SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

CSE 316 – Software Design With UML

Unit – 3 & Unit 4 Presentation Slides

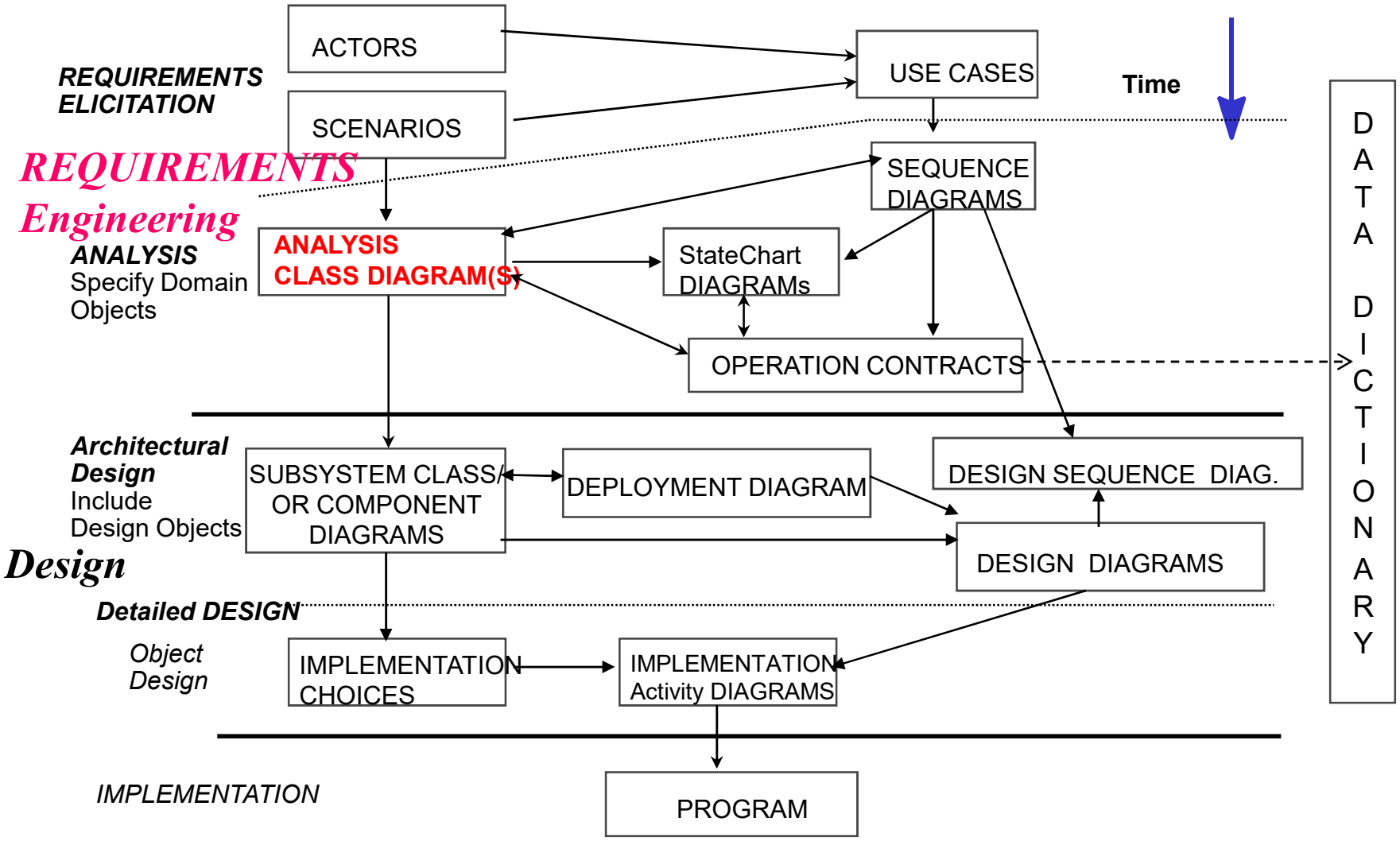
Course Coordinator : Dr. G. Pradeep

Course Content (Unit 3 & Unit 4)

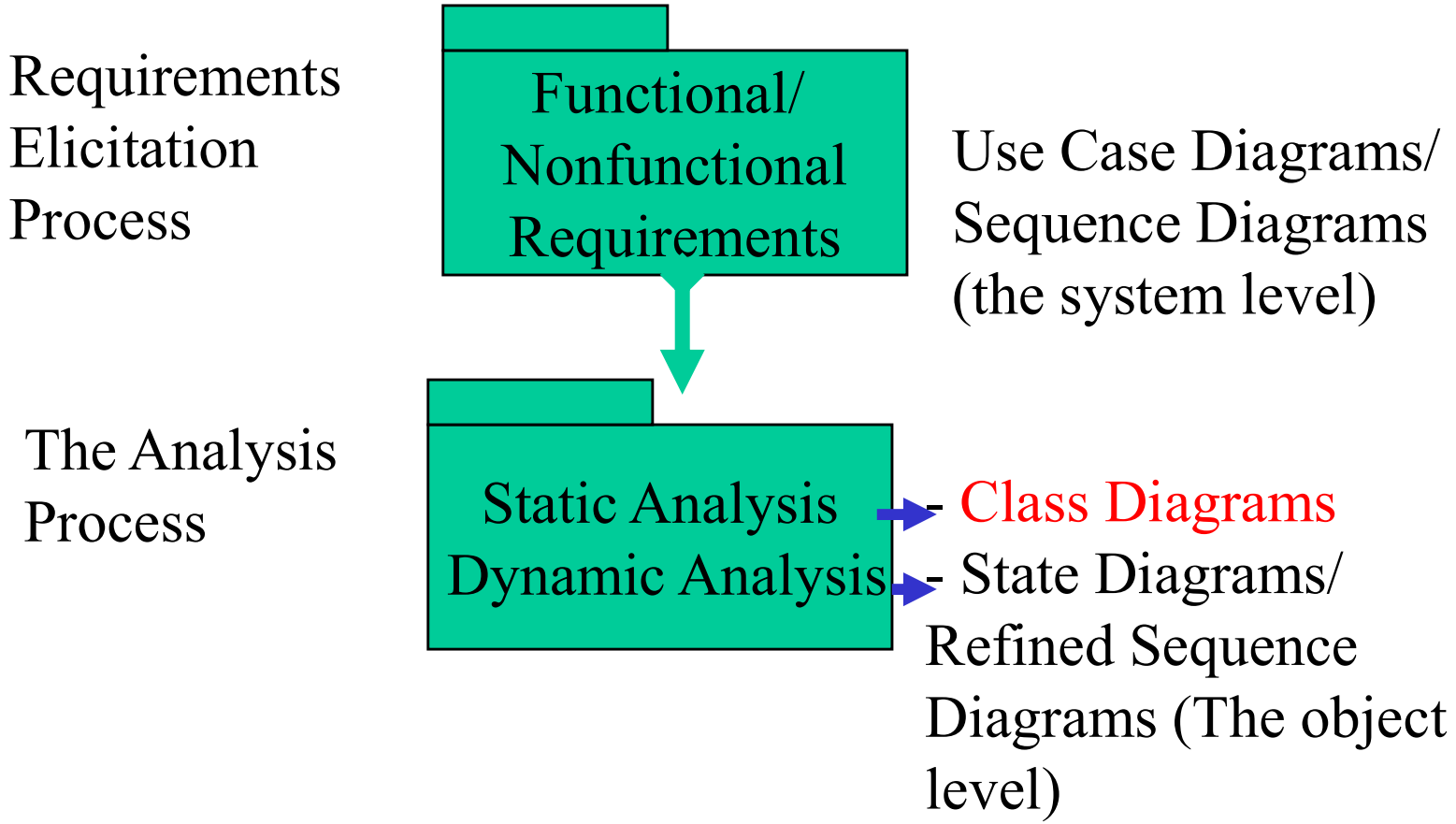
- UML Development
- Structural & Behavioral Diagrams

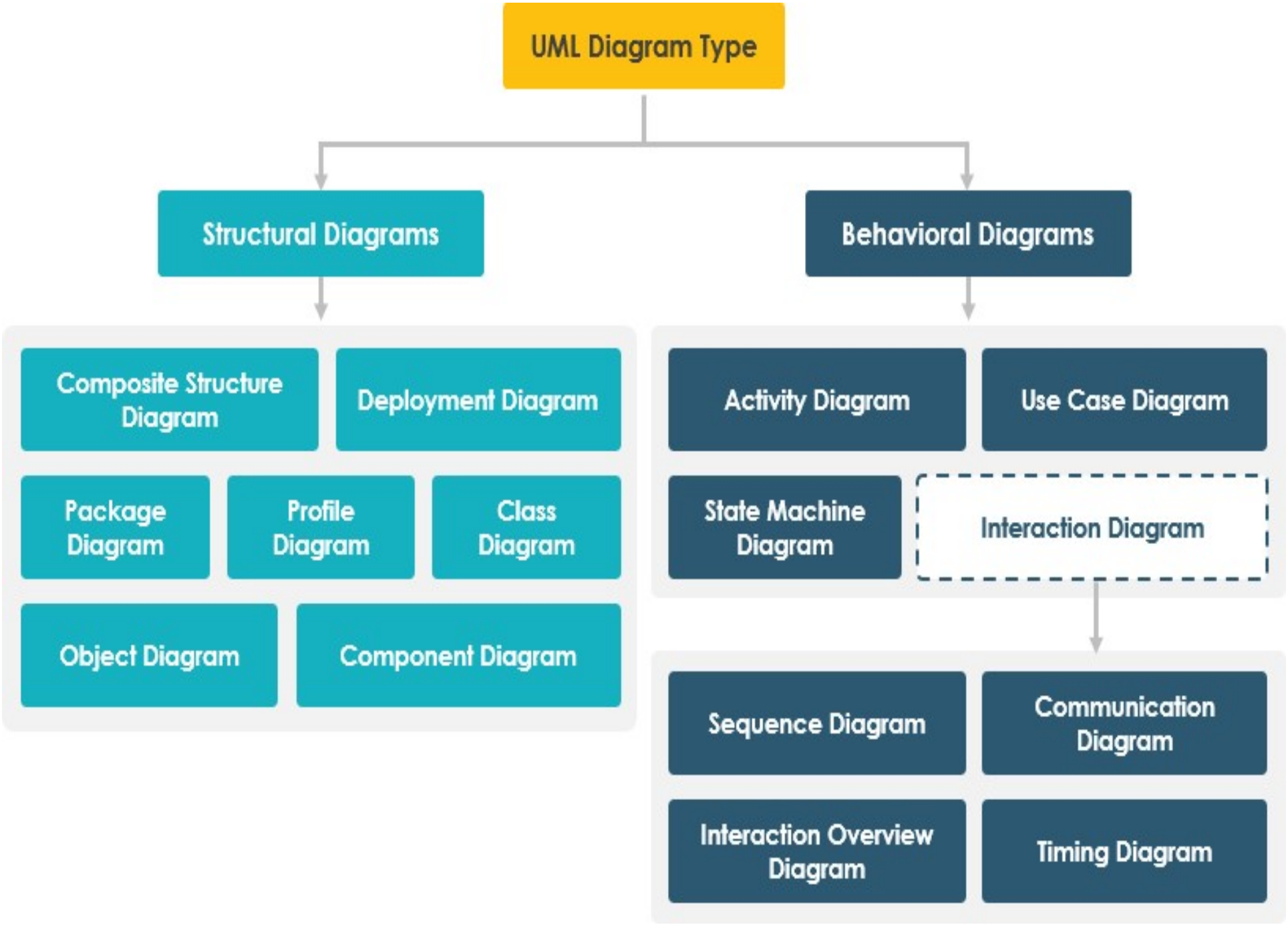
  Class Diagram - Relationships
  Object Diagram
  Package Diagram
  Activity Diagram
  State Diagram
  Component Diagram
  Interface
  Deployment Diagram
  Profile Diagram
  Timing Diagram & Communication Diagram

UML Development - Overview



The Requirements Model and the Analysis Model




UML Diagram Type
Structural Diagrams
Behavioral Diagrams
Composite Structure Diagram
Deployment Diagram
Package Diagram
Profile Diagram
Class Diagram
Object Diagram
Component Diagram
Activity Diagram
Use Case Diagram
State Machine Diagram
Interaction Diagram
Sequence Diagram
Communication Diagram
Interaction Overview Diagram
Timing Diagram

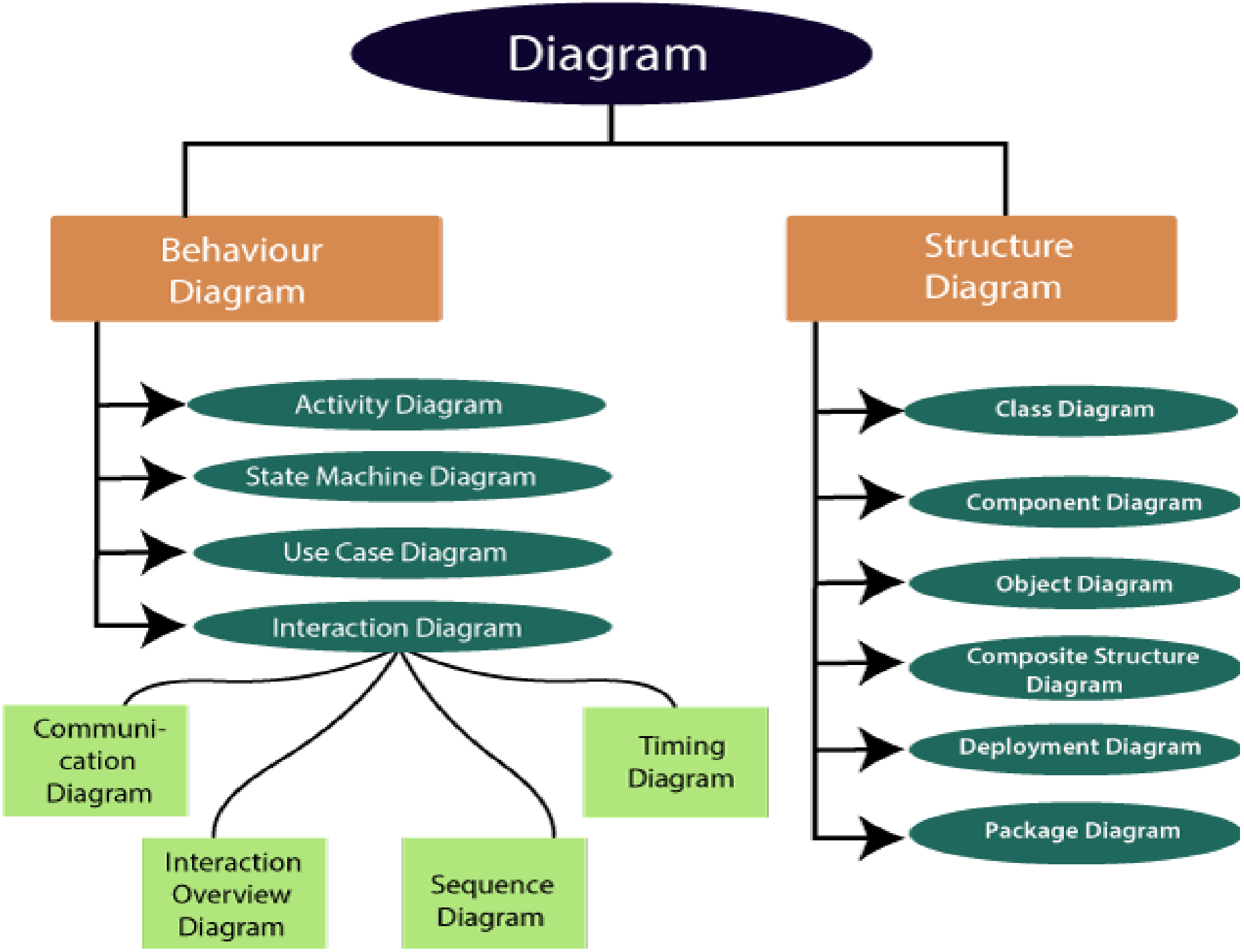


Diagram
Behaviour Diagram
Structure Diagram
Activity Diagram
State Machine Diagram
Use Case Diagram
Interaction Diagram
Communi-cation Diagram
Interaction Overview Diagram
Sequence Diagram
Timing Diagram
Class Diagram
Component Diagram
Object Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram

Structural Diagram - Class Diagram

- The class diagram depicts **a static view of an application**.
- A **class diagram** is used to **visualize, describe, document** various different aspects of the system, and also construct executable software code.

A **Class** is defined as

- Real world **entity type about which information** is stored
- Represents a collection of identical objects (instances)–Described by means of attributes (data items)
- Has operations to access data maintained by objects
- Each object instance can be uniquely identified

Relationships between classes

–Associations

–Composition / Aggregation

–Generalization / Specialization

Purpose of Class Diagrams

- The main purpose of class diagrams is to build a **static view of an application**.
- It is the only diagram that is widely used for **construction**, and it can be **mapped with object-oriented languages.**
- It is one of the most popular UML diagrams.
- It **analyses and designs a static view** of an application.
- It describes the major **responsibilities** of a system.
- It is a base for **component and deployment** diagrams.
- It incorporates **forward and reverse** engineering.

Benefits of Class Diagrams

- It can represent the **object model** for complex systems.
- It reduces the maintenance time by providing an overview of how an application is structured before coding.
- It provides a general schematic of an application for better understanding.
- It represents a detailed chart by highlighting the desired code, which is to be programmed.
- It is helpful for the stakeholders and the developers.

- **Class diagrams are useful in many stages of system design.**
- In the **analysis stage**, a class diagram can help you **to understand the requirements** of your problem domain and **to identify its components.**
- In an object-oriented software project, the class diagrams which are created during the early stages of the project contain classes that often translate into **actual software classes and objects when you write code**.
- Later, you can refine your earlier **analysis and conceptual models** into class diagrams that show the **specific parts of your system**, **user interfaces, logical implementations, and so on**.
- Class diagrams then become a snapshot that describes exactly how your system works, the relationships between system components at many levels, and how you plan to implement those components.

Create class diagrams to perform the following functions

- Capture and **define the structure of classes** and other classifiers
- **Define relationships** between classes and classifiers
- Illustrate the **structure of a model by using attributes, operations, and signals**
- Show the common **classifier roles and responsibilities** that **define the behavior of the system**
- Show the **implementation classes in a package**
- Show the **structure and behavior of one or more classes**
- Show an **inheritance hierarchy** among classes and classifiers
- Show the workers and entities as business object models

- UML Development, the Requirements Model and the Analysis model
- The Static Analysis Model – The Analysis Process
  - The Conceptual Level
    - Identifying the Classes of Objects
  - The Analysis Level
    - Identifying Class relationships, class attributes, and class operations

The Static Model

- Defines the static structure of the logical model
- Represents classes, class hierarchies using packages, classes, and their relationships,
- Evolve in three phases
  - Conceptual phase,
  - Analysis phase, and
  - Design phase.

The conceptual Level

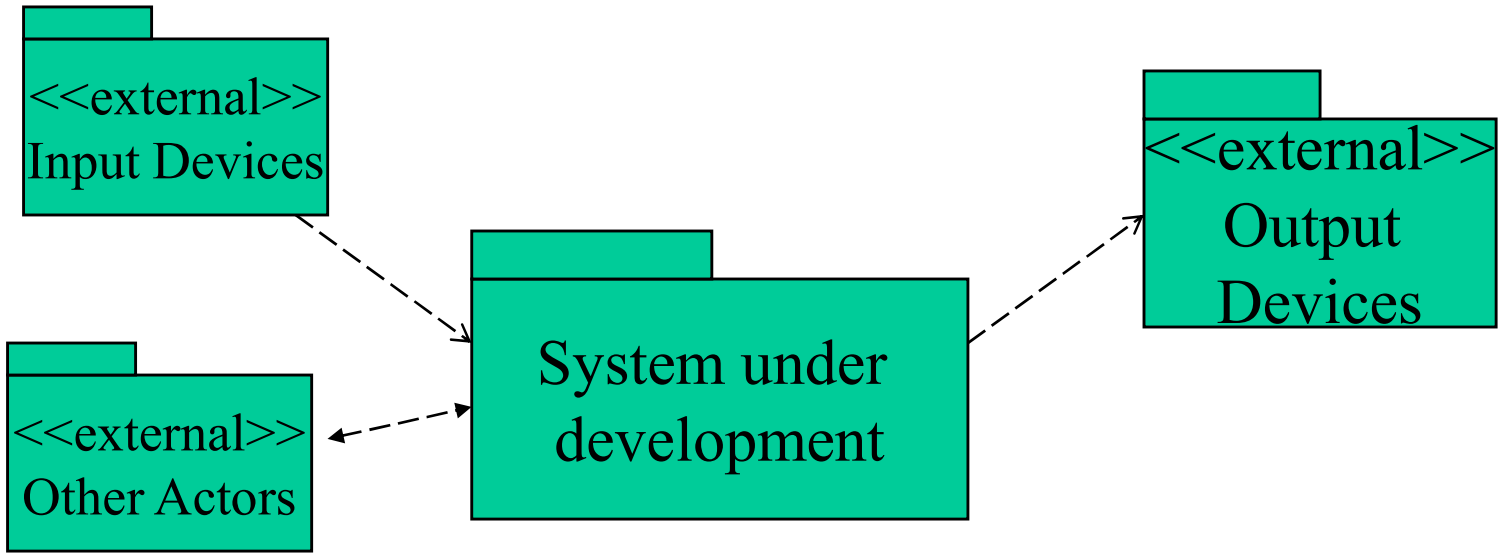
- At the conceptual phase, classes are defined based on the classes found in the problem domain descriptions
- A context class diagram is defined first, where the system under development is represented by one package, and other classes represent external classes representing the actors

The conceptual Level

- The system package is defined by a diagram representing the main classes and interface classes to external classes
- Each subsystem is represented by a class diagram defining the classes of objects needed to realize the use cases defined in the use case diagrams

Context Class Diagram

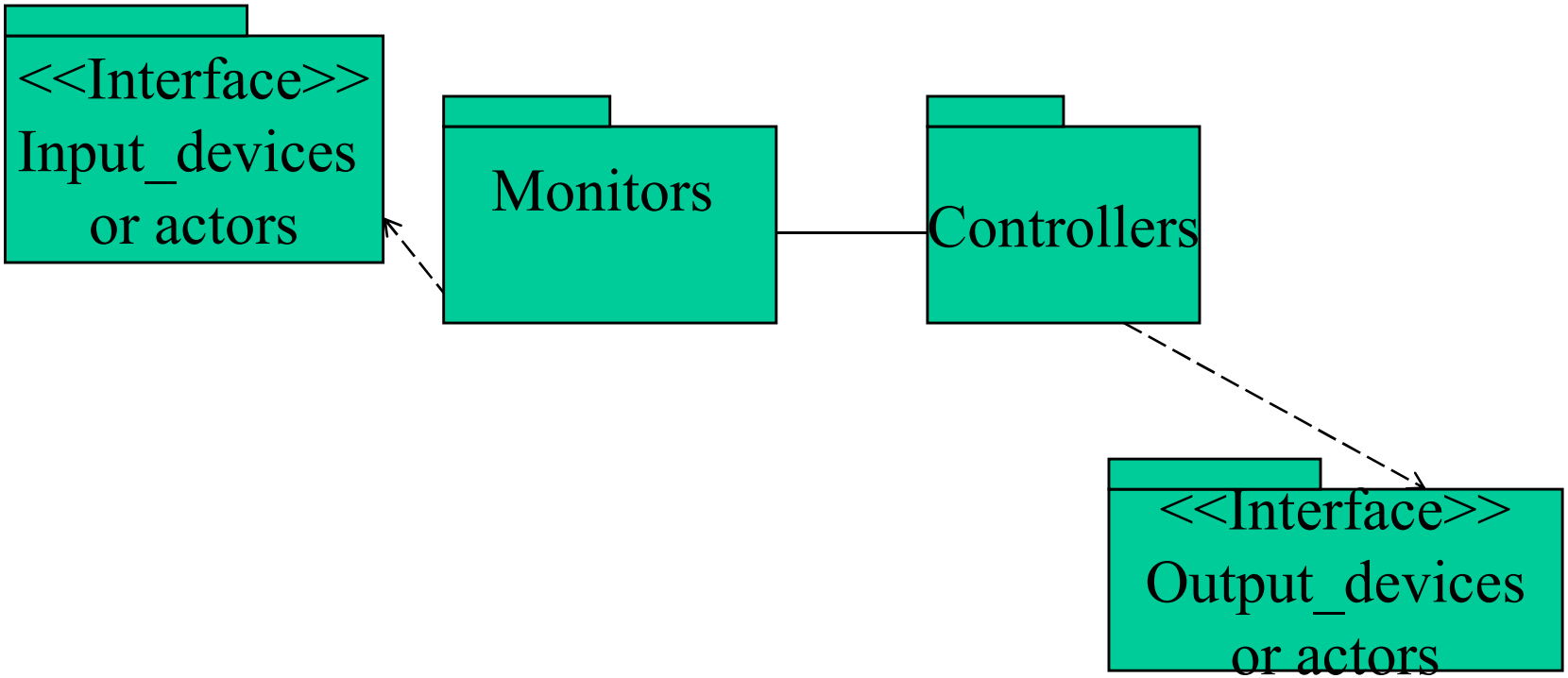
Defines the Boundary of the system



Specify the classes of the external input/output devices and other actors (users, other systems, etc.) and the system classes

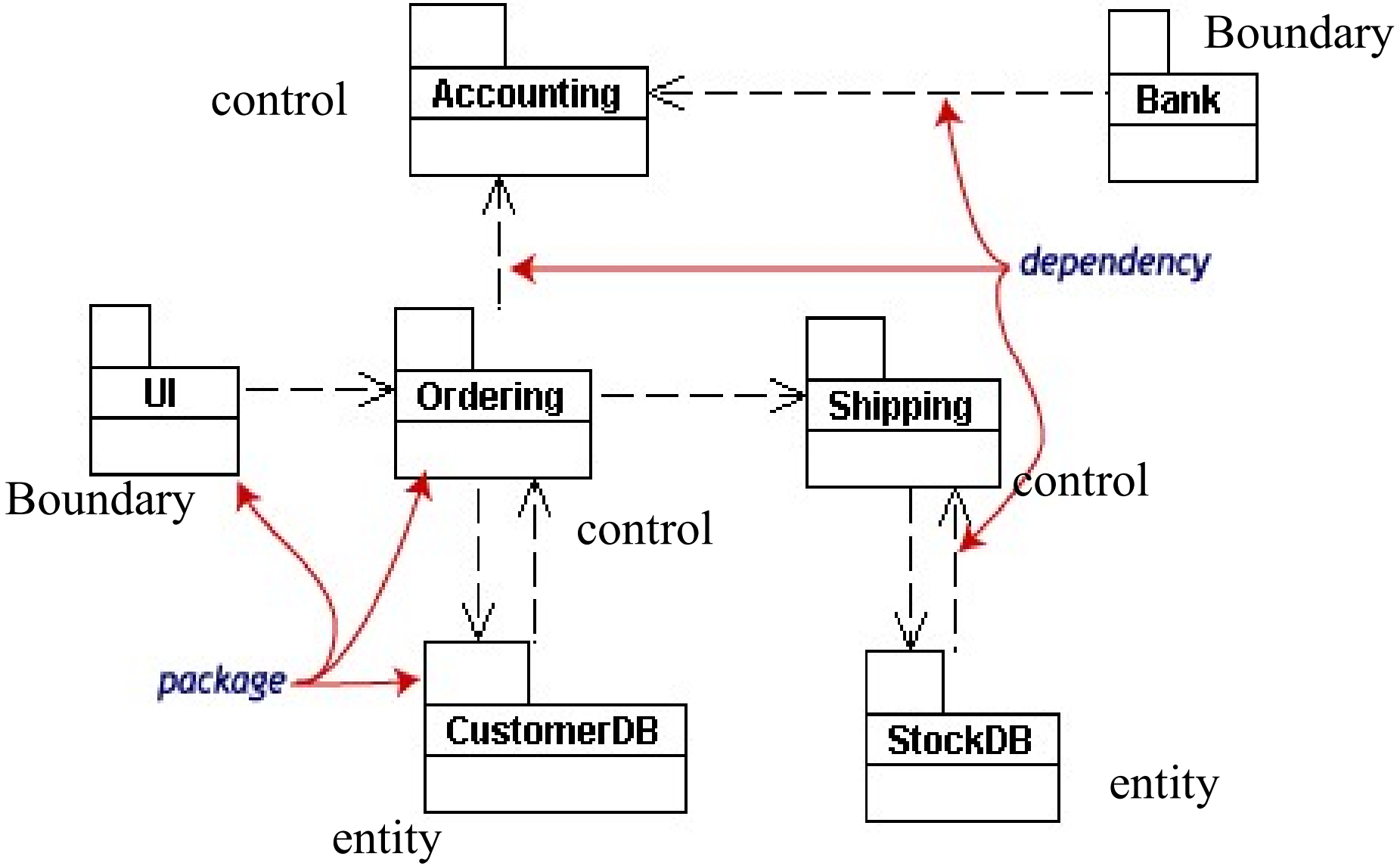
The conceptual Level

Identify the system classes as Interface objects, Monitors objects, controllers objects



Example of System packages of E-Commerce Application

3 types of classes: Boundary, entity, and control



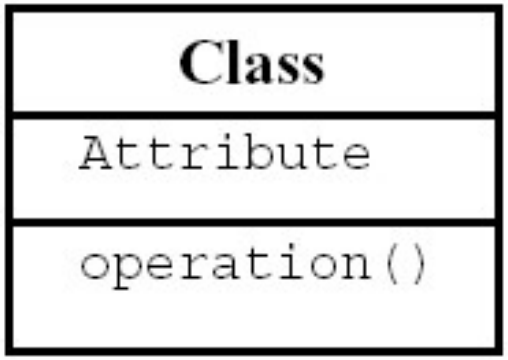
The Analysis Level

- At the analysis level, *class* diagrams are refined by **adding relationships between classes, *attributes and methods*** depicting how objects of the static view are used to realize use cases in sequence diagrams

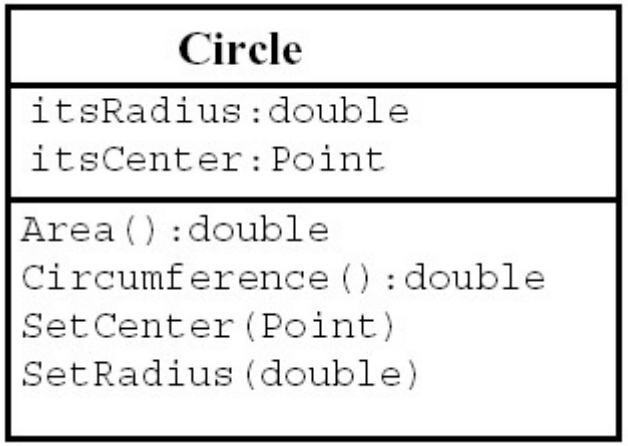
- Emphasis is placed on **distributing behavior, resolving software interfaces and identifying generalization relationships** that will maximize the effectiveness of the object model

The Class Diagram Notation

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



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



The notation that precedes the **attribute, or operation name, indicates the visibility of the element:**

if the + symbol is used, the attribute, or operation, has a **public** level of visibility;

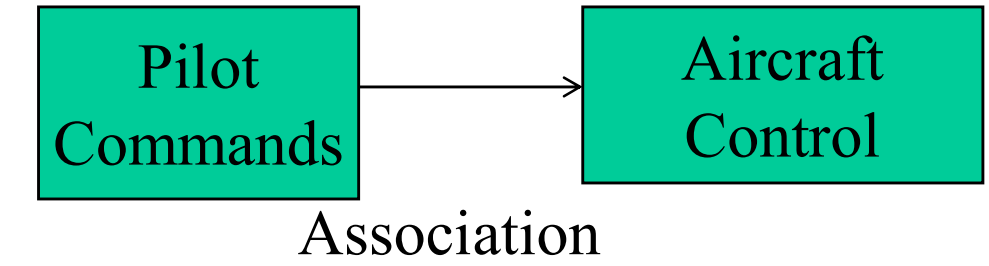
if a **-** symbol is used, the attribute, or operation, is **private**.

In addition the # symbol allows an operation, or attribute, to be defined as **protected**,

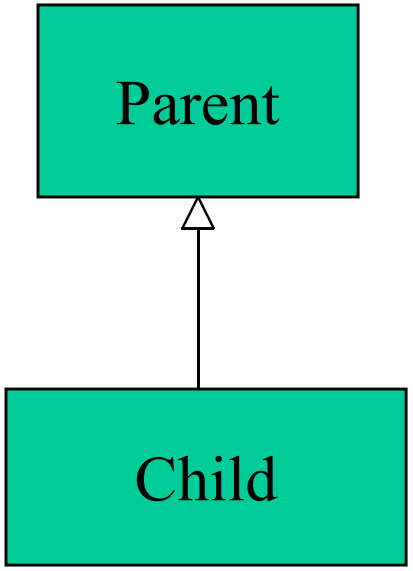
while the ~ symbol indicates **package** visibility.

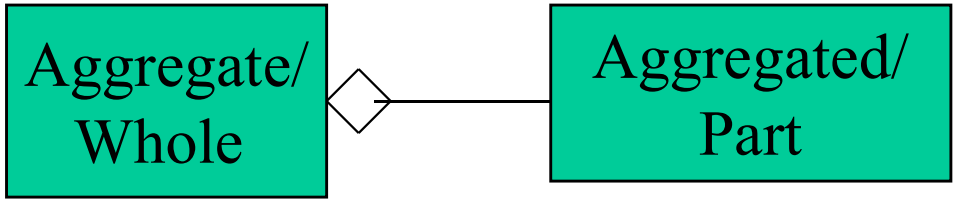
•

Identify Class relationships



Association





Aggregation (hollow diamond)/
Composition (solid diamond)

Generalization

Associations Between Classes

- Associations between classes are generally shown as solid lines connecting the associated classes.
- A notable exceptions to the solid line rule are the use of dashed lines to depict *dependencies as special case of association,*

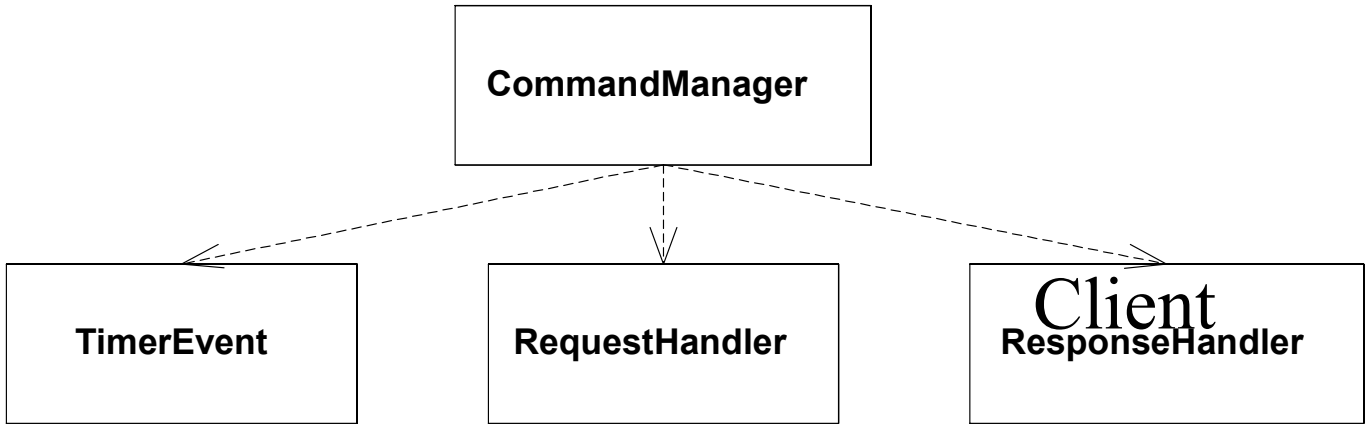
Associations

- Association is
  - –static, structural relationship between classes
    - –E.g, Employee works in Department
  - – Multiplicity of Associations
    - Specifies how many instances of one class may relate to a single association, Company has President
      - – 1-to-many association, Bank manages Account
      - – Optional association (0, 1, or many) –Customer owns Credit Card instance of another class
      - – 1-to-1
      - – Many-to-Many association –Course has Student, and

        Student attends Course

Dependency: A Special Case of Association

Dependency: A dependency is a semantic relationship between two or more classes where a change in one class cause changes in another class. It forms a weaker relationship.

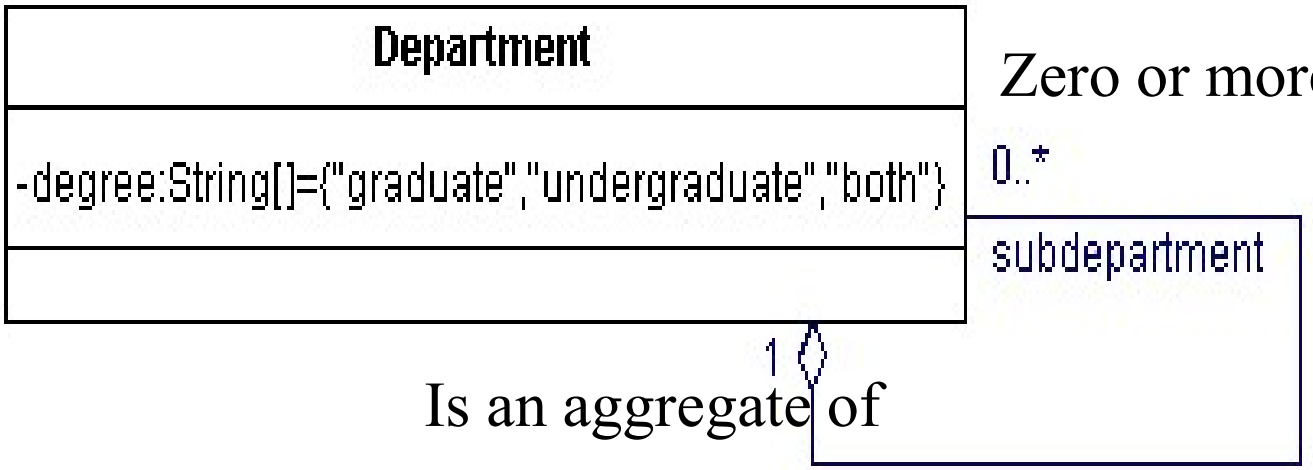


Command Manager (Client class) depends on services provided by the other three server classes

Aggregation Relation

- An aggregation is a subset of association, which represents has a relationship.
- It is more specific then association.
- It defines a part-whole or part-of relationship.
- In this kind of relationship, the **child class can exist independently of its parent class.**
- A hollow diamond is attached to the end of the path to indicate aggregation. The diamond is attached to the class that is the aggregate.
- Aggregation provides a definitive conceptual whole part relationship

Aggregation Example



Aggregation is a transitive relation:

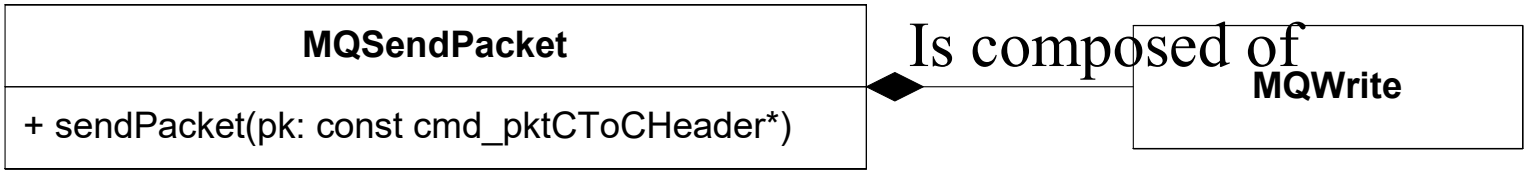
if A is a part of B and B is a part of C then A is also a part of C

Aggregation is an anti symmetric relation:

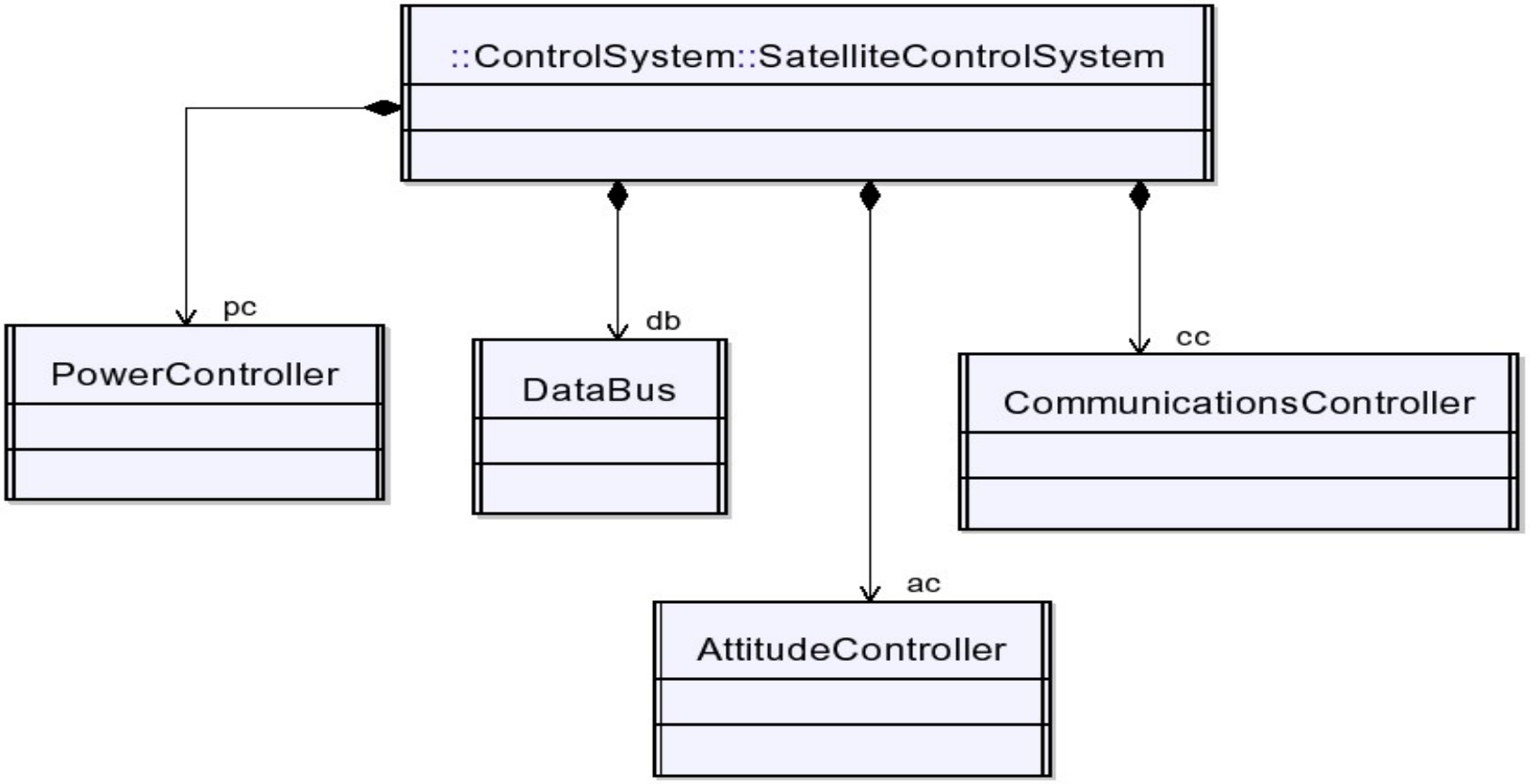
If A is a part of B then B is not a part of A.

Composition: A Special Case of Aggregation

- The composition is a subset of aggregation.
- It describes the dependency between the parent and its child, which means if one part is deleted, then the other part also gets discarded.
- It represents a whole-part relationship.
- Composition is shown as a solid filled diamond, with the diamond attached to the class that is the composite.
  It requires coincident lifetime of the part with the whole and singular ownership; i.e. the part is owned by only one whole and is deleted when the whole is deleted



Composition example

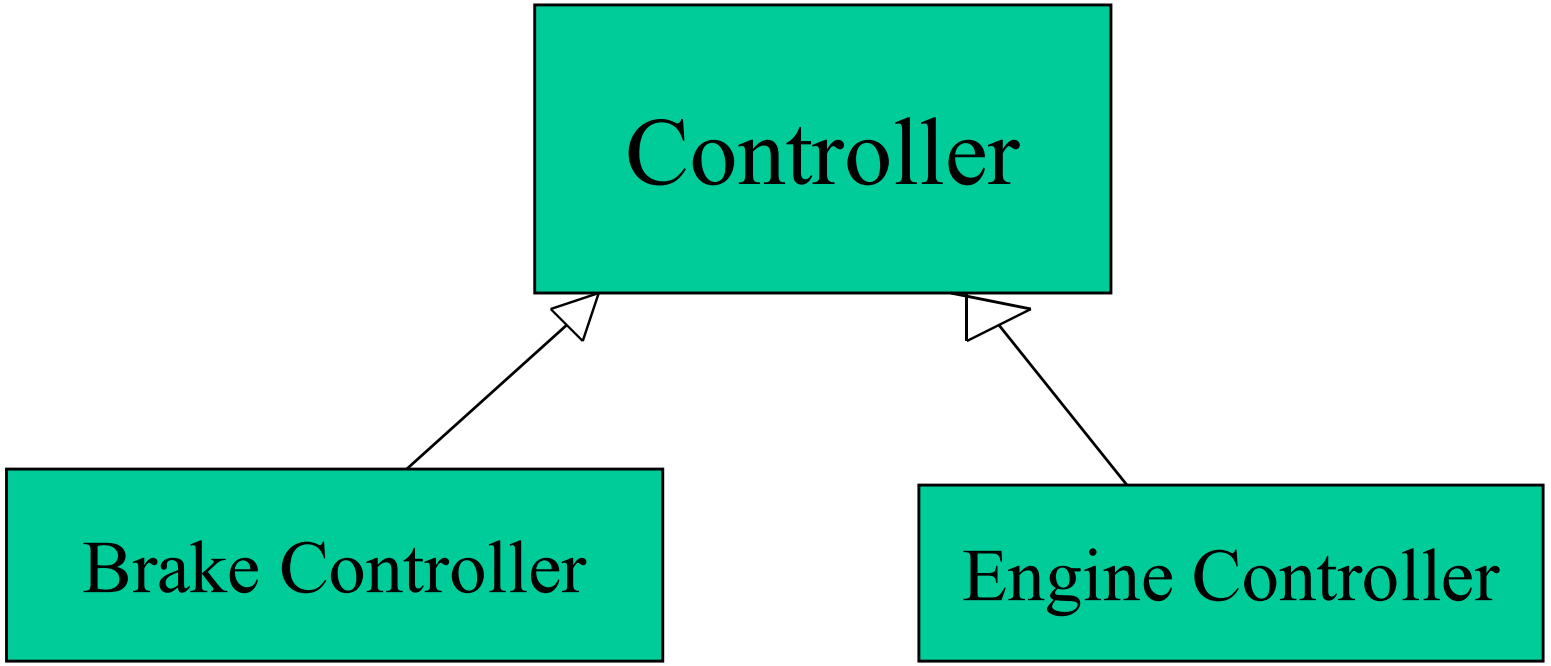


Generalization/Specialization Relation

- A generalization is a relationship between a parent class (superclass) and a child class (subclass). In this, the child class is inherited from the parent class.
- Generalization is shown as a solid-line arrow from the child (the more specific element) to the parent (the more general element)
- Should be used to define class hierarchies based on abstraction

Generalization/Specialization Relation

- Controllers and Monitors are examples of abstract classes



Use of generalization

Used for three purposes:

- Support of polymorphism:
  - polymorphism increases the flexibility of software.
  - Adding a new subclass and automatically inheriting superclass behavior.
- Structuring the description of objects:
  - Forming a taxonomy (classification), organizing objects according to their similarities. It is much more profound than modeling each class individually and in isolation of other similar classes.
- Enabling code reuse:
  - Reuse is more productive than repeatedly writing code from scratch.

