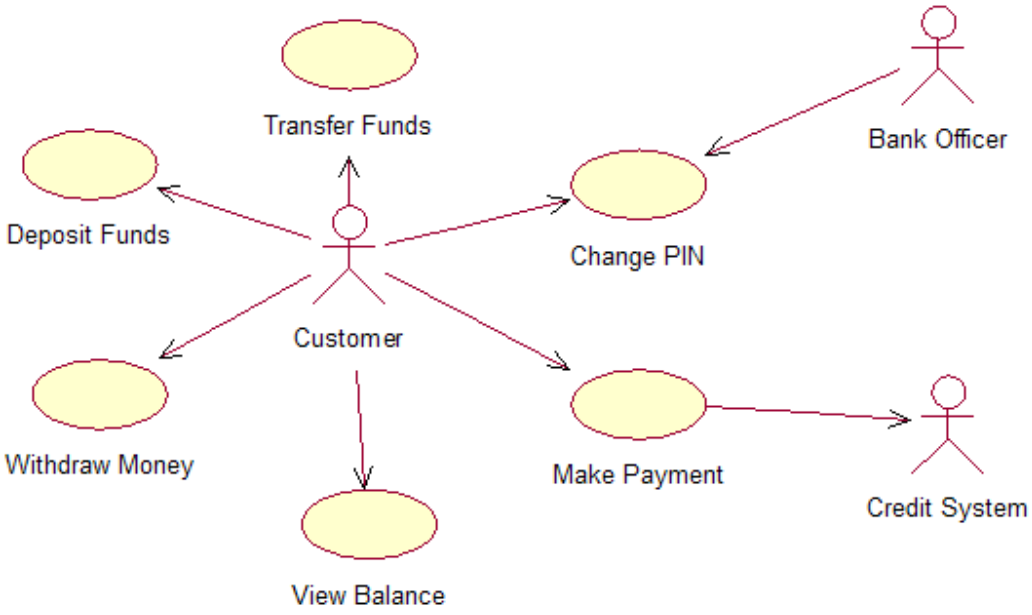
Class notes on UML

The Unified Modeling Language (UML) is a standard language for **specifying**, **visualizing**, **constructing**, and **documenting** the artifacts of software systems, as well as for business modeling and other non-software systems.



Use Case diagram for an ATM (Automated Teller Machine)

- Do not model actor to actor communication
- Do not draw an arrow directly between two use cases
- Every use case must be initiated by an actor
- A database has to be thought as underneath layer of the entire use case diagram

Documenting the flow of events
The flow of events typically includes

- A brief description
- Preconditions
- Primary flow of events
- Alternative flow of events
- Post-conditions

A brief description describes what that use case will do. E.g. Transfer Funds use case will allow a customer or bank employee to move funds from one checking or savings account to another checking or savings account.

The preconditions for a use case list any conditions that have to be met before the use case can start at all. Precondition may be termination of another use case.

The flow of event describes step-by-step what will happen to execute the functionality in the use case. The primary and alternate flow of events includes:

- How the use case starts
- The various paths through the use case
- The normal, or primary, flow through the use case
- Any derivations from the primary flow, known as alternate flow, through the use case
- Any error flows
- How the use case ends

According to the ATM transaction example primary and alternate flows are

Primary flow

1. The use case begins when the bank customer inserts their card into the ATM.
2. The ATM presents a welcome message and prompts the user to enter their personal identification number (PIN).
3. The bank customer enters their PIN.
4. The ATM confirms that the PIN is valid. If the PIN is not valid, alternate flow A1 is performed.
5. The ATM presents the options available:
    - ❖ Deposit funds
    - ❖ Withdraw cash
    - ❖ Transfer funds
6. The use selects the withdraw Cash option.
7. The ATM prompts for the amount to be withdrawn.
8. The user enters the amount to be withdrawn.
9. The ATM determines whether the amount has sufficient funds. If there are insufficient funds, Alternate Flow A2 is performed. If there is an error in attempting to verify funds, Error Flow E1 is performed.
10. The ATM deducts the withdrawal amount from the Customer's account.
11. The ATM provides the customer with the requested cash.
12. The ATM prints a receipt for the customer.
13. The ATM ejects the customer's card.
14. The use case ends.

Alternate Flow A1: Invalid PIN Entered

1. The ATM notifies the customer that the PIN entered was invalid.
2. The ATM ejects the customer's card.
3. The use case ends.

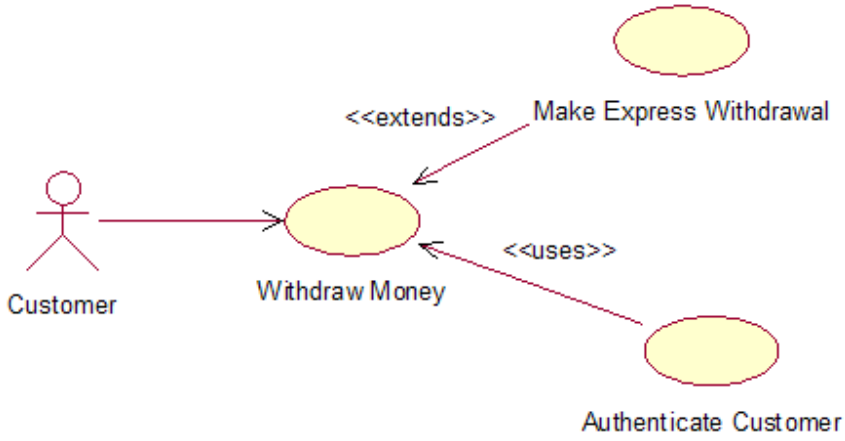
Alternate Flow A2: Insufficient Funds

1. The ATM notifies the customer that the account has insufficient funds.
2. The ATM ejects the customer's card.
3. The use case ends.

