Mid 1 SDA

Stakeholders in SE

- **Users**: These are the people who will use the software.
- **Customers/Clients:** These are the people who make the decisions about ordering and paying for the software. They may or may not be users – the users may work for them.
- **Software developers:** These are the people who develop and maintain the software, many of whom may be called software engineers.
- **Development managers:** These are the people who run the organization that is developing the software.

OOP

- Object-oriented programming uses objects, not algorithms, as its fundamental logical building blocks
- Each object is an instance of some class, it is a tangible entity that exhibits some well-defined behavior
- Classes may be related to one another via inheritance relationships

Object Model

For all things object oriented, the conceptual framework is the object model. There are four major elements of this model:

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

Abstraction

- Abstraction is the process of hiding the internal details of an application from the outer world.
- Abstraction is used to describe things in simple terms. It's used to create a boundary between the application and the client programs.
- Abstraction means displaying only essential information and hiding the details.
- Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.
- Example: our car is a great example of abstraction. You can start a car by turning the key or pressing the start button. You don't need to know how the engine is getting started, what all components your car has. The car internal implementation and complex logic is completely hidden from the user.
- Objects are the building blocks of Object-Oriented Programming.
- An object contains some properties and methods. We can hide them from the outer world through **access modifiers**. We can provide access only for required functions and properties to the other programs.

Data Abstraction	Process Abstraction
When the object data is not visible to the outer world	When we hide the internal implementation of the different functions involved in a user operation
If needed, access to the Objects' data is provided through some methods.	allows us to encapsulate a set of operations or steps into a single entity
In data abstraction, we define abstract data types (ADTs) that provide a clear and well-defined interface for manipulating data	
promotes modularity and information hiding, allowing us to separate the interface (public methods) from the implementation (private details)	

Abstraction in Header files	**Abstraction using Access Specifiers**
The **pow()** method present in math.h header file. Pass the numbers as arguments without knowing the underlying algorithm.	• Members declared as **public** in a class can be accessed from anywhere in the program. • Members declared as **private** in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of the code outside the class.

Advantages of Data Abstraction

- Helps the user to avoid writing the low-level code
- Avoids code duplication and increases reusability.
- Can change the internal implementation of the class independently without affecting the user.
- Helps to increase the security of an application or program as only important details are provided to the user.
- It reduces the complexity as well as the redundancy of the code, therefore increasing the readability.

Encapsulation

- encapsulation describes bundling data and methods that work on that data within one unit
- Encapsulation allows you to hide specific information and control access to the object's internal state.
- you can use the encapsulation concept to implement an information-hiding mechanism.
- You implement an information-hiding mechanism by making your class attributes inaccessible from the outside.
- You can also provide getter and/or setter methods for attributes to be readable or updatable by other classes.
- Each modifier specifies a different level of accessibility, and you can only use one modifier per class, method or attribute.
- As a rule of thumb, you should always use the most restrictive modifier that still allows you to implement your business logic.
- Encapsulation involves wrapping data and methods within a class to create protective barrier around them
- By using getters and setters, the class can enforce its own data validation rules and ensure that its internal state remains consistent.
- Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of its own class in which it is declared as **combination of data-hiding and abstraction**.
- Encapsulation can be achieved by declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables

Modularity

- Partitioning a program creates a number of well-defined, documented boundaries within the program.
- Modularization consists of dividing a program into modules which can be compiled separately, but which have connections with other modules.
- The overall goal of the decomposition into modules is the reduction of software cost by allowing modules to be designed and revised independently

Module simply means the software components that are been created by dividing the software

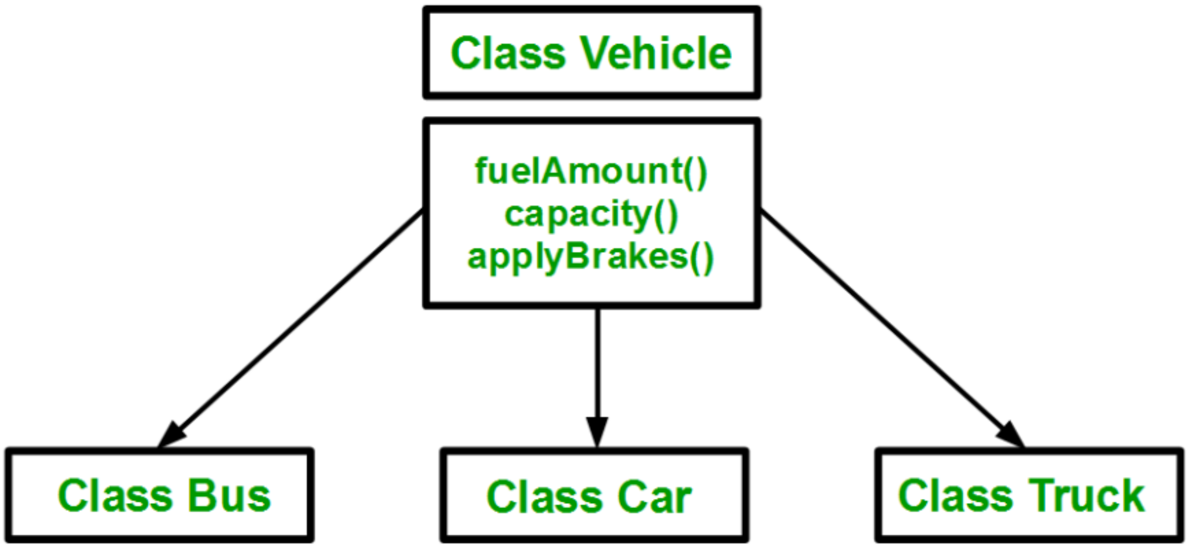
- **Modular Decomposability –** Modular decomposability means to break down the problem into different sub-problems in a systematic manner. The decomposition helps in reducing the complexity of the problem, and sub-problems created can be solved independently. This helps in achieving the basic principle of modularity.
- **Modular Composability –** Composability simply means the ability to combine modules that are created. Modular composability means to assemble the modules into a new system that means to connect the combine the components into a new system.
- **Modular Understandability –** Modular understandability means to make it easier for the user to understand each module so that it is very easy to develop software and change it as per requirement.
- **Modular Continuity –** Modular continuity means making changes to the system requirements that will cause changes in the modules individually without causing any effect or change in the overall system or software.
- **Modular Protection –** Modular protection means to keep safe the other modules from the abnormal condition occurring in a particular module at run time. The abnormal condition can be an error or failure also known as run-time errors. The side effects of these errors are constrained within the module.

Hierarchy/Inheritance

- **Inheritance represents the IS-A relationship.**
- The child class will inherit all the public and protected properties and methods from the parent class. In addition, it can have its own properties and methods.