Multiplicity of Relationships

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

Aggregation versus Association

- Aggregation is a special form of association, not an independent concept.
- Aggregation adds semantic connotations:
  - If two objects are tightly bound by a **part-whole relation** it is an **aggregation.**
  - If the two objects are usually considered as **independent,** even though they **may often be linked**, it is an **association.**
- Discovering aggregation
  - Would you use the phrase **part of** ?
  - Do some **operations on the whole automatically apply to its parts**?
  - Do some attributes values propagates from the whole to all or some parts?
  - Is there an asymmetry to the association, where one class is subordinate to the other?

Aggregation versus Composition

- **Composition** is a form of aggregation with additional constraints:
  - A constituent part can belong to **at most one** assembly (whole).
    - it has a coincident lifetime with the assembly.
    - Deletion of an assembly object triggers automatically a deletion of all constituent objects via composition.
  - Composition implies ownership of the parts by the whole.
    - Parts cannot be shared by different wholes.

Aggregation versus Composition

Composition is a strong form of aggregation where:

- Deletion of a “whole” triggers automatically deletion of all its “parts”. **(Lifetime)**

- “Parts” are included in at most one “whole” at a time. **(Sharing)**

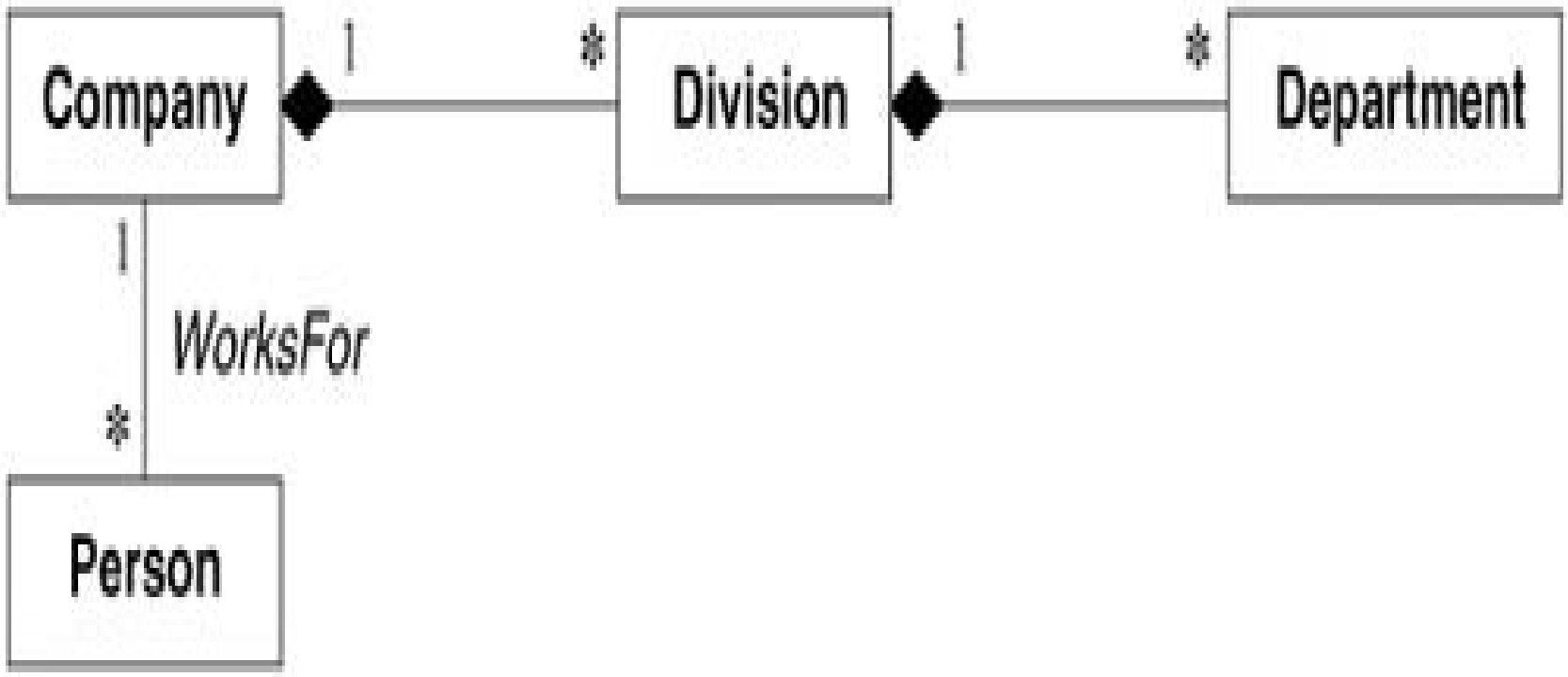
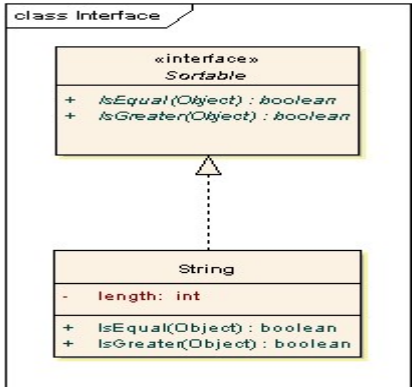


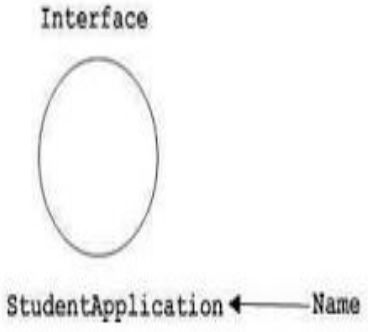
Figure 4.10 Composition. With composition a constituent part belongs to at most one assembly and has a coincident lifetime with the assembly.

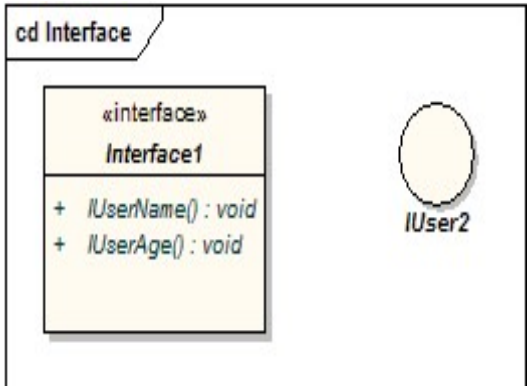
Interfaces

interfaces are model elements that define **sets of operations** that other model elements such as classes, or components must implement.

- An interface is a **specification of behavior** that implementers agree to meet; it is a contract.
- By realizing an interface, classes are guaranteed to support a required behavior, which allows the system to treat non-related elements in the same way – that is, through the common interface.







- Interfaces may be drawn in a similar style to a class, with operations specified, as shown below. They may also be drawn as a circle with no explicit operations detailed.
- When drawn as a circle, realization links to the circle form of notation are drawn without target arrows.

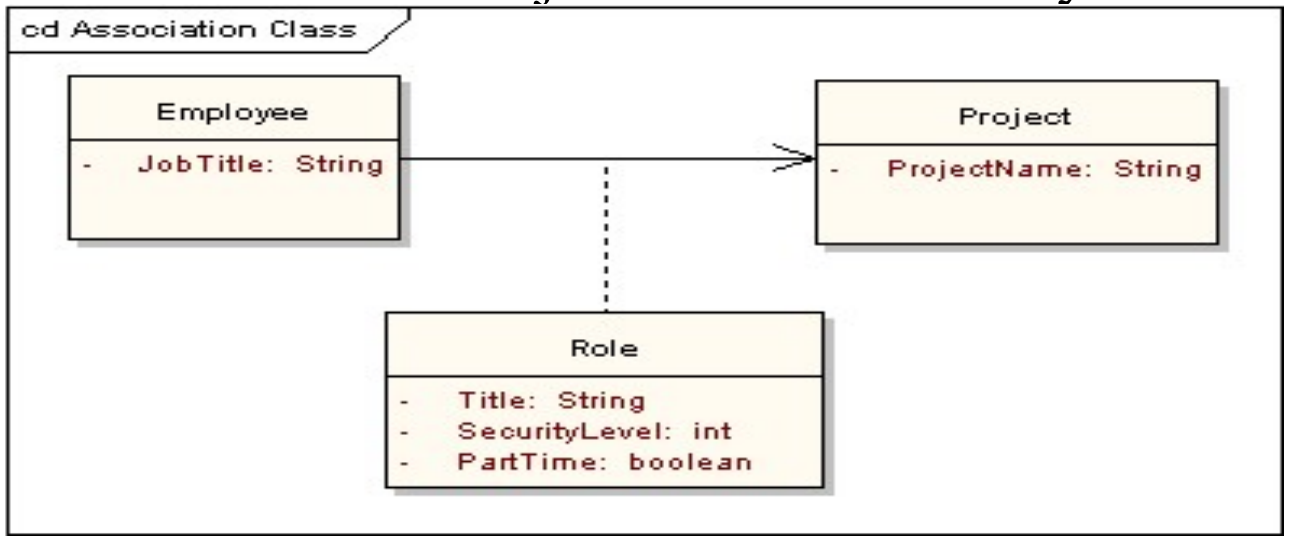
Association Classes

An association class is a construct that allows an association connection to have operations and attributes.

The following example shows that there is more to allocating an employee to a project than making a simple association link between the two classes:

the **role the employee takes up on the project is a complex entity** in its own right and contains **detail that does not belong in the employee or project class.**

For example, an employee may be working on several projects at the same time and have different job titles and security levels on each.



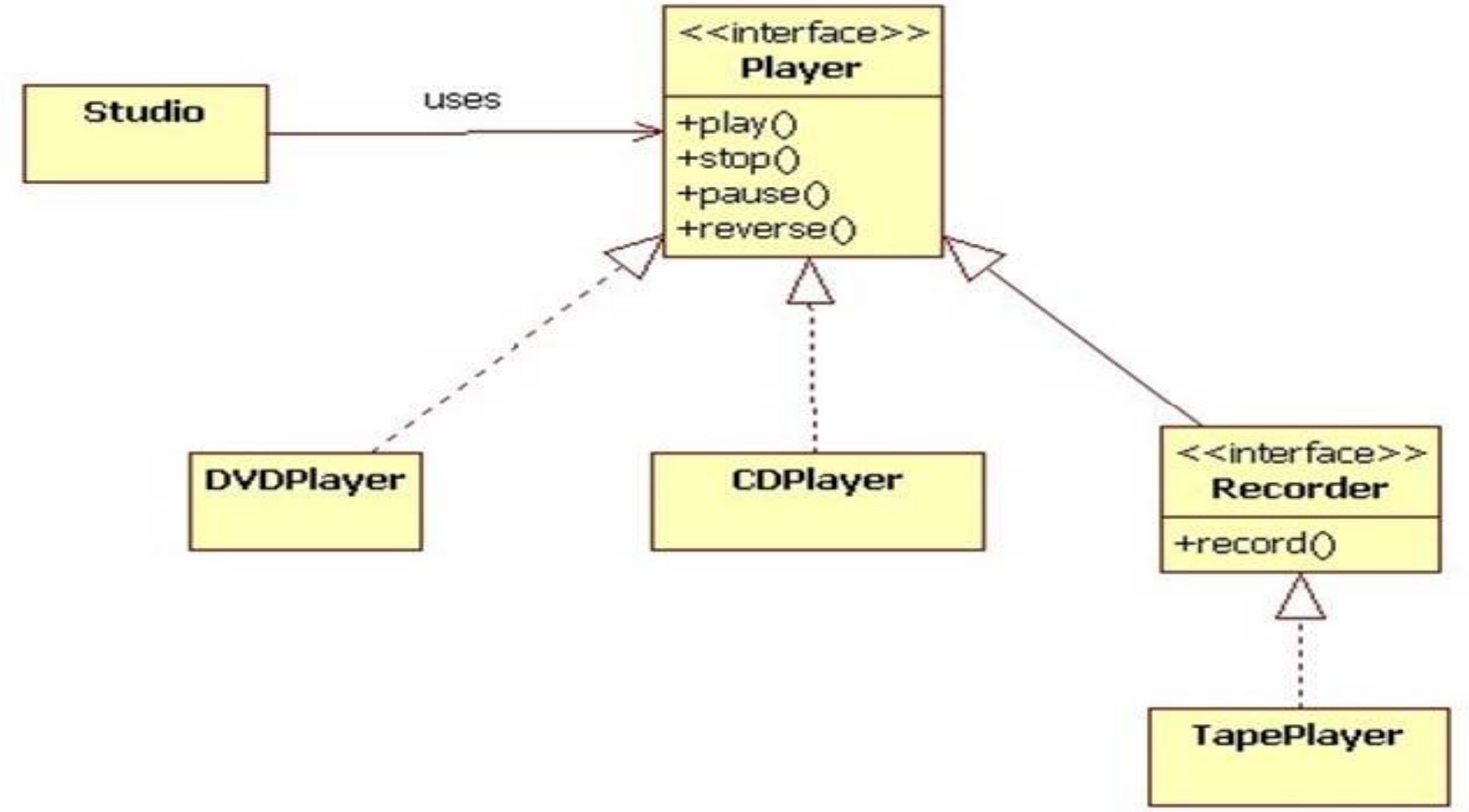
Interface Types

Provided interfaces: these interfaces describe the services that instances of a classifier (supplier) offer to their clients

Required interfaces: these interfaces specify the services that a classifier needs to perform its functions and to fulfill its own obligations to its clients

Ball and socket notation, in which the implementation dependency from a classifier to the provided interface is displayed as a circle (ball) and the usage dependency from a classifier to the required interface is displayed as a half-circle (socket).
This notation is also called the external view.
Provided interface (circle shape)
Required interface (socket shape)

UML notation for two interfaces and three implementing classes



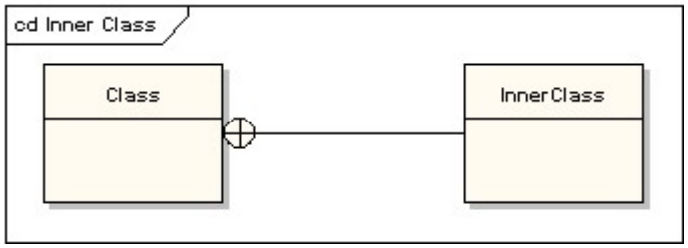
Pseudo Code Representation using Java

```
interface Player {
   void play();
   void stop();
   void pause();
   void reverse();
}
interface Recorder extends Player {
   void record();
}
class TapePlayer implements Recorder {
   public void play() { ... }
   public void stop() { ... }
   public void pause() { ... }
   public void reverse() { ... }
   public void record() { ... }
}
```

Nesting / Nested

A nesting is connector that shows the source element is nested within the target element.

Diagram shows the definition of an inner class, although in EA it is more usual to show them by their position in the project view hierarchy.



Tables

Although not a part of the base UML, a table is an example of what can be done with stereotypes.

It is drawn with a small table icon in the upper right corner. Table attributes are stereotyped «column».

Most tables will have a primary key, being one or more fields that form a unique combination used to access the table, plus a primary key operation which is stereotyped «PK».

Some tables will have one or more foreign keys, being one or more fields that together map onto a primary key in a related table, plus a foreign key operation which is stereotyped «FK».

Modeling the Static Aspects - Class Diagrams, Object Diagrams, and ER Diagrams

Class diagrams, object diagrams, and ER diagrams are all used to model the **static aspects** of an object-oriented system.
Each type of diagram has its own specific use case and can be used at different stages of the software development process.

- Class diagrams are used in the design phase of the software development process.
- Object diagrams are used for debugging and testing specific instances of the system
- ER diagrams are used in the database design phase of the software development process.
- The choice of which diagram to use depends on the **specific requirements of the software development project** and the **stage of the development process.**

Differences between class diagrams and object diagrams

A class diagram is used to represent the static structure of a software system, depicting the classes, their attributes, and their relationships with other classes.
It is a blueprint of the system, illustrating how the different components fit together.
Class diagrams are typically created early in the development process to help design the system's architecture.

On the other hand, an object diagram is used to represent a specific instance of a class at a particular moment in time.
It shows the actual objects in the system and the relationships between them.
Object diagrams are useful for understanding how the different objects in the system interact with each other and can be used to debug specific instances of the system.

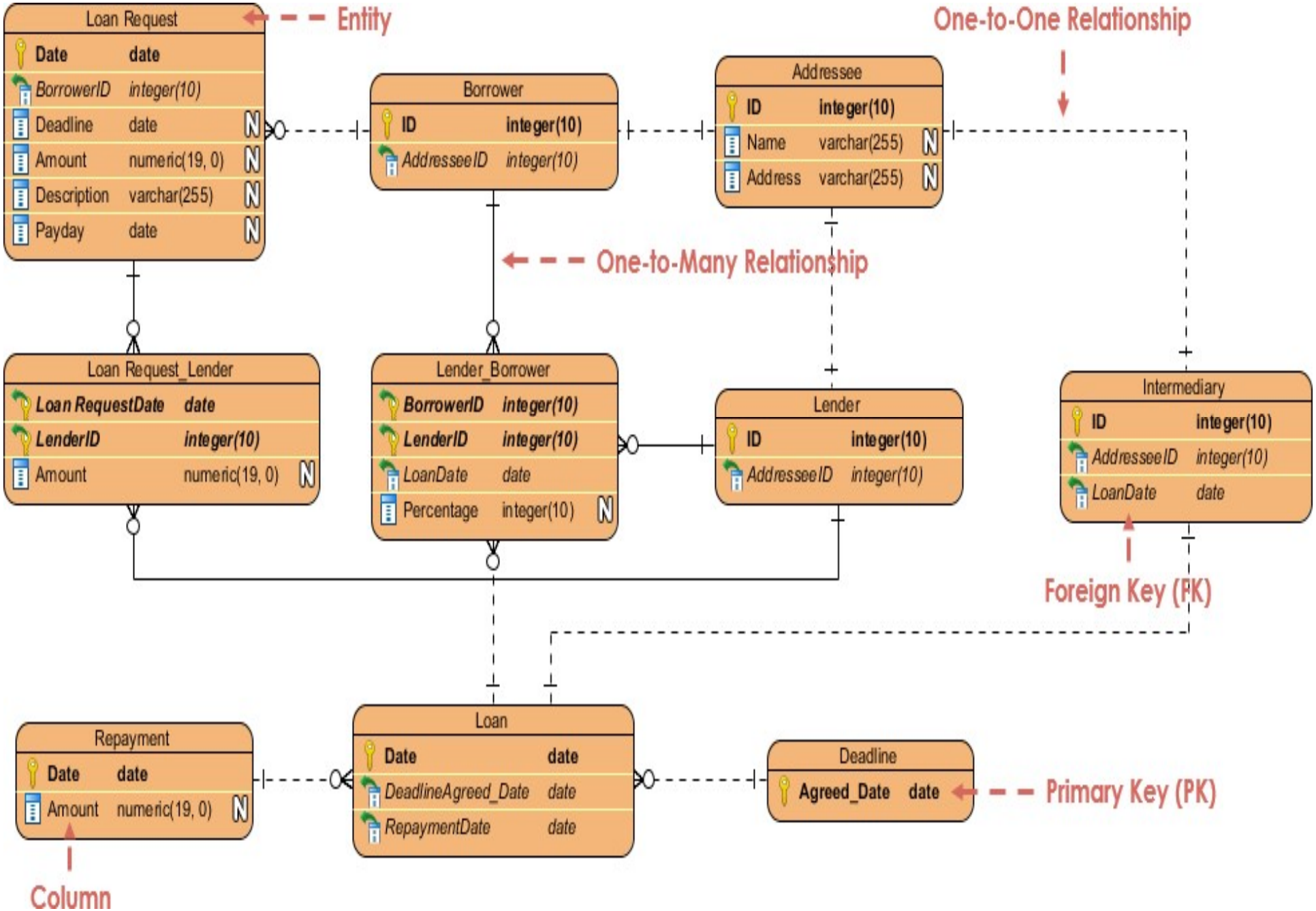
	Class Diagram	Object Diagram
Scope	Class diagrams show the structure of the entire system	object diagrams focus on a specific instance of the system
Level of detail (abstraction)	Class diagrams provide a high-level view of the system	object diagrams show a more detailed view of a specific instance.
Time	Class diagrams are created early in the development process and are used to design the system's architecture	Object diagrams are created later in the development process and are used for debugging and testing specific instances of the system.
Relationships	Class diagrams show the relationships between classes	object diagrams show the relationships between objects
useful tools for software developers	used to design the system's architecture	used to debug and test specific instances of the system

Class Diagram vs ER Diagram: Understanding the Differences and Use Cases

Class diagrams and Entity-Relationship (ER) diagrams are two popular types of diagrams used in software development to represent the structure of a system.
While they share some similarities, they are used for different purposes.

A class diagram is used to represent the static structure of a software system, depicting the classes, their attributes, and their relationships with other classes.
It is primarily used in object-oriented programming to design the structure of the system.

On the other hand, an ER diagram is used to represent the data structure of a system, depicting the entities, their attributes, and the relationships between them.
It is primarily used in database design to model the data that will be stored in the system


Loan Request
Date date
BorrowerID integer(10)
Deadline date N
Amount numeric(19, 0) N
Description varchar(255) N
Payday date N
Entity
Borrower
ID integer(10)
AddresseeID integer(10)
Addressee
ID integer(10)
Name varchar(255) N
Address varchar(255) N
One-to-One Relationship
One-to-Many Relationship
Loan Request_Lender
Loan RequestDate date
LenderID integer(10)
Amount numeric(19, 0) N
Lender_Borrower
BorrowerID integer(10)
LenderID integer(10)
LoanDate date
Percentage integer(10) N
Lender
ID integer(10)
AddresseeID integer(10)
Intermediary
ID integer(10)
AddresseeID integer(10)
LoanDate date
Foreign Key (FK)
Repayment
Date date
Amount numeric(19, 0) N
Column
Loan
Date date
DeadlineAgreed_Date date
RepaymentDate date
Deadline
Agreed_Date date
Primary Key (PK)

	Class Diagram	ER Diagram
Scope	Class diagrams show the structure of the entire system	ER diagrams are used to represent the structure of a database system.
Level of detail (abstraction)	Class diagrams are more abstract and focus on the design of the system	ER diagrams are more concrete and focus on the data that will be stored in the system.
Relationships	Class diagrams show the relationships between classes	ER diagrams show the relationships between entities.
Attributes	Class diagrams show the attributes of classes	ER diagrams show the attributes of entities

e-commerce platform for a Retail Company

Suppose you are tasked with designing a new e-commerce platform for a retail company. The company wants to allow customers to browse and purchase products online, as well as manage their account information and order history.

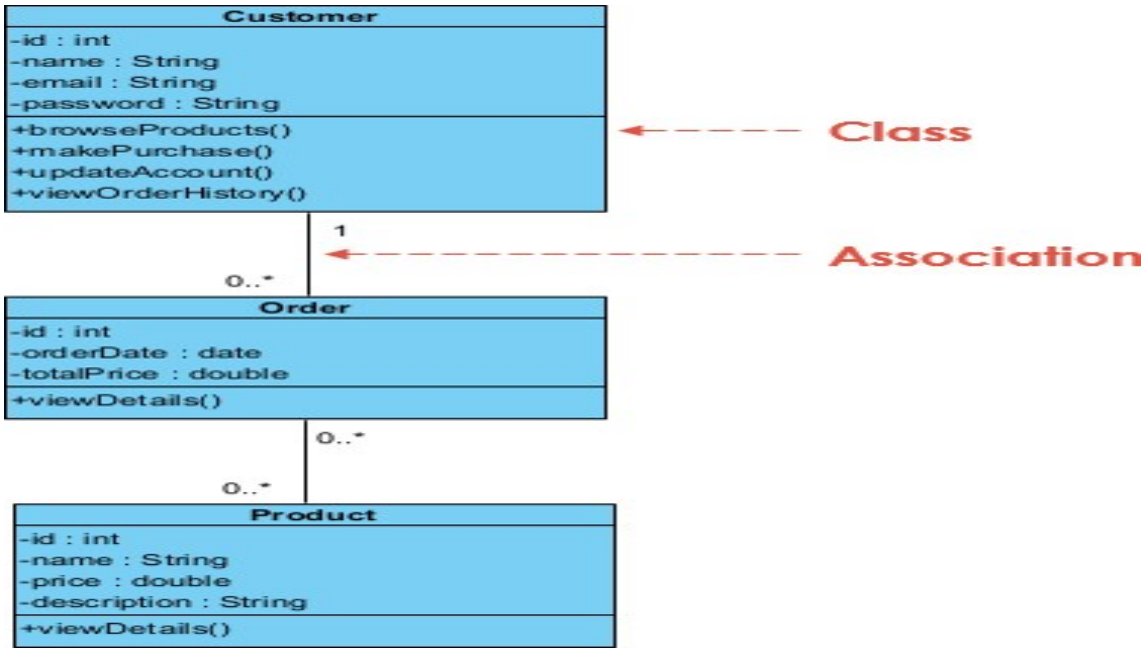
The platform needs to be scalable, secure, and able to handle a large number of concurrent users.

To develop this platform, you need to create a detailed blueprint that describes the architecture and functionality of the system.

This is where class diagrams, ER diagrams, and object diagrams come in handy.

Develop the Class Diagram

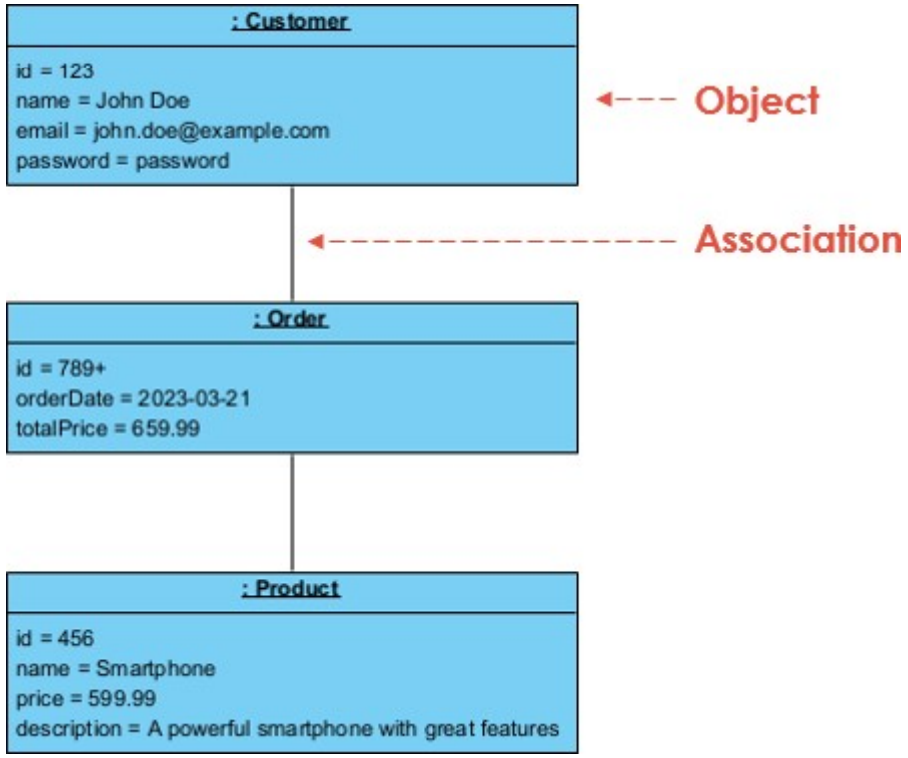
The Class Diagram as shown below, provides an overview of the classes and their relationships in an object-oriented system.
In the example generated above, the classes identified include Customer, Product, and Order, each with its respective attributes and methods.
The Class Diagram also indicates the relationships between classes, such as the one-to-many relationship between Customer and Order, and the many-to-many relationship between Order and Product.



Object Diagram

It represents the objects in the system and their relationships. In the example generated above, the Object Diagram shows a specific instance of Customer, Order, and Product.

The diagram indicates that the Customer object is associated with a specific Order object, and that the Order object contains specific Product objects.



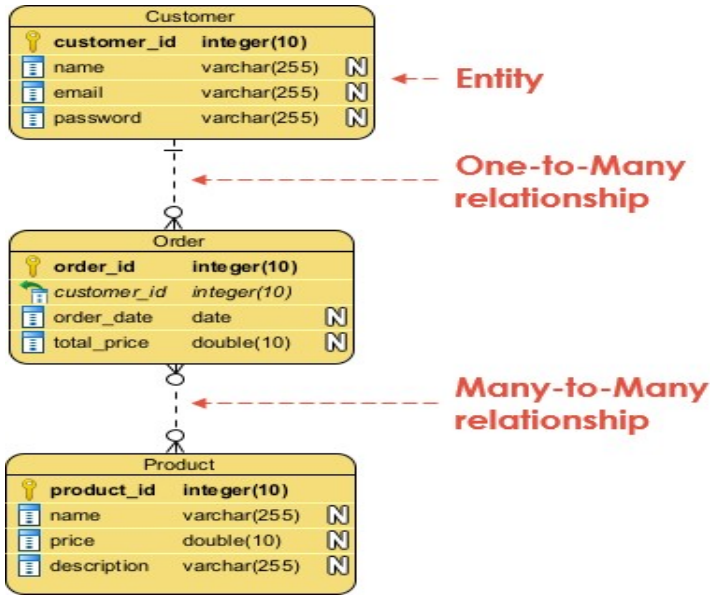
Develop the ERD

The Class Diagram and ERD (Entity Relationship Diagram) are both modeling tools used to represent data structures and relationships between entities in a system.

The Class Diagram is mainly used in object-oriented systems to show the classes, their attributes, methods, and relationships with other classes. It is often used to depict the **static structure of an OO system.**

The ERD, on the other hand, is used to represent the **data structure of a system** and the relationships between entities in that system. It is used primarily in database systems to describe the logical structure of the database.

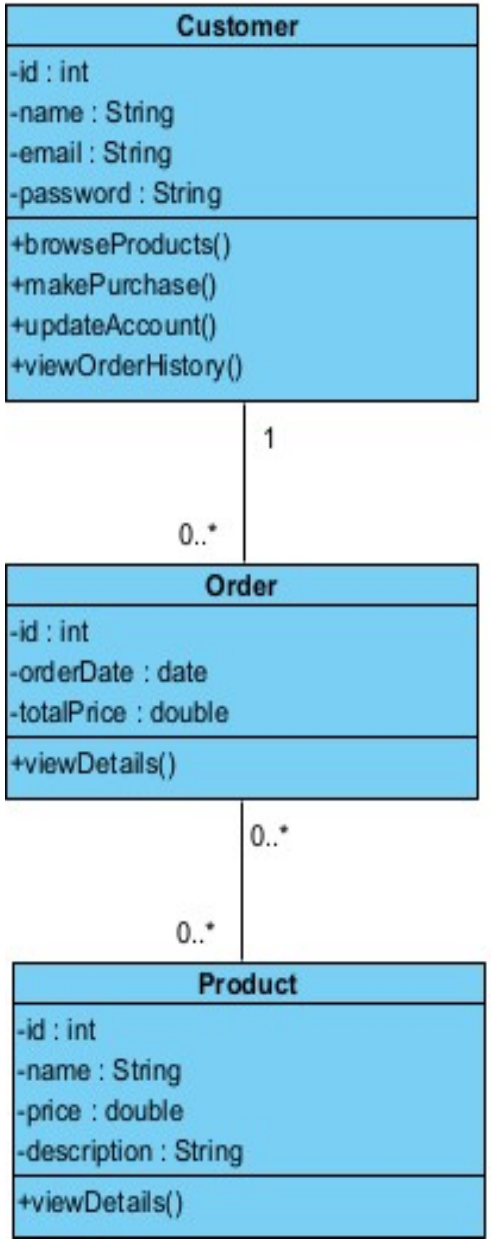
The ERD also indicates the relationships between entities, such as the one-to-many relationship between Customer and Order, and the many-to-many relationship between Order and Product.



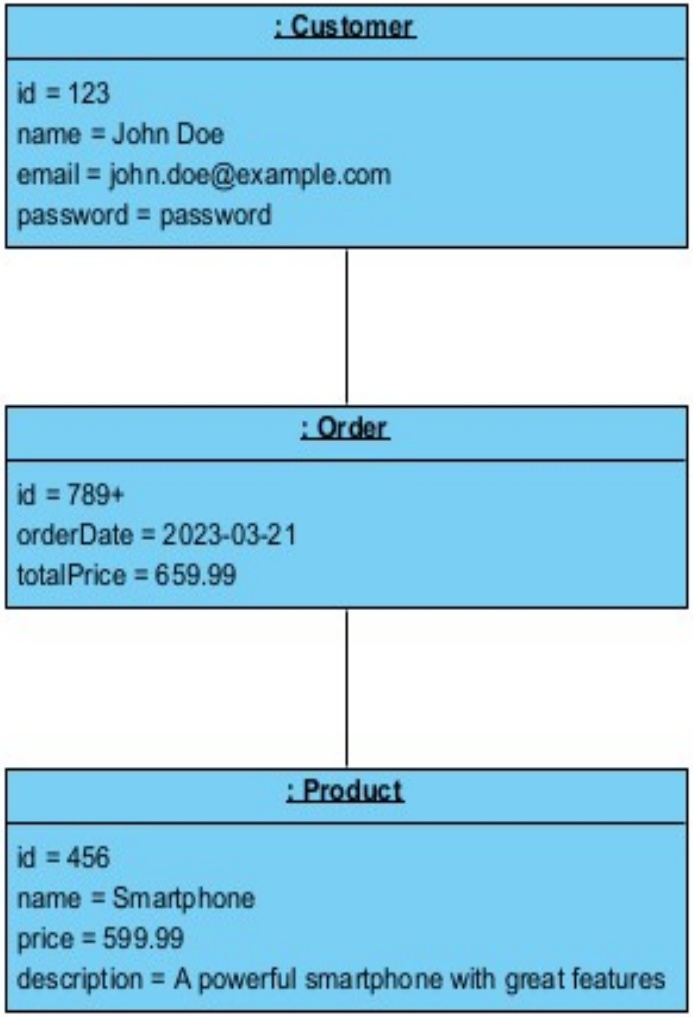
DB Schema

Customer
– customer_id (PK)
– name
– email
– password
Order
– order_id (PK)
– customer_id (FK)
– order_date
– total_price
Order_Product
– order_id (FK, PK)
– product_id (FK, PK)
– quantity
Product
– product_id (PK)
– name
– price
– description

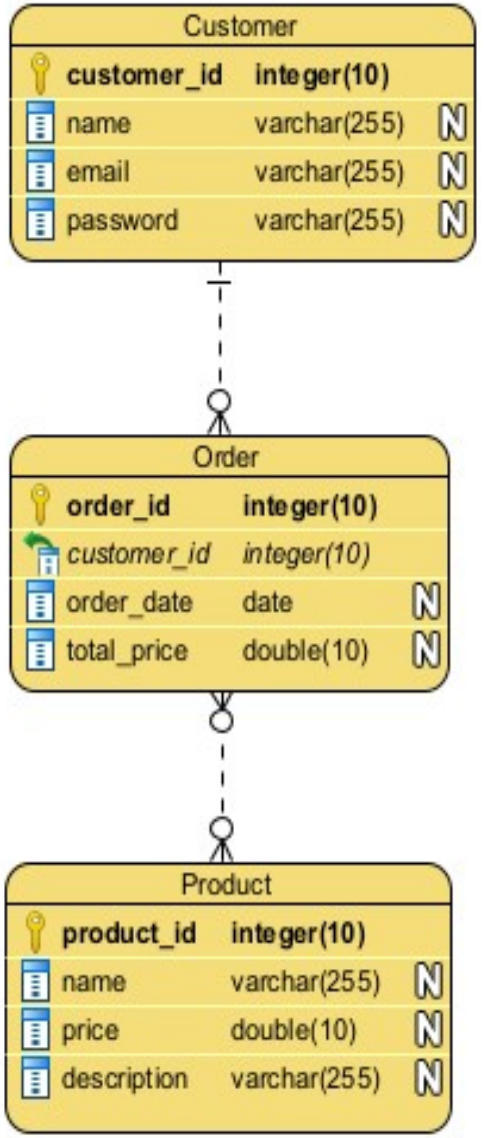
Class Diagram



Object Diagram



ER Diagram



Propagation of operations

- Propagation is the automatic application of an operation to a network of objects when the operation is applied to some starting object.
- Propagation of operations to parts is often a good indicator of propagation.

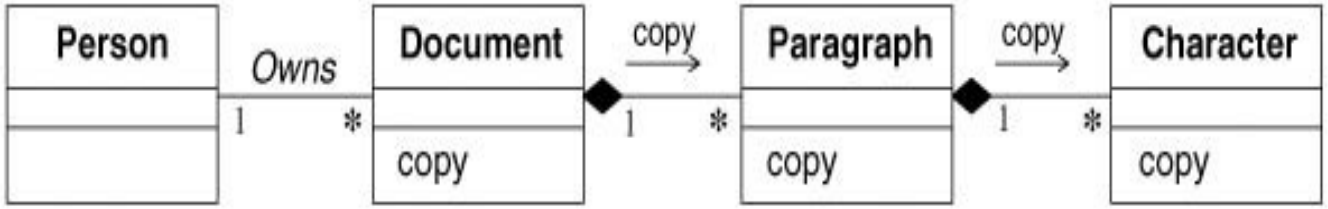


Figure 4.11 Propagation. You can propagate operations across aggregations and compositions.

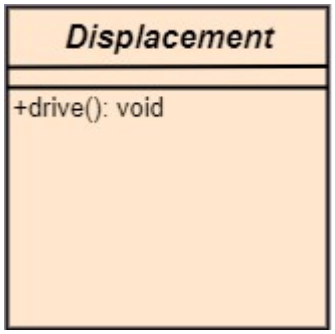
Abstract Class

- In the abstract class, **no objects can be a direct entity of the abstract class**.

- The abstract class can neither be declared nor be instantiated. It is used to find the functionalities across the classes.

- The notation of the abstract class is similar to that of class; the only difference is that the **name of the class is written in italics.**

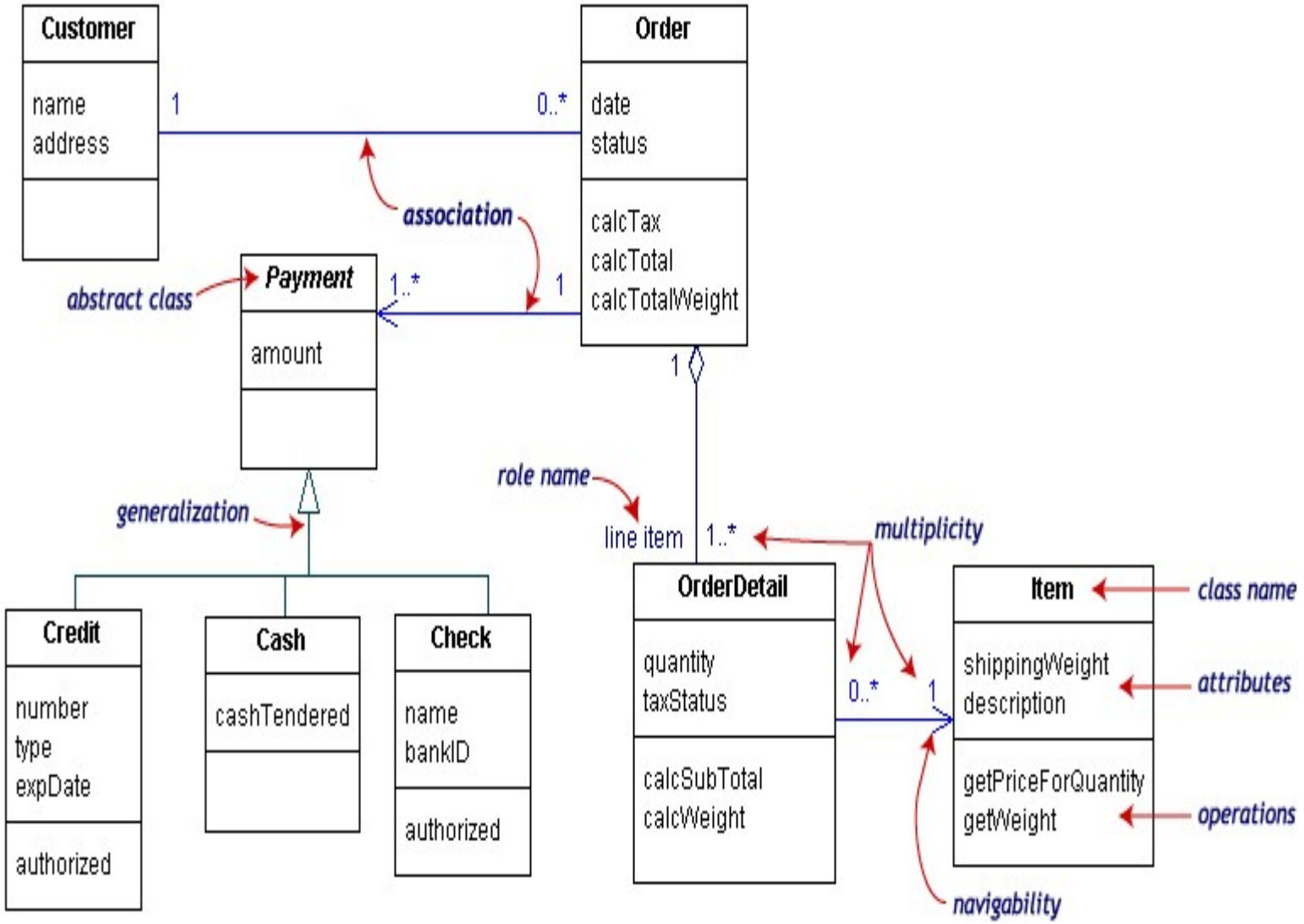
- Since it does not involve any implementation for a given function, it is best to use the abstract class with multiple objects.



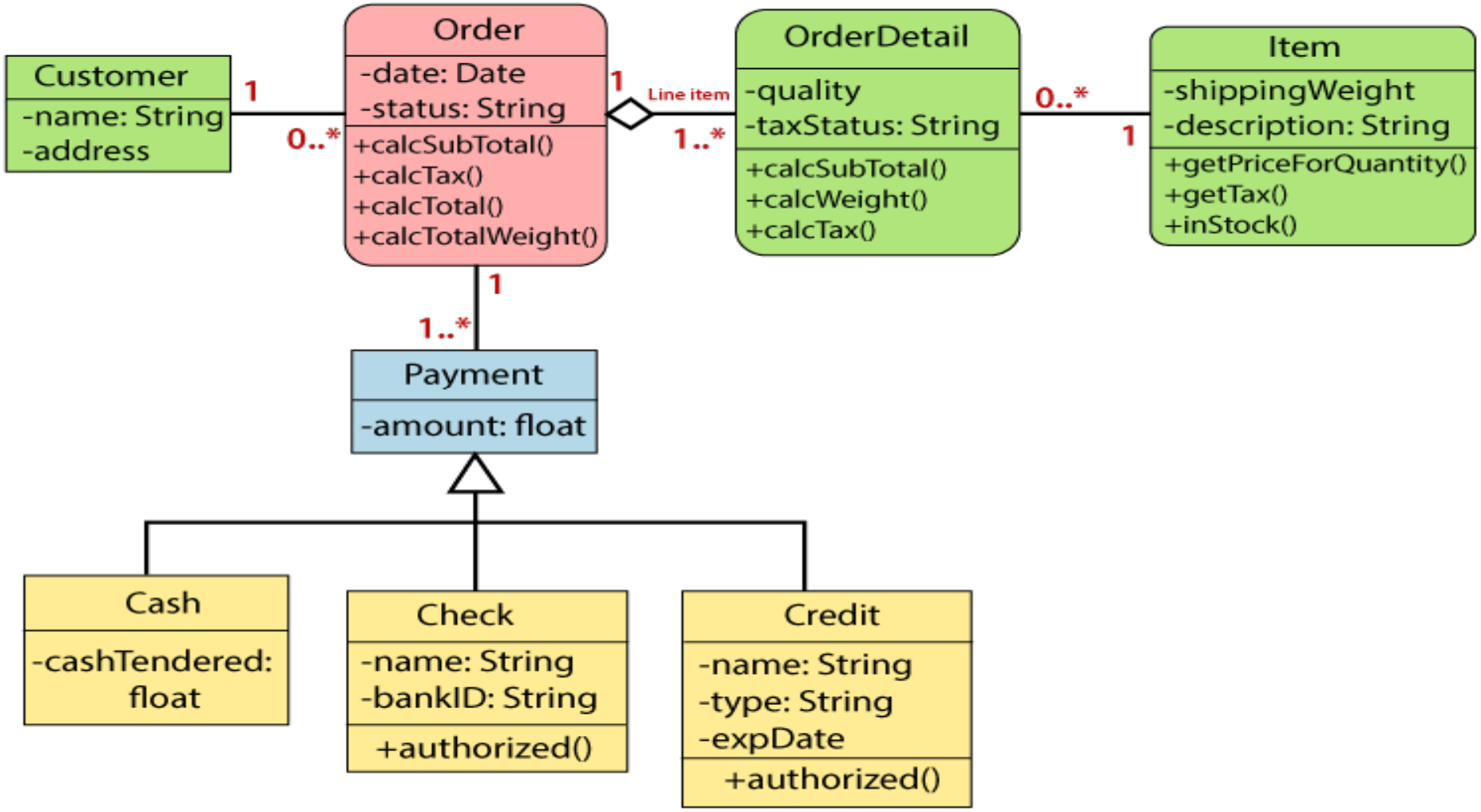
How to draw Class Diagram

- To describe a complete aspect of the system, it is suggested to give a meaningful name to the class diagram.
- The objects and their relationships should be acknowledged in advance.
- The attributes and methods (responsibilities) of each class must be known.
- A minimum number of desired properties should be specified as more number of the unwanted property will lead to a complex diagram.
- Notes can be used as and when required by the developer to describe the aspects of a diagram.
- The diagrams should be redrawn and reworked as many times to make it correct before producing its final version.