Error Flow E1: Error in Verifying Sufficient Funds

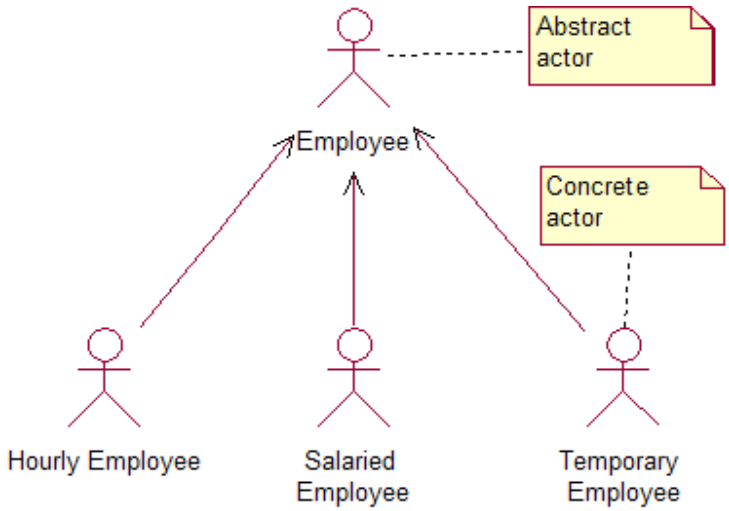
1. The ATM notifies the customer that an error occurred in verifying funds, and provides the customer with the telephone number for the bank's customer service support.
2. The ATM notes the error in the error log. The log entry includes the date and time of the error, the customer's name and account number, and the error code.
3. The ATM ejects the customer's card.
4. The use case ends.

An *abstract use* case is one that is no\t started directly by an actor. Instead, an abstract use case provides some additional functionality that can be used by other use cases. Abstract use cases are the use cases that participate in a uses or extends relationship.



Abstract use cases

An *abstract actor* is an actor that has no instances i.e. the actor's cardinality is exactly zero.
A *concrete actor* is an actor, which will be directly instantiated. The individual and corporate actors are concrete actors.



Abstract Actor

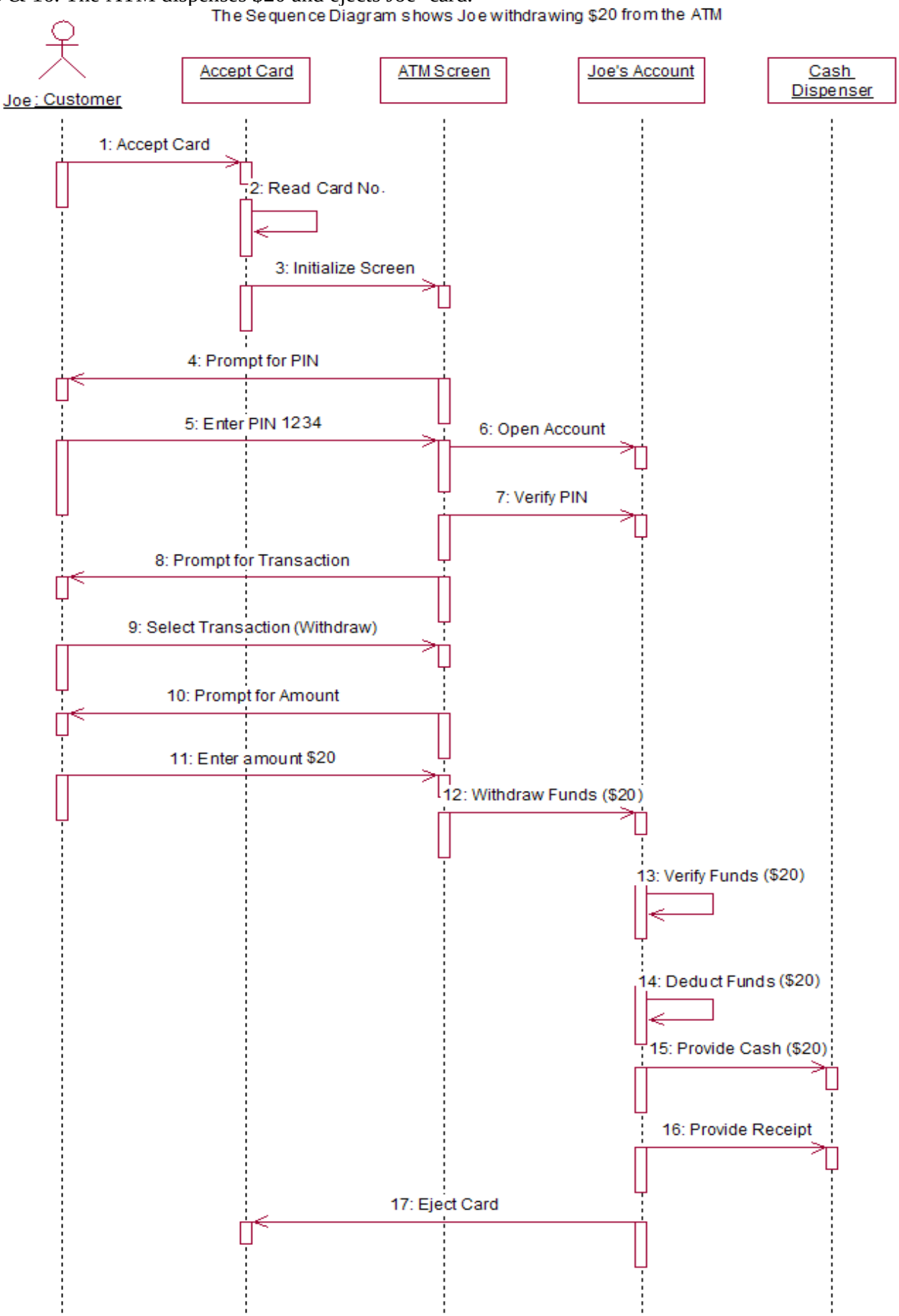
Sequence diagram:

1. The process begins when Joe insert his card into the card reader.
2. The card reader reads the number on Joe's card.
3. Then tells the ATM screen to initialize itself.
4. The ATM prompts Joe for his PIN.
5. Joe enters his PIN (1234).
6. The ATM opens his account.
7. Joe's PIN is validated.
8. ATM prompts him for a transaction.
9. Joe selects withdraw.
10. The ATM prompts Joe for the withdrawal amount.