- **Sub Class:** The class that inherits properties from another class is called Subclass or Derived Class.

- **Super Class:** The class whose properties are inherited by a subclass is called Base Class or Superclass.

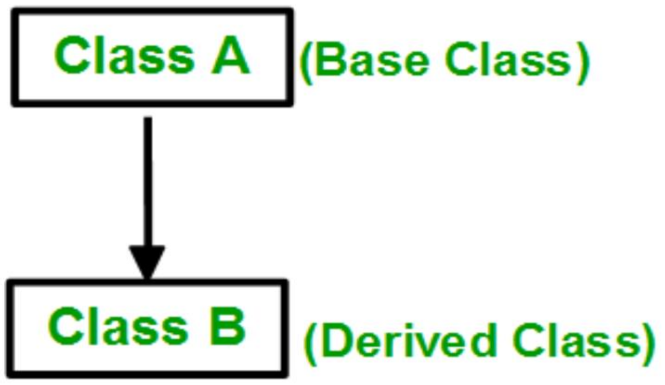


Public	Protected	Private
If we derive a subclass from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in the derived class.	If we derive a subclass from a Protected base class. Then both public members and protected members of the base class will become protected in the derived class.	If we derive a subclass from a Private base class. Then both public members and protected members of the base class will become Private in the derived class.

- The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed

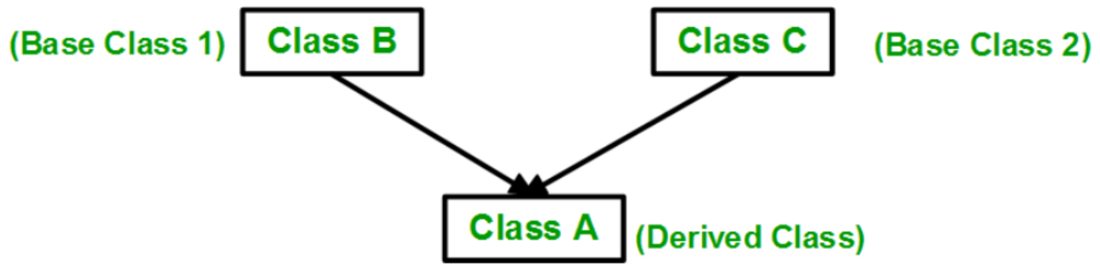
Single Inheritance:

In single inheritance, a class is allowed to inherit from only one class. i.e. one subclass is inherited by one base class only.



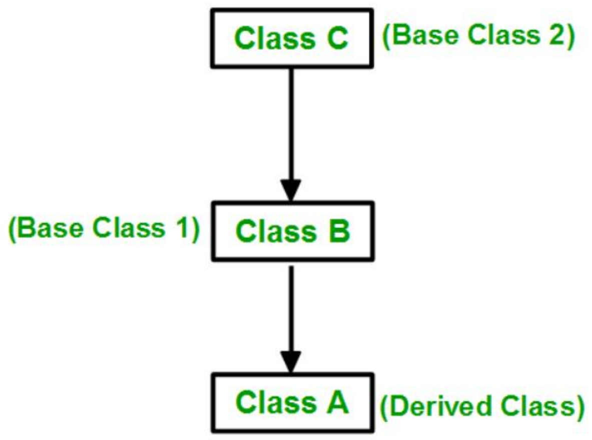
Multiple Inheritance:

Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one **subclass** is inherited from more than one **base class**



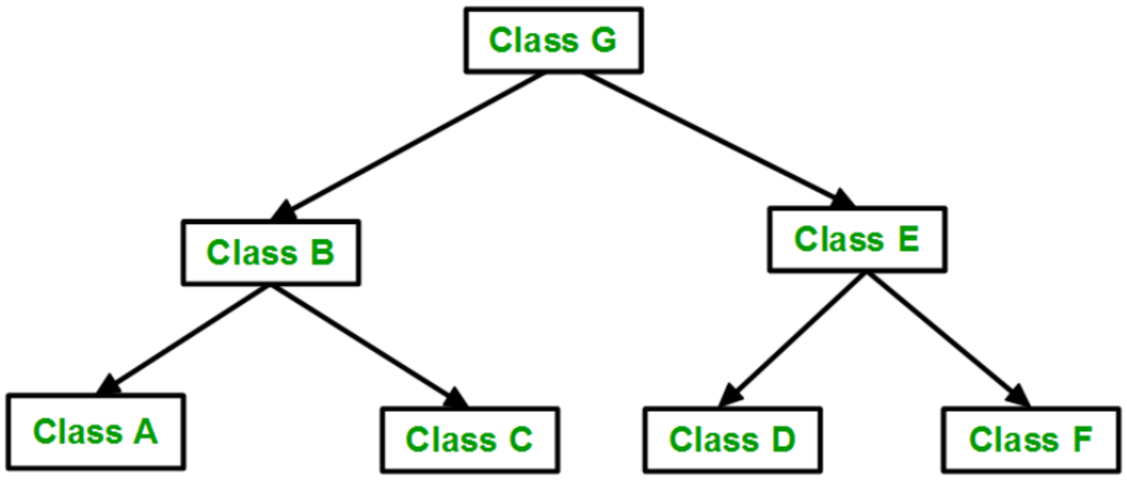
Multilevel Inheritance:

In this type of inheritance, a derived class is created from another derived class.



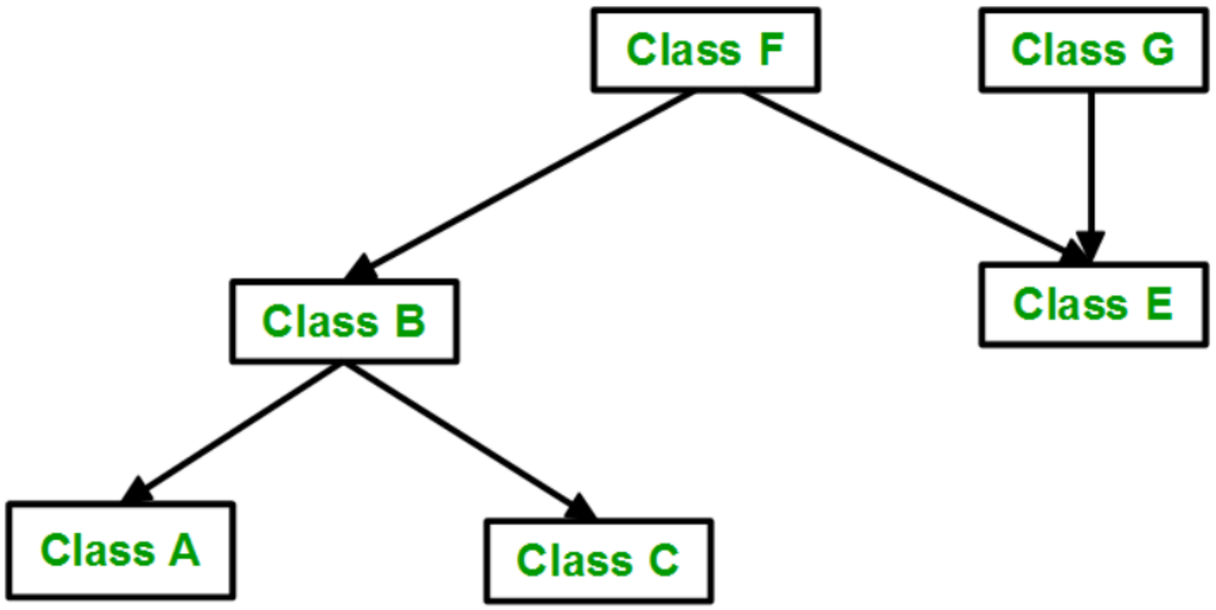
Hierarchical Inheritance:

In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.



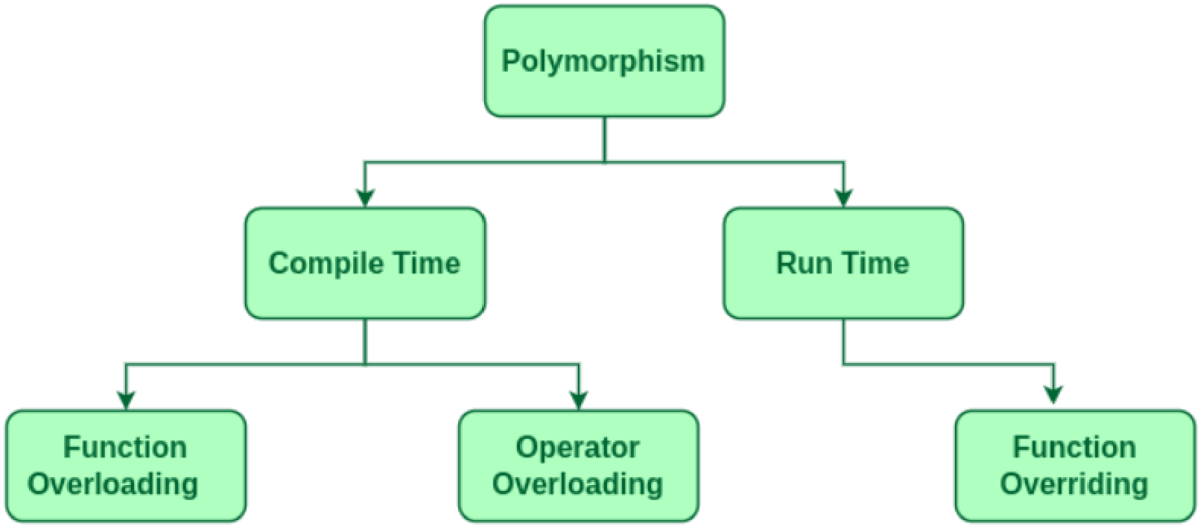
Hybrid (Virtual) Inheritance:

Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.



Polymorphism

Polymorphism is the ability of any data to be processed in more than one form.



Polymorphism is the ability of objects to take on different forms or behave in different ways depending on the context in which they are used.

Compile-Time Polymorphism

Function Overloading	**Operator Overloading**
When there are multiple functions with the same name but different parameters, then the functions are said to be **overloaded**	C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.
Functions can be overloaded by **changing the number of arguments** or/and **changing the type of arguments**.	

Runtime Polymorphism

- This type of polymorphism is achieved by **Function Overriding**. Late binding and dynamic polymorphism are other names for runtime polymorphism.
- The function call is resolved at runtime in runtime polymorphism.
- In contrast, with compile time polymorphism, the compiler determines which function call to bind to the object after deducing it at runtime.

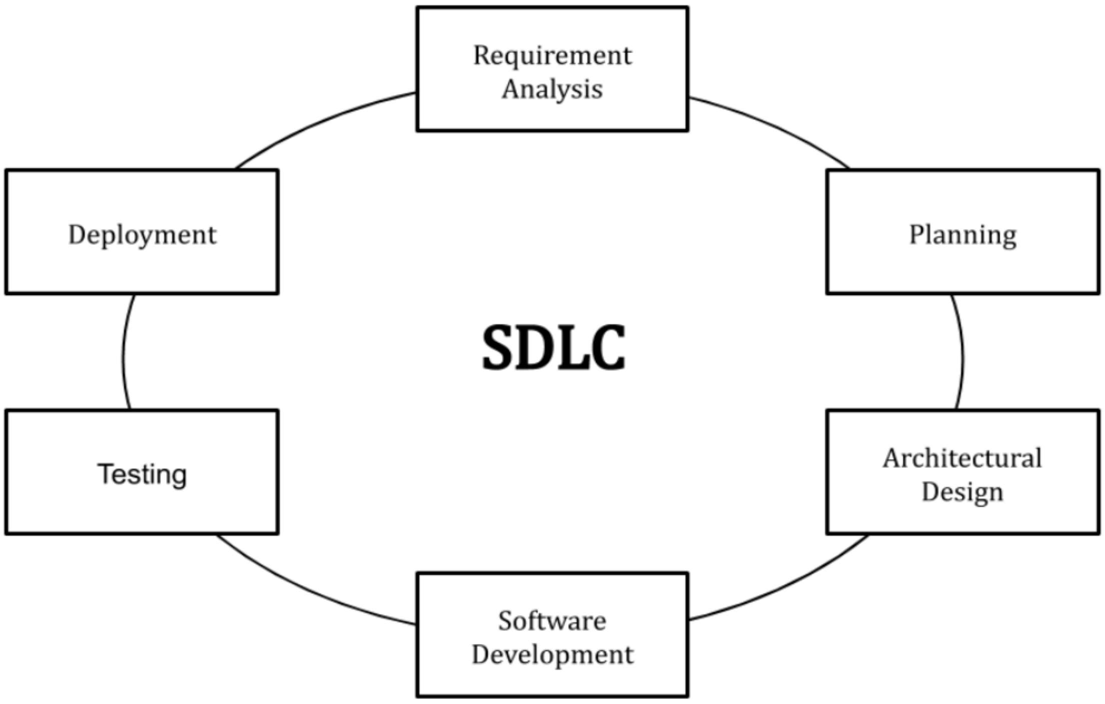
Function Overriding	**Virtual Function**
Function Overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden	A virtual function is a member function that is declared in the base class using the keyword virtual and is re-defined (Overridden) in the derived class.

