Chapter Three

Design with UML

Chapter Outlines

- Unified Modeling Language Diagrams
  - Structural Diagram
    - Class, Object, and Package Diagram
    - Component, Node, and Deployment Diagram
  - Behavioral Diagram
    - Use case Diagram
    - State Machine Diagram
    - Activities Diagram
    - Sequence Diagram

Problems

- Software system is getting increasingly complex. It always require a team of programmers to develop
- Each programmer will response to part of the system development and share their codes between developers. However, it is not easily understandable for developers who did not write that codes
- We need a simpler and standard way to present the complex systems for sharing information
  - For people to understand the role of each object
  - For people to understand the relationship between objects

What is modeling?

- Modeling consists of building an abstraction of reality
- Abstractions are simplifications because:
  - They ignore irrelevant details
  - They only represent the relevant details
- What is *relevant* or *irrelevant* depends on the purpose of the model

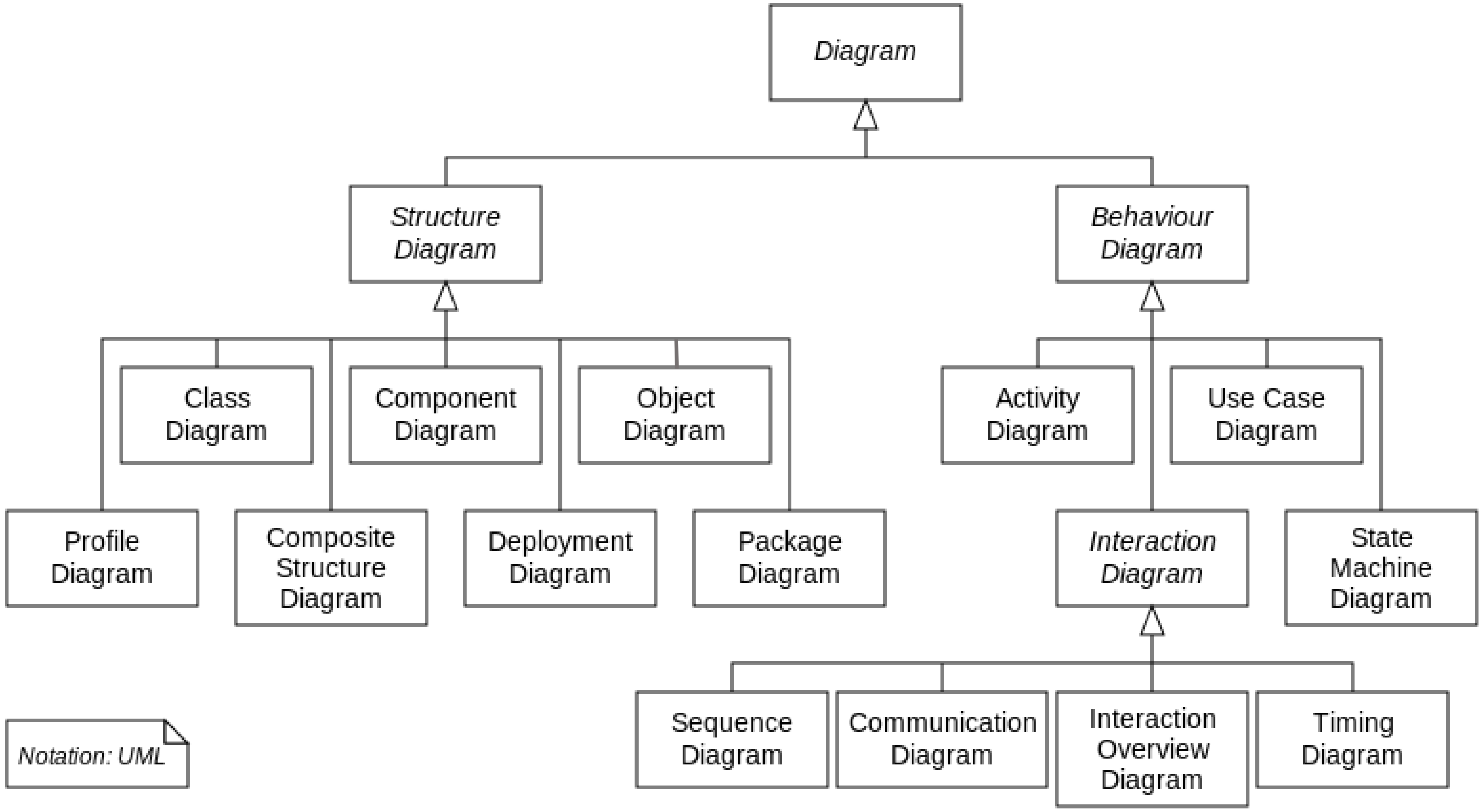
What is UML?

- Unified Modeling Language (UML) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems
- It has a direct relation with OO analysis and design
- It is not a programming language but UML diagrams can convert to programming codes with some powerful tools
- Common UML design tools
  - Commercial: Visual Paradigm, Microsoft Visual Studio, Microsoft Visio, etc.
  - Free: www.gliffy.com, www.draw.io, etc.

UML Diagrams

- Different UML diagrams are used for different purposes
- Structural Modeling Diagram
  - Class, Object, Package, Deployment, etc.
- Behavioral Modeling Diagram
  - Use case, Activities, State Machine, Sequence, etc.
- Architectural Modeling represents the overall architecture of the system. It contains both structural and behavioral elements

UML Architecture

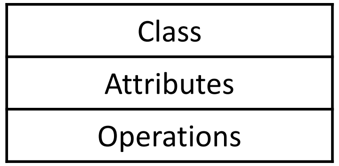


UML Building Blocks

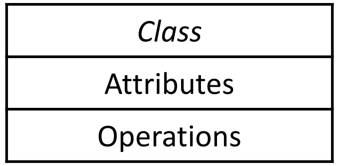
- In order to create the UML diagrams, we must use the UML building blocks
- Building blocks are the syntaxes of UML and they can be classified as
  - Things
    - Class, Interface, Use Case, Component, Node, State Machine, Package, Note, etc.
  - Relationships
    - Association, Dependency, Generalization, Realization, etc.

Fundamental Notations

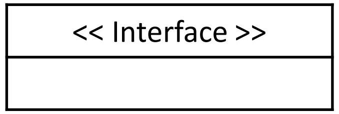
- ***Class*** represents set of objects having similar responsibilities



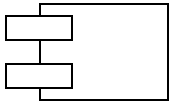
- ***Abstract Class*** defines a super class (Class name is italic)



- ***Interface*** defines a set of operations which specify the responsibility of a class

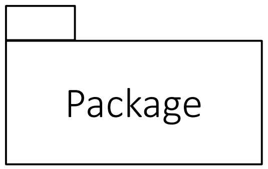


- ***Component*** describes the part of a system

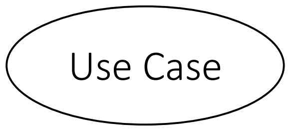


Fundamental Notations (cont.)

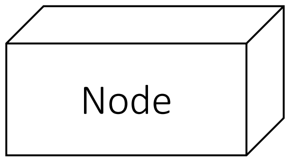
- ***Package*** is the only one grouping thing available for gathering structural and behavioral things



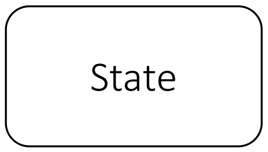
- ***Use case*** represents a set of actions performed by a system for a specific goal



- A ***node*** can be defined as a physical element that exists at run time

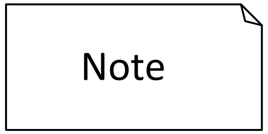


- ***State Machine*** defines the sequence of states an object goes through in response to events



Fundamental Notations (cont.)

- A ***note*** is used to render comments, constraints, etc. of an UML element



- ***Association*** is basically a set of links that connects elements of an UML model

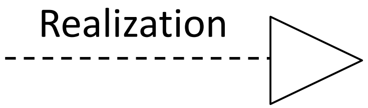
- ***Interaction*** is defined as a behavior that consists of a group of messages exchanged among elements to accomplish a specific task

- ***Dependency*** is a relationship between two things in which change in one element also affects the other one

Fundamental Notations (cont.)

- ***Generalization*** describes the inheritance relationship in the world of objects

- ***Realization*** defines the relationship of two elements. One element describes some responsibility which is not implemented and the other one implements them



- ***Aggregation*** is a relationship of an instance of class A holds a collection of instances of class B

- ***Composition*** is a relationship of an instance of class A holds a reference to an instance of class B is entirely contained by A

UML vs Codes

- Forward Engineering
  - Following the UML model to write Java codes
  - Oracle NetBeans IDE is able to generate Java codes from UML diagrams
- Reverse Engineering
  - Read the Java codes to generate UML model
- Roundtrip Engineering
  - Move between forward and reverse engineering
  - Useful when requirements are changing frequently

Class Diagram

- It is the most commonly used UML diagram
- It describes the attributes (variables) and operations (methods) of a class
- Many class diagrams together to form a ***System architecture***
- It shows a collection of classes, interfaces, associations, and constraints. So, it is known as structural diagram
- It is used for development purpose because they can be converted to real programming codes

Recall the Student Class

```
public class Student {
  private int studentID;
  private String studentName;

  public Student(int id, String name) {
    this.studentID = id;
    this.studentName = name;
  }

  public int getStudentID() {
    return studenID;
  }

  public String getStudentName() {
    return studentName;
  }
}
```

Class Diagram Example

- Student Class has two variables and three methods
- First row defines the **class**
- Second row defines the **attributes**
    - Data type is written at the end after a colon
    - + symbol for *public*, - symbol for *private*, and # symbol for *protected*
- Third row defines the **operations**
    - *in* (optional) to specify the input parameters

Student
-studentID:int -studentName:String
+Student(in id:int, in name:String) +getStudentID():int +getStudentName():String