11 & 12. Joe enters $20.

13. The ATM verifies that the Joe's account has sufficient funds.
14. And subtract $20 from the account.

15 & 16. The ATM dispenses $20 and ejects Joe' card.



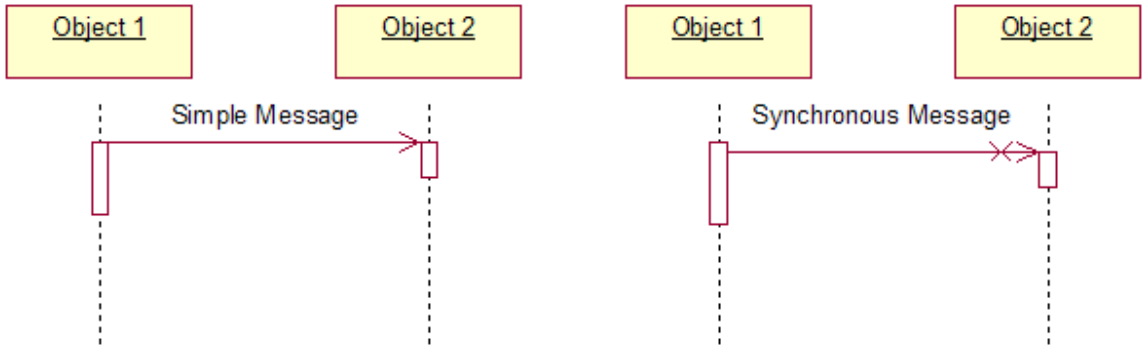
Each object has a life line, drawn as a vertical dashed line below the object. A message is drawn between the lifelines of two objects to show that the objects communicate.

Setting message synchronization option

In the detail tab of the message specification window we can specify the concurrency of the message being sent.

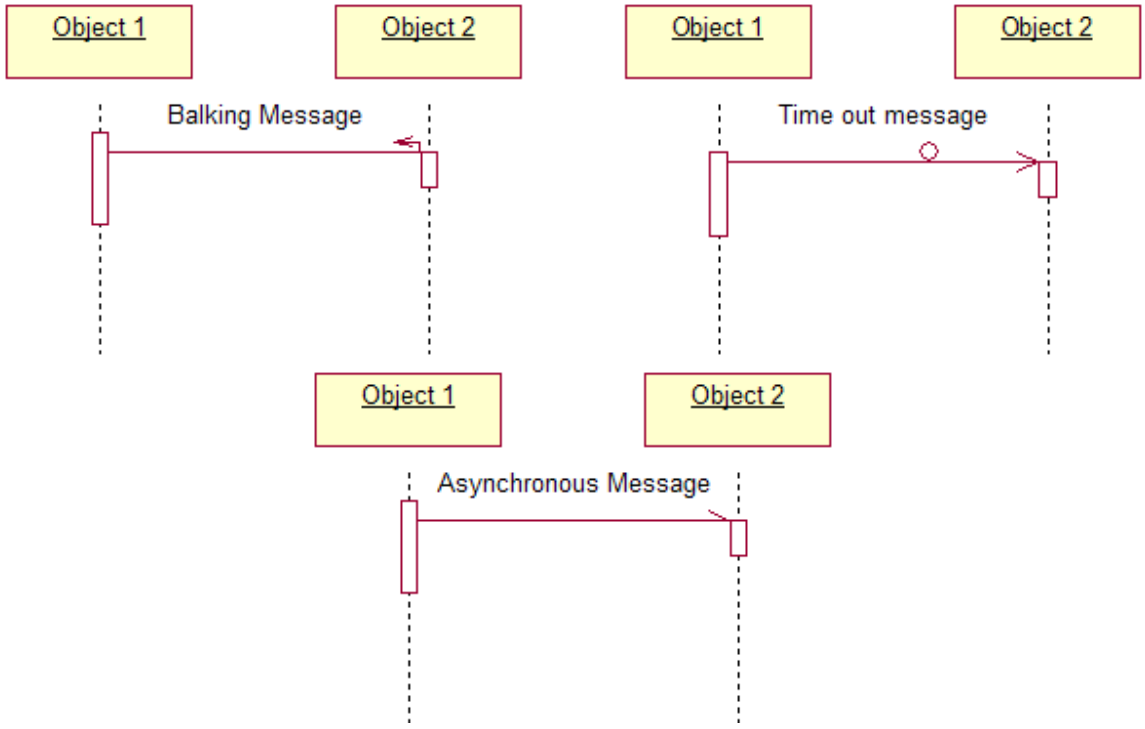
Simple: This is the default value for messages. It specifies that the message run in a single thread of control.

Synchronous: Use this option when the client sends the message and waits until the supplier has acted upon the message.

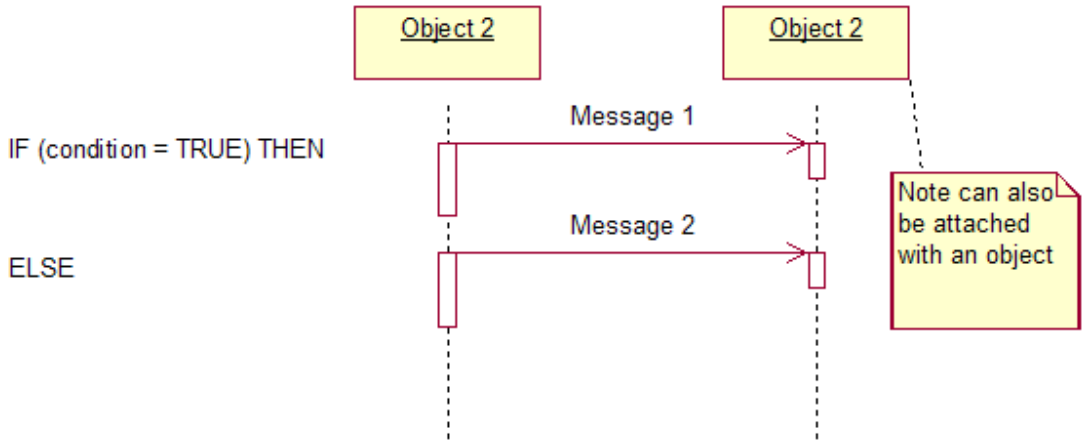


Balking: With this option, the client sends the message to the supplier. If the supplier is not immediately ready to accept the message, the client abandons the message.

