Chapter-3
Structural, Behavioral and architectural Modeling

- ✓ Classes Relationship,
- ✓ Conceptual Model of UML
- ✓ Class diagram
- ✓ Advanced classes
- ✓ Advanced Relationship
- ✓ Interface
- ✓ Object Diagram
- ✓ Interactions
- ✓ Use cases
- ✓ Use Case Diagram
- ✓ Interaction Diagram
- ✓ Activity Diagram
- ✓ State chart Diagram
- ✓ Component and Components Diagram
- ✓ Deployment Diagram

Conceptual Model of UML

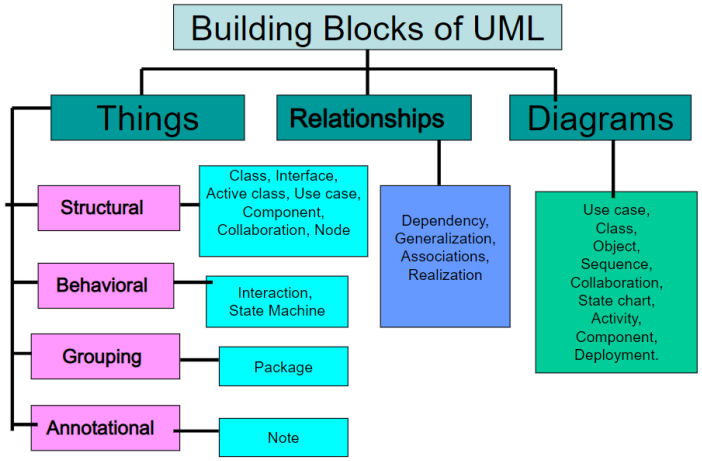
- It can be defined as a model which is made of concepts and their relationships.
- A conceptual model is the first step before drawing UML diagram. It helps to understand the entities in the real world and how they interact with each other.
- As UML describes the real time systems, it is very important to make a conceptual model and then proceed gradualy.

Conceptual Model of UML

The conceptual model of UML contains the fundamentals of UML. It consists of three parts. They are:

- ❑**Building blocks** of UML (syntax / vocabulary)
- ❑**Rules** (semantics) that dictate how these building blocks may be put together
- ❑**Common Mechanisms** that apply throughout the UML.

Building Blocks of UML



Building Blocks of UML

UML consists of three kinds of building blocks:

Things: Things are used to describe different parts of a system: existing types of things in UML are presented in table.

There are 4 kinds of things in the UML

1. Structural Things
2. Behavioral Things
3. Grouping Things
4. Annotational Things

Types of things

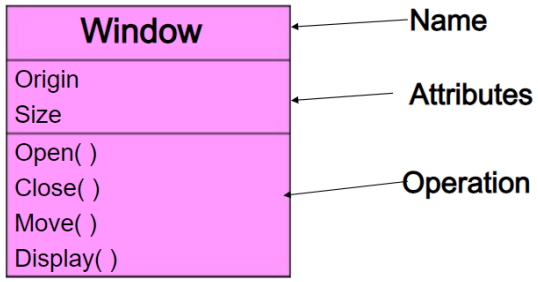
1. **Structural things:** They are nouns of UML model. These are the static parts that represent elements that are either conceptual or physical. It includes class, interface, collaboration, use case, active class, component, and node.
2. **Behavioral things:** They are dynamic parts of UML model. It includes interaction, state machine .
3. **Grouping things:** They are the organizing parts of UML model. It includes package.
4. **Annotational things:** They are the explanatory parts of UML model. It includes notes.

Structural things

a. Class:

- A class is a description of a set of objects that share the same attributes, operations, relationships and semantics.
- A class implements one or more interfaces.

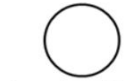
class



Structural things

b. Interface:

- Interface is a collection of operations that specify a service of a class or component.
- An interface describes the externally visible behavior of that element.
  An interface defines a set of operation specifications but never a set of operation implementations.
- Graphically, an interface is represented as a circle with its name.



Structural things

c. Use case:

A use case is a collection of actions, defining the interactions between a role (actor) and the system.

Graphically use case is represented as a solid ellipse with its name written inside or below the ellipse.

Structural things

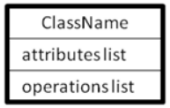
d. Collaboration:

- A collaboration is the collection of interactions among objects to achieve a goal.
- Graphically collaboration is represented as a dashed ellipse.
- A collaboration can be a collection of classes or other elements.

Structural things

e. Active class:

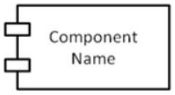
- An active class is a class whose objects own one or more processes or threads and therefore can initiate control activity.
- It initiates and controls the flow of activity, while passive classes store data and serve other classes.
- Graphically active class is represented as a rectangle with thick borders.

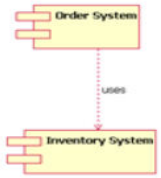


Structural things

f. component:

- A component is a physical and replaceable part of a system.
- Graphically component is represented as a tabbed rectangle.
- Examples of components are executable files, database tables, files and documents.

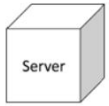




Structural things

g. Node:

- A node is a physical element that exists at run time and represents a computational resource.
- Graphically node is represented as a cube.
- Examples of nodes are PCs, laptops, smartphones or any embedded system.



Behavioral things

- These are the dynamic parts of UML models.
- These are the verbs of a model representing the behavior over time and space.
- There are two types of behavioral things:

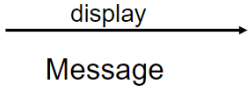
i. **Interaction**

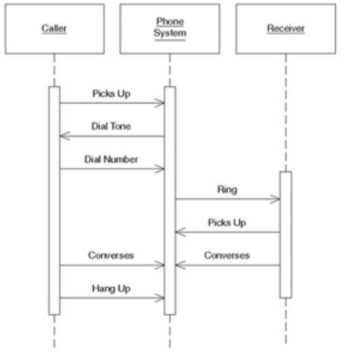
ii. **State machine**

Behavioral things

i. Interaction:

- An Interaction is a behavior that comprises a set of messages exchanged among a set of objects within a particular context to perform a specific purpose.
- An interaction involves a number of other elements, including messages, action sequences and links (connection between objects.)

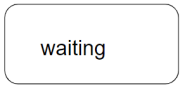


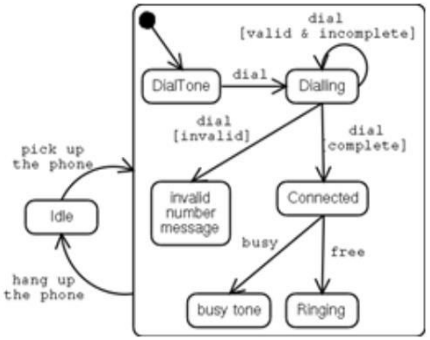


Behavioral things

ii. State machine:

- State machine is a behavior that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events.
- It involves a number of other elements, including states, transitions(the flow from state to state), events (things that trigger a transition), and activities (the response to a transition).



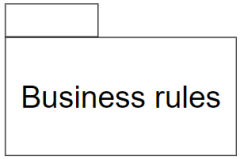


Grouping Things

- Grouping things are the organizational parts of UML models.
- In all, there is one primary kind of grouping thing, namely, Package.

Package:

- A Package is a general-purpose mechanism for organizing elements into groups.
- Structural things, behavioral things and even other grouping things that may be placed in a package.
- Unlike component (which exist at run time), package is purely conceptual (meaning that it exist only at development time).



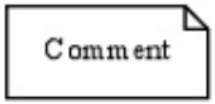
Annotational Things

❑They are the explanatory parts of the UML models.

❑These are the comments you may apply to describe, illuminate and remark about any element in a model.

Note:

- Note is simply a symbol for representing constraints and comments attached to an element or a collection of elements.
- It is rendered as a rectangle with a dog-eared corner.



Relationships in UML

- There are four kinds of relationships in the UML.

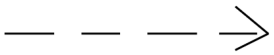
i. **Dependency**
ii. **Association**
iii. **Generalization**
iv. **Realization**

- These relationships are the basic relational building blocks of the UML.
- These are used to write well-formed models.

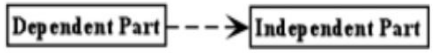
Relationships in UML

i. Dependency:

- A semantic relationship, in which a change in one thing (the independent thing) may cause changes in the other thing (the dependent thing).
- This relationship is also known as “using” relationship.
- It is graphically represented as dashed line with stick arrow head.



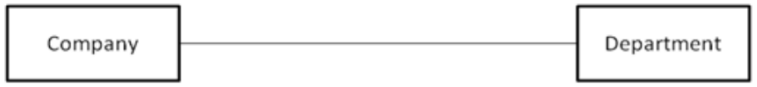
Dependencies



Relationships in UML

ii. Association:

- A structural relationship describing connections between two or more things.
- Graphically represented as a solid line with optional stick arrow representing navigation.



Relationships in UML

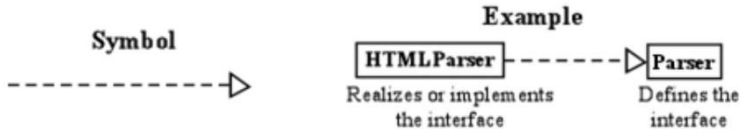
iii. Generalization:

- It is a specialization/generalization relationship in which objects of the specialized elements (the child) are substitutable for objects of the generalized elements (the parent).
- In this way, the child shares the structure and the behavior of the parent.
- Represented as a solid line with a hollow arrow head pointing to the parent.

Relationships in UML

iv. Realization:

- Defines a semantic relationship in which one class specifies something that another class will perform.
- Example: The relationship between an interface and the class that realizes or executes that interface.



Diagrams in UML

- A diagram is the graphically presentation of a set of elements, most often represented as a connected graph of vertices (things) and arcs (relationships).
- Diagrams visualize a system from different perspectives, so a diagram is a projection into system.
- It includes nine diagrams:

1.Class diagram 2.Object diagram
3.Use case diagram 4.Sequence diagram
5.Collaboration diagram 6.State chart diagram
7.Activity diagram 8.Component diagram
9.Deployment diagram

Diagrams in UML

- **Use case diagrams:** shows a set of use cases, and how actors can use them
- **Class diagrams:** describes the structure of the system, divided in classes with different connections and relationships
- **Sequence diagrams:** shows the interaction between a set of objects, through the messages that may be dispatched between them
- **State chart diagrams:** state machines, consisting of states, transitions, events and activities
- **Activity diagrams:** shows the flow through a program from an defined start point to an end point
- **Object diagrams:** a set of objects and their relationships, this is a snapshot of instances of the things found in the class diagrams
- **Collaboration diagrams:** collaboration diagram emphasize structural ordering of objects that send and receive messages.
- **Component diagrams:** shows organizations and dependencies among a set of components. These diagrams address static implementation view of the system.
- **Deployment diagrams:** show the configuration of run-time processing nodes and components that live on them.

Rules of UML

- The rules of UML specify how the UMLs building blocks come together to develop diagrams.
- The rules enable the users to create well-formed models. A well-formed model is self-consistent and also consistent with the other models.
- UML has rules for:
  - *Names* – What elements can be called as things, relationships and diagrams
  - *Scope* – The context that gives a specific meaning to a name
  - *Visibility* – How these names are seen and can be used by the other names
  - *Integrity* – How things properly relate to one another
  - *Execution* – What it means to run or simulate a model

Common Mechanisms in UML

- The 4 common mechanisms that apply consistently throughout the language.

1. **Specifications**
2. **Adornments**
3. **Common Divisions**
4. **Extensibility mechanisms**

Common Mechanisms in UML

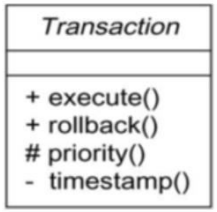
1. **Specifications:**

- Behind every graphical notation in UML there is a precise specification of the details that element represents.
- For example, a class icon is a rectangle and it specifies the name, attributes and operations of the class.

Common Mechanisms in UML

2. Adornments:

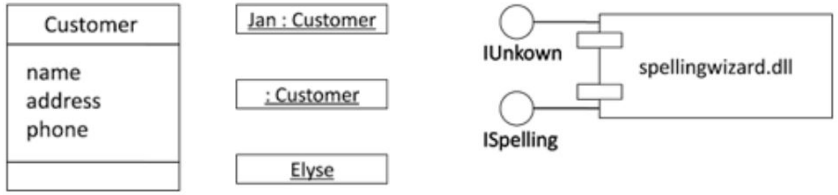
- The mechanism in UML which allows the users to specify extra information with the basic notation of an element is the adornments.
- In given example, the access specifiers: + (public), # (protected) and – (private) represent the visibility of the attributes which is extra information over the basic attribute representation.