Recall Shape Superclass

```
public abstract class Shape {
  public abstract double getArea();
  public abstract double getPerimeter();

  public String getName() {
    return this.getClass().getSimpleName();
  }

  public void showInfo() {
    System.out.println(getName() + " Information:");
    System.out.println("Area is " + getArea());
    System.out.println("Perimeter is " + getPerimeter());
  }
}

```

Recall Rectangle Class

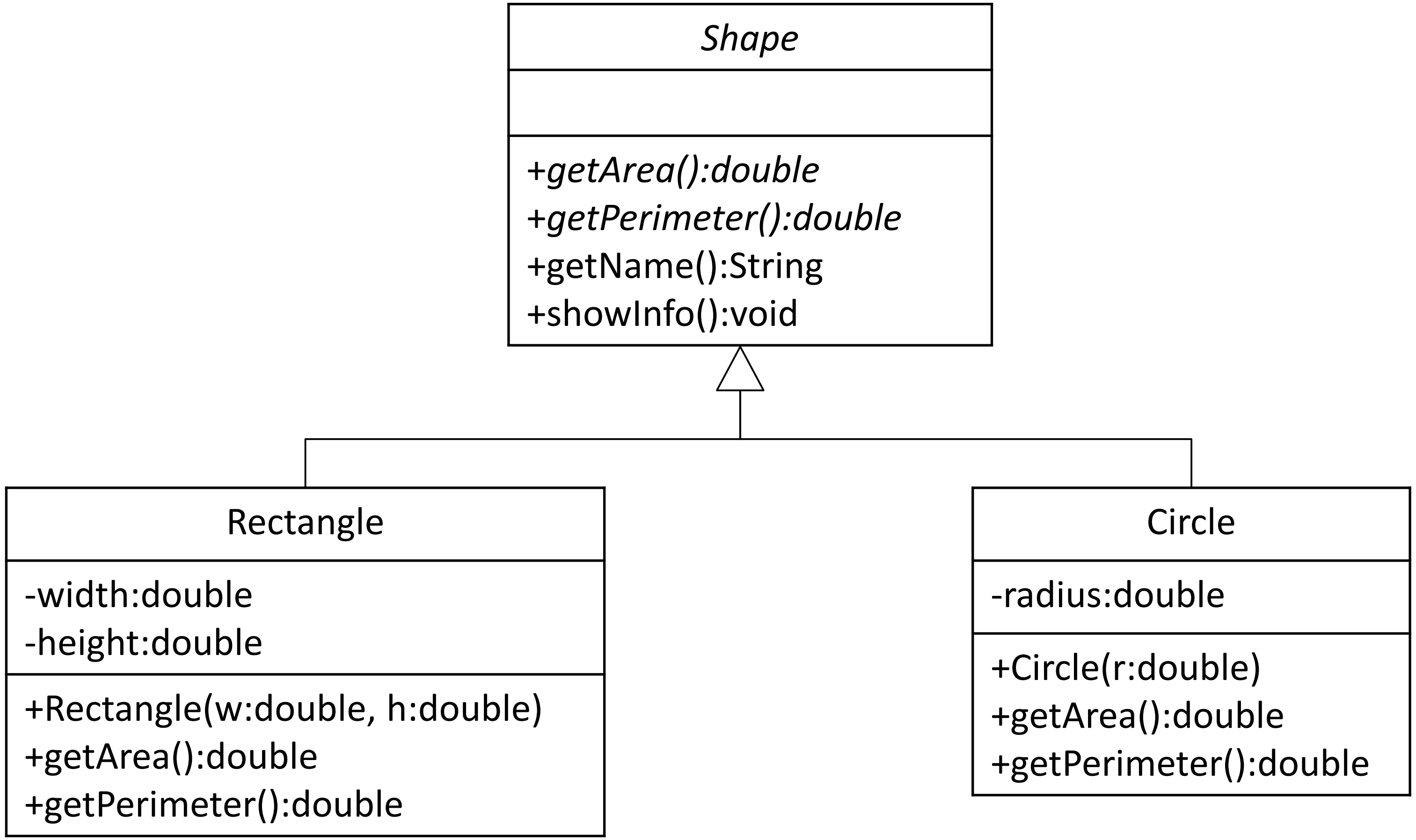
```
public class Rectangle extends Shape {
  private double width, height;

  public Rectangle(double w, double h) {
    this.width = w;
    this.height = h;
  }
  @Override
  public double getArea() {
    return width * height;
  }
  @Override
  public double getPerimeter() {
    return 2 * (width + height);
  }
}

```

Inheritance

- *Rectangle and Circle* classes extend the superclass *Shape*

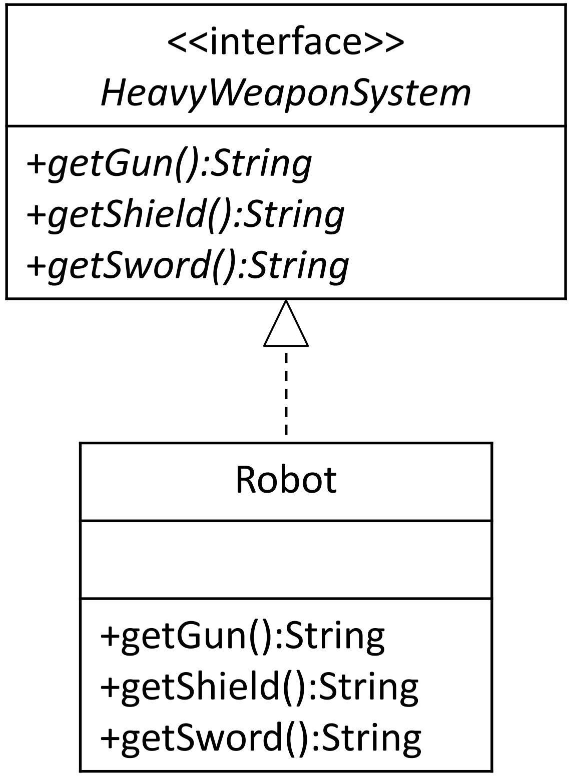


Interface

- Interface class is surrounded by double arrows
- *HeavyWeaponSystem* interface has three operations

```
public interface HeavyWeaponSystem {
  public String getGun();
  public String getShield();
  public String getSword();
}
```

```
public class Robot implements HeavyWeaponSystem {
  public String getGun() {…}
  public String getShield() {…}
  public String getSword() {…}
}
```



Object Diagram

- Object diagrams can be described as an instance of the class diagram
- It is near the real life scenarios where we implement a system
- Object diagrams are a set of objects and their relationships just like class diagrams
- It represents the static view of the system
- The usage of object diagrams is similar to class diagrams but they are used to build prototype of a system

Object Diagram Example

- *JohnSmith* object is the instance of *Student* Class

johnSmith:Student
studendID = 1234 studentName = "John Smith"

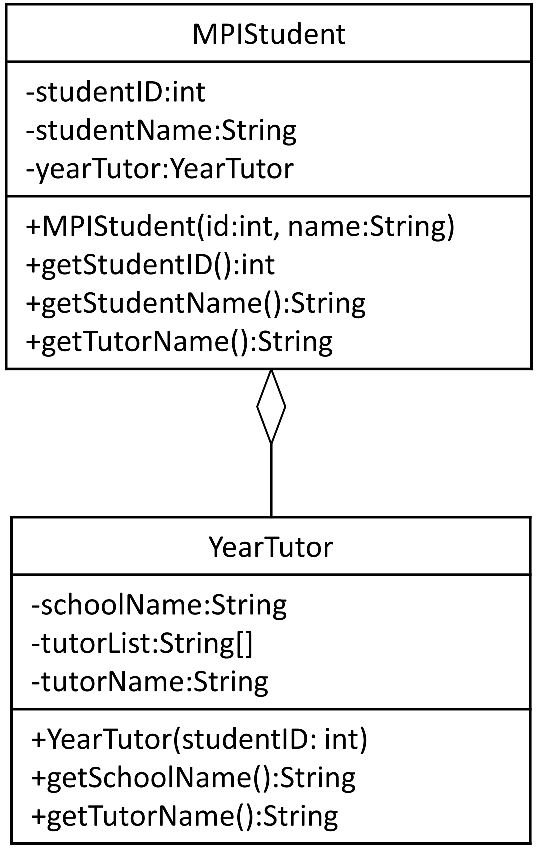
- First row writes the instance name and it is underlined
- Second row writes the variable names and their values

When to use Object Diagram?