Timeout: With this option, the client sends the message to the supplier, and waits a specified amount of time. If the supplier isn't ready to receive the message in that time, the client abandons the message.



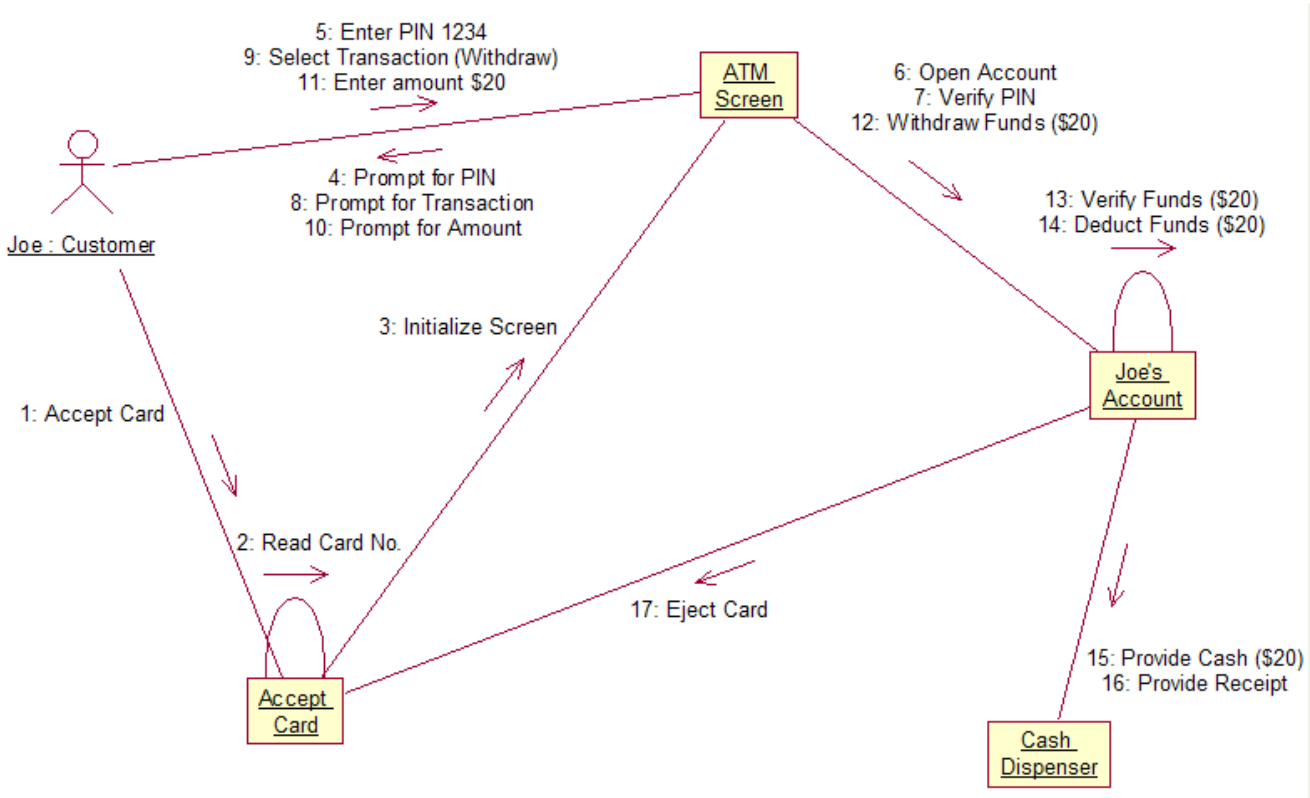
Asynchronous: With this option, the client sends the message to the supplier. The client then continues processing, with out waiting to see if the message was received or not.