Common Mechanisms in UML

3. Common Divisions:

- In UML there is clear division between semantically related elements like: separation between a class and an object and the separation between an interface and its implementation.



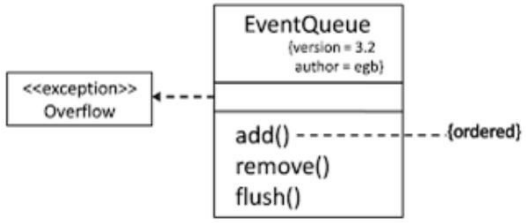
Common Mechanisms in UML

4. Extensibility Mechanisms:

- UMLs extensibility mechanisms allow the user to extend (new additions) the language in a controlled way. The extensibility mechanisms in UML are:
  - **Stereotypes:** Extends the vocabulary of UML. Allows users to declare new building blocks (icons) or extend the basic notations of the existing building blocks by stereotyping them using guillemets.
  - **Tagged Values:** Extends the properties of an UML building block. Allows us to specify extra information in the elements specification. Represented as text written inside braces and placed under the element name. The general syntax of a property is:

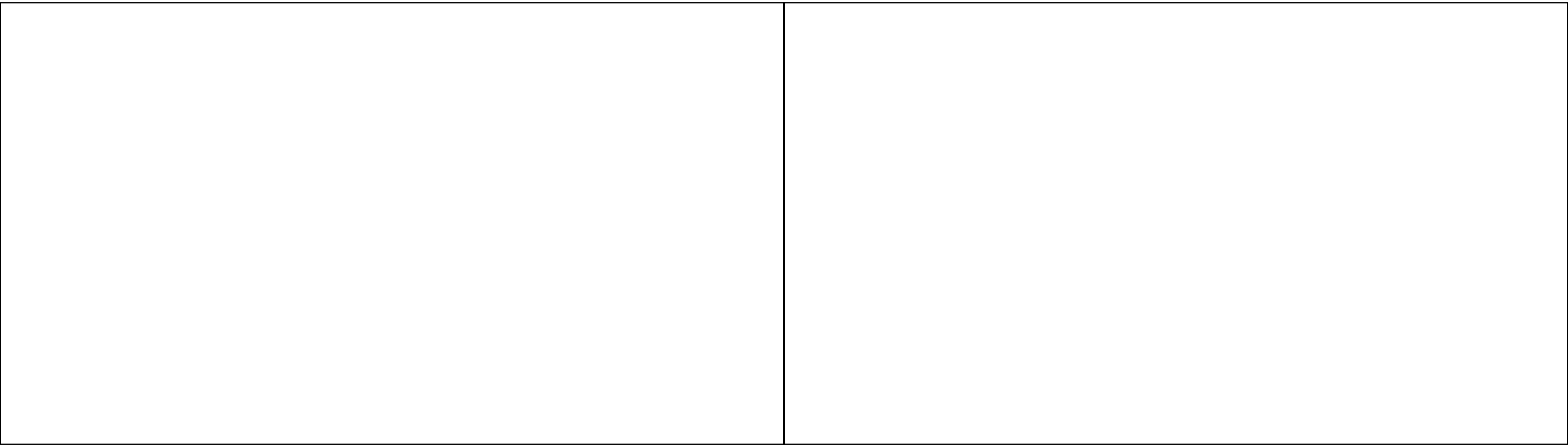
    { property name = value }
  - **Constraints:** Extends the semantics of a UMLs building block such as specifying new rules or modifying existing rules. Represented as text enclosed in braces and placed adjacent or beside the element name.

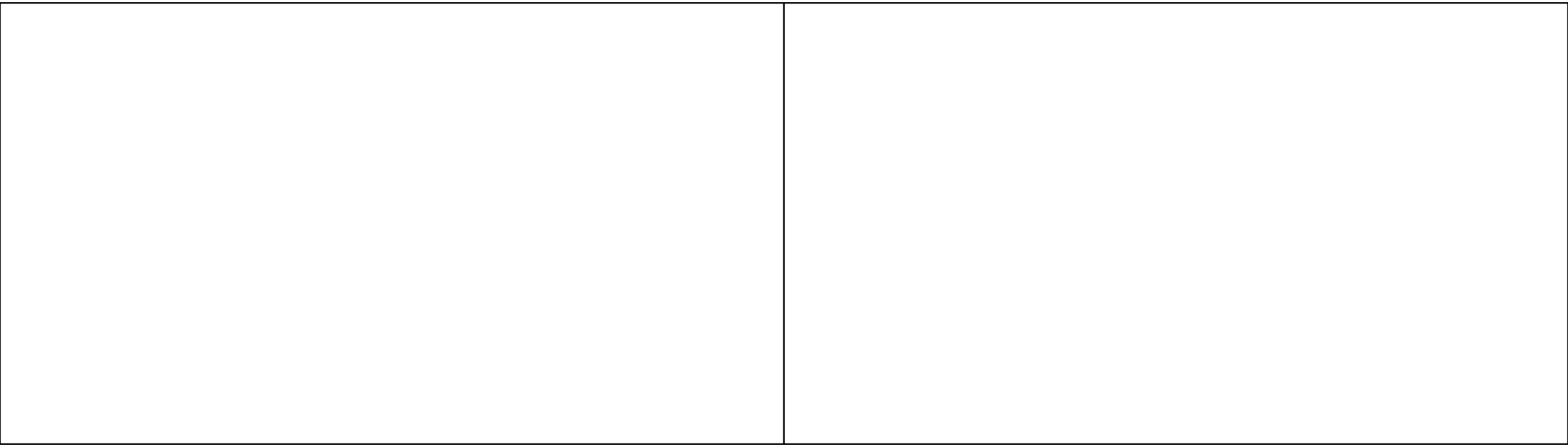
Common Mechanisms in UML

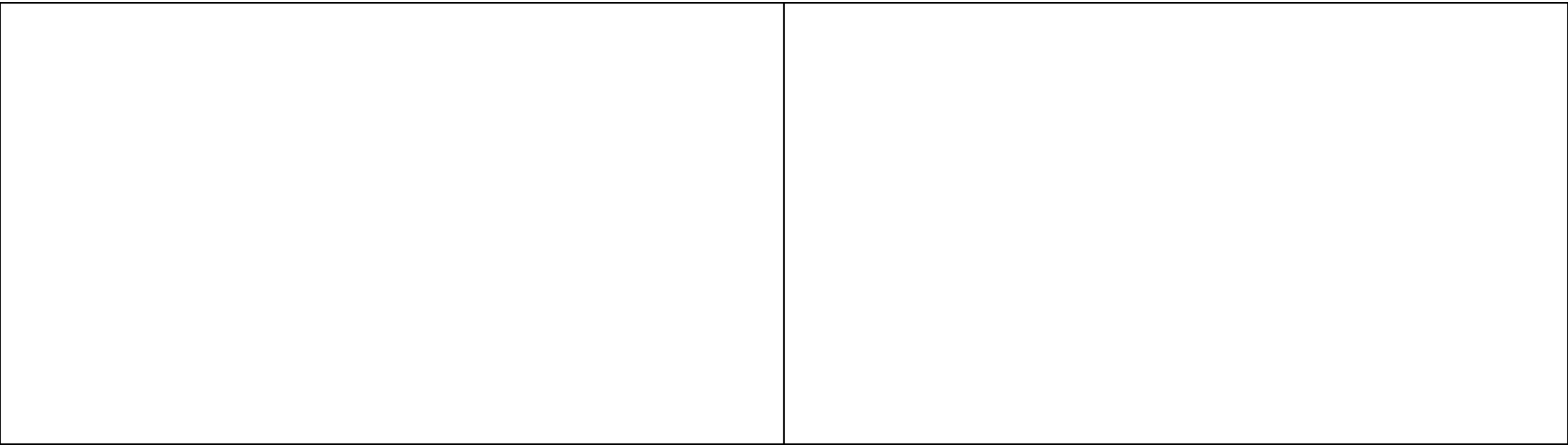


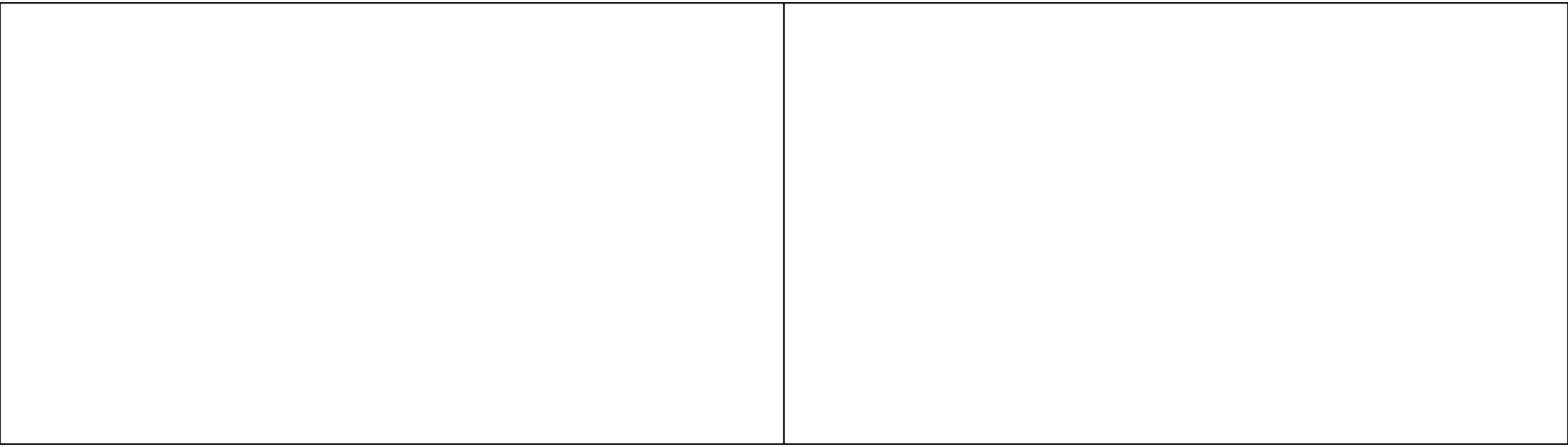
In the above example, we are specifying the exception “Overflow” using the class symbol and stereo typing it with “exception”. Also under the class name, “EventQueue” we are specifying additional properties like “version” and “author” using tagged values.

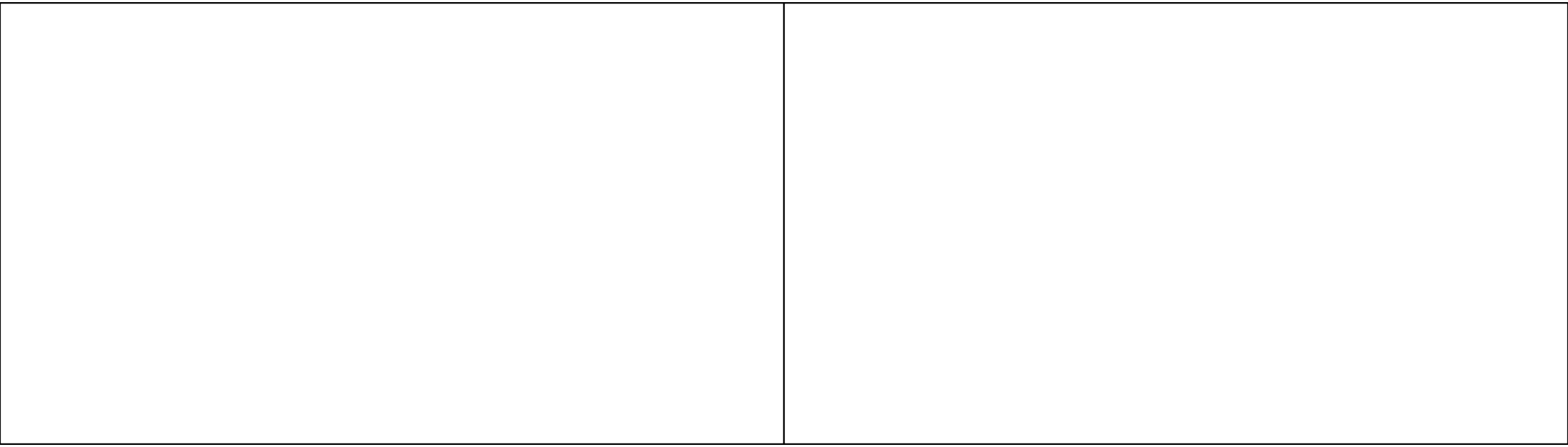
Finally, we are constraining the operation named “add” that before adding a new event to the EventQueue object, all the events must be “ordered” in some manner. This is specified using constraints in UML.