Example of identifying Class Relations, Multiplicities, Attributes and operations



Class Diagram describing the sales order system



Classes

In UML, a *class* represents an object or a set of objects that share a common structure and behavior. Classes, or instances of classes, are common model elements in UML diagrams.

Objects

In UML models, *objects* are model elements that represent instances of a class or of classes. You can add objects to your model to represent concrete and prototypical instances. A concrete instance represents an actual person or thing in the real world. For example, a concrete instance of a Customer class represents an actual customer. A prototypical instance of a Customer class contains data that represents a typical customer.

Packages

Packages group related model elements of all types, including other packages.

Signals

In UML models, *signals* are model elements that are independent of the classifiers that handle them. Signals specify one-way, asynchronous communications between active objects.

Enumerations

In UML models, *enumerations* are model elements in class diagrams that represent user-defined data types. Enumerations contain sets of named identifiers that represent the values of the enumeration. These values are called enumeration literals.

Data types

In UML diagrams, *data types* are model elements that define data values. You typically use data types to represent primitive types, such as integer or string types, and enumerations, such as user-defined data types.

Artifacts

In UML models, *artifacts* are model elements that represent the physical entities in a software system. Artifacts represent physical implementation units, such as executable files, libraries, software components, documents, and databases.

Relationships in class diagrams

In UML, a relationship is a connection between model elements. A UML relationship is a type of model element that adds semantics to a model by defining the structure and behavior between model elements.

Qualifiers on association ends

In UML, *qualifiers* are properties of binary associations and are an optional part of association ends. A qualifier holds a list of association attributes, each with a name and a type. Association attributes model the keys that are used to index a subset of relationship instances.

Usage of Class Diagram

The class diagram is used to represent a static view of the system. It plays an essential role in the establishment of the component and deployment diagrams. It helps to construct an executable code to perform forward and backward engineering for any system, or we can say it is mainly used for construction. It represents the mapping with object-oriented languages that are C++, Java, etc.

Used for
To describe the static view of a system.
To show the collaboration among every instance in the static view.
To describe the functionalities performed by the system.
To construct the software application using object-oriented languages.

Tables & Columns

A table in the UML Data Profile is a class with the **«Table» stereotype**, displayed as above with a table icon in the **top right corner**.

Database columns are modelled as attributes of the «Table» class.

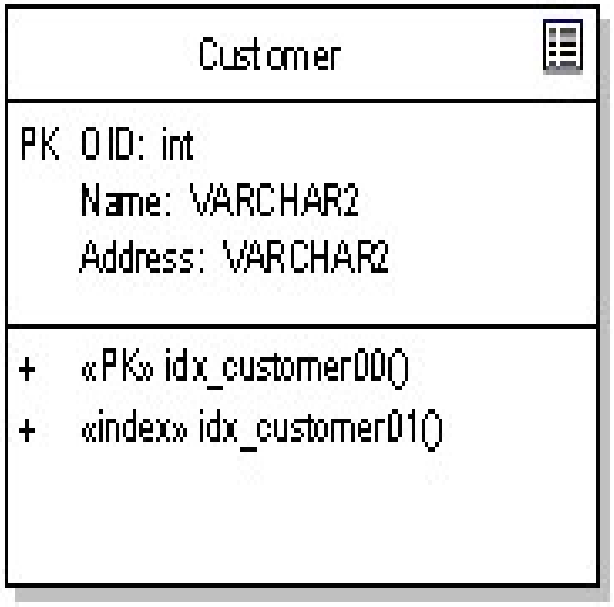
an object id has been defined as the primary key, as well as two other columns, Name and Address.

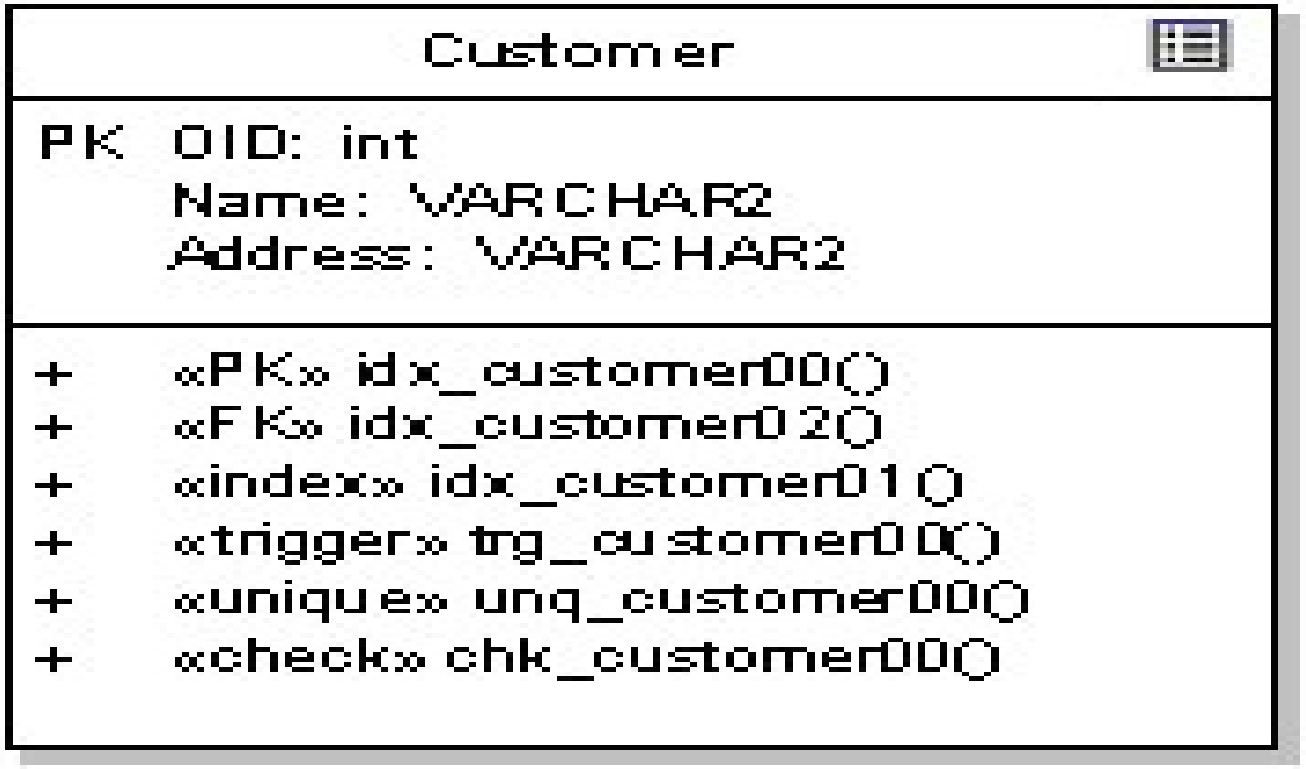
Behavior

W.r.to Class Diagram, So far we have only defines the logical (static) structure of the table;

In addition we should describe the behavior associated with columns, including indexes, keys, triggers, procedures & etc. Behavior is represented as stereotyped **operations**.

a primary key constraint and index, both defined as stereotyped operations:





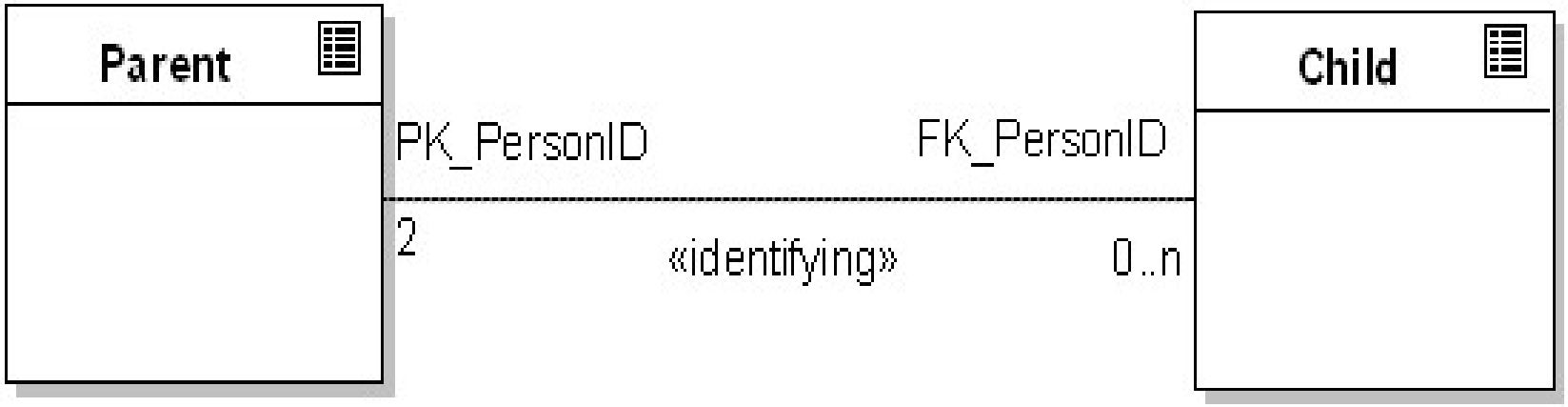
Behavior

A primary key constraint (PK);
A Foreign key constraint (FK);
An index constraint (Index);
A trigger (Trigger);
A uniqueness constraint (Unique);
Validity check (Check).

Relationships

defines a relationship as a dependency of any kind between two tables. It is represented as **a stereotyped association and includes a set of primary and foreign keys**.

The data profile goes on to require that a relationship always involves a parent and child, the parent defining a primary key and the child implementing a foreign key based on all or part of the parent primary key.



An identifying relationship between child and parent, with role names based on primary to foreign key relationship.

Mapping from the Class Model to the Relational Model

Having described the two domains of interest and the notation to be used, we can now turn our attention as to how to map or translate from one domain to the other. The strategy and sequence presented below is meant to be suggestive rather than proscriptive - adapt the steps and procedures to your personal requirements and environment.

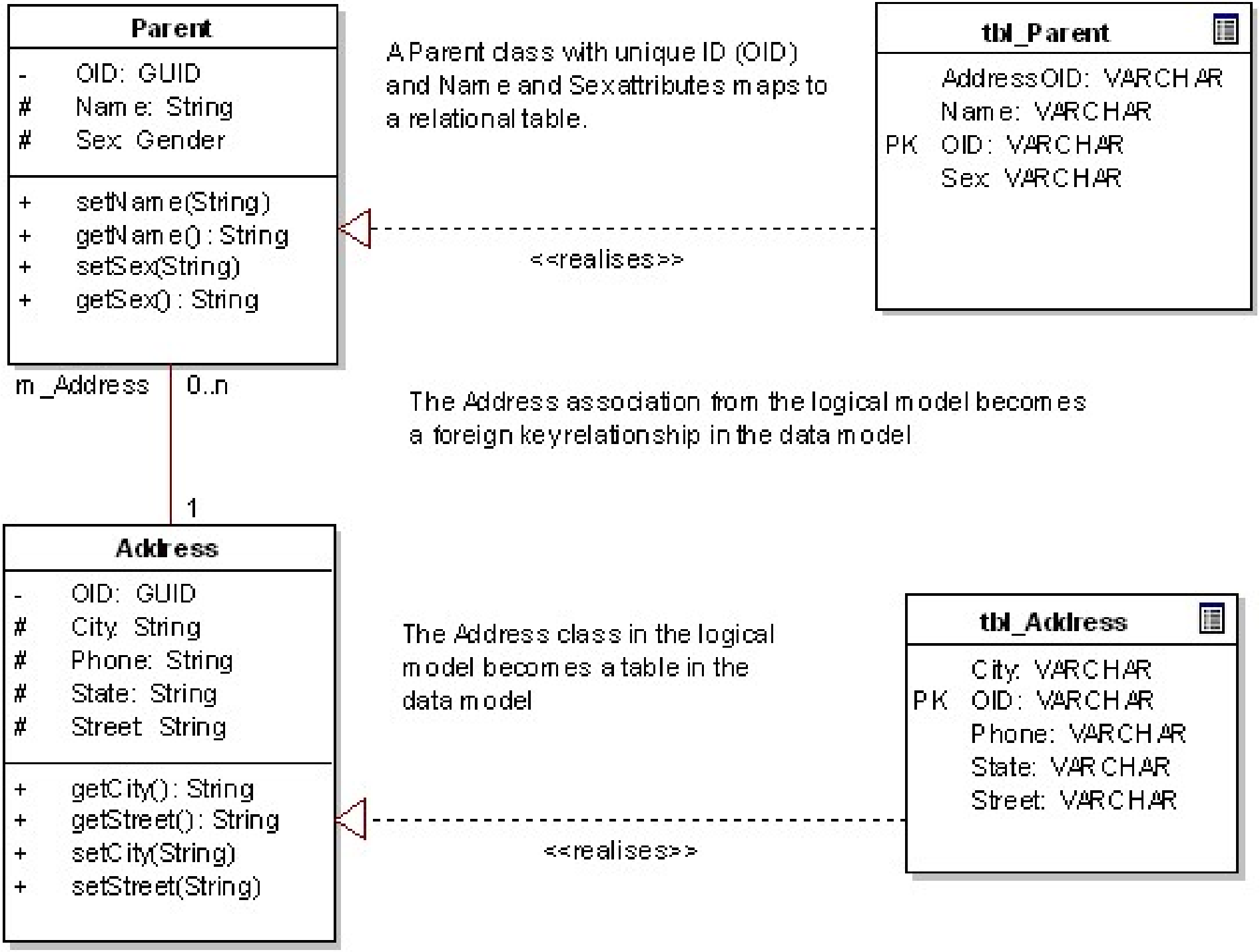
1. **Model Classes**

2. Identify persistent objects

Having built our class model we need to separate it into those elements that require persistence and those that do not. For example, if we have designed our application using the **Model-View-Controller design pattern**, then only classes in the model section would require persistent state.

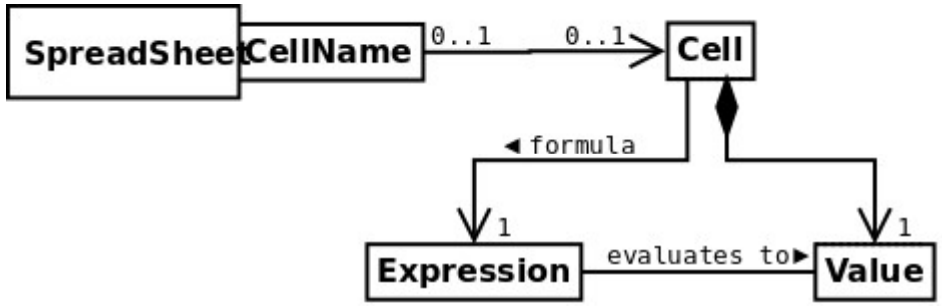
3. Assume each persistent class maps to one relational table

In the simplest model a class from the logical model maps to a relational table, either in whole or in part. The logical extension of this is that a single object (or instance of a class) maps to a single table row.


Parent
- OID: GUID
Name: String
Sex: Gender
+ setName(String)
+ getName() : String
+ setSex(String)
+ getSex() : String
A Parent class with unique ID (OID) and Name and Sex attributes maps to a relational table.
tbl_Parent
AddressOID: VARCHAR
Name: VARCHAR
PK OID: VARCHAR
Sex: VARCHAR
<<realises>>
m_Address
0..n
1
The Address association from the logical model becomes a foreign key relationship in the data model
Address
- OID: GUID
City: String
Phone: String
State: String
Street: String
+ getCity() : String
+ getStreet() : String
+ setCity(String)
+ setStreet(String)
The Address class in the logical model becomes a table in the data model
tbl_Address
City: VARCHAR
PK OID: VARCHAR
Phone: VARCHAR
State: VARCHAR
Street: VARCHAR
<<realises>>

Qualification

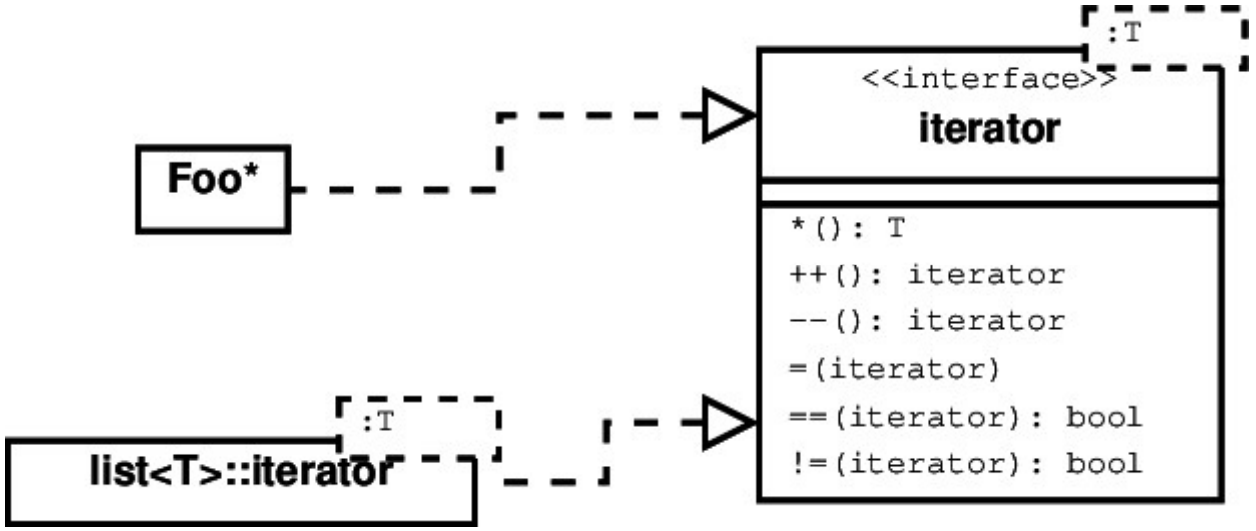
A qualified association describes a situation in which one class is related to multiple instances of another, but the collection of related instances is “indexed” by a third class.



This diagram indicates that a spreadsheet provides access to cells retrieved by cell name. It suggests an eventual implementation by something along the lines of a map or table.

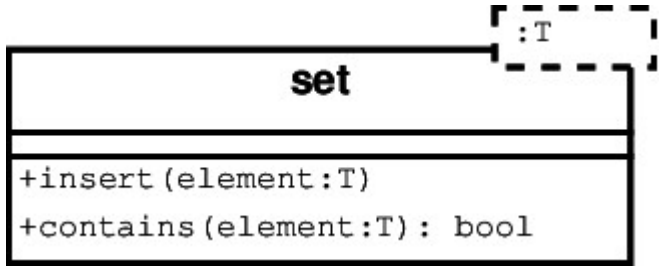
Satisfies / Realizes

this is actually a combination of dependency with the arrowhead of generalization.

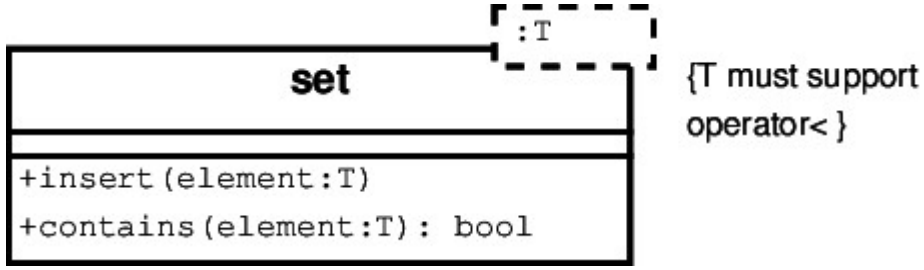


Parameterized Classes

Used to represent templates and similar concepts.

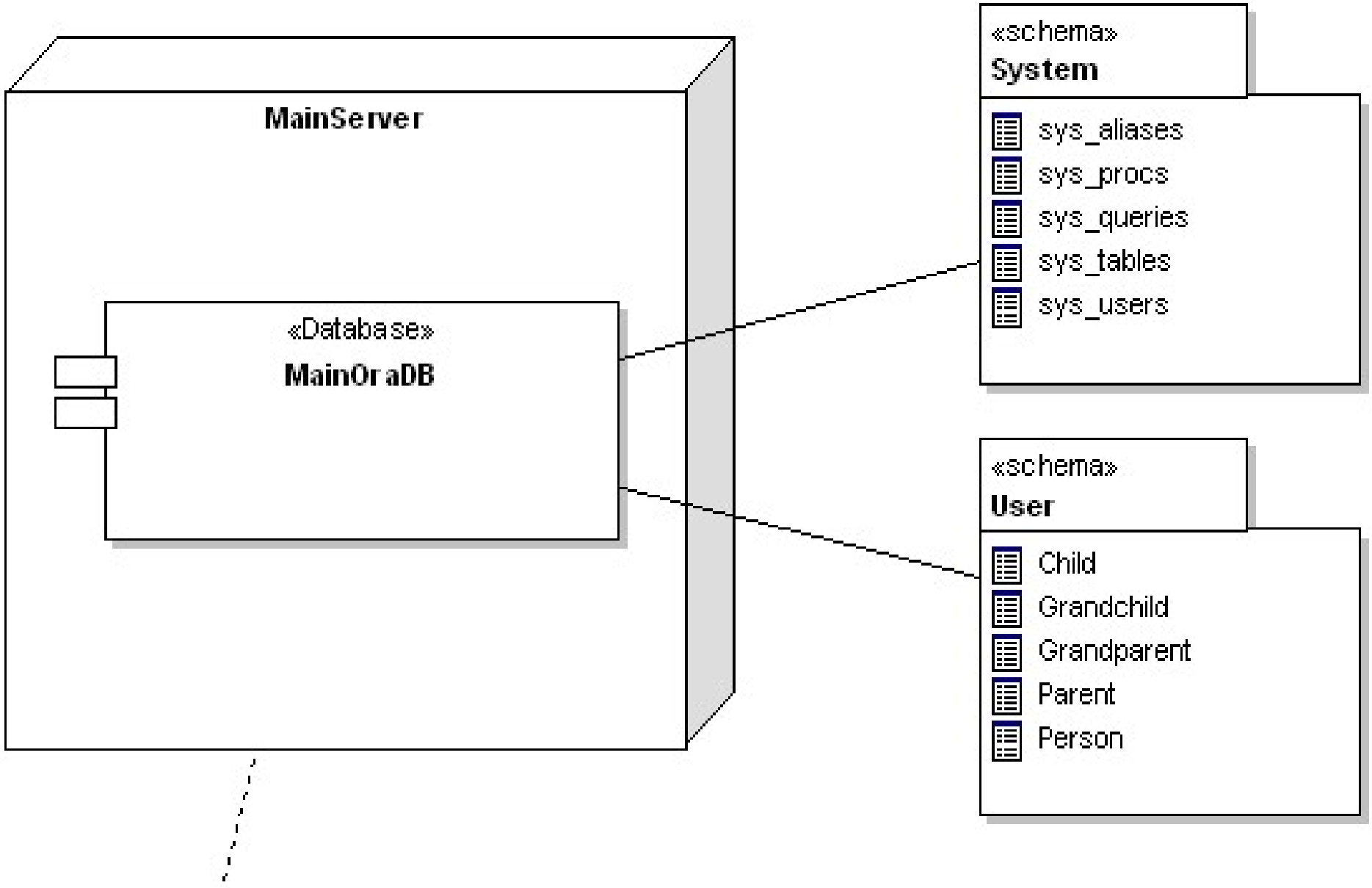


Constraints can be added almost any place by writing them within brackets



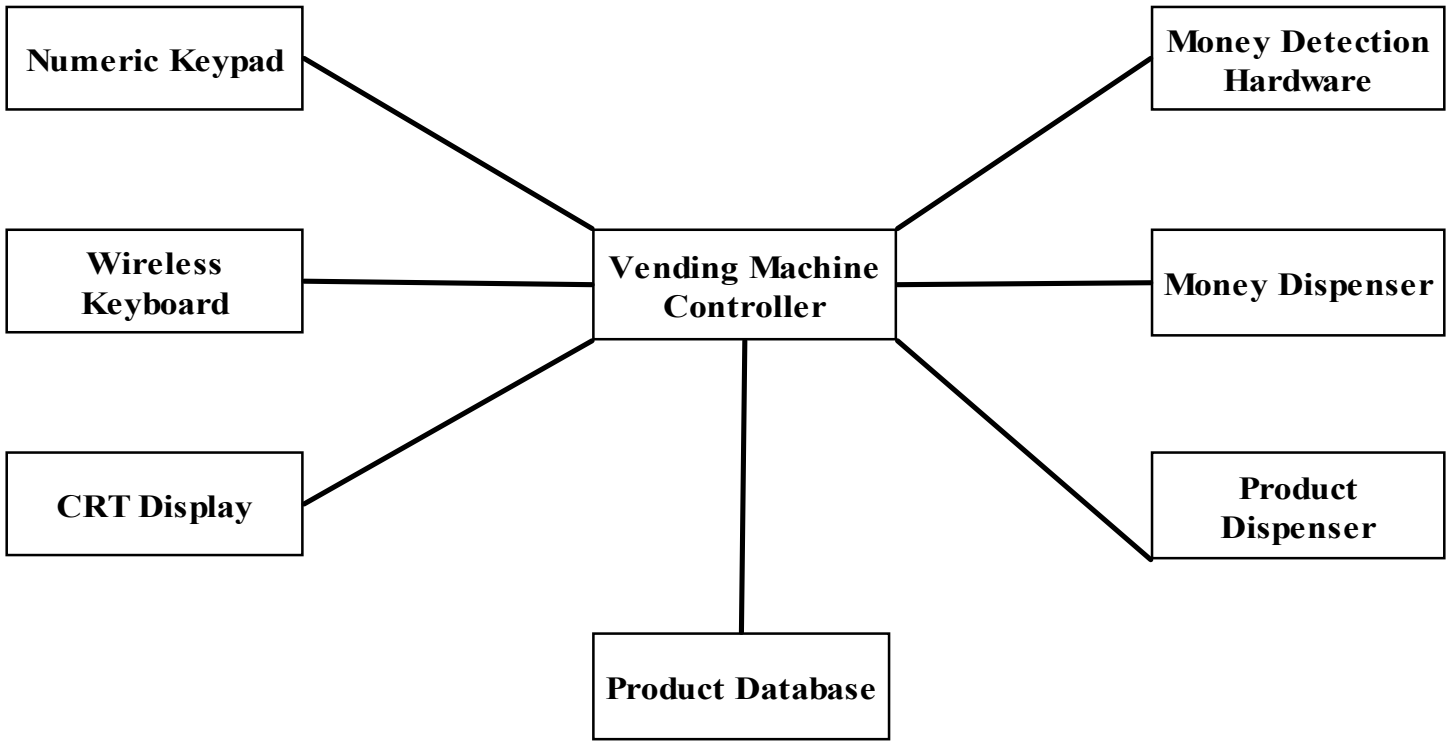
A signal event represents a named object that is dispatched (thrown) asynchronously by one object and then received (caught) by another. Exceptions are an example of internal signal.

- A signal event is an asynchronous event
- Signal events may have instances, generalization relationships, attributes and operations. Attributes of a signal serve as its parameters.
- A signal event may be sent as the action of a state transition in a state machine or the sending of a message in an interaction.
- Signals are modeled as stereotyped classes and the relationship between an operation and the events by using a dependency relationship, stereotyped as send

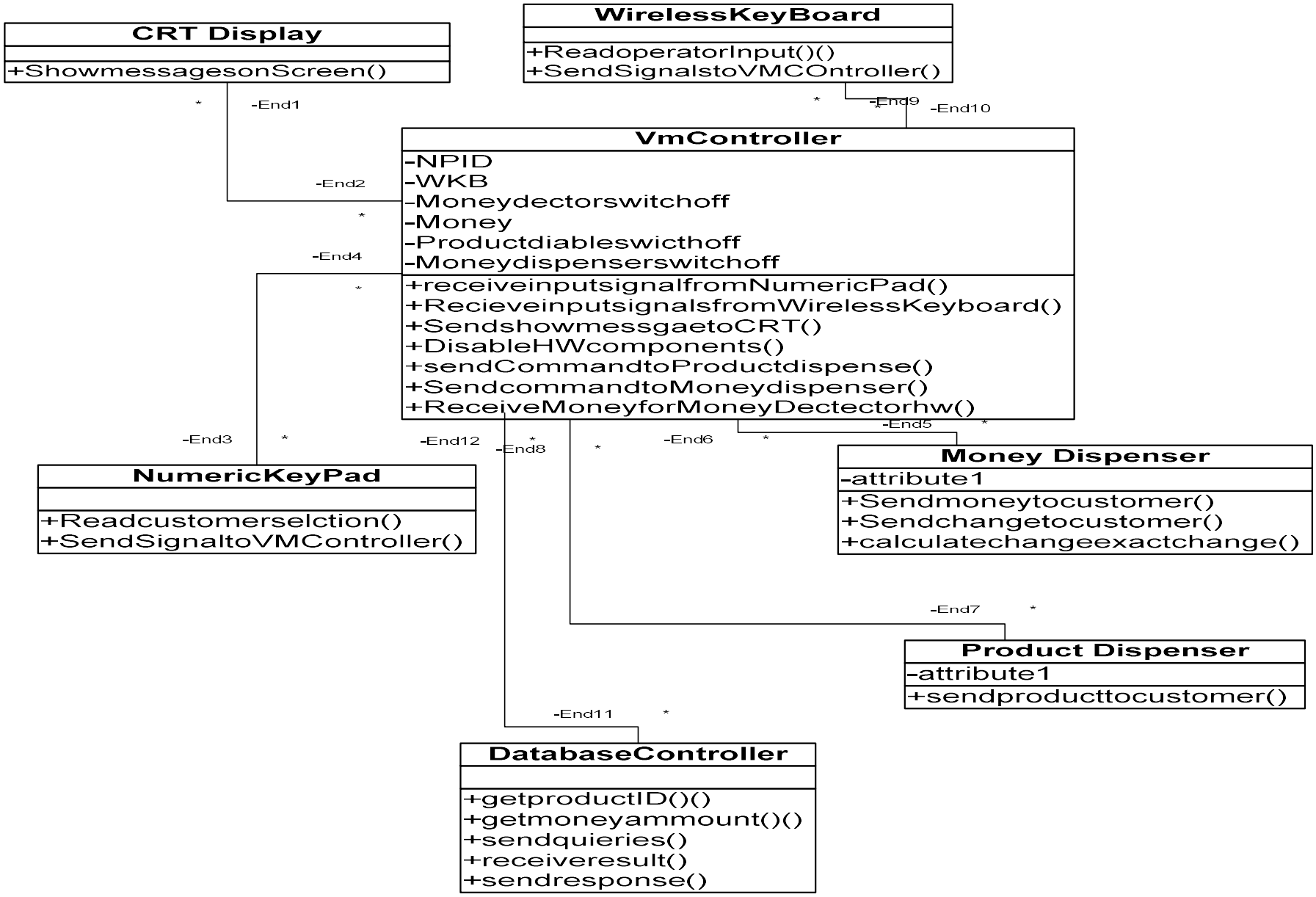


A Node is a physical piece of hardware (such as a Unix server) on which components are deployed. The database component in this example is also mapped to two logical «schema», each of which contains a number of tables.

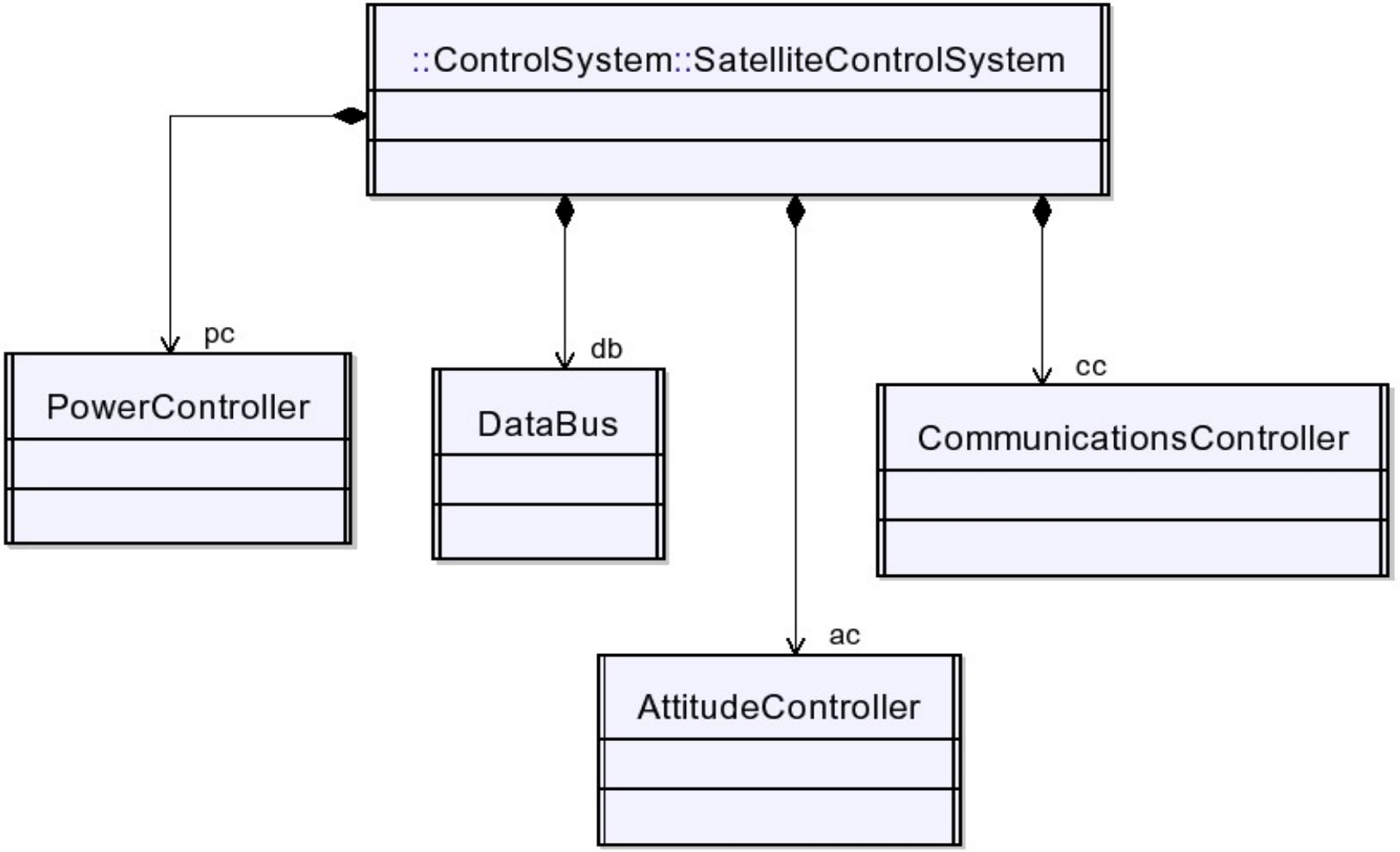
The Vending Machine Analysis Level Class Diagram



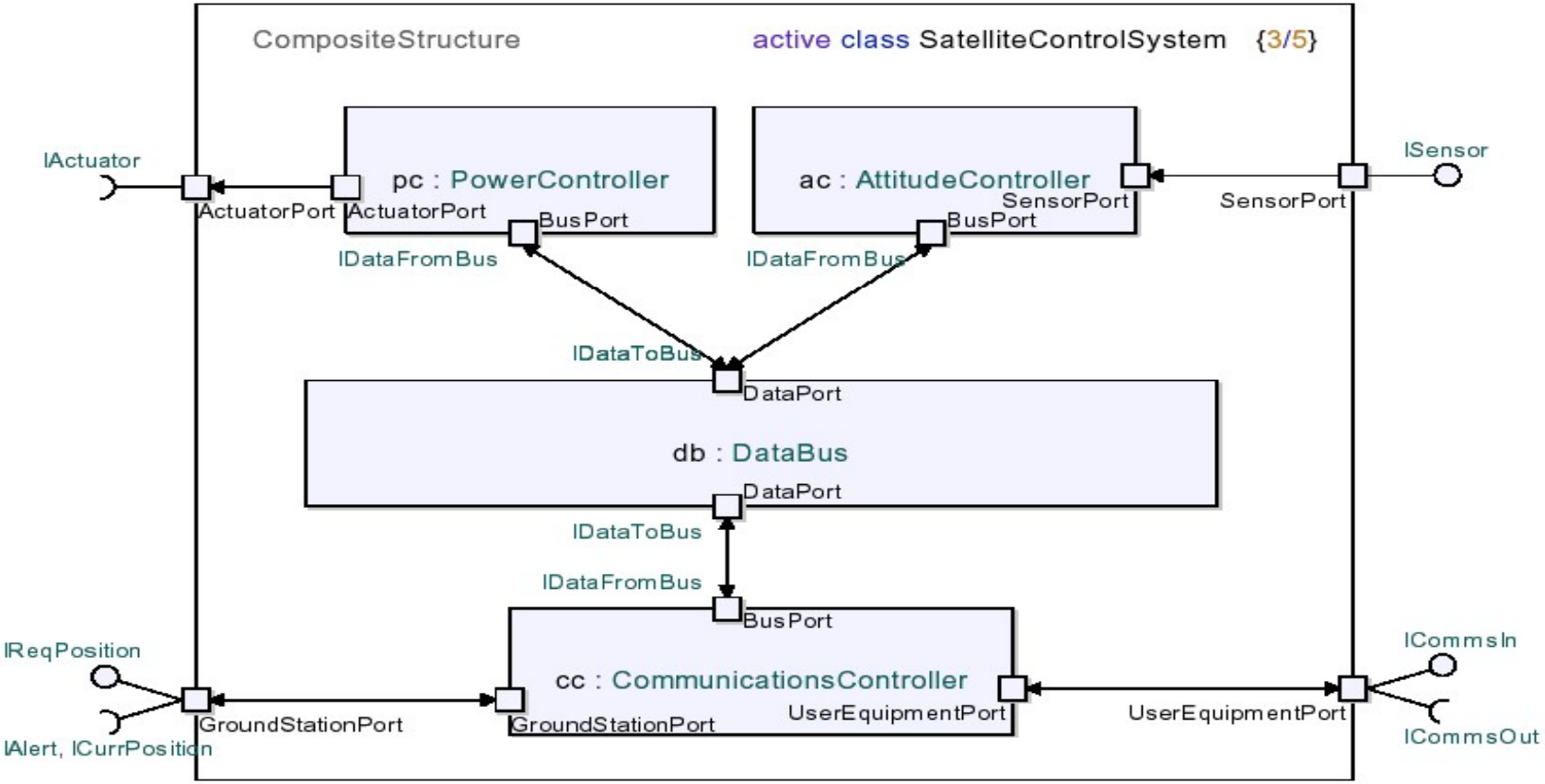
The Vending Machine
Design Level Class Diagram



Example of Software Architecture Using UML2



A Simple Example of Software Architecture Using UML2



When to Use Class Diagrams

- Class diagram is a static diagram and it is used to model the static view of a system.
- The static view describes the vocabulary of the system.
- Class diagram clearly shows the mapping with object oriented language such as Java, C++ etc.
- Practically class diagram is generally used for construction purpose.
- Class diagram is also considered as the foundation for component and deployment diagrams.
- In a nutshell it can be said, class diagrams are used for:
  - Describing the static view of the system
  - Showing the collaboration among the elements of the static view.

Static vs. Dynamic Design

- Static design describes code structure and object relations
  - Class relations
  - Objects at design time
  - Doesn’t change
- Dynamic design shows communication between objects
  - Similarity to class relations
  - Can follow sequences of events
  - May change depending upon execution scenario
  - Called Object Diagrams

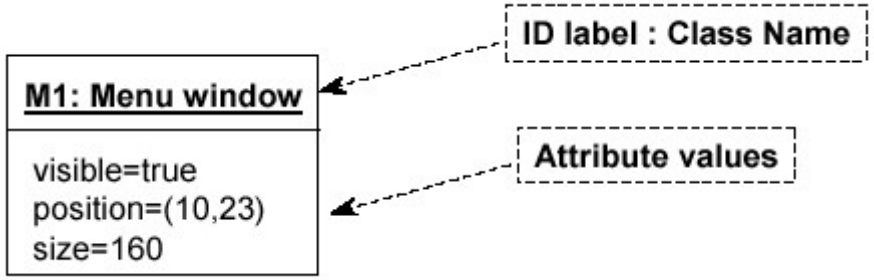
Object Diagrams

- Shows instances of Class Diagrams and links among them
  - An object diagram is a snapshot of the objects in a system
    - At a point in time
    - With a selected focus
      - Interactions – Sequence diagram
      - Message passing – Collaboration diagram
      - Operation – Deployment diagram

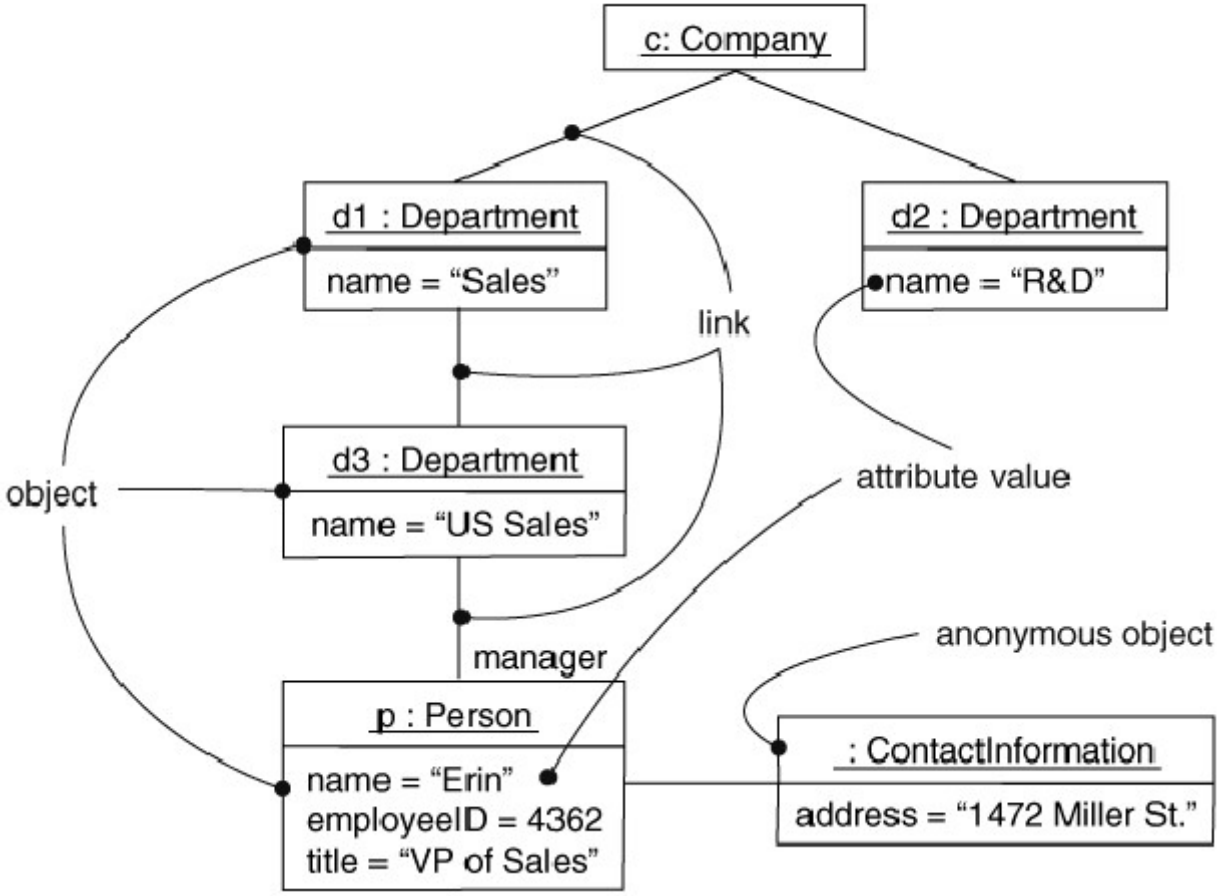
Object Diagrams

- Format is
  - Instance name : Class name
  - Attributes and Values

  - Example:



Objects and Links



Can add association type and also message type

Package Diagram

- A Package diagram falls under the **structural diagramming family.**
- A Package diagram shows the arrangement and organization of model elements in a project.
- It shows both structure and dependencies between sub-systems or modules.
- Package is the major element used in the Package diagram.

