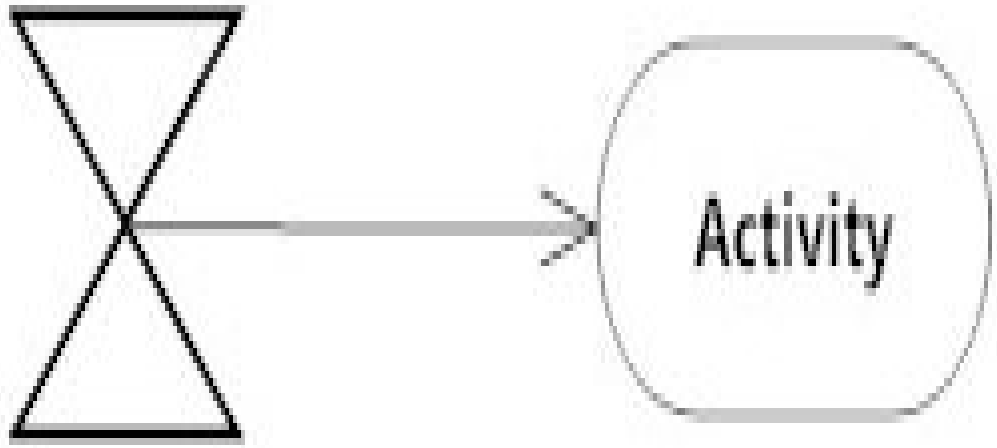
Synchronization

- A fork node is used to split a single incoming flow into multiple concurrent flows. It is represented as a straight, slightly thicker line in an activity diagram.
- A join node joins multiple concurrent flows back into a single outgoing flow.
- A fork and join mode used together are often referred to as synchronization.



Time Event

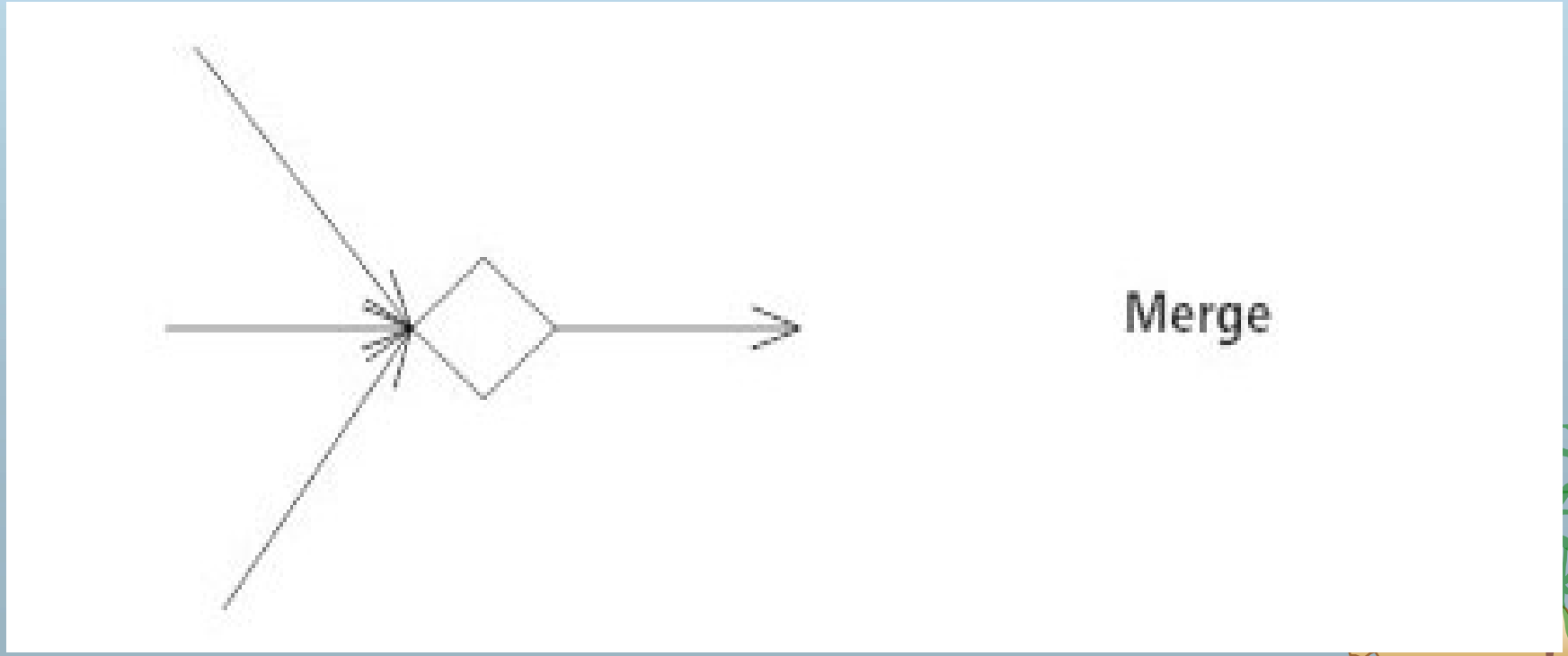
- **Time Event**
  - This refers to an event that stops the flow for a time; an hourglass depicts it.



Time Event

Merge Event

- **Merge Event**
- A merge event brings together multiple flows that are not concurrent.

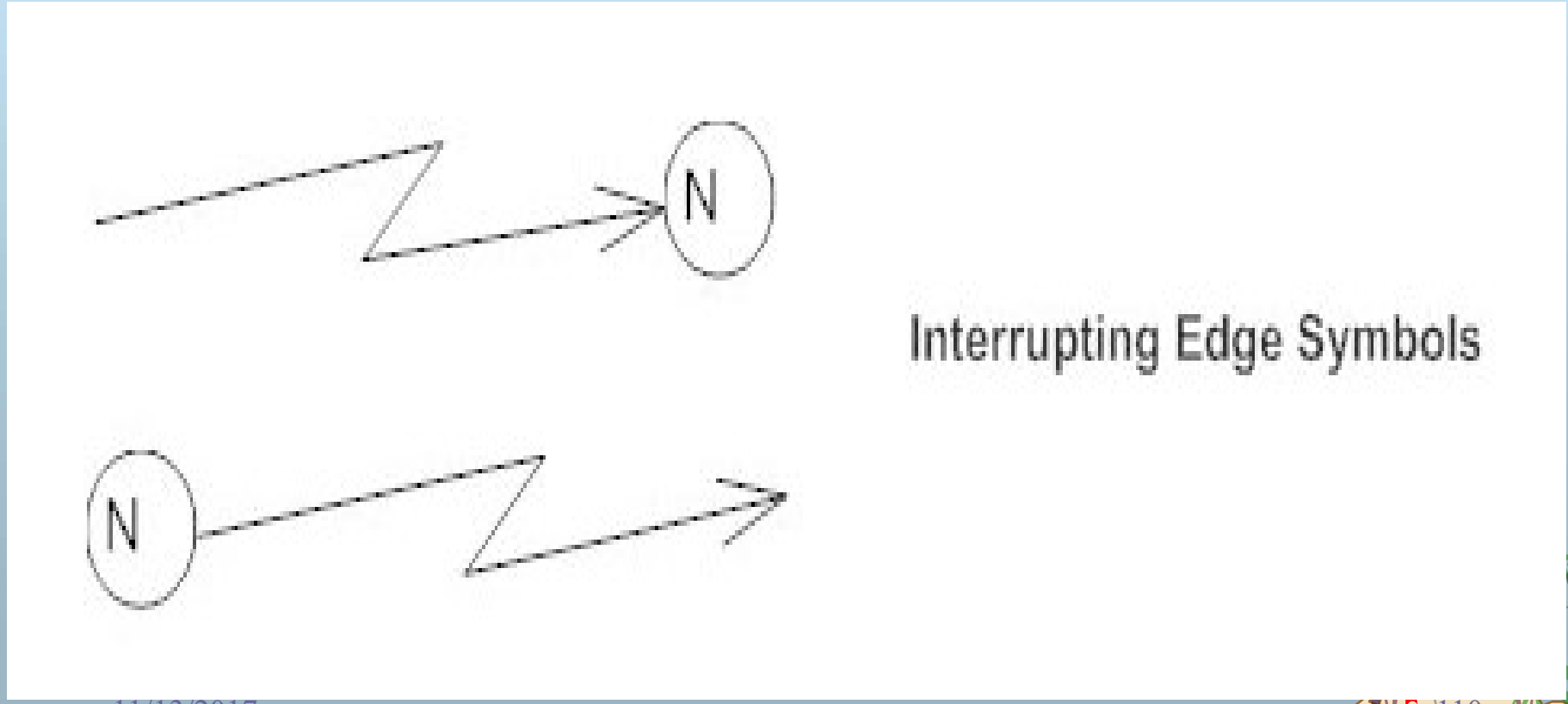


Sent and Received Signals

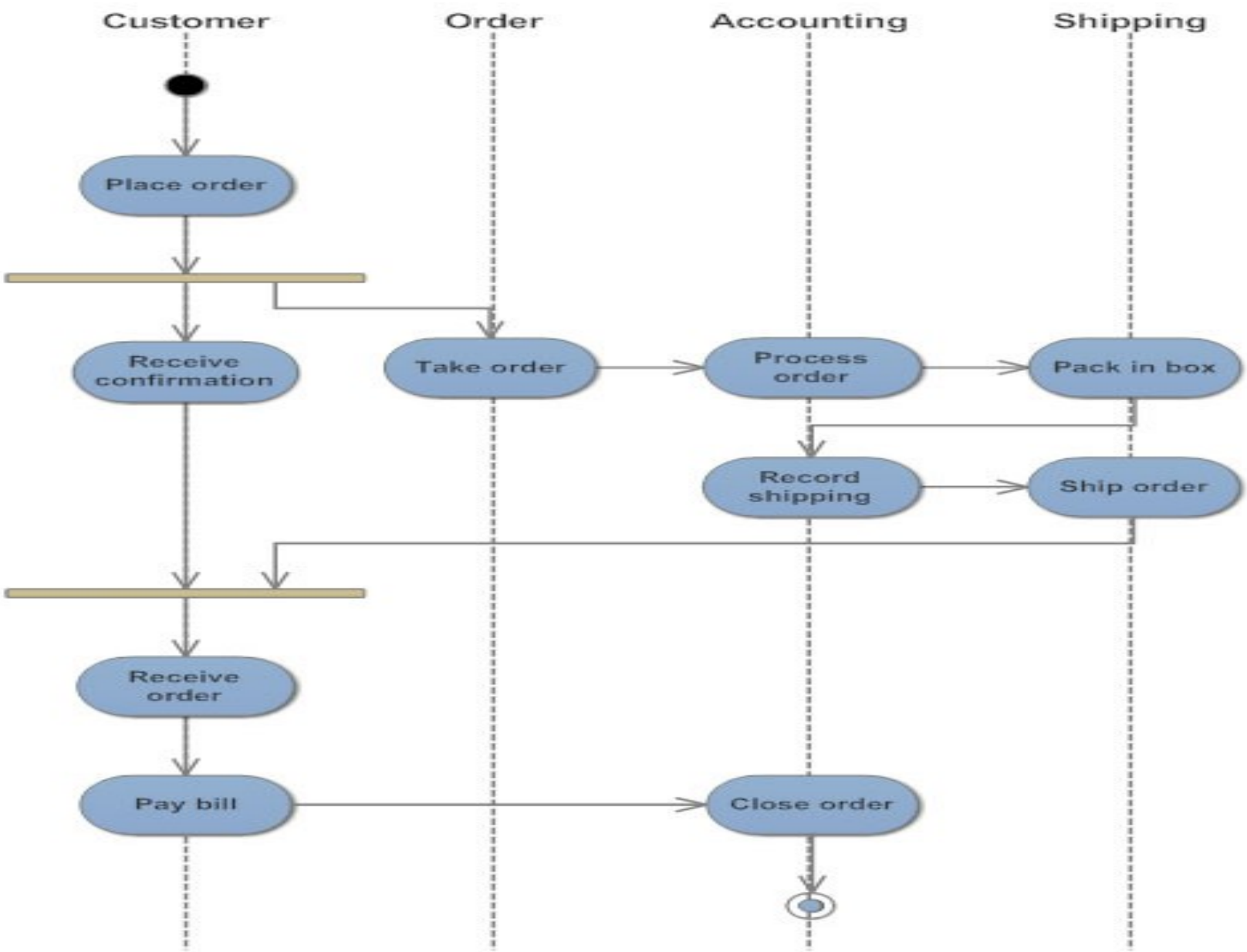
- Signals represent how activities can be modified from outside the system. They usually appear in pairs of sent and received signals, because the state can't change until a response is received, much like synchronous messages in a sequence diagram.
- For example, an authorization of payment is needed before an order can be completed.

Interrupting Edge

- An event, such as a cancellation, that interrupts the flow denoted with a lightning bolt.

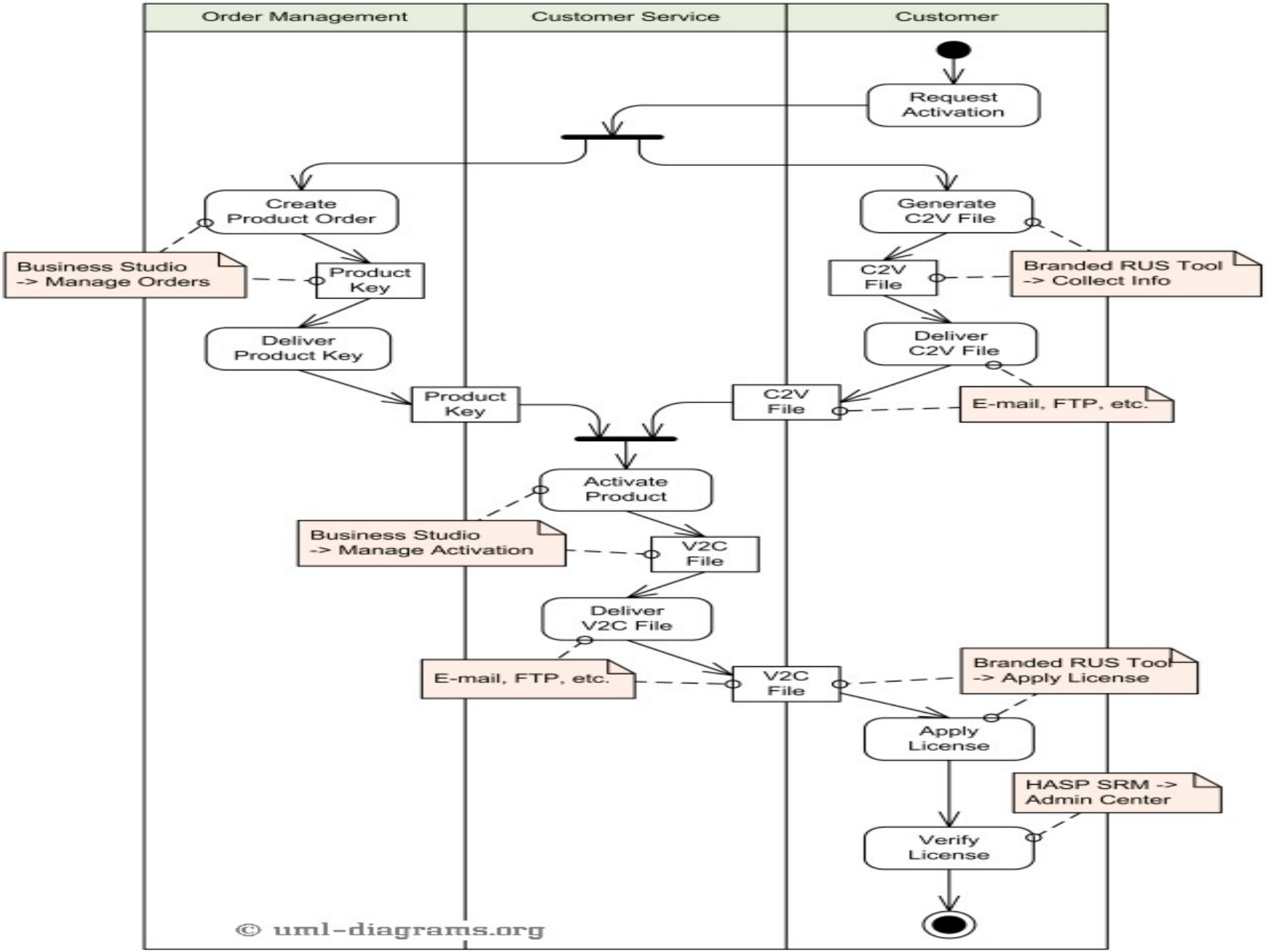


UML Activity Diagram: Order Processing



Sentinel HASP

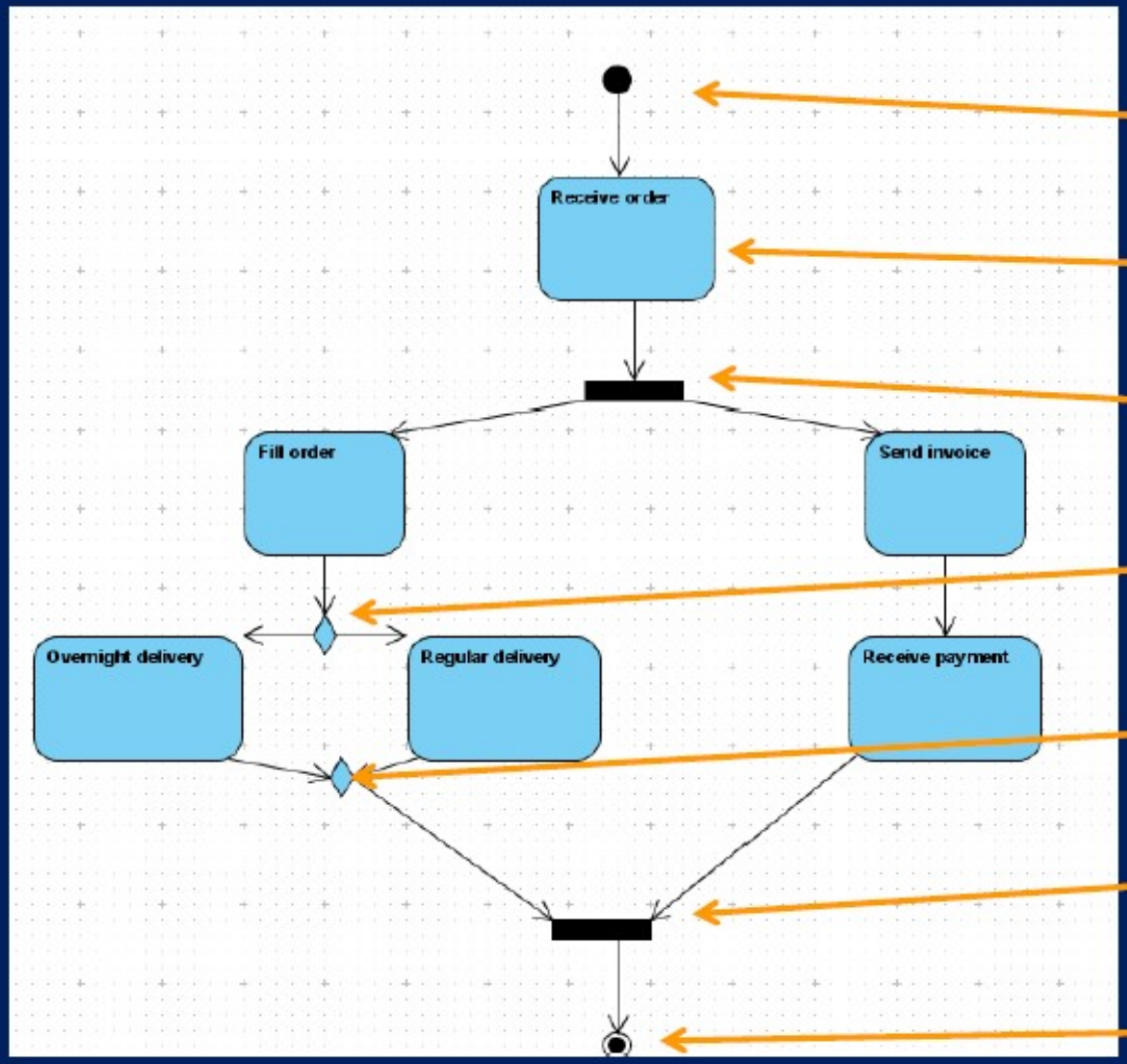
- **Sentinel HASP** (Formerly **Aladdin HASP** SRM) is a concurrent usage software licensing solution provided by SafeNet. It comes in two flavors: **HASP** SL employs software protection keys to enforce software protection and licensing.
- In example of activity diagram describing manual activation of trial (provisional) product which was protected by **Sentinel HASP SL** software key of the Sentinel HASP - software protection and licensing security solution.
- **Sentinel HASP** protects against losses from software piracy and intellectual property theft. For example, it offers industry-leading support for licensing in virtual environments, and is the first software licensing and reverse engineering protection tool solution on the market to support J2EE applications.
- Sentinel HASP software includes **Business Studio** - a powerful, role-based software licensing and management tool. The Business Studio is used by product, marketing and development staff to prepare software product for market and includes all of the tools necessary to license and lock application to a Sentinel HASP HL hardware or HASP SL software product key, to manage and track licenses, to create product keys that are later used for the product activation process, etc.