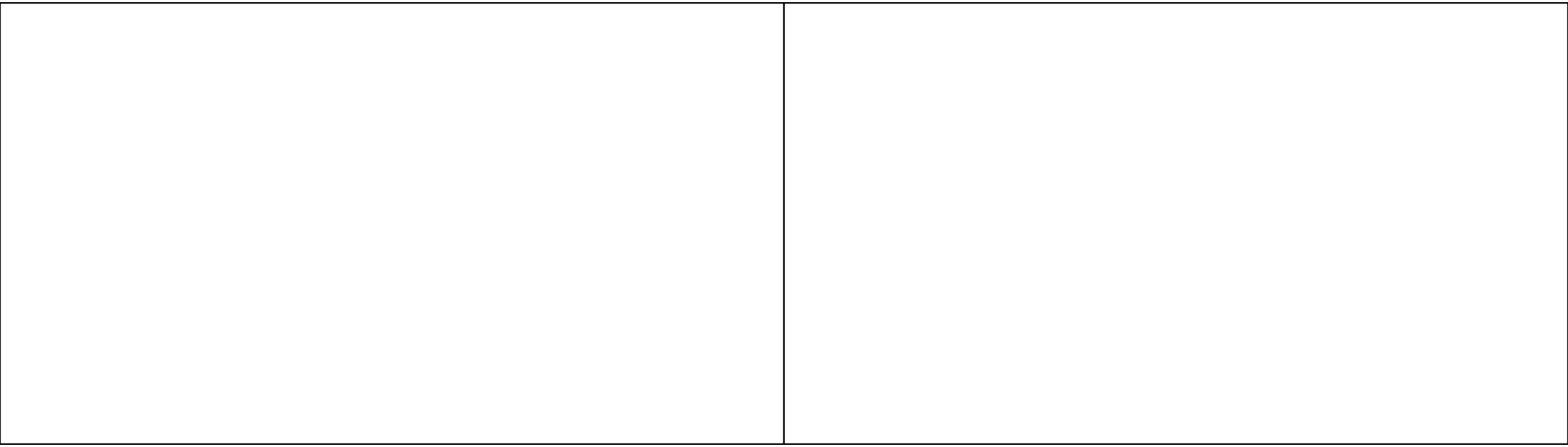












Conceptual Model

Conceptual Model/Domain Model

- It is a representation of real-world concepts, not software components.
- A domain model is a visual representation of conceptual classes or real-situation objects in a problem domain.
- It is not a set of diagrams describing software classes, or software objects and their responsibilities.
- Domain models have also been called **conceptual models, domain object models, and analysis object models.**
- According to unified Process, the domain model is a part of business modeling. Hence, domain model is also called **business object model**.

Conceptual Model/Domain Model

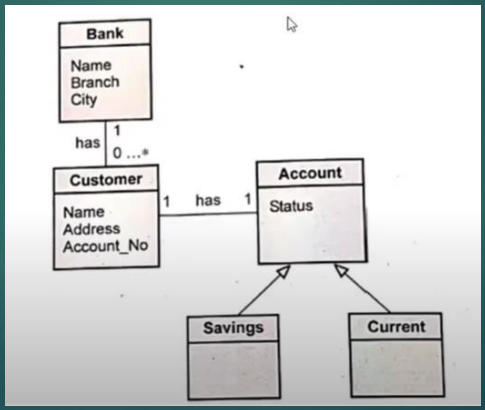
- The Domain Model is your organized and structured knowledge of the problem. The Domain Model should represent the vocabulary and key concepts of the problem domain and it should identify the relationships among all of the entities within the scope of the domain
- A domain model is the most important and classic model in OO analysis.
- It illustrates noteworthy concepts in a domain.
- It can act as a source of inspiration for designing some software objects and will be an input to several artifacts explored in the case studies.

Conceptual Model/Domain Model

- **Applying UML notation**, a domain model is illustrated with a set of **class diagrams** in which no operations (method signatures) are defined. Thus the domain model represents:
  - Domain object or conceptual classes
  - Association or relationship between conceptual classes
  - Attributes of conceptual classes

Conceptual Model/Domain Model

- Consider a banking system in which a bank has one or more branches. Each branch has customer which account is present in that branch. The account can be of two types saving accounts and current account. Design the domain model.



Need of Domain Model

- Gives a conceptual framework of the things in the problem space.
- Focuses on Semantics(logical part) rather than syntax
- Provides a glossary of terms- noun based (Id, Name for attributes)
- It is a static view – meaning it allows us to convey time invariant business rules.
- Foundation for use case/ workflow modeling
- Based on the defined structure, we can describe the state of the problem domain at any time.

Conceptual Model

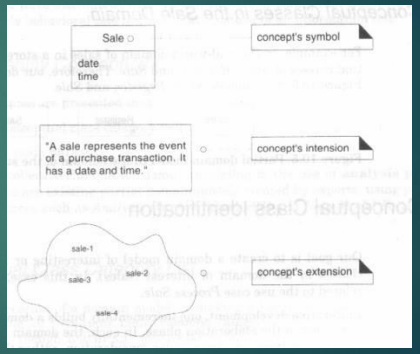
- It provides the conceptual perspective of the problem.
- Not part of the UML
- It may show: domain objects or conceptual classes, associations between conceptual classes and attributes of conceptual classes
- Steps to build a domain/conceptual model:
  - Find Conceptual Classes/ Domain Entities
  - Add associations
  - Add attributes

How to make a domain model?

- Prepare the conceptual Class Category List
- Use noun phrase identification technique
- Draw these objects in a domain model
- Add necessary associations
- Add required attributes

Conceptual Class

- They are either an idea, or thing or an object, identified as a result of different level of abstraction of the domain.
- They have three parts:
  - **Symbol:** usually represented by a rectangle
  - **Intension:** The definition of the class.
  - **Extension:** It is object instances of the classes. (generally absent).



Guidelines for identifying conceptual classes

- Try to identify as much of the conceptual classes as possible.
- Do not worry about missing some of the conceptual classes, but make sure that they will be incorporated later when they are identified in the process of identifying attributes and associations among domain objects.
- Include in the model also those conceptual classes that are not indicated clearly by the requirements ·
- Classes not having attributes or having only behavioral role could also be taken as a conceptual class.

Features of Domain Model

- **Domain class:** each domain class denotes a type of object.
- **Attributes:** an attribute is the description of a named slot of a specified type in a domain class; each instance of the class separately holds a value.
- **Associations:** an association is a relationship between two (or more) domain classes that describes links between their object instances. Association can have roles, describing the multiplicity and participation of a class in that relationship.
- **Additional rules:** complex rules that cannot be shown with symbolically can be shown with attached notes.

Domain Class

- Each domain class denotes a type of object. It is a descriptor for set of things that share common features.
- Classes can be:
  - **Business objects:** represent things that are manipulated in the business. E.g. order
  - **Real world objects:** things that business keeps track . E.g. Contract, site
  - **Events that transpire:** E.g. Sales and payment

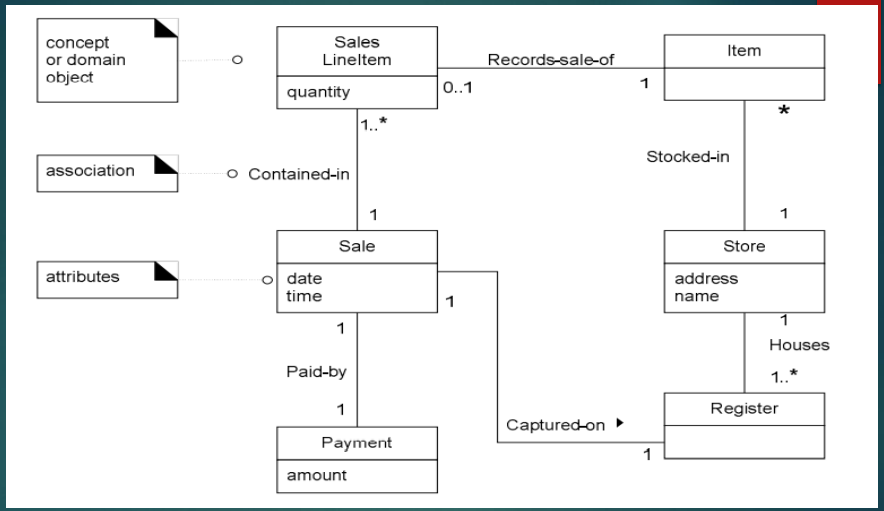


Fig: Simple Domain class model

How to identify domain classes

- **Reuse an existing domain model**

  - There are many published, well crafted domain models.
- **Use a conceptual class category list**

  - Make a list of all candidate conceptual classes
- **Identify noun phrases**

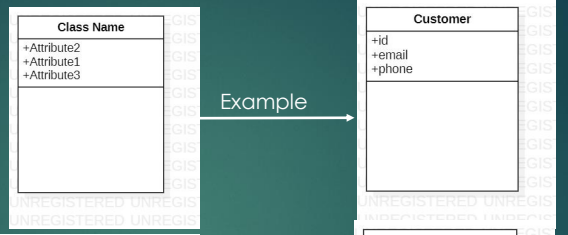
  - Identify nouns and phrases in textual descriptions of a domain (use cases or other documents)

Conceptual Class Category List

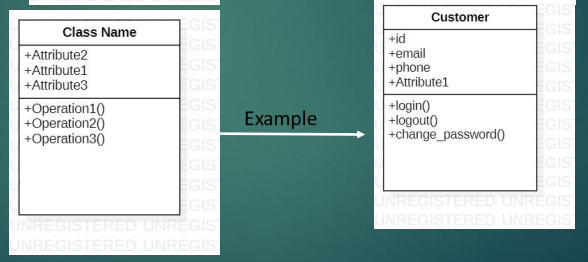
- **Physical objects**: Register, Plane
- **Specifications or description of things**: Productspecification, Flightdescription
- **Places**: Store, Airport
- **Transactions**: sale, payment, reservation
- **Roles of people**: Cashier, pilot
- **Container of other things**: store, Hanger, Airplane
- **Things in a container**: Item, Passenger
- **Computer systems**: Airtrafficcontrol, Creditpaymentauthorizationsystem
- **Catalogs**: Productscatalog, Partscatalog
- **Organizations**: SalesDepartment, Airline

Conceptual Class vs Complete Class

- Conceptual Class:



- Complete Class:

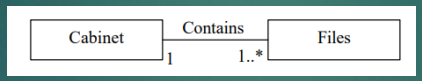


Building a Conceptual Model

- **Find Conceptual Classes/ Domain Entities:**
  - idea, thing, object
  - Is this an actor?
  - Is this an process?
  - Is this entity required?
  - Use Noun Phrase Identification to Single Out Conceptual Classes
  - identify the nouns and noun phrases in textual description of a domain, and considering them as candidate conceptual classes or attributes.

Adding Associations to Domain Model

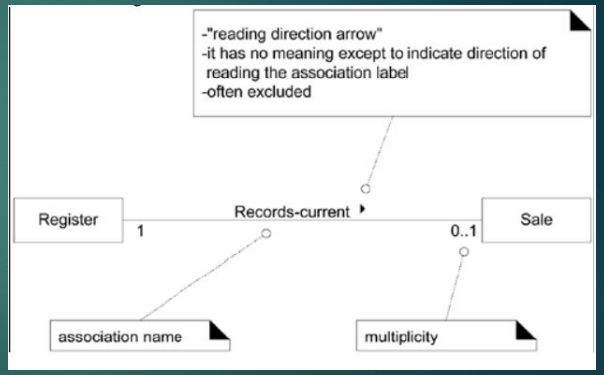
- An association is a semantic relationship between things, concepts or ideas. It represents meaningful connection.
- It is a relationship between types (or more specifically instances of those types) that indicates some interesting and meaningful connection.