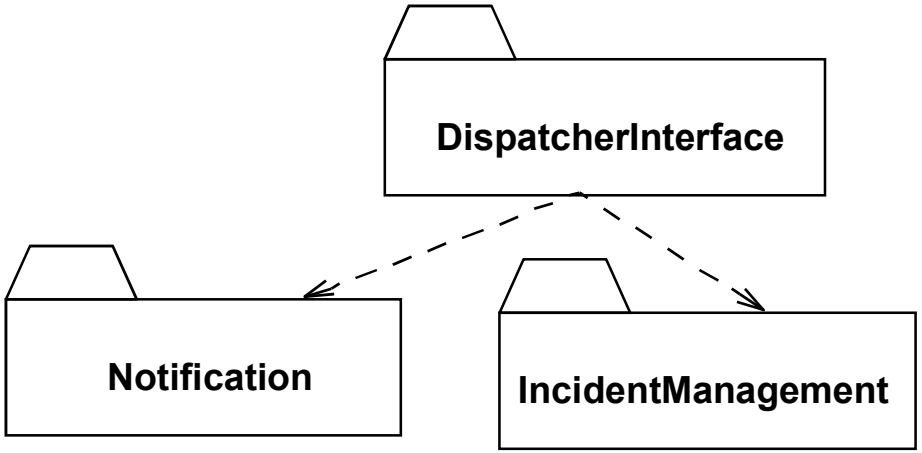
Package Diagrams

- To organize complex class diagrams, you can group classes into packages.
- A package is a collection of logically related UML elements.
- Packages are depicted as file folders and can be used on any of the UML diagrams.
- Notation
  - Packages appear as rectangles with small tabs at the top.
  - The package name is on the tab or inside the rectangle.
  - The dotted arrows are dependencies. One package depends on another if changes in the other could possibly force changes in the first.
  - Packages are the basic grouping construct with which you may organize UML models to increase their readability

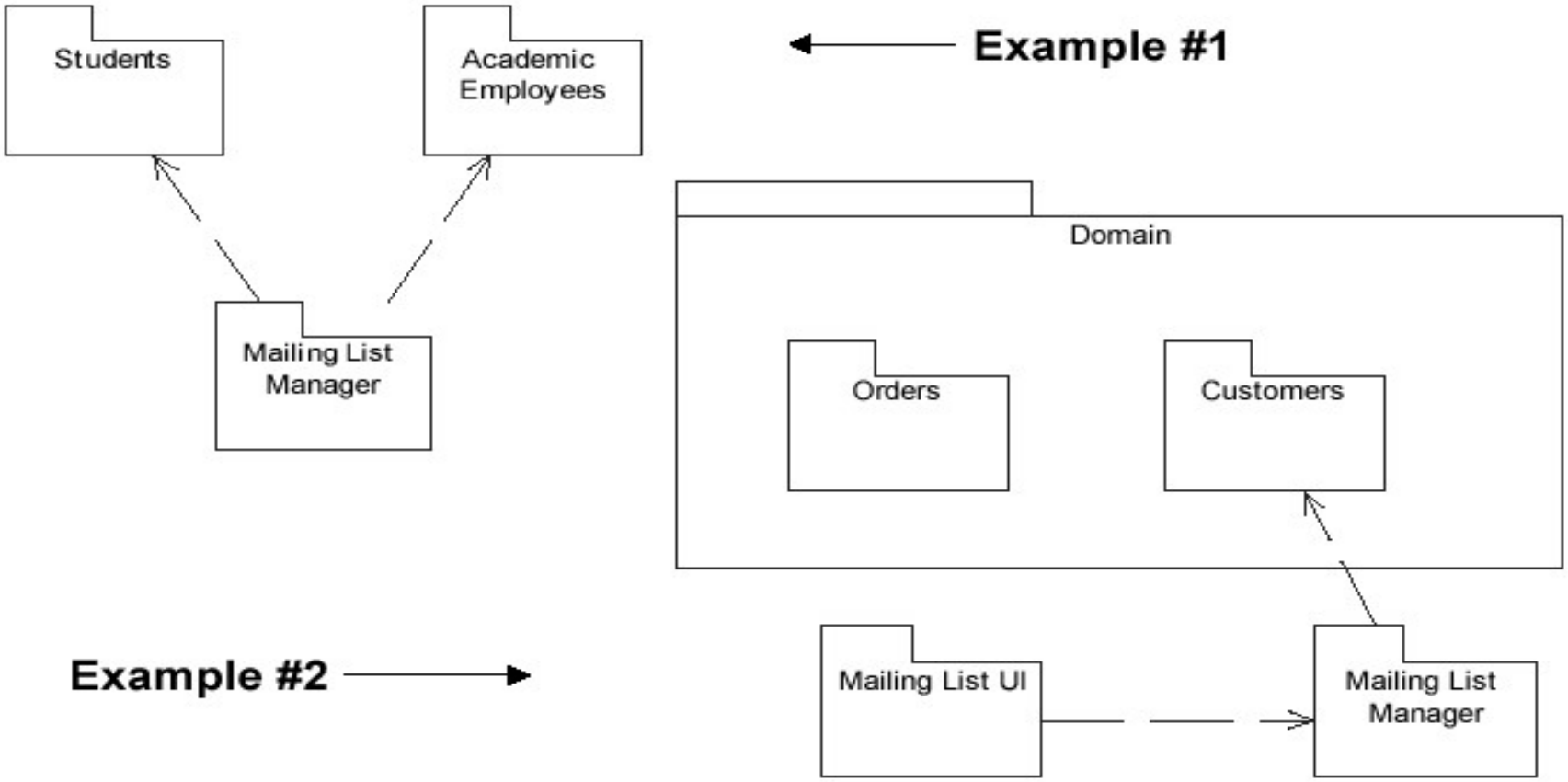
Guidelines for Creating Package Diagram

- Give **Packages Simple, Descriptive Names**
- Apply **Packages to Simplify Diagrams**
- Packages Should be **Cohesive**
- Indicate Architectural Layers With Stereotypes on Packages
- Avoid Cyclic Dependencies Between Packages
- Package Dependencies Should Reflect Internal Relationships

Package Example



More Package Examples



NameSpace:

- It represents the package's name in the diagram.
- It generally appears on top of the package symbol which helps to uniquely identify the package in the diagram.

Package Merge:

- It is a relationship that signifies how a package can be merged or combined.
- It is represented as a direct arrow between two packages. Signifying that the contents of one package can be merged with the contents of the other.

Package Import:

- It is another relationship that shows one package's access to the contents of a different package.
- It is represented as a Dashed Arrow.

Dependency:

- Dependencies are used to show that there might be some **element or package** that can be **dependent upon any other element or package,** meaning that changing anything of that package will result in alteration of the contents of the other package which is dependent upon the first one.

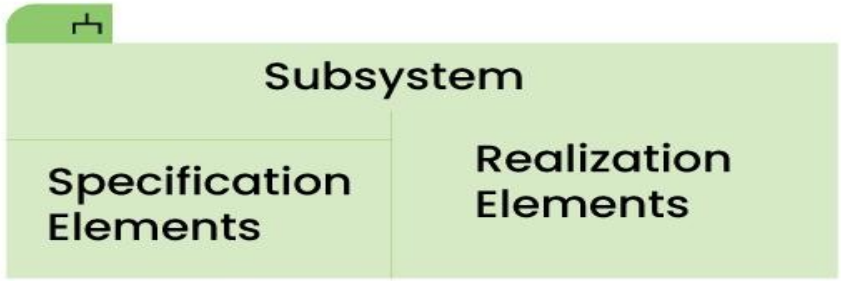
Element:

- An element can be a single unit inside of a package, it can be a **class, an interface or subsystems**.
- This packages are connected and reside inside of packages that hold them.

Constraint:

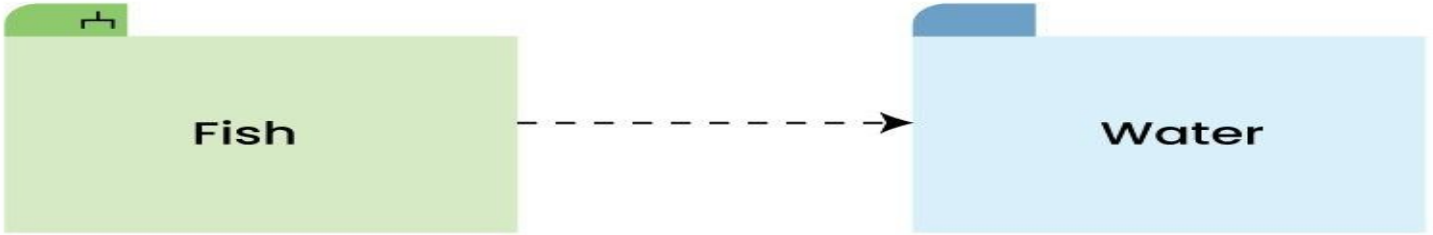
It is like a condition or requirement set related to a package. It is represented by curly braces.

Subsystem



The dashed arrow sign is used to show the dependency among two elements or two packages.

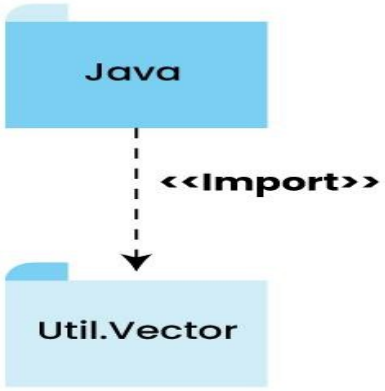
Dependency



Fish is Dependant on Water

Import & Merge

import



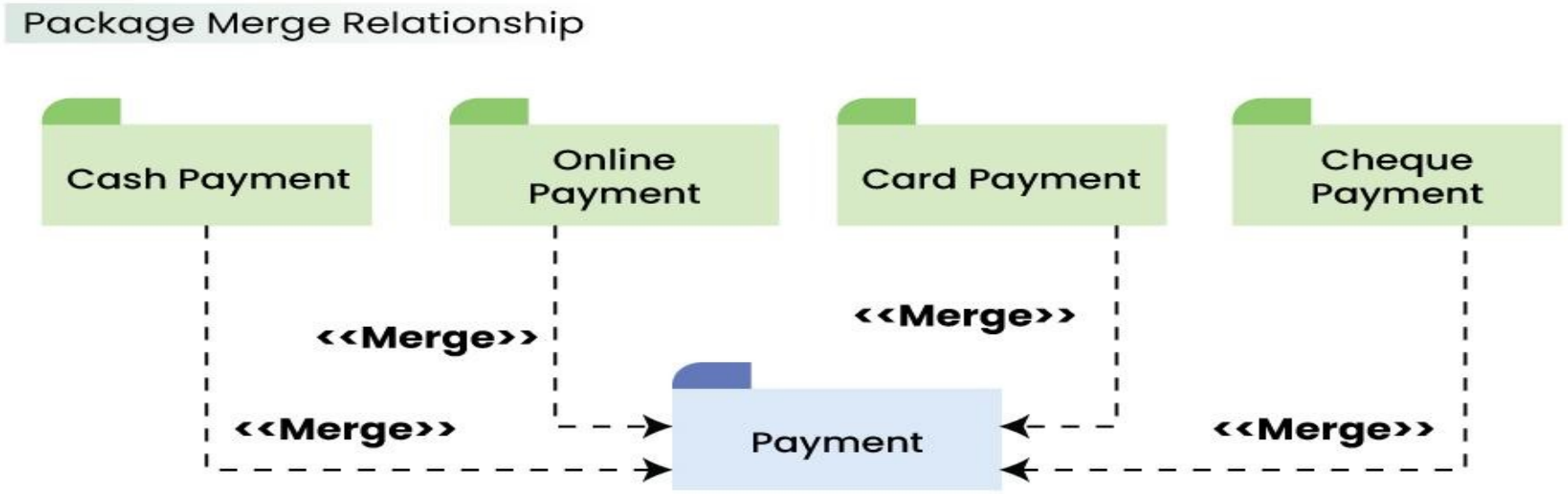
Package Diagram

Merge

Package Diagram

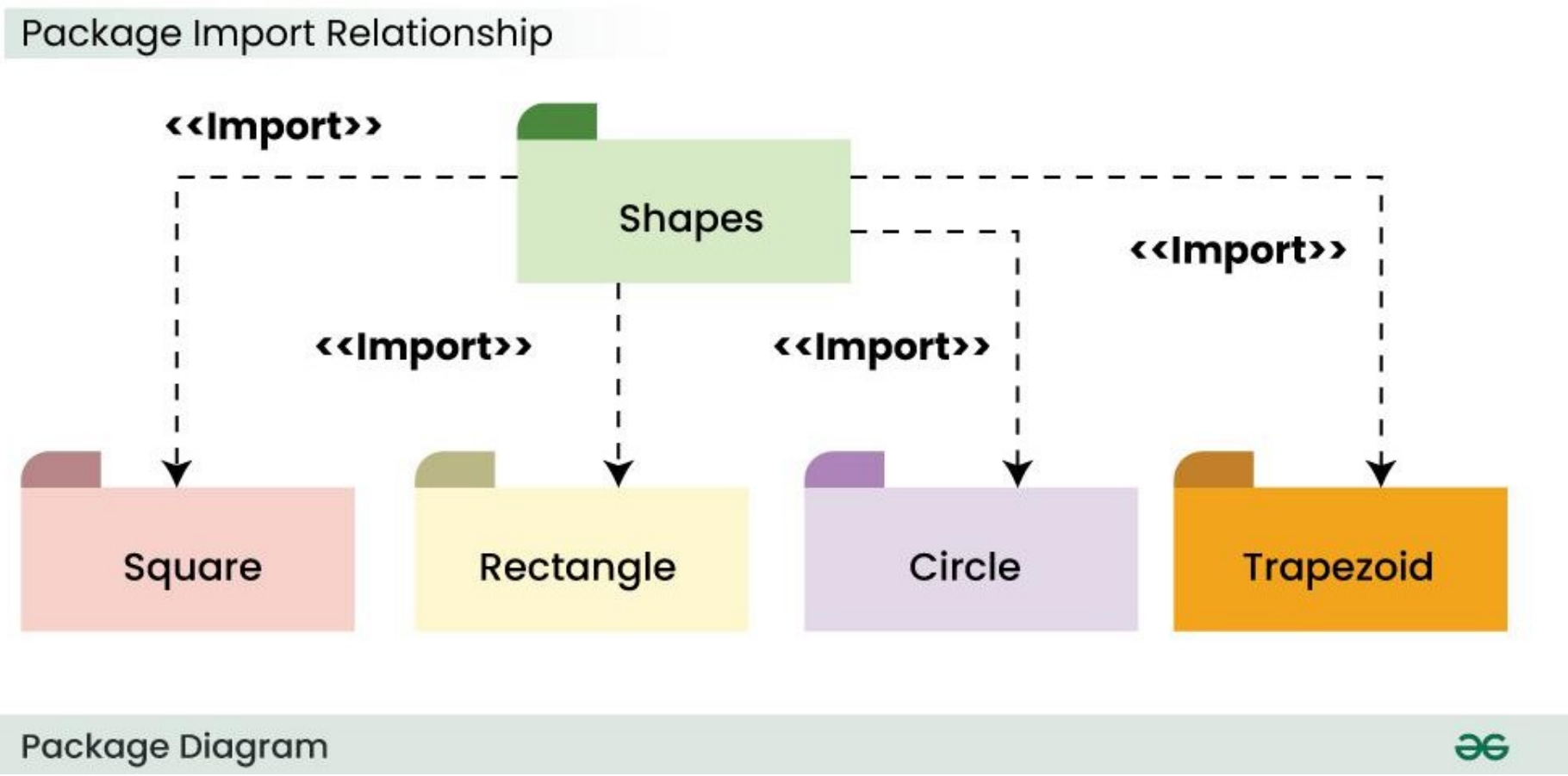
Package Relationships

Used to represent that the contents of a **package can be merged with the contents of another package**. This implies that the **source and the target package has some elements common** in them.



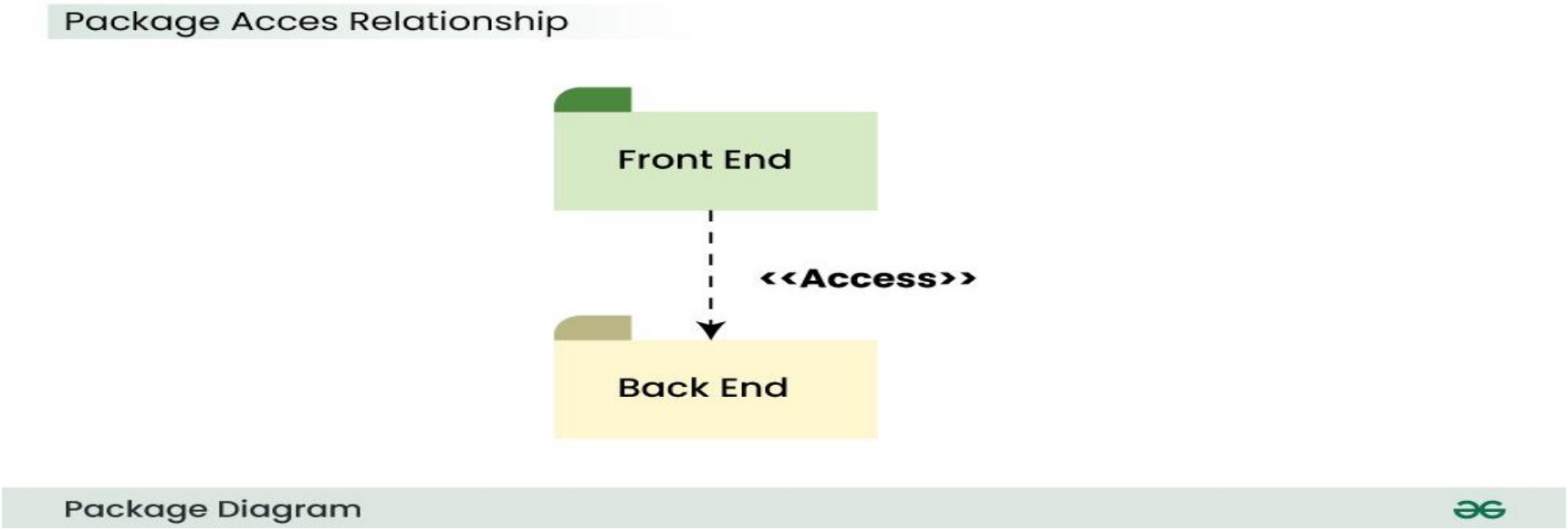
Package Import Relationship

- This relationship is used to represent that a package is **importing another package to use**.
- It signifies that the importing package can access the **public contents of the imported package**.



Package Access Relationship

- This type of relationship signifies that there is a **access relationship between two or more packages,** meaning that **one package can access the contents of another package without importing** it.



Use Cases of Package Diagrams

System Structure Visualization: Package Diagram is helpful to represent the system's structure in a diagrammatic format.

- It also helps in organizing the elements into smaller and compact packages.
- This type of visualization help in understanding the structure of the system in a more concise manner.

Module and Component Managemen:

- Package Diagram is useful to organize and manage modules or components within a system.
- Package acts as a container for items which are related to each other. It helps in grouping and categorizing parts of the system.

Dependency Managemen:

- Developers use Package Diagram to represent dependency between the different packages, this is useful to represent the system's architecture neatly and show the potential impact of the changes..

System Decomposition:

- System Architects tend to break down a complex problem into a group of smaller and easily managable problem at the initial stage of the System Design.
- This helps in designing the components easier and easier implementation.

Versioning and Release Planning:

- Project Managers often use Package Diagram to plan release of the product, ensuring that all the newly added components and changes are well understood and coordinated

Package Diagram Best Practices

Clearly Define the Package Names:

- It is always recommended to clearly define the package names in such a manner that there is no duplicate present in the same package which might create confusion.

Organize Packages Hierarchically:

- Organize the packages in a hierarchical manner so that it represents the **proper structure of the system and the relationship** of the various parts of the system.

Maintain Modularity:

- Make the packages concise, use a single package to represent a single function or element.
- Don’t use overcomplicated and complex packages.

Document Dependencies:
Clearly mention and document the dependencies between the packages using the suitable symbols .
This includes dependencies such as associations, generalizations, or dependencies.
Use Colors and Styles Sparingly:
Use different colors and styles to differentiate between different packages, but ensure the used colors are meaningful and consistent.

Benefits of Package Diagram

Clarity and Understanding: It provides a visual representation of the system's architecture, showing that how each elements are organized and interact with each other.
Modularity and Encapsulation: Package Diagram encourages a modular approach to represent any system in form of smaller and easy to understand packages. It also supports the encapsulation of elements into packages which share a same trait.

Communication and Collaboration:

- It servers as a common language of communication between the developers and the stakeholders.

Dependency Identification:

- By showing the dependency between the different packages, it become easy to identify and manage dependencies between the packages, also it becomes easy to address issues related to coupling and cohesion.

Scalability and Maintainability:

- Package Diagram highly encourages Scalability and Maintainability
- Any package can be changed or modified independently

Challenges of Package Diagrams

- **Overly Detailed or Abstract:**
- Keeping the balance between providing enough detail and maintaining simplicity and abstraction can be challenging task.
- Any single package of a Package Diagram must not be overwhelmed with information so that it becomes complex to comprehend.
- **Dynamic Aspects Missing:**
- The main focus of Package Diagram is **Static Structural** Aspects of elements of the system.
- It might not catch the **dynamic nature of the system** such as the runtime behaviors or interaction between different components of the system.

Overemphasis on Dependencies:

- While it is a must to **show the dependencies among the packages** in a Package Diagram, putting extra emphasis on them might **create clutter and complex diagram**, which becomes challenging to understand.
- Emphasis should be put on to **display the dependencies in a clear and concise manner** without making the diagram hard to understand

Limited Support for Behavioral Aspects:

- It is not well suited when it comes to dynamic or behavioral attributes of a system, like the change of behavior of the system during its runtime, or the interaction between different packages during the runtime.

Pitfalls of Package Diagrams

- **Misinterpretation of Relationships:**
- If the relationships are not defined or represented correctly, some misinterpretation might arise regarding them.
- This misinterpretation might lead into incorrect assumptions about the system's architecture and behavior.
- **Inconsistent Naming Conventions:**
- Giving unappropriate name to packages which might lead into confusion regarding the system.
- It also create confusion amongst the team members, as they do not understand the purpose each package due to their inconsistent name.

- **Ignoring Updates / Changes:**
- If the package diagram is not updated accordingly with the changes made in the system, then it will become outdated and useless, eventually it will lost its significance as a visualization and documentation tool.
- **Lack of involvements from Stakeholders:**
- If the stakeholders don't provide their input while making of the package diagram, it might not align with their need and actual requirement of the project.

Package Diagrams in Software Development

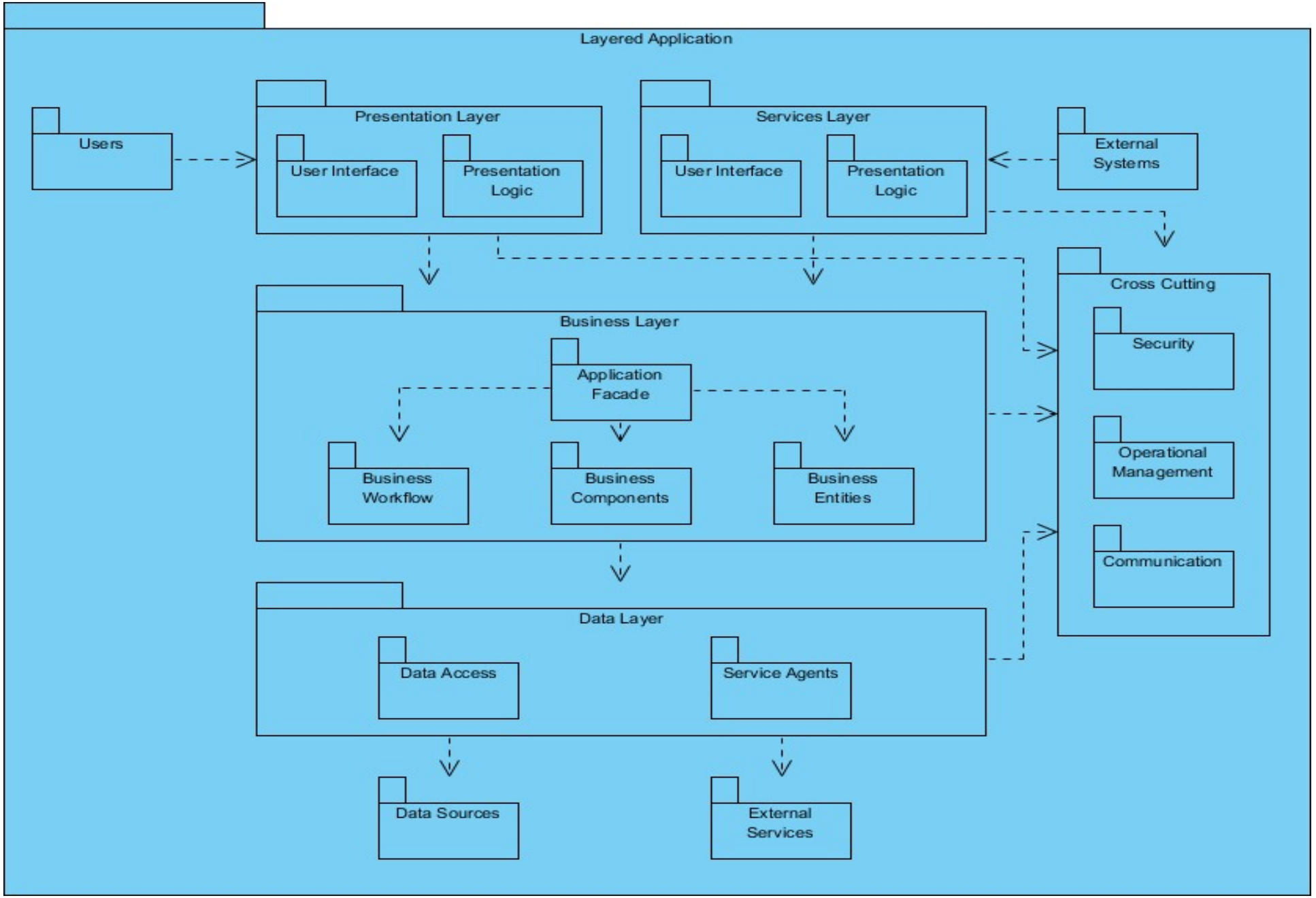
- Package Diagram plays a **significant role in the field of Software Development as a visual representation** of an architecture of a system.
- It helps in **understanding , organizing and communication of the structural component of a system**

- Package Diagram provides a **high level view of the system's architecture by grouping related elements together to form packages**, and representing the dependencies and interactions with each other.
- Package Diagram **promotes modularity and encapsulation**, which allows the developers to create software with clear boundaries among its components.
- By visually **representing classes, interfaces, functions and other elements in a concise manner** by including them in a package, software developers can manage the complexity of large software/systems.
- Package Diagram also used as a **communication tool between the team members and the stakeholders**, as it becomes easy to show the internal architecture of the system and provide a detail view of it, as it assist in finding the dependencies between the elements of the software, it also helps in terms of the maintenance of the system.

Tools for Creating Package Diagram

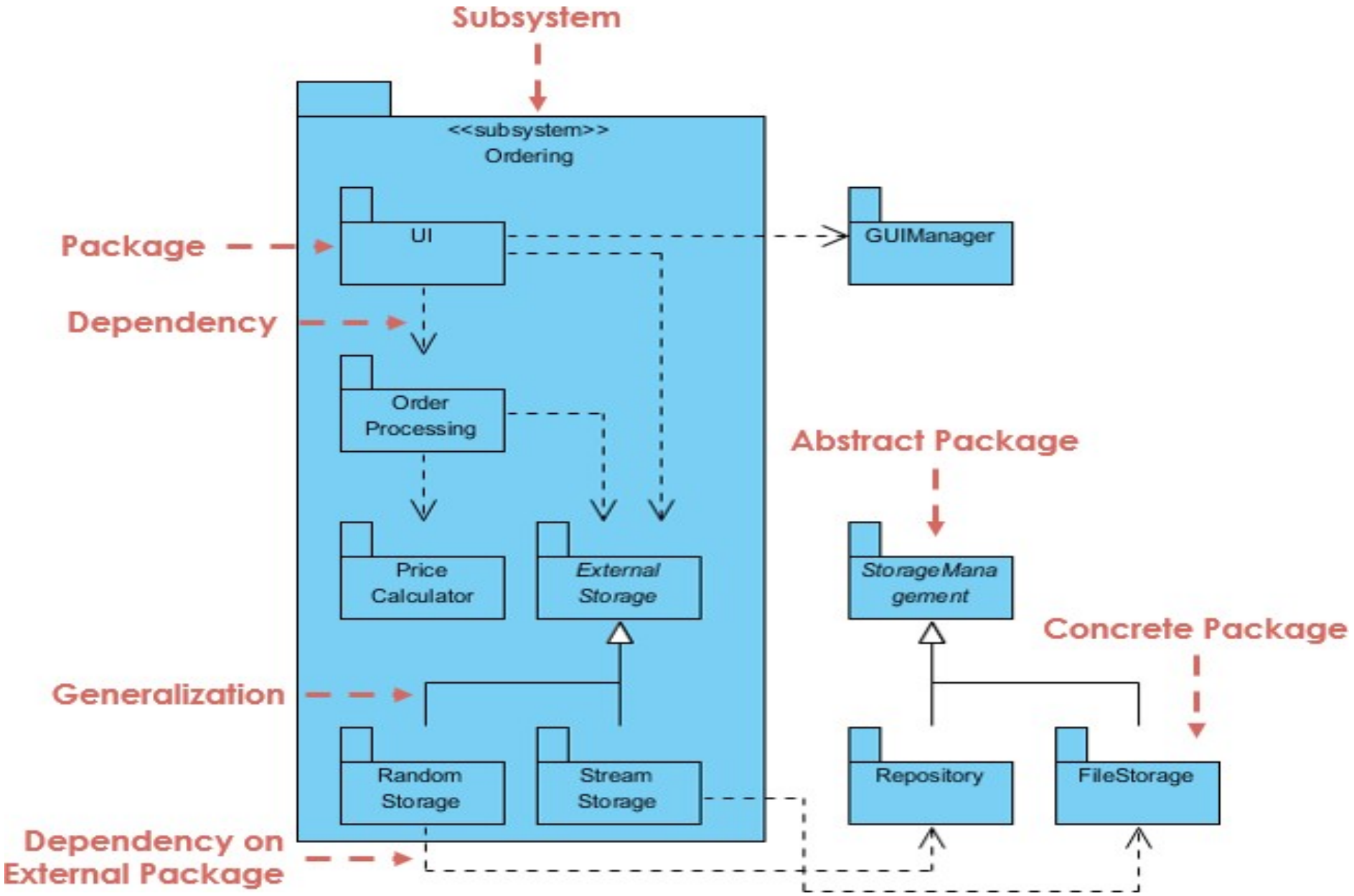
- Enterprise Architect
- Lucidchart
- Visual Paradigm
- Draw.io
- IBM Rhapsody
- PlantUML
- StarUML
- Creately
- Microsoft Visio

Modeling Complex Grouping

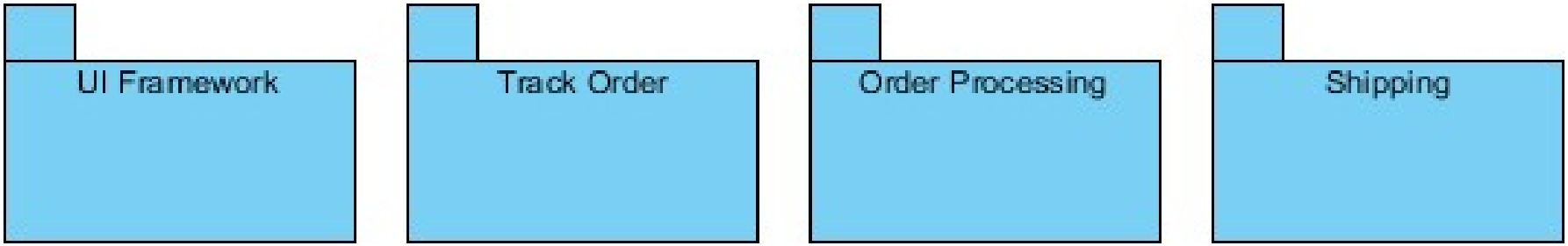


- A package diagram is often used to describe the hierarchical relationships (groupings) between packages and other packages or objects.
- A package represents a namespace.

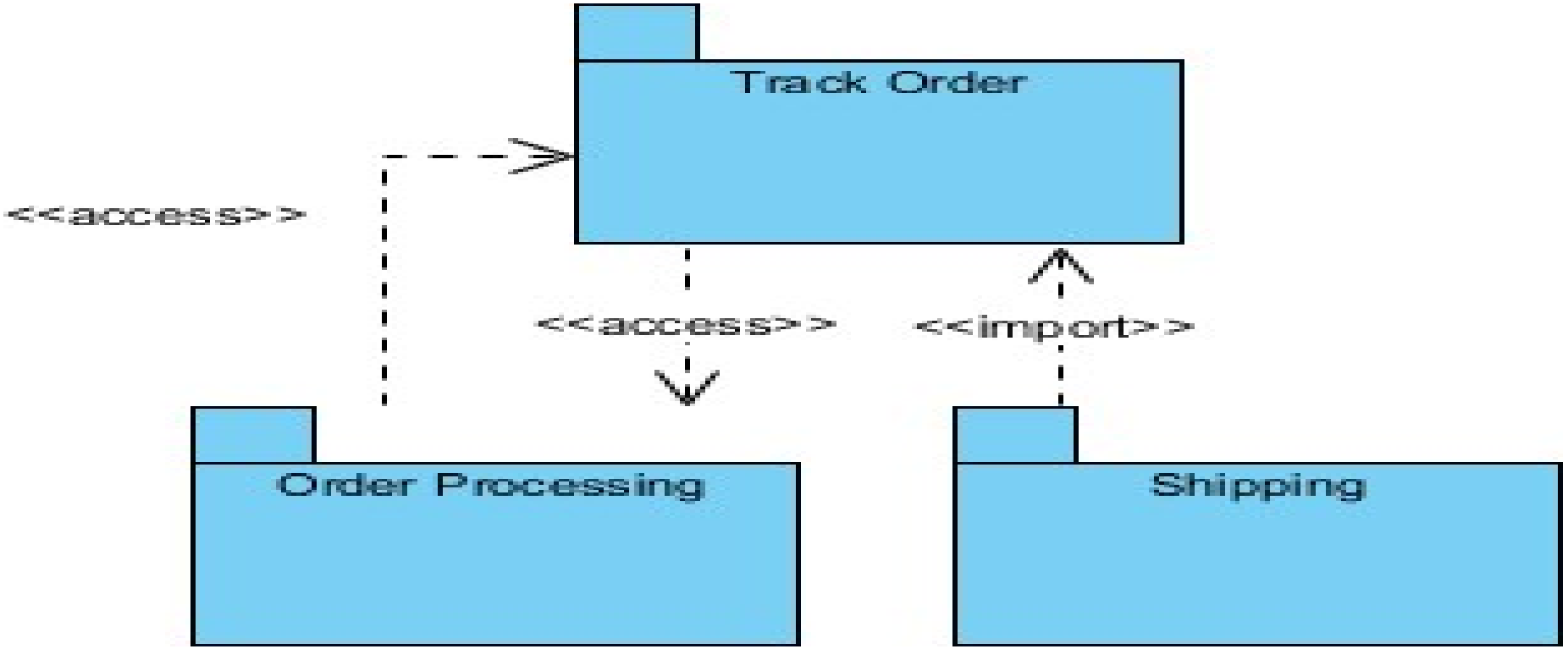
Package - Order SubSystem



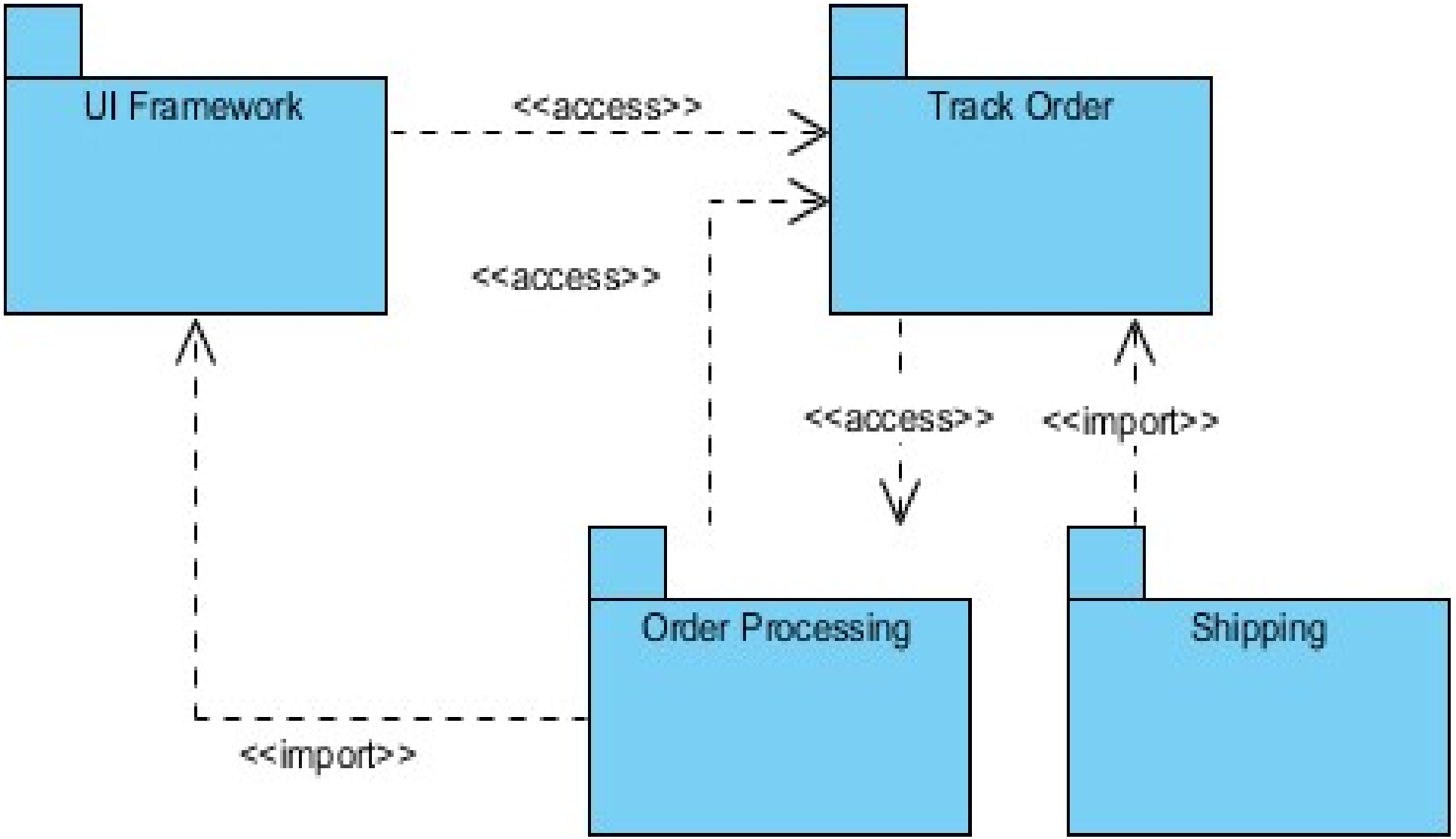
Package Order Processing System



Identify the dependencies in the System



Track Order dependency to UI Framework



Behavioral Diagram - Activity Diagram

- Activity diagrams describe the workflow behavior of a system.
  - Activity diagrams are used in **process modeling and analysis of during requirements engineering**.
  - A typical business process which **synchronizes several external incoming events** can be represented by activity diagrams.
- They are most useful for understanding work flow analysis of synchronous behaviors across a process.

- UML Activity Diagram is a type of Behavior Diagrams that graphically describes decomposition of the some activity on the components.

- Activity Diagrams are used in modeling of **business processes, technological processes, sequential and parallel computations.**
- UML Activity Diagram allows to show the **sequence, branching and synchronization of processes.**

Activity Diagram

- Activity diagrams are used for
  - documenting existing process
  - analyzing new Process Concepts
  - finding reengineering opportunities.

- The diagrams describe the state of activities by showing the sequence of activities performed.
  - they can show activities that are conditional or parallel.

- To design UML Activity Diagrams use the following shape types:
- **rounded rectangles** to describe the actions;
- **diamonds** to describe decisions;
- **bars** to represent the start or end of the activities that occur at the same time;
- **black circle** to indicate the start of the workflow;
- **encircled black circle** to indicate the end of the workflow;
- **arrows** to represent the order in which activities happen.

Activity Diagram Concepts

- An activity is trigged by one or more events and activity may result in one or more events that may trigger other activity or processes.
- Events start from start symbol and end with finish marker having activities in between connected by events.
- The activity diagram represents the decisions, iterations and parallel/random behavior of the processing.
  - **They capture actions performed.**
  - **They stress on work performed in operations (methods).**

UML Activity Diagram library contains 37 shapes:

- Object
- Data store
- Central buffer
- Expansion region
- Control flow (direct)
- Control flow (smart)
- Object flow (direct)
- Object flow (smart)
- Divider

- Decision/Merge
- Time event action
- Send signal node
- Receive signal node
- Expansion node
- Pin
- Vertical fork/join
- Horizontal fork/join
- Vertical swimlane
- Horizontal swimlane
- Connector
- Initial node
- Final node
- Flow final

- Control flow
- Condition
- Symbol { }
- Symbol << >>
- Subactivity state
- Call behavior
- Frame, fragment
- Note
- Action
- Swimlanes (vertical)
- Swimlanes (horizontal)
- UML connector (direct)
- UML connector
- Note connector

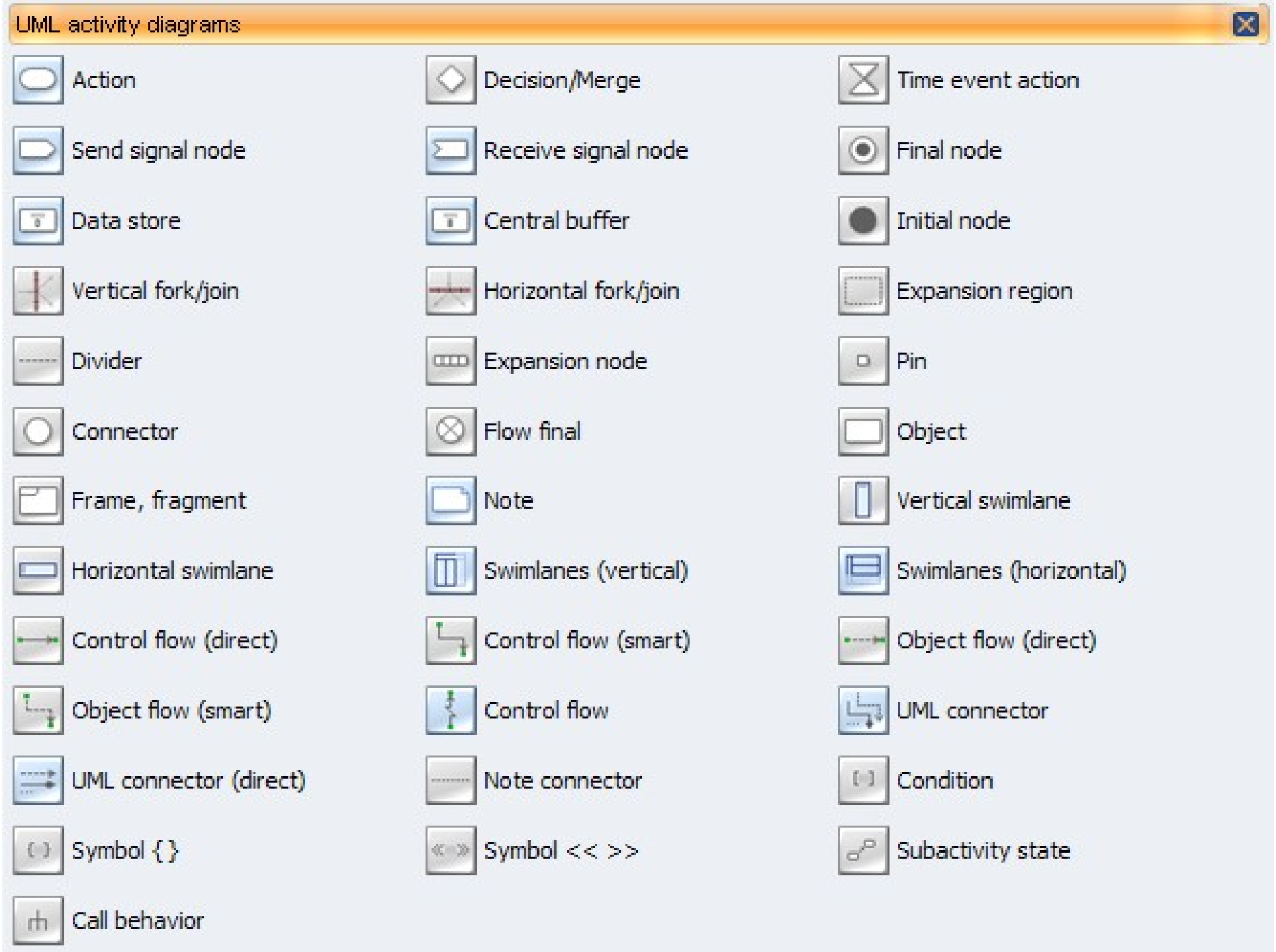
Components

- An *activity* is an ongoing, though interruptible, execution of a step in a workflow (such as an operation or transaction)

  – Represented with a rounded rectangle.
  – Text in the activity box should represent an activity (verb phrase in present tense).

Components

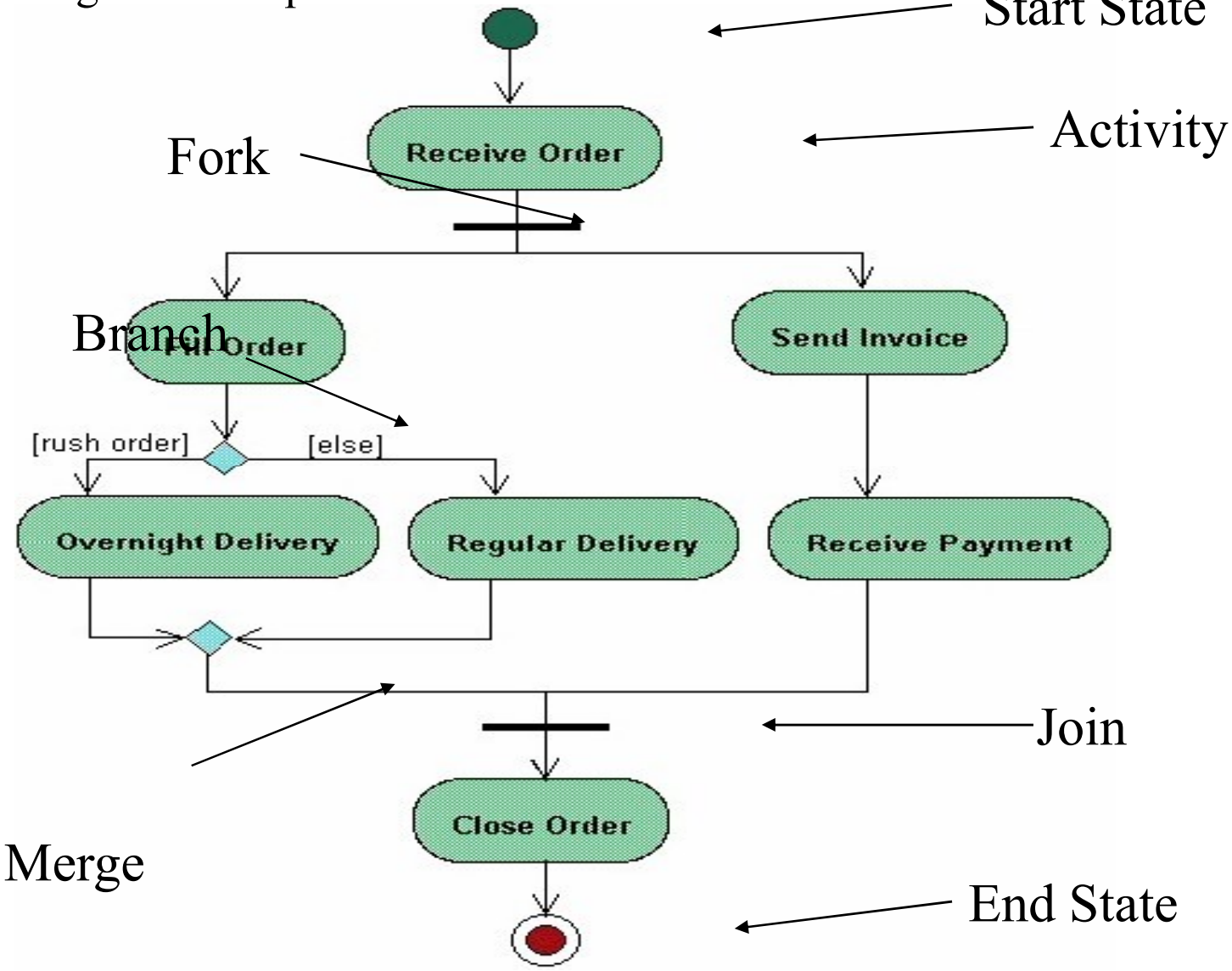
- An *event* is triggered by an activity. It specifies a significant occurrence that has a location in time and space.
  - An instance of an event (trigger) results in the flow from one activity to another.
  - These are represented by directed straight lines emerging from triggering activity and ending at activity to be triggered. Label text for events should represent event but not the data involved.
- A *decision* may be shown by labeling multiple output transitions of an activity with different guard conditions.
  - For convenience a stereotype is provided for a decision: the traditional diamond shape, with one or more incoming arrows and with two or more outgoing arrows, each labeled by a distinct guard condition with no event trigger.