Order Management
Customer Service
Customer
Request Activation
Create Product Order
Generate C2V File
Business Studio -> Manage Orders
Product Key
C2V File
Branded RUS Tool -> Collect Info
Deliver Product Key
Deliver C2V File
Product Key
C2V File
E-mail, FTP, etc.
Activate Product
Business Studio -> Manage Activation
V2C File
Deliver V2C File
E-mail, FTP, etc.
V2C File
Branded RUS Tool -> Apply License
Apply License
HASP SRM -> Admin Center
Verify License
© uml-diagrams.org

Activity diagram

- Shows flow of messages, logic, actions
- This is at a much higher level of abstraction than flow charts
  - Flow charts show logic for single method (if statements, loops, etc.)
  - Activity diagrams show flow among objects

Activity diagram example



Initial Node

Activity Node

Fork (parallel)

Decision (either-or)

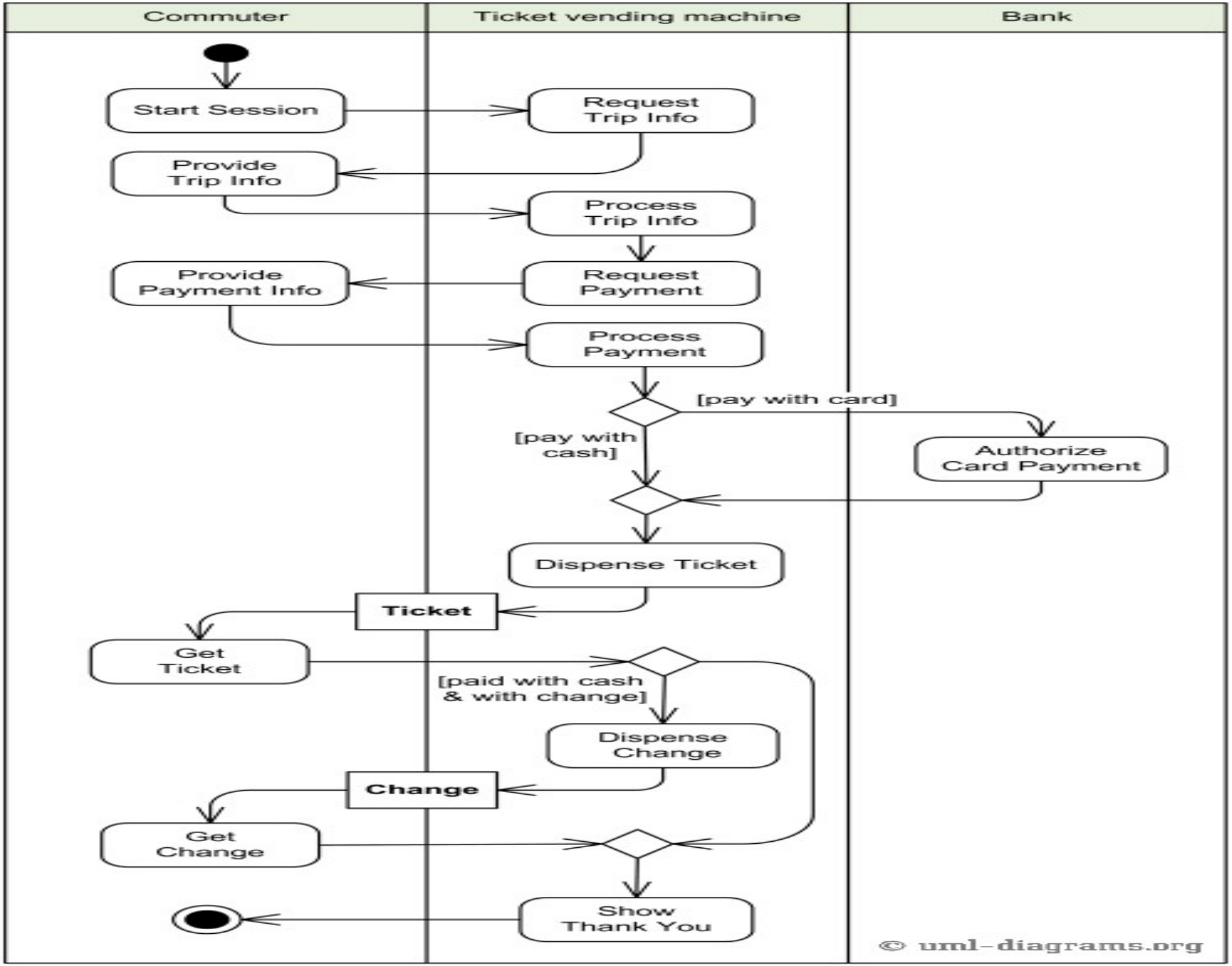
Merge (either-or)

Join (parallel)

Final Node

Ticket vending machine

- **Purpose**:
  - An example of UML activity diagram describing behavior of the **Purchase Ticket use case** for a Ticket vending machine.
  - Activity is started by Customer actor who needs to buy a ticket.
  - Ticket vending machine will request trip information from Customer.
  - Based on the info machine will calculate payment due and request payment options. After payment is complete, ticket is dispensed to the Commuter.

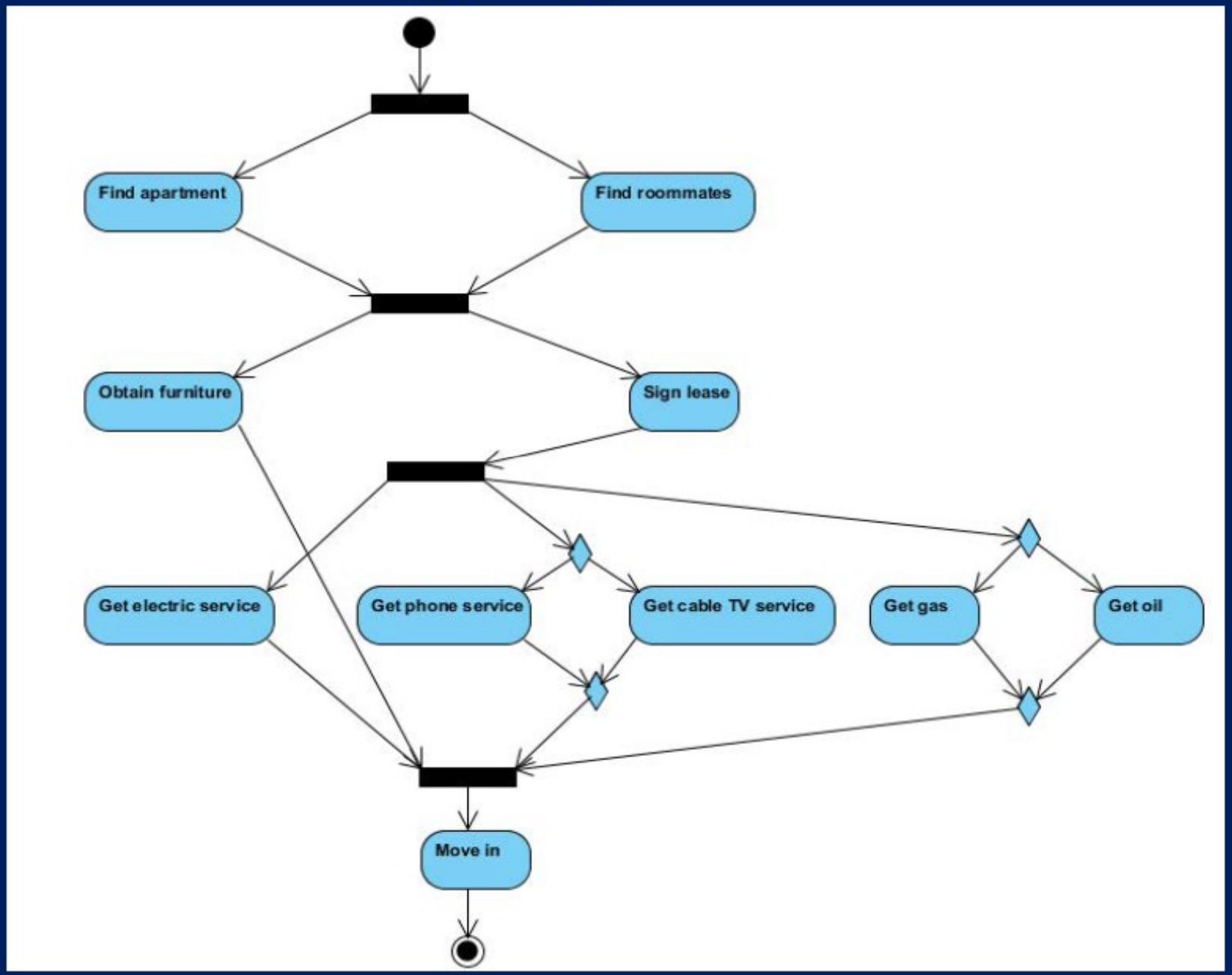

Commuter
Ticket vending machine
Bank
Start Session
Request Trip Info
Provide Trip Info
Process Trip Info
Provide Payment Info
Request Payment
Process Payment
[pay with card]
[pay with cash]
Authorize Card Payment
Dispense Ticket
Ticket
Get Ticket
[paid with cash & with change]
Dispense Change
Change
Get Change
Show Thank You
© uml-diagrams.org

- An activity is a state with an internal action and one or more outgoing transitions which automatically follow the termination of the internal activity
- If an activity has more than one outgoing transitions, then these must be identified through conditions .
- An interesting feature of the activity diagrams is the swim lanes.
- Swim lanes enable you to group activities based on who is performing them.
  - e.g. academic department vs. hostel office. Thus swim lanes subdivide activities based on the responsibilities of some components.
  - The activities in a swim lane can be assigned to some model elements, e.g. classes or some component, etc

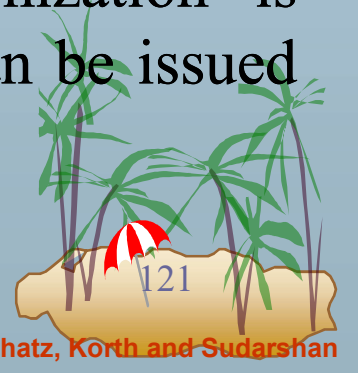
Activity diagram exercise

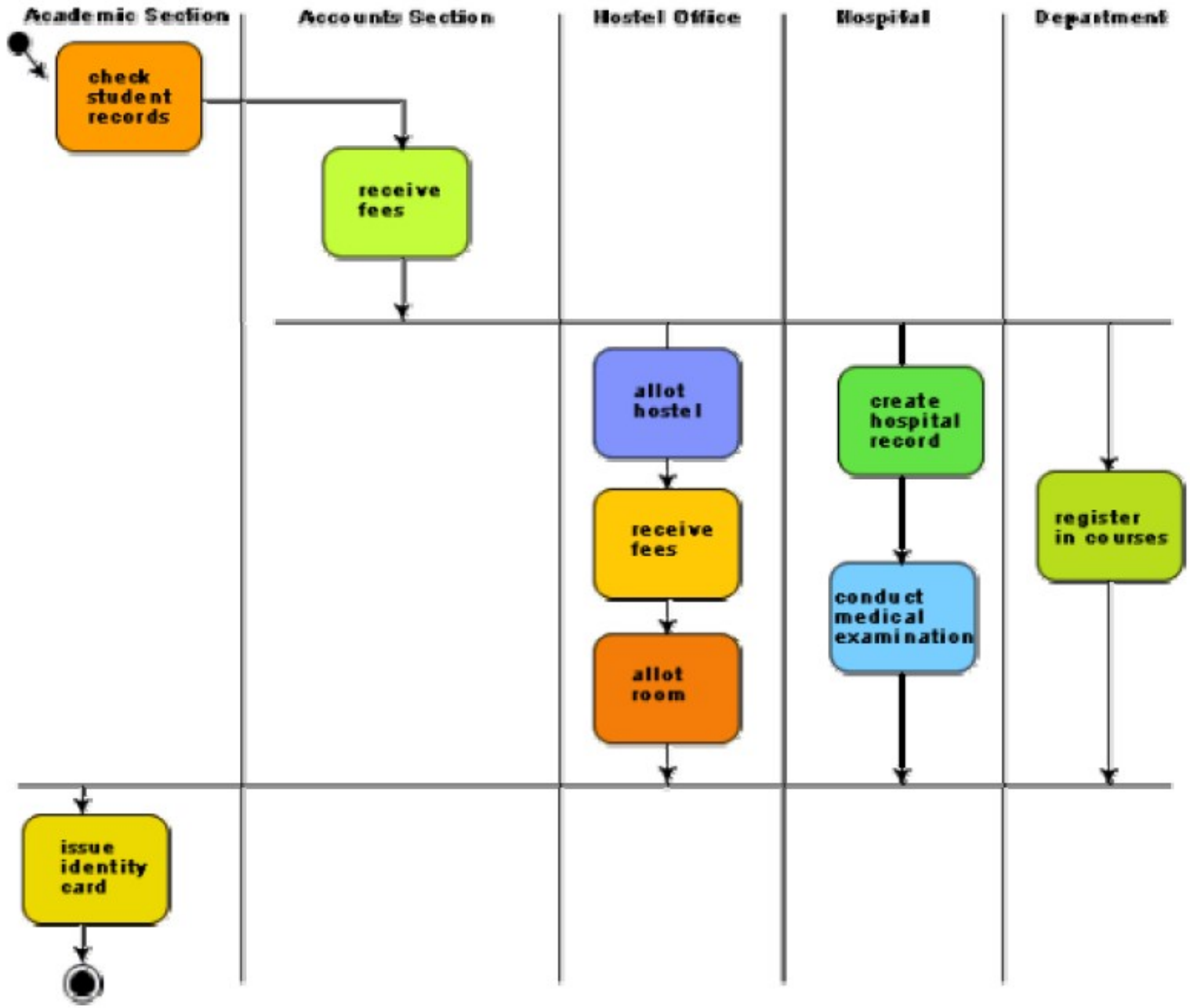
- Draw an activity diagram for getting an apartment. Example activities are:
  - Find roommates
  - Find apartment
  - Sign apartment lease
  - Get electric service
  - Get phone or cable TV service
  - Get gas or oil heat account set up
  - Obtain furniture
  - Move in
- (Use this as a simple model of setting up a warehouse…)
- Use activities, decision nodes, fork/join nodes

Activity diagram solution



- Activity diagrams are normally employed in business process modeling. This is carried out during the initial stages of requirements analysis and specification.
- Activity diagrams can be very useful to understand complex processing activities involving many components. Later these diagrams can be used to develop interaction diagrams which help to allocate activities (responsibilities) to classes.
  - The student admission process in IIT is shown as an activity diagram in this it shows the part played by different components of the Institute in the admission procedure.
  - After the fees are received at the account section, parallel activities start at the hostel office, hospital, and the Department. After all these activities complete (this synchronization is represented as a horizontal line), the identity card can be issued to a student by the Academic section



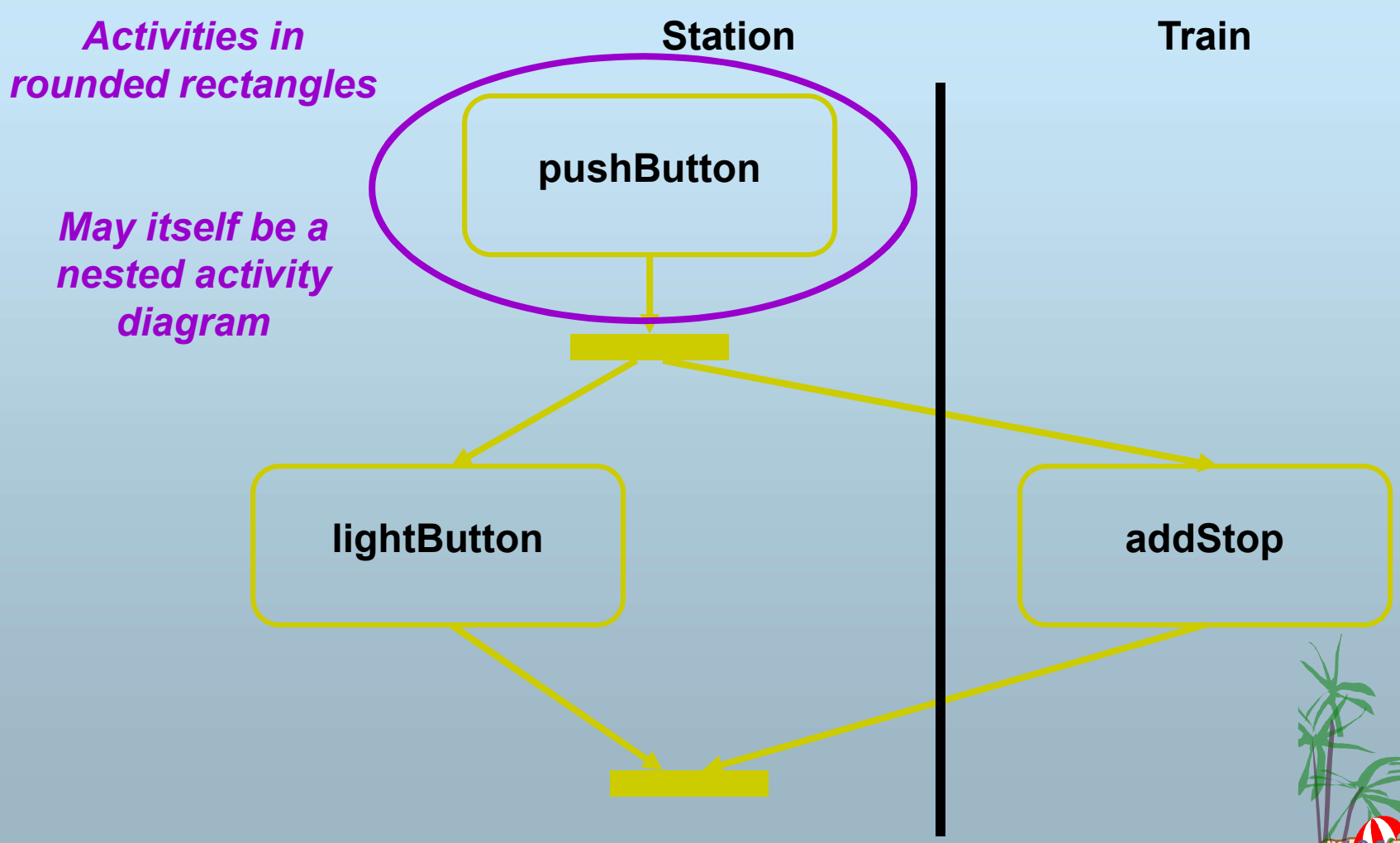

Academic Section
Accounts Section
Hostel Office
Hospital
Department
check student records
receive fees
allot hostel
receive fees
allot room
create hospital record
conduct medical examination
register in courses
issue identity card

- Activity diagrams are similar to the procedural flow charts. The difference is that activity diagrams support description of parallel activities and synchronization aspects involved in different activities.