- Terminology in Association:
  - Role
  - Association Name
  - Multiplicity

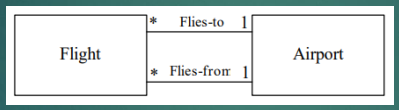
Adding Associations to Domain Model

- The ends of an association may contain a multiplicity expression indicating the numerical relationship between instances of the classes.
- The association is inherently bidirectional, meaning that from instances of either class, logical traversal to the other is possible.
- An optional "reading direction arrow" indicates the direction to read the association name; it does not indicate direction of visibility or navigation. If the arrow is not present, the convention is to read the association from left to right or top to bottom, although the UML does not make this a rule.



Adding Associations to Domain Model

- **Multiple Associations:**
  - More than an association can exist between two or more classes.

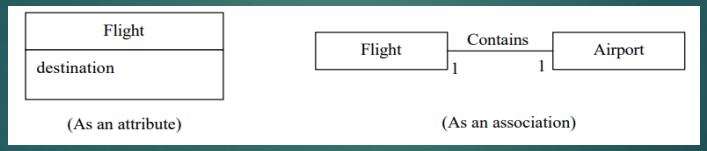


- **Implementation of Associations:**
  - It is more important to identify conceptual classes that associations at the Elaboration phase. ·
  - Some associations of domain model may not be required to be implemented. ·
  - Some associations might be discovered at the time of implementation only and the domain model should have to be changed to reflect these discoveries.

Adding Attributes to Domain Model

Guidelines in adding attributes in domain model

1. **Keep attributes simple**
   - Make attributes of primitive data type as far as possible.
   - Complex attributes can often be identified as a new role (conceptual class) and hence are related with the original one with an association.



Adding Attributes to Domain Model

2. **Attributes should be data types**
   - We decide a thing (noun phrase) whether it should be an association or an attribute.
   - If it can be expressed in terms of data types (integer, text etc.), then we will take it as an attributes. The primitive data types are the simpler ones but there might be complex concepts that could be of non-primitive type.
3. **Make model free from design decisions**
   - Some of the associations between objects expressed in the domain model will be implemented as attributes.
   - The design decision will have to be deferred in the conceptual modeling. Design solutions may be different depending upon the tools.

Things not in a Domain Model

- **Software artifacts**

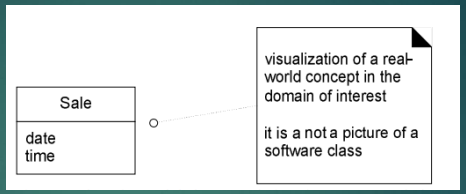


Fig : A domain model shows real-situation conceptual classes, not software classes.

- **Responsibilities or methods**

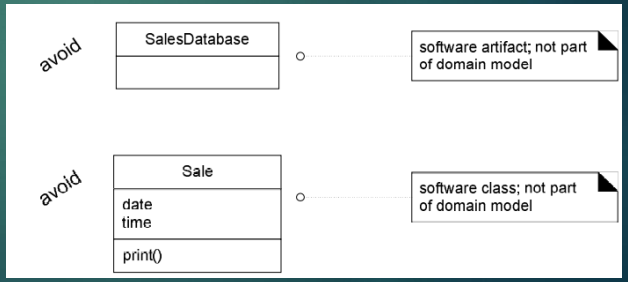


Fig : A domain model does not show software artifacts or classes

Car Rental System

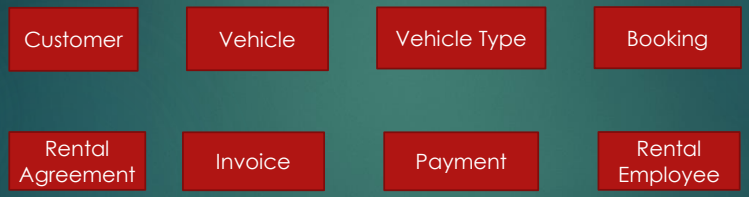
The problem statement of this project is to develop an *online process of vehicle rental service in Pune*. The current system is manual and it is time consuming as well as ineffective in terms of returns. Currently, customers have to call manually in order to rent or reserve a vehicle. The staff of the vehicle rental service company will check their file to see which vehicle is available for rental. *The goal of this project is to automate vehicle rental and reservation so that customers do not need to call and spend unnecessary time in order to reserve a vehicle.*

They can go online and reserve any kind of vehicle they want and that is available. Even when a customer chooses to visit the booking centre to personally hire a vehicle, computers are available for him to go online and perform his reservation. When he choose to reserve by phone, any of the customer service representatives can help him reserve the vehicle speedily and issue him a reservation number.

Car Rental System

- **Conceptual Classes List:**

Customer, Vehicle, Vehicle Type, Booking, Rental Agreement, Invoice, Payment

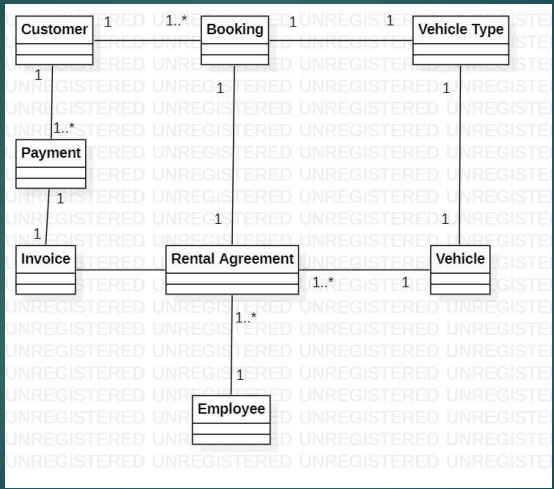


Building Conceptual Model

Add Associations :

- After listing out the conceptual classes, visualize the connection of listed classes with each other through association relationships.
- Add multiplicity in the relationships
- You can add the connection name if required

Car Rental System

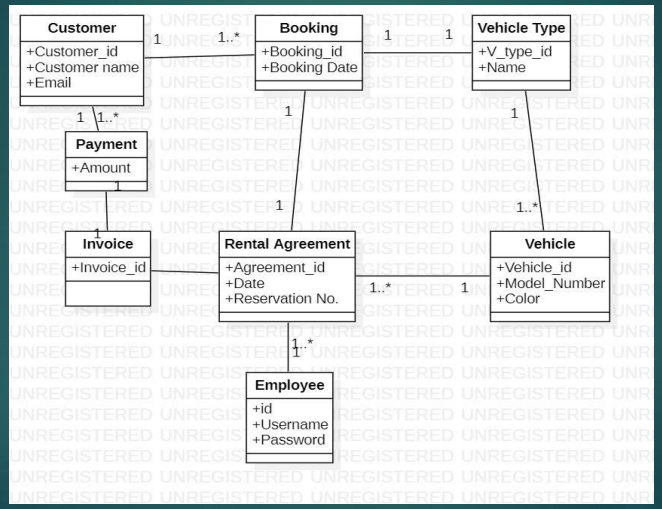


Building Conceptual Model

Add Attributes:

- It is useful to identify those attributes of conceptual classes that are needed to satisfy the information requirements of the current scenarios under development.
- An attribute is a logical data value of an object

Car Rental System



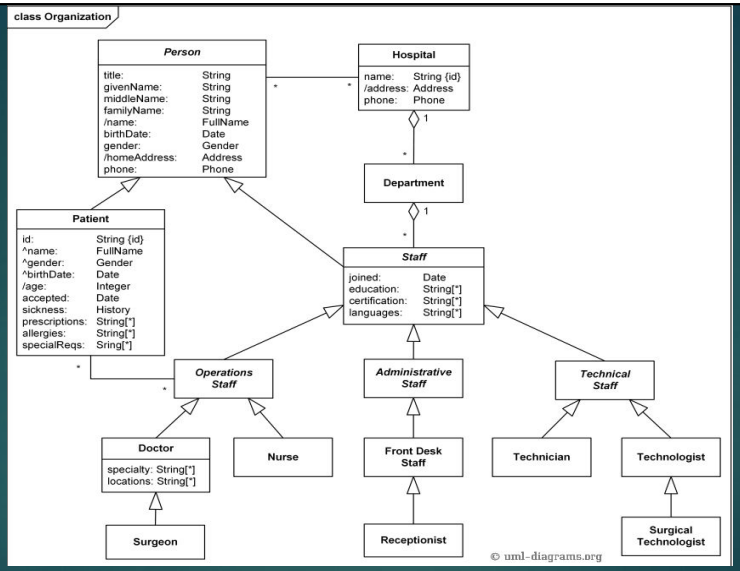


Fig: Hospital Management System

Conceptual Model

- Draw a Conceptual Model of:
  - Online Airline Reservation System
  - Online Library Management System

Class Diagram

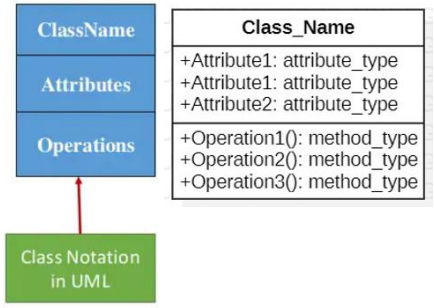
- UML CLASS DIAGRAM gives an overview of a software system by displaying classes, attributes, operations, and their relationships.
- Class Diagram defines the types of objects in the system and the different types of relationships that exist among them.
- This Diagram includes the class name, attributes, and operation in separate designated compartments.
- It helps for better understanding of general schematics of an application.

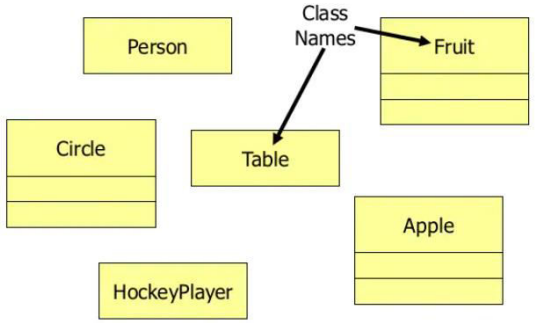
What is a Class?

- A class is the blueprint of an object which can share the same relationships, attributes, operations, & semantics.
- **Structural features** (attributes) define what objects of the class "know"
  - Represent the state of an object of the class
  - Are descriptions of the structural or static features of a class
- **Behavioral features** (operations) define what objects of the class "can do"
  - Define the way in which objects may interact
  - Operations are descriptions of behavioral or dynamic features of a class
- **Following rules must be taken care of while representing a class**:
  - A class name should always start with a capital letter.
  - A class name should always be in the center of the first compartment.
  - A class name should always be written in **bold** format.

Class Notation

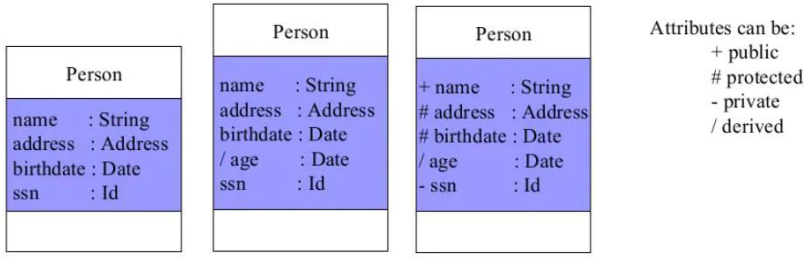
- **Class Name**
  - The name of the class appears in the first partition.





Class Notation

- **Class Attributes**
  - Attributes are shown in the second partition.
  - Attributes are named property of a class that describes the object being modelled.
  - Attributes are usually listed in the form- attributeName: Type
  - A derived attribute is designated by preceding '/' as in: / age: Date



Attribute Modifiers

Attributes can be:
+ public
# protected
- private
/ derived

- In object-oriented design, there is a notation of visibility for attributes and operations. UML identifies four types of visibility: **public**, **protected**, **private**, and **package**.