Some Key Points About Virtual Functions:

- Virtual functions are Dynamic in nature.
- They are defined by inserting the keyword "**virtual**" inside a base class and are always declared with a base class and overridden in a child class
- A virtual function is called during Runtime

SDLC

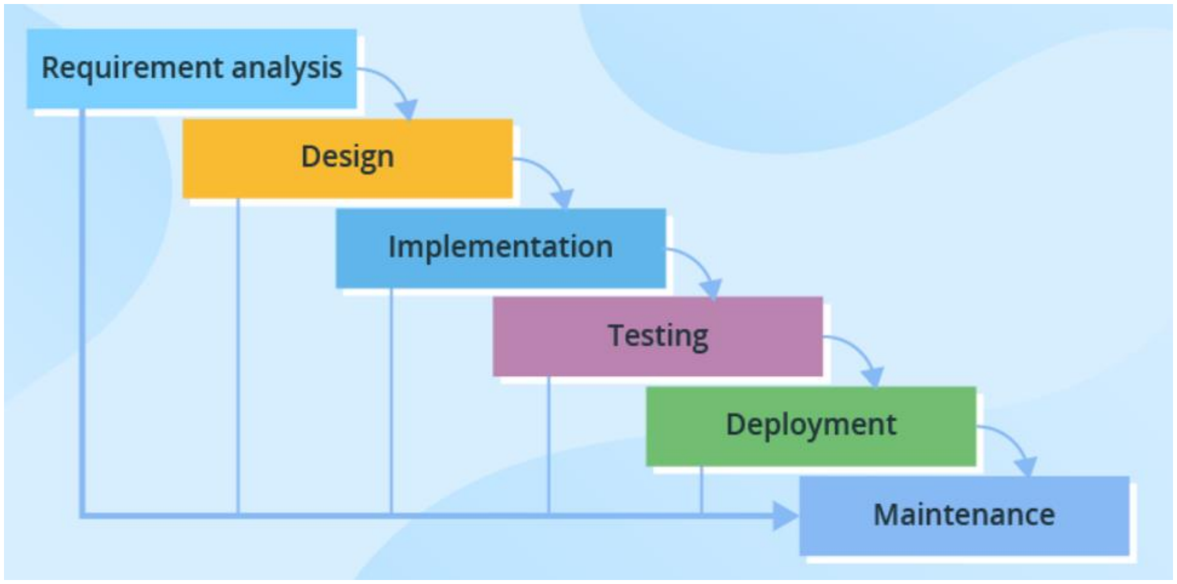
- The Software Development Life Cycle (SDLC) refers to a methodology with clearly defined processes for creating high-quality software. SDLC methodology focuses on the following phases of software development:
    - Requirement analysis
    - Planning
    - Software design such as architectural design
    - Software development
    - Testing
    - Deployment/Maintenance



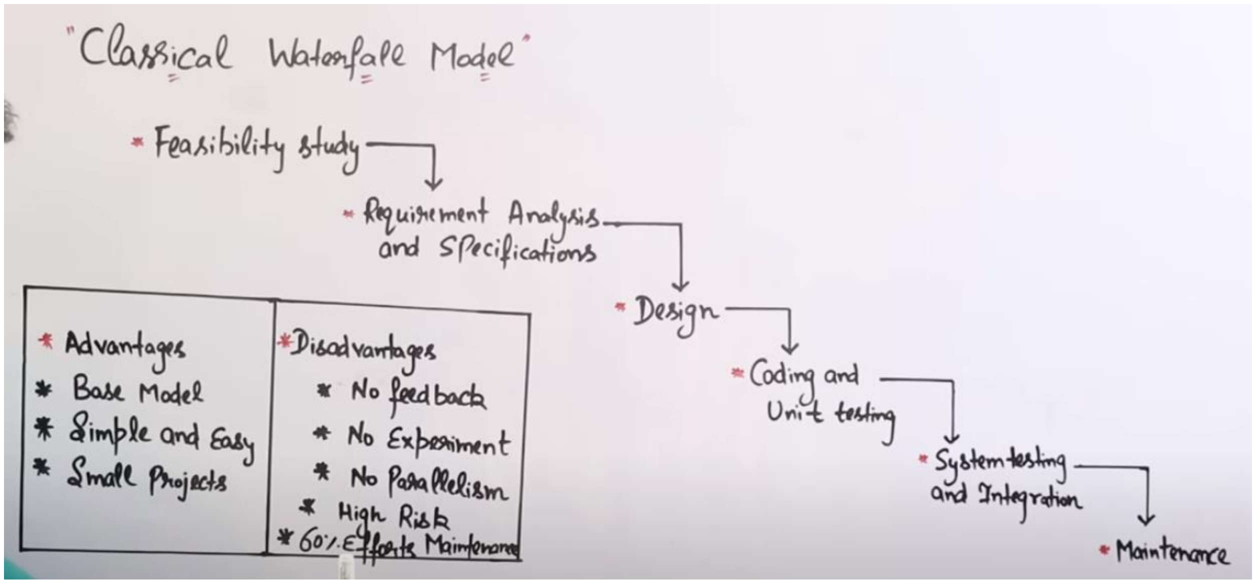
1. **Requirement Analysis:** involves getting input from all stakeholders, including customers, salespeople, industry experts, and programmers.
2. **Planning:** the team determines the cost and resources required for implementing the analyzed requirements. It also details the risks involved and provides sub-plans for softening those risks.
3. **Design:** starts by turning the software specifications into a design plan called the Design Specification. All stakeholders then review this plan and offer feedback and suggestions.
4. **Development:** the actual development starts by writing code.
5. **Testing:** we test for defects and deficiencies. We fix those issues until the product meets the original specifications.
6. **Deployment/Maintenance:** the goal is to deploy the software to the production environment so users can start using the product. During maintenance, as conditions in the real-world change, we need to update and advance the software to match.