- In general, object diagrams will not only contain a single object
- It can be imagined as a ***snapshot*** of a running system in a particular moment
- A *ClassroomSystem* has *Student* classes and the object diagram is used to show all the students in a particular classroom
  - Room A214 has 44 student instances at 11:00

Aggregation Example

- A *MPIStudent* class encapsulates a *YearTutor* class



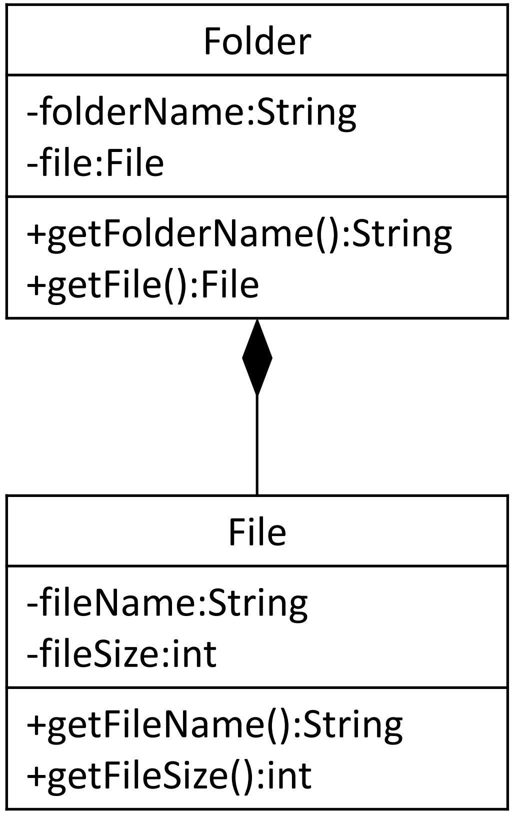
Strong Type Aggregation

- ***Composition*** is a strong type aggregation
- It composites the parent class
- A *Person* class encapsulates (has a) a *Body* class
- If a *Person* dies, his/her *Body* will also die

```
public class Person {
  private String name;
  private int age;
  private String gender;
  private Body humanBody; // Strong type aggregation
  …
}
```

Composition Example

- A *Folder* class encapsulates a *File* class
- *Folder* could contain many files, while each File has exactly one Folder parent
- *If Folder is deleted, all contained Files are deleted as well*



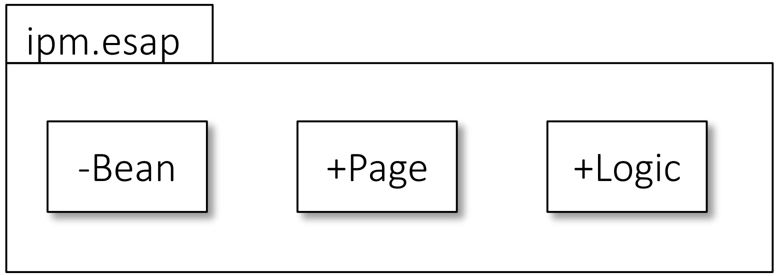
www.c-sharpcorner.com/UploadFile/b1df45/dependency-generalization-association-aggregation-compos

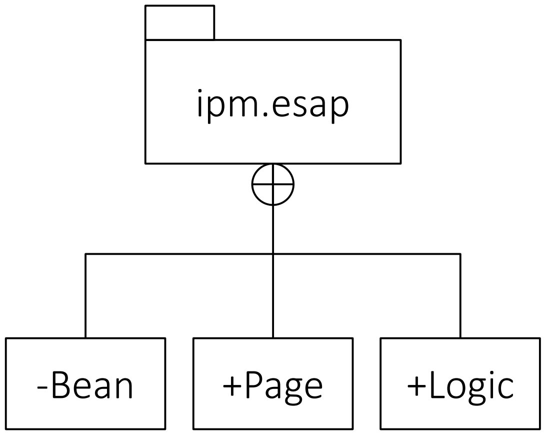
Package Diagram

- Package defines a namespace for elements
- Package can be defined as a file folder to hold the same type of programs in groups physically
- Package diagram organizes the model elements into groups, making the UML diagrams simpler and easier to understand
- Package diagram must contain a package name and can optionally show the elements within the package
- Two special types for defining package relationship
    - Package import and Package merge

Package Diagram Example

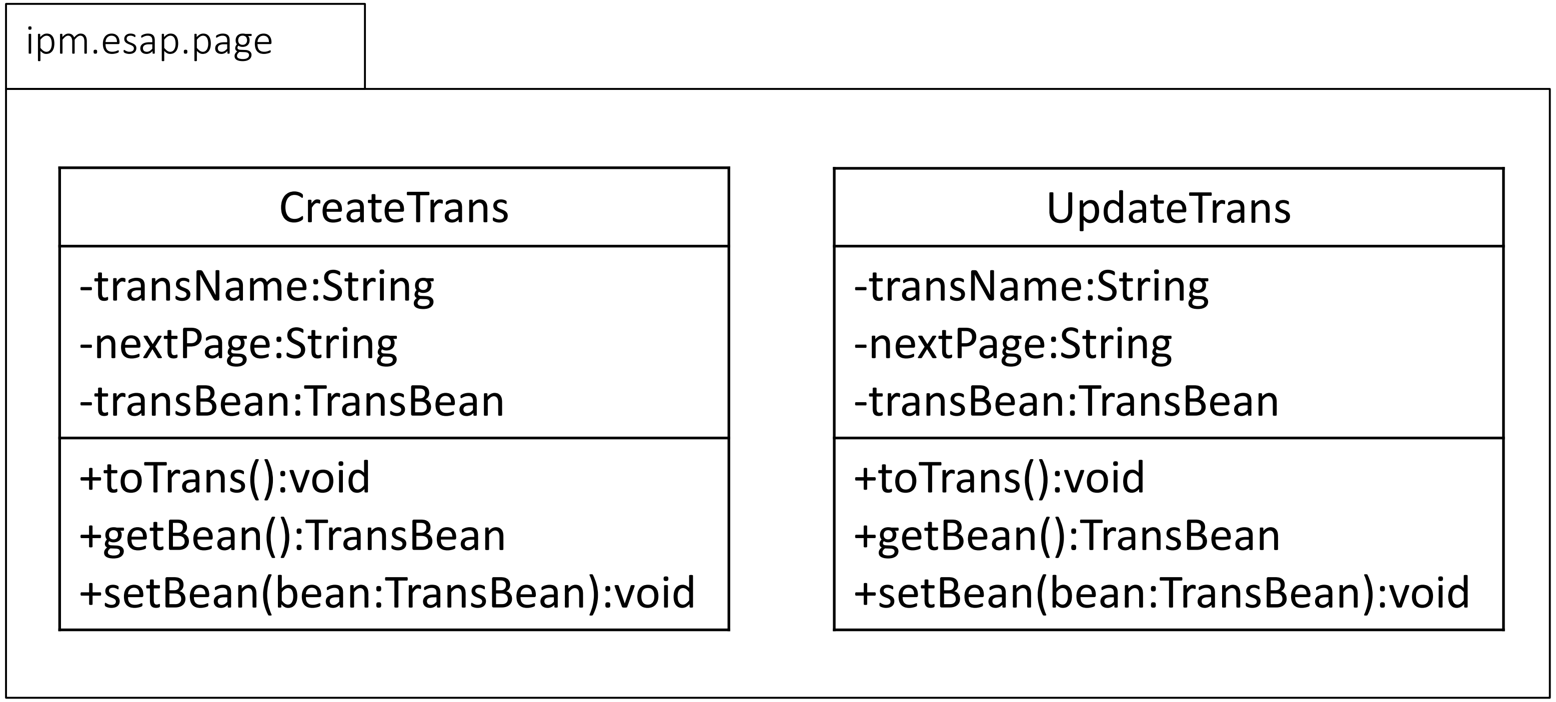
- A package with a name *ipm.esap*
- A Package can contain many elements (packages and classes)
- This package has three elements and it can be shown in both ways
  - ipm.esap.Bean, ipm.esap.Page, and ipm.esap.Logic





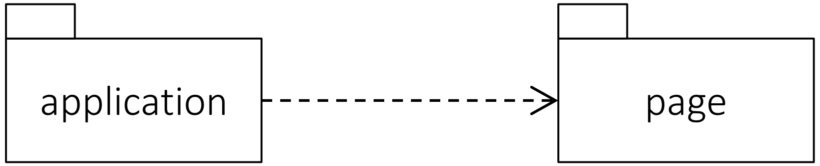
Package Diagram Example

- A package contains two classes



Package Import

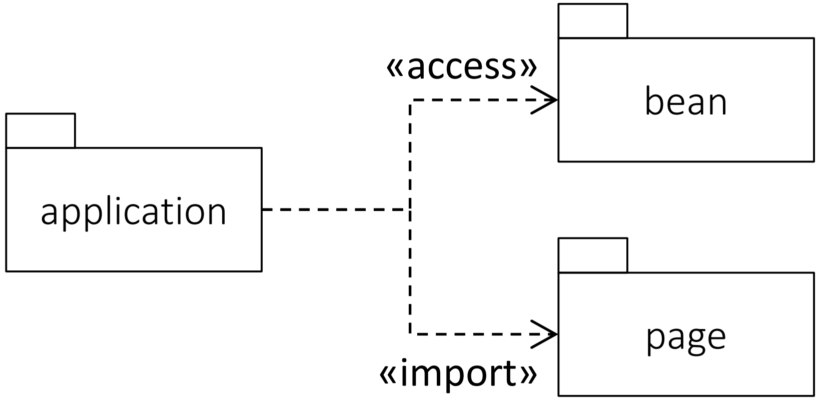
- A package import is a directed relationship between an importing namespace and imported package, that allows the use of unqualified names to refer to the package members from the other namespace(s)
- The following example shows the *application* package imports the *page* package
- It looks exactly the same as the *dependency* relationship
  - If *page* package changes, the *application* package will also change




```
package application;
import page.*;
public class MyApps {
…
}
```