In the above sequence diagram condition script has been used

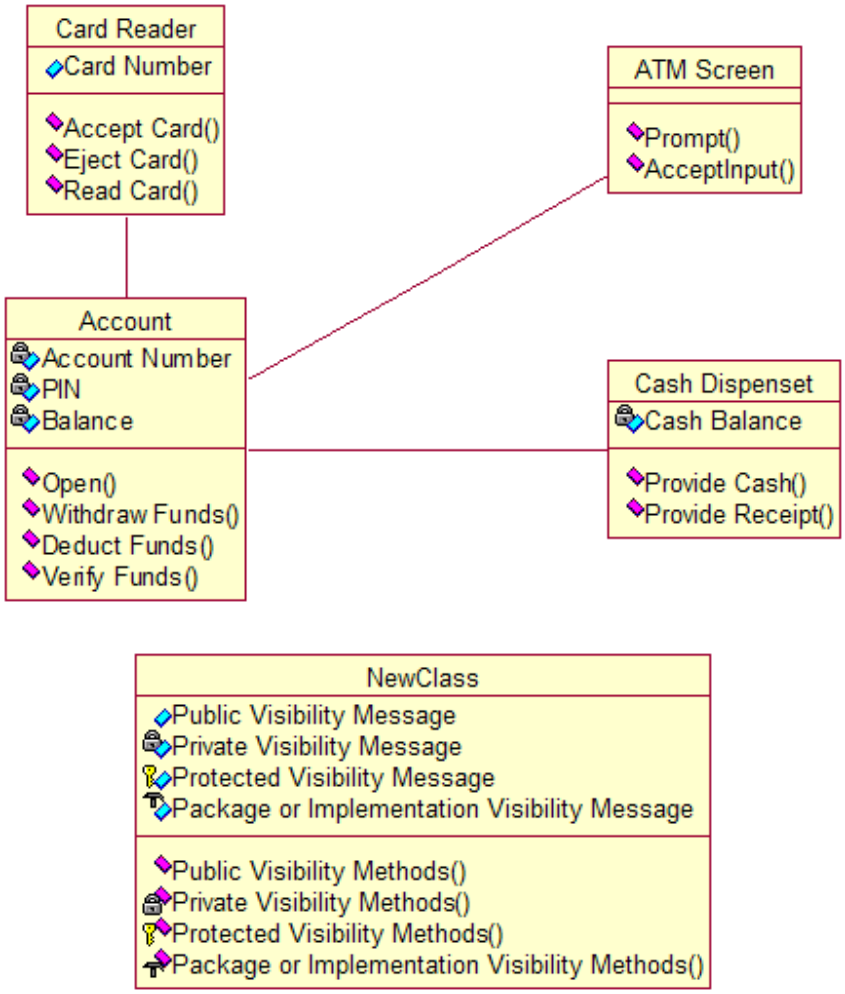
Collaboration Diagrams

Press F5 (or Browse → Go to collaboration diagram) to get the collaboration diagram from sequence diagram. Then to return to sequence diagram again press F5 (or Browse → Go to sequence diagram).



Class Diagram

A class is something that encapsulates information and behavior. The top section of the class holds the class name and, optionally, its stereotype. The middle section holds the attributes or the information that a class holds. The lower section holds the operations or the behavior of a class. We can also show the visibility of each attribute and operation, data type of each attribute, and the signature of each operation on these diagrams.



Relationships and type of Relationships

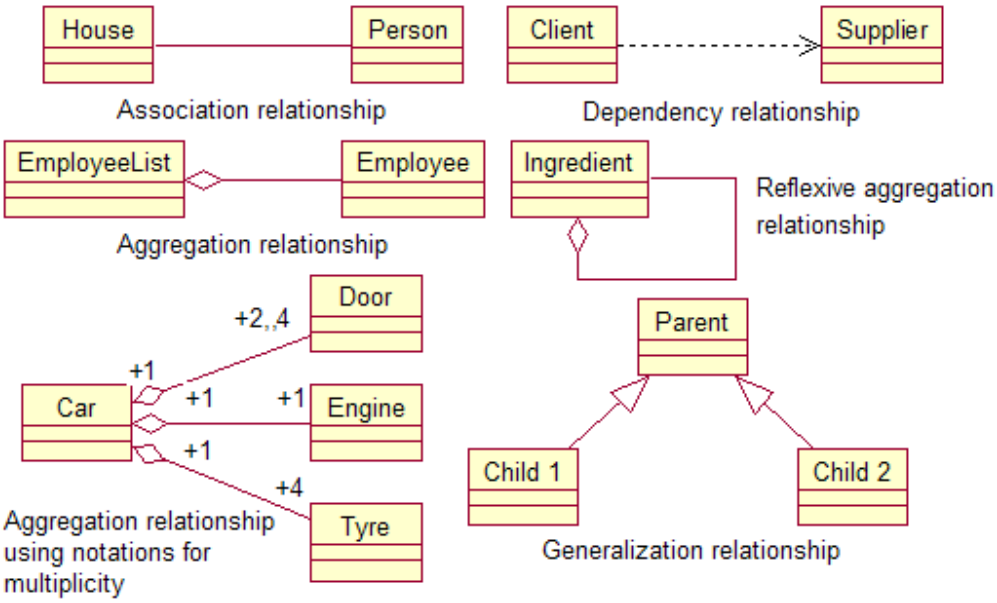
There are four types of relationships, which we set up between classes: associations, dependencies, aggregations and generalizations.

Associations are semantic connections between classes. It may be both bi-directional and unidirectional. Bi-directional associations are drawn either with arrowheads on both ends or with out arrow heads altogether.

Dependencies also connect two classes. Dependencies are also unidirectional and show that one class depends on the definitions of another class and may be depicted by arrow.

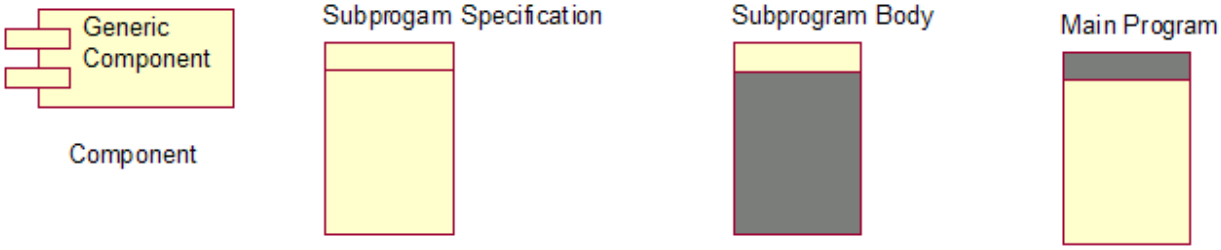
Aggregations are a stronger form of association. An aggregation is a relationship between a whole and parts.

*Generalization*s are used to show an inheritance relationship between two classes.



Component diagram

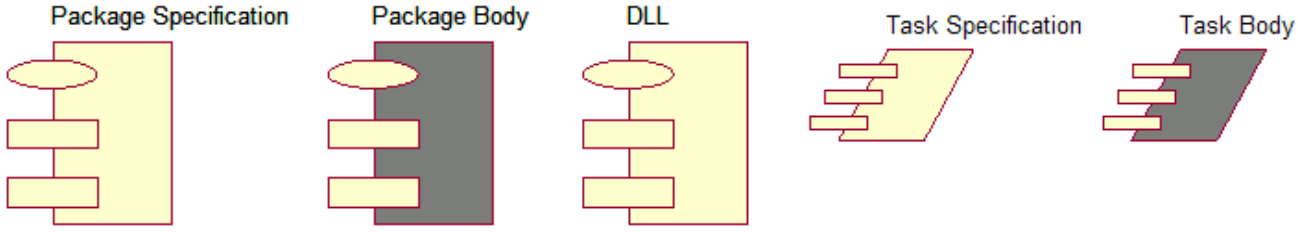
Component diagram shows us a physical view of our model. A component diagram shows us the software components in our system and the relationships between them. A component is a physical module of code. Components can include both source code libraries and run time files. E.g. in C++ header and library files are components. Also the .exe file obtained after compilation is also a component. There are two types of components on the diagram: executable components and code libraries.