Software Models

Waterfall



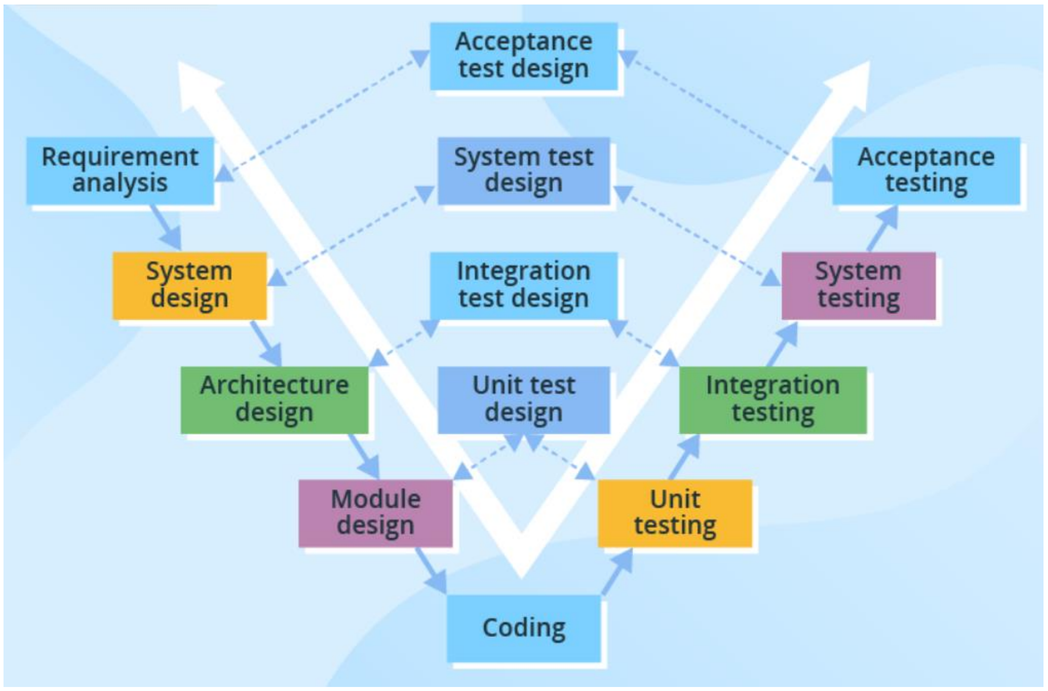
- Through all development stages (analysis, design, coding, testing, deployment), the process moves in a cascade mode.
- Each stage has concrete deliverables and is strictly documented.
- The next stage cannot start before the previous one is fully completed.
- Thus, for example, software requirements cannot be re-evaluated further in development.
- There is also no ability to see and try software until the last development stage is finished, which results in high project risks and unpredictable project results.
- Testing is often rushed, and errors are costly to fix.


"Classical Waterfall Model"
* Feasibility study
* Requirement Analysis and Specifications
* Design
* Coding and Unit testing
* System testing and Integration
* Maintenance

* Advantages	* Disadvantages
* Base Model	* No feedback
* Simple and Easy	* No Experiment
* Small Projects	* No Parallelism
	* High Risk
	* 60% Efforts Maintenance

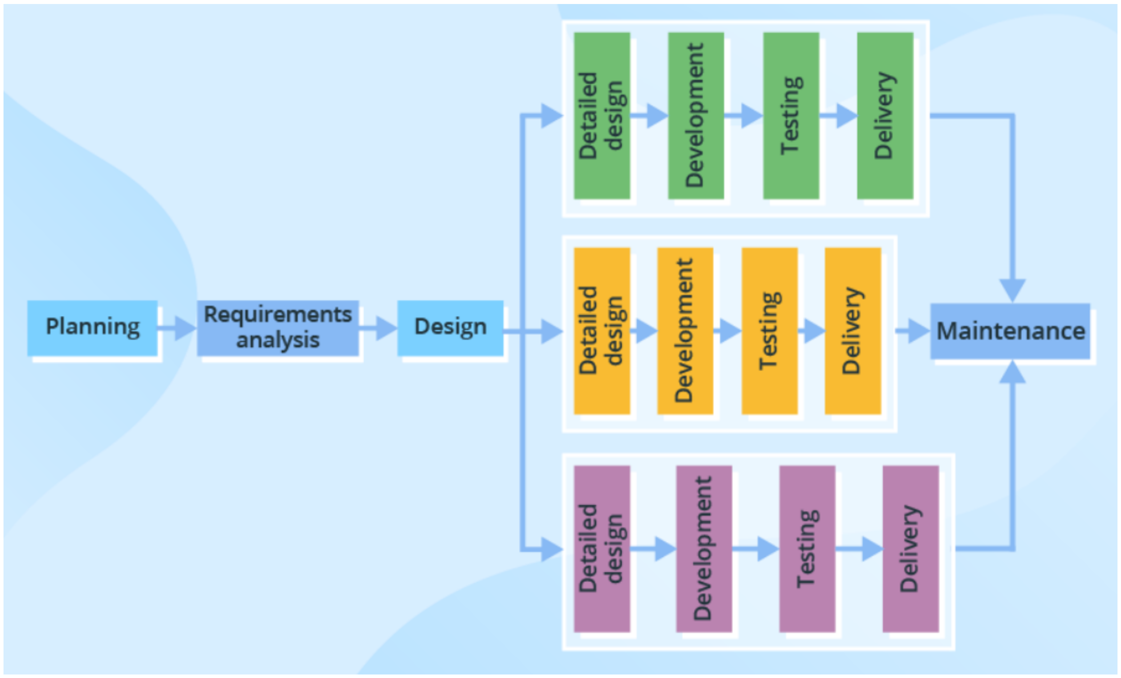
Only difference in Iterative and Classical is that Iterative has Feedback Option.

V-model (Validation and Verification model)



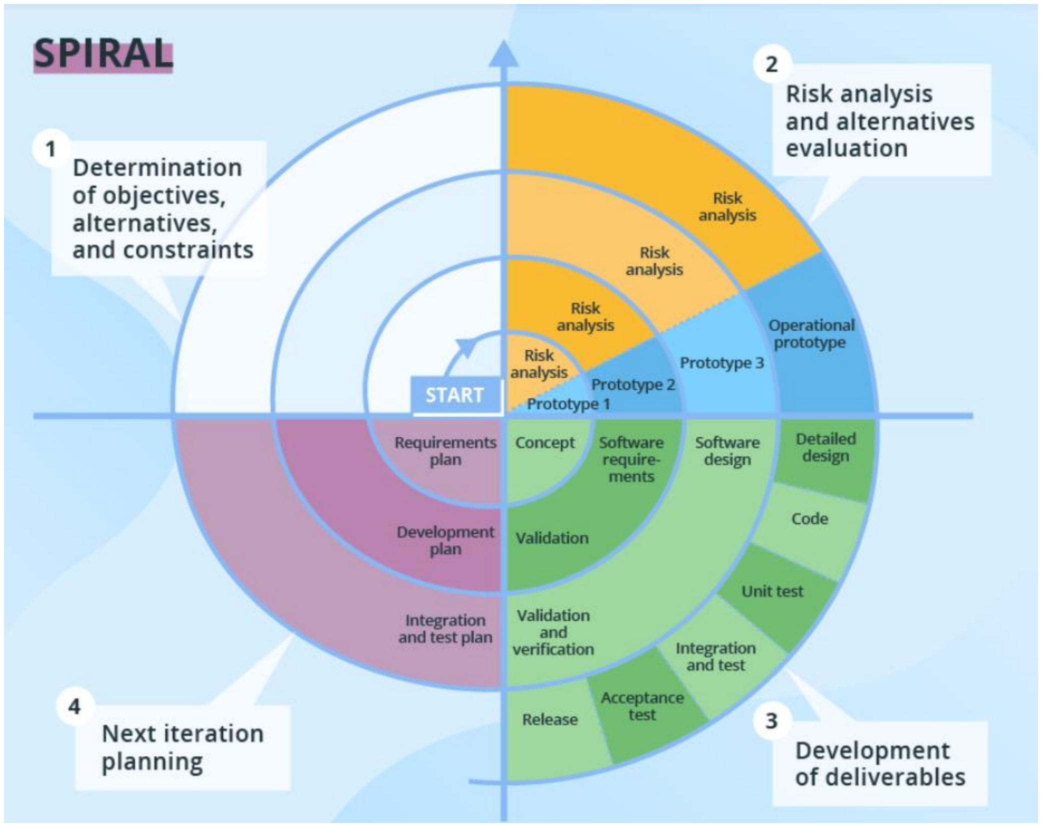
- The V-model is another linear model with each stage having a corresponding testing activity.
- Such workflow organization implies exceptional quality control, but at the same time, it makes the V-model one of the most expensive and time-consuming models.
- Moreover, even though mistakes in requirements specifications, code and architecture errors can be detected early, changes during development are still expensive and difficult to implement.