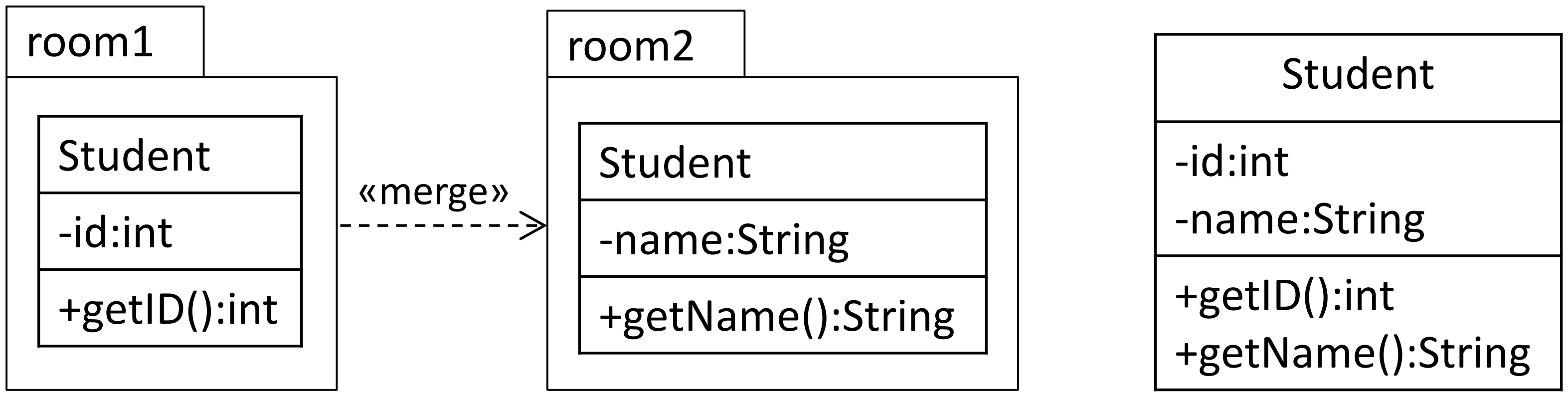
Package Import Types

- There are two types of import: public or private
- The imported elements are added to namespace but
  - They are visible outside the namespaces in public import
  - They are invisible outside the namespaces in private import
- The keywords to use are *«import»* for public and *«access»* for private package import
  - private package import of bean
  - public package import of page



Package Merge

- A package merge is a directed relationship between two packages to indicate that the contents of the two packages are to be combined
- A package merge can be viewed as an operation that takes the contents of two packages and produces a new package that combines all the contents
- It is like the *generalization* relationship that the source elements add the characteristics of target elements to its own

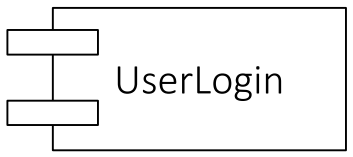


Component Diagram

- A component diagram has a higher level of abstraction than a Class Diagram - usually a component is implemented by one or more classes (or objects) at runtime
- Component diagrams are used to visualize the architecture-level artifact
- It does not describe the functionality of the system but it describes the components used to make those functionalities
- It can model the business software architecture, the technical software architecture
- However, physical architecture issues, in particular hardware issues, are better addressed via UML deployment diagrams

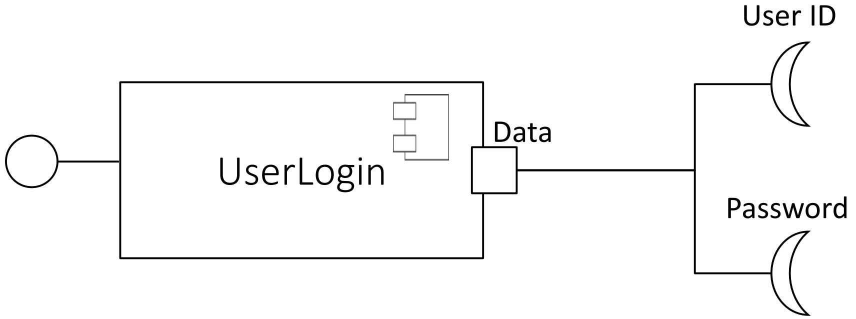
Component Interfaces

- A system can contain many components
  - User login, user purchase component, etc.
- Components have different types of interface
  - Provided Interfaces
    - A component provides interfaces to other components as the output
    - It use a lollipop notation to represent
  - Required Interfaces
    - A component requires an interface as its input
    - It use a socket notation to represent



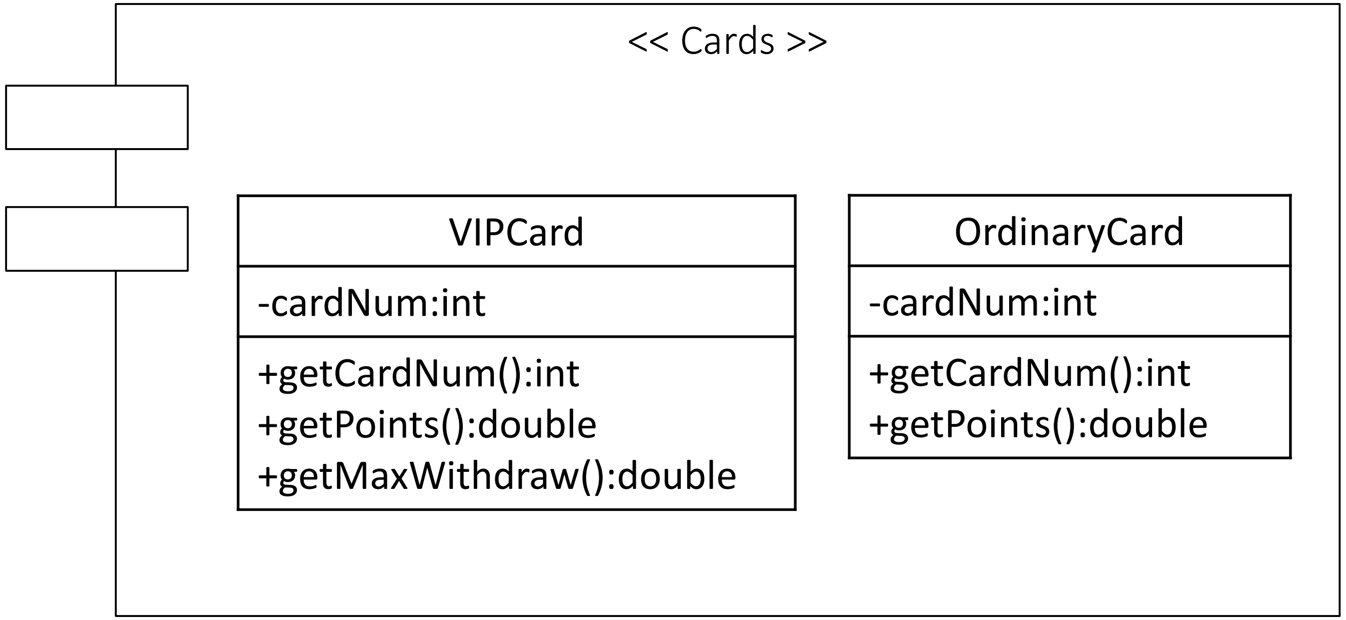
Component Ports

- Components have ports for multiple interfaces
- A port is a feature of a classifier that specifies a distinct interaction point between the classifier and its environment
- Ports are depicted as small squares on the sides of classifiers



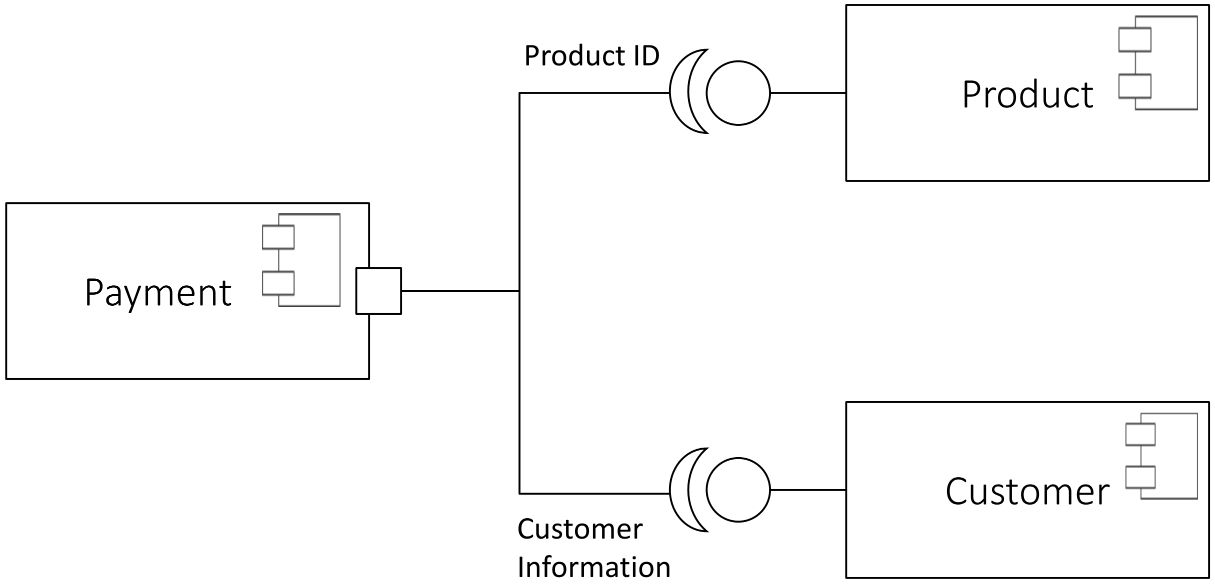
Cards Component Diagram

- A component can contain some classes
- Cards component has two classes
  - VIP card and Ordinary card



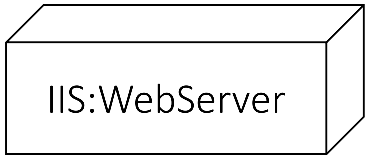
Online Store System

- An online store system has three basic components
  - *Payment* component requires product ID and customer information
  - *Product* component provides product information
  - *Customer* component provides customer information



Node Diagram

- Node diagram is a computational resource upon which UML artifacts may be deployed for execution
- It usually represent two things
  - Device nodes: hardware devices
    - server machine, smart phone, etc.
  - Execution environments: software containers
    - operation systems, JVM, web containers, web server, application server, database server, etc.