UML activity diagrams
Action
Decision/Merge
Time event action
Send signal node
Receive signal node
Final node
Data store
Central buffer
Initial node
Vertical fork/join
Horizontal fork/join
Expansion region
Divider
Expansion node
Pin
Connector
Flow final
Object
Frame, fragment
Note
Vertical swimlane
Horizontal swimlane
Swimlanes (vertical)
Swimlanes (horizontal)
Control flow (direct)
Control flow (smart)
Object flow (direct)
Object flow (smart)
Control flow
UML connector
UML connector (direct)
Note connector
Condition
Symbol { }
Symbol << >>
Subactivity state
Call behavior

How to Draw an Activity Diagram

- Diagrams are read from top to bottom and have branches and forks to describe conditions and parallel activities.
  - A fork is used when multiple activities are occurring at the same time.
  - A branch describes what activities will take place based on a set of conditions.
  - All branches at some point are followed by a **merge** to indicate the end of the conditional behavior started by that branch.
  - After the merge all of the parallel activities must be combined by a **join** before transitioning into the final activity state.

Activity Diagram Example



When to Use Activity Diagrams

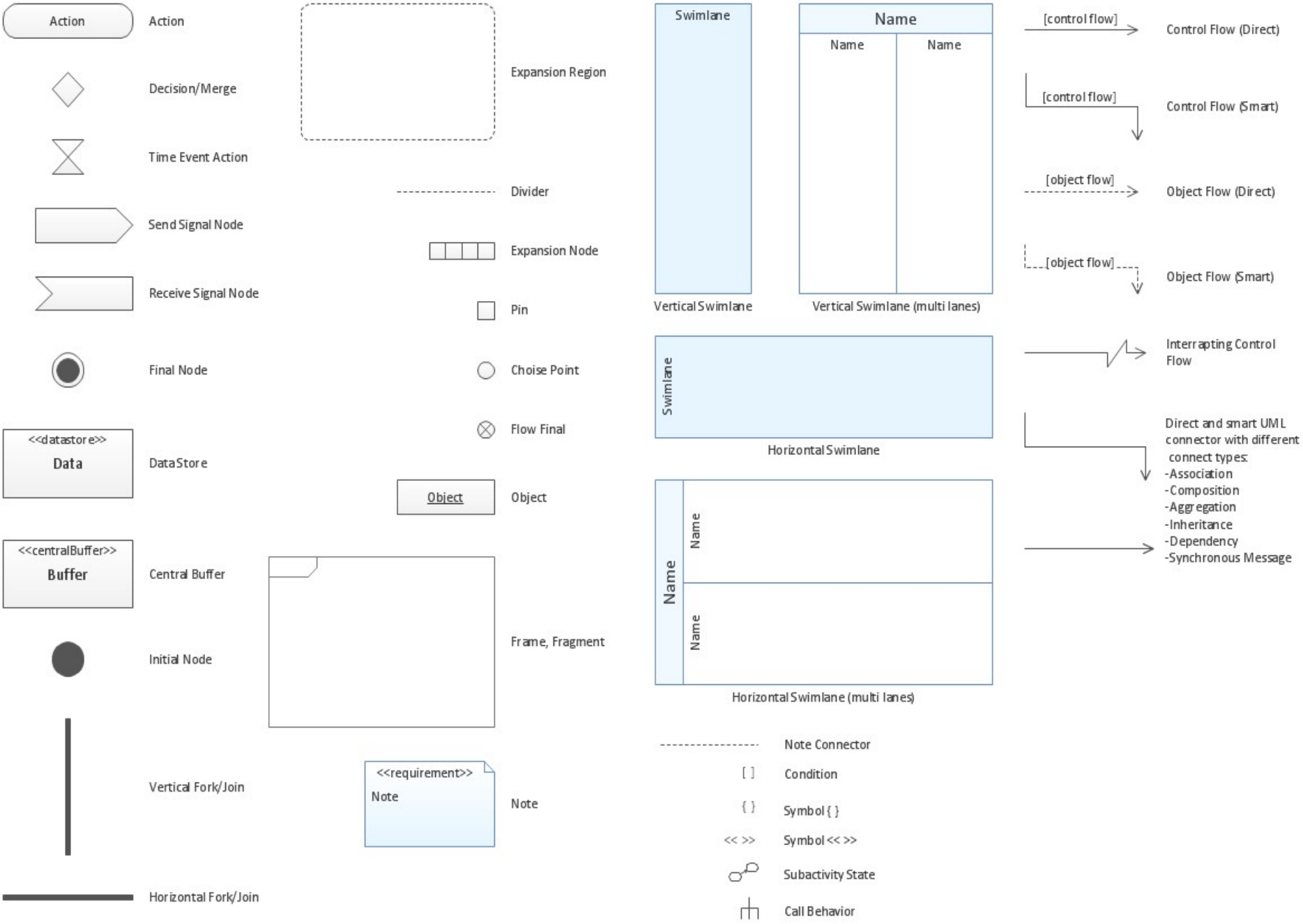
- **The main reason to use activity diagrams is to model the workflow behind the system being designed.**

- **Activity Diagrams are also useful for:**
  - **analyzing a use case by describing what actions need to take place and when they should occur**
  - **describing a complicated sequential algorithm**
  - **modeling applications with parallel processes**

- **Activity Diagrams should not take the place of interaction diagrams and state diagrams.**

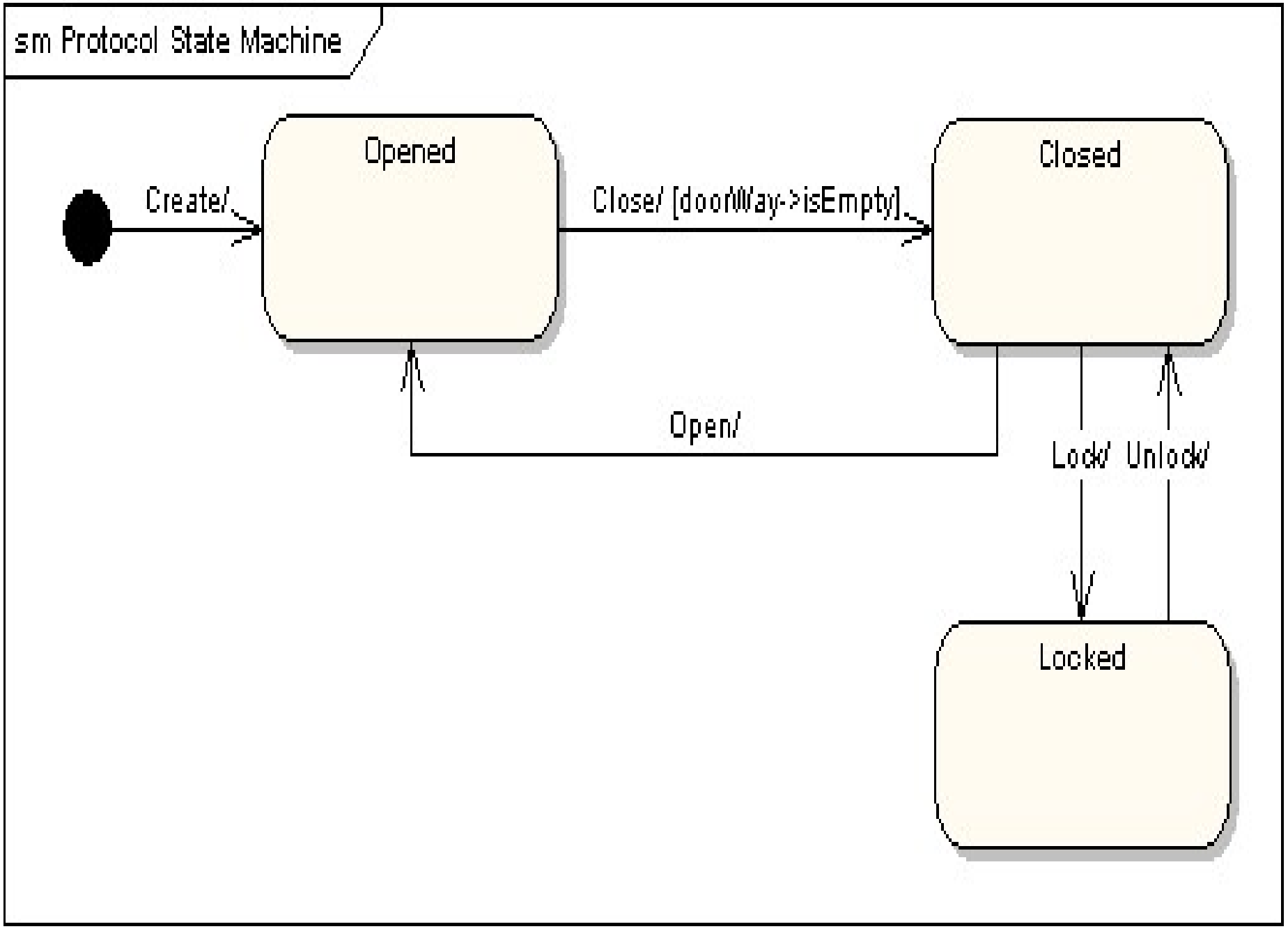
- **Activity diagrams do not give detail about how objects behave or how objects collaborate.**

UML Activity Diagram



State Machine Diagram

- State Machine diagram is also called the **State-chart or State Transition diagram**, which shows the **order of states** underwent by an **object** within the system. It provides a **visual representation of states, transitions, events, and actions.**
- It captures the software system's behavior.
- It models the **behavior of a class, a subsystem, a package and a complete system.**
- It models event-based systems to handle the state of an object. It also defines several distinct states of a component within the system.
- Each object/component has a **specific state**.


sm Protocol State Machine
Opened
Closed
Create/
Close/ [doorWay->isEmpty]
Open/
Lock/
Unlock/
Locked

- The door can be in one of three states: "Opened", "Closed" or "Locked".
- It can respond to the events Open, Close, Lock and Unlock.
- **Not all events are valid in all states**;
- for example, if a door is opened, you cannot lock it until you close it.
- state transition can have a **guard condition attached**:

  if the door is Opened, it can only respond to the Close event if the condition

  doorWay->isEmpty is fulfilled.

Types

Behavioral state machine

The behavioral state machine diagram **records** the **behavior of an object within the system**.

It depicts an **implementation of a particular entity**. It models the **behavior of the system**.

Protocol state machine

It captures the **behavior of the protocol**. The protocol state machine depicts the change in the **state of the protocol and parallel changes within the system**.

But it does not portray the implementation of a particular component.

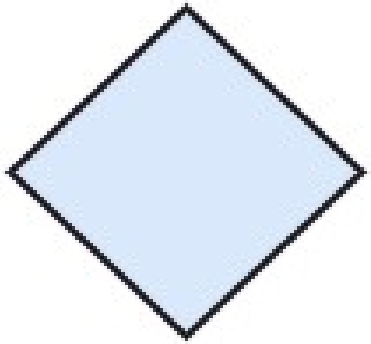
Why State Diagram

- During a lifespan, an object underwent several states, such that the lifespan exist until the program is executing.
- Each state depicts some useful information about the object.
- It blueprints an interactive system that response back to either the internal events or the external ones.
- The execution flow from **one state to another is represented by a state machine diagram**.
- It visualizes an object state from its creation to its termination.

Notations of State Diagram

Initial state

State-box

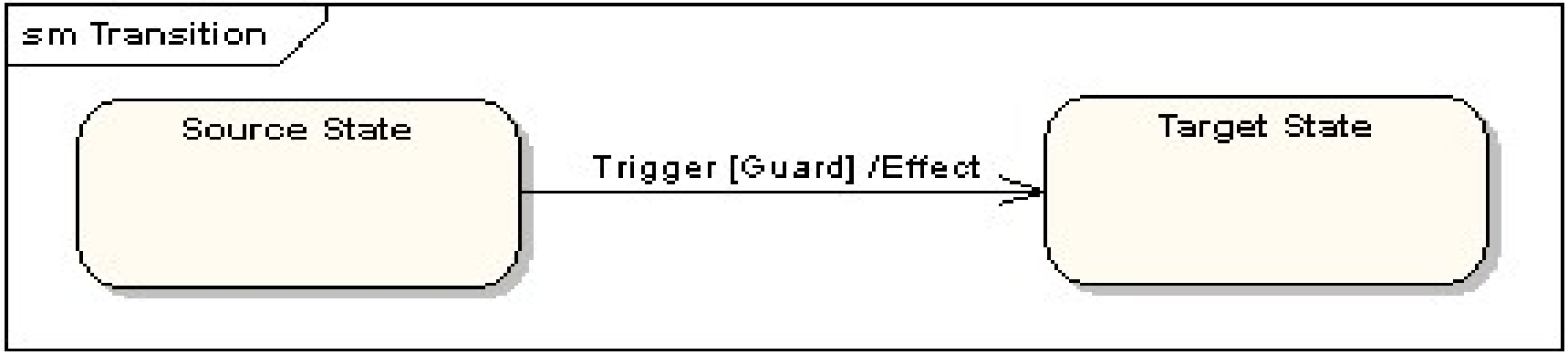


Decision-box

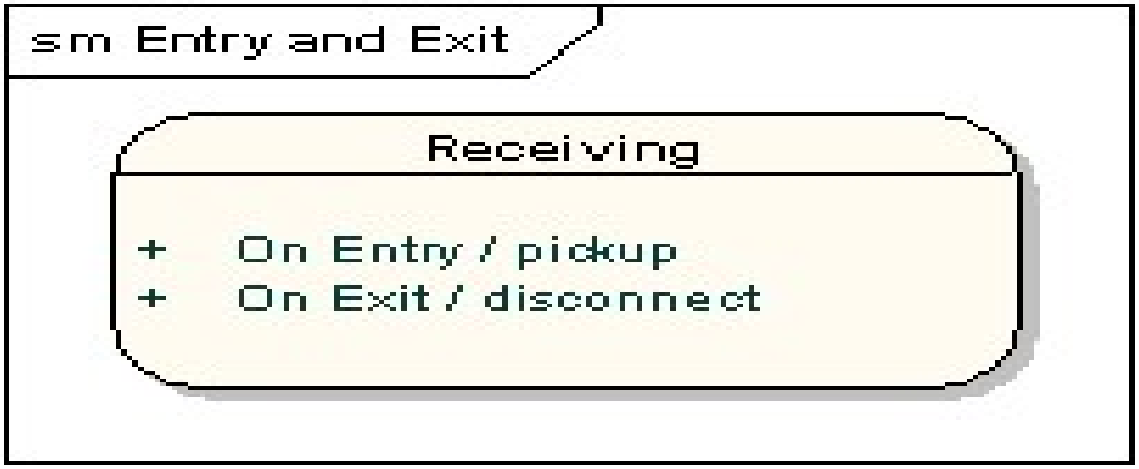
Final State

Transition

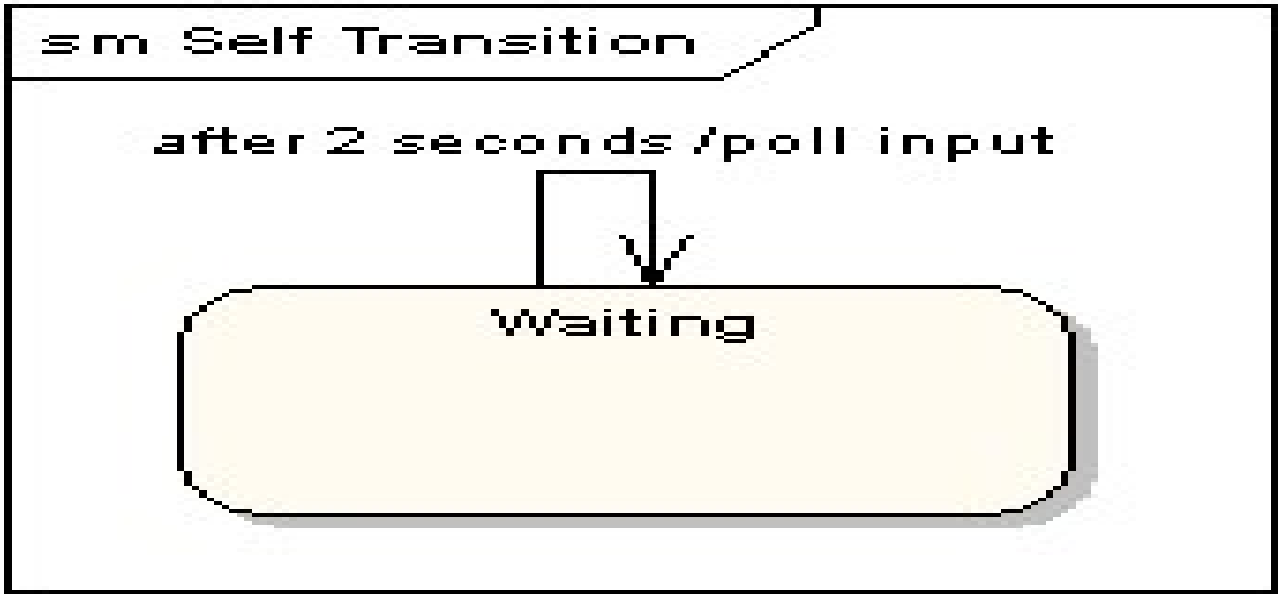
A transition may have a trigger, a guard and an effect



- State Action



Self Transition



Types of State

- **Simple state:** It does not constitute any substructure.
- **Composite state:** It consists of nested states (substates), such that it does not contain more than one initial state and one final state. It can be nested to any level.
- **Submachine state:** The submachine state is semantically identical to the composite state, but it can be reused.

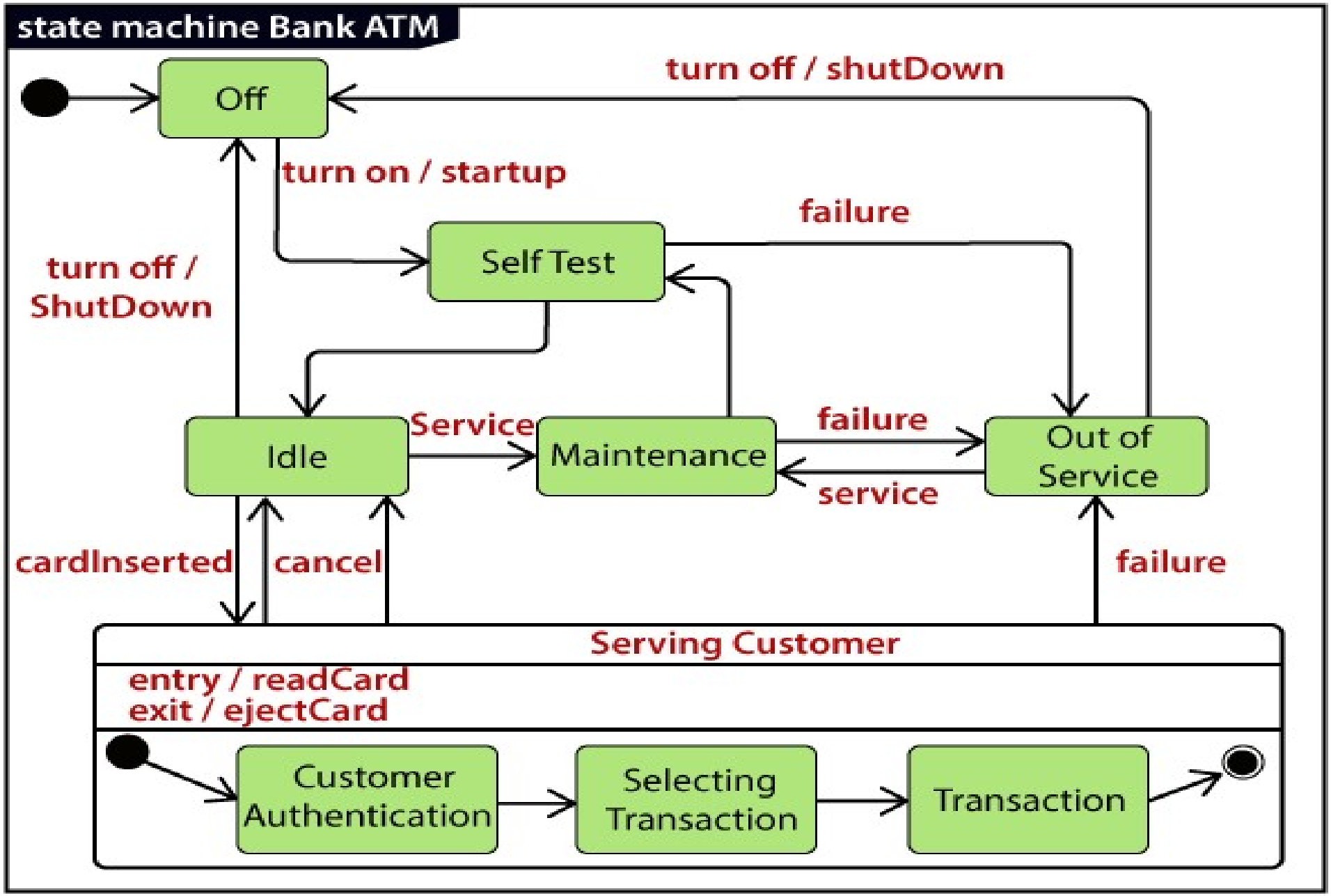
When to use a State Machine Diagram?

- The state machine diagram implements the real-world models as well as the object-oriented systems.
- It records the dynamic behavior of the system, which is used to differentiate between the dynamic and static behavior of a system.
- It portrays the changes underwent by an object from the start to the end.
- It basically envisions how triggering an event can cause a change within the system.

State machine diagram is used for:

- For **modeling the object states** of a system.
- For **modeling the reactive system** as it consists of reactive objects.
- For **pinpointing the events responsible for state transitions.**
- For **implementing forward and reverse engineering.**

State Machine Diagram - Example

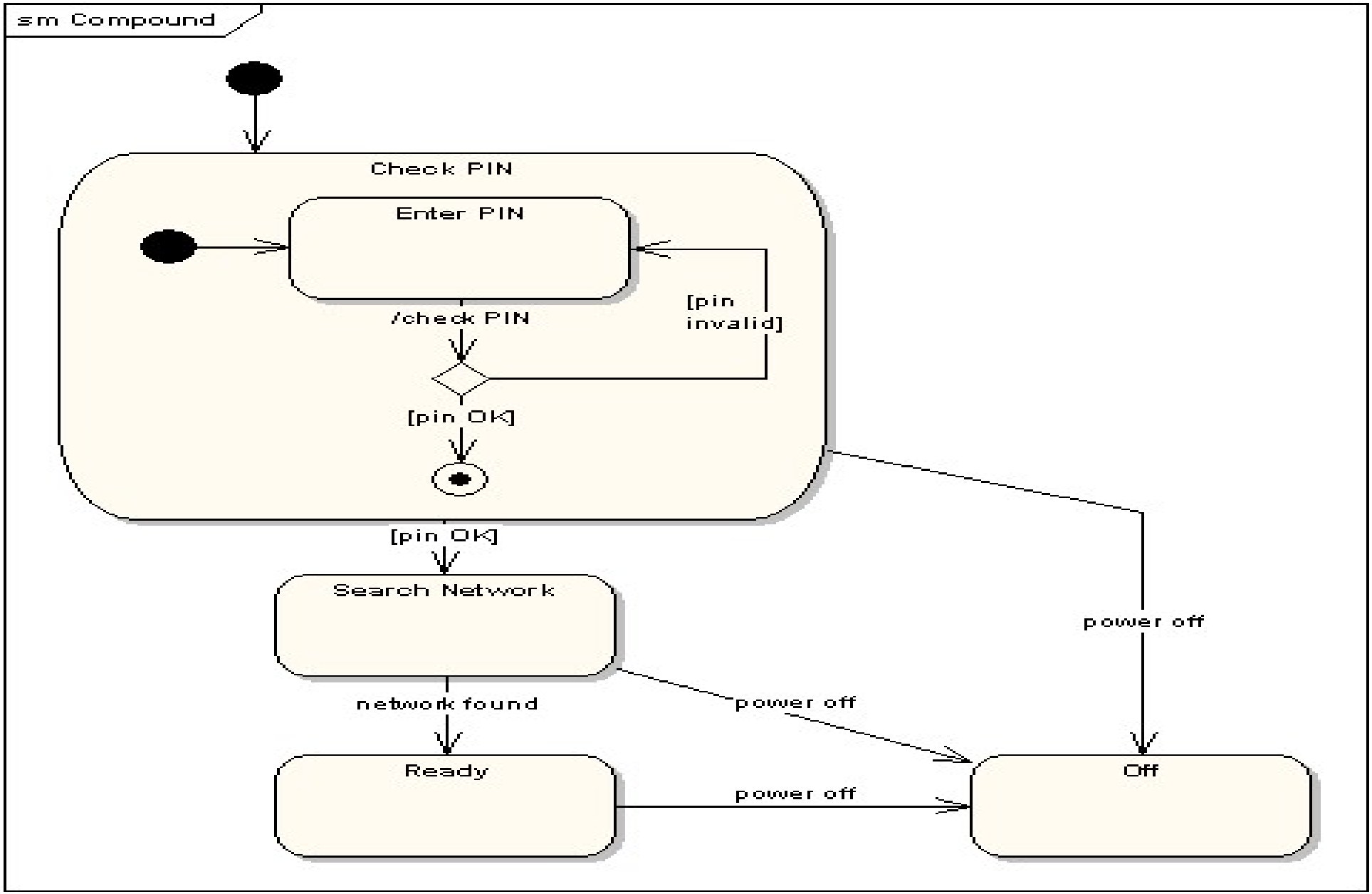


ATM Scenario- State Machine

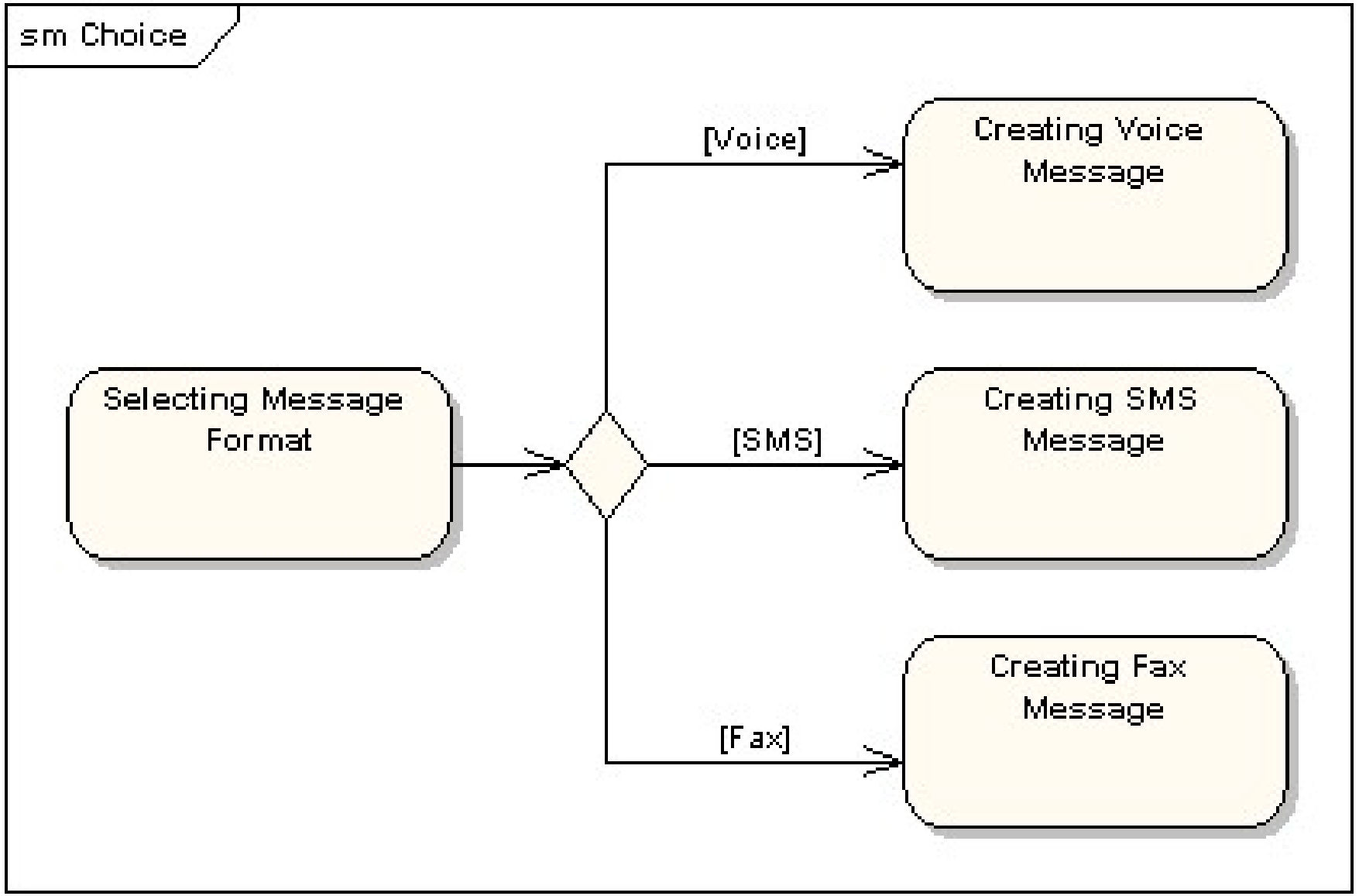
- After the power supply is turned on, the ATM starts performing the startup action and enters into the **Self Test** state.
- If the test fails, the ATM will enter into the **Out Of Service** state, or it will undergo **a triggerless transition** to the **Idle** state.
- This is the state where the customer waits for the interaction.

- Whenever the customer inserts the bank or credit card in the ATM's card reader,
- the ATM state changes from **Idle** to **Serving Customer**,
- the entry action **readCard** is performed after entering into **Serving Customer** state.
- Since the customer can cancel the transaction at any instant, so the transition from **Serving Customer** state back to the **Idle** state could be triggered by **cancel** event.

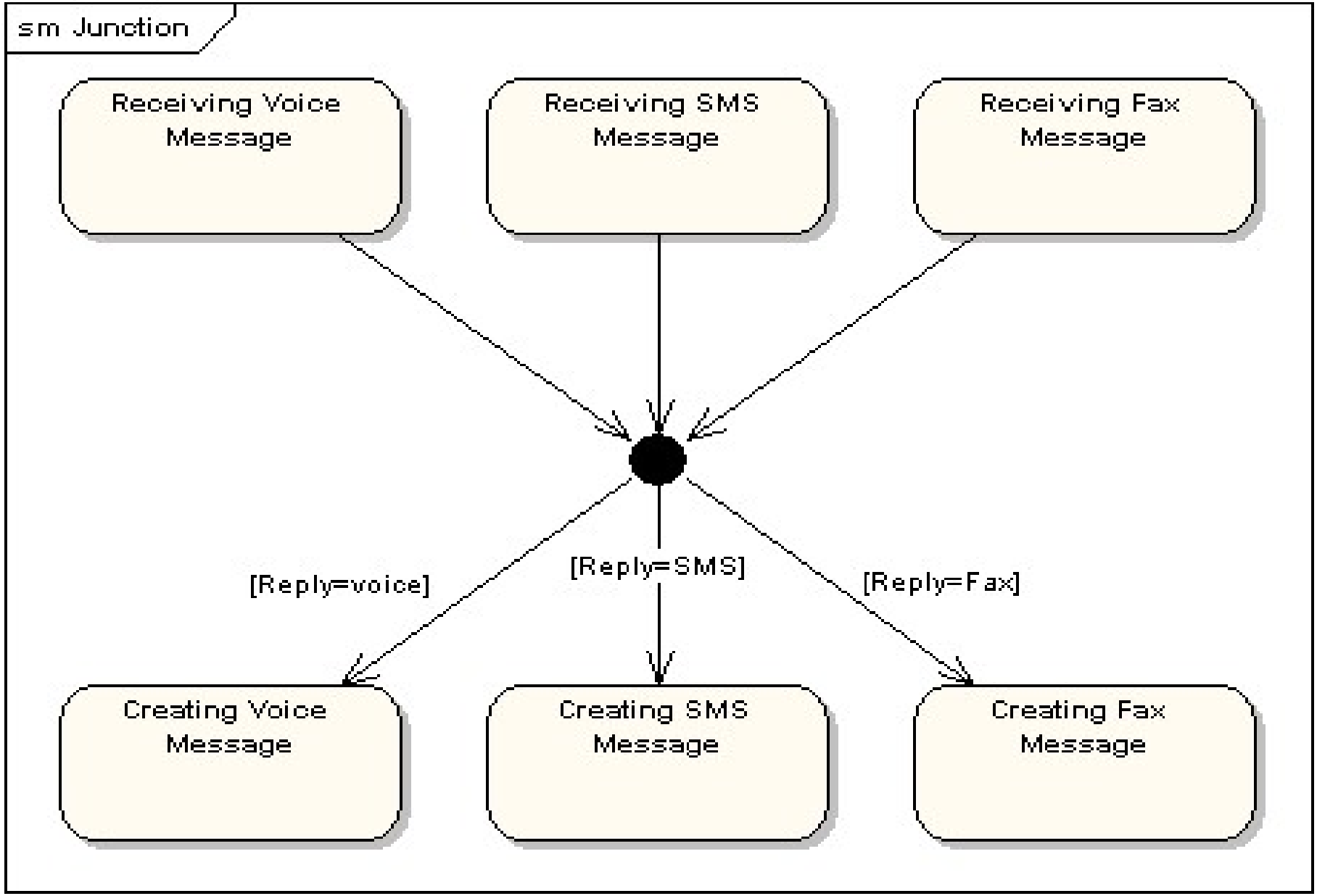
Compound States



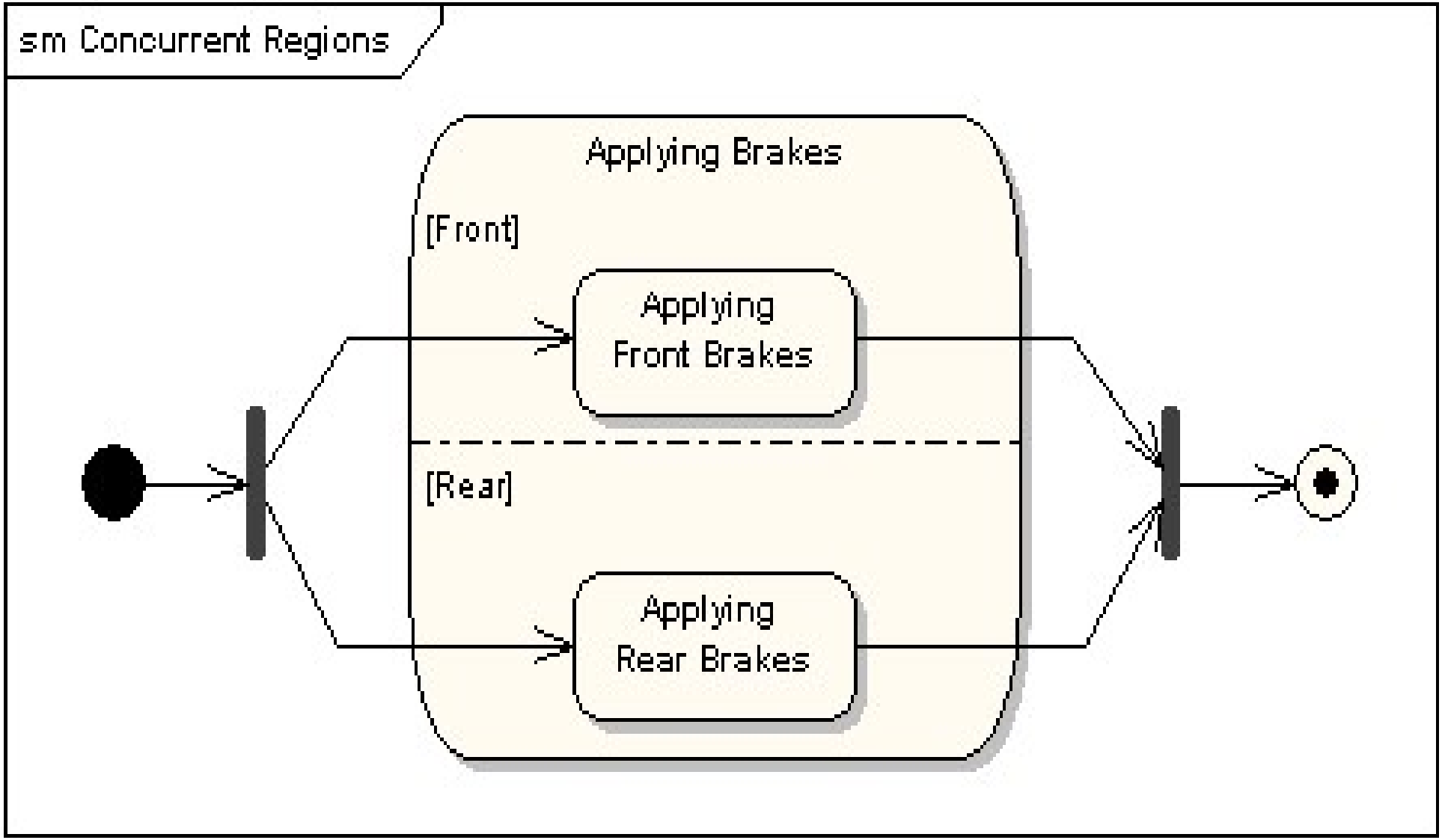
Choice PseudoStates



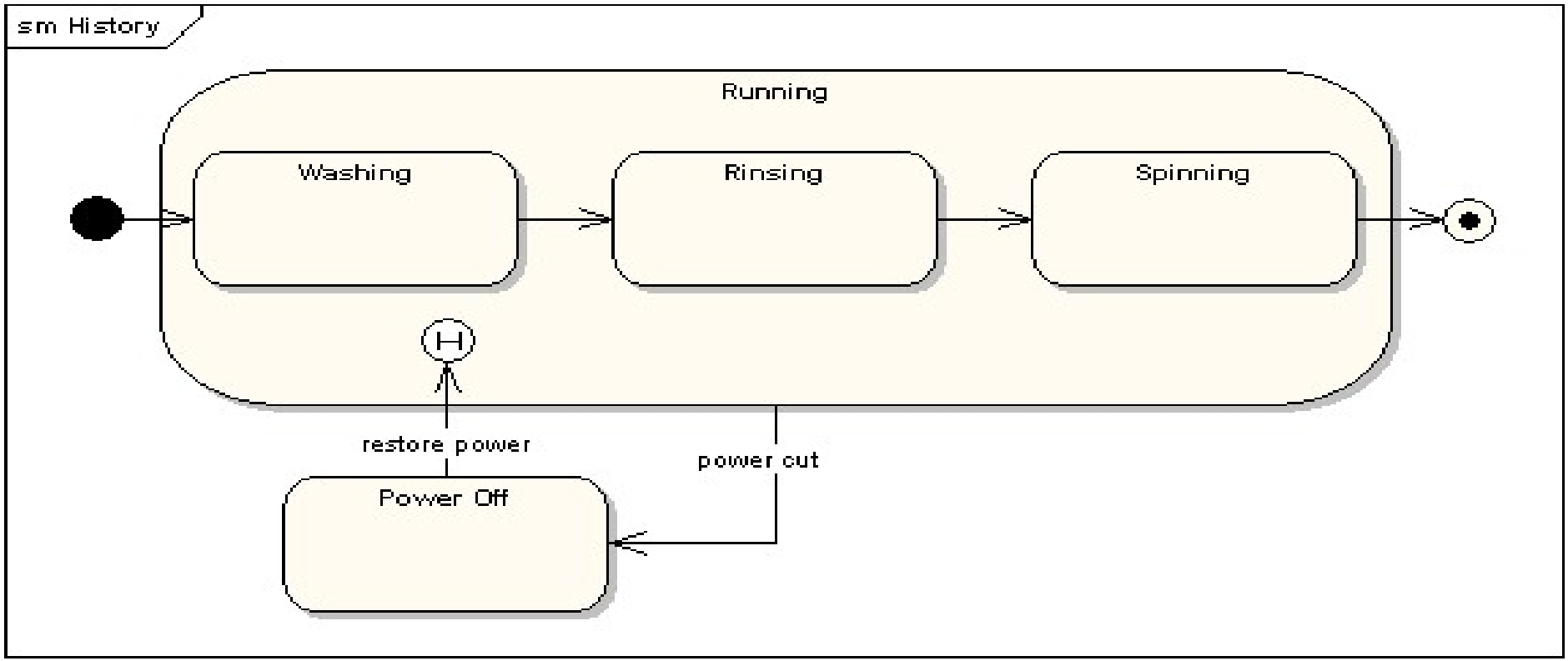
Junction Pseudo State



Concurrent Sub states / Regions

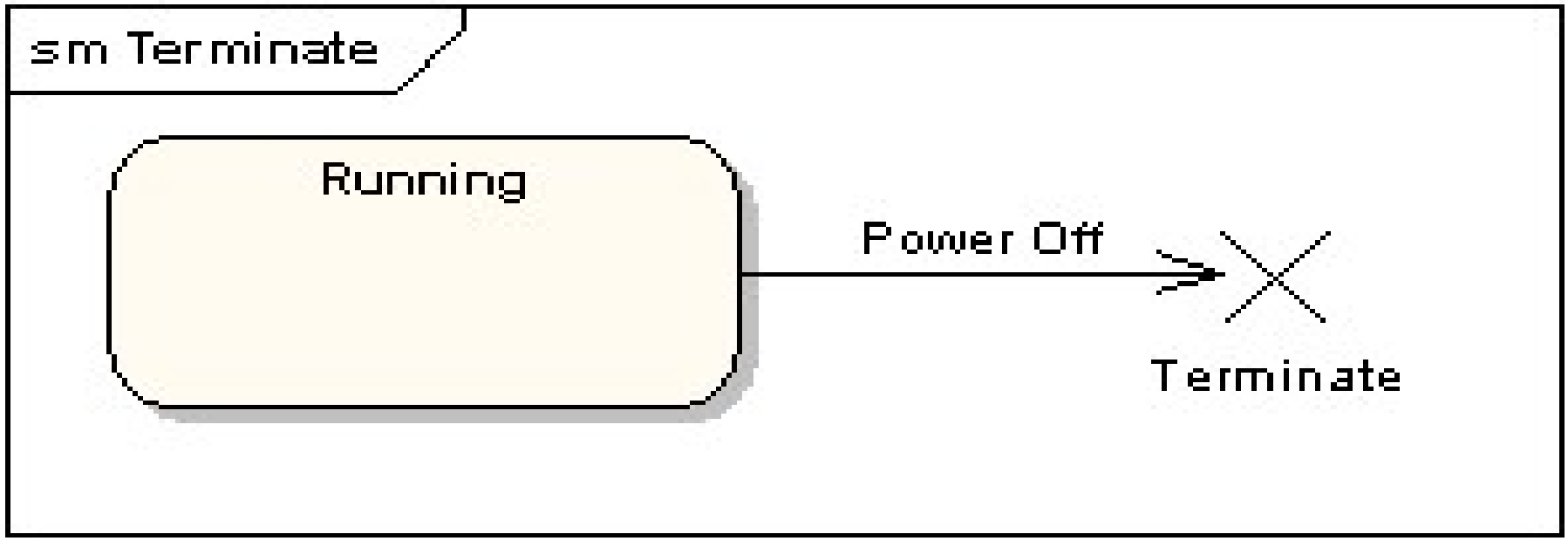


History State



A history state is used to remember the previous state of a state machine when it was interrupted.

Termination State



Purpose:

- **Activity Diagram**:
  - Describes the workflow or process in a system.
  - Models the flow of activities, focusing on what happens and the sequence in which these actions take place.
  - Useful for modeling business processes or flow of control between different tasks.
- **State Machine Diagram**:
  - Describes the states of an object and the transitions between these states based on events.
  - Focuses on how an object transitions from one state to another based on internal or external events.
  - Useful for modeling the life cycle of an object or a system.

Focus:

- **Activity Diagram**:
  - Focuses on the flow of activities and actions.
  - Describes how tasks are carried out and how they depend on each other.
- **State Machine Diagram**:
  - Focuses on the states of an object and how events cause transitions between those states.
  - Models the behavior of a single object across different conditions.

Activity Diagram: Focuses on **processes, workflows, and actions.**

State Machine Diagram: Focuses on the **states of an object and how it changes in response to events.**

Flow vs. State:

- **Activity Diagram**:
  - Represents a flow of control from one activity to another, usually from a starting point to an ending point.
  - Can have decision nodes, forks, and joins to represent branching or parallel processes.
- **State Machine Diagram**:
  - Represents discrete states of an object and the events or triggers that cause transitions between those states.
  - Transitions are triggered by events or conditions, and each state represents a situation or phase.

Use Case:

- **Activity Diagram**:
  - Suitable for representing business processes, workflows, and use case implementations.
  - Emphasizes actions performed by the system or objects.
- **State Machine Diagram**:
  - Suitable for modeling the states of objects and systems that have well-defined states and transitions (e.g., a vending machine, a user account system).
  - Emphasizes the state-dependent behavior of an object.

Concurrency:

- **Activity Diagram**:
  - Supports parallel flows of activities with **forks** and **joins**, representing concurrent activities.
- **State Machine Diagram**:
  - Typically models sequential transitions between states, though it can model concurrency in some cases using composite states.

Granularity:

- **Activity Diagram**:
  - Models the high-level workflow or algorithm of a system or process.
- **State Machine Diagram**:
  - Models the state transitions of individual objects, often at a finer level of detail compared to activity diagrams.