Incremental model



- The development process based on the **Incremental model** is split into several iterations
- New software modules are added in each iteration with no or little change in earlier added modules.
- The development process can go either sequentially or in parallel.
- Parallel development adds to the speed of delivery, while many repeated cycles of sequential development can make the project long and costly.

Spiral model



- The Spiral model puts focus on thorough risk assessment.
- Thus, to reap the benefits of the model to the fullest, you'll need to engage people with a strong background in risk evaluation.
- A typical Spiral iteration lasts around 6 months and starts with 4 important activities - thorough planning, risk analysis, prototypes creation, and evaluation of the previously delivered part.
- Repeated spiral cycles seriously extend project timeframes.
- This is the model where intensive customer involvement appears.
- They can be involved in the exploration and review stages of each cycle.
- At the development stage, the customer's amendments are not acceptable.

The Agile group

- Nowadays, more than 70% of organizations employ this or that Agile approach in their IT projects.

- In general, at the heart of Agile are iterative development, intensive communication, and early customer feedback.

- Each Agile iteration usually takes several weeks and delivers a complete working software version.

- The models of this group put more focus on delivering a functioning part of the application quickly.

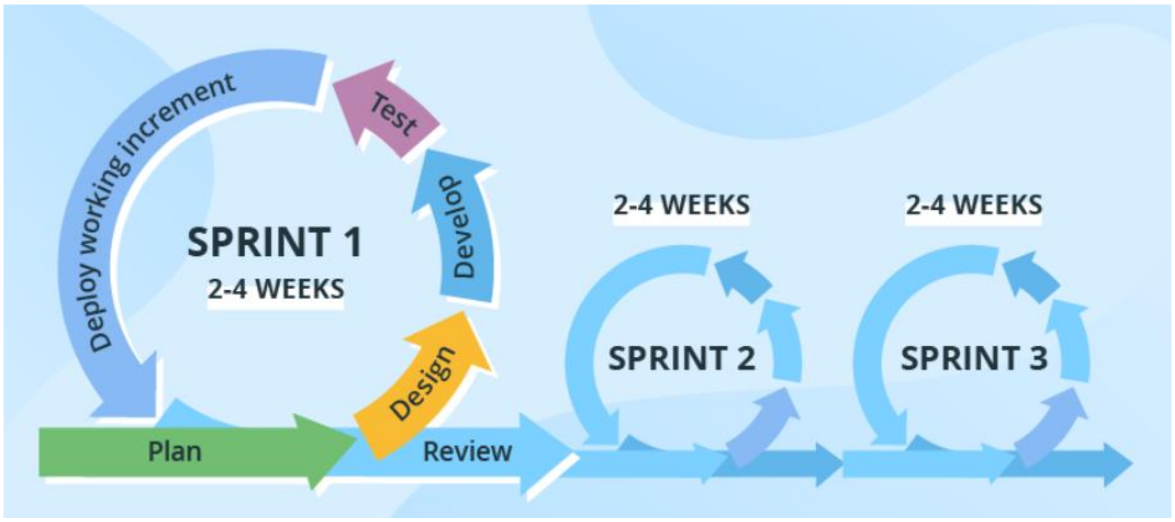
- They pay less attention to detailed software documentation (detailed requirement specification, detailed architecture description), and more to software testing activities.

- Agile is about working in close collaboration both across the team and with the customers. At the end of each iteration, stakeholders review the development progress and re-evaluate the priority of tasks for the future iteration to increase the return on investment (ROI) and ensure alignment with user needs and business goals.

- They also allow for continuous software improvement with easy fixes and changes, quick updates, and feature addition, and help to deliver applications that satisfy users' needs better.

- However, the lack of detailed planning and openness to changes make it difficult to accurately estimate budget, time and people required for the project.