Activity Diagrams

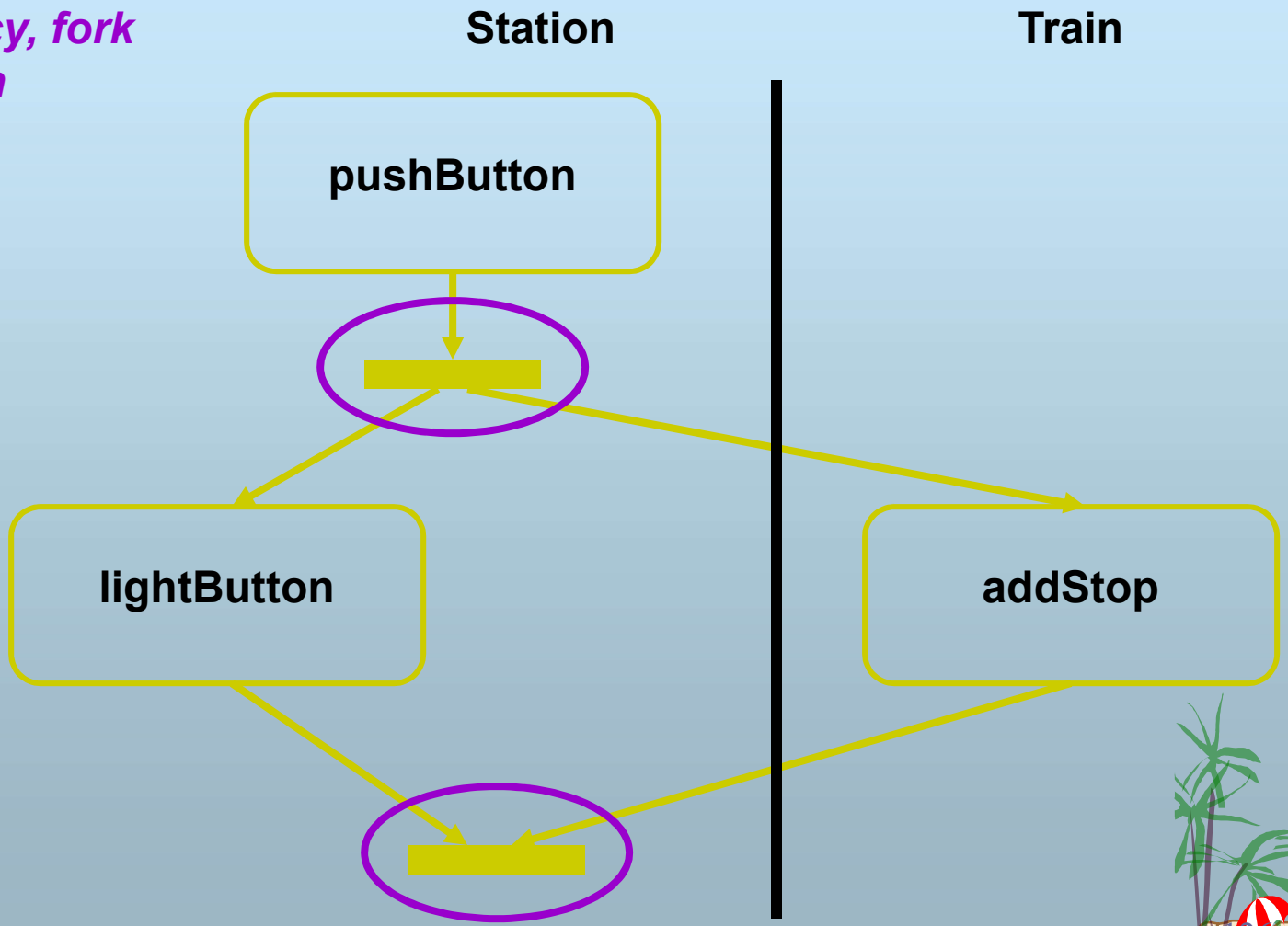
- Reincarnation of flow charts
  - Uses flowchart symbols
- Emphasis on control-flow
- Two useful flowchart extensions
  - Hierarchy
    - A node may be an activity diagram
  - Swim lanes

Example Activity Diagram



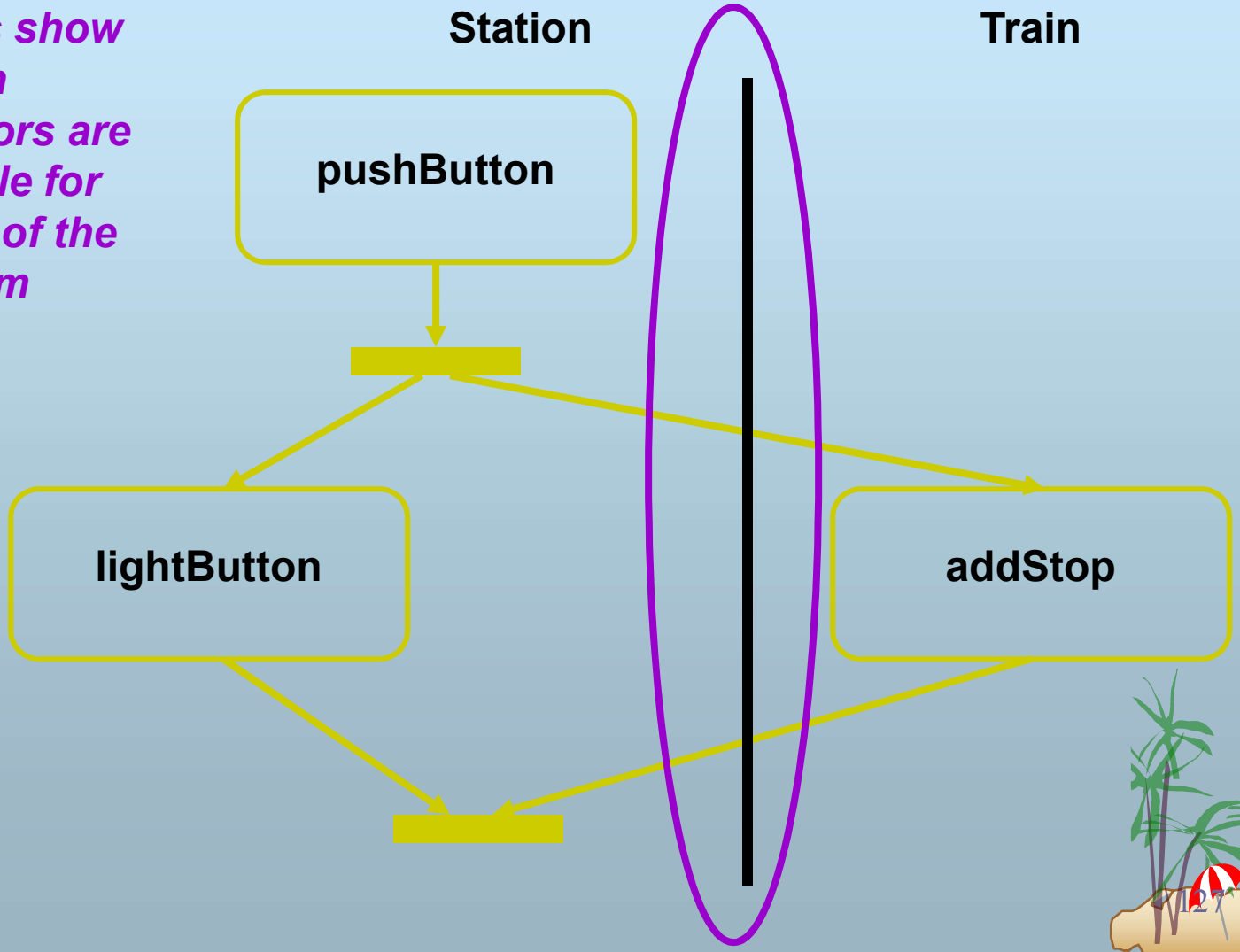
Example Activity Diagram

Concurrency, fork & join



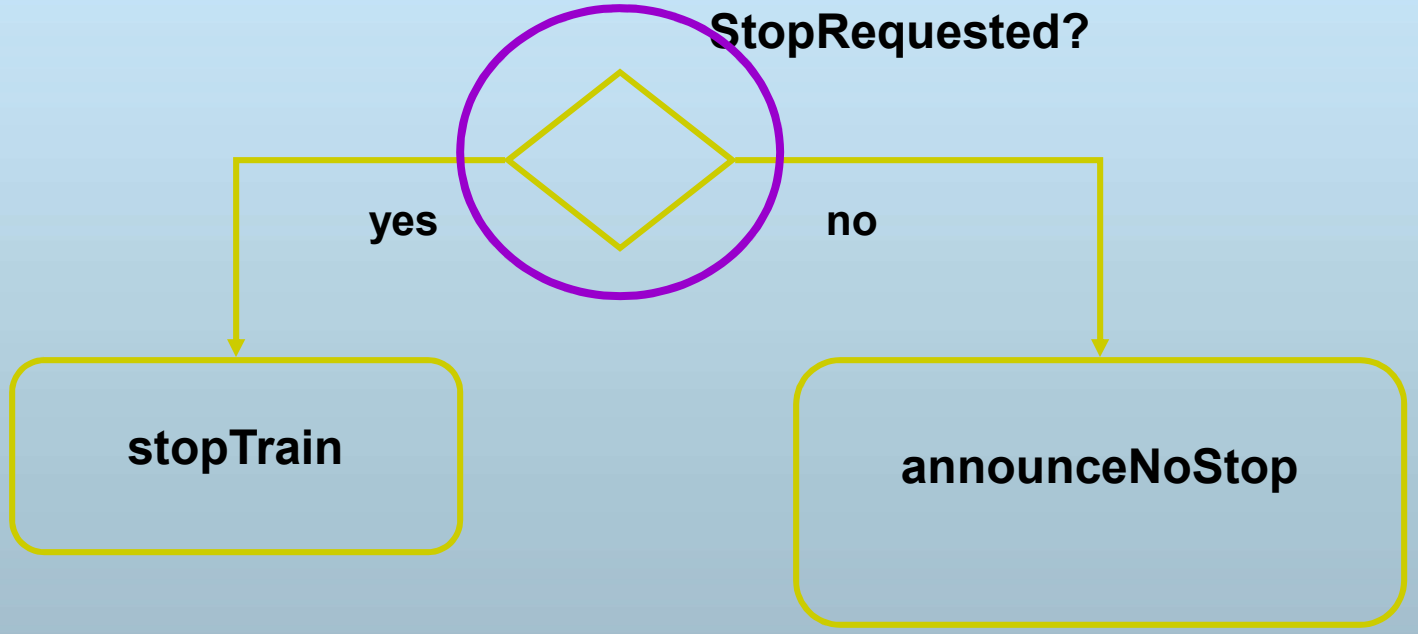
Example Activity Diagram

Swim lanes show which classes/actors are responsible for which part of the diagram



Another Example Activity Diagram

Classic flow-chart if-then-else



State chart diagram

- State chart diagram is normally used to model how the state of an object changes in its lifetime.
- State chart diagrams are good at describing
  - how the behavior of an object changes across several use case executions.
  - However, if we are interested in modeling some behavior that involves several objects collaborating with each other, state chart diagram is not appropriate.
  - State chart diagrams are based on the finite state machine (FSM) formalism.

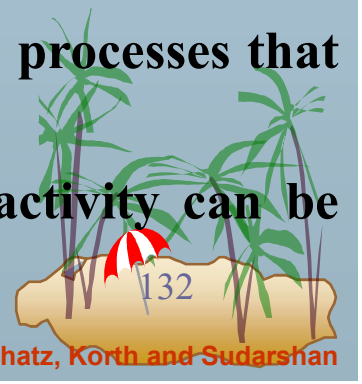
State Diagram

- A **state diagram** is a type of **diagram** used in computer science and related fields to describe the behavior of systems.
- **State diagrams** require that the system described is composed of a finite number of **states**; sometimes, this is indeed the case, while at other times this is a reasonable abstraction.

FSM

- State diagram consists of a finite number of states corresponding to those of the object being modeled.
  - The object undergoes state changes when specific events occur.
  - The FSM formalism existed long before the object-oriented technology and has been used for a wide variety of applications.
- Apart from modeling, it has even been used in theoretical computer science as a generator for regular languages.

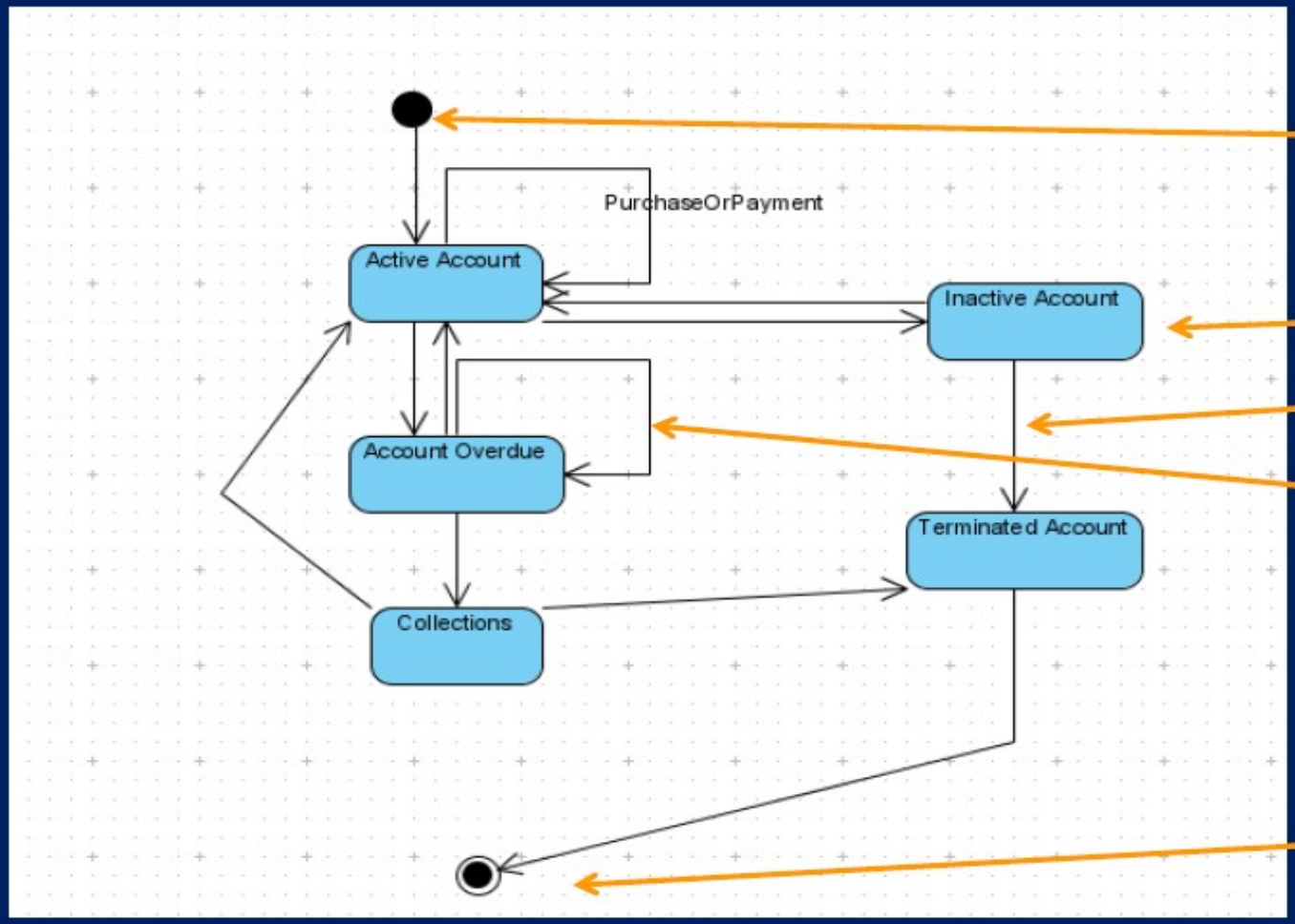
- A state diagram shows the behavior of classes in response to external stimuli. Specifically a state diagram describes the behavior of a single object in response to a series of events in a system.
  - Sometimes it's also known as a Harel state chart or a state machine diagram. This UML diagram models the dynamic flow of control from state to state of a particular object within a system.
- A major disadvantage of the FSM formalism is the state explosion problem.
  - The number of states becomes too many and the model too complex when used to model practical systems. This problem is overcome in UML by using state charts. The state chart formalism was proposed by David Harel [1990]. A state chart is a hierarchical model of a system and introduces the concept of a composite state (also called nested state).
- Actions are associated with transitions and are considered to be processes that occur quickly and are not interruptible.
- Activities are associated with states and can take longer. An activity can be interrupted by an event.



What is the Difference between a State Diagram and a Flowchart?

- A flowchart illustrates processes that are executed in the system that change the state of objects. A state diagram shows the actual changes in state, not the processes or commands that created those changes

State diagram example



An object (account in this example) can be in only one state at any time

Dynamic modelling

- Events
- states
- state diagrams
- conditions
- operations

Events

- Something that happens at a point in time
  - Mouse button clicked / Signal changes
- Logically ordered events - causally related
- Concurrent events - causally unrelated
  - do not effect each other
  - there is no order between them
- 1-way transmission of information from one object to another

Event Classes and Attributes

- Aeroplane flight departs (airline, flight no, city)
- Mouse button pushed (button, location)
- Input string entered (text)
- phone receiver lifted
- digit dialled (digit)
- engine speed enters danger zone

States

- A state is an abstraction of the attribute values and links of an object.
- Sets of values are grouped together into a state according to properties that affect the gross behaviour of the object.
- E.G.. A bank is solvent or insolvent depending on whether it's assets exceed it's liabilities.
- A state corresponds to the interval between 2 events received by an object.
  - A state separates 2 events.
  - An event separates 2 states.