•**Public attributes**: denoted with the modifier '+', can be accessed from outside the class

•**Private attributes:** denoted with the modifier '-', can only be accessed from within the class

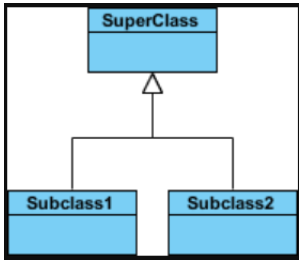
•**Protected attributes:** denoted with the modifier '#', can be accessed from within the class, or any descendant (e.g. a subclass)

Class Notation

- **Class Operations** (Methods)
  - Operations are shown in the third partition. They are services the class provides.
  - Operations map onto class methods in code

Relationships in Class

- **Generalization**:
  - Represents an "is-a" relationship.
  - SubClass1 and SubClass2 are specializations of Super Class.
  - A solid line with a hollow arrowhead that point from the child to the parent class



Simple Association

- A structural link between two peer classes.
- There is an association between Class1 and Class2
- A solid line connecting two classes

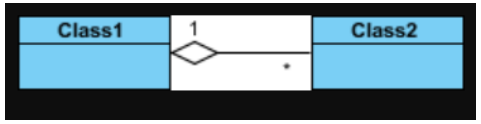
Multiplicity

- How many objects of each class take part in the relationships and multiplicity can be expressed as:
- Exactly one - 1
- Zero or one - 0..1
- Many - 0..* or *
- One or more - 1..*
- Exact Number - e.g. 3..4 or 6
- Or a complex relationship - e.g. 0..1, 3..4, 6.* would mean any number of objects other than 2 or 5

Types of Associations

Aggregation:

- A special type of association. It represents a "part of" relationship.
- Class2 is part of Class1.
- Many instances (denoted by the *) of Class2 can be associated with Class1.
- Objects of Class1 and Class2 have separate lifetimes.
- A solid line with an unfilled diamond at the association end connected to the class of composite



Types of Association

Composition:

- A special type of aggregation where parts are destroyed when the whole is destroyed.
- Objects of Class2 live and die with Class1.
- Class2 cannot stand by itself.
- A solid line with a filled diamond at the association connected to the class of composite

Aggregation vs Composition

- **Aggregation** indicates a relationship where the child can exist separately from their parent class. Example: Automobile (Parent) and Car (Child). So, If you delete the Automobile, the child Car still exist.

- **Composition** display relationship where the child will never exist independent of the parent. Example: House (parent) and Room (child). Rooms will never separate into a House.

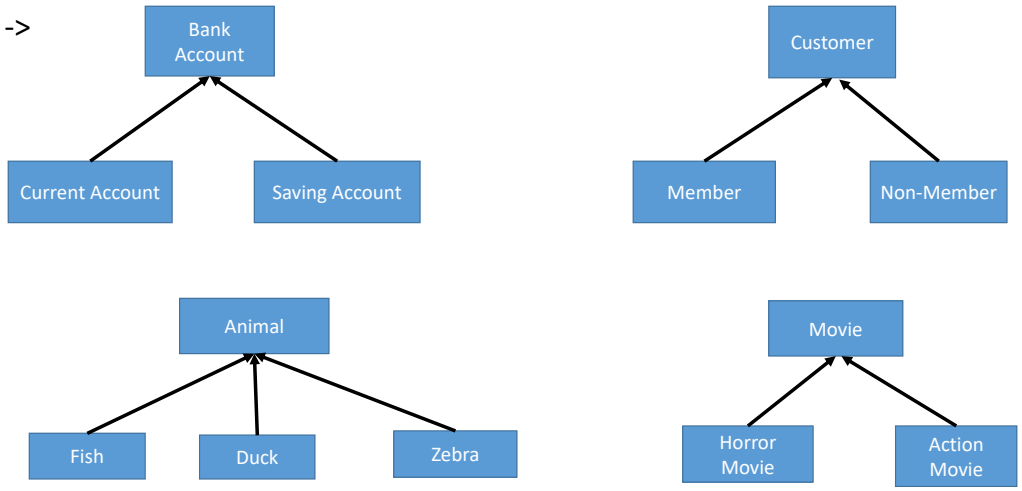
Aggregation vs Composition

- Example: A Library contains students and books.
- Relationship between library and student is aggregation.
  - A student can exist without a library and therefore it is aggregation.
- Relationship between library and book is composition.
  - A book cannot exist without a library and therefore its a composition.

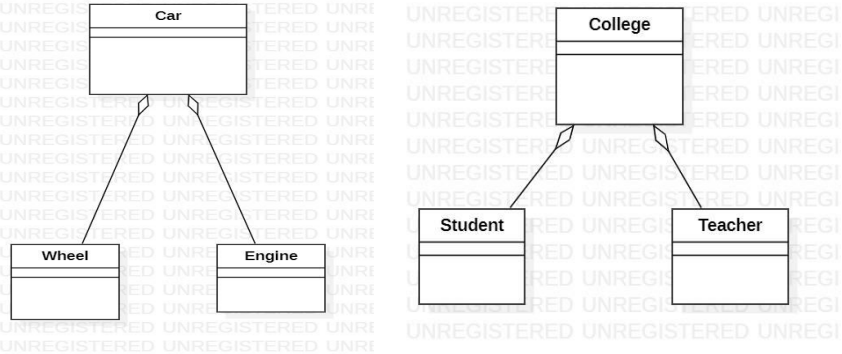
Inheritance vs Aggregation/Composition

- **Inheritance** describes 'is a' / 'is a kind of' relationship between classes (base class - derived class) whereas **aggregation** describes 'has a' relationship between classes.
- **Inheritance** means that the object of the derived class inherits the properties of the base class; aggregation means that the object of the whole has objects of the part. For example, the relation "cash payment is a kind of payment" is modeled using inheritance; "purchase order has a few items" is modeled using aggregation.
- **Inheritance** is used to model a "generic-specific" relationship between classes whereas **aggregation/composition** is used to model a "whole-part" relationship between classes.
- **Inheritance** means that the objects of the subclass can be used anywhere the super class may appear, but not the reverse; i.e. wherever we could use instances of 'payment' in the system, we could substitute it with instances of 'cash payment', but the reverse can not be done.

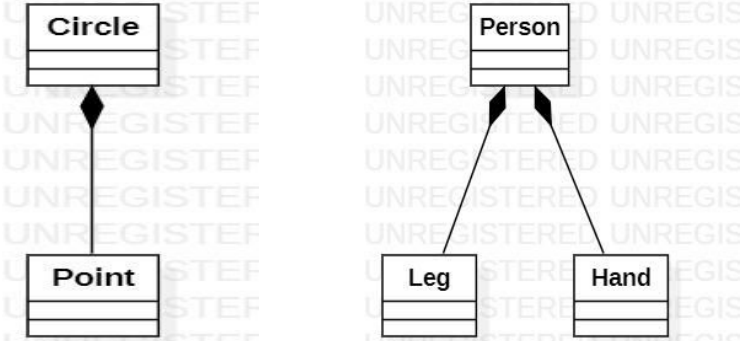
Examples of Inheritance



Examples of Aggregation

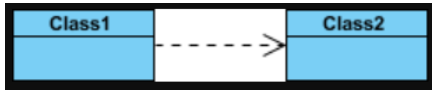


Examples of Composition

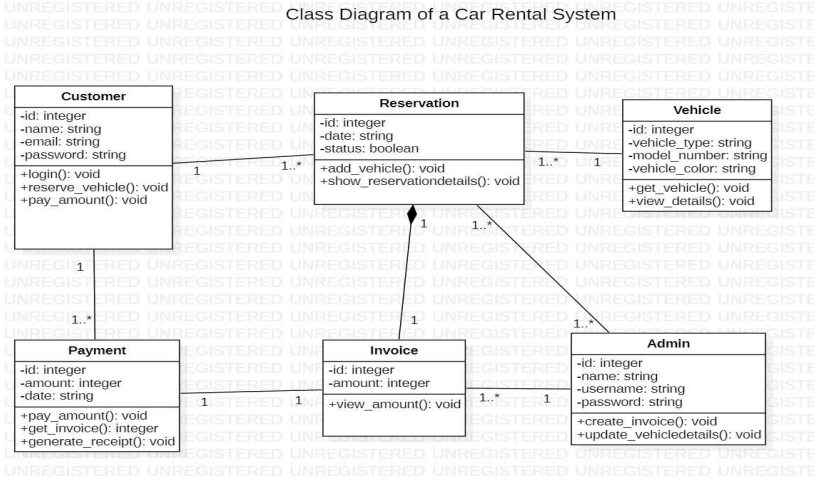


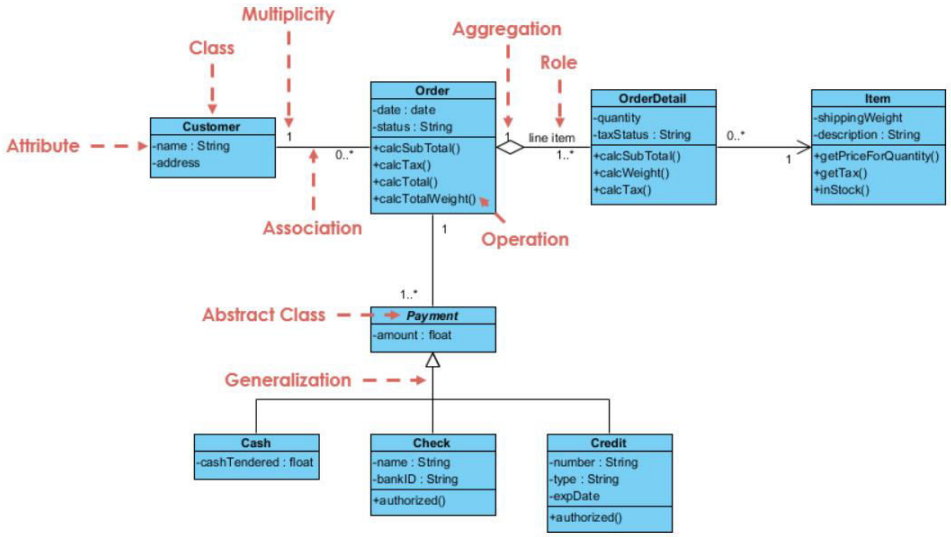
Dependency

- Exists between two classes if the changes to the definition of one may cause changes to the other (but not the other way around).
- Class1 depends on Class2
- A dashed line with an open arrow

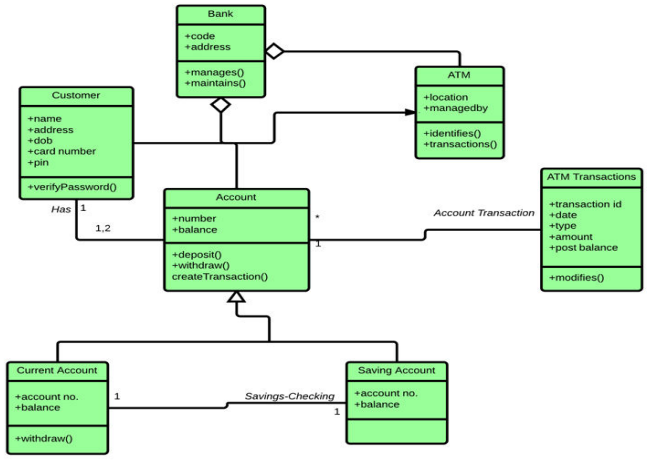


Example Of Class Diagram





Example of Simple Banking System



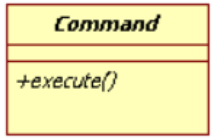
Draw Class Diagram of:

- Online Library management System
- Online Airline Reservation System

Old Questions

- 2073 Bhadra Question 3.a
- 2074 Bhadra Question 9
- 2075 Baisakh Question 8

Abstract Classes



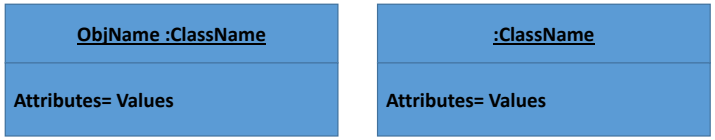
- In the abstract class, no objects can be a direct entity of the abstract class. The abstract class can neither be declared nor be instantiated.
- It is used to find the functionalities across the classes.
- The name of the abstract class is written in italics. Since it does not involve any implementation for a given function, it is best to use the abstract class with multiple objects.
- It is also possible to have an abstract class with no operations declared inside of it.

Object Diagram

- Object diagrams are derived from class diagrams so object diagrams are dependent upon class diagrams.
- Object diagrams represent an instance of a class diagram. The basic concepts are similar for class diagrams and object diagrams.
- Object diagrams also represent the static view of a system but this static view is a snapshot of the system at a particular moment.
- Captures Run time/ Real Time changes of a system
- It includes attributes and data values of any instance.