Component: The component icon is used to represent a software module with a well-defined interface. The type of component may be ActiveX, Applet, Application, DLL and executables etc.

Subprogram specification and body: These icons are used to represent a subprogram's visible specification and the implementation body. A subprogram is typically a collection of subroutines. Subprograms do not contain class definitions.

Main program: A main program is the file that contains the root of a program.

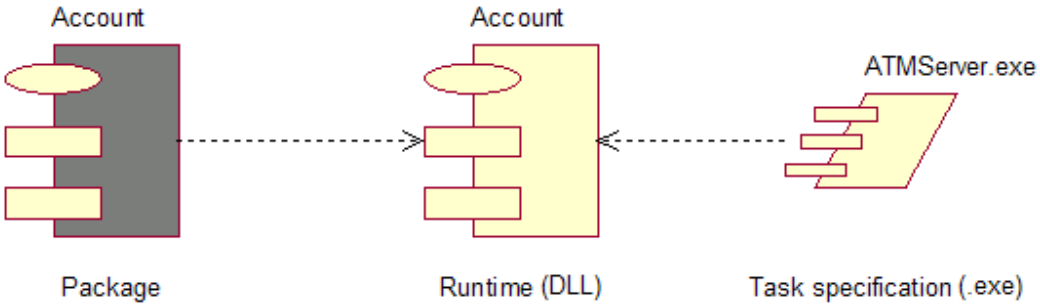


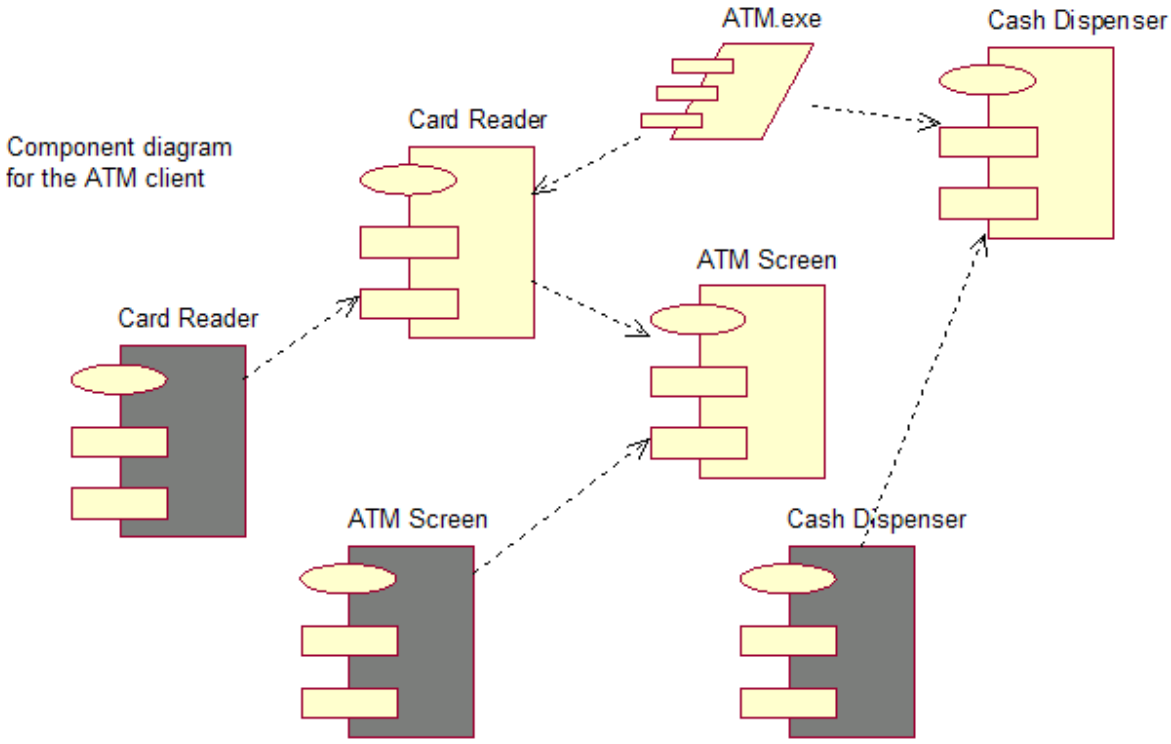
Package specification and body: A package is the implementation of a class. A package specification is a header file, which contains function prototype information for the class.

There are additional Component icons that are used for run time components. Runtime component include executable files, DLL files and tasks.

DLL file: This icon is used to represent data link library (DLL) file.

Task specification and body: These icons are used to represent package that have independent threads of control. An executable file is commonly represented as a task specification with a .EXE extension.





Deployment diagram

Deployment diagram shows all of the nodes of the network, the connections between them, and the process that will run on each one.