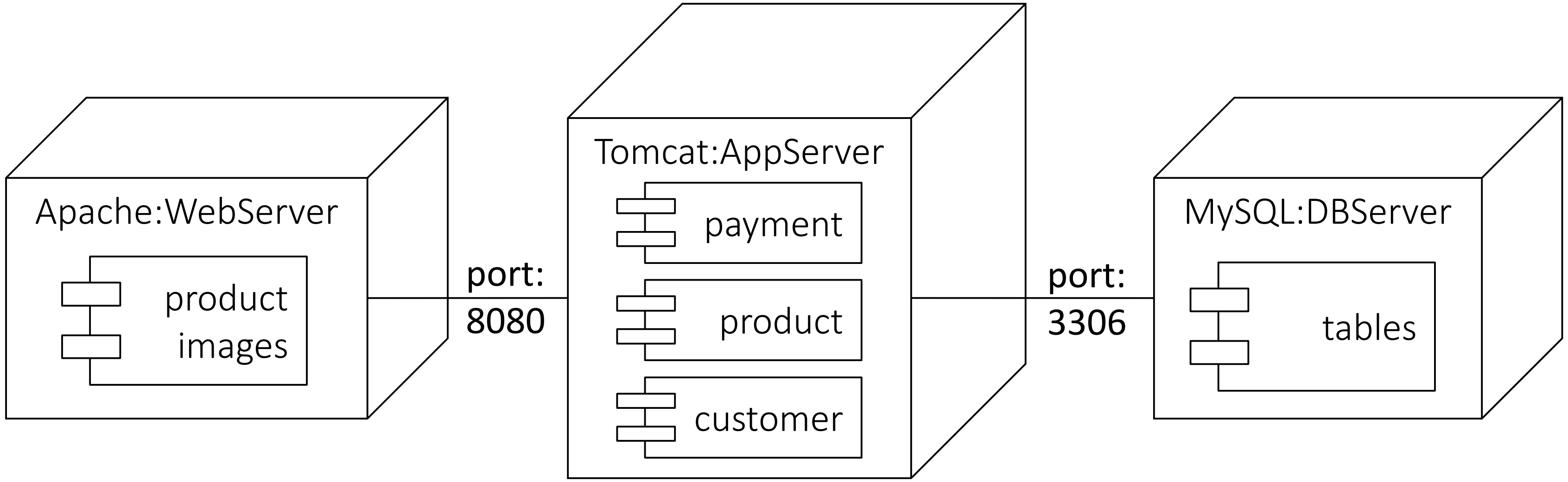


Deployment Diagram

- Deployment diagrams are a set of nodes and their relationships
- Nodes are physical entities where the components are deployed
- Deployment diagrams are used for visualizing deployment view of a system
- It is generally used by the deployment team

Deployment Diagram Example

- An online store system may consists of
  - Web Server (Apache HTTP Server): to hold static content (HTML, images, etc.)
  - Application Server (Apache Tomcat): to hold dynamic programs (Java Servlet, JSP, JavaBean, etc.)
  - Database Server (MySQL): to keep data in tables

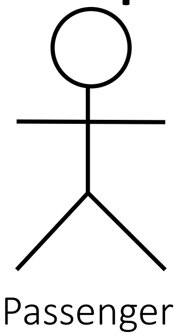


Use Case Diagram

- Use case diagrams are a set of use cases, actors and their relationships
- It is used during requirements elicitation and analysis to represent external behaviors (visible from the outside of the system)
- An ***Actor*** represents a role, a user type of the system
- An ***Use Case*** represents a class of functionality provided by the system
- It is focus on presenting the functions of the system in the ***user's point-of-view*** (outside the system)

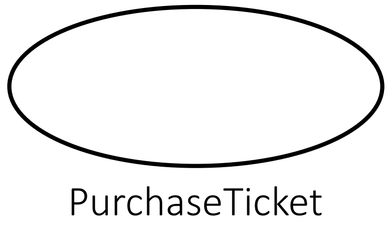
Actors

- An actor is a model for an external entity which interacts (communicates) with the system
  - User type of the system (administrator, manager, etc.)
  - External system (Another system)
  - Physical environment (e.g. Weather)
- An actor has a unique name and an optional description
  - *Passenger*: A person in the train
  - *GPS satellite*: An external system that provides the system with GPS coordinates



Textual Use Case

- Use cases can be described textually, with a focus on the event flow between actor and system
- The textual use case description consists of 6 parts:
  1. Unique name
  2. Participating actors
  3. Entry conditions
  4. Exit conditions
  5. Flow of events
  6. Special requirements



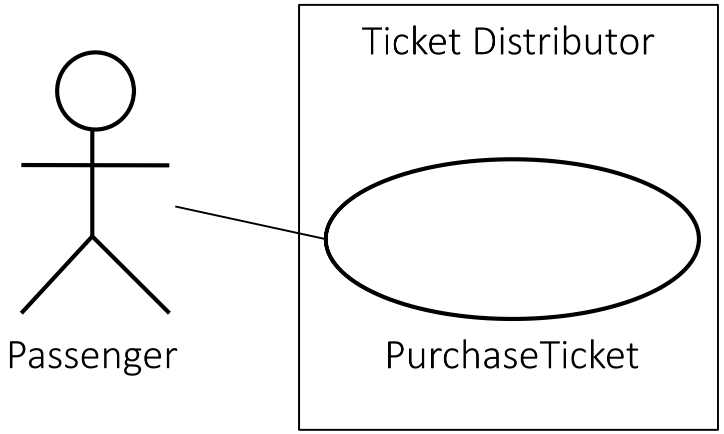
- On the other hand, we can start writing down all the descriptions before drawing the diagram

Textual Use Case Example

1. Unique Name
   - Purchase ticket
2. Participating actor
   - Passenger
3. Entry conditions
   - Passenger stands in front of ticket distributor
   - Passenger has sufficient money to purchase ticket
4. Exit condition
   - Passenger has ticket
5. Flow of events
   - Passenger selects the number of zones to be traveled
   - Ticket Distributor displays the amount due
   - Passenger inserts money, at least the amount due
   - Ticket Distributor returns the change
   - Ticket Distributor issues ticket
6. Special requirements
   - Cash only

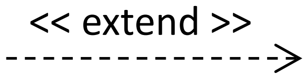
Use Case Diagram Example

- A Ticket Distributor for passenger to buy tickets
- An **association** is existed whenever an actor is involved with an interaction described by an use case
- An association between actor and use case is represented by a solid line
- System boundary boxes is ***optionally*** to represent the scope of a system

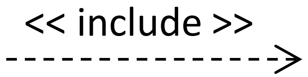


Relationships of Use Case Diagram

- Three commonly used relationships in Use Case Diagram are
  - Extend
    - It is the exceptional functions



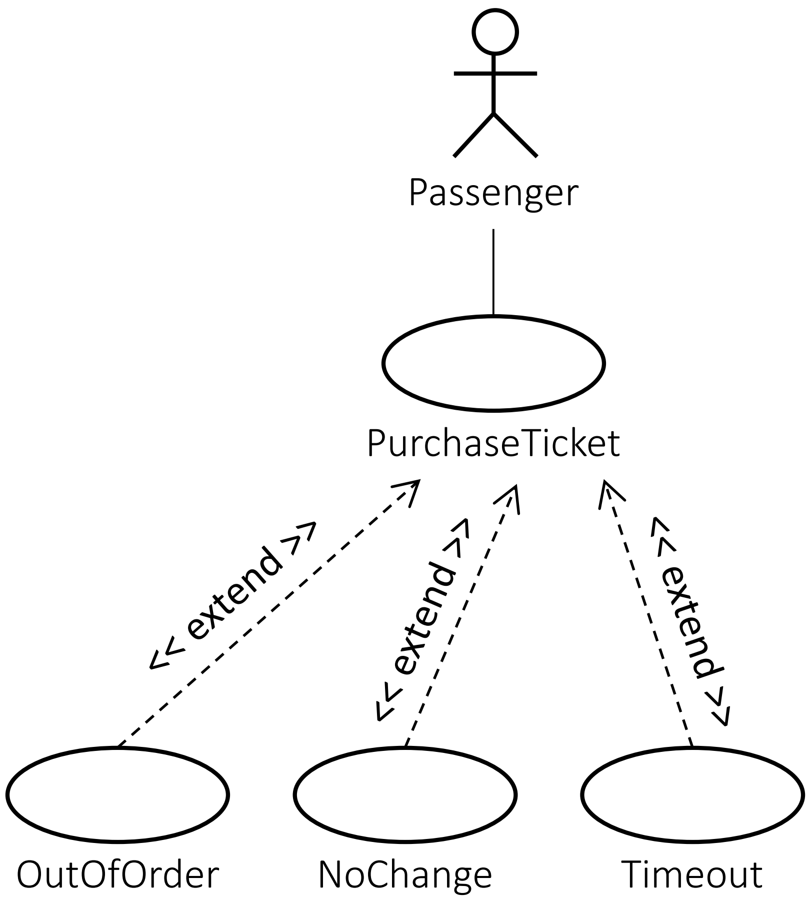
  - Include
    - It calls other procedures or methods