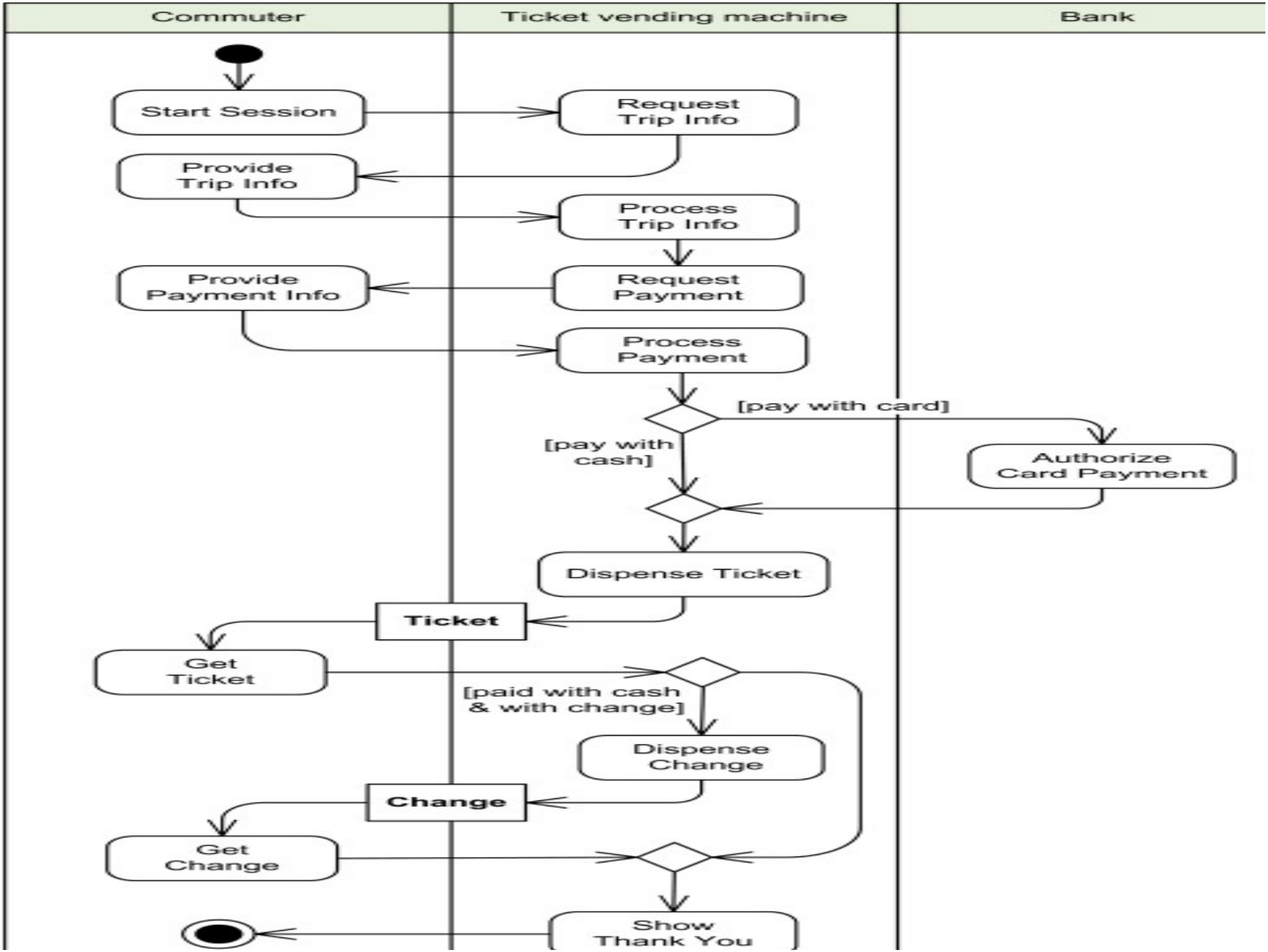
State Machine	Flowchart
It portrays several states of a system.	It demonstrates the execution flow of a program.
It encompasses the concept of WAIT, i.e., wait for an event or an action.	It does not constitute the concept of WAIT.
It is for real-world modeling systems.	It envisions the branching sequence of a system.
It is a modeling diagram.	It is a data flow diagram (DFD)
It is concerned with several states of a system.	It focuses on control flow and path.

Feature	UML Activity Diagram	Flowchart
Purpose	Software/system modeling	General process flow
Scope	Complex workflows with interactions	Simple, sequential workflows
Formality	Part of UML standard	Less formal
Concurrency	Supports concurrency	Sequential, no concurrency
Symbols	Specialized symbols (actions, nodes)	Generic symbols (rectangles, diamonds)
Audience	Software engineers, system architects	General audience, process analysts
Usage	Software workflows, system logic	Simple workflows, decision making

Ticket Vending Machine

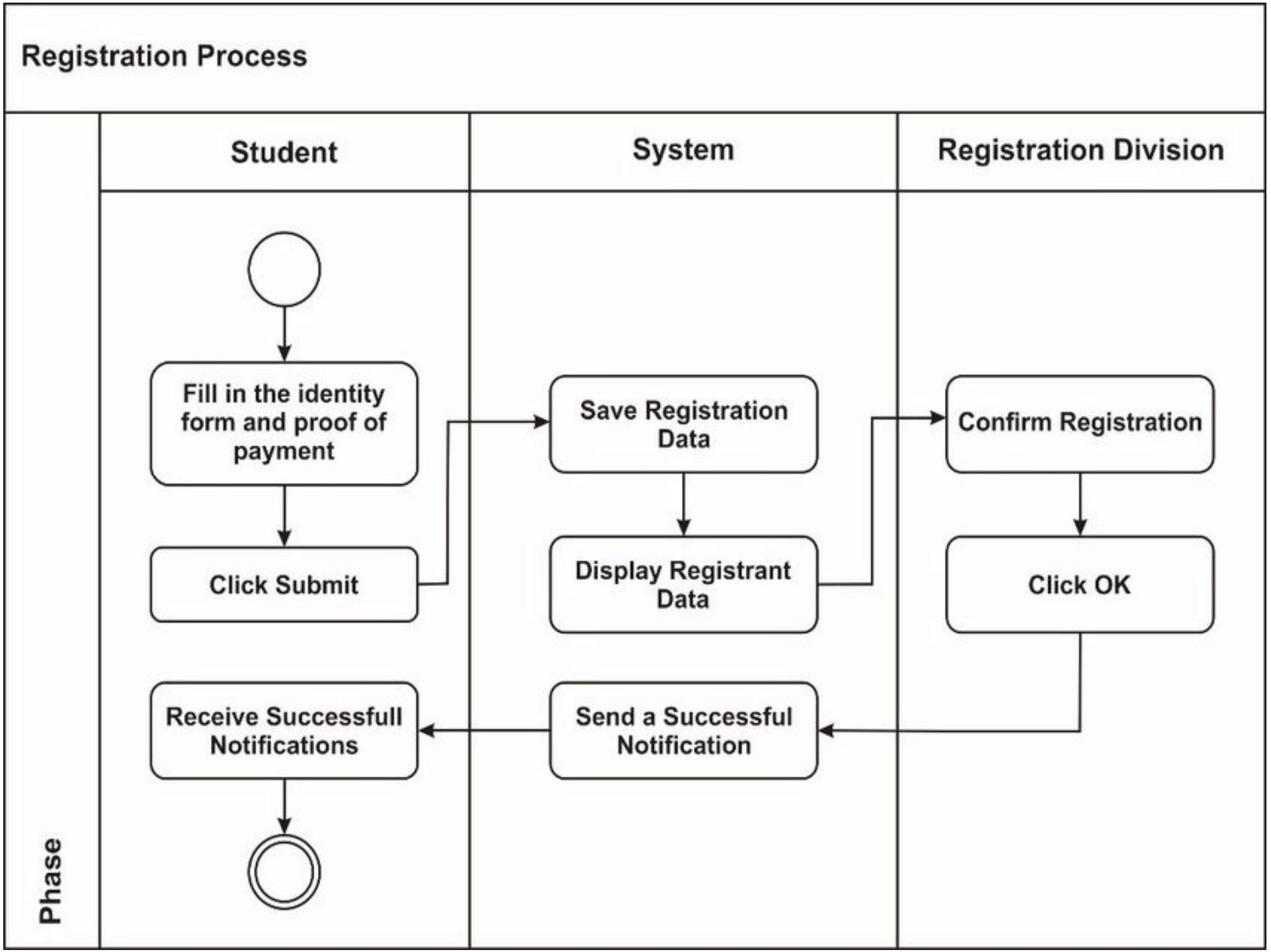
- Activity is started by Commuter **actor** who needs to buy a ticket.
- **Ticket vending machine** will request trip information from Commuter.
- This information will include number and type of tickets, e.g. whether it is a monthly pass, one way or round ticket, route number, destination or zone number, etc.
- Based on the provided trip info ticket vending machine will calculate payment due and request payment options. Those options include payment by **cash, or by credit or debit card**. If payment by card was selected by Commuter, another actor,

- **Bank** will participate in the activity by authorizing the payment.
- After payment is complete, ticket is dispensed to the Commuter.
- Cash payment might result in some change due, so the change is dispensed to the Commuter in this case.
- Ticket vending machine will show some "Thank You" screen at the end of the activity.

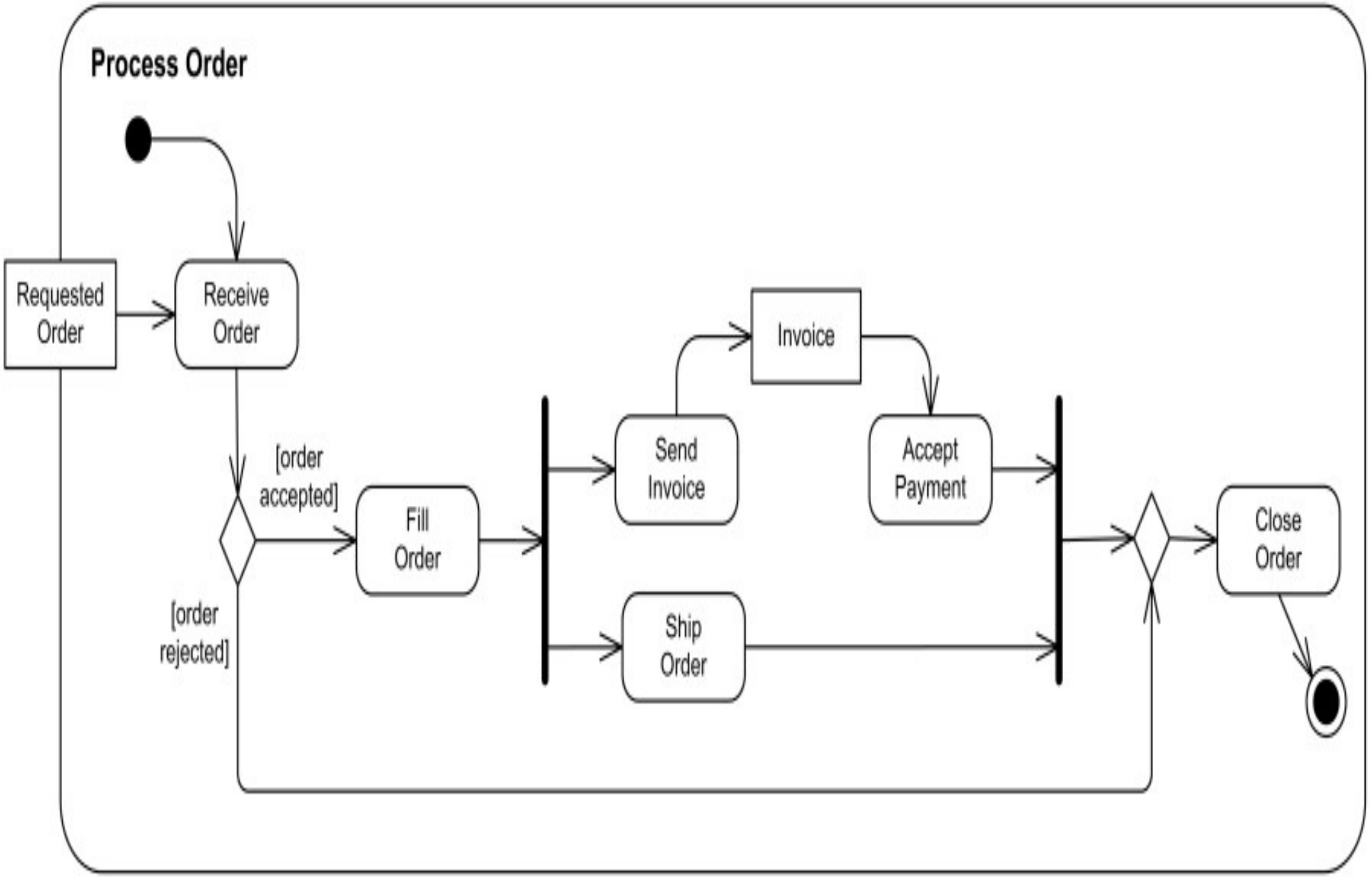

Commuter
Ticket vending machine
Bank
Start Session
Request Trip Info
Provide Trip Info
Process Trip Info
Provide Payment Info
Request Payment
Process Payment
[pay with card]
[pay with cash]
Authorize Card Payment
Dispense Ticket
Ticket
Get Ticket
[paid with cash & with change]
Dispense Change
Change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

Student registration process activity diagram

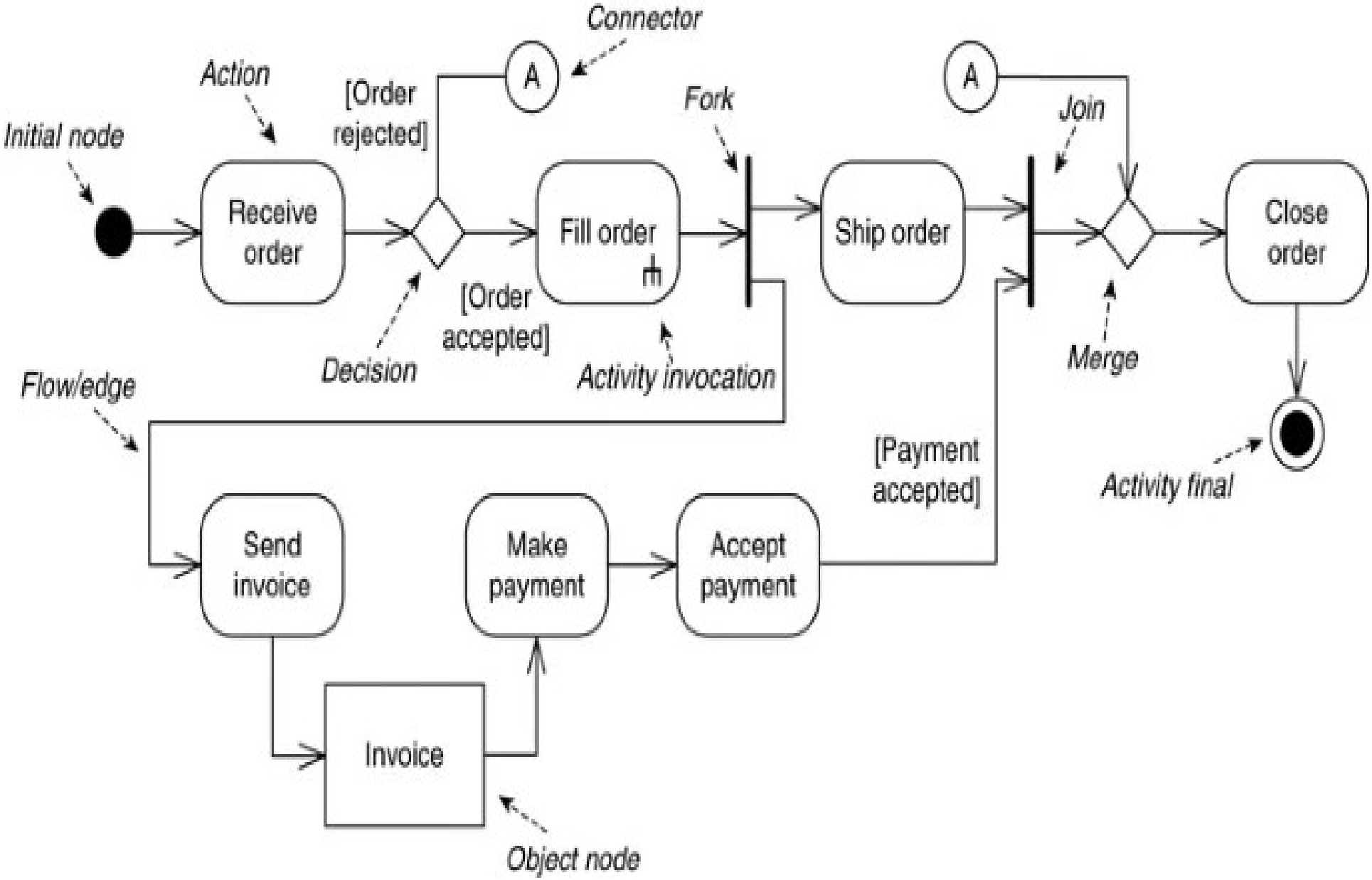
This activity diagram shows the series of actions performed by the **student**, the **student registration system and the registration division** to complete the student registration process.


Registration Process
Student
System
Registration Division
Phase
Fill in the identity form and proof of payment
Click Submit
Receive Successfull Notifications
Save Registration Data
Display Registrant Data
Send a Successful Notification
Confirm Registration
Click OK

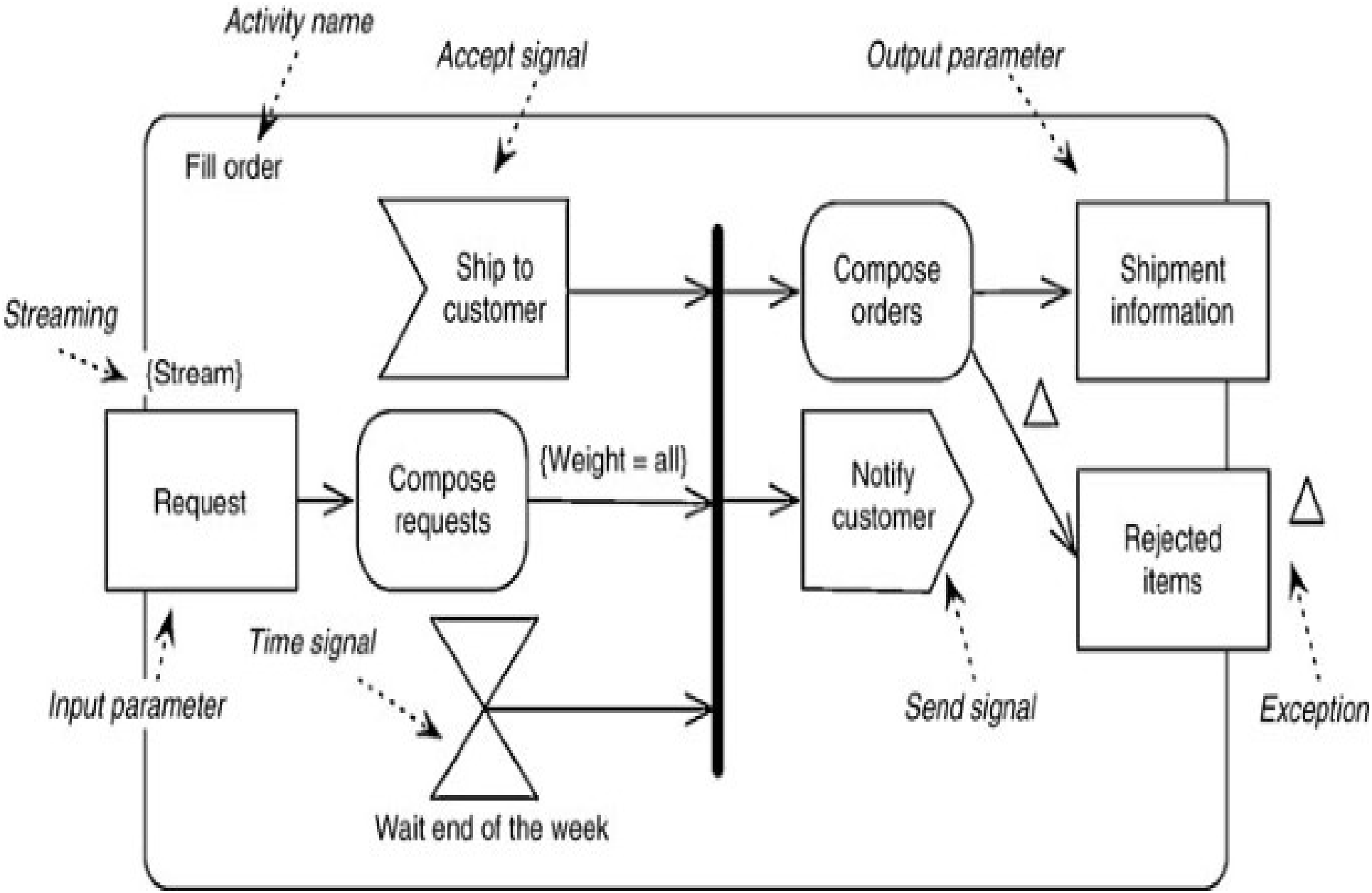
Process - Purchase Order



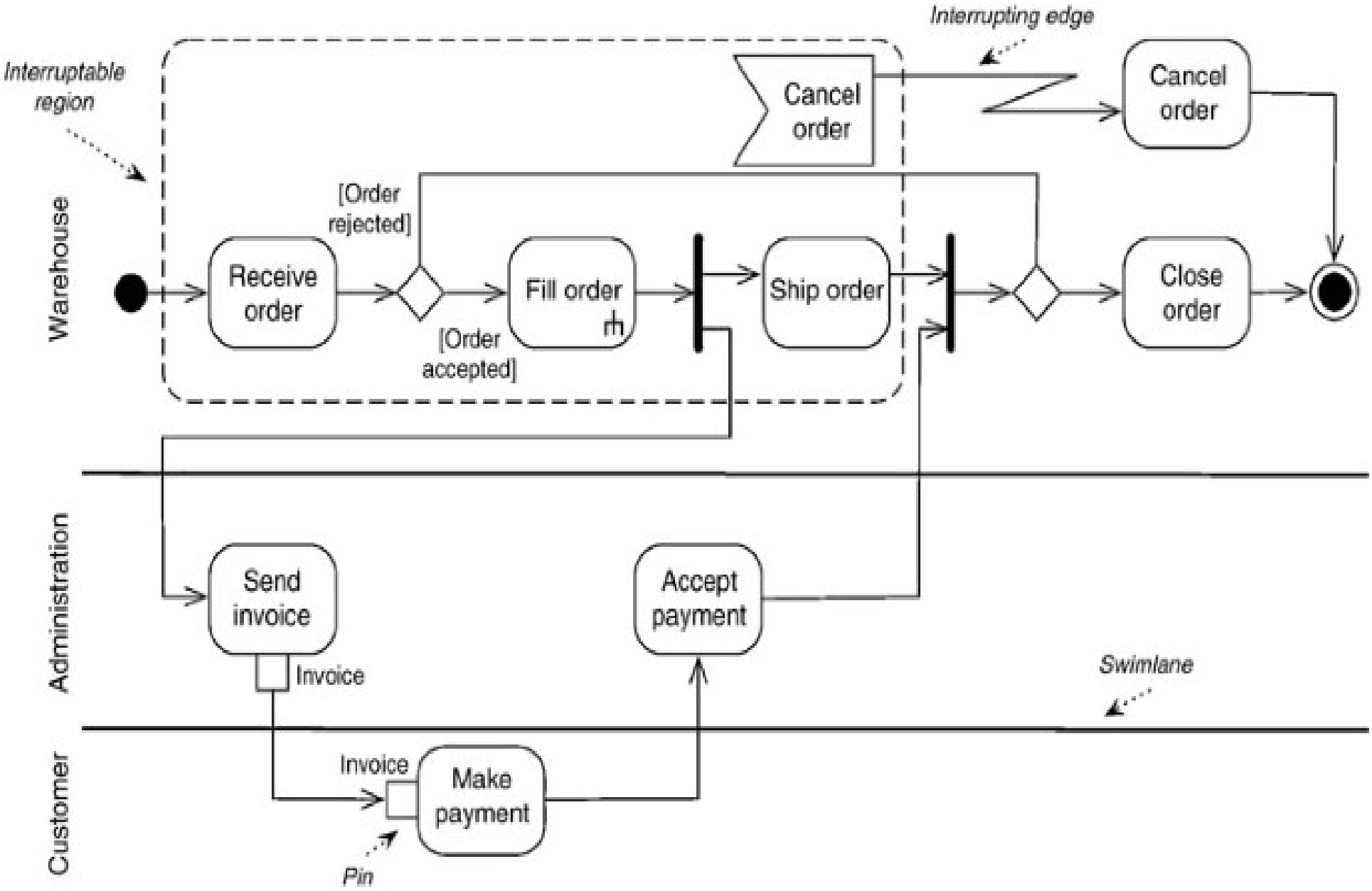
Activity Diagram –order management Level - 1



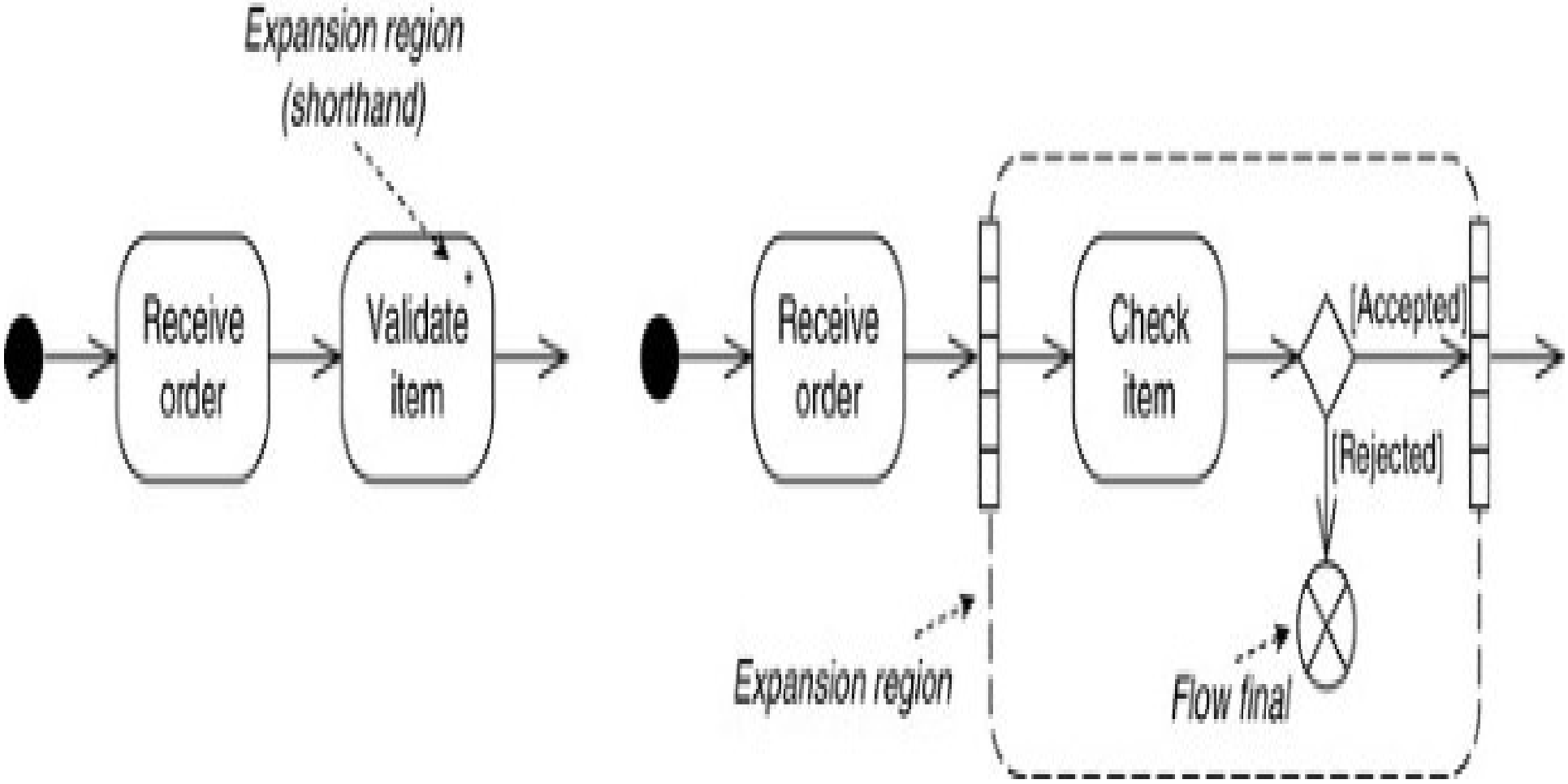
Activity Diagram with Signals



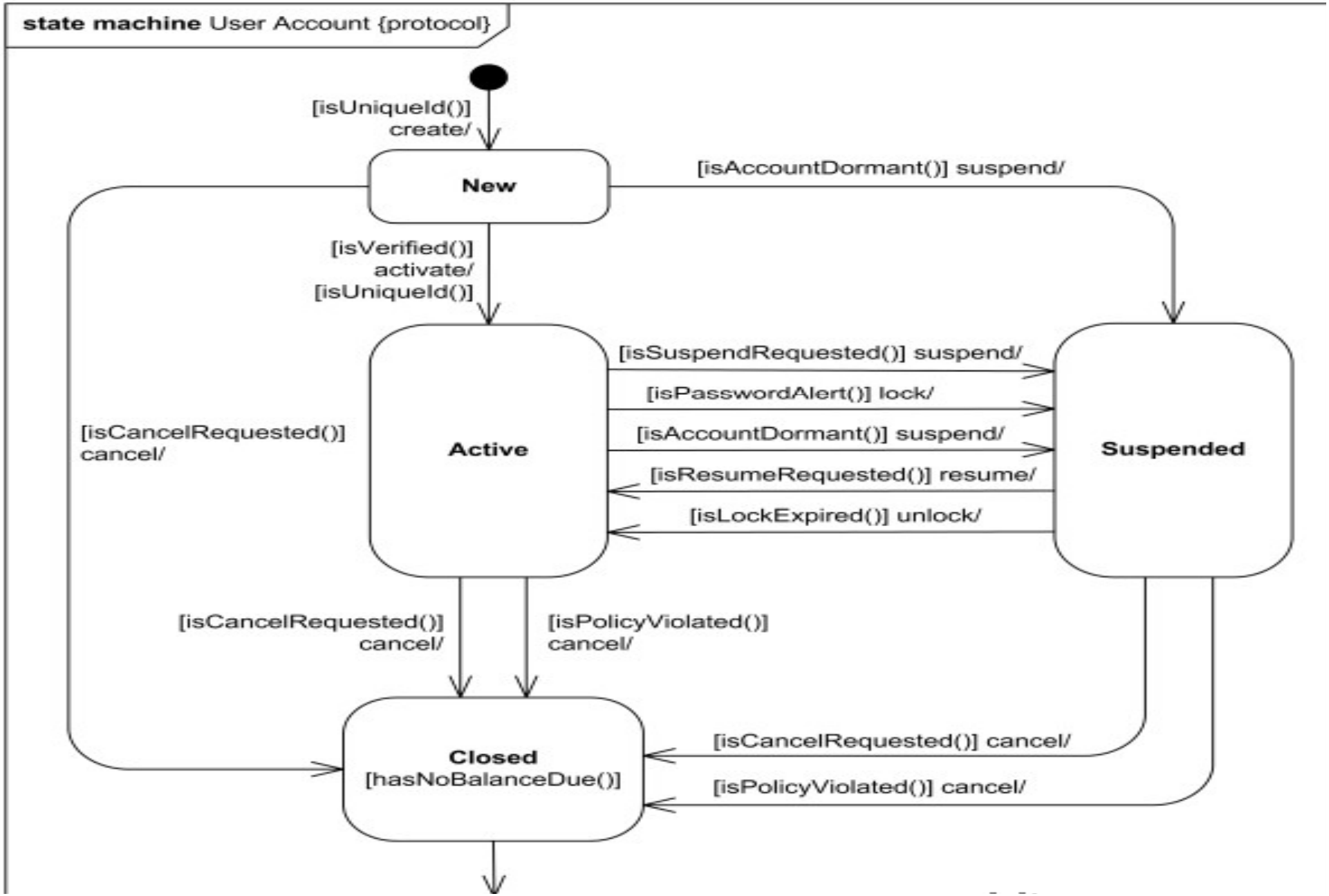
Activity Diagram with Swimlanes



Activity Diagram with Expansion



Online shopping user account protocol state machine diagram.

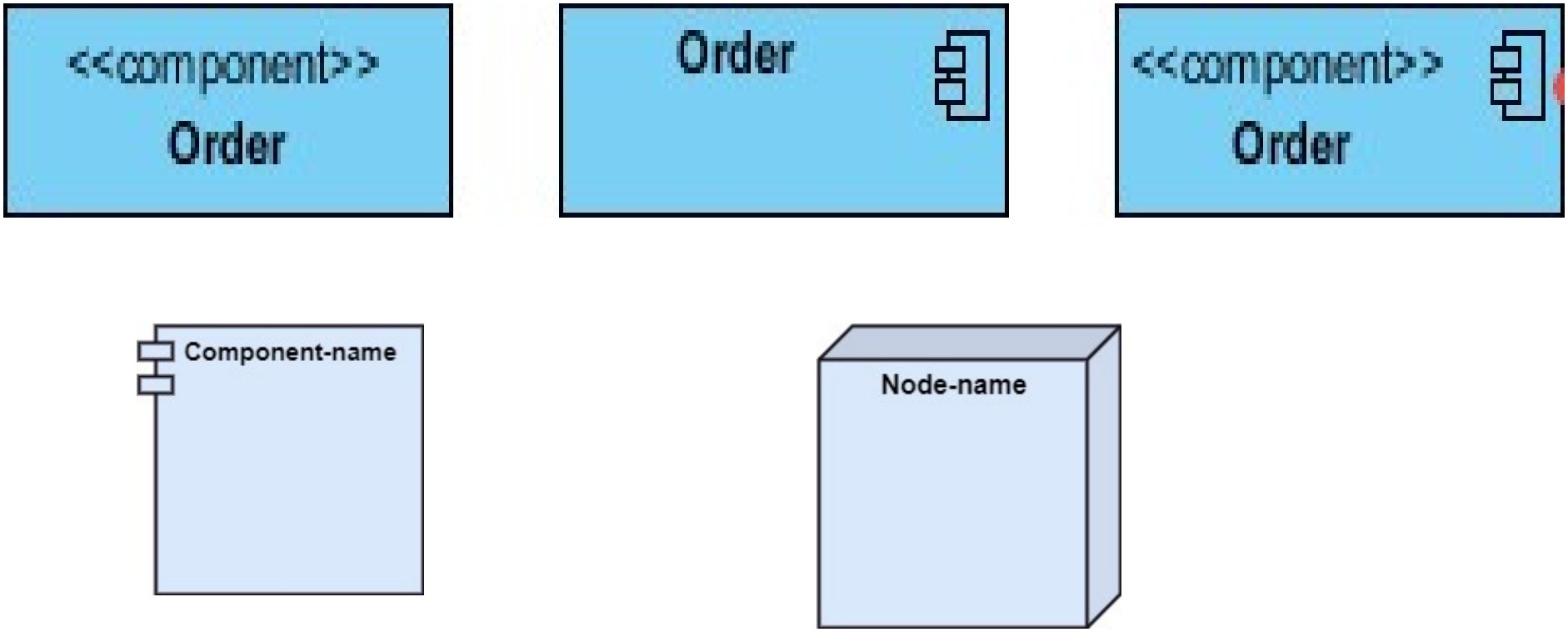


Component Diagram

- A component diagram is used to **break down a large object-oriented system into the smaller components**, so as to make them more manageable.
- It models the **physical view of a system** such as executables, files, libraries, etc.,
- A component is a **single unit** of the system, which is replaceable and executable.
- The **implementation details of a component are hidden**, and it necessitates an interface to execute a function.
- It is like a **black box** whose behavior is explained by the provided and required interfaces.

Basic Concepts of Component Diagram

- A component represents a **modular part of a system that encapsulates its contents** and whose manifestation is replaceable within its environment.
- **Component diagram** shows components, provided and required interfaces, ports, and relationships between them. This type of diagrams is used in **Component-Based Development** (**CBD**) to describe systems with **Service-Oriented Architecture** (**SOA**).
- In UML 2, a component is drawn as a rectangle with **optional compartments stacked vertically**.
- A high-level, abstracted view of a component in UML 2 can be modeled as:
- A rectangle with the component's name
- A rectangle with the component icon
- A rectangle with the stereotype text and/or icon



Example of Components in a Diagram:

A user interface component (UI)
A business logic component (Service Layer)
A database component (Data Access Layer)

Purpose of Component Diagram

- It describes all the individual components that are used to make the functionalities, but not the functionalities of the system.
- It visualizes the physical components inside the system.

- It envisions each component of a system.
- It constructs the executable by incorporating forward and reverse engineering.
- It depicts the relationships and organization of components.

Why Component Diagram

- It is used to depict the functionality and behavior of all the components present in the system, unlike other diagrams that are used to represent the architecture of the system, working of a system, or simply the system itself.
- It portrays the components of a system at the runtime.
- It is helpful in testing a system.
- It envisions the links between several connections.

Key Elements of a Component Diagram:

- **Components**: Represented by a rectangle with a smaller rectangle (representing a package) inside, components denote parts of a system (e.g., modules, subsystems, libraries).

- **Interfaces**: These are shown as circles or lollipop symbols. They define the methods or services that the component provides or requires.

- **Relationships**: These show how components interact. The most common ones are:
  - **Dependency**: A dashed line with an arrow showing that one component depends on another.
  - **Association**: A solid line showing a relationship or connection between components.
  - **Realization**: A dashed line with a triangle, showing that a component implements an interface.

- **Nodes**: Represent the physical infrastructure that hosts the system components (servers, databases, etc.).

Artifacts that are needed to be identified before drawing a component diagram:

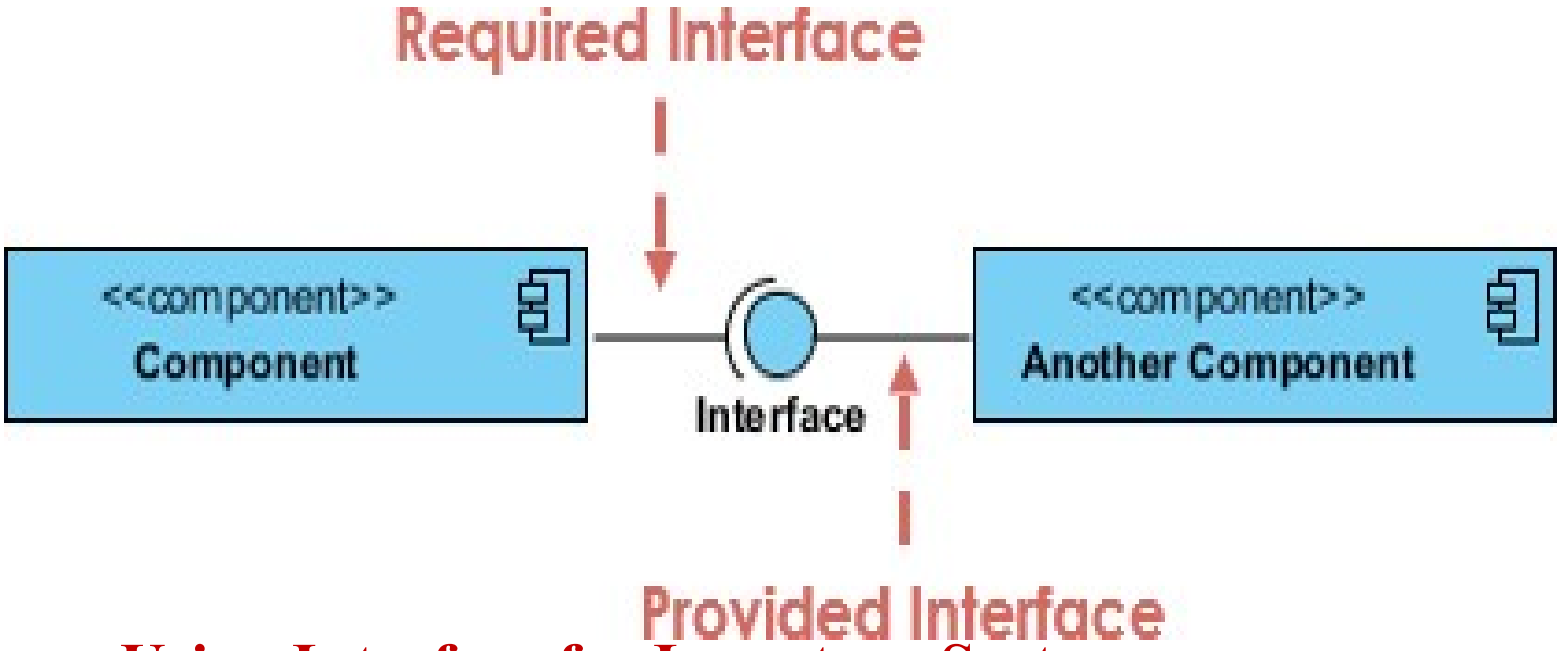
- What files are used inside the system?
- What is the application of relevant libraries and artifacts?
- What is the relationship between the artifacts?
- Following are some points that are needed to be kept in mind after the artifacts are identified:
- Using a meaningful name to ascertain the component for which the diagram is about to be drawn.
- Before producing the required tools, a mental layout is to be made.
- To clarify the important points, notes can be incorporated.
- Example of a Component Diagram

Usage of Component Diagram

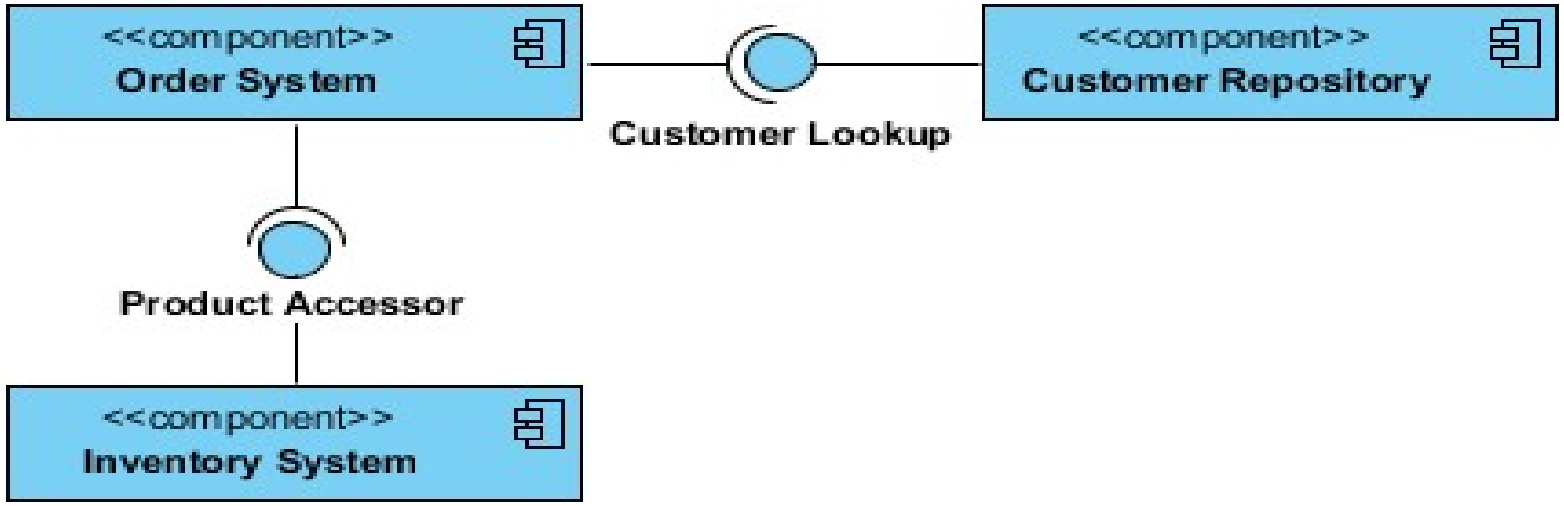
- To model the components of the system.
- To model the schemas of a database.
- To model the applications of an application.
- To model the system's source code.
- Tupes of /component
- **logical components** (e.g., business components, process components), and
- **physical components** (e.g., CORBA components, EJB components, COM+ and .NET components, WSDL components, etc.),

Interface in Component Diagram

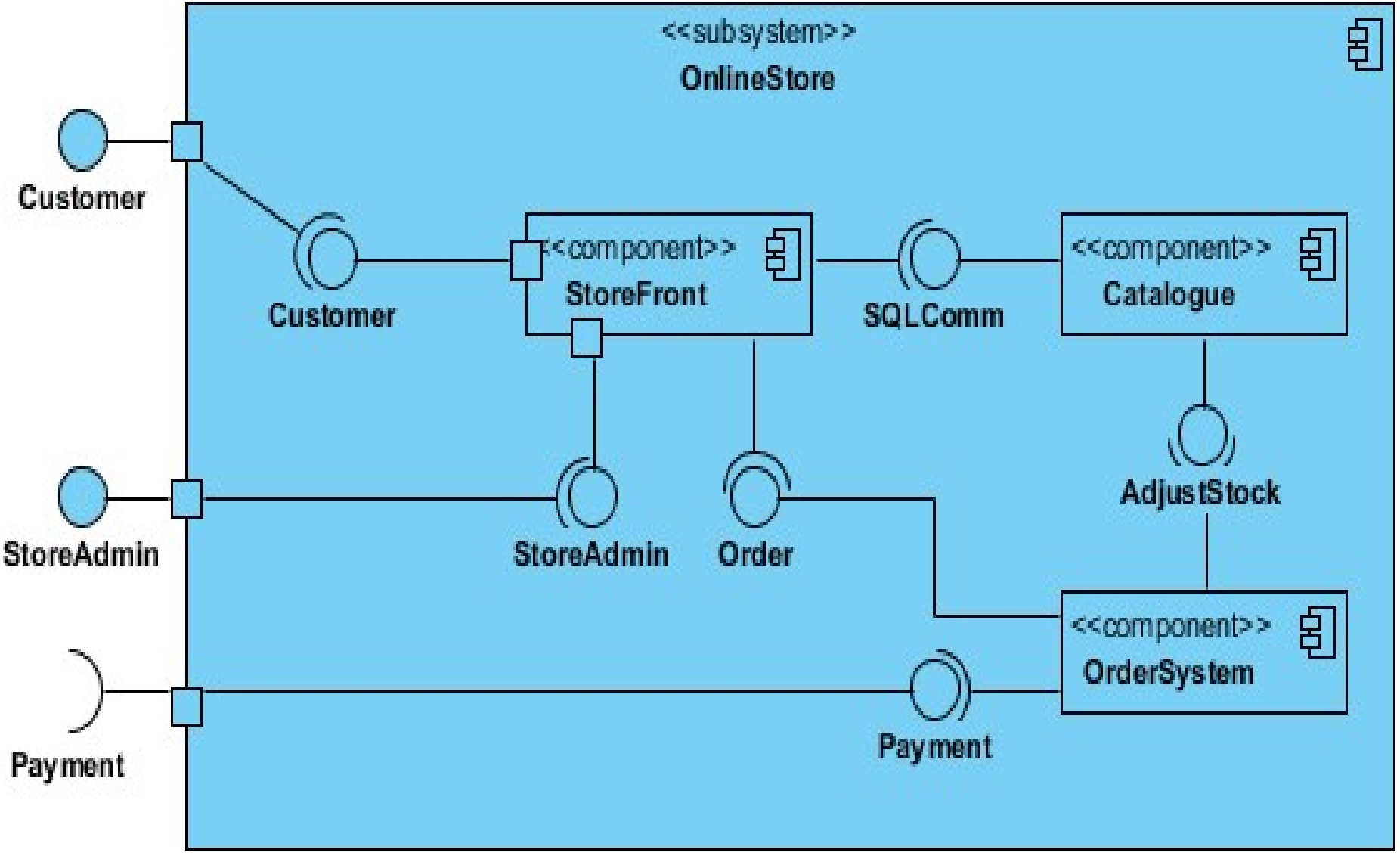
- **Provided interface** symbols with a complete circle at their end represent an interface that the component provides - this **"lollipop"** symbol is shorthand for a realization relationship of an interface classifier.
- **Required Interface** symbols with only a half circle at their end (sockets) represent an interface that the component requires (in both cases, the interface's name is placed near the interface symbol itself).