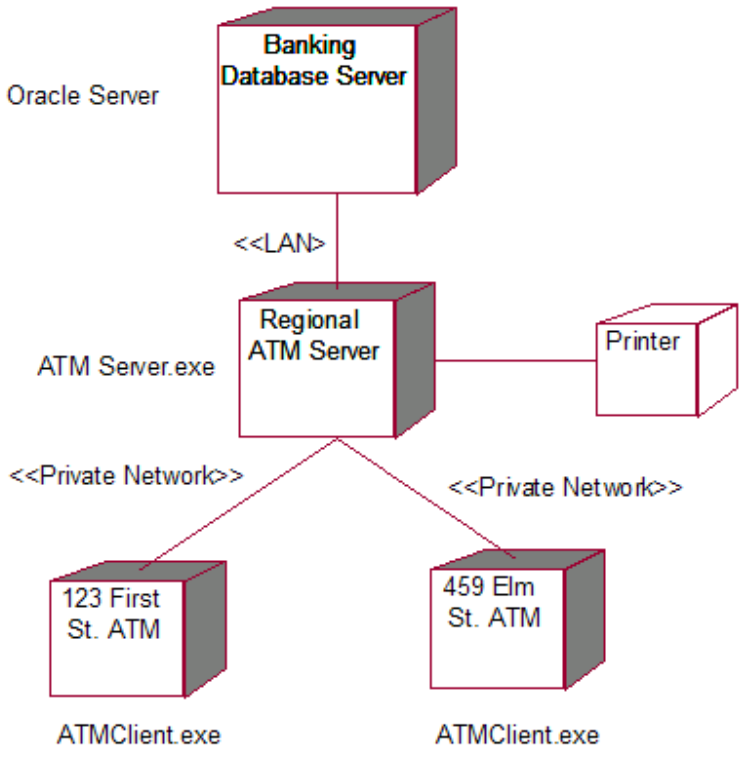
Processor: A processor is any machine that has processing power. The servers, workstations ans other machines with processors are included in this category. The scheduling field documents the type of process scheduling used by the processor. The options are

- **Preemptive:** Indicates that the high priority processes can preempt low priority processes.
- **Non Preemptive:** Indicates that the processes have no priority. The current process executes until it is finished, at which time the next process begins.

- **Cyclic:** Indicates that the control cycles between the processes; each process is given set amount of time to execute, then control passes to the next process.
- **Executive:** Indicates that there is some sort of computational algorithm that controls the scheduling.
- **Manual:** Indicates that the processes are scheduled by the user.

Device: A device is a machine or piece of hardware without processing power. Devices include item such a dumb terminals, printers or scanners.

Connection: A connection represents some type of hardware coupling between two entities. An entity is either a processor or a device. The hardware coupling can be direct, such as an RS232 cable, or indirect, such as satellite-to-ground communication. Connections are usually bi-directional.



Deployment diagram for the ATM system

State transition diagram

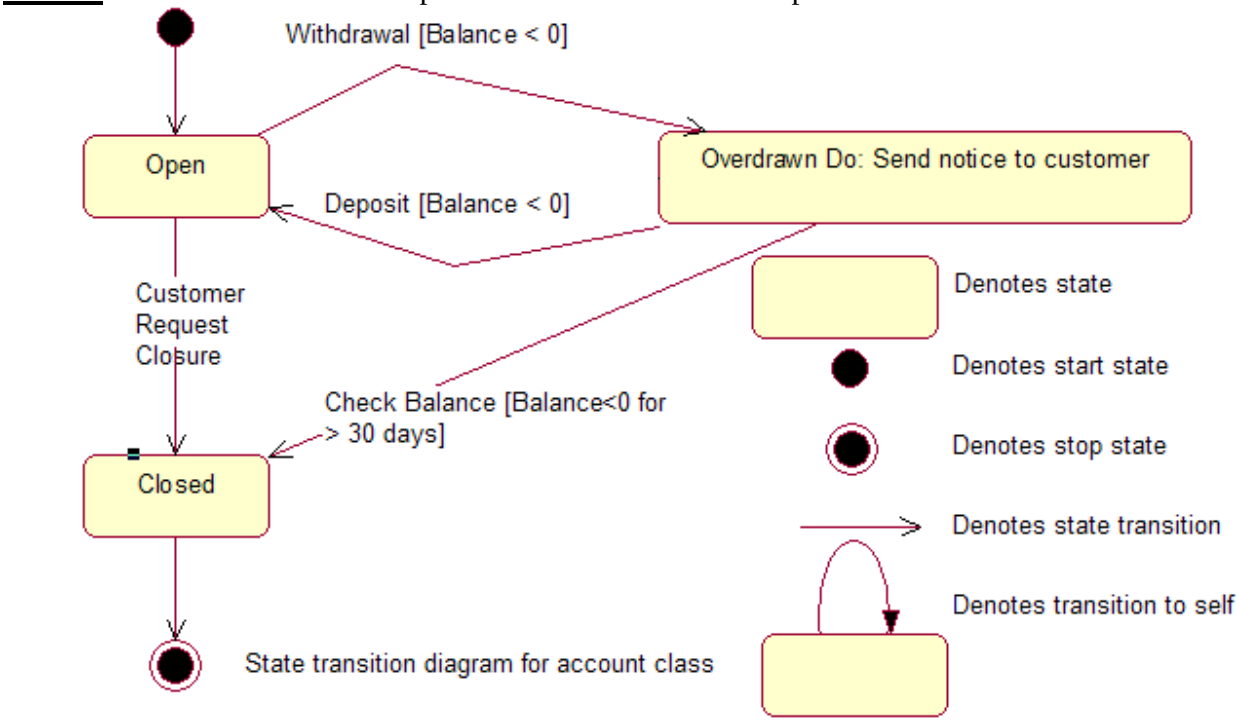
State transition diagram shows the life cycle of a single object, from the time it is created until it is destroyed. These diagrams are a good way to model the dynamic behavior of a class.

State: A State is one of the possible conditions in which an object mat exist.

Activity: An activity is some behavior that an object carries out while it is in a particular state. For example, when an account is in the closed state, the account holder's signature card is pulled. An activity is an interruptible behavior. An activity is shown inside the state itself, preceded by the word do and a colon.

Entry Action: An entry action is a behavior that occurs while the object is transitioning into the state. Unlike an activity, an entry action is considered to be uninterruptible.