Characterisations of a state

- State: Alarm ringing
- Description: alarm on watch is ringing to indicate target time
- Event sequence that produces the state:
- set alarm (target time)
- any sequence not including clear alarm
- current time = target time

Condition that characterises the state:

alarm = on,

and

target time <= current time <= target time + 20s

and

button has not been pushed since the target time

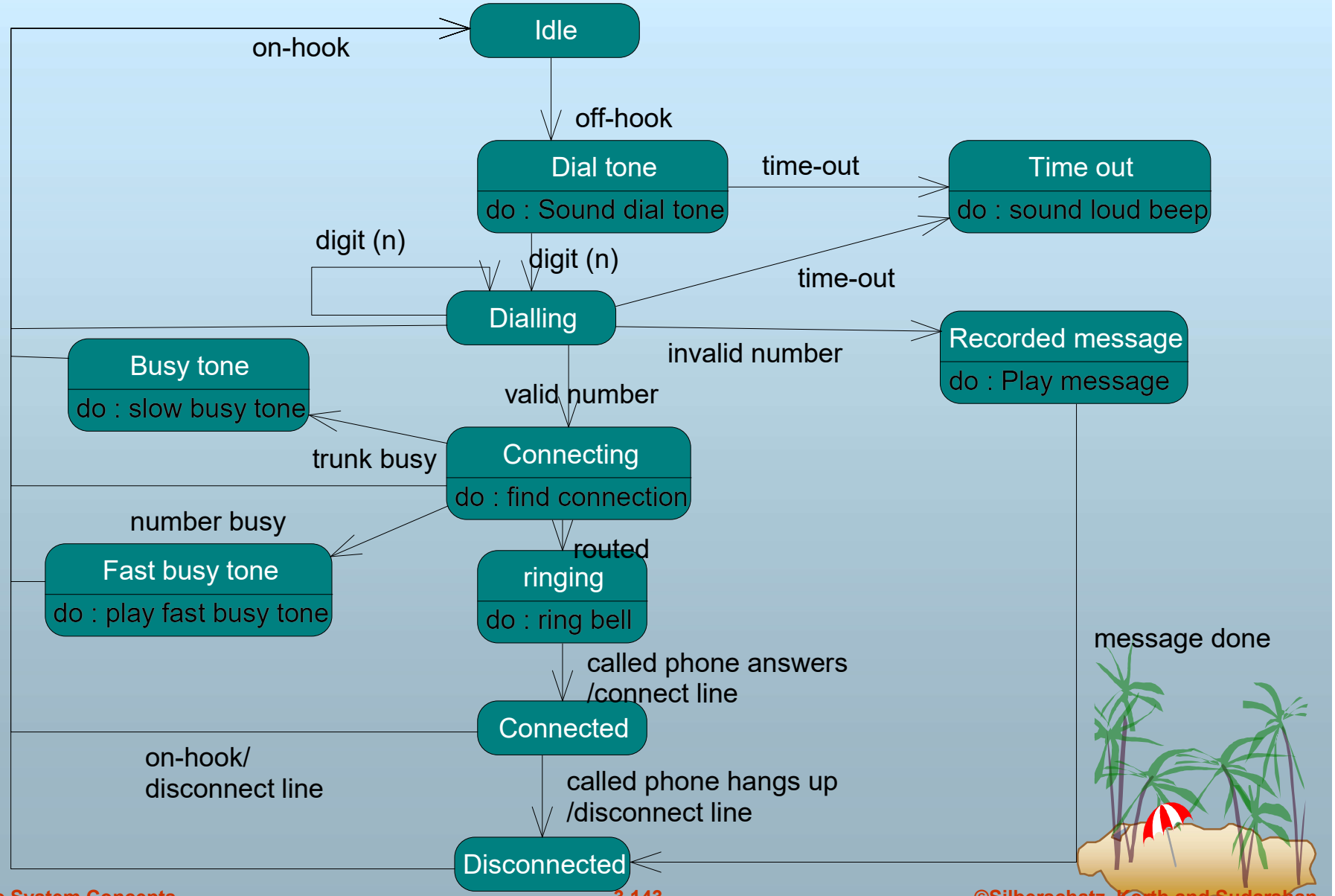
Events accepted in the state:

event	action	next state
current time = target time + 20s	reset alarm	normal
button pushed (any button)	reset alarm	normal

State Diagrams

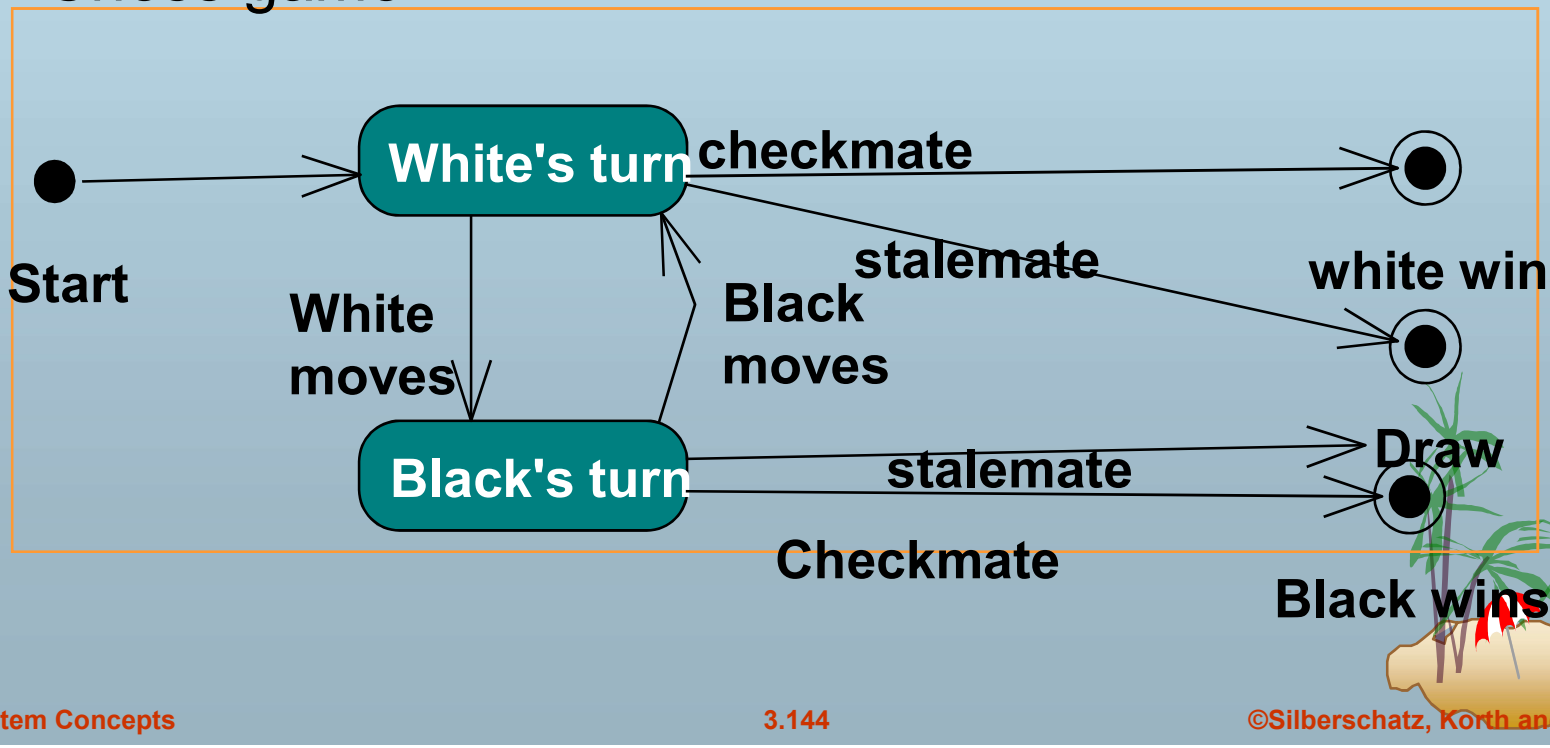
- Relates to a specific object
- Relate states and events
- A change of state is called a transition
- All transitions leaving a state must correspond to different events
- The transition fires
- An event that has no transition is ignored
- A sequence of events corresponds to a path through the graph

State Transition Diagram



Initial/final States

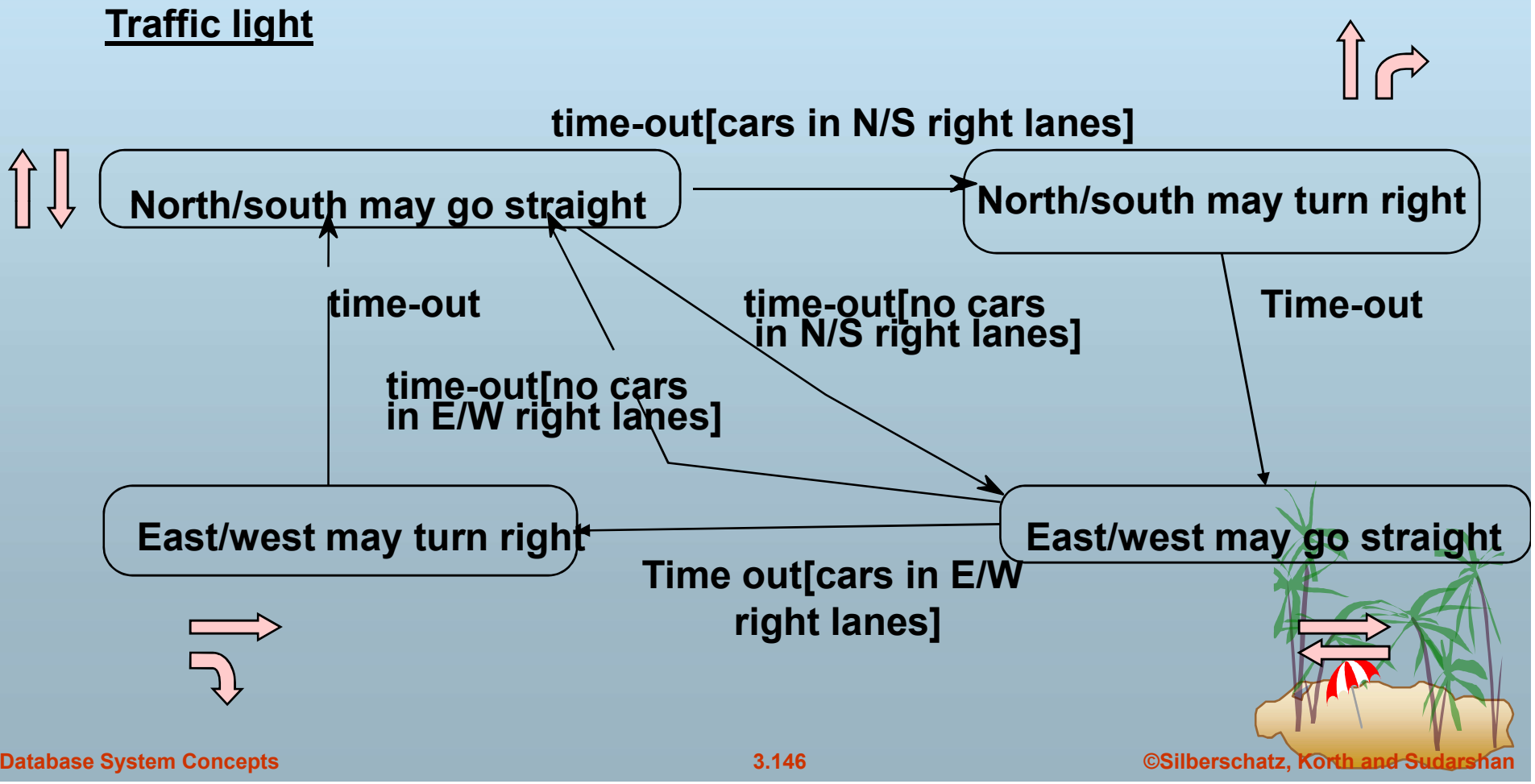
- The previous example is a continuous loop
- Most situations will have an initial and final state
- Chess game



Conditions

- Used as guards on transitions.
  - A guarded transition only fires when the condition is true e.g.
    - When you go out in the morning (event),
    - If the temperature is below freezing (condition).
    - Put on your gloves (next state).

State Transition Diagram With Conditions

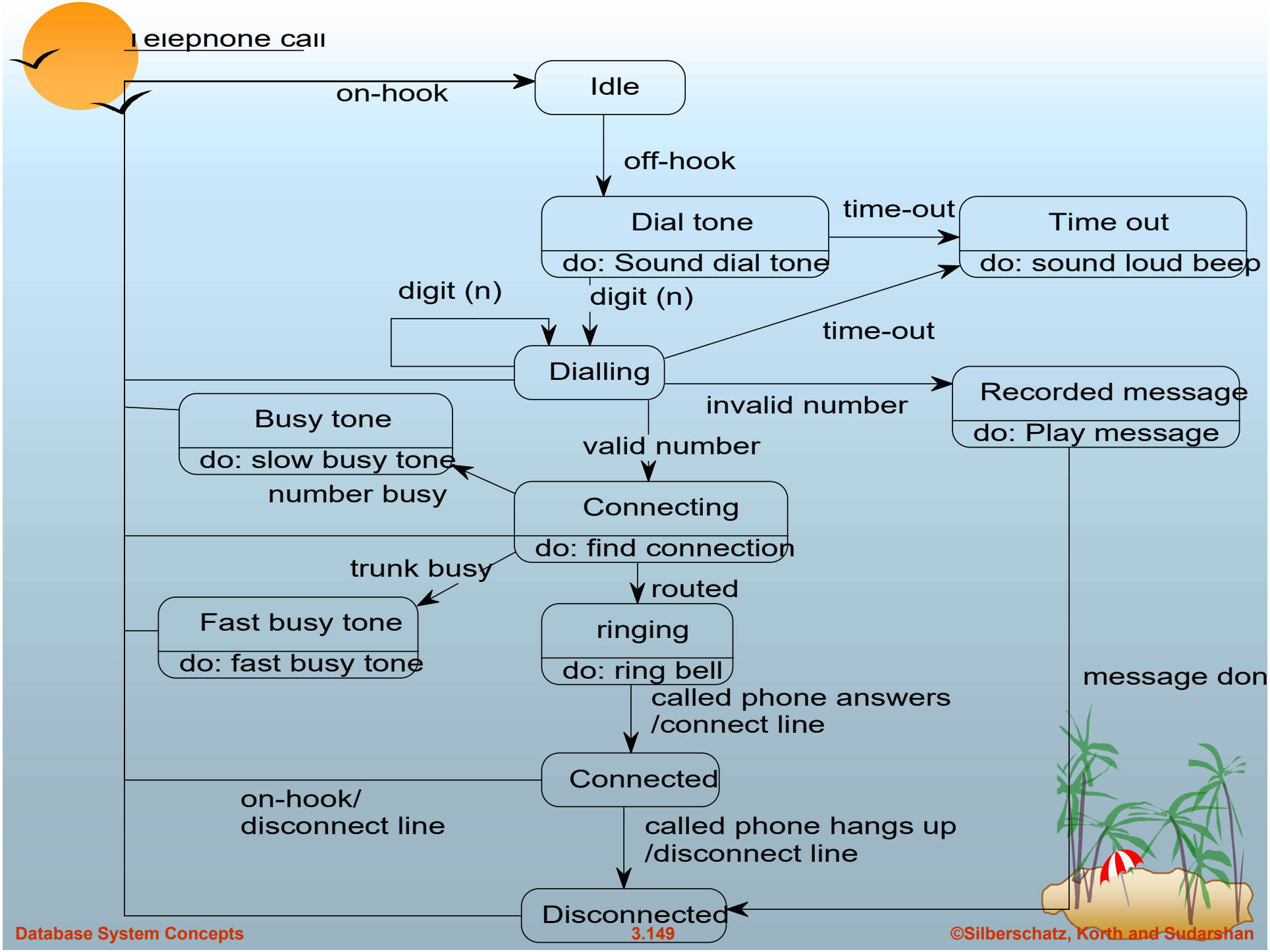


Operations

- Attached to state
- Performed in response to the state
- Attached to a transition
- Performed in response to the event

Activities and Actions

- An activity is an operation that takes time.
  - E.G.. Display a picture on a screen.
  - “Do:a ” indicates that activity A occurs throughout the lifetime of the state to which it is attached.
- An action is an instantaneous operation.
  - E.G.. Disconnect phone line.
  - An action is shown on a transition as “action / event.”



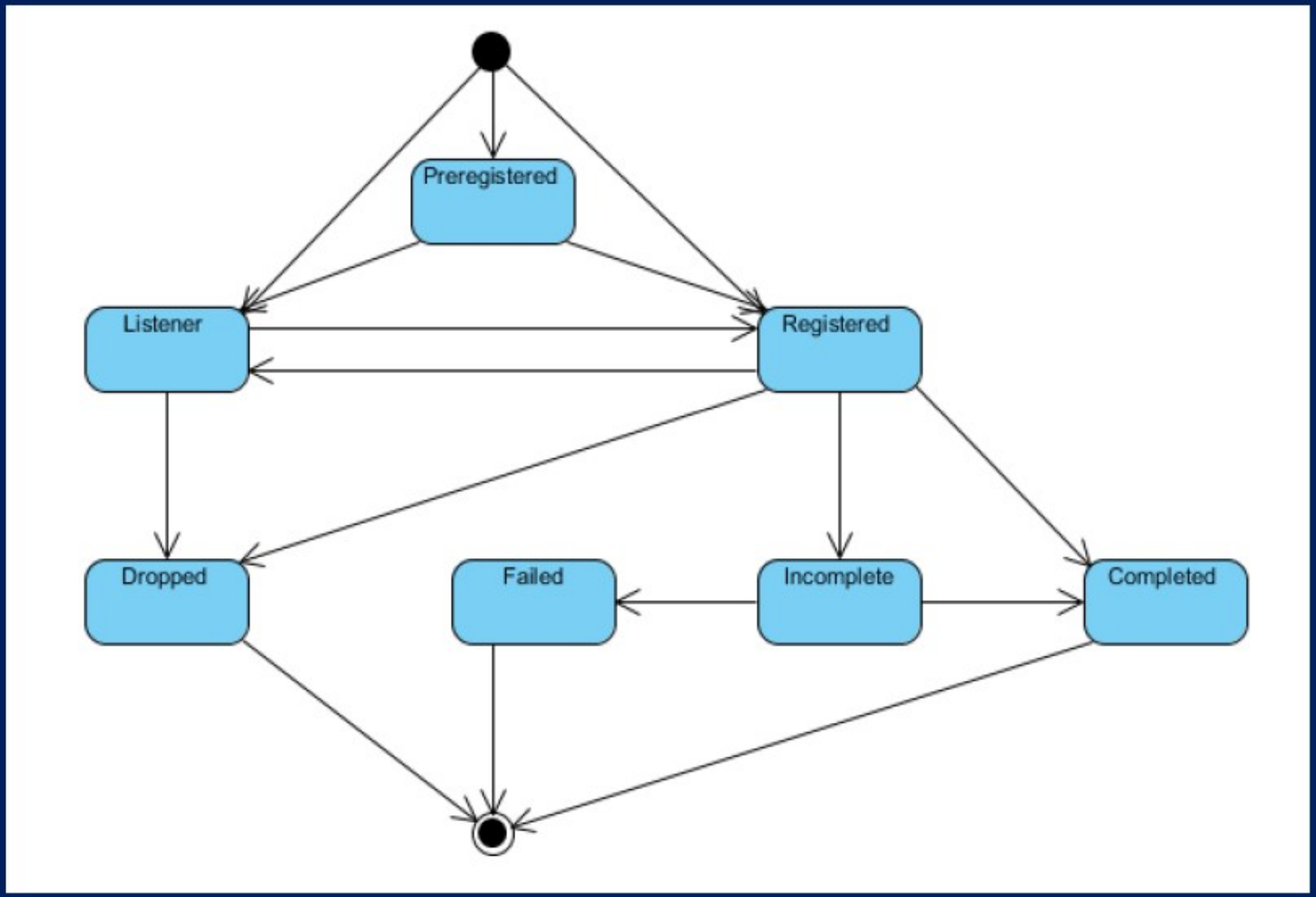
Nesting Diagrams

- Done by generalising states
  - E.G. Aircraft in flight
- Done by generalising events
  - E.G. Booking clerk books flight

State diagram exercise

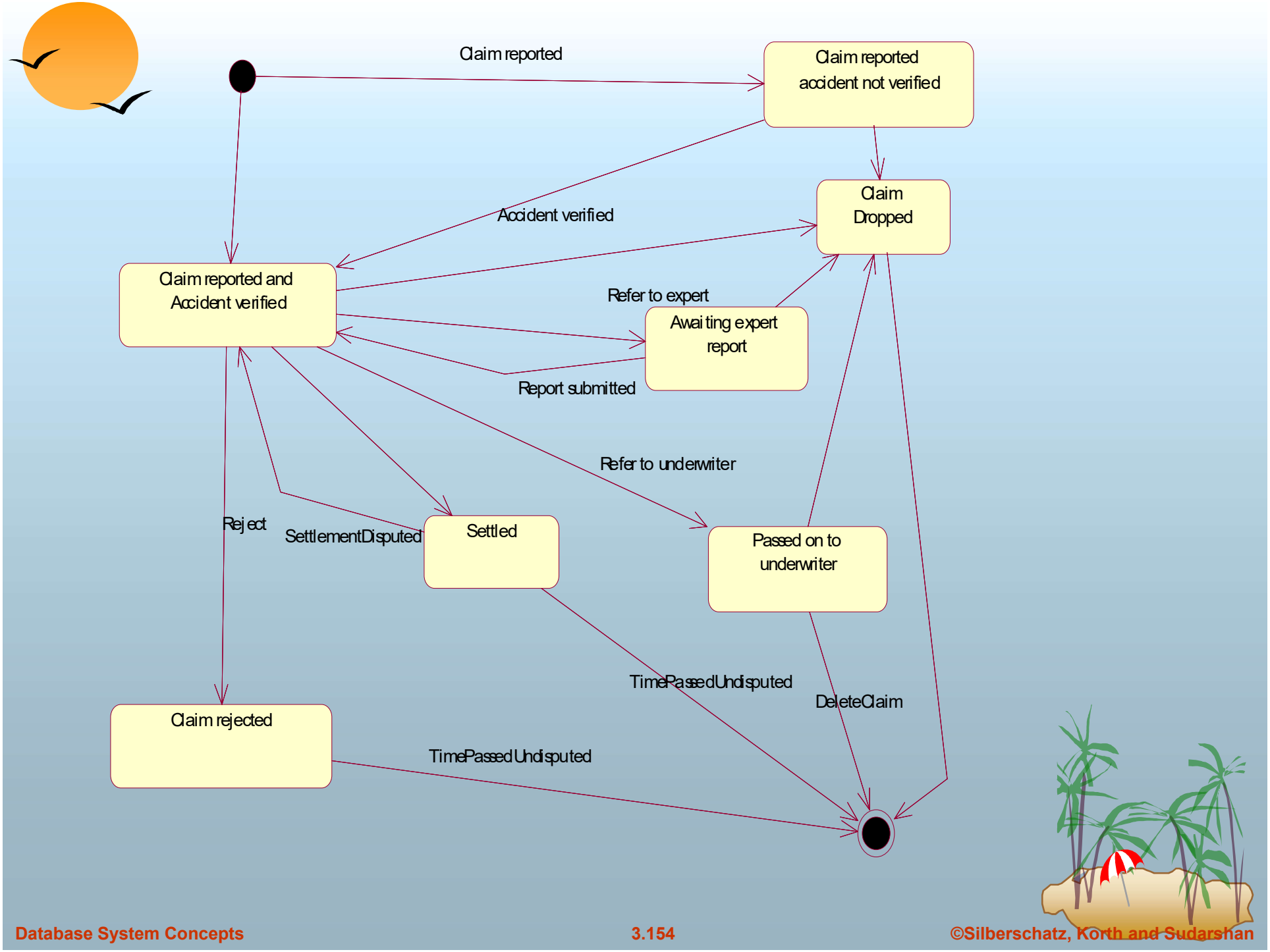
- Model the state transitions of a student's registration in this class:
  - Preregistered
  - Registered
  - Listener
  - Dropped
  - Complete, incomplete (not resolved), etc.
- Remember that an entity can only be in one state at any time. It cannot be in two or more states.