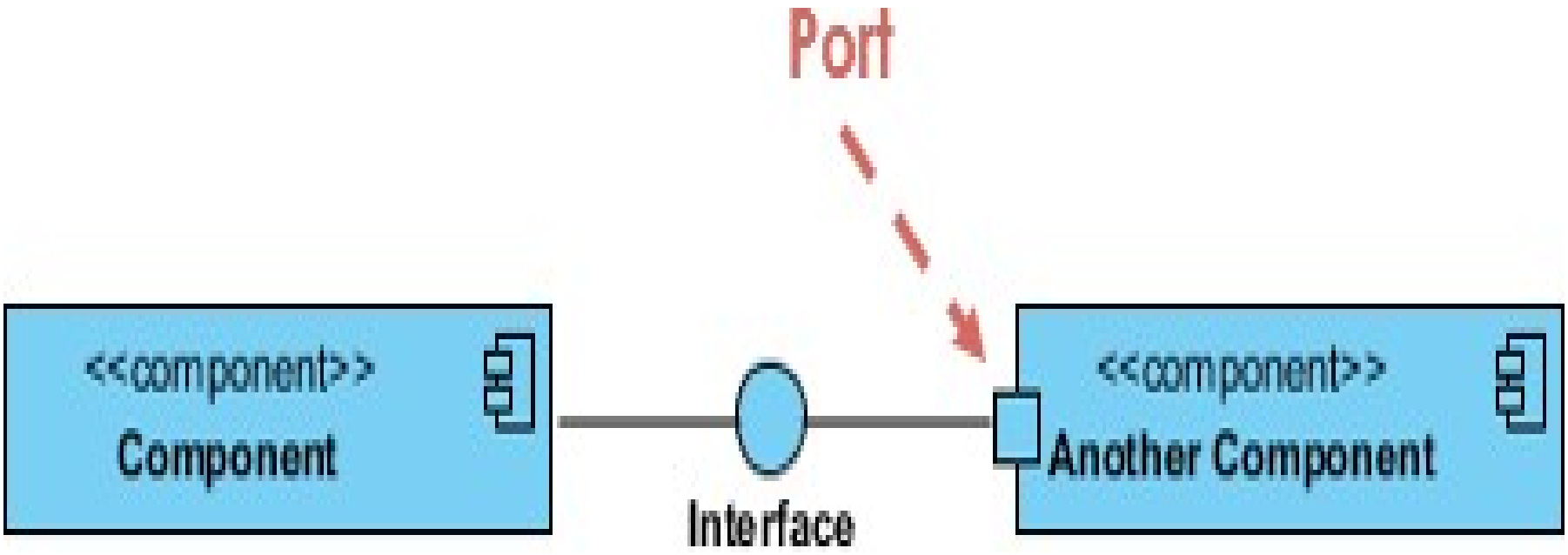
Using Interface for Inventory System



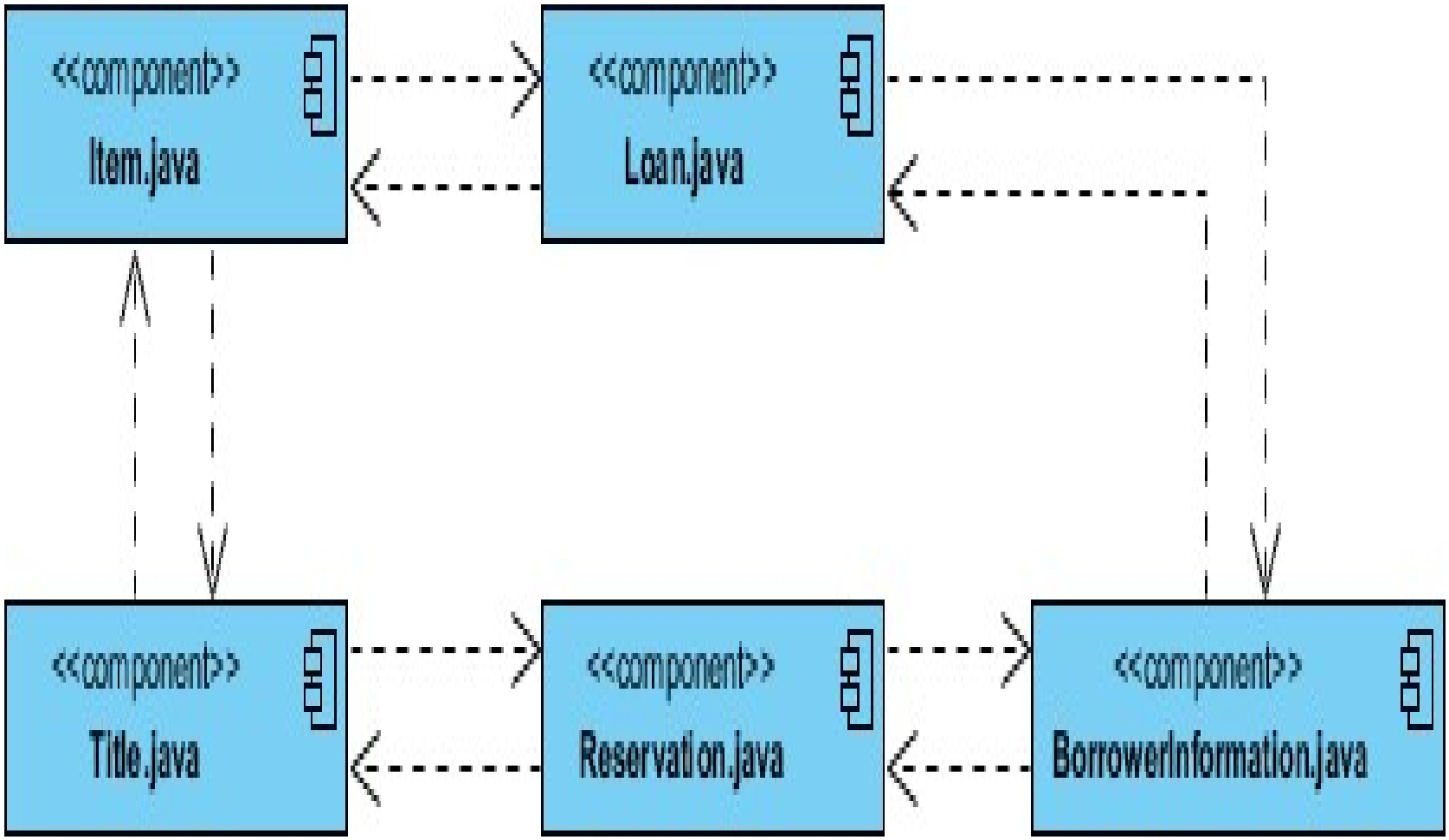
With Subsystems



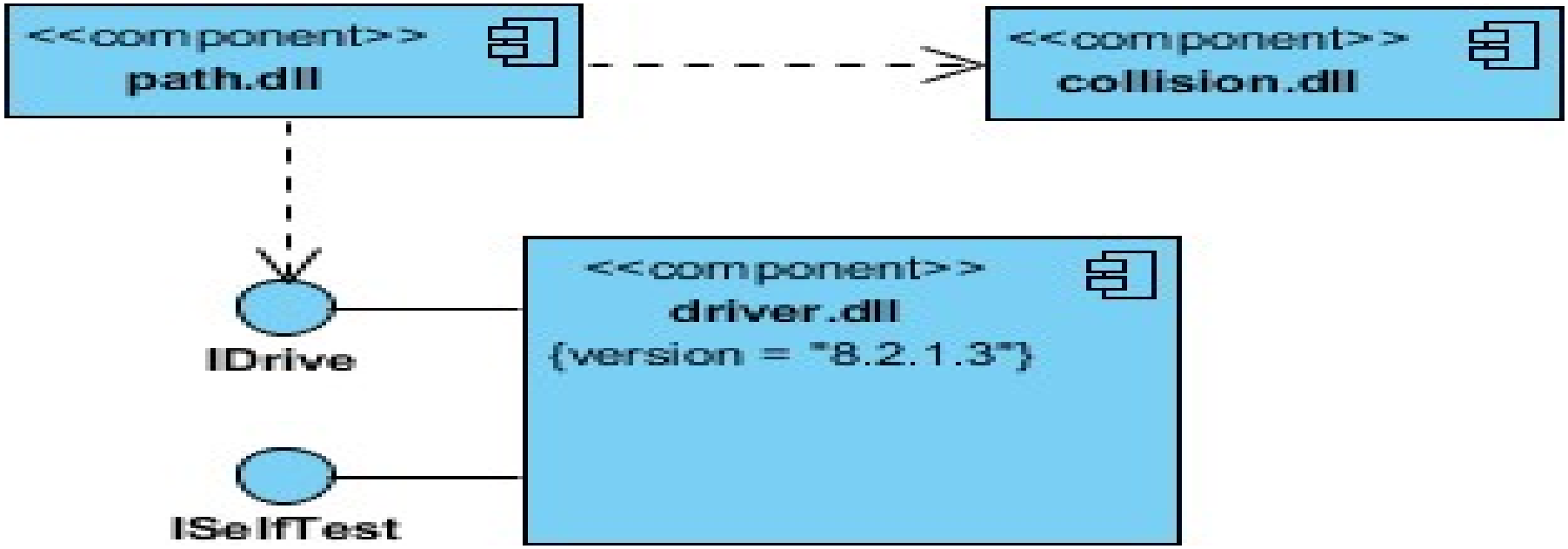
- Ports are represented using a square along the edge of the system or a component.
- A port is often used to help expose required and provided interfaces of a component.



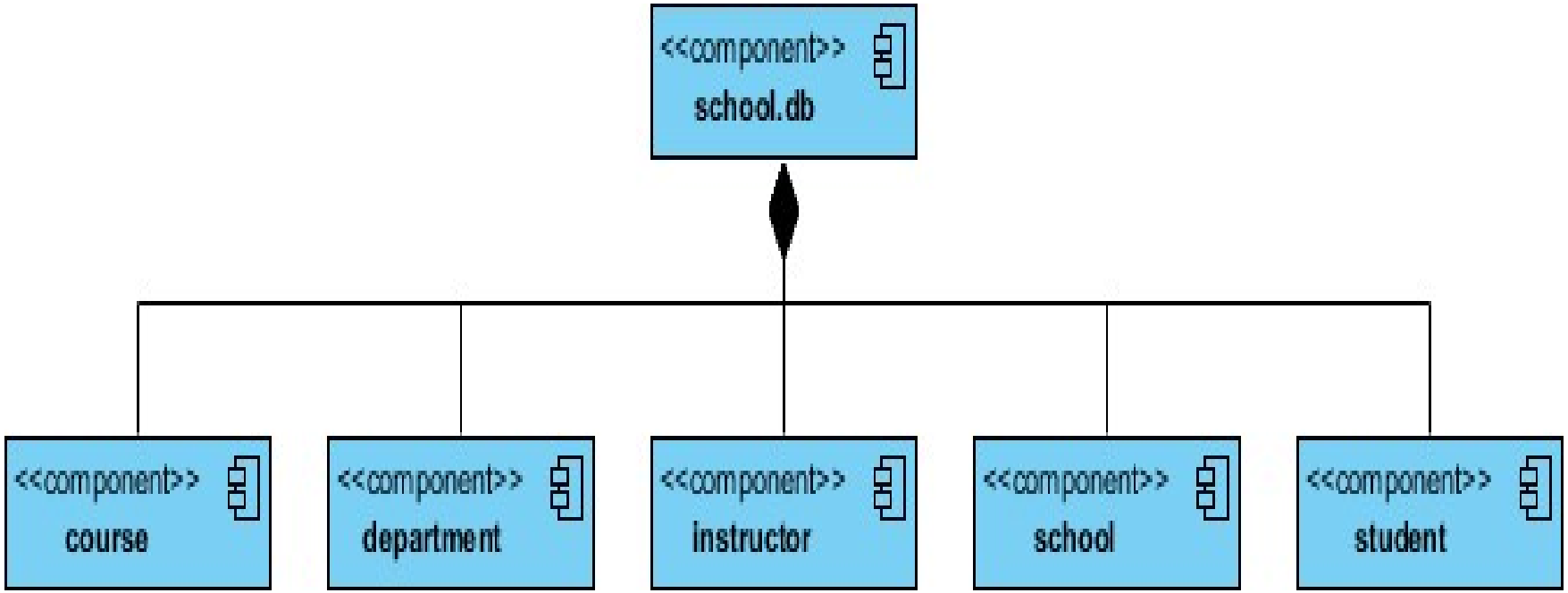
Component with Java Code



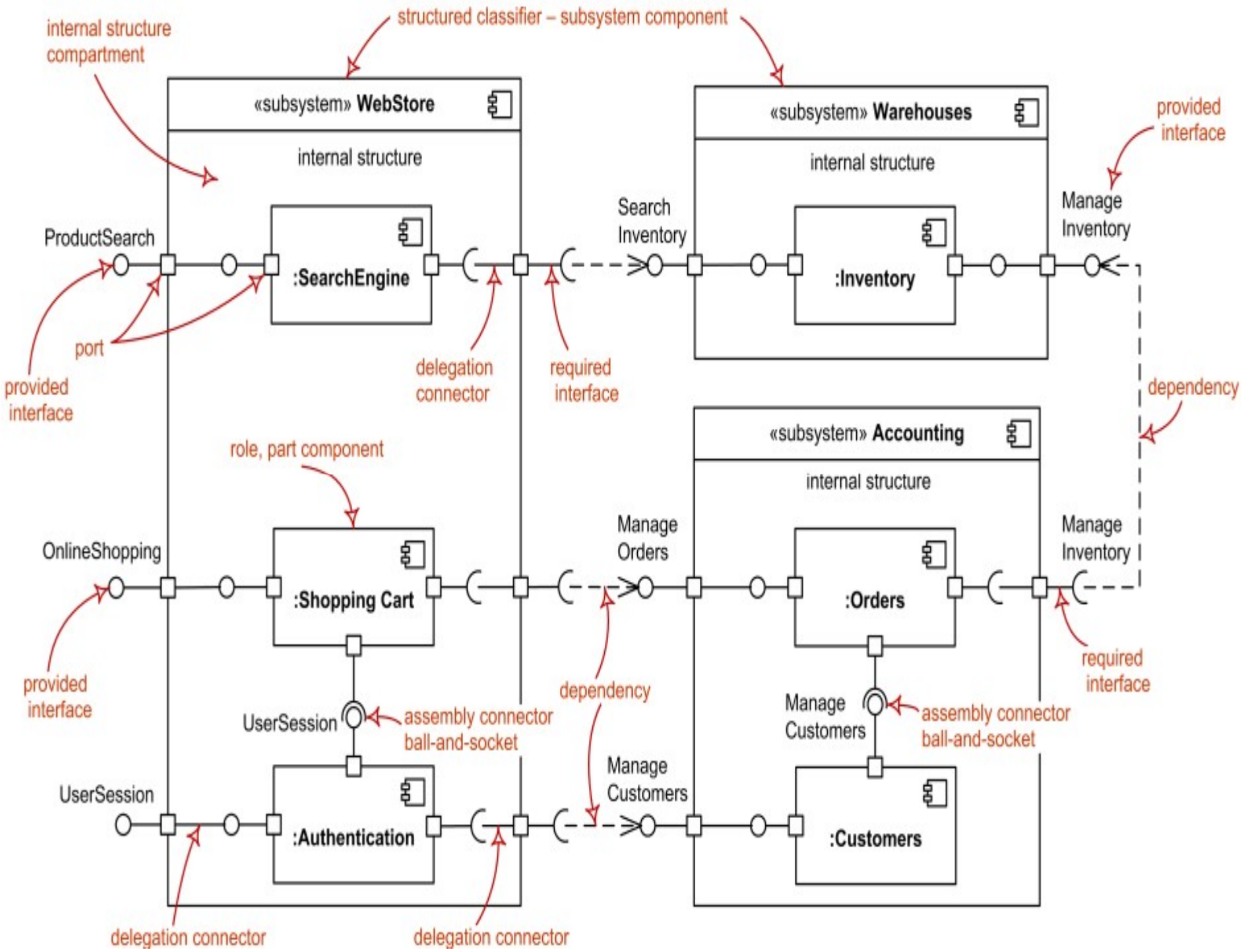
Modeling an Executable



Modeling a Physical Database



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


internal structure compartment
structured classifier – subsystem component
«subsystem» WebStore
internal structure
«subsystem» Warehouses
internal structure
provided interface
ProductSearch
:SearchEngine
Search Inventory
:Inventory
Manage Inventory
port
provided interface
delegation connector
required interface
dependency
«subsystem» Accounting
internal structure
role, part component
OnlineShopping
:Shopping Cart
Manage Orders
:Orders
Manage Inventory
provided interface
UserSession
assembly connector ball-and-socket
dependency
Manage Customers
assembly connector ball-and-socket
required interface
UserSession
:Authentication
Manage Customers
:Customers
delegation connector
delegation connector

Components include

- **Frontend**: The user interface component.
- **Backend**: The server-side logic component.
- **Database**: Where data is stored.
- **API Gateway**: The component that manages communication between the frontend and backend.
- **External Services**: Like a third-party authentication service.
- **Diagram Overview:**
- **Frontend ←→ API Gateway ←→ Backend ←→ Database**
- The **Backend** also communicates with **External Services** for user authentication.

Key Elements of a Database System:

- **Database**: A structured collection of data, typically organized in tables, schemas, and relations.
- **DBMS (Database Management System)**: Software that handles the storage, retrieval, and updating of data in the database.
- **Tables**: Represent entities in a system (e.g., Customers, Orders, Products).
- **Relationships**: The associations between tables, such as one-to-many or many-to-many.
- **Indexes**: Used for efficient retrieval of data.
- **Queries**: SQL (Structured Query Language) commands to interact with the database.
- **Example of a Database System:**

DEPLOYMENT DIAGRAMS

- **Deployment diagram** is a **structure diagram** which shows architecture of the system as deployment (distribution) of software artifacts to deployment targets. There is a strong link between components diagrams and deployment diagrams

- Deployment diagrams
  - Show the physical relationship between hardware and software in a system
  - Hardware elements:
    - Computers (clients, servers)
    - Embedded processors
    - Devices (sensors, peripherals)
  - Are used to show the nodes where software components reside in the run-time system

- **Artifacts** represent concrete elements in the physical world that are the result of a development process.
  Examples of artifacts are executable files, libraries, archives, database schemas, configuration files, etc.
- **Deployment target** is usually represented by a **node** which is either hardware device or some software execution environment.
- Nodes could be connected through **communication paths** to create networked systems of arbitrary complexity.
- Note, that **components** were directly deployed to nodes in UML 1.x deployment diagrams.
- In UML 2.x **artifacts** are deployed to nodes, and artifacts could **manifest** (implement) components.
- Components are deployed to nodes indirectly through artifacts.

- Deployment diagrams could describe architecture at **specification level** (also called type level) or at **instance level** (similar to class diagrams and object diagrams).

- **Specification level** deployment diagram shows some overview of **deployment** of **artifacts** to **deployment targets**, without referencing specific instances of artifacts or nodes.
- **Instance level** deployment diagram shows **deployment** of instances of **artifacts** to specific instances of **deployment targets**.

- It could be used for example to show differences in deployments to development, staging or production environments with the names/ids of specific build or deployment servers or devices.

Some common types of deployment diagrams are

- **Implementation (manifestation) of components by artifacts**,
- **Specification level deployment diagram**,
- **Instance level deployment diagram**,
- **Network architecture of the system**.

Deployment diagram

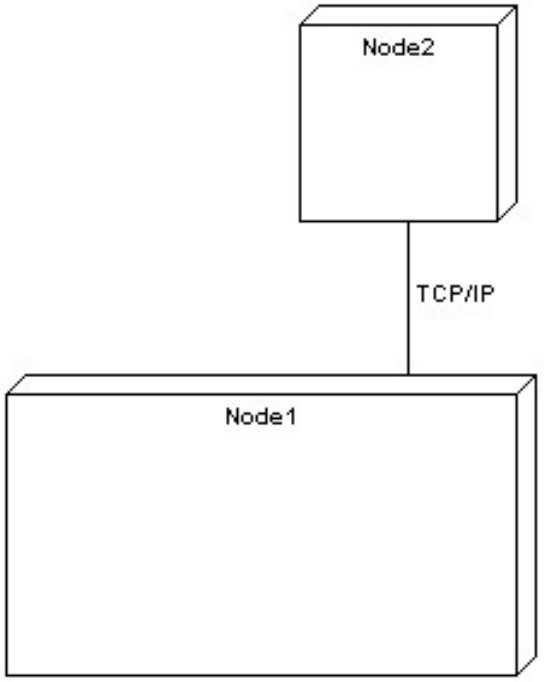
- Depicts a static view of the runtime configuration of hardware nodes and the software components that run on those nodes.
- UML deployment diagrams show the hardware for your system, the software that is installed on that hardware, and the middleware used to connect the disparate machines to one another.

Creating an UML deployment diagram to:

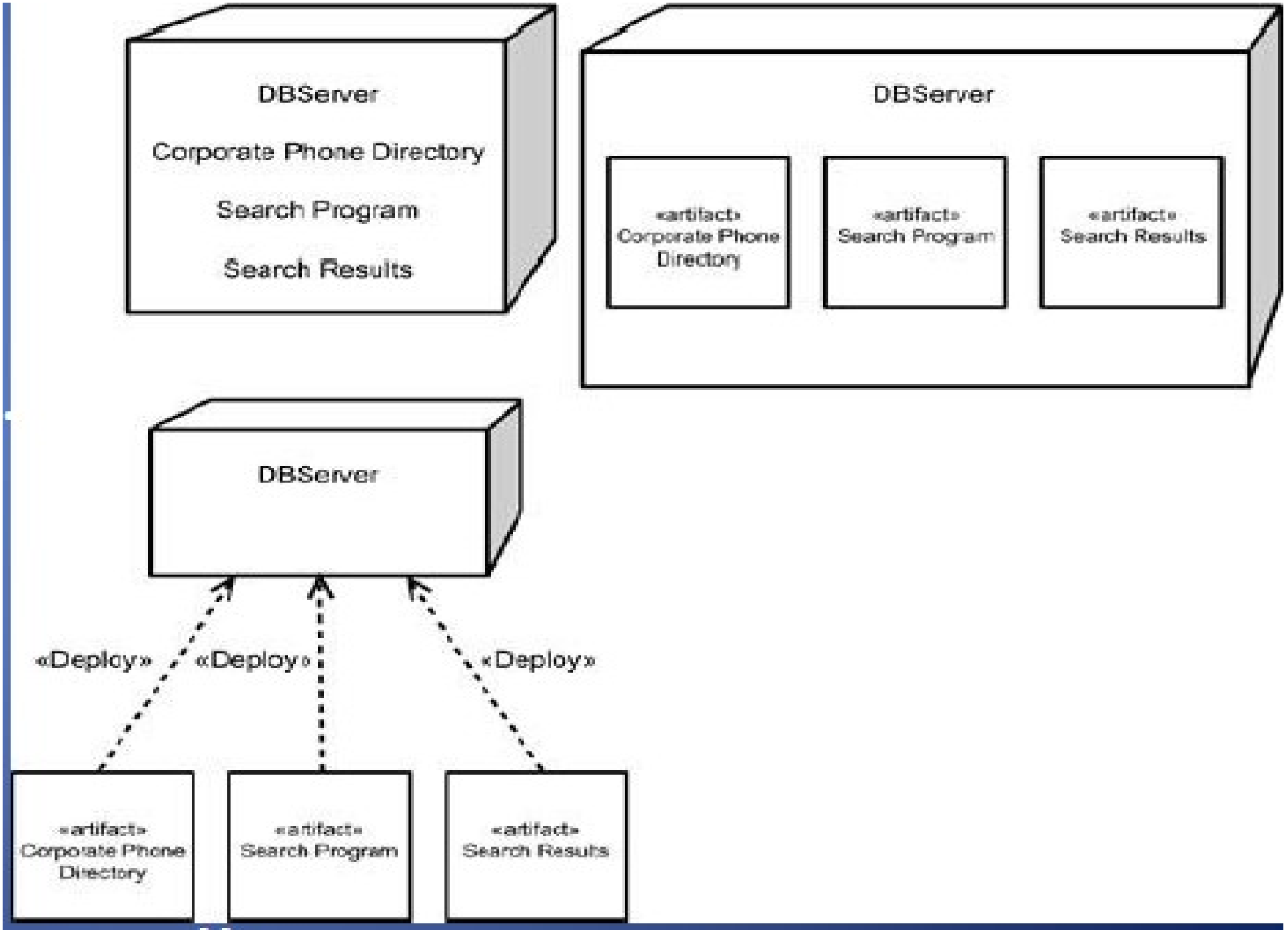
- explore the issues involved in installing your system into production,
- explore the dependencies that your system has with other systems that are currently in, or planned for,your production environment,
- depict a major deployment configuration of a business application,
- design the hardware and software configuration of an embedded system, or
- depict the hardware/network infrastructure of an organization.

DEPLOYMENT DIAGRAMS

- Deployment diagram
  - Contains nodes and connections
  - A node usually represent a piece of hardware in the system
  - A connection depicts the communication path used by the hardware to communicate
  - Usually indicates the method such as TCP/IP

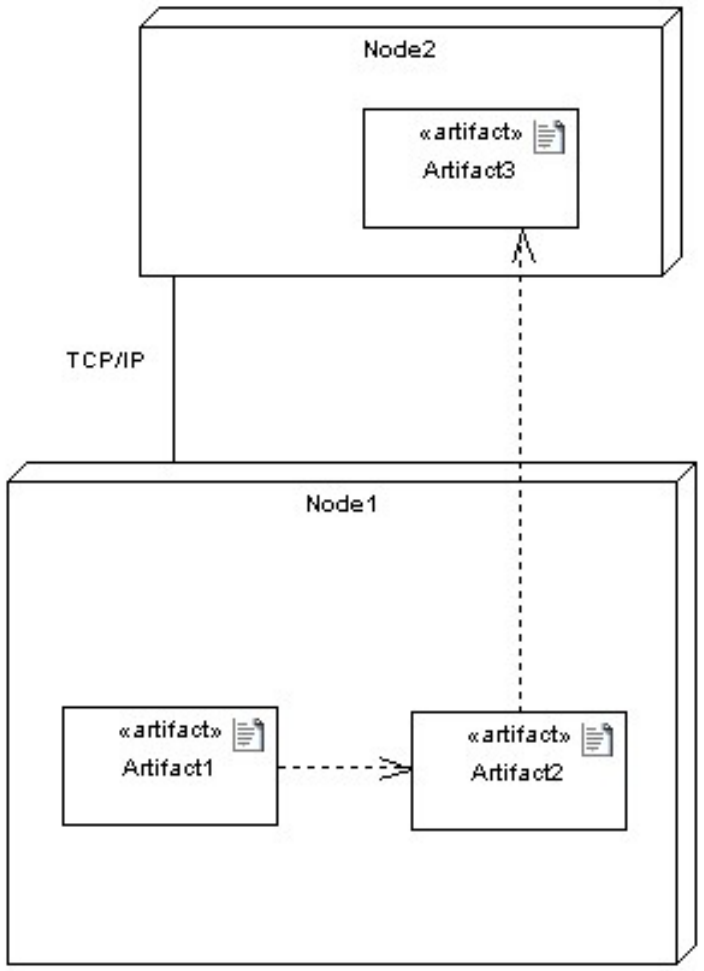


- UML 2.0 now formally defines a device as a node that executes artifacts.
- (Remember an Executable is now classified as an artifact).
- Representing Node:
- You supply a name for the node, and you can add the keyword ≪Device≫, although it's not necessary.


DBServer
Corporate Phone Directory
Search Program
Search Results
DBServer
«artifact»
Corporate Phone Directory
«artifact»
Search Program
«artifact»
Search Results
DBServer
«Deploy»
«Deploy»
«Deploy»
«artifact»
Corporate Phone Directory
«artifact»
Search Program
«artifact»
Search Results

DEPLOYMENT DIAGRAMS

- Deployment diagrams contain artifact
- An artifact
  - Is the specification of a phisycal piece of information
  - Ex: source files, binary executable files, table in a database system,….
  - An artifact defined by the user represents a concrete element in the physical world



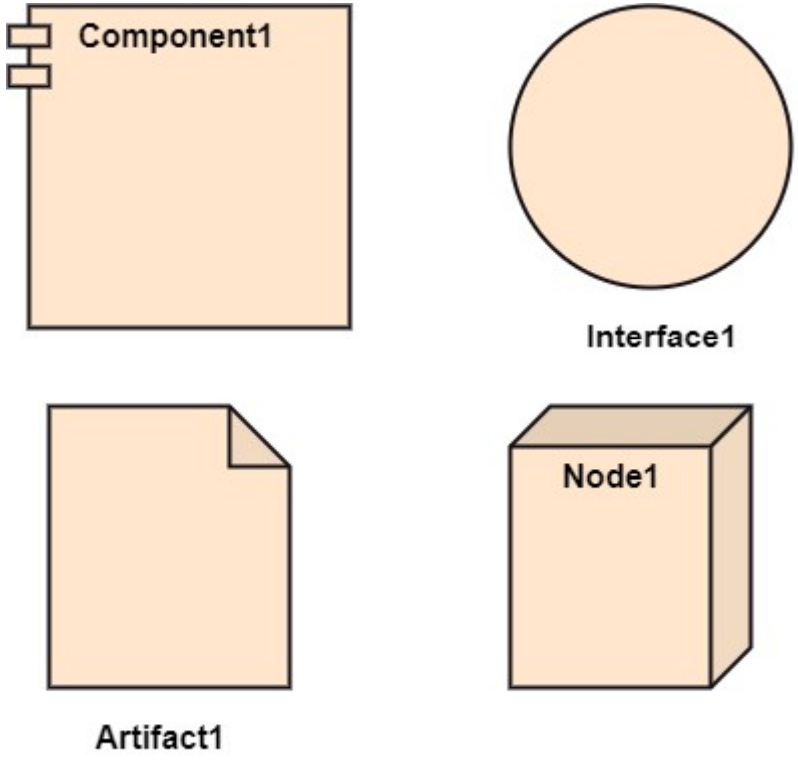
Purpose of Deployment Diagram

- To represent how software is installed on the hardware component. It depicts in what manner a software interacts with hardware to perform its execution.
- To envision the hardware topology of the system.
- To represent the hardware components on which the software components are installed.
- To describe the processing of nodes at the runtime.

Notation of Deployment diagram

The deployment diagram consist of the following notations:

- A component
- An artifact
- An interface
- A node



- Artifacts are the tangible elements, such as software programs or files, that live on the nodes.

- The deployment diagram portrays the deployment view of the system.
- It helps in visualizing the topological view of a system. It incorporates nodes, which are physical hardware.
- The nodes are used to execute the artifacts.
- The instances of artifacts can be deployed on the instances of nodes
- Since it plays a critical role during the administrative process, it involves the following parameters:
- High performance
- Scalability
- Maintainability
- Portability
- Easily understandable

When to use a Deployment Diagram

- The deployment diagram is mostly employed by network engineers, system administrators, etc. with the purpose of representing the deployment of software on the hardware system.
- It envisions the interaction of the software with the hardware to accomplish the execution.
- The selected hardware must be of good quality so that the software can work more efficiently at a faster rate by producing accurate results in no time.

Deployment diagrams can be used for the followings:

- To model the network and hardware topology of a system.
- To model the distributed networks and systems.
- Implement forwarding and reverse engineering processes.
- To model the hardware details for a client/server system.
- For modeling the embedded system.

Deployment Diagram – Simple Steps

- **Identify the nodes** - physical location, network structure, and processing environment of each node.
- **Outline the software artifacts** - includes compiled code, databases, libraries, or any other executables.
- **Establish relationships** - setting up association, dependency, or deployment relationships, which are depicted through specific connectors and directional arrows
- **Configure artifacts on nodes** - includes web servers hosting your applications, database servers managing your data, or any middleware necessary for the communication between software components.

- **Detail the environment -** includes the execution environment which could be a specific operating system, a cloud-based platform, or a containerized solution like Docker.
- **Annotate the diagram - i**ncludes specific network configurations, firewall rules, or any other pertinent information that affects deployment.

Benefits of Deployment Diagram

- Simplification of complex systems
- Enhanced communication
- Documentation and standardization

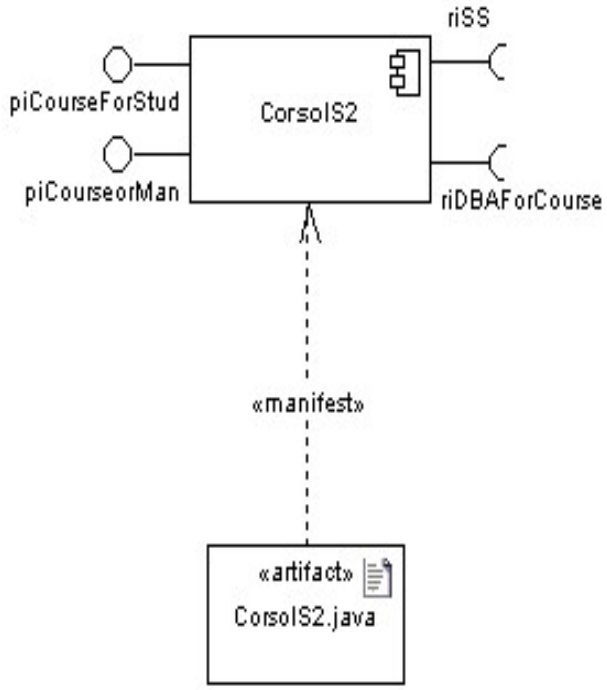
Manifestation diagram

- While **component diagrams** show components and relationships between components and classifiers, and **deployment diagrams - deployments** of artifacts to deployment targets,
- **some missing intermediate diagram** is **manifestation diagram** to be used to show **manifestation** (implementation) of **components** by **artifacts** and internal structure of artifacts.

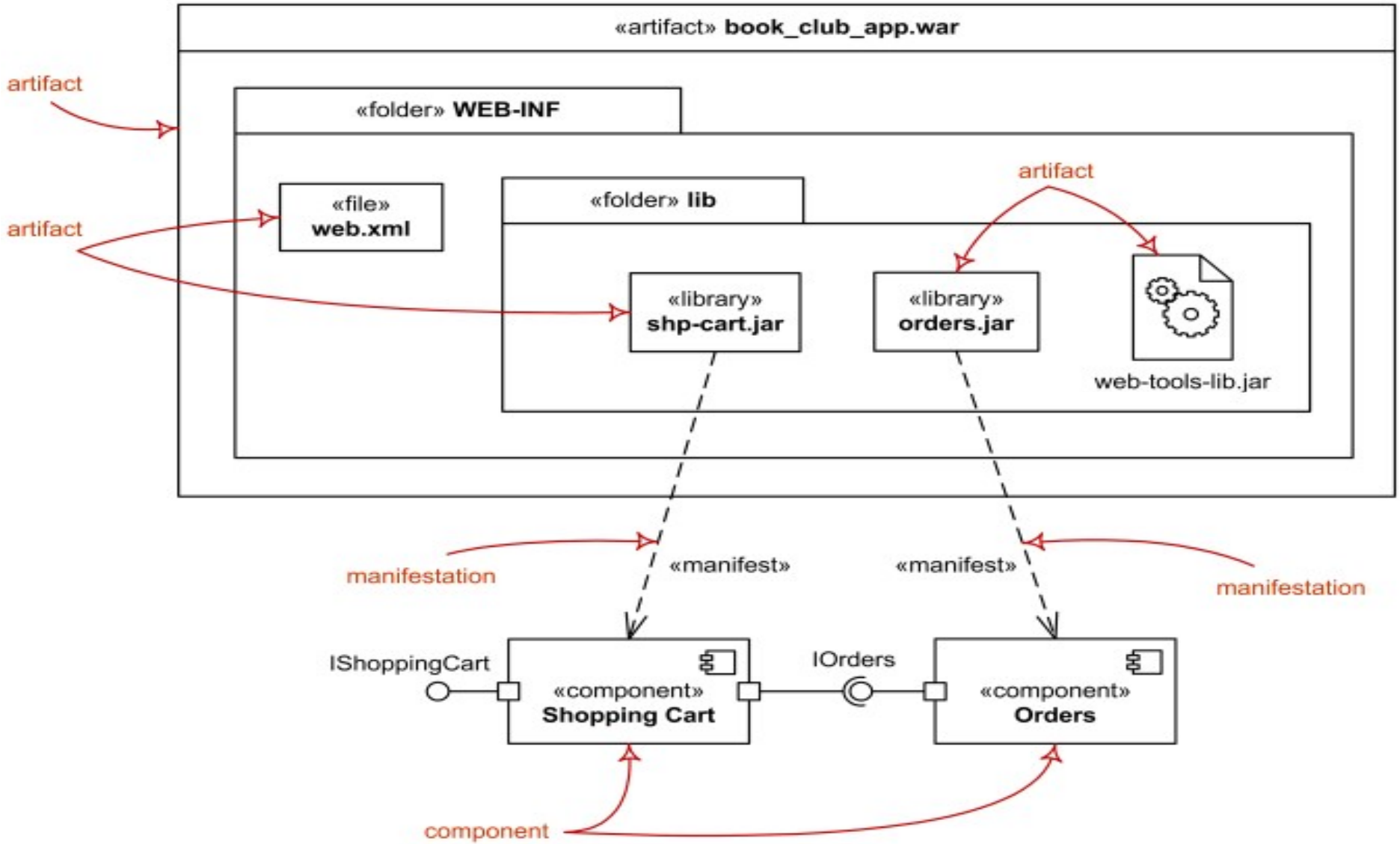
Manifestation diagram

- An artifact manifest one or more model elements
- A <<manifestation>> is the concrete physical of one or more model elements by an artifact
- This model element often is a component

- A manifestation is notated as a dashed line with an open arrow-head labeled with the keyword <<manifest>>



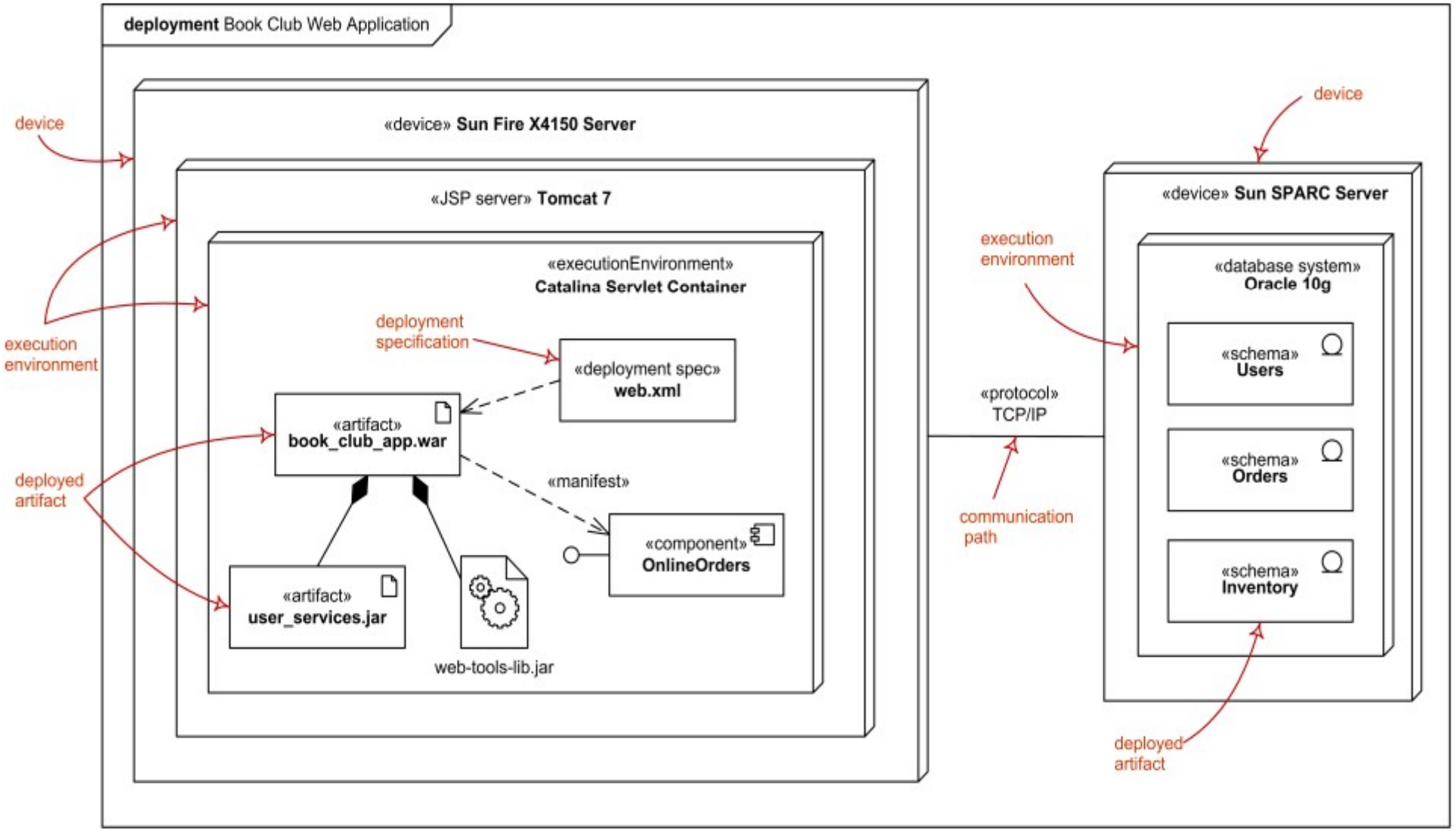
Deployment Diagram - Manifestation diagram



Manifestation of components by artifacts

Specification Level Deployment Diagram

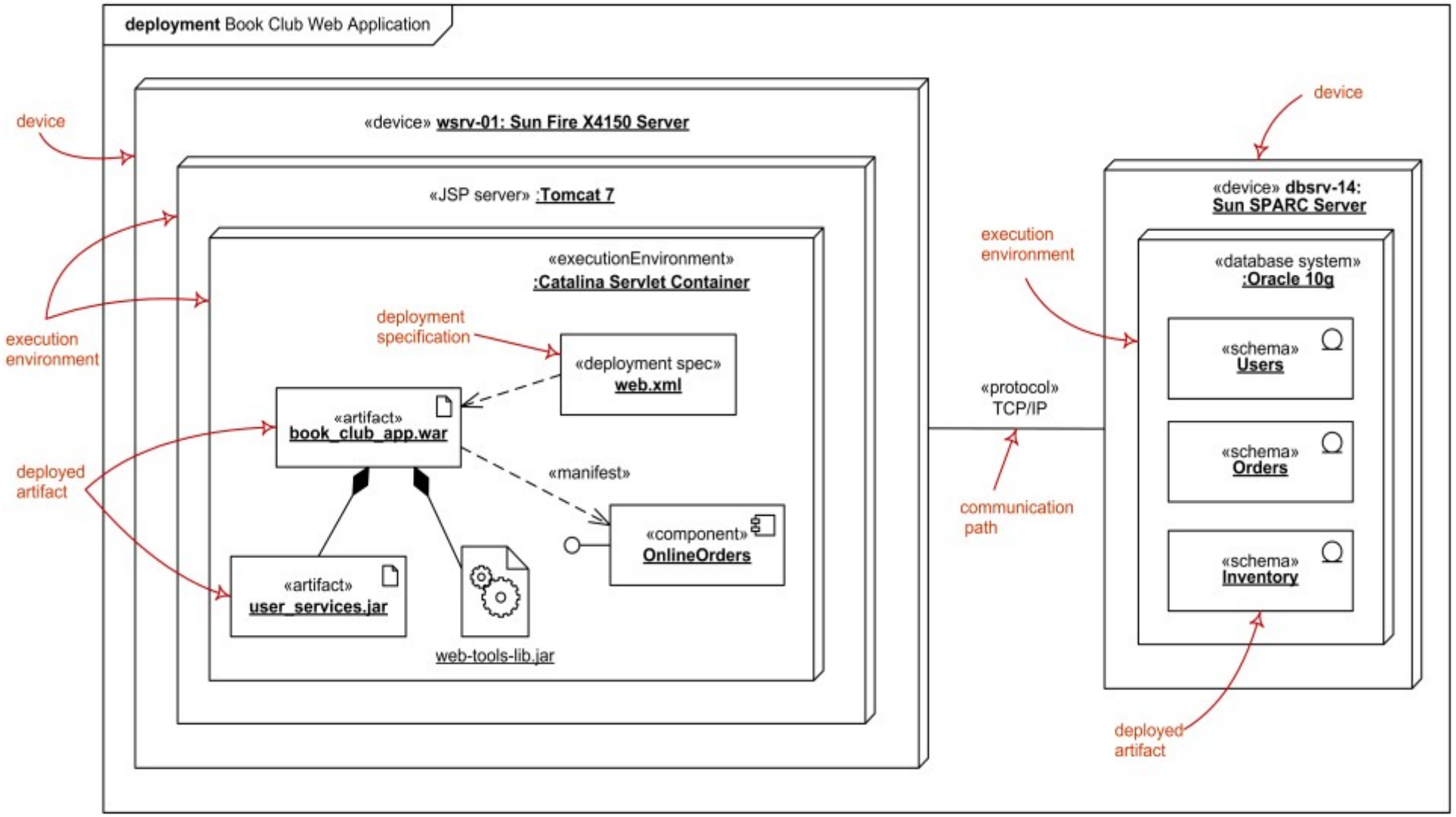
Specification level deployment diagram - web application **deployed to Tomcat JSP server and database schemas - to database system**.