Exit Action: An exit action occurs as part of the transition out of a state.
Transition: A transition is a movement from one state to another. The set of transitions on a diagram shows how the object moves from one state to another. Reflexive transition is having arrow starting and ending on the same state.
Event: An event is something that occurs that causes a transition from one state to another.
Action: An action is an uninterruptible behavior that occurs as part of a transition.



Do we really need UML?

- Complex applications need collaboration and planning from multiple teams and hence require a clear and concise way to communicate amongst them.
- Businessmen do not understand code. So, UML becomes essential to communicate with non-programmers essential requirements, functionalities and processes of the system.
- A lot of time is saved down the line when teams are able to visualize processes, user interactions and static structure of the system.

UML is linked with **object-oriented** design and analysis. UML makes the use of elements and forms associations between them to form diagrams. Diagrams in UML can be broadly classified as:

1. **Structural Diagrams –** Capture static aspects or structure of a system. Structural Diagrams include - Component Diagrams, Object Diagrams, Class Diagrams and Deployment Diagrams.
2. **Behavior Diagrams –** Capture dynamic aspects or behavior of the system. Behavior diagrams include - Use Case Diagrams, State Diagrams, Activity Diagrams and Interaction Diagrams.

Object Oriented Concepts Used in UML –

1. **Class –** A class defines the blueprint i.e., structure and functions of an object.
2. **Objects –** Objects help us to decompose large systems and help us to modularize our system. Modularity helps to divide our system into understandable components so that we can build our system piece by piece. An object is the fundamental unit (building block) of a system which is used to depict an entity.
3. **Inheritance –** Inheritance is a mechanism by which child classes inherit the properties of their parent classes.
4. **Abstraction –** Mechanism by which implementation details are hidden from user.
5. **Encapsulation –** Binding data together and protecting it from the outer world is referred to as encapsulation.
6. **Polymorphism –** Mechanism by which functions, or entities can exist in different forms.

Structural UML Diagrams –

1. **Class Diagram –** The most widely use UML diagram is the class diagram. It is the building block of all object oriented software systems. We use class diagrams to depict the static structure of a system by showing system's

classes, their methods, and attributes. Class diagrams also help us identify relationship between different classes or objects.

2. **Composite Structure Diagram –** We use composite structure diagrams to represent the internal structure of a class and its interaction points with other parts of the system. A composite structure diagram represents relationship between parts and their configuration which determine how the classifier (class, a component, or a deployment node) behaves. They represent internal structure of a structured classifier making the use of parts, ports, and connectors. We can also model collaborations using composite structure diagrams. They are like class diagrams except they represent individual parts in detail as compared to the entire class.
3. **Object Diagram –** An Object Diagram can be referred to as a screenshot of the instances in a system and the relationship that exists between them. Since object diagrams depict behavior when objects have been instantiated, we are able to study the behavior of the system at a particular instant. An object diagram is like a class diagram except it shows the instances of classes in the system. We depict actual classifiers and their relationships making the use of class diagrams. On the other hand, an Object Diagram represents specific instances of classes and relationships between them at a point of time.
4. **Component Diagram –** Component diagrams are used to represent the how the physical components in a system have been organized. We use them for modelling implementation details. Component Diagrams depict the structural relationship between software system elements and help us in understanding if functional requirements have been covered by planned development. Component Diagrams become essential to use when we design and build complex systems. Interfaces are used by components of the system to communicate with each other.
5. **Deployment Diagram –** Deployment Diagrams are used to represent system hardware and its software. It tells us what hardware components exist and what software components run on them. We illustrate system architecture as distribution of software artifacts over distributed targets. An artifact is the information that is generated by system software. They are primarily used when a software is being used, distributed, or deployed over multiple machines with different configurations.
6. **Package Diagram –** We use Package Diagrams to depict how packages and their elements have been organized. A package diagram simply shows us the dependencies between different packages and internal composition of packages. Packages help us to organize UML diagrams into meaningful groups and make the diagram easy to understand. They are primarily used to organize class and use case diagrams.

Behavior Diagrams –

1. **State Machine Diagrams –** A state diagram is used to represent the condition of the system or part of the system at finite instances of time. It’s a behavioral diagram and it represents the behavior using finite state

transitions. State diagrams are also referred to as **State machines** and **State-chart Diagrams**. These terms are often used interchangeably. So simply, a state diagram is used to model the dynamic behavior of a class in response to time and changing external stimuli.

2. **Activity Diagrams –** We use Activity Diagrams to illustrate the flow of control in a system. We can also use an activity diagram to refer to the steps involved in the execution of a use case. We model sequential and concurrent activities using activity diagrams. So, we basically depict workflows visually using an activity diagram. An activity diagram focuses on condition of flow and the sequence in which it happens. We describe or depict what causes a particular event using an activity diagram.
3. **Use Case Diagrams –** Use Case Diagrams are used to depict the functionality of a system or a part of a system. They are widely used to illustrate the functional requirements of the system and its interaction with external agents(actors). A use case is basically a diagram representing different scenarios where the system can be used. A use case diagram gives us a high-level view of what the system or a part of the system does without going into implementation details.
4. **Sequence Diagram –** A sequence diagram simply depicts interaction between objects in a sequential order i.e., the order in which these interactions take place. We can also use the terms event diagrams or event scenarios to refer to a sequence diagram. Sequence diagrams describe how and in what order the objects in a system function. These diagrams are widely used by businessmen and software developers to document and understand requirements for new and existing systems.
5. **Communication Diagram –** A Communication Diagram(known as Collaboration Diagram in UML 1.x) is used to show sequenced messages exchanged between objects. A communication diagram focuses primarily on objects and their relationships. We can represent similar information using Sequence diagrams; however, communication diagrams represent objects and links in a free form.
6. **Timing Diagram –** Timing Diagram are a special form of Sequence diagrams which are used to depict the behavior of objects over a time frame. We use them to show time and duration constraints which govern changes in states and behavior of objects.
7. **Interaction Overview Diagram –** An Interaction Overview Diagram models a sequence of actions and helps us simplify complex interactions into simpler occurrences. It is a mixture of activity and sequence diagrams.