Scrum

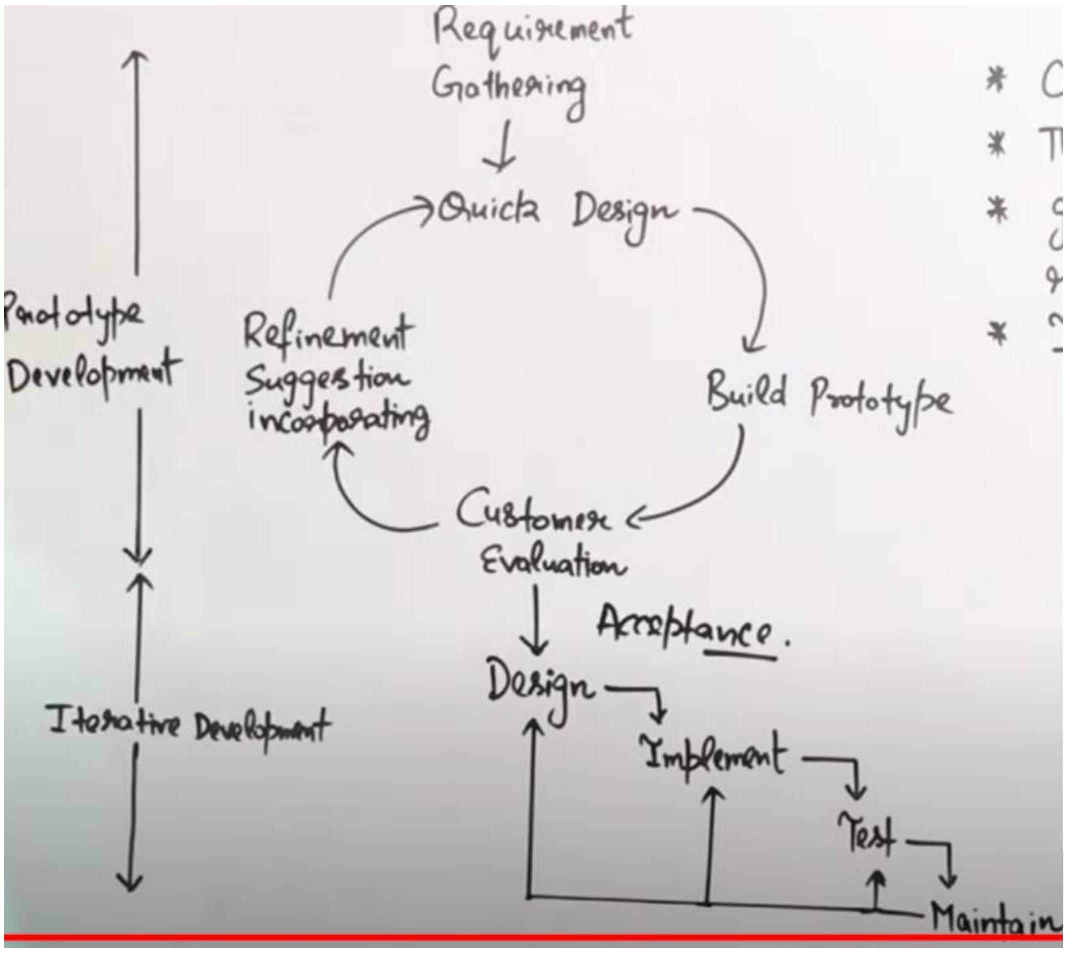


- **Scrum** is probably the most popular Agile model.
- The iterations ('sprints') are usually 2-4 weeks long, and they are preceded with thorough planning and previous sprint assessment.
- No changes are allowed after the sprint activities have been defined.

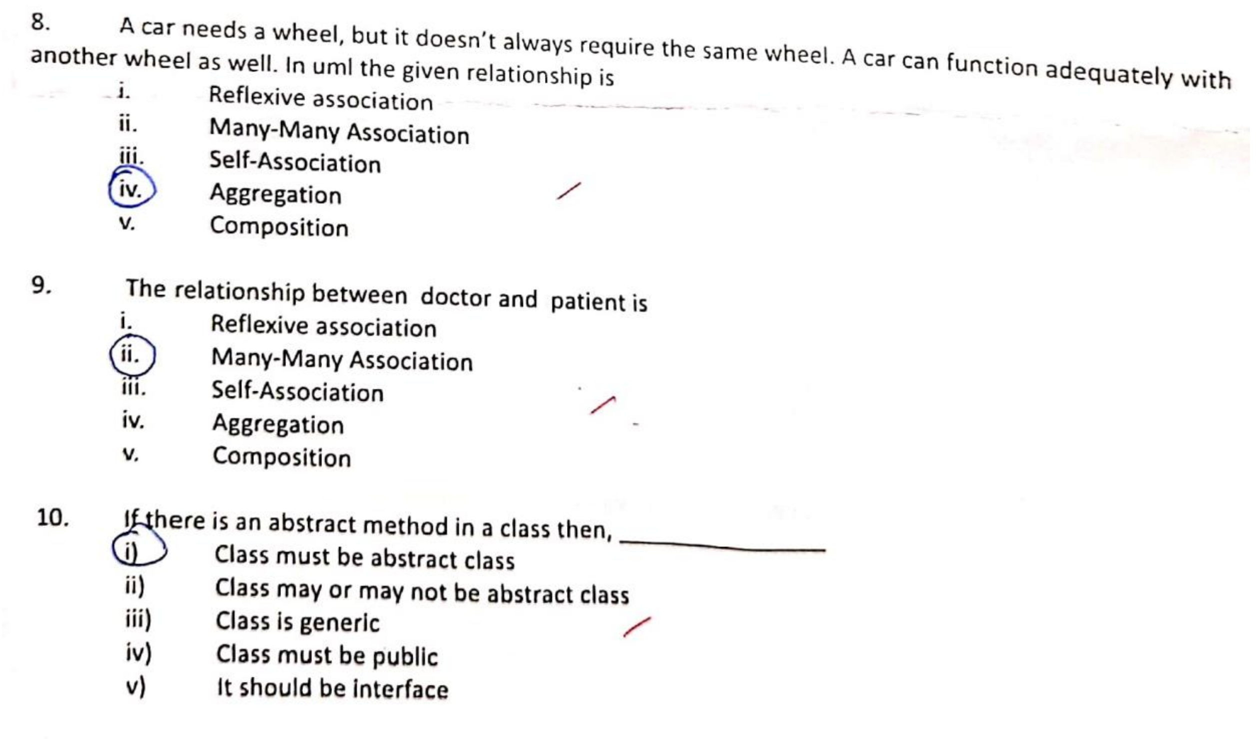
Prototype Model

The prototyping model is a systems development method in which a prototype is built, tested and then reworked as necessary until an acceptable outcome is achieved from which the complete system or product can be developed.

1. The new system requirements are defined in as much detail as possible. This usually involves interviewing a number of users representing all the departments or aspects of the existing system.
2. A preliminary, simple design is created for the new system.
3. The first prototype of the new system is constructed from the preliminary design. This is usually a scaled-down system and represents an approximation of the characteristics of the final product.
4. The users thoroughly evaluate the first prototype and note its strengths and weaknesses, what needs to be added and what should be removed. The developer collects and analyzes the remarks from the users.
5. The first prototype is modified, based on the comments supplied by the users and a second prototype of the new system is constructed.
6. The second prototype is evaluated in the same manner as the first prototype.
7. The preceding steps are iterated as many times as necessary, until the users are satisfied that the prototype represents the final product desired.
8. The final system is constructed, based on the final prototype.
9. The final system is thoroughly evaluated and tested. Routine maintenance is carried out on a continuing basis to prevent large-scale failures and to minimize downtime.



Incremental: This technique breaks the concept for the final product into smaller pieces and prototypes are created for each one. In the end, these prototypes are merged into the final product.