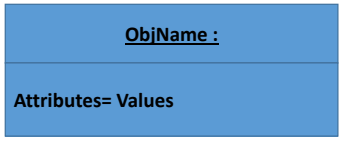
Object Notation

->



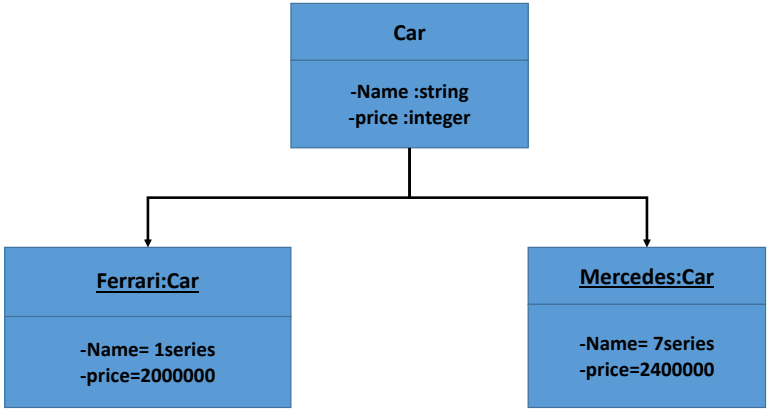
Named Instance

Anonymous Instance

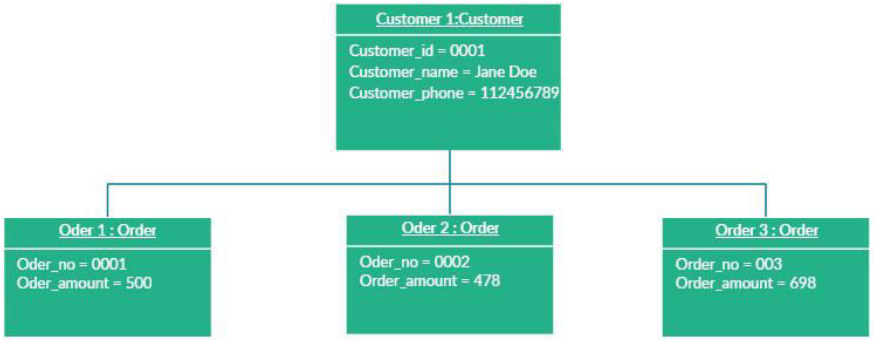


Orphan Instance

Objects Example



Object Diagram Example



Use Cases

- A UML use case diagram is the primary form of system/software requirements for a new software program underdeveloped.
- Use cases specify the expected behavior (what), and not the exact method of making it happen (how).
- A key concept of use case modeling is that it helps us design a system from the end user's perspective. It is an effective technique for communicating system behavior in the user's terms by specifying all externally visible system behavior.
- It only summarizes some of the relationships between use cases, actors, and systems.
- It does not show the order in which steps are performed to achieve the goals of each use case.

Purpose of Use Case Diagram

Use case diagrams are typically developed in the early stage of development and people often apply use case modeling for the following purposes:

- Specify the context of a system
- Capture the requirements of a system
- Validate a systems architecture
- Drive implementation and generate test cases
- Developed by analysts together with domain experts

Essential Elements of Use Case Diagram

- Actors:
  - Someone interacts with use case (system function).
  - Named by noun.
  - Actor plays a role in the business
  - Similar to the concept of user, but a user can play different roles
  - Actor triggers use case(s).
  - Actor has a responsibility toward the system (inputs), and Actor has expectations from the system (outputs).
  - Denoted by this Notation:

  - Example :student, teacher, bank, admin, customer etc.

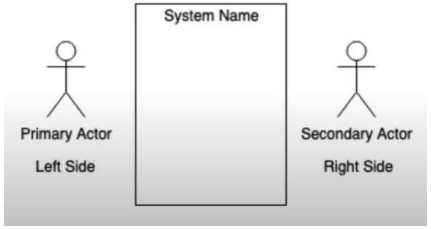
Types of actor

1.Primary Actor :People who use the main system functions are refereed as primary or principle actors.

Example: in ATM system primary actor is customer.

2. Secondary Actor : People who perform administrative or maintenance task are referred as secondary actors.

Example: in ATM system a person in charge of loading money into the system is a secondary actor.

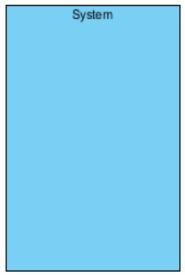


Essential Elements of Use Case Diagram

- Use Case
  - System function (process - automated or manual)
  - i.e. Do something
  - All the functionalities that system does is use case.
  - Each Actor must be linked to a use case, while some use cases may not be linked to actors.
  - Denoted by following Notation:

Essential Elements of Use Case Diagram

- **System Boundary:**
  - The system boundary is potentially the entire system as defined in the requirements document.
  - can form a system boundary for use cases specific to each of these business functions.
  - Symbol:



Relationship in Use Case Diagram

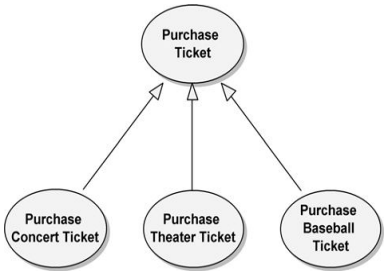
- Association
- Include
- Extends
- Generalization

Association

- communication between an actor and a use case is represented by a solid line.

Generalization

- Generalization relationships, can be used to relate use cases.
- Use cases can have common behaviors that other use cases (i.e., child use cases) can modify by adding steps or refining others,
- Generalization:
  - relationship between one general use case and a special use case.
  - Represented by a line with a triangular arrow head toward the parent use case.

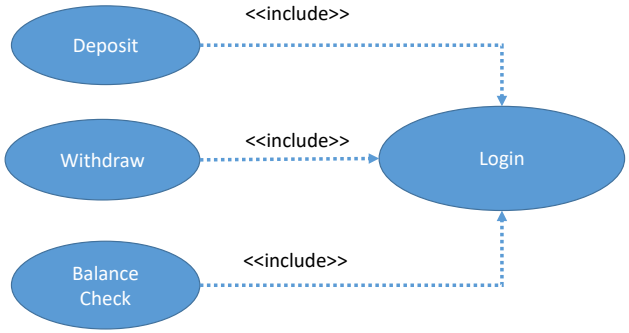


«include» Relationships

- A use cases may contain the functionality of another use case.
- In general it is assumed that any included use case will be called every time the basic path is run.
- is used to show how the system can use a pre existing components
- Is used to show common functionality between use cases
- Is used to document the fact that the project has developed a new reusable component
- Dotted line labelled <<include>> beginning at base use case and ending with an arrows pointing to the include use case.

«include» Relationships

- Example for an typical ATM transaction:



Example of include Relationship

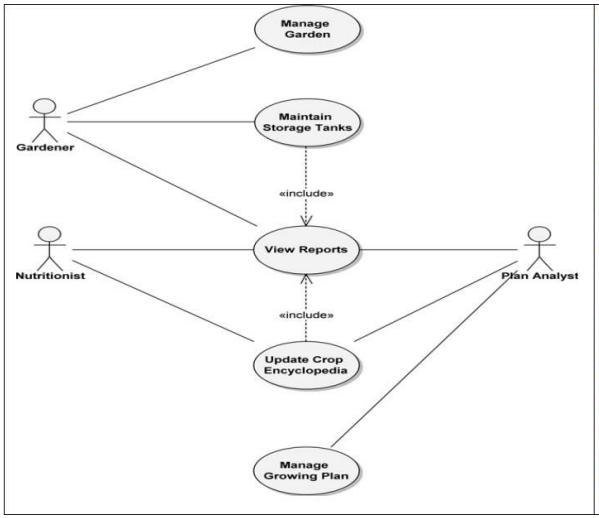


Fig: Example of Include Relationship

This diagram states, for example, that the Update Crop Encyclopedia use case includes the View Reports use case. This means that View Reports must be executed when Update Crop Encyclopedia is executed. Update Crop Encyclopedia would not be considered complete without View Reports.

Where an included use case is executed, it is indicated in the use case specification as an inclusion point. The inclusion point specifies where, in the flow of the including use case, the included use case is to be executed.

<<Extends>> relationship:

- While developing your use cases, you may find that certain activities might be performed as part of the use case but are not mandatory for that use case to run successfully.

Example of Extend Relationship

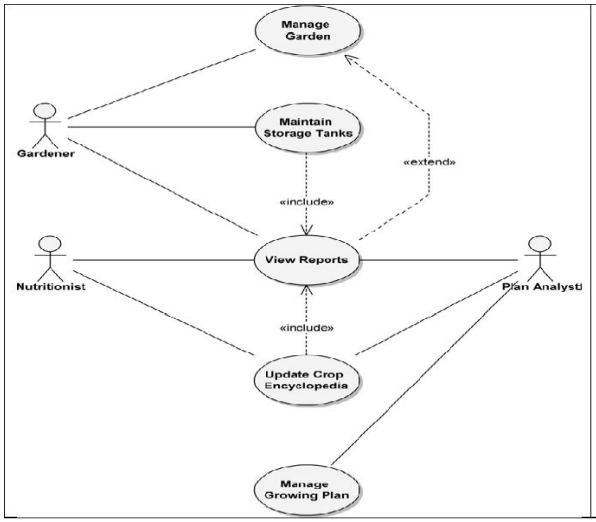


Fig: Example of extends relationship

Extend: *One use case may be used to extend the behaviour of another usecase. This is typically used in exceptional circumstances.*
Dotted line labelled <<extend>> with an arrow toward the base case. The extending use case may add behaviour to the base use case.

Online Shopping System

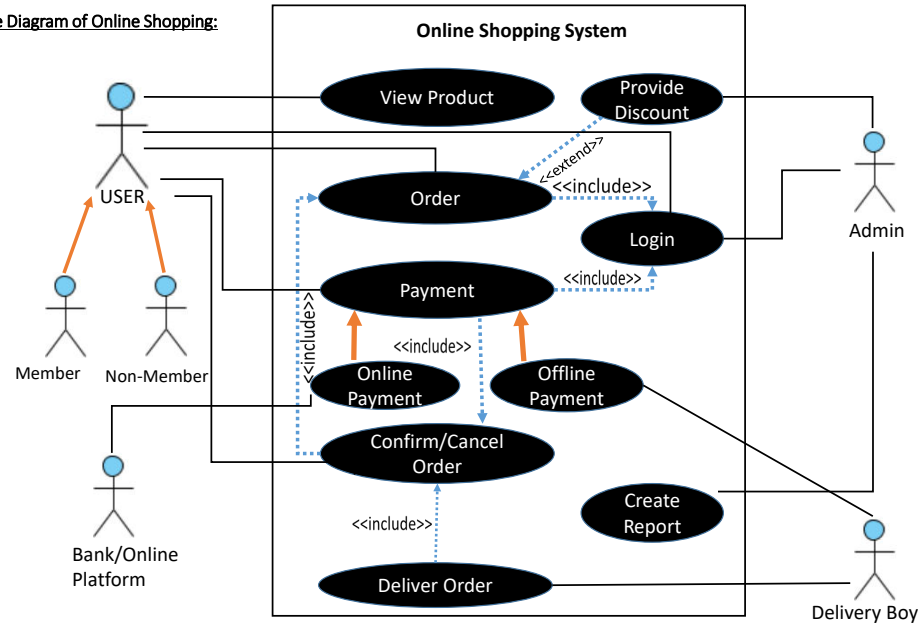
Problem statement:

A customer either a member or a non-member of the system can view the available products in the Online Shopping System. The Customer must create account and login to order the particular product. There may or may not be discount in the product. The customer can cancel the order. For the payment of the product, the customer must confirm the order. Both Online and Offline payment can be done by the customer. The delivery Boy will deliver the particular confirmed order of the customer. The administration personnel will verify the login and create reports based on their statistics.

Online Shopping System

- Actors: Customer, Bank/Online Platform, Admin, Deliver Boy
- Use Case: View Product, Order Product, Confirm/Cancel Order, Payment, Discount, Deliver Order, Login, Report

Use Case Diagram of Online Shopping:



Use Case Diagram of ATM

Practice:

- **Problem Statement: Card Holder enter the card and pin to login in the system. The Card Holder can Withdraw Cash, Check Balance and Change pin only after the logging in the system. He/She will insert the required cash to be withdrawal. He/ She may print the receipt after the cash withdrawal. Administrator will verify the login and update the transactions of the system. Bank Employee will add cash and issue atm card in the machine. Atm Technician will maintain the machine.**
- **Actors: Card Holder, Administrator, Bank Employee, ATM Technician**
- **Use Cases: Login, Cash Withdraw, Check Balance, Change Pin, Print Receipt, Insert Cash, Update Transactions, Maintenance**

Use Case Assignment

Write a Problem statement on these topics. Based on the problem statement list the actors and use cases for each system and draw a use case diagram.