  - Inheritance
    - It inherits the target behaviors

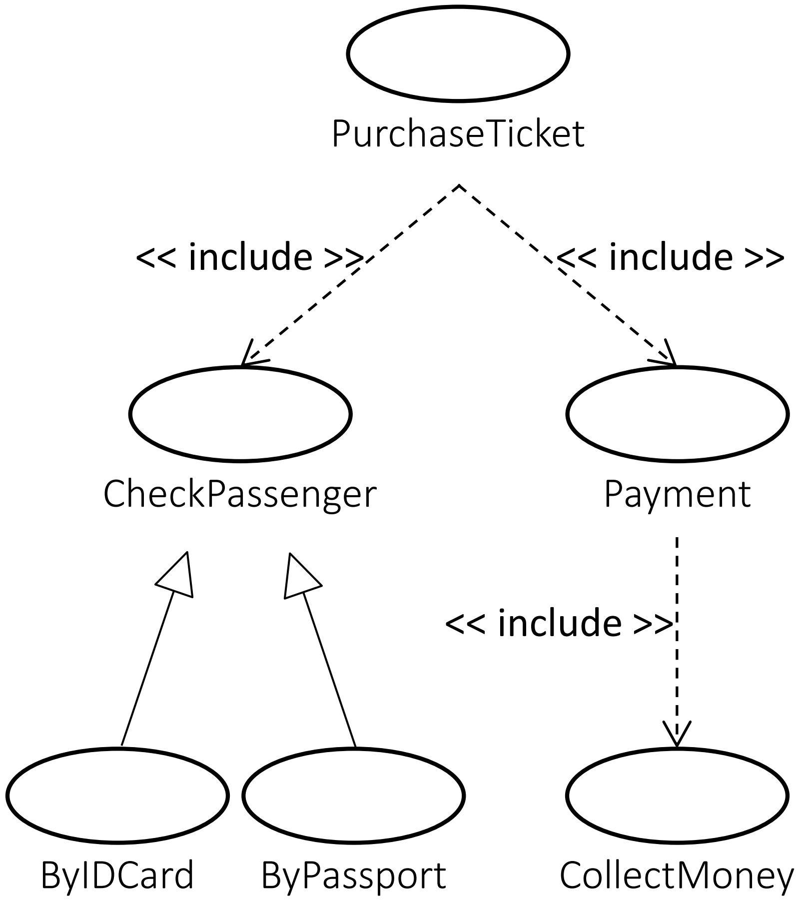
Extend Relationship

- It is the exceptional cases of the model
- The exceptional event flows are factored out of the main event flow for clarity
- The exceptional flows can extend many use cases
- The diagram shows the cases that a passenger cannot purchase tickets



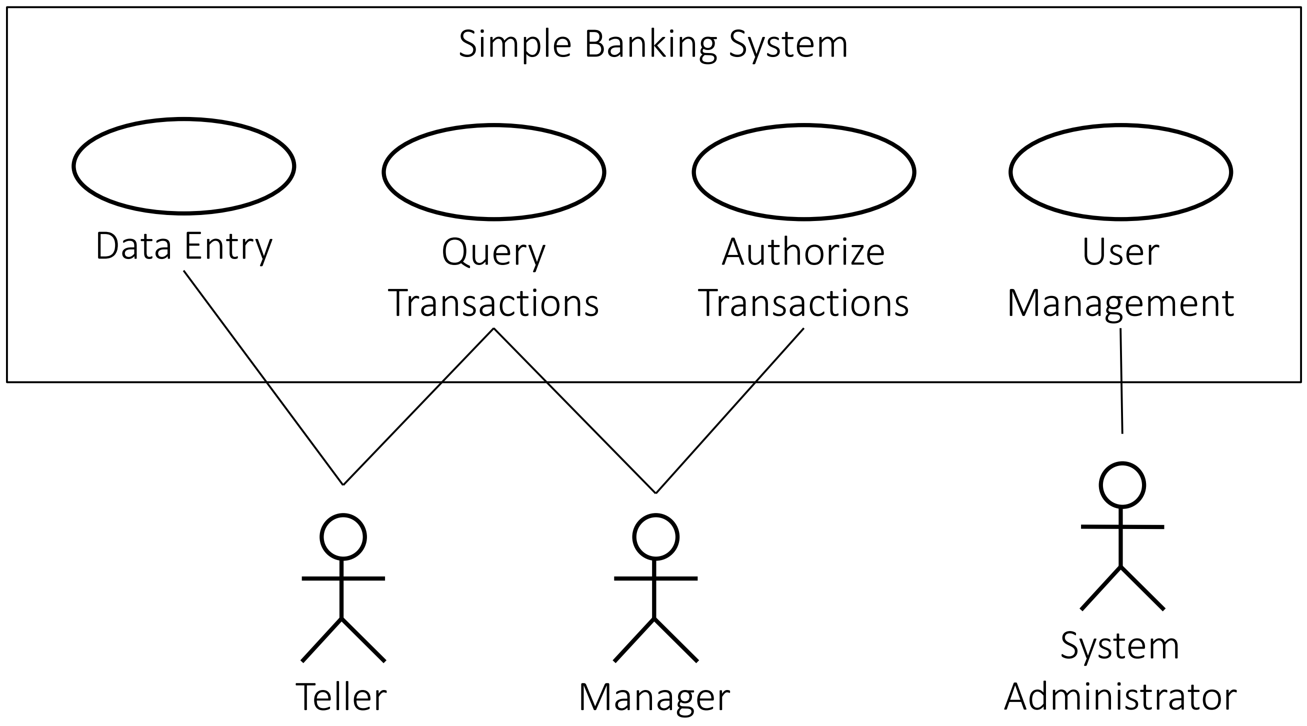
Include Relationship

- A use case can contain the functionality of another use case as part of the whole process
- It breaks down a big module into small steps for reuse
- The *ByIDCard* and *ByPassport* inherits the *CheckPassenger* use case



Simple Banking System

- Actors: Teller, Manager, System Administrator
- Use Cases: Data Entry, Authorize Transaction, User Management

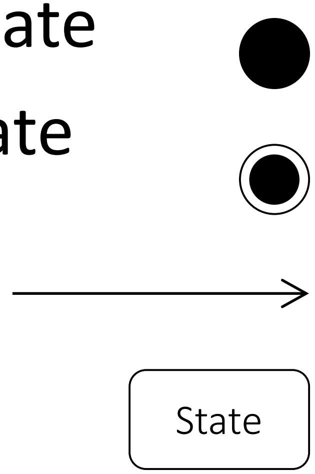


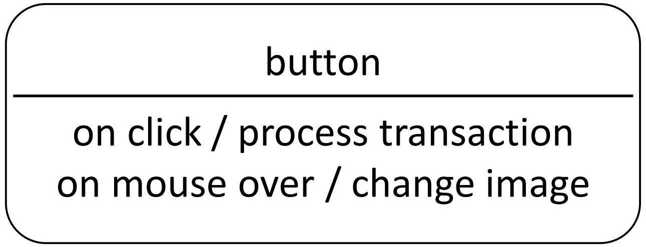
State Machine Diagram

- State Machine Diagram or called state diagram
- Any real time system is expected to be reacted by some kind of internal/external events. These events are responsible for state change of the system
- It is used to visualize the reaction of a system by internal/external factors
- It basically describes the state change of a class, interface, etc.
- It is focus on representing the ***changing state*** of a system over the time or under different conditions

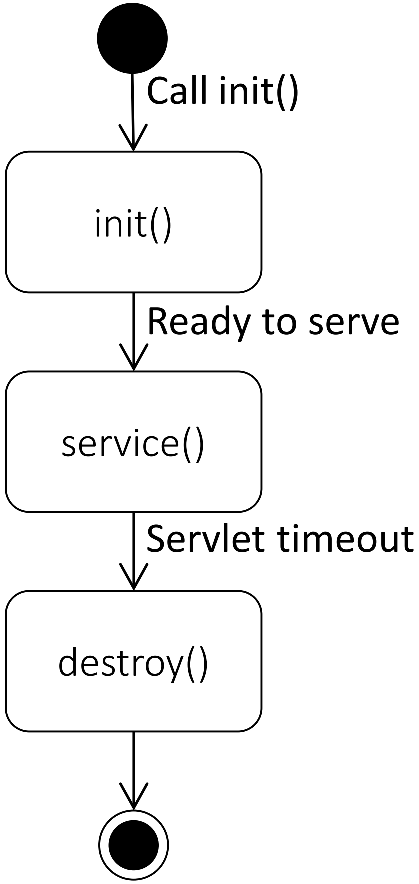
State Machine Diagram Notation

- Start / Initial state
- End / Finish state
- Transition
- State / Entity
- State Action
  - event / result





Java Servlet lifecycle

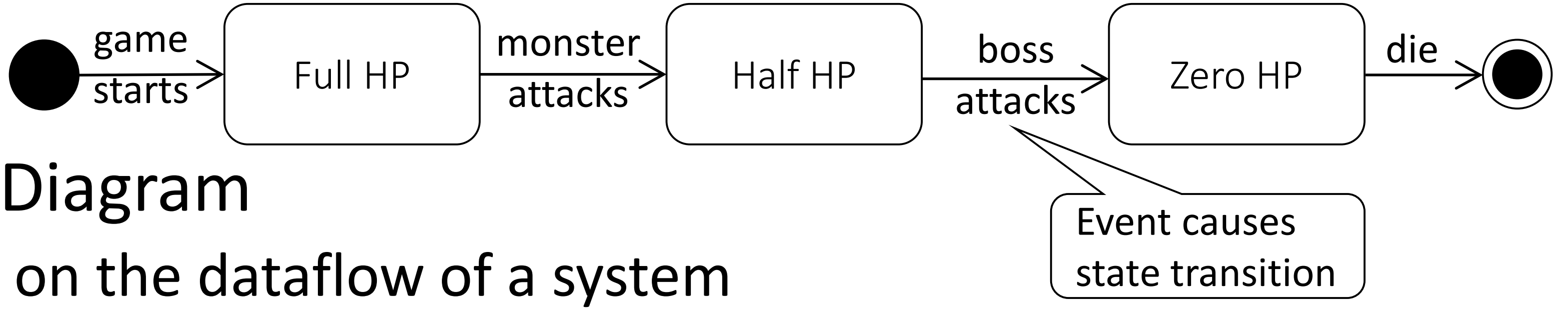


Activity Diagram

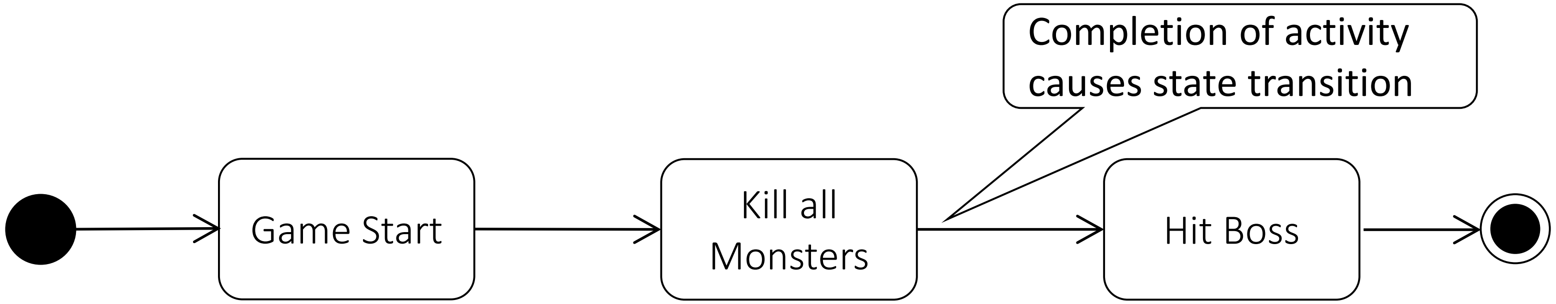
- Activity diagram is a special kind of state diagram
- It describes the flow of control in a system
- Activities can be described as an operation of the system
- It does not show any message flow between activities
- It focus on describing the ***system flow*** from one activity to another
- It is general used as a **Flow Chart**