8. A car needs a wheel, but it doesn't always require the same wheel. A car can function adequately with another wheel as well. In uml the given relationship is

- i. Reflexive association
- ii. Many-Many Association
- iii. Self-Association
- [✓ iv.] Aggregation ✓
- v. Composition

9. The relationship between doctor and patient is

- i. Reflexive association
- [✓ ii.] Many-Many Association
- iii. Self-Association ✓
- iv. Aggregation
- v. Composition

10. If there is an abstract method in a class then, ____________

- [✓ i)] Class must be abstract class
- ii) Class may or may not be abstract class
- iii) Class is generic ✓
- iv) Class must be public
- v) It should be interface

UML

- UML diagrams can be classified into two groups: **structure diagrams** and **behavior diagrams.**
- System complexity is driven both by the number and organization of elements in the system (i.e., structure) and the way all these elements collaborate to perform their function (i.e., behavior).

Structure Diagrams

- These diagrams are used to show the **static structure** of elements in the system, **architectural organization** of the system, the **physical elements** of the system, its **runtime configuration**.
- Structure diagrams are often used in conjunction with behavior diagrams to depict a particular aspect of your system. Each class may have an associated state machine diagram that indicates the event-driven behavior of the class's instances.

Following are the Structure Diagrams in UML:

- Package diagram
- Class diagram

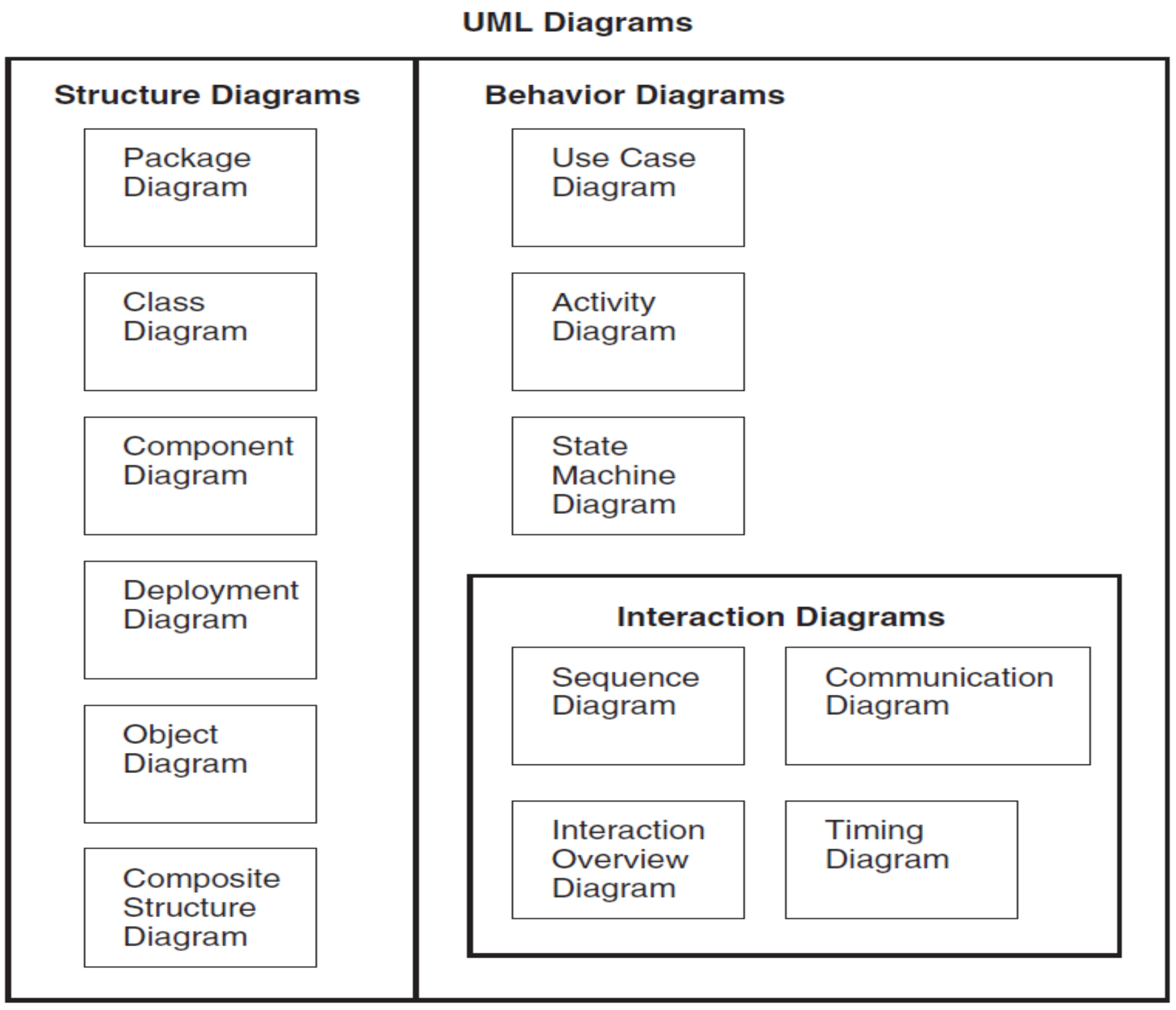
- Component diagram
- Deployment diagram
- Object diagram
- Composite structure diagram

Behavior Diagrams

- Events happen **dynamically** in all software-intensive systems: Objects are **created and destroyed**, objects **send messages** to one another in an orderly fashion, external events **trigger operations** on certain objects.

Following are the Behavior Diagrams in UML:

- Use case diagram
- Activity diagram
- State machine diagram
- Interaction diagrams
- Sequence diagram
- Communication diagram
- Interaction overview diagram
- Timing diagram



Association

- Association is a semantically weak relationship (a semantic dependency) between otherwise unrelated objects.
- An association is a “using” relationship between two or more objects in which the objects have their own lifetime and there is no owner.
- As an example, imagine the relationship between a doctor and a patient. A doctor can be associated with multiple patients. At the same time, one patient can visit multiple doctors for treatment or consultation
- Each of these objects has its own life cycle and there is no “owner” or parent. The objects that are part of the association relationship can be created and destroyed independently.

- Association can be one-to-one, one-to-many, many-to-one, many-to-many.

Aggregation

- **It represents a "part of" relationship.**
- Aggregation is a specialized form of association between two or more objects in which each object has its own life cycle but there exists an ownership as well.

- Aggregation is a typical whole/part or parent/child relationship but it may or may not denote physical containment.

- An essential property of an aggregation relationship is that the whole or parent (i.e. the owner) can exist without the part or child and vice versa.

- As an example, an employee may belong to one or more departments in an organization. However, if an employee's department is deleted, the employee object would not be destroyed but would live on. Note that the relationships between objects participating in an aggregation cannot be reciprocal—i.e., a department may "own" an employee, but the employee does not own the department

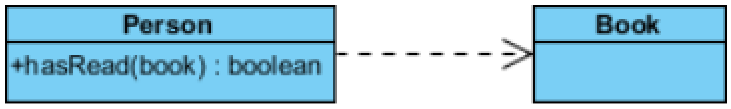
Composition