- **Online Airline Reservation System**
- **Online Library Management System**

Activity Diagram

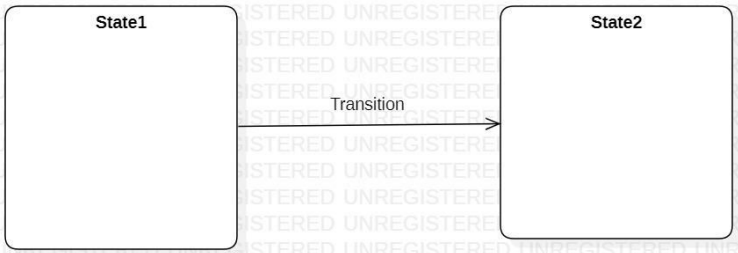
- An activity diagram visually presents a series of actions or flow of control in a system similar to a flowchart or a data flow diagram.
- Activity diagrams are often used in business process modeling.
- Activity Diagrams describe how activities are coordinated to provide a service which can be at different levels of abstraction.
- This UML diagram focuses on the execution and flow of the behavior of a system instead of implementation.
- Activity diagrams consist of activities that are made up of actions that apply to behavioral modeling technology

Symbols/Notations

- **Action Box**: An activity represents execution of an action on objects or by objects. The users or software perform a given task.

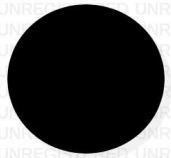
Symbols/Notations

- **Action Flow:** A solid arrow represents the path between different states of an object. Label the transition with the event that triggered it and the action that results from it. A state can have a transition that points back to itself.

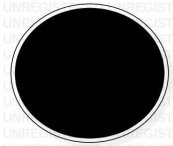


Symbols/Notations

- **Initial State:**
  - ✓ A filled circle followed by an arrow represents the object's initial state.
  - ✓It indicates the initial stage or beginning of the set of actions.



- **Final State:**
  - ✓It is the stage where all the control flows and object flows end.
  - ✓An arrow pointing to a filled circle nested inside another circle represents the object's final state.



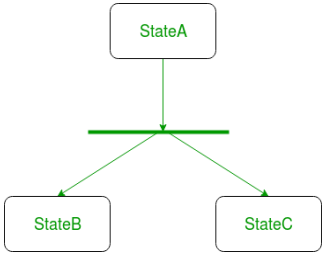
Symbols/Notations

Decision Box:

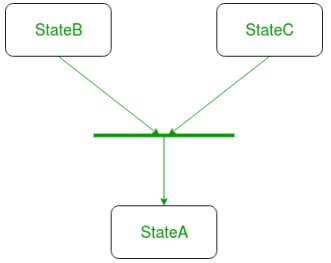
It is a diamond shape box which represents a decision with alternate paths. It represents the flow of control.

Symbols/Notations

- **Synchronization and Splitting of Control**
  A short heavy bar with two transitions entering it represents a synchronization of control. The first bar is often called a fork where a single transition splits into concurrent multiple transitions. The second bar is called a join, where the concurrent transitions reduce back to one.



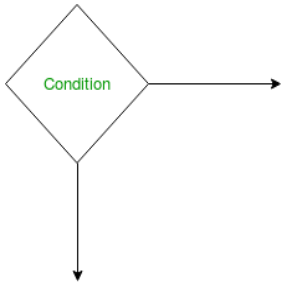
Fork

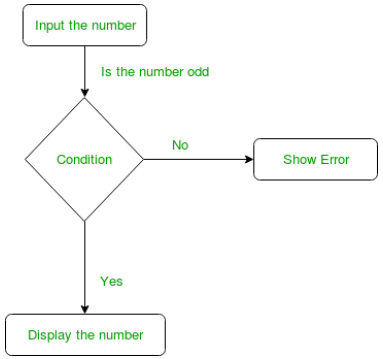


Join

Symbols/Notations

- **Decision node and Branching :** When we need to make a decision before deciding the flow of control, we use the decision node.





Rules to develop Activity Diagram

Following rules must be followed while developing an activity diagram:

- All activities in the system should be named.
- Activity names should be meaningful.
- Constraints must be identified.
- Activity associations must be known.

Example of Activity Diagrams

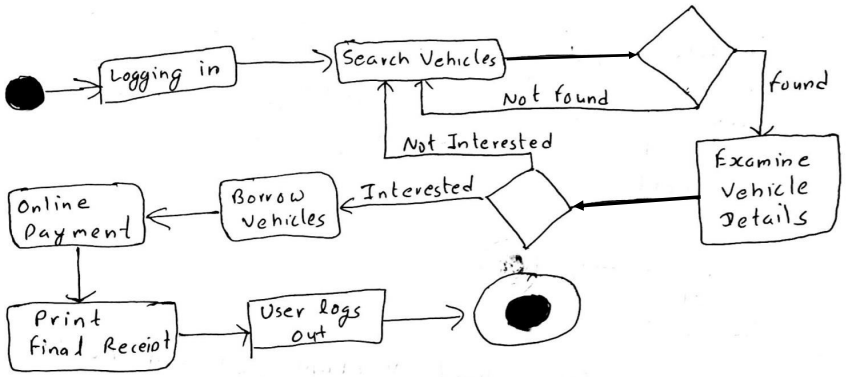


Fig: Activity Diagram Of Car Rental System

Example of Activity Diagrams

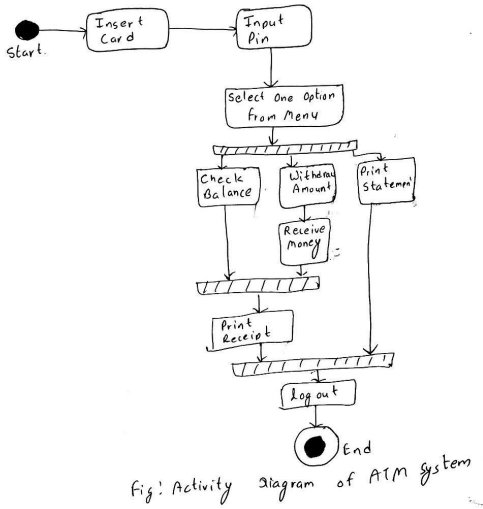
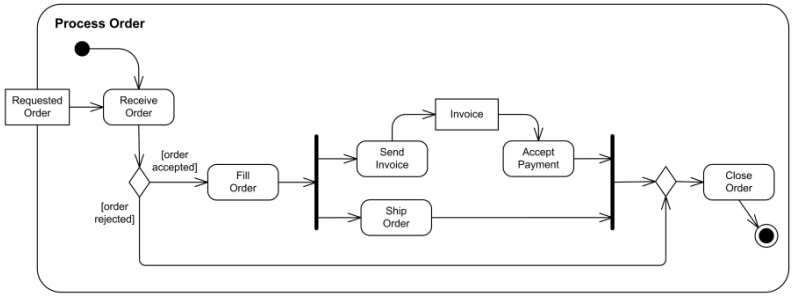


Fig: Activity Diagram of ATM System

Example of Activity Diagrams



State Chart Diagram

State Chart Diagram

- Also called: **State Chart, State Machines, State machine diagram**
- A **state diagram** is used to represent the condition of the system or part of the system at finite instances of time.
- Statechart diagrams are used to describe various states of an entity within the application system.
- Statechart diagram describes the flow of control from one state to another state. States are defined as a condition in which an object exists and it changes when some event is triggered.

State Chart Diagram

- The main purposes of using Statechart diagrams:
  - To model the dynamic aspect of a system.
  - To model the life time of a reactive system.
  - To describe different states of an object during its life time.
  - Define a state machine to model the states of an object.
- Before drawing a Statechart diagram we should clarify the following points:
  - Identify the important objects to be analyzed.
  - Identify the states.
  - Identify the events.

Symbols/Notations

Initial State:

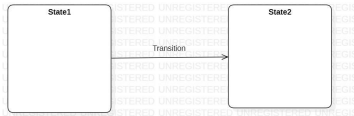
This shows the starting point of state chart diagram that is where the activity starts.

States:

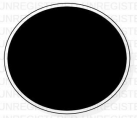
States represent situations during the life of an object in which some action is performed.

Symbols/Notations

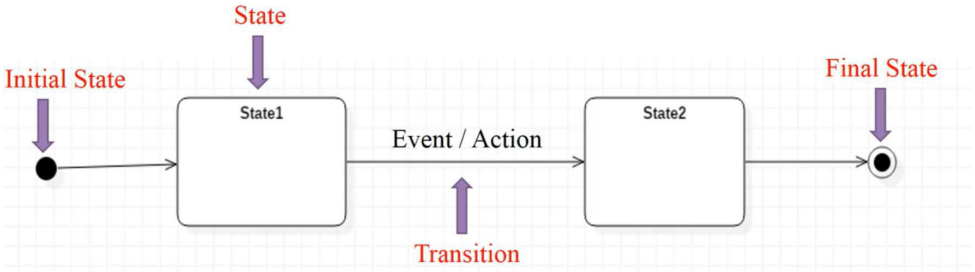
- Transition: It is relationship between two states which indicates that an object in the first state will enter the second state and performs certain specified actions.



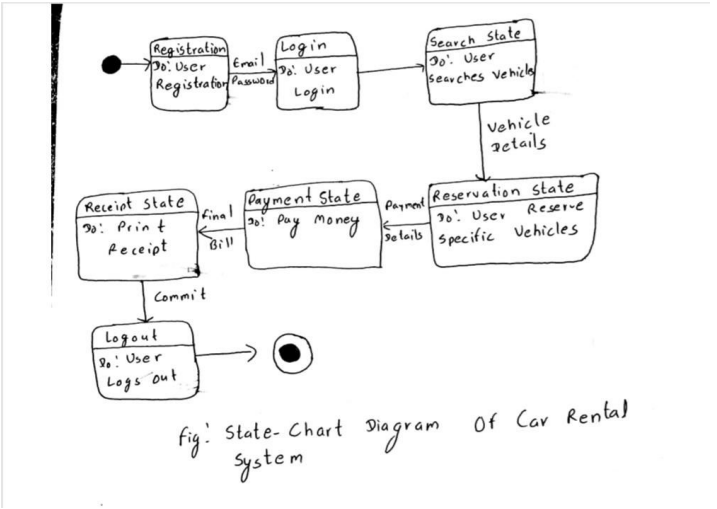
- Final state: The end of the state chart diagram is represented by solid circle surrounded by a circle.



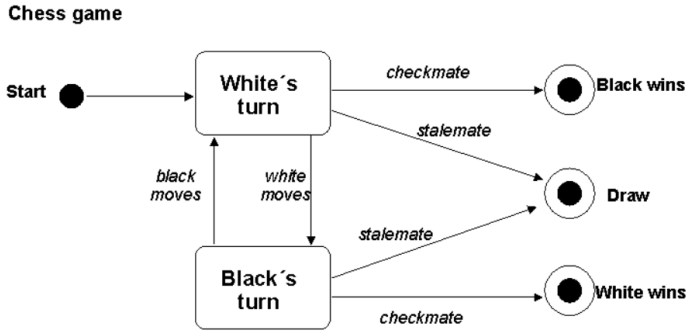
Example of State Chart Diagrams



Example of State Chart Diagrams



Example of State Chart Diagrams



Component Diagram

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

Purpose of Component Diagram

- Visualize the components of a system.
- Describe the organization and relationships of the components.
- It does not describe the functionality of the system but it describes the components used to make those functionalities.
- Component diagrams can also be described as a static implementation view of a system. Static implementation represents the organization of the components at a particular moment.

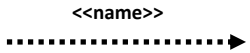
Symbols and Relationships

Component represents a modular part of a system. A component defines its behavior in terms of provided and required interfaces.

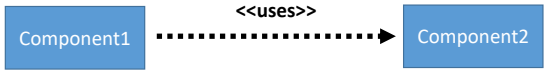
Package container is used to define UML elements such as classes, use cases, and components.

Symbols and Relationships

- Optional Dependency: If one component may or may not or sometimes uses the other component then this kind of symbol is used.