Activity Diagram vs State Diagram

- State Diagram
  - Focus on the set of attributes of a single abstraction (object, system)



- Activity Diagram
  - Focus on the dataflow of a system

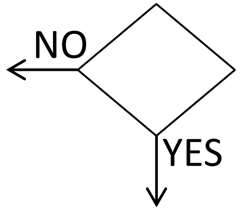


Activity Diagram Notation

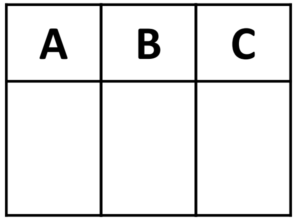
- Fork
  - When an activity splits into two activities

- Join
  - When two activities join together

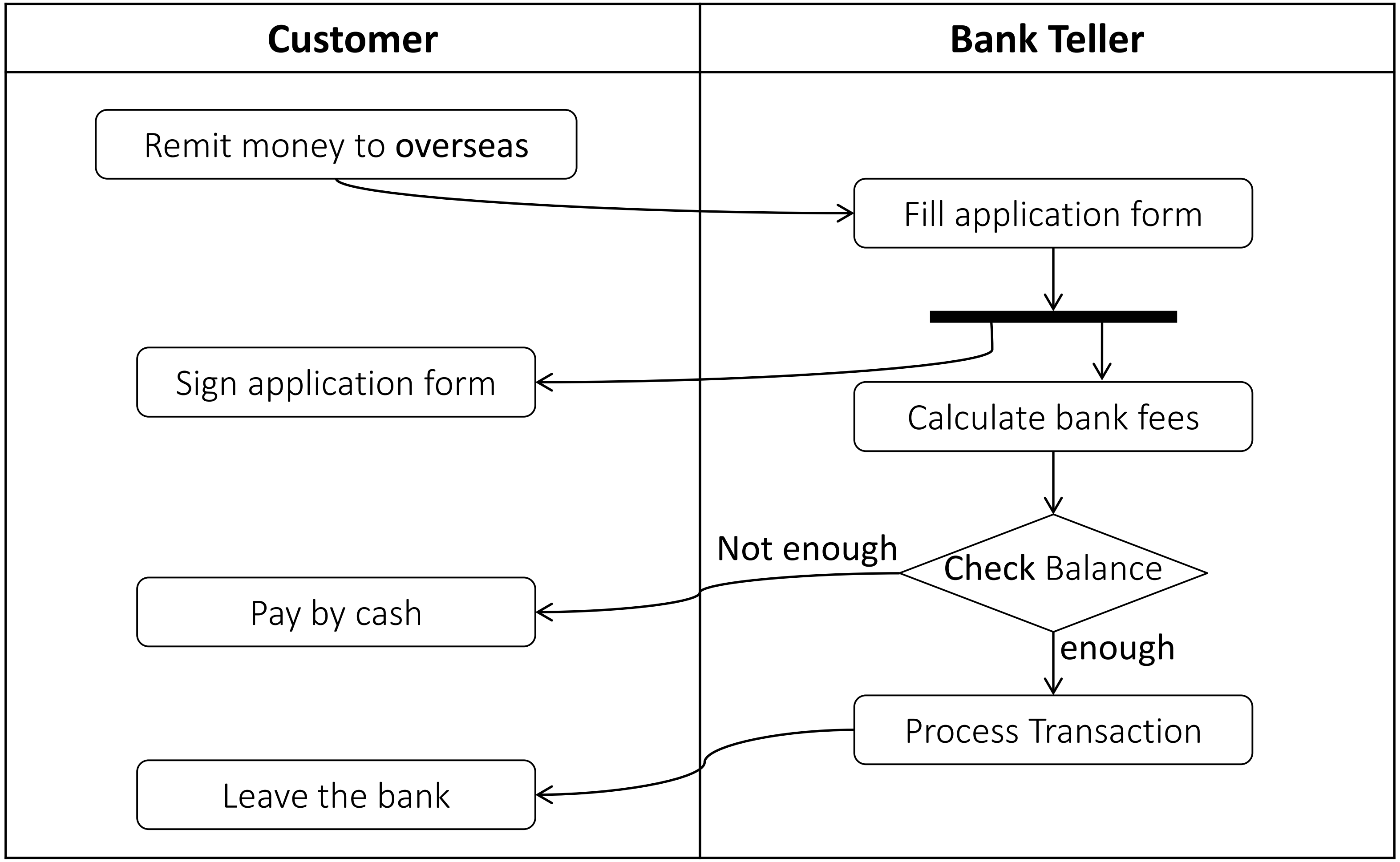
- Decision
  - When there is a decision to make (yes or not, true or false, etc.)



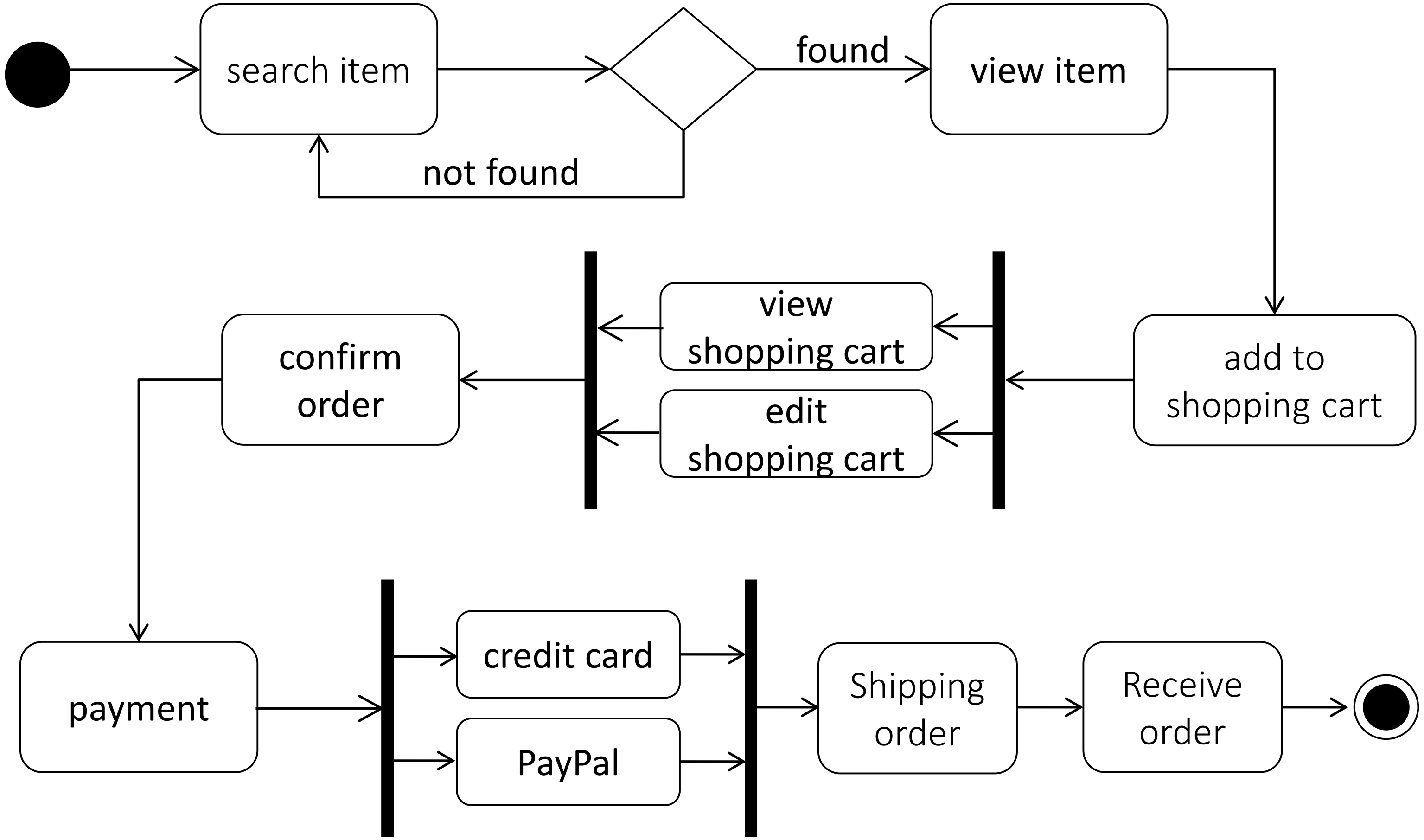
- Swimlanes
  - Reserve each lane for a stakeholder