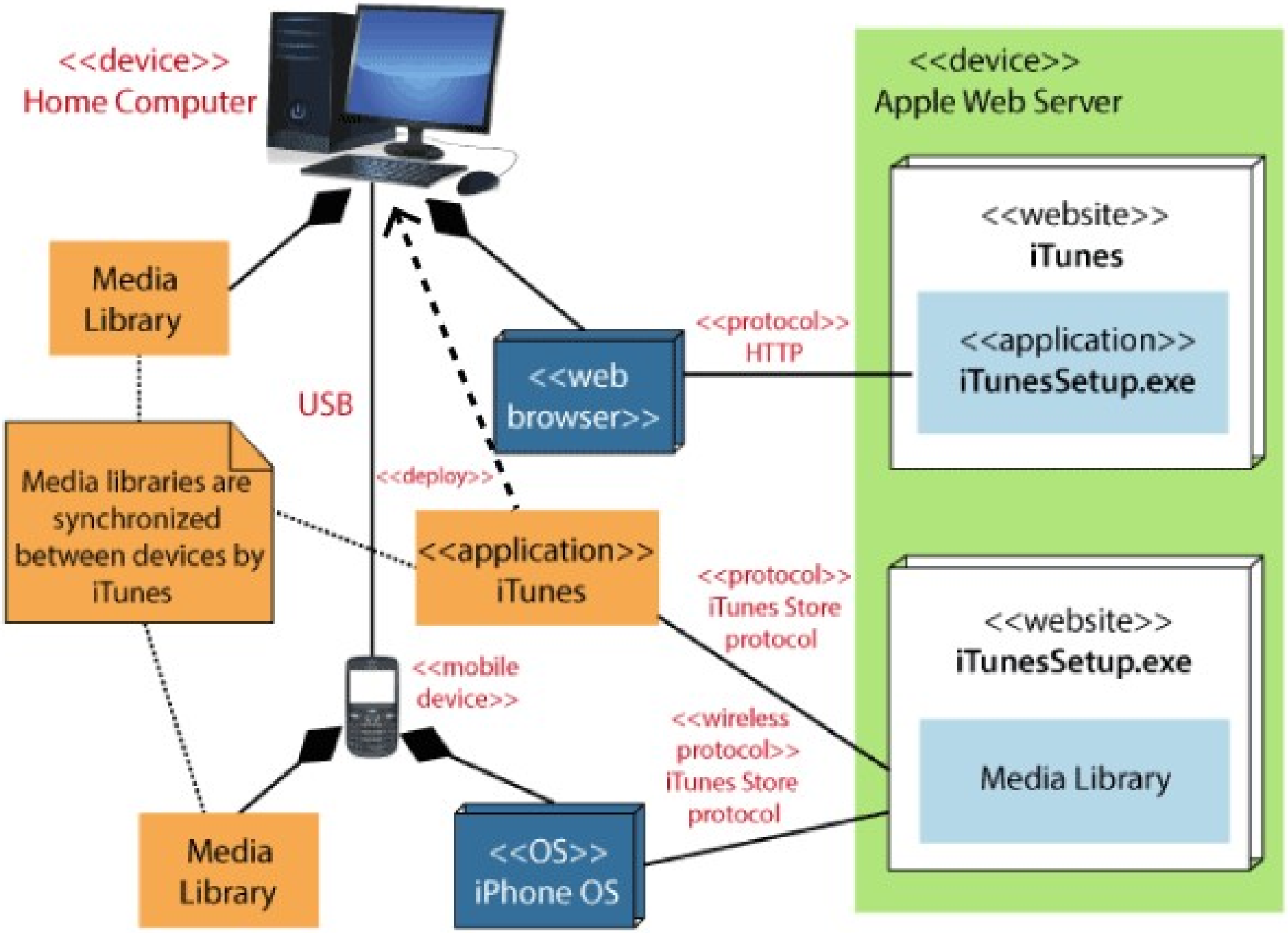
Instance level deployment diagram

- Instance level deployment diagram shows **deployment** of instances of **artifacts** to specific instances of **deployment targets**.
- It could be used for example to show differences in deployments to **development, staging or production environments with the names/ids of specific deployment servers or devices**

Instance level deployment diagram



Instance level deployment diagram - web application deployed to Tomcat JSP server and database schemas - to database system.


<<device>>
Home Computer
<<device>>
Apple Web Server
Media
Library
<<website>>
iTunes
<<application>>
iTunesSetup.exe
<<protocol>>
HTTP
<<web
browser>>
USB
Media libraries are
synchronized
between devices by
iTunes
<<deploy>>
<<application>>
iTunes
<<protocol>>
iTunes Store
protocol
<<website>>
iTunesSetup.exe
<<mobile
device>>
<<wireless
protocol>>
iTunes Store
protocol
Media Library
Media
Library
<<OS>>
iPhone OS

Profile Diagram

- Profile Diagrams in UML_are used to define custom extensions to UML models.
- They allow you to tailor UML to specific domains or platforms, making it more expressive for particular needs.
- Profiles enable you to create **stereotypes, tagged values, and constraints** that extend standard **UML elements**, thereby providing a more accurate representation of specific system requirements.

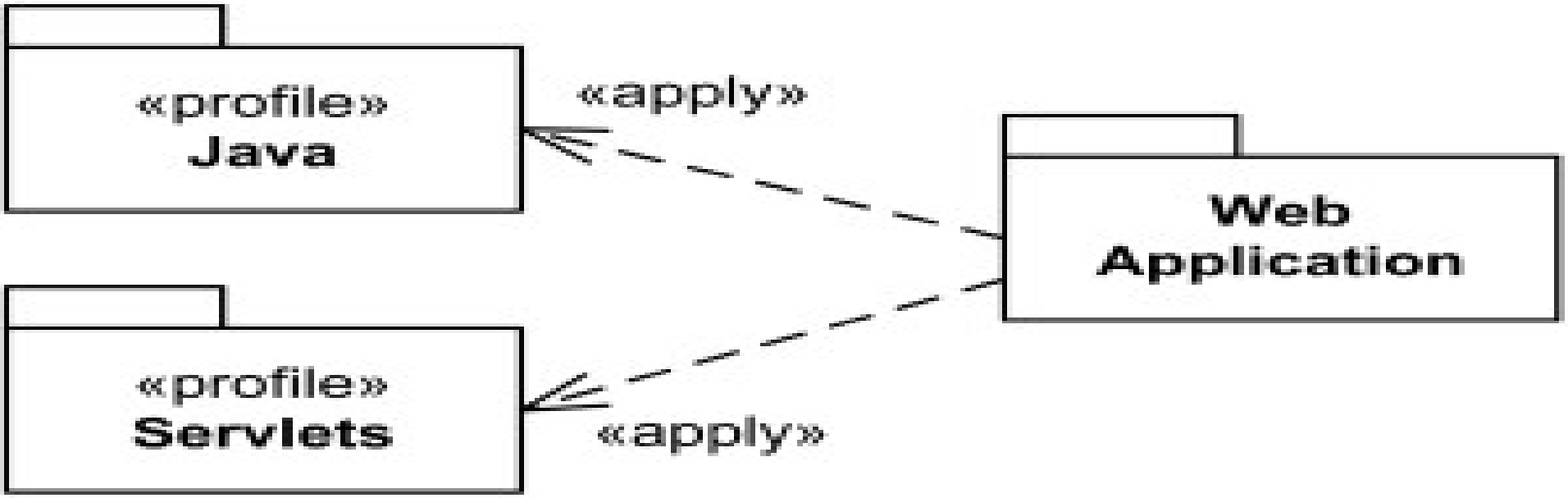
Importance of Profile Diagrams

Profile diagrams in UML are important because they:

- **Customize UML**: Allow customization for specific domains or industries.
- **Clarify Semantics**: Define precise meanings and behaviors of model elements.
- **Ensure Consistency**: Establish standards and enforce best practices.
- **Support Automation**: Enable tooling support for validation and code generation.
- **Promote Reusability**: Facilitate reuse across projects and scalability.
- **Enhance Communication**: Serve as clear documentation for stakeholders.

Components of a Profile Diagram

- **Profile -** This is the main container in the diagram that represents a collection of stereotypes, tagged values, and constraints.
- It serves as an extension mechanism to tailor UML for specific domains or purposes.



Profile diagram has three types of extensibility mechanisms:

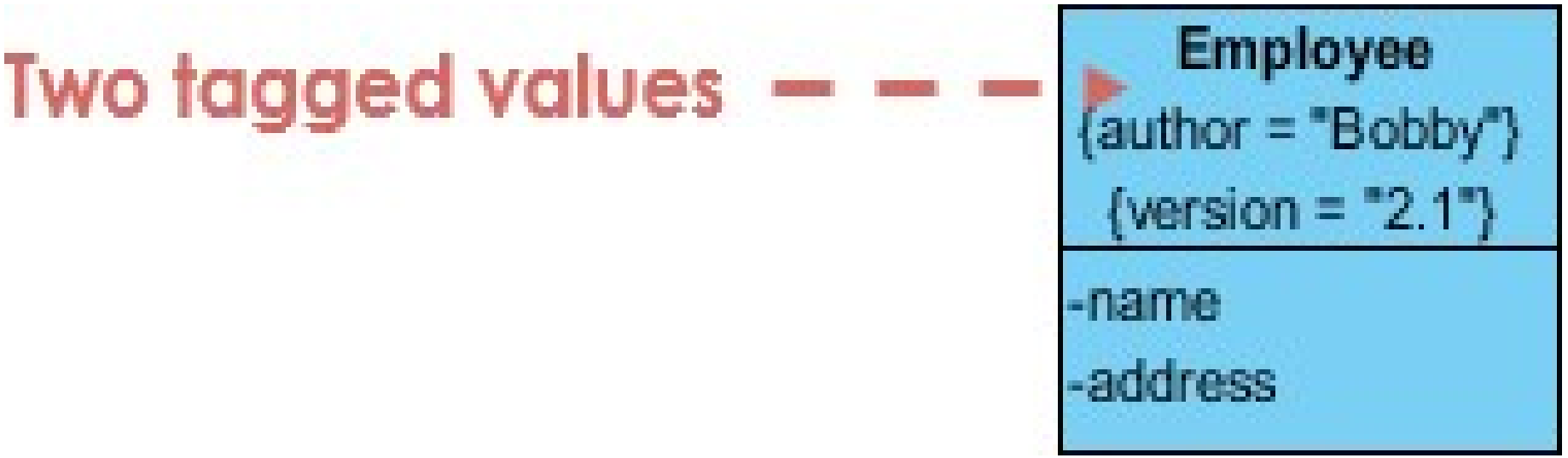
- **Stereotypes**
- **Tagged Values**
- **Constraints**
- **Stereotypes** - allow you to increase vocabulary of UML.
- To **add, create new model elements**, derived from **existing ones** but that have specific properties that are suitable to your problem domain.
- Stereotypes are used to introduce new building blocks that speak the language of your domain and look primitive.
- It allows you to introduce new **graphical symbols**.

- **Tagged values** are used to **extend the properties of UML** so that you can **add additional information** in the specification of a model element.
- It allows you to **specify keyword value pairs** of a model where keywords are the attributes.
- Tagged values are **graphically rendered as string** enclose in brackets.

  Ex. Consider a s/w release team responsible for assembling, testing and deployment of a system.

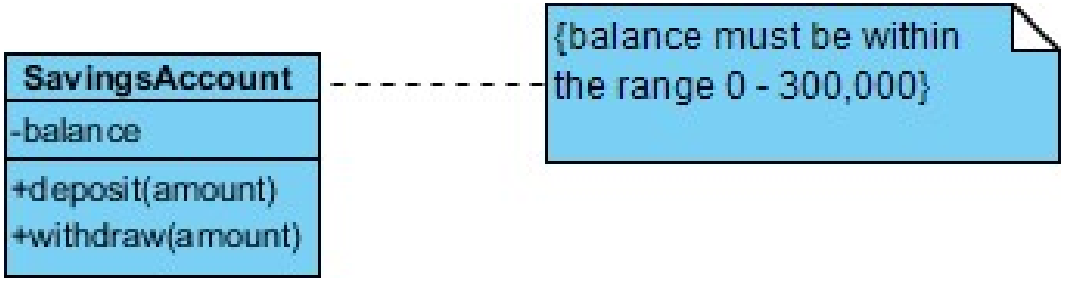
  In such case it is necessary to keep a track on version and test results of the main subsystem. Tagged values are used to add such info.

- Code generation
- Version control
- Configuration management
- Authorship
- Etc



- **Constraints -** They are the properties for specifying semantics or conditions that must be held true at all the time.
- It allows you to extend the semantics of UML building block by adding new protocols.
- Graphically a constraint is rendered as string enclose in brackets placed near associated element.

- For example: In development of a real time system it is necessary to adorn the model with some necessary information such as response time.
- A constraint defines a relationship between model elements that must be use *{subset}* or *{xor}*.
- Constraints can be on attributes, derived attributes and associations. It can be attached to one or more model elements shown as a note as well.



When to Use Profile Diagram - UML Extension Meacham

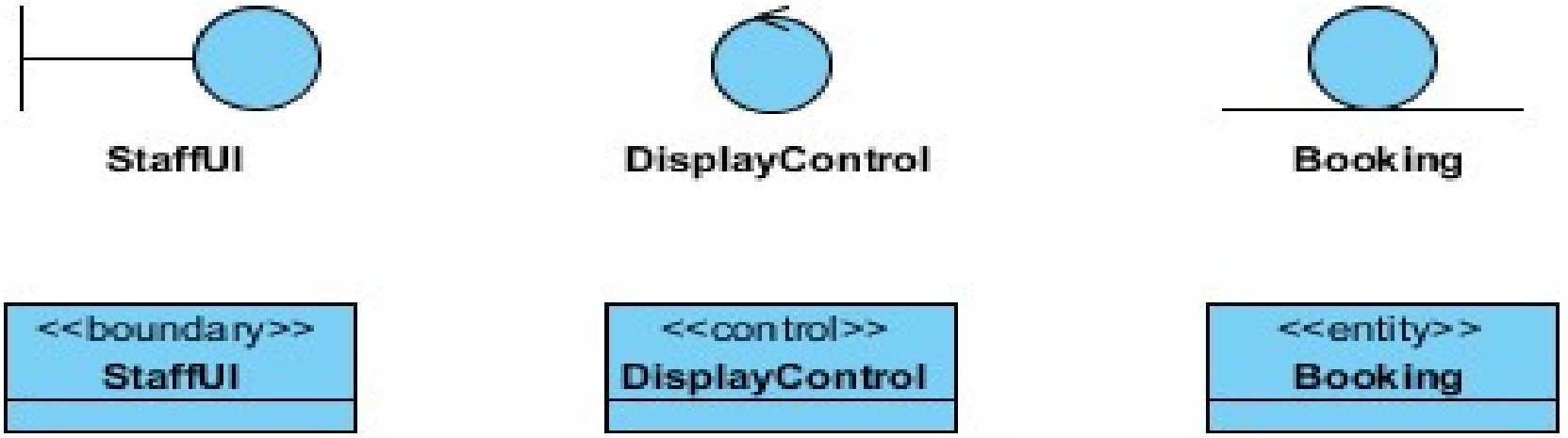
- As an alternative to creating a new meta model, you can also extend **Extension and modification of the UML meta model** and modify the UML meta-model in accordance with your requirements.
- In UML we call it **lightweight extensions** which are based on stereotypes and profiles.
- UML Profile can be defined in one of the following 3 ways:
- Creation of a new meta model
- Extension and modification of the UML meta model
- Extension of the UML meta model with language-inherent mechanisms

- **Profile Diagram - How it Works**
- The extension mechanism in UML 1.1 is relatively imprecise in the sense that extensions could be made only on the basis of the primitive data type string. UML 2.0 lets you use arbitrary data structures for extended elements, which means that more extensive and more precise model extensions are now possible.
- The profiles mechanism is not a first-class extension mechanism. It does **NOT allow** to:
- Modify existing metamodels
- Create a new metamodel like MOF does

Textual vs Graphic Icon Stereotype

Stereotypes can be in textual or graphical representation. The icon can also replace the normal class box.

For Example: People often use these 3 stereotyped class representations to model the software MVC framework:



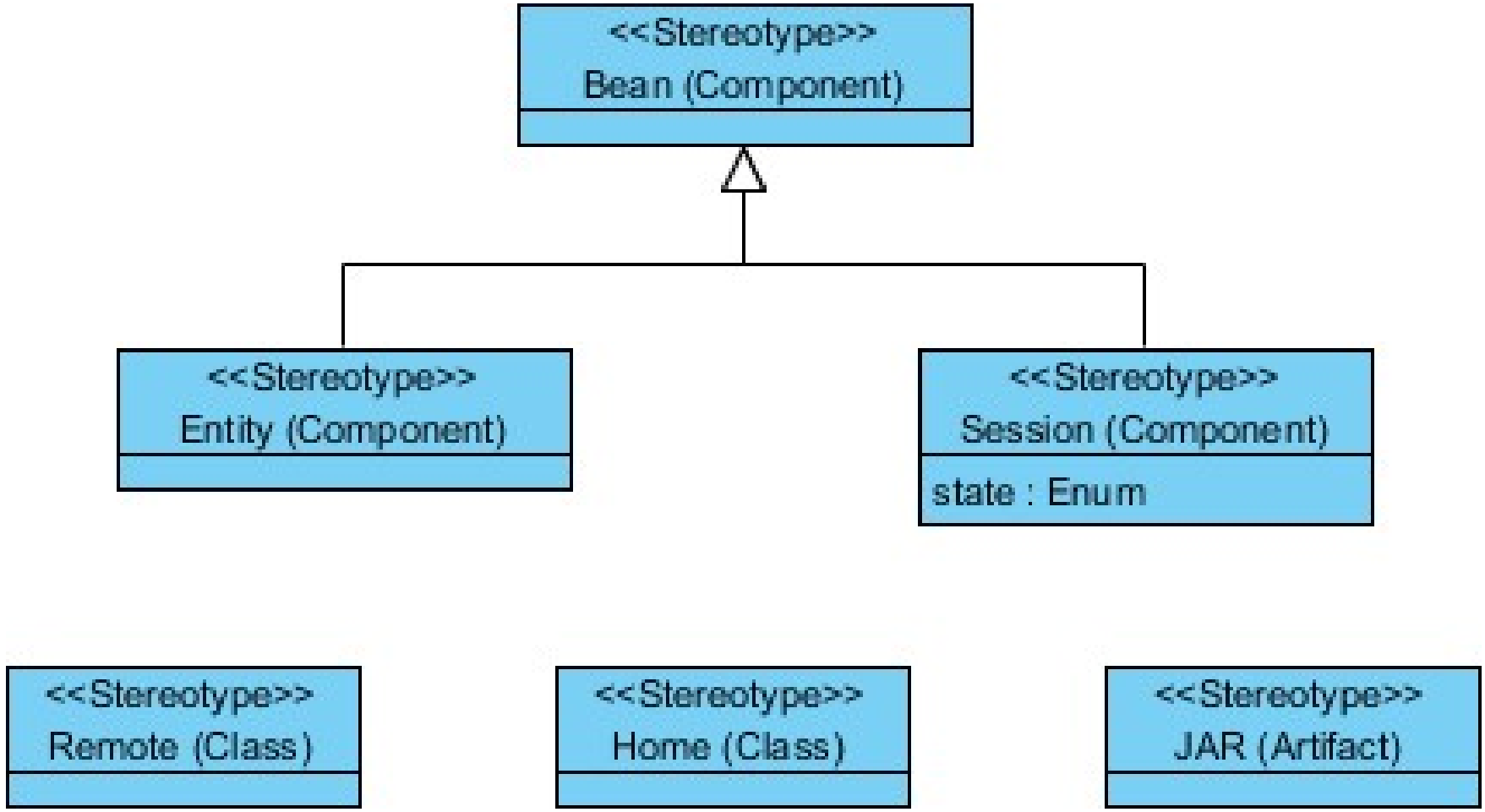
Popular Usages for UML profiles

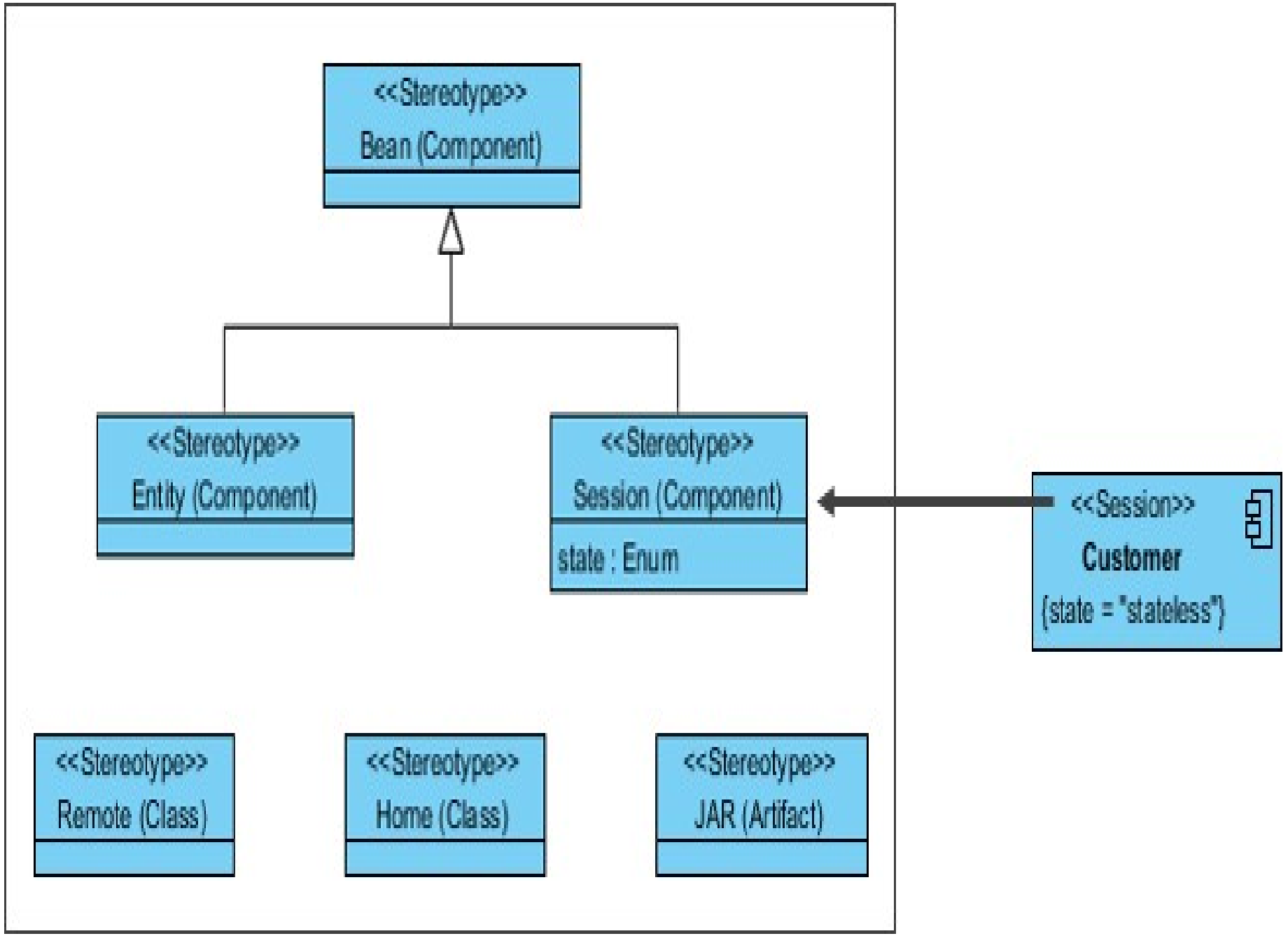
- Every technical target, i.e. programming language, middleware, library or database is a natural candidate for defining UML profile.

  For examples:

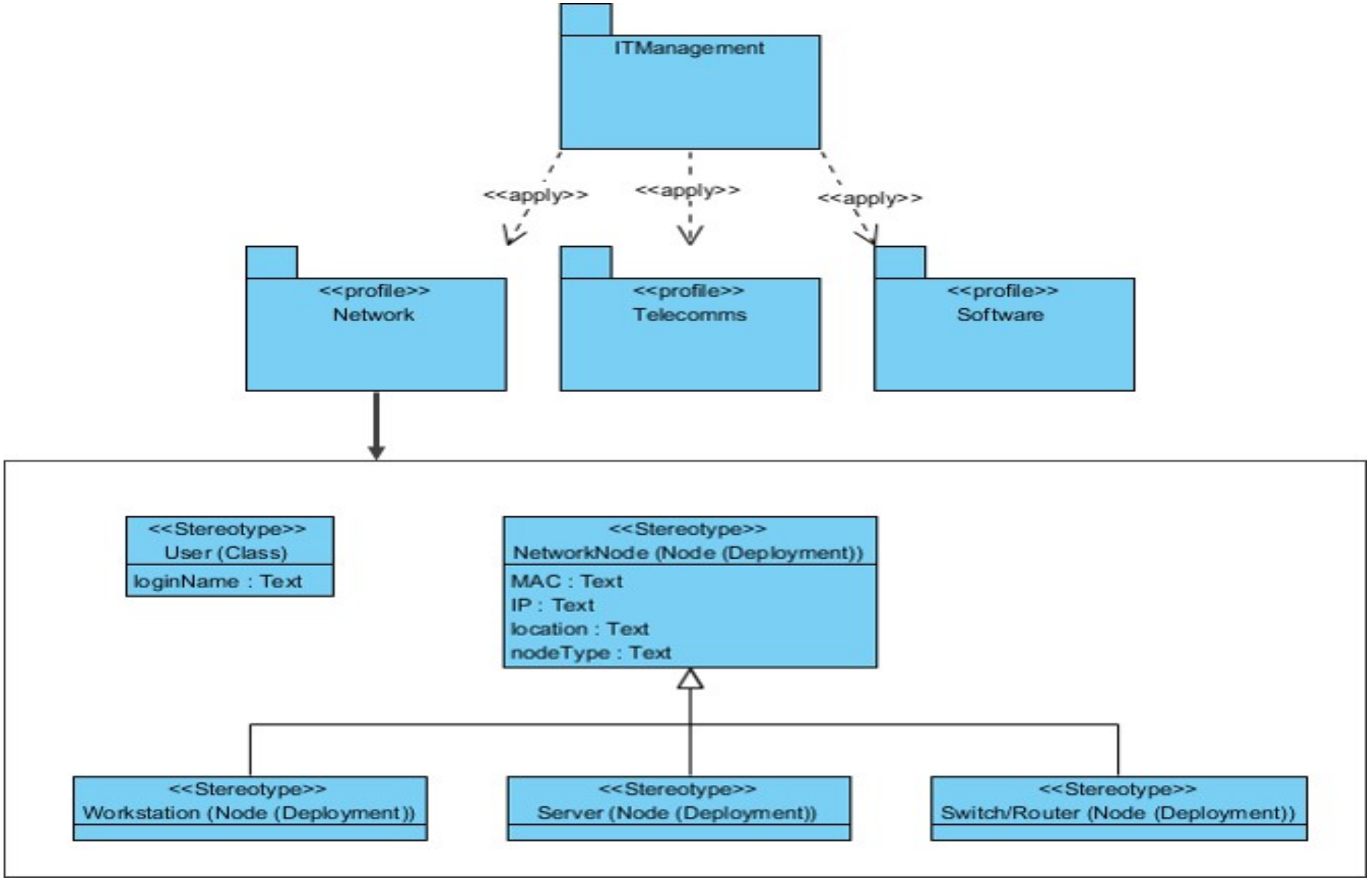
  - CORBA
  - EJB
  - C++ or JAVA
  - ORACLE or MYSQL
  - Etc

Profile Diagram at a Glance




<<Stereotype>>
Bean (Component)
<<Stereotype>>
Entity (Component)
<<Stereotype>>
Session (Component)
state : Enum
<<Session>>
Customer
{state = "stateless"}
<<Stereotype>>
Remote (Class)
<<Stereotype>>
Home (Class)
<<Stereotype>>
JAR (Artifact)

Profile Diagram – IT Management



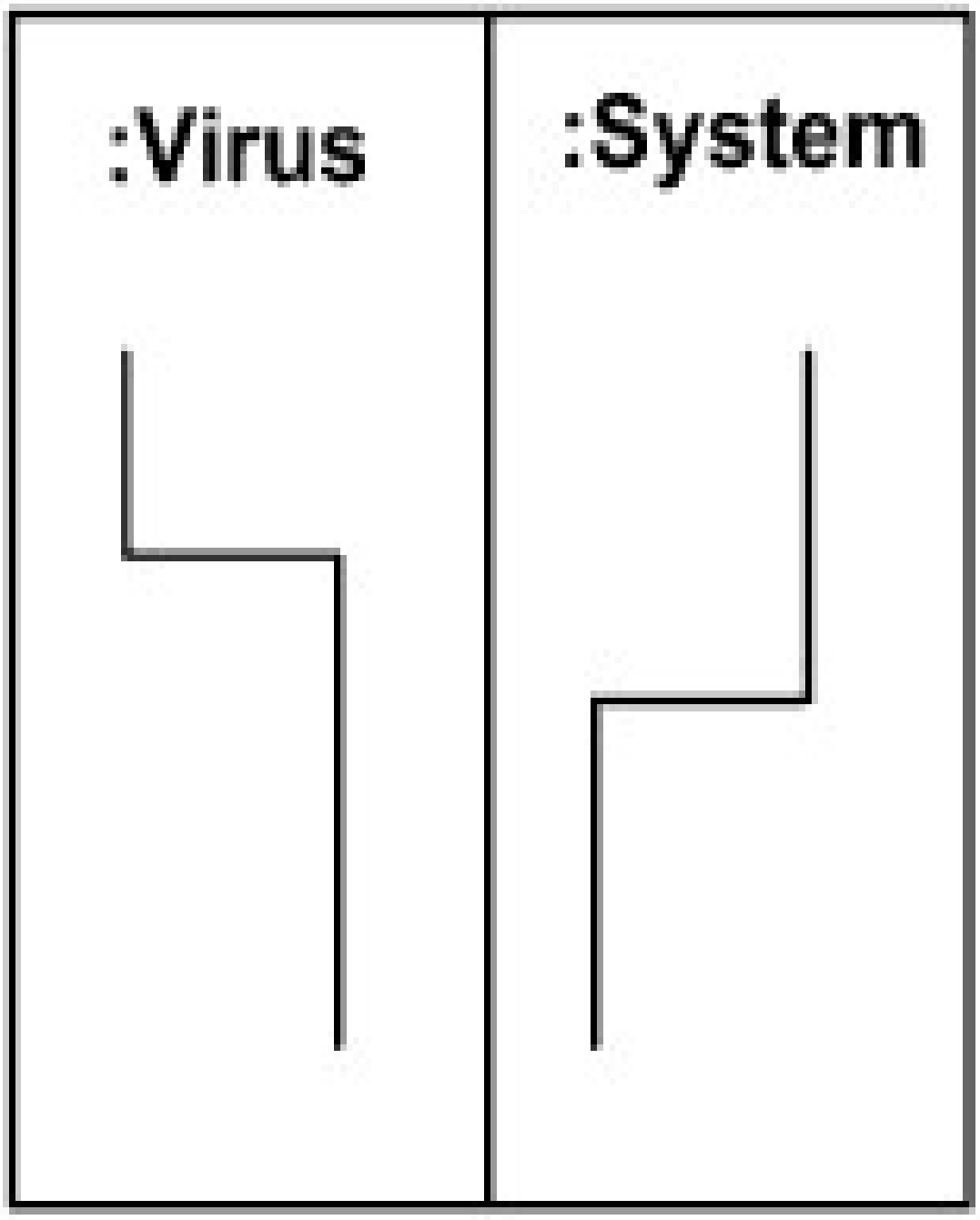
Interaction - Timing Diagram

- **Timing diagrams** are **UML interaction diagrams** used to show interactions when a primary purpose of the diagram is to reason about **time**.
- Timing diagrams focus on conditions changing within and among **lifelines** along a linear time axis.
- Timing diagrams describe behavior of both individual **classifiers** and interactions of classifiers, focusing attention on time of events causing changes in the modeled conditions of the lifelines.

Components of Timing Diagram

- **Lifeline** is a **named element** which represents an **individual participant** in the interaction. While **parts** and **structural features** may have multiplicity greater than 1,
  lifelines represent **only one** interacting entity.

- Lifeline on the timing diagrams is represented by the **name** of classifier or the instance it represents.
- It could be placed inside diagram frame or a "swimlane".


:Virus
:System

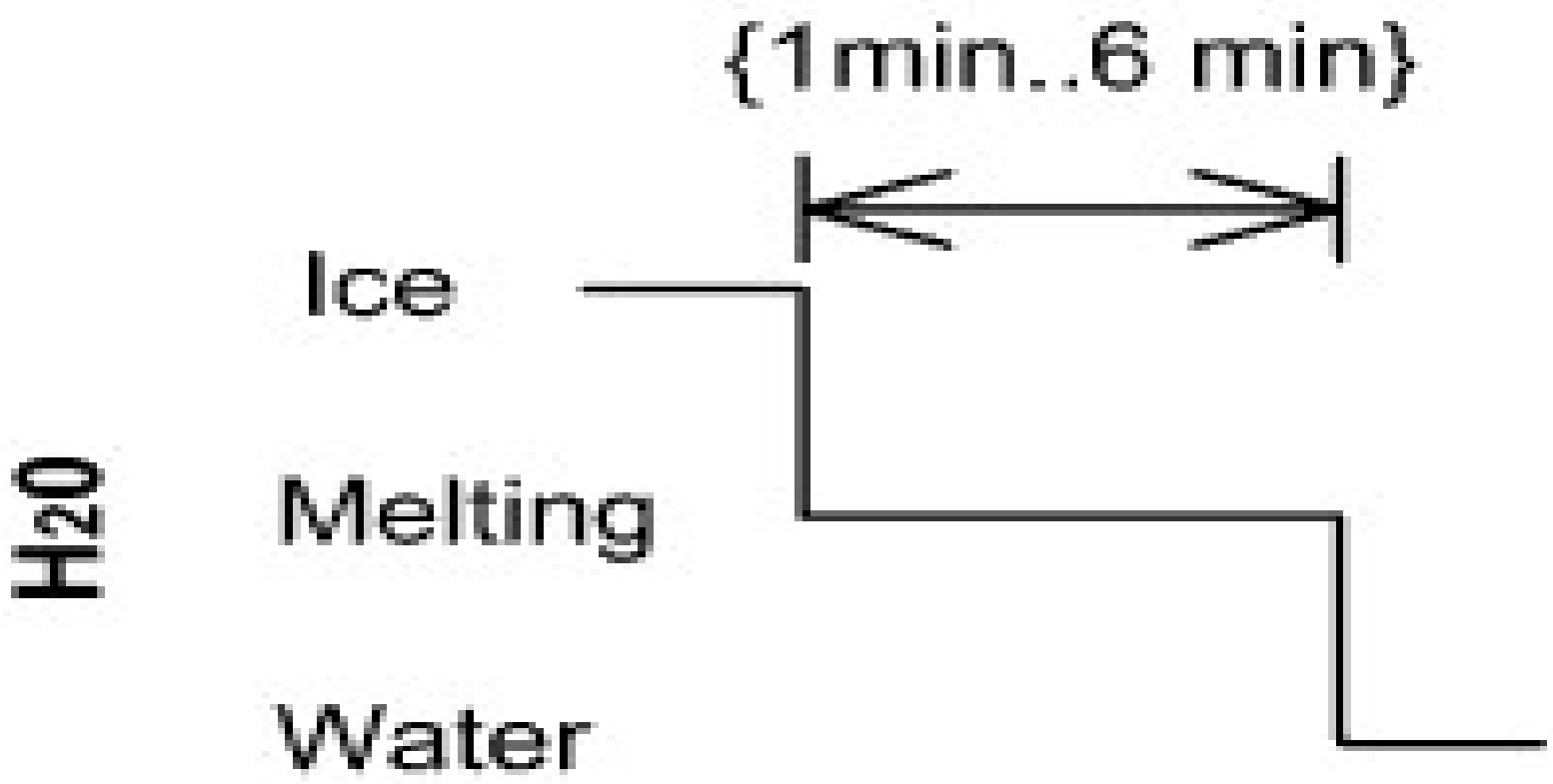
State or Condition Timeline

- Timing diagram could show **states** of the participating **classifier** or attribute, or some testable **conditions**, such as a **discrete or enumerable value** of an attribute.
- UML also allows the state/condition dimension be **continuous**. It could be used in **scenarios** where entities undergo **continuous state changes**, such as temperature or density.

Duration Constraint

- **Duration constraint** is an **interval constraint** that refers to a **duration interval**.
- The duration interval is duration used to determine whether the constraint is satisfied.
- The semantics of a duration constraint is inherited from constraints.
- If constraints are violated, traces become negative which means that system is considered as failed.
- Duration constraint is shown as some graphical association between a **duration interval** and the constructs that it constrains.

Duration Constraint



Time Constraint

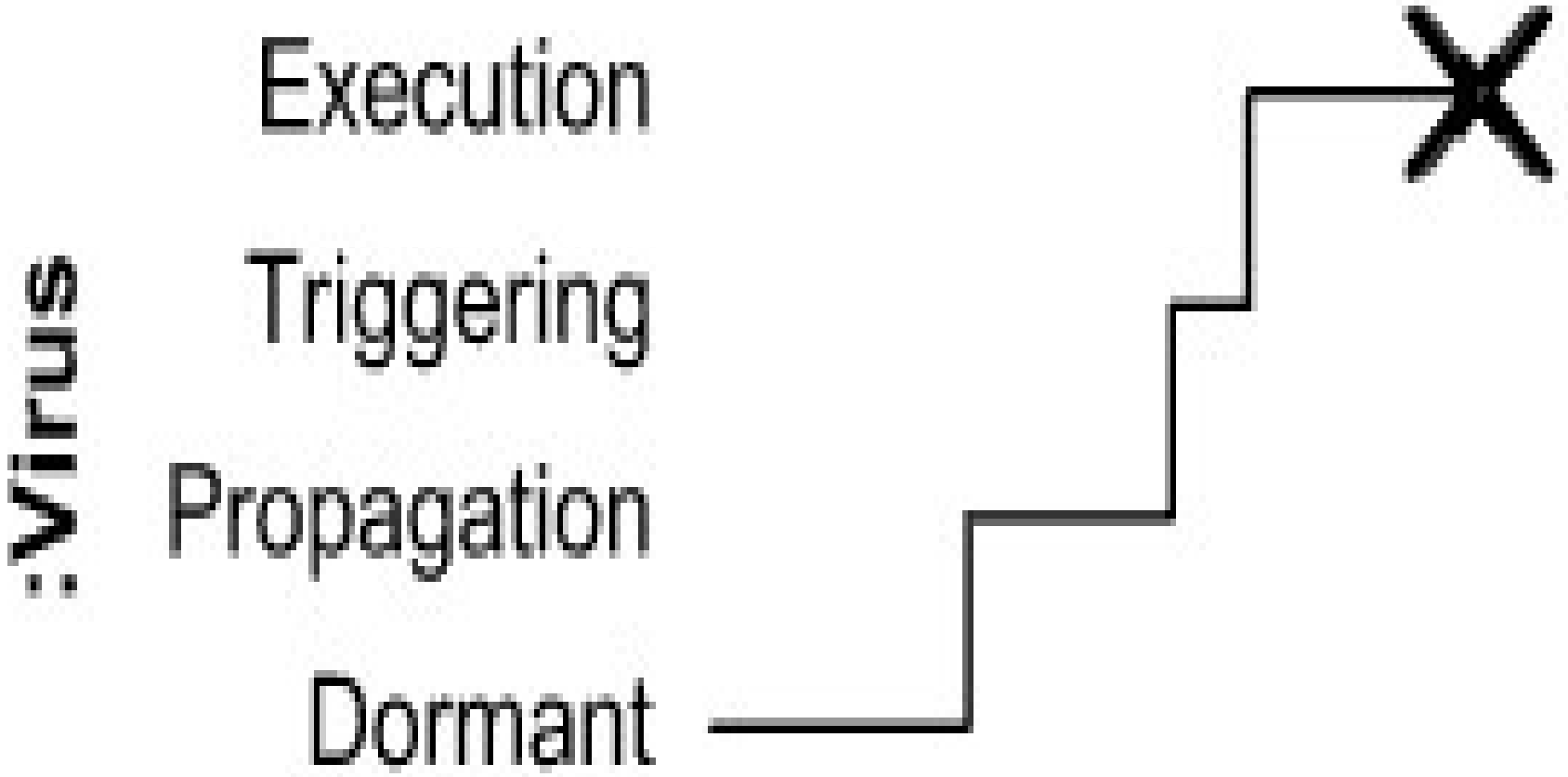
- **Time constraint** is an **interval constraint** that refers to a **time interval**.
- The time interval is **time expression used to determine whether the constraint is satisf**ied.
- The semantics of a time constraint is inherited from constraints.
- All traces where the constraints are violated are negative traces, i.e., if they occur, the system is considered as failed.
- Time constraint is shown as graphical association between a time interval and the construct that it constrains.
- e.g., between an occurrence specification and a time interval.

Destruction Occurrence

Destruction occurrence is a **message occurrence** which represents the destruction of the instance described by the **lifeline**.

It may result in the subsequent destruction of other objects that this object owns by **composition**. No other occurrence may appear after the destruction event on a given lifeline

Notation



MCQ’s for students – To check the understandability of the concepts

1. What is the Interaction diagram?
   a) Interaction diagrams are the UML notations for dynamic modeling of collaborations
   b) Interaction diagrams are a central focus of engineering design
   c) All of the mentioned
   d) None of the mentioned

2. What are the different interaction diagram notations does UML have?
   a) A sequence diagram
   b) A communication diagram
   c) An interaction overview diagram
   d) All of the mentioned

3. Which type of diagram is a sequence diagram?
   (A). Structural Diagram (B). Behavioral UML Diagrams
   (C). Interaction diagram (D)both Behavioral and Interaction diagram

4. Sequence diagram is belong to same type of diagrams as?

(A). Communication Diagram

(B). Timing Diagram

(C). both Communication Diagram and Timing Diagram

(D). Class Diagram

5.Sequence diagram have some different purpose as compared to which of the following diagram?

(A). State machine Diagram

(B). Interaction Diagram

(C). Class Diagram

(D). Use Case Diagram

6. What is a sequence diagram?
   a) A diagram that shows interacting individuals along the top of the diagram and messages passed among them arranged in temporal order down the page
   b) A diagram that shows messages superimposed on a diagram depicting collaborating individuals and the links among them
   c) A diagram that shows the change of an individual's state over time
   d) All of the mentioned

7. What is a lifeline?
   a) It is a frame consisting of a rectangle with a pentagon in its upper left-hand corner
   b) It is a rectangle containing an identifier with a dashed line extending below the rectangle
   c) It is a name compartment; the interaction is represented inside the rectangle
   d) None of the mentioned

8. What are the three different types of message arrows?
   a) Synchronous, asynchronous, asynchronous with instance creation
   b) Self, Multiplied, instance generator
   c) Synchronous, Asynchronous, synchronous with instance creation
   d) None of the mentioned

9. Which of the following is a kind of message that defines a particular communication between Lifelines of an Interaction.
   (a). Call message
   (b). Return Message
   (c). Self Message
   (d). Recursive Message

10. What does a simple name in UML Class and objects consist of?
   a) Letters
   b) Digits
   c) Punctuation Characters
   d) All of the mentioned

11. What Does a Composite name consists of in a UML Class and object diagram?
    a) Delimiter
    b) Simple names
    c) Digits
    d) All of the mentioned

12. A Class consists of which of these abstractions?
    a) Set of the objects
    b) Operations
    c) Attributes
    d) All of the mentioned

13. A class is divided into which of these compartments?
    a) Name Compartment
    b) Attribute Compartment
    c) Operation Compartment
    d) All of the mentioned

14. An attribute is a data item held by which of the following?
    a) Class
    b) Object
    c) All of the mentioned
    d) None of the mentioned

15. What should be mentioned as attributes for conceptual modelling?
    a) Initial Values
    b) Names
    c) All of the mentioned
    d) None of the mentioned

16. An operation can be described as?
    a) Object behavior
    b) Class behavior
    c) Functions
    d) Object & Class behavior

17. Activity diagram have some different purpose as compared to which of the following diagram?

(A). State machine Diagram

(B). Interaction Diagram

(C). Class Diagram

(D). Use Case Diagram

18. Which of the following Combines two concurrent activities and re-introduces them to a flow where only one activity can be performed at a time?

(A). Joint symbol

(B). Fork symbol

(C). Decision symbol

(D). Note symbol

19.Which of the following Demonstrates the acceptance of an event?

(A). Send signal symbol
(B). Receive signal symbol
(C). Decision symbol
(D). Note symbol

20. Which type of diagram is Activity diagram?

(A). Structural Diagram
(B). Behavioral UML Diagrams
(C). Both of these
(D). None of these

21.Activity diagram is belong to same type of diagrams as?

(A). Use Case Diagram
(B). State Machine Diagram
(C). Both of Use Case and State Machine Diagram
(D). Class Diagram

22. Which symbol that combine two parallel activities/actions into one activity/action or bring back together a set of parallel or concurrent flows of activities/actions?

23. Which of the following shows the order of states underwent by an object within the system?

(A). State Transition diagram
(B). Activity Diagram
(C). Sequence Diagram
(D). Class Diagram

24. Which of the following captures the software system's behavior?

(A). State Transition diagram
(B). Activity Diagram
(C). use case diagram
(D). All of these

25. Activity diagrams are good at showing how different actions can effect how the program will respond.

True
False

26. List some actions that might be found in an activity diagram for making cookies

271. When a fork is used, only one of the paths is followed.

True

False

28. Match the terms with the symbols used to represent them, if they were to be incorporated into an activity diagram. Please type your answers exactly.

1. The plan for the day depends on the weather.
2. Multiple employees work on a single fast food order.
3. The completed fast food order is given to the customer.
4. The person is about to start their task.

Initial node

Join

Fork

Decision node

Swimlanes

29. Which of the following diagram displays the structural relationship of components of a software?

- (A). Component Diagram
- (B). class diagram
- (C). use case diagram
- (D). sequence diagram

30. **Which of the following diagram is required when working with big and complex systems with many components?**

- (A). Component Diagram
- (B). class diagram
- (C). use case diagram
- (D). sequence diagram

31. In Component Diagram, Components communicate with each other using which of the following?

- (A). Components
- (B). interfaces
- (C). Use cases
- (D). Attributes

32. Which of the following diagram describe the organization of source code, binary code, and executable?

- (A). Component Diagram
- (B). class diagram
- (C). use case diagram
- (D). sequence diagram

33. The interfaces in component diagrams are linked using which of the following?

- (A). connectors
- (B). interfaces
- (C). Components
- (D). None of these

34. Which of the following diagram is used to illustrate the structure of arbitrarily complex systems?

(A). Component Diagram

(B). class diagram

(C). sequence diagram

(D). Data flow diagram

35. Which aspect is shared by Component Diagrams and Deployment Diagrams?

a) Both model the sequence of interactions

b) Both show the physical architecture of the system

c) Both focus on the logical grouping of elements

d) Both illustrate the dynamic behavior of objects

36. Deployment diagram consists _____?

(A) Computational resource
(B) Communication path in the mid of resource
(C) Artifacts that run resource
(D) All of the above
(E) None of these

The End…..