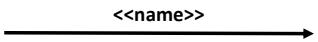


- i.e: **Component1** sometimes uses **Component2** as per need.

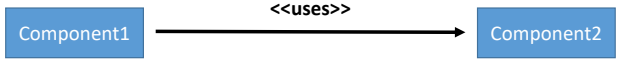


Symbols and Relationships

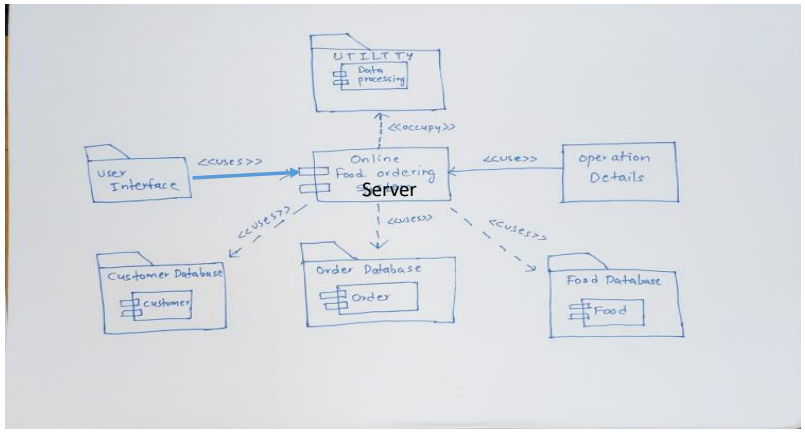
- Full Dependency: If one component definitely use the other Component then this kind of symbol is used.



- i.e **Component1** always uses **Component2.**



Component Diagram of Online Food Ordering



Deployment Diagram

Deployment Diagram

- Deployment diagram visualizes the physical hardware on which the software will be deployed.
- It portrays the static deployment view of a system.
- It involves the nodes and their relationships.
- It indicate how the software is to be installed across systems—for example, what will be installed on the server and what will be installed on the admin PCs.
- Deployment diagrams show the way software is distributed across the available hardware and the basic hardware architecture in terms of processors and communication links.

Purpose of Deployment Diagram

- 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.

Symbols and Relationships

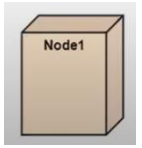
Artifact:

- It represents the specification of a concrete real-world entity related to software development.
- We can use the artifact to describe a framework which is used during the software development process or an executable file. Artifacts are deployed on the nodes.
- common artifacts are: Source files, Executable files, Database tables, Scripts, DLL files, User manuals or documentation, Output files etc.

Symbols and Relationships

Node:

- Node is an essential UML element that describes the execution of code and the communication between various entities of a system.
- It is denoted by a 3D box with the node-name written inside of it.
- Nodes help to convey the hardware which is used to deploy the software.
- It is a computational resource upon which artifacts are deployed for execution. It can execute one or more artifacts.

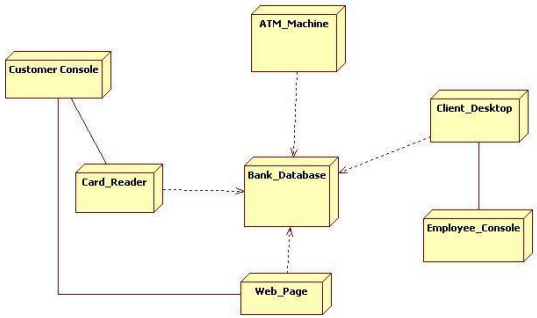


Symbols and Relationships

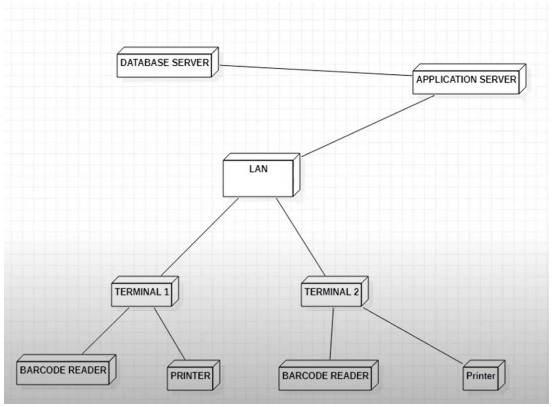
Component:

- It represents other software elements present in the system.
- For instance, a bank application running on an Android device is a component of the node (Android device).

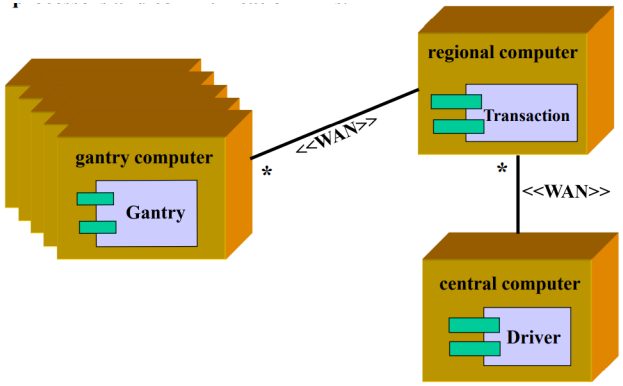
ATM Machine Example



LAN Example



Example



System Sequence Diagram

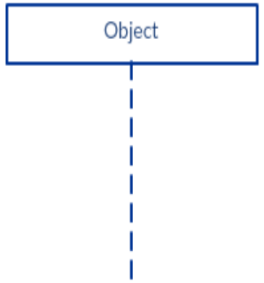
- The most commonly used **interaction** diagram.
- Also known as **Event diagram**.
- A sequence diagram simply depicts interaction between objects in a sequential order i.e. the order in which these interactions take place.
- Sequence diagrams describe how and in what order the objects in a system function.
- They illustrate how the different parts of a system interact with each other to carry out a function, and the order in which the interactions occur when a particular use case is executed.
- In simpler words, a sequence diagram shows different parts of a system work in a ‘sequence’ to get something done.

System Sequence Diagram

- A sequence diagram is structured in such a way that it represents a timeline which begins at the top and descends gradually to mark the sequence of interactions.
- Each object has a column and the messages exchanged between them are represented by arrows.
- Purpose is to model high-level interaction among active objects within a system.

Sequence Diagram Notations

- **Lifeline**:
  - An individual participant in the sequence diagram is represented by a lifeline. It is positioned at the top of the diagram.
  - So basically each instance in a sequence diagram is represented by a lifeline.
  - They represent the different objects or parts that interact with each other in the system during the sequence.
  - No two lifeline notations should overlap each other.

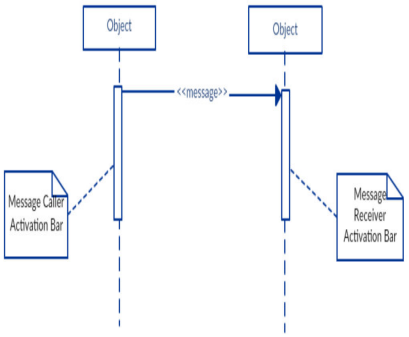


Sequence Diagram Notations

- **Actor:**
  - A role played by an entity that interacts with the subject is called as an actor.
  - It represents the role, which involves human users and external hardware or subjects.
  - An actor may or may not represent a physical entity, but it purely depicts the role of an entity.
  - It is important to note here that an actor is always outside the scope of the system we aim to model using the UML diagram.

Sequence Diagram Notation

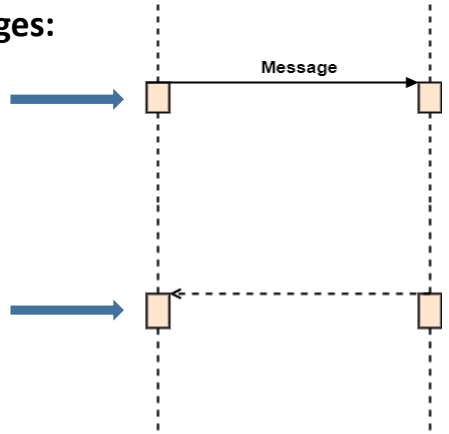
- **Activation Bars:**
  - The use of the activation bar on the lifelines of the Message Caller (the object that sends the message) and the Message Receiver (the object that receives the message) indicates that both are active/is instantiated during the exchange of the message.
  - Activation bar is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



Sequence Diagram Notation

Messages:

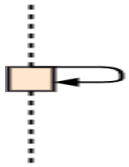
- **Call Message:** It defines a particular communication between the lifelines of an interaction, which represents that the target lifeline has invoked an operation.



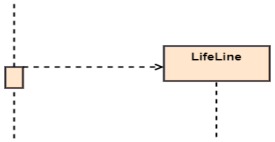
- **Reply Message:** It defines a particular communication between the lifelines of interaction that represent the flow of information from the receiver of the corresponding caller message.

Sequence Diagram Notation

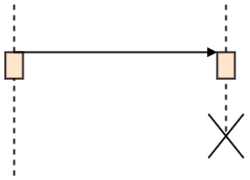
- **Self Message:** It describes a communication, particularly between the lifelines of an interaction that represents a message of the same lifeline, has been invoked.



- **Create Message:** It describes a communication, particularly between the lifelines of an interaction describing that the target (lifeline) has been instantiated.



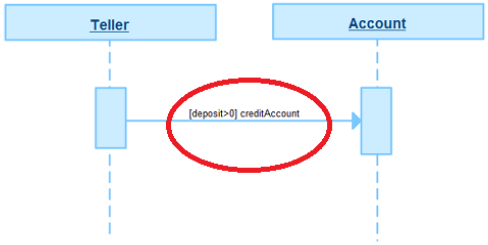
- **Destroy Message:** It describes a communication, particularly between the lifelines of an interaction that depicts a request to destroy the lifecycle of the target.



Sequence Diagram Notation

- **Guards:** To model conditions we use guards in UML. They are used when we need to restrict the flow of messages on the pretext of a condition being met. Guards play an important role in letting software developers know the constraints attached to a system or a particular process.

For example: In order to be able to withdraw cash, having a balance greater than zero is a condition that must be met as shown below.



Sequence Diagram Notation

Extra Notations:

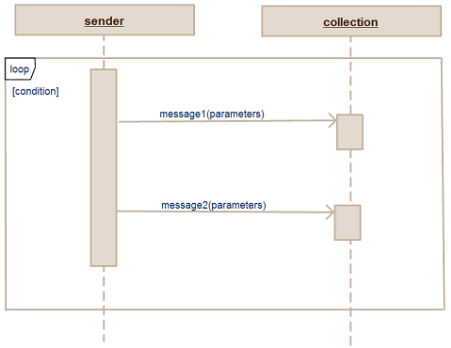
- **Synchronous Message:** a synchronous message is used when the sender waits for the receiver to process the message and return before carrying on with another message. The arrowhead used to indicate this type of message is a solid one, like the one below.

- **Asynchronous Message:** An asynchronous message is used when the message caller does not wait for the receiver to process the message and return before sending other messages to other objects within the system.

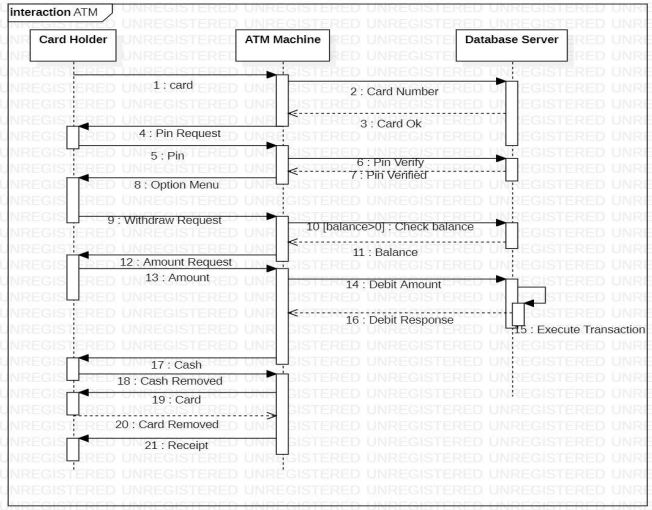
Sequence Diagram Notation

Extra Notations:

- **Loop:** This is something that is used when you need to model a repetitive sequence. In UML, modeling a repeating sequence has been improved with the addition of the loop combination.



Examples of Sequence Diagram



Examples of Sequence Diagram