State diagram solution

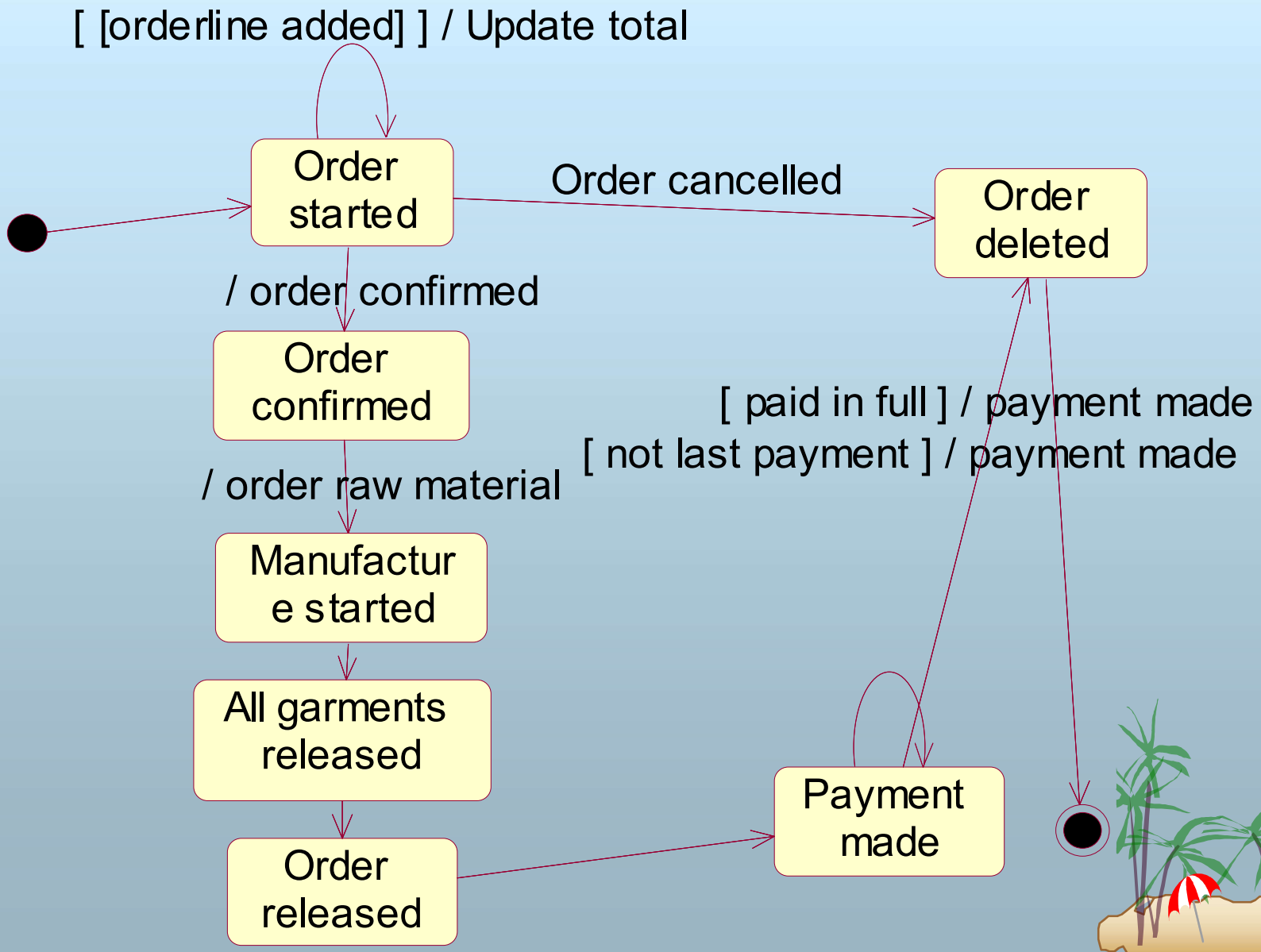


Exercise 1 - Claims

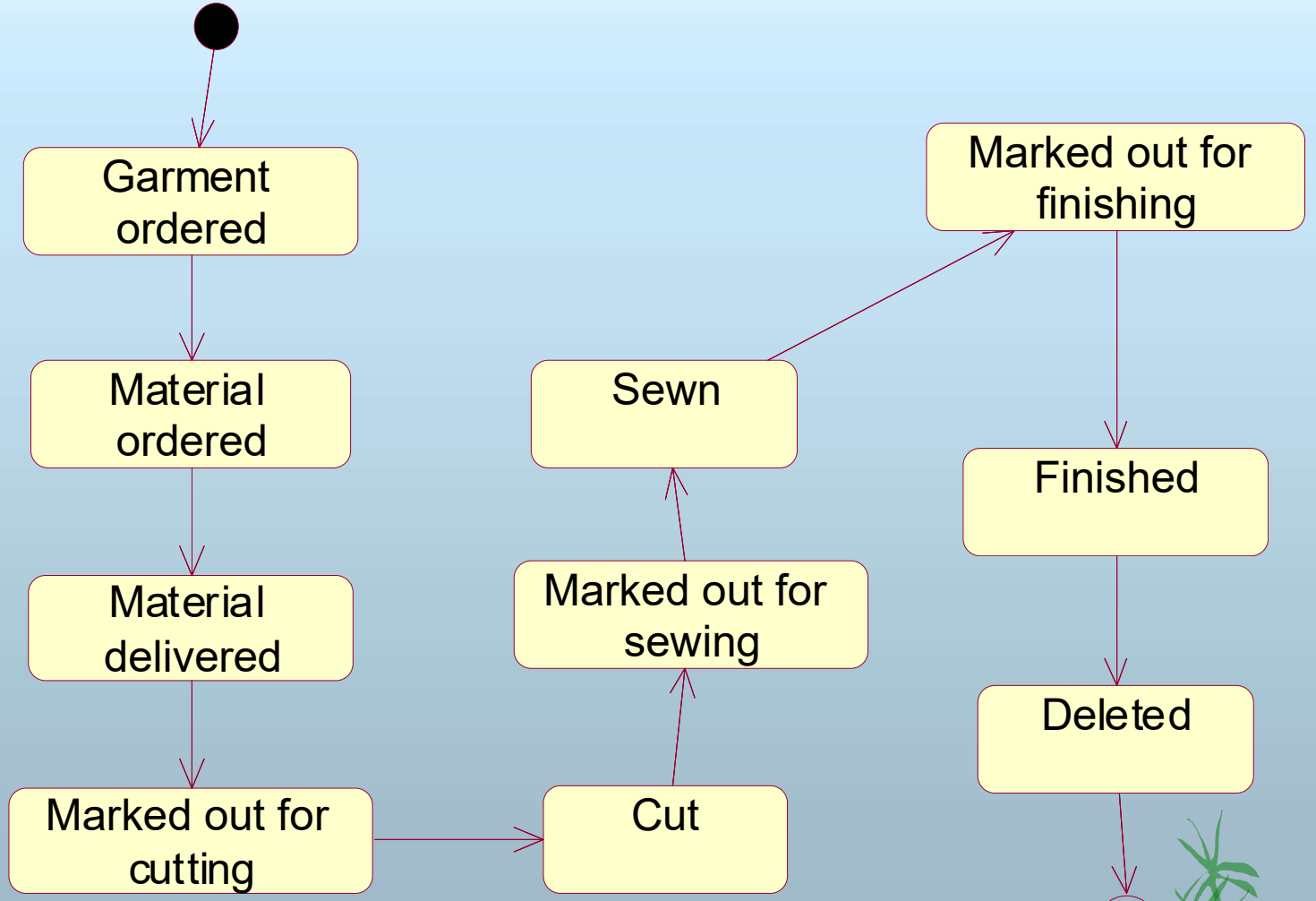
- A third party claim arises from an incident that has occurred. The claim must be reported to the claims office within the statutory period to be valid.
- When the claim is reported, the incident is preferably verified by a member of staff who was present at the time.
- The claim is recorded and evaluated. If it is a trivial claim, the claims assessor checks the claimant for previous claims and, if there are no or 1 previous claims then a payment is made and the claim is settled. If there are previous claims, the claimant will be referred to the courts. More serious claims may await expert evidence and several different court hearings.
- At any stage, the claim may be dropped, or a payment made without admitting liability. If the claimant is a minor, the claim must remain on the books until the claimant is over 21. If a claim is settled by a court, it may be appealed. During this time, the claim remains open.
- Draw a state diagram for a claim.


Claim reported
Claim reported accident not verified
Accident verified
Claim Dropped
Claim reported and Accident verified
Refer to expert
Awaiting expert report
Report submitted
Refer to underwriter
Reject
SettlementDisputed
Settled
Passed on to underwriter
TimePassedUndisputed
DeleteClaim
Claim rejected
TimePassedUndisputed

States of Order



States of Garment (Garments system)



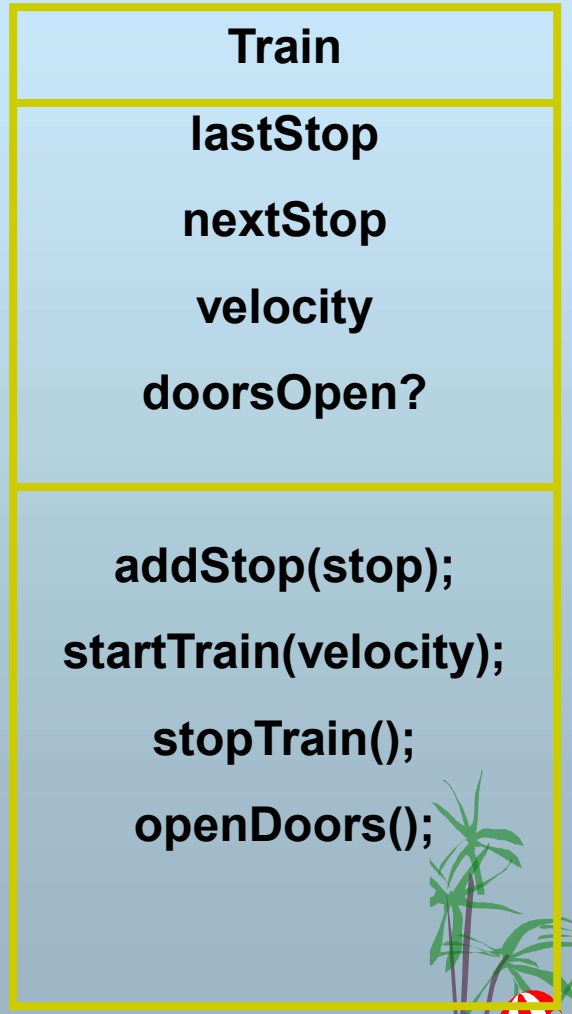
Summary of Use Cases

- Use Case Diagram
  - Shows all actors, use cases, relationships
  - Actors are agents external to the system
    - E.g., users

- Supplemental information
  - Entry/Exit Conditions, Story, Main and Alternative flows, Nonfunctional requirements
  - Specified in a separate document
    - In English

Class Diagrams

- Describe classes
  - In the OO sense
- Each box is a class
  - List fields
  - List methods
- The more detail, the more like a design it becomes

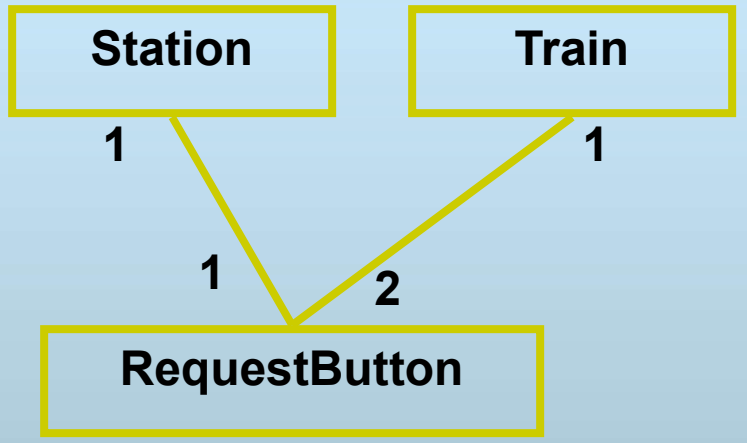


Class Diagrams: Relationships

- Many different kinds of edges to show different relationships between classes
- Mention just a couple

Associations

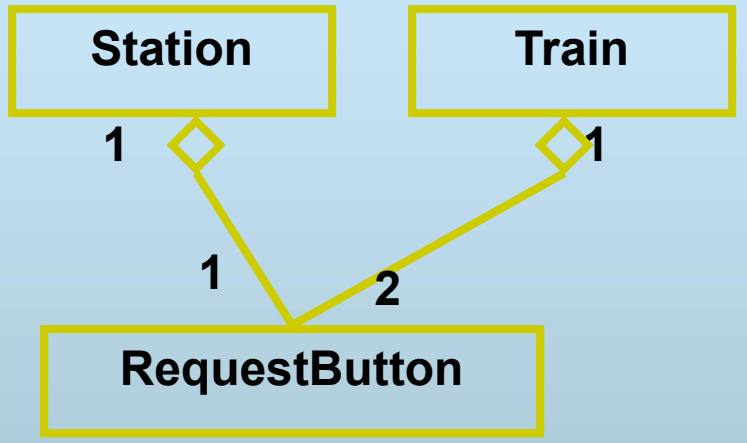
- Capture n-m relationships
  - Subsumes ER diagrams
- Label endpoints of edge with cardinalities
  - Use * for arbitrary
- Typically realized with embedded references
- Can be directional (use arrows in that case)



One request button per station; each train has two request buttons

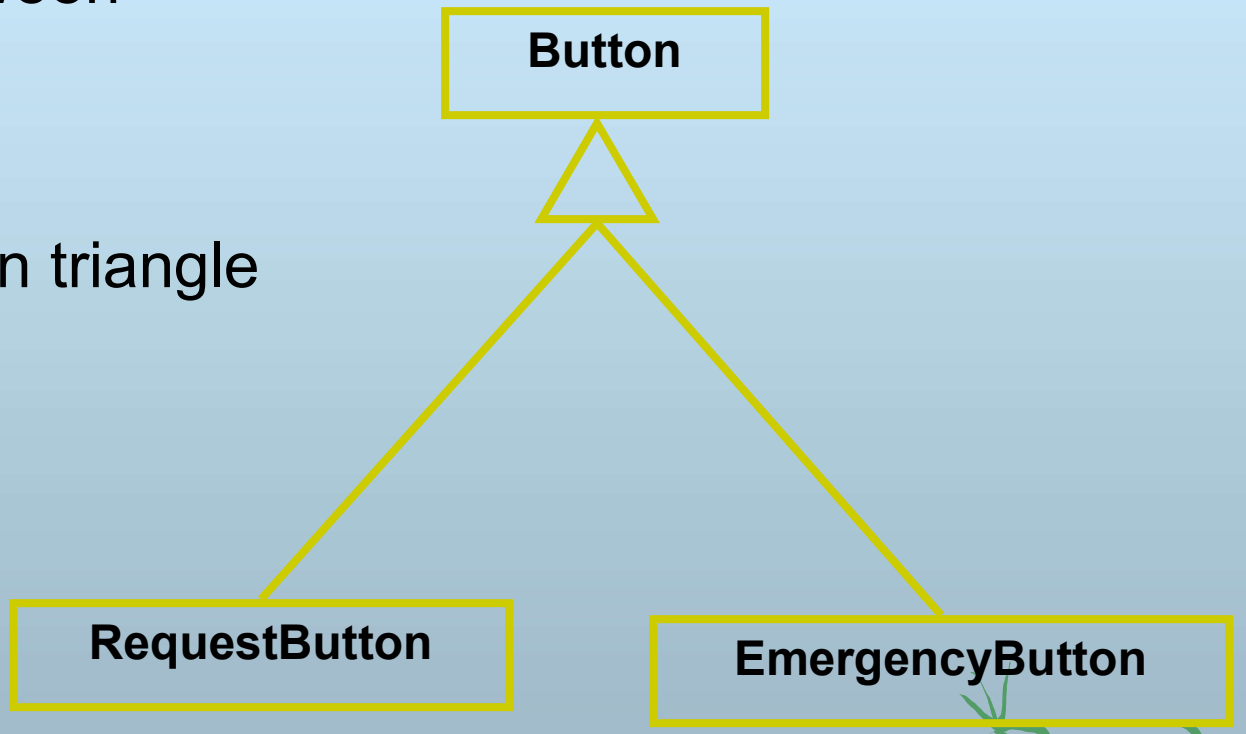
Aggregation

- Show *contains a* relationships
- Station and Train classes can contain their respective buttons
- Denoted by open diamond on the “contains” side



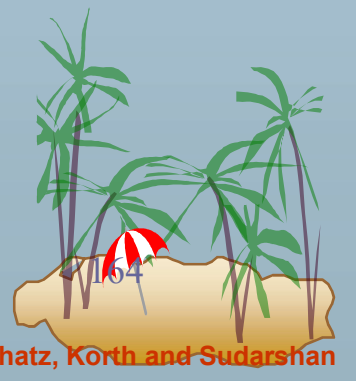
Generalization

- Inheritance between classes
- Denoted by open triangle



More about Class Diagrams

- Classes vs Objects
  - Same diagrams can be used to specify relationships between instances of classes
- Roles and Association Classes
  - More detail on relationships between classes
- Hierarchical Diagrams

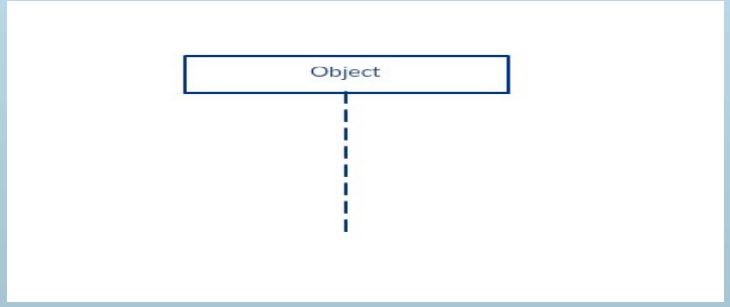


What is a Sequence Diagram?

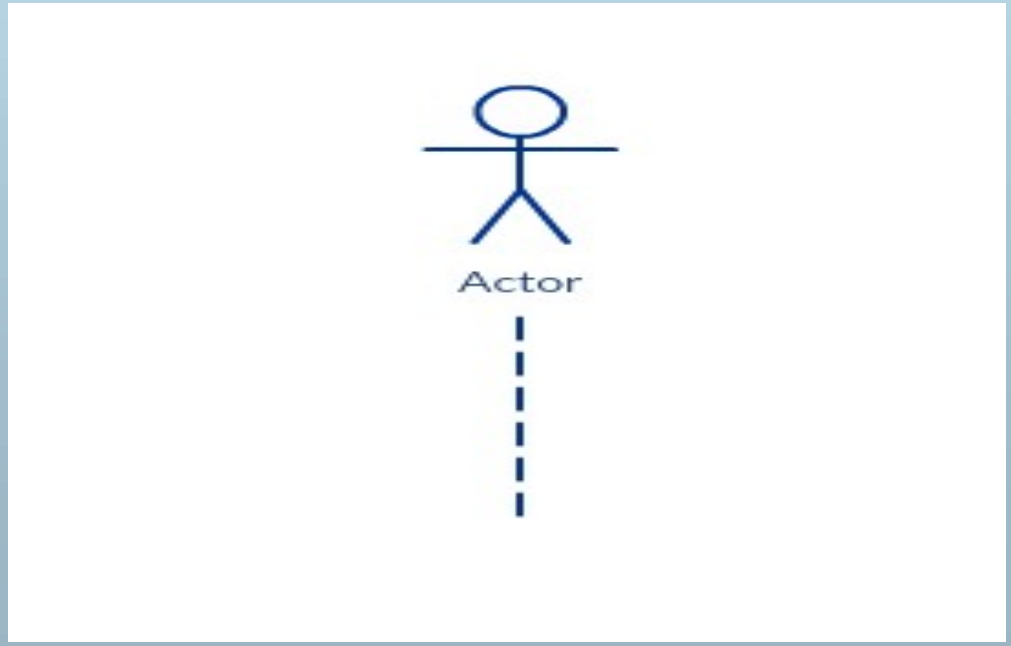
- Sequence diagrams, commonly used by developers, model the interactions between objects in a single use case.
- They illustrate how the different parts of a system interact with each other to carry out a function, and the order in which the interactions occur when a particular use case is executed.
- In simpler words, a sequence diagram shows different parts of a system work in a ‘sequence’ to get something done.

Sequence Diagram Notations

- A sequence diagram is structured in such a way that it represents a timeline which begins at the top and descends gradually to mark the sequence of interactions.
- Each object has a column and the messages exchanged between them are represented by arrows.

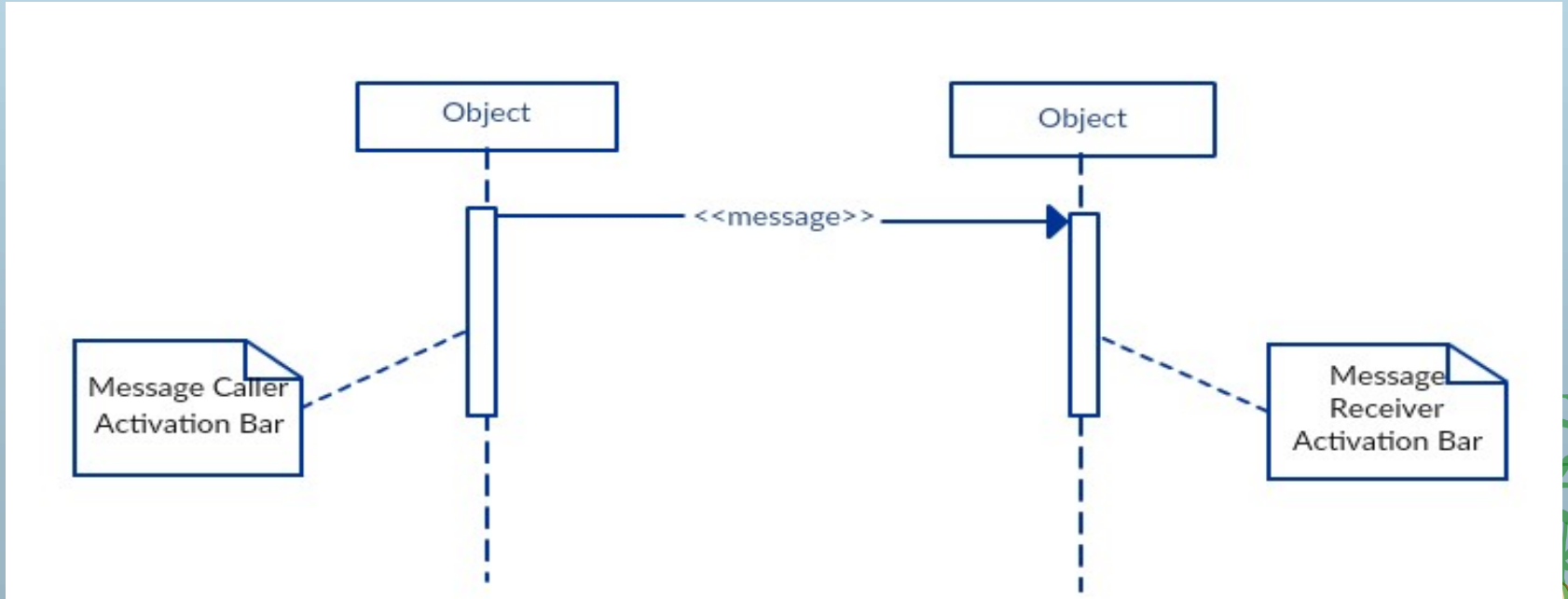


- A sequence diagram is made up of several of **these lifeline notations that should be arranged horizontally across the top of the diagram.**
- No two lifeline notations should **overlap each other.** They represent the different objects or parts that interact with each other in the system during the sequence.
- A lifeline notation with an actor element symbol is used when the particular sequence diagram is owned by a use case.



Activation Bars

- Activation bar is the box placed on the lifeline.
- It is used to indicate that an object is active (or instantiated) during an interaction between two objects. The length of the rectangle indicates the duration of the objects staying active.
- In a sequence diagram, an interaction between two objects occurs when one object sends a message to another. The use of the activation bar on the lifelines of the Message Caller (the object that sends the message) and the Message Receiver (the object that receives the message) indicates that both are active/is instantiated during the exchange of the message.



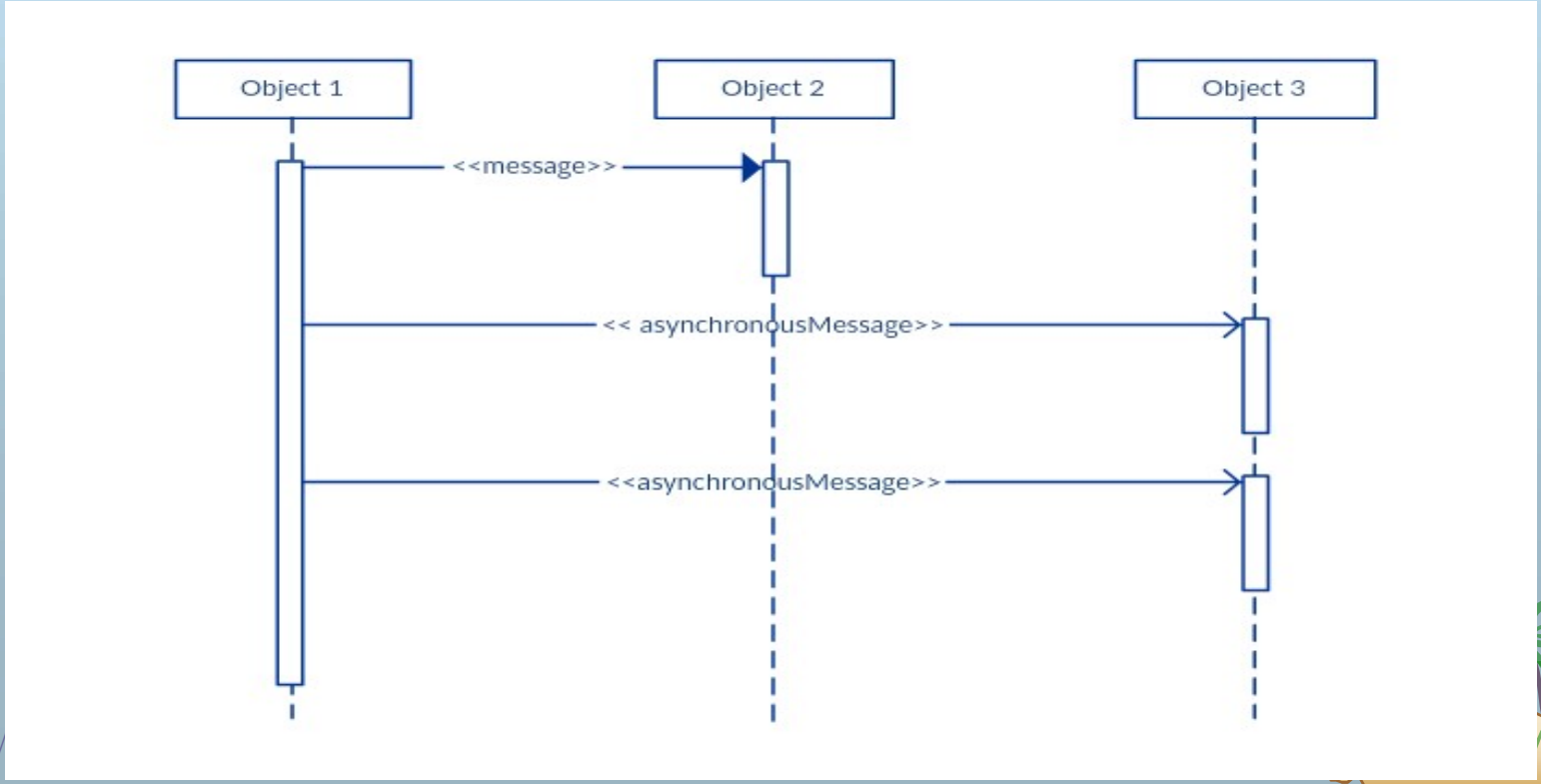
Message Arrows

- An arrow from the Message Caller to the Message Receiver specifies a message in a sequence diagram.
- A message can flow in any direction; from left to right, right to left or back to the Message Caller itself. While you can describe the message being sent from one object to the other on the arrow, with different arrowheads you can indicate the type of message being sent or received.
- The message arrow comes with a description, which is known as a message signature, on it. The format for this message signature is below. All parts except the message_name is optional.
- *attribute = message_name (arguments): return_type*
- *Synchronous message*
- As shown in the activation bars example, a synchronous message is used when the sender waits for the receiver to process the message and return before carrying on with another message. The arrow head used to indicate this type of message is a solid one, like the one below.

synchronous message

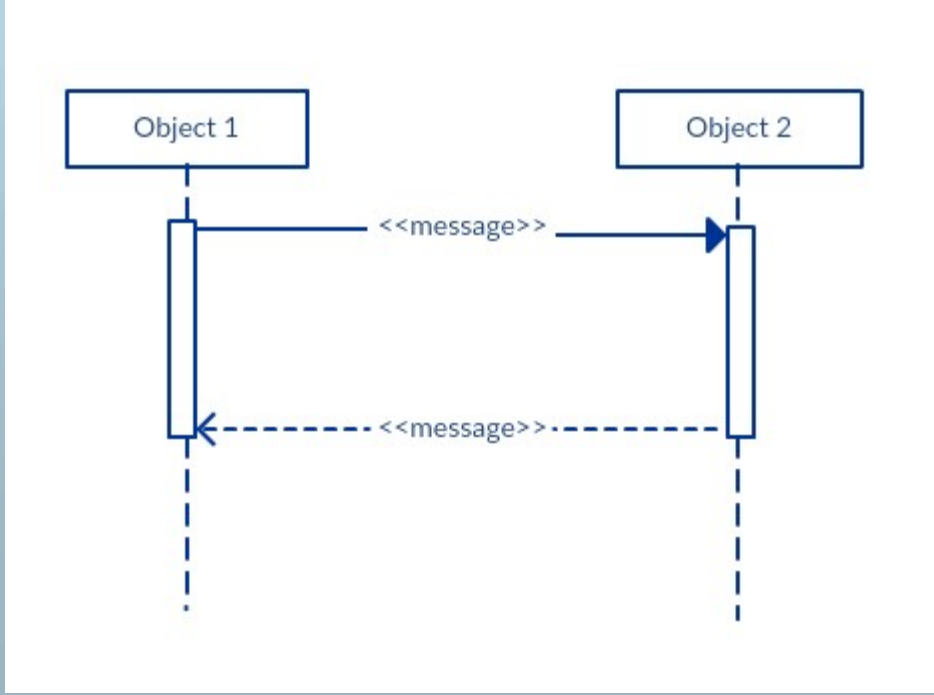
- An asynchronous message is used when the message caller does not wait for the receiver to process the message and return before sending other messages to other objects within the system. The arrow head used to show this type of message is a line arrow.

The arrow head used to indicate this type of message is a solid one,



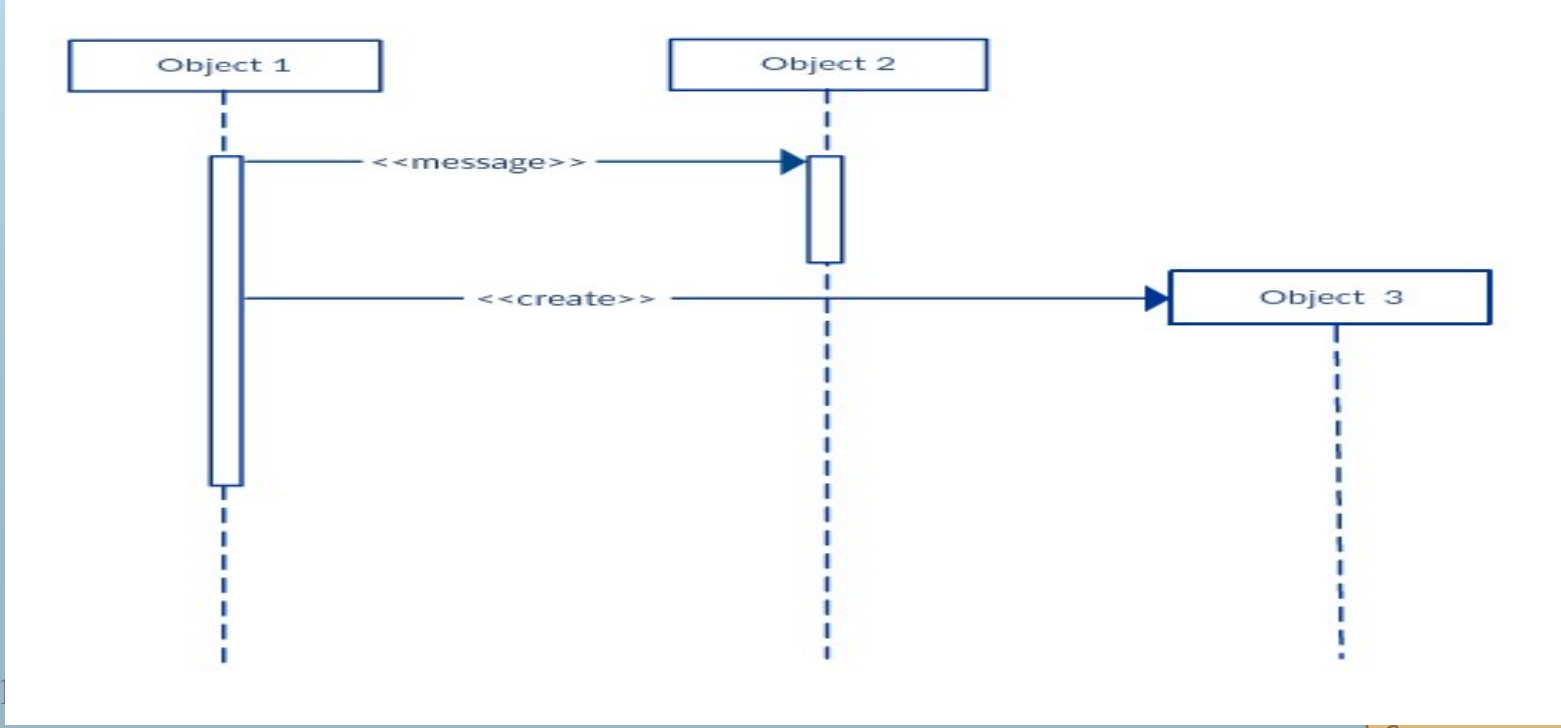
Return message

- A return message is used to indicate that the message receiver is done processing the message and is returning control over to the message caller. Return messages are optional notation pieces, for an activation bar that is triggered by a synchronous message always implies a return message.
- Tip: You can avoid cluttering up your diagrams by minimizing the use of return messages since the return value can be specified in the initial message arrow itself.



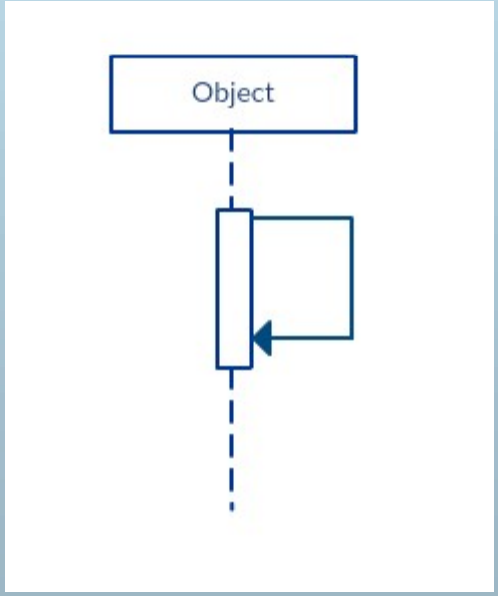
Participant creation message

- Objects do not necessarily live for the entire duration of the sequence of events. Objects or participants can be created according to the message that is being sent.
- The dropped participant box notation can be used when you need to show that the particular participant did not exist until the create call was sent. If the created participant does something immediately after its creation, you should add an activation box right below the participant box.



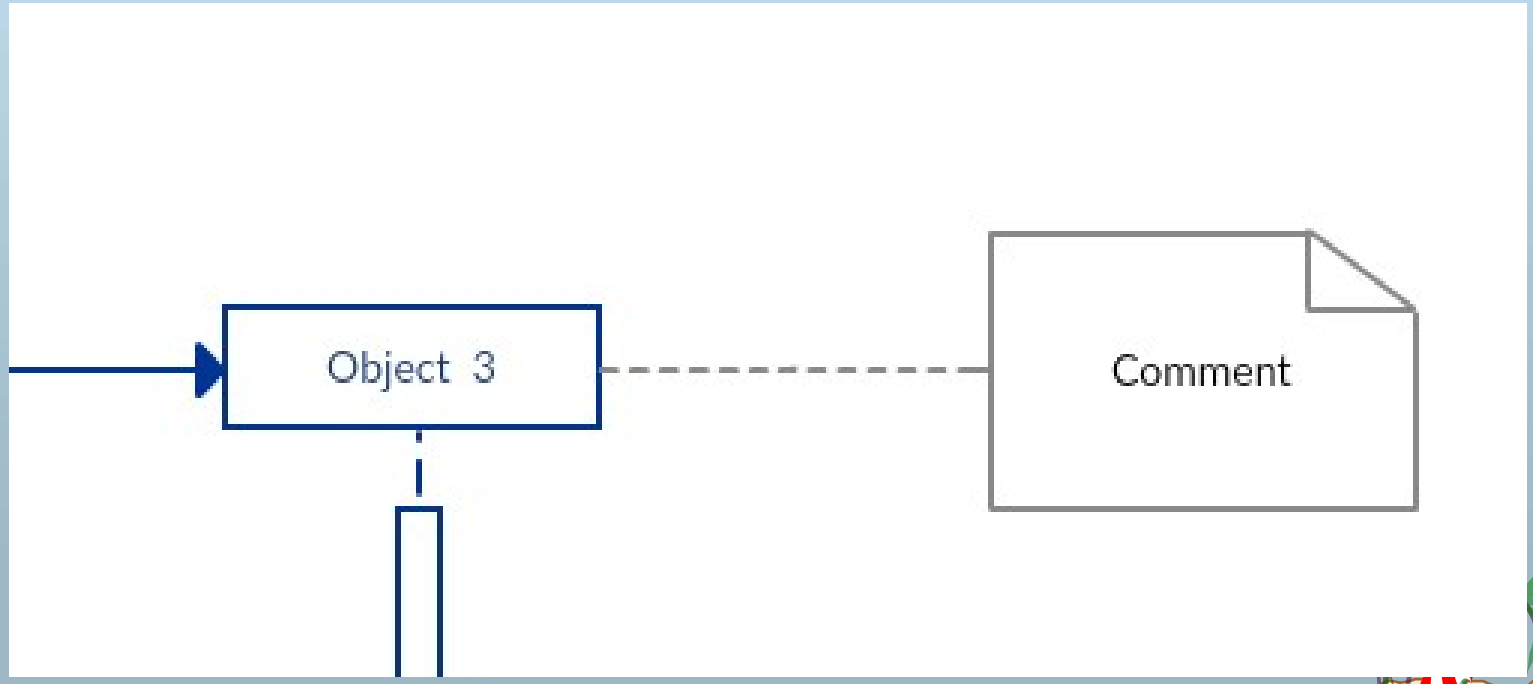
Reflexive message

- When an object sends a message to itself, it is called a reflexive message. It is indicated with a message arrow that starts and ends at the same lifeline like shown in the example below.



Comment

- UML diagramsgenerally permit the annotation of comments in all UML diagram types
- The comment object is a rectangle with a folded-over corner as shown below. The comment can be linked to the related object with a dashed line.

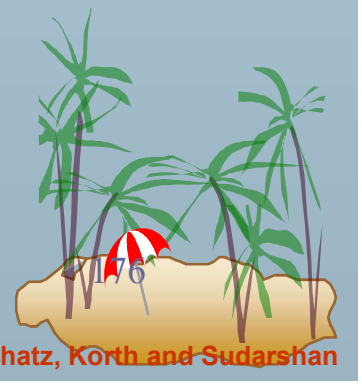


Sequence Diagram Best Practices

- **Manage complex interactions with sequence fragments**
- A sequence fragment is represented as a box that frames a section of interactions between objects (as shown in the examples below) in a sequence diagram.
- It is used to show complex interactions such as alternative flows and loops in a more structured way. On the top left corner of the fragment sits an operator. This – the fragment operator – specifies what sort of a fragment it is.
- *Alternatives*
- The alternative combination fragment is used when a choice needs to be made between two or more message sequences. It models the "if then else" logic.
- The alternative fragment is represented by a large rectangle or a frame; it is specified by mentioning 'alt' inside the frame's name box (a.k.a. fragment operator).
- To show two or more alternatives, the larger rectangle is then divided into what is called interaction operands using a dashed line, like shown in the sequence diagram example above. Each operand has a guard to test against and it is placed at the top left corner of the operand.

Sequence Diagrams

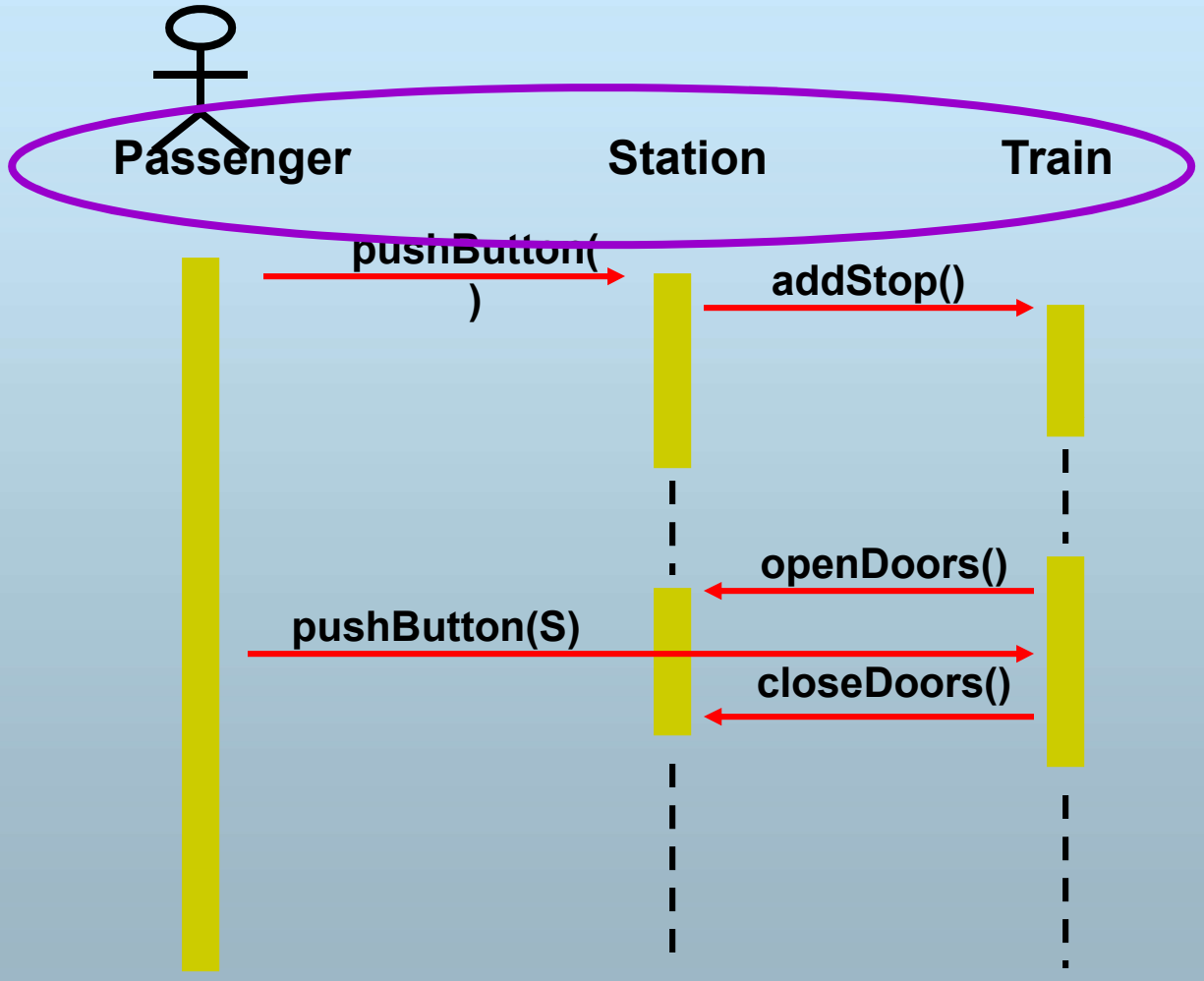
- A table
  - Columns are classes or actors
  - Rows are time steps
  - Entries show control/data flow
    - Method invocations
    - Important changes in state



Sequence diagram (optional)

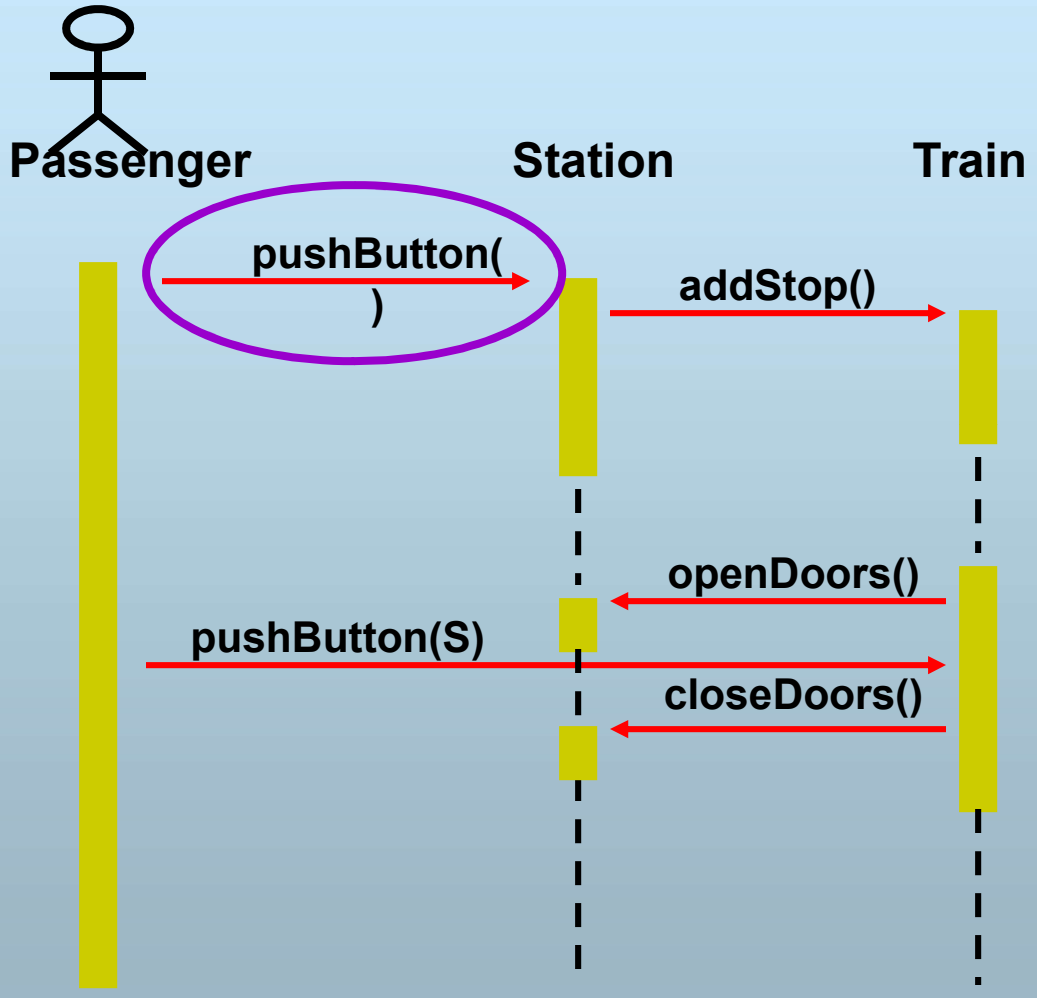
- Objects or entities are diagrammed at the top
- Each object's life is represented by a vertical line from creation to destruction
- Messages or events are diagrammed from the sending object to the receiving object, in the order in which they occur
- Responses may or may not be diagrammed, depending on complexity/obviousness
- These are sometimes called 'swim lane' diagrams
  - Swim lanes can be used in activity diagrams as well

Example Sequence Diagram



Classes & Actors

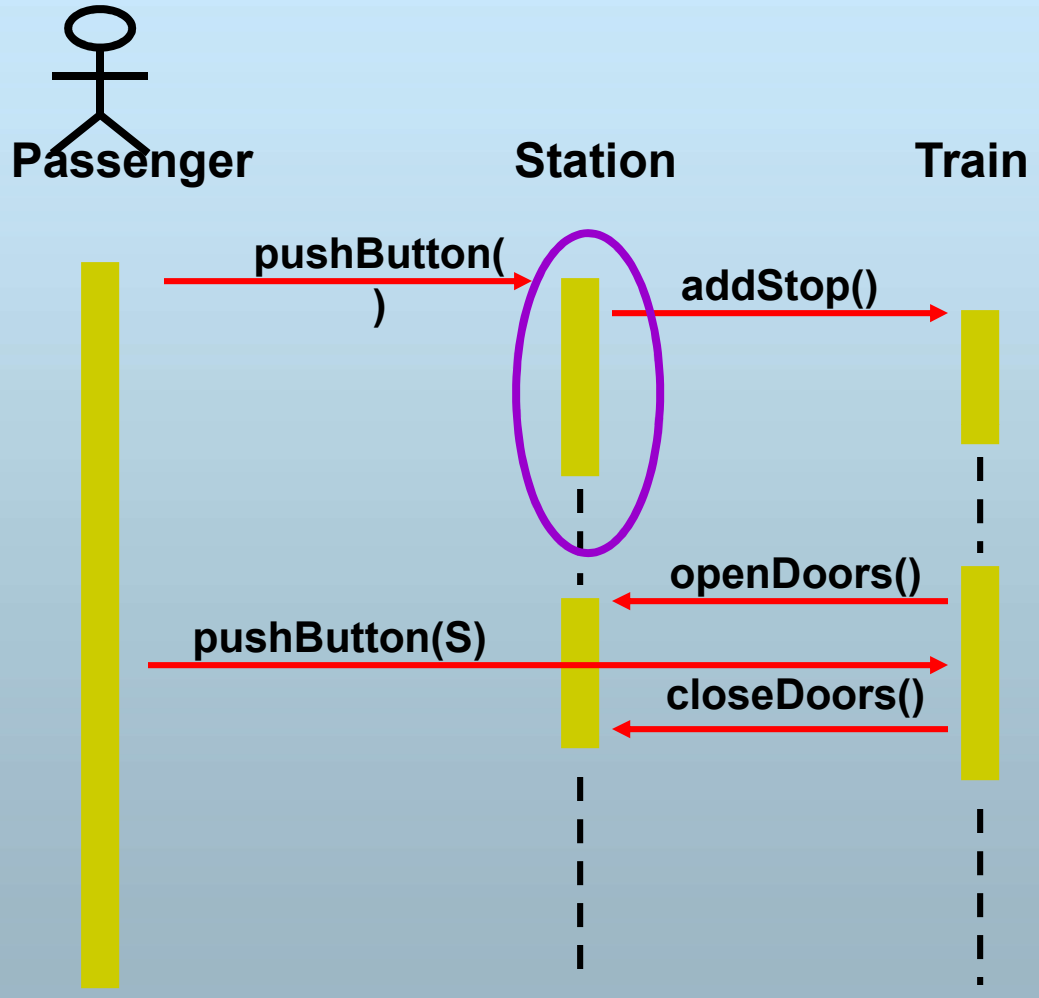
Example Sequence Diagram



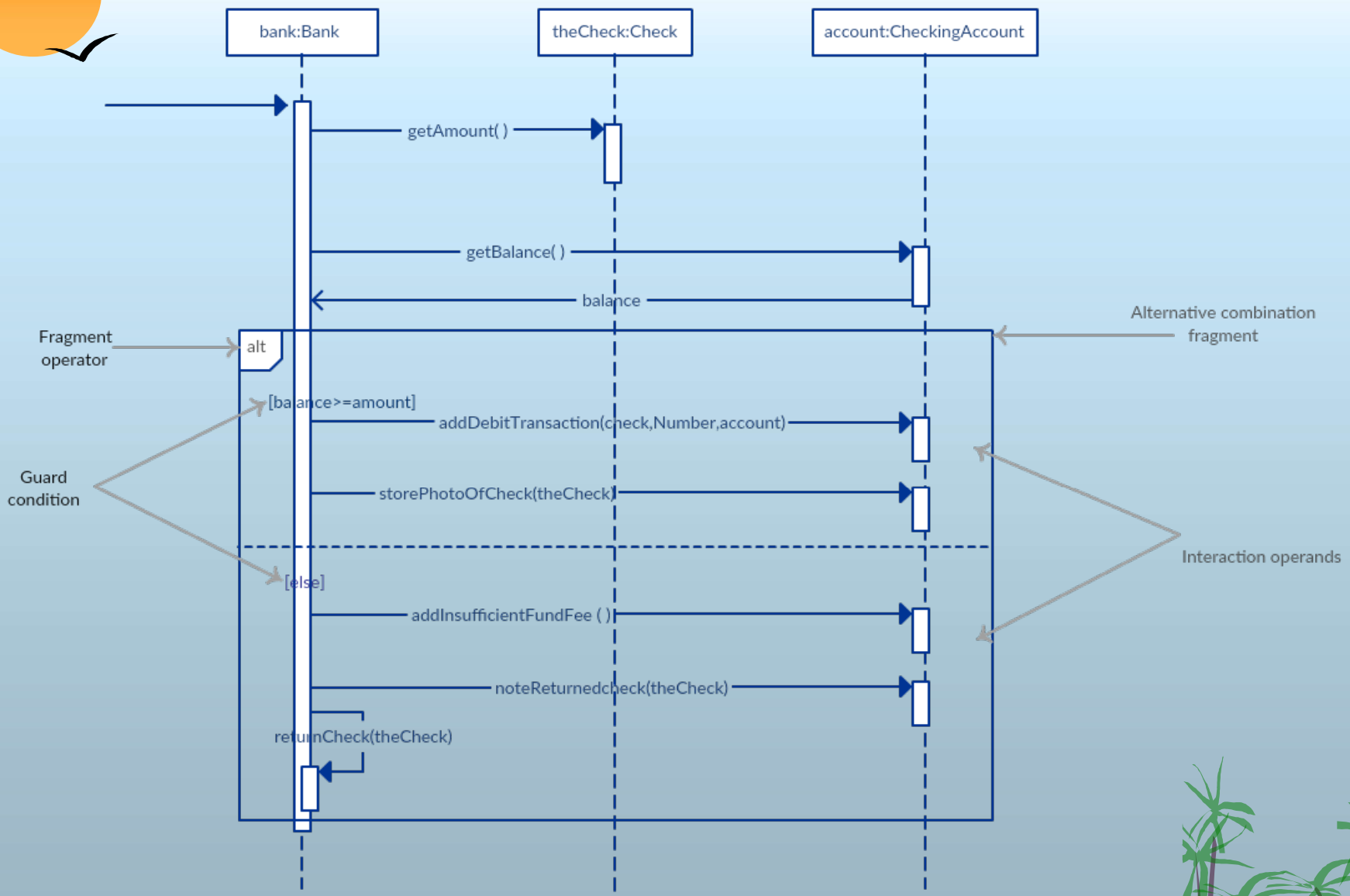
Method invocation

Note: These are all synchronous method calls. There are other kinds of invocations.

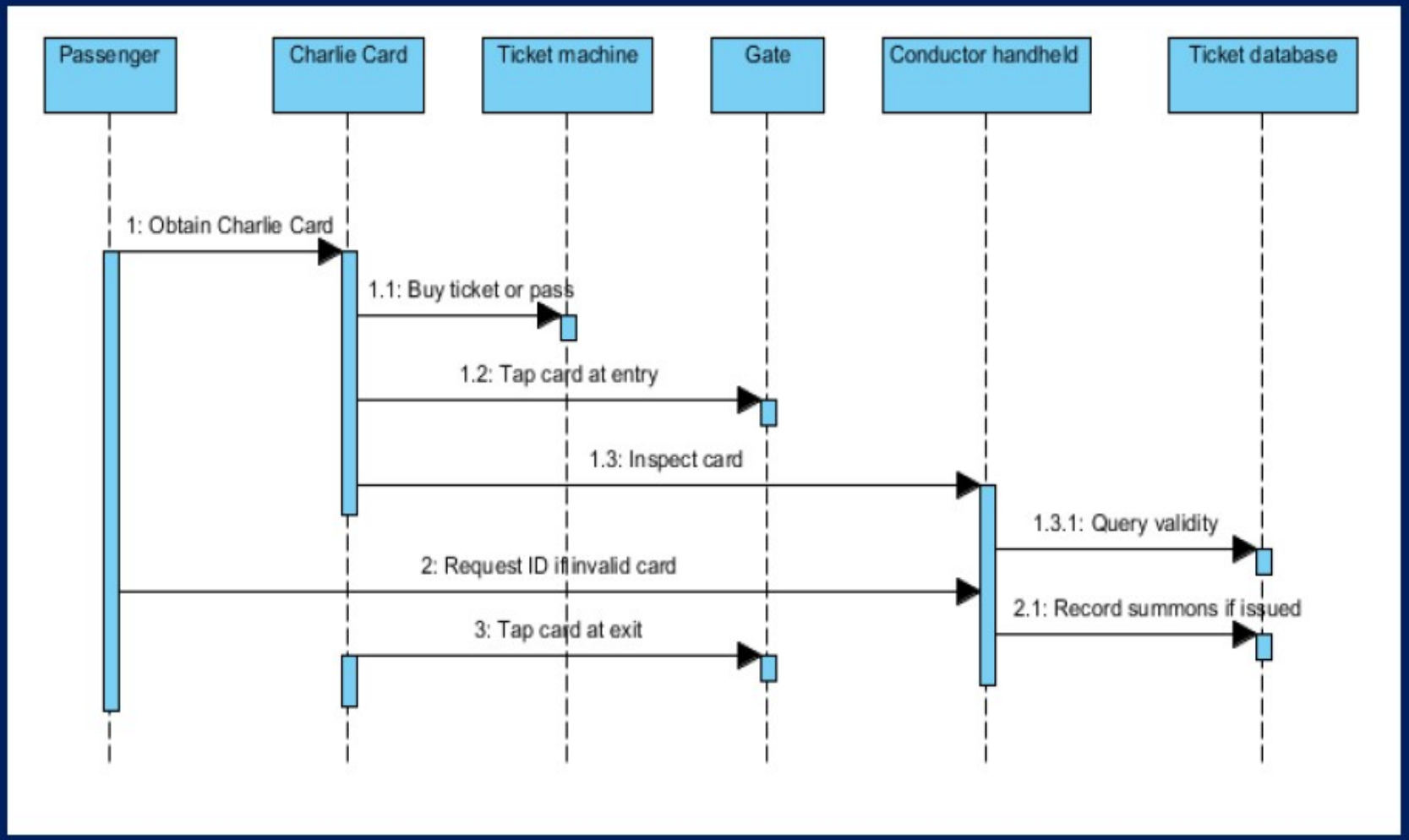
Example Sequence Diagram



Invocation lifetime spans lifetimes of all nested invocations



Sequence diagram example

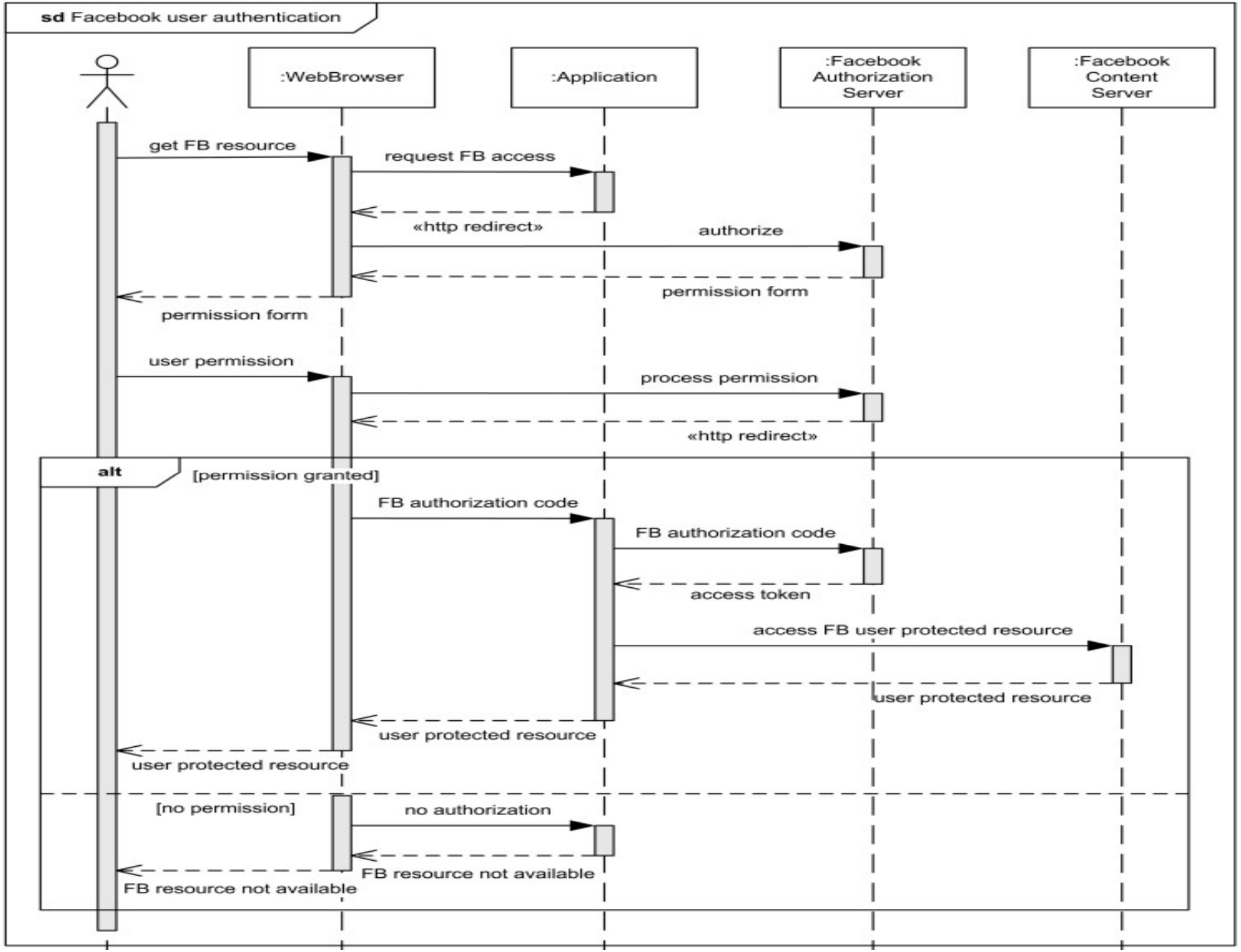


Facebook Web User Authentication

- An example of UML sequence diagram which shows how **Facebook** (FB) user could be authenticated in a web application to allow access to his/her FB resources.
- Facebook uses **OAuth 2.0** (**OAuth 2** is an authorization framework that enables applications to obtain limited access to user accounts on an HTTP service, such as Facebook, GitHub, and DigitalOcean. ... **OAuth 2** provides authorization flows for web and desktop applications, and mobile devices. _)
- protocol framework which enables web application (called "client"), which is usually not the FB resource owner but is acting on the FB user's behalf, to request access to resources controlled by the FB user and hosted by the FB server. Instead of using the FB user credentials to access protected resources, the web application obtains an access token.

Cont…

- Web application should be registered by Facebook to have an application ID (client_id) and secret (client_secret).
- When request to some protected Facebook resources is received, web browser ("user agent") is redirected to Facebook's authorization server with application ID and the URL the user should be redirected back to after the authorization process.
- User receives back Request for Permission form. If the user authorizes the application to get his/her data, Facebook authorization server redirects back to the URI that was specified before together with authorization code ("verification string"). The authorization code can be exchanged by web application for an OAuth access token.

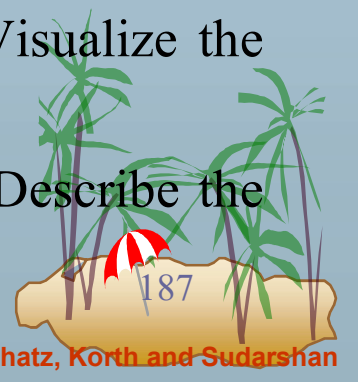

sd Facebook user authentication
:WebBrowser
:Application
:Facebook Authorization Server
:Facebook Content Server
get FB resource
request FB access
«http redirect»
authorize
permission form
permission form
user permission
process permission
«http redirect»
alt
[permission granted]
FB authorization code
FB authorization code
access token
access FB user protected resource
user protected resource
user protected resource
user protected resource
[no permission]
no authorization
FB resource not available
FB resource not available

UML Component Diagrams

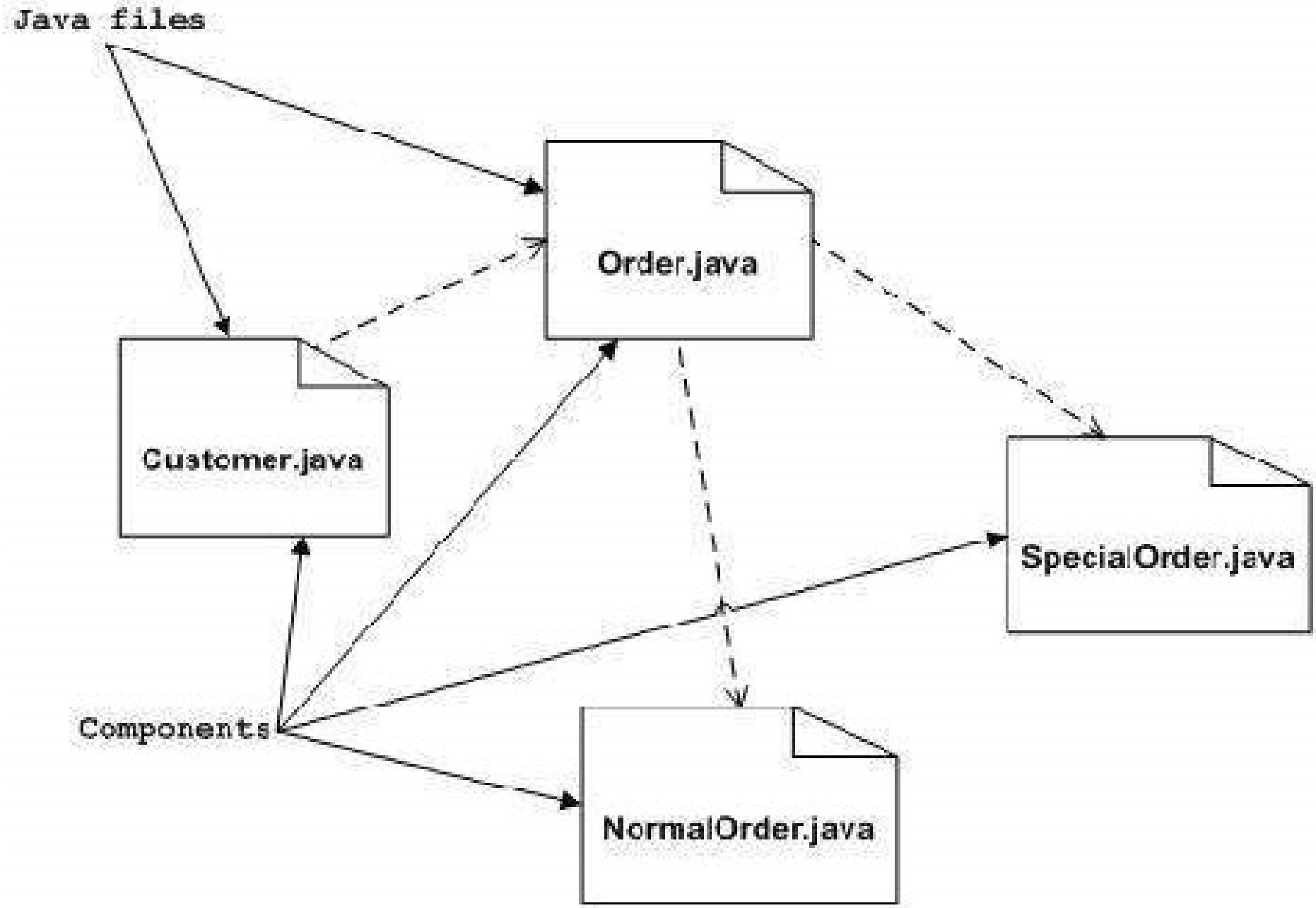
- Component diagrams are different in terms of nature and behavior.
- Component diagrams are used to model the physical aspects of a system. Now the question is
- what are these physical aspects?
- Physical aspects are the elements such as executables, libraries, files, documents, etc. which reside in a node.
- Component diagrams are used to visualize the organization and relationships among components in a system.
- These diagrams are also used to make executable systems.

Purpose of Component Diagrams

- Component diagram is a special kind of diagram in UML. The purpose is also different from all other diagrams discussed so far.
- It does not describe the functionality of the system but it describes the components used to make those functionalities.
- Thus from that point of view, component diagrams are used to visualize the physical components in a system. These components are libraries, packages, files, etc.
- Component diagrams can also be described as a static implementation view of a system. Static implementation represents the organization of the components at a particular moment.
- A single component diagram cannot represent the entire system but a collection of diagrams is used to represent the whole.
- The purpose of the component diagram can be summarized as Visualize the components of a system.
- Construct executables by using forward and reverse engineering.Describe the organization and relationships of the components.



Component diagram of an order management system



Where to Use Component Diagrams?

- component diagrams are used to visualize the static implementation view of a system. Component diagrams are special type of UML diagrams used for different purposes.

Deployment diagrams

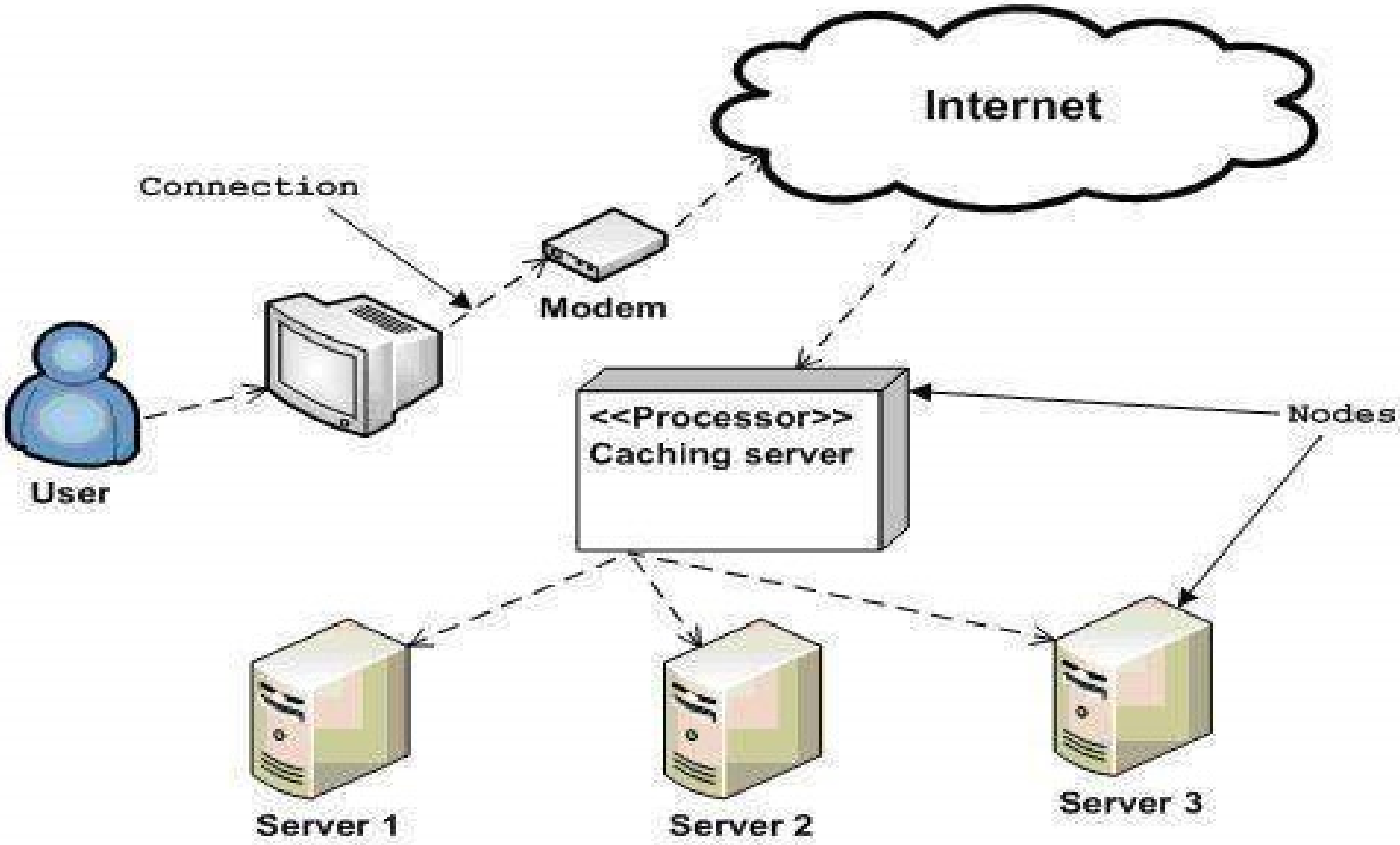
- Deployment diagrams are used to visualize the topology of the physical components of a system, where the software components are deployed.
- Deployment diagrams are used to describe the static deployment view of a system. Deployment diagrams consist of nodes and their relationships.

Purpose of Deployment Diagrams

- The term Deployment itself describes the purpose of the diagram. Deployment diagrams are used for describing the hardware components, where software components are deployed. Component diagrams and deployment diagrams are closely related.
- Component diagrams are used to describe the components and deployment diagrams shows how they are deployed in hardware.
- UML is mainly designed to focus on the software artifacts of a system. However, these two diagrams are special diagrams used to focus on software and hardware components.
- Most of the UML diagrams are used to handle logical components but deployment diagrams are made to focus on the hardware topology of a system. Deployment diagrams are used by the system engineers.

- **Deployment diagrams are useful for system engineers. An efficient deployment diagram is very important as it controls the following parameters −**
- **Performance**
- **Scalability**
- **Maintainability**
- **Portability**
- Before drawing a deployment diagram, the following artifacts should be identified −
- **Nodes**
- **Relationships among nodes**
- Following is a sample deployment diagram to provide an idea of the deployment view of order management system. Here, we have shown nodes as −
- **Monitor**
- **Modem**
- **Caching server**
- **Server**

Deployment diagram of an order management system



UML Summary

- Use UML while writing scenarios and narratives as an initial requirements document
  - Diagram use cases, then refine them into scenarios
  - Focus on completeness of use cases
- Use UML component diagrams to list all system elements
  - Focus on completeness, and use to set system boundaries
- Prepare the initial data model (next lecture)
  - Add operations/methods to the entities, after understanding the data, to create a class diagram
- Use UML state diagrams, sequence diagrams and activity diagrams to specify objects and processes
  - Prepare these selectively for complex or interesting objects
- UML is becoming a 'universal' language: staff coming to a project know it, which sharply reduces learning curve
  - Developers and analysts can both understand it readily
  - Consultants/analysts use UML even for analysis-only projects (as well as writing requirements and modeling data)
  - Business process execution language (BPEL) in Web lectures is UML extension to directly create systems

“Software Defined Cities: Integrating the Cyber World with Internet of Things”

By

Prof.Antonio Puliafito (Italy) Antonio