Activity Diagram Example



Online Shopping

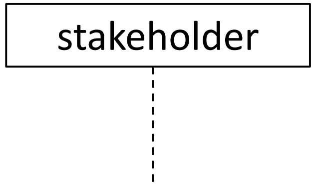


Sequence Diagram

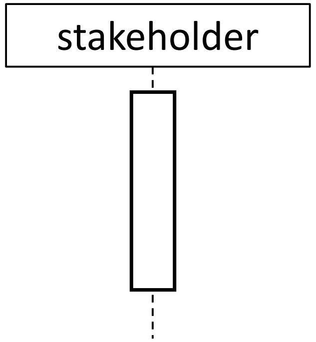
- A sequence diagram is used to present the sequence of messages flowing from one object to another
- Sequence diagram is used to visualize the sequence of calls in a system to perform a specific functionality
- It focuses on the ***interaction*** among the components of a system
- Sequence diagram focuses more on the behaviors of stakeholders, and activity diagram focuses more on the functionalities of each state object

Sequence Diagram Notation

- Lifeline
  - Represents an individual stakeholder in a sequence diagram

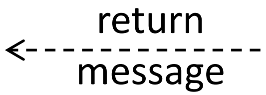


- Activation / method-invocation boxes
  - The long thin boxes on top of the lifeline
  - Indicates processing is being performed by the target object/class to fulfill a message

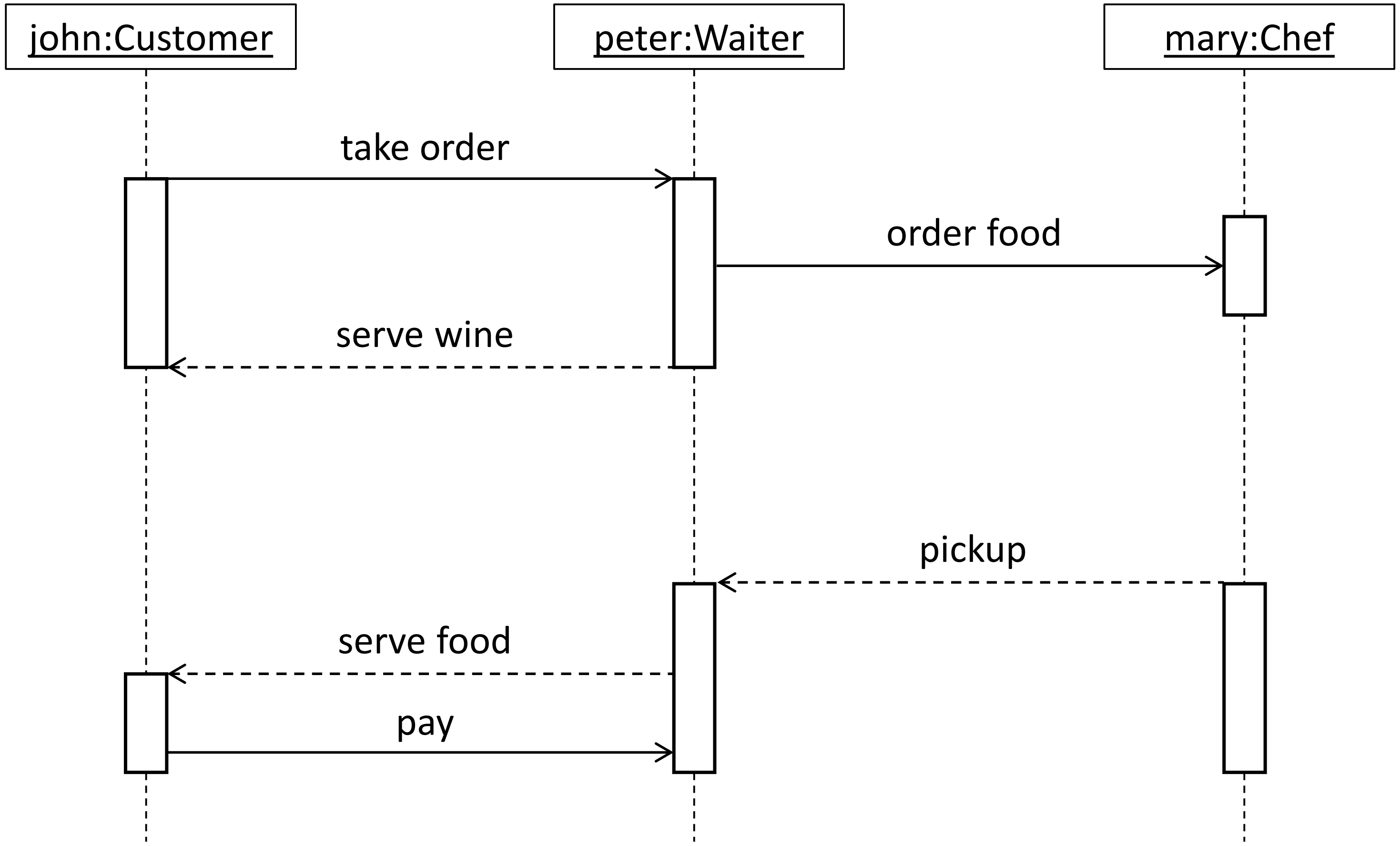


- Send Message

- Return Message



Sequence Diagram Example



Batch Job System

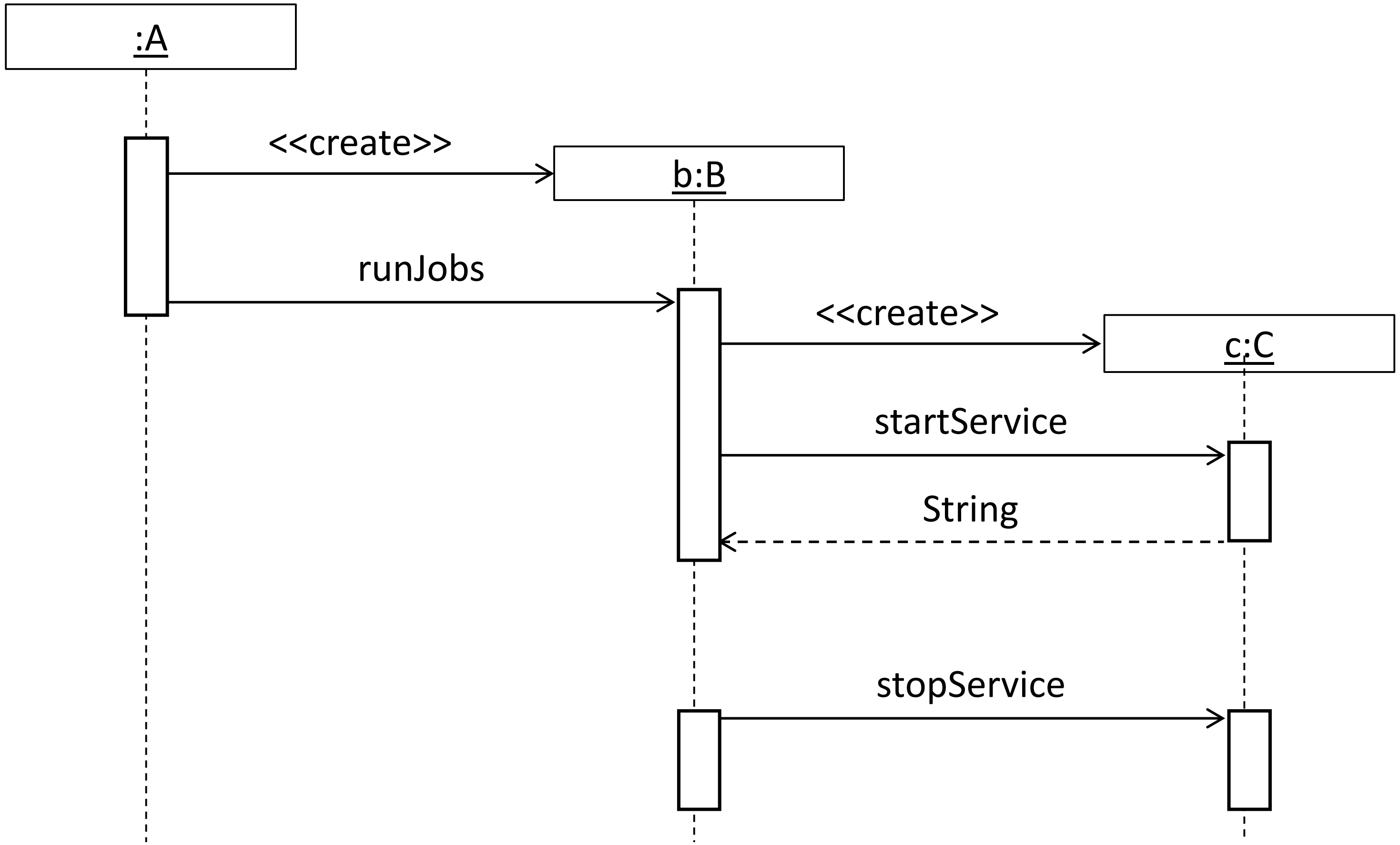
- A batch job system consists of three classes

```
public class A {
  public static void main(String[] args) {
    B b = new B();
    b.runJobs();
  }
}
```

```
public class B {
  public void runJobs() {
    C c = new C();
    System.out.println("Running batch");
    System.out.println(c.startService());
    c.stopService();
  }
}
```

```
public class C {
  public String startService() {
    return "Service running...";
  }
  public void stopService() {
    System.out.println("Service stop!");
  }
}
```

System Sequence Diagram



Summary

- UML provides a wide variety of notations for representing many aspects of software development
  - Powerful but complex
- UML is not a programming language
  - It can be misused to generate unreadable models
  - It can be misunderstood when using too many exotic features
- We concentrate on a few notations
  - Structural Diagram: class, object, package, component, node, and deployment diagrams
  - Functional model: use case diagram
  - Dynamic models: state, activity, and sequence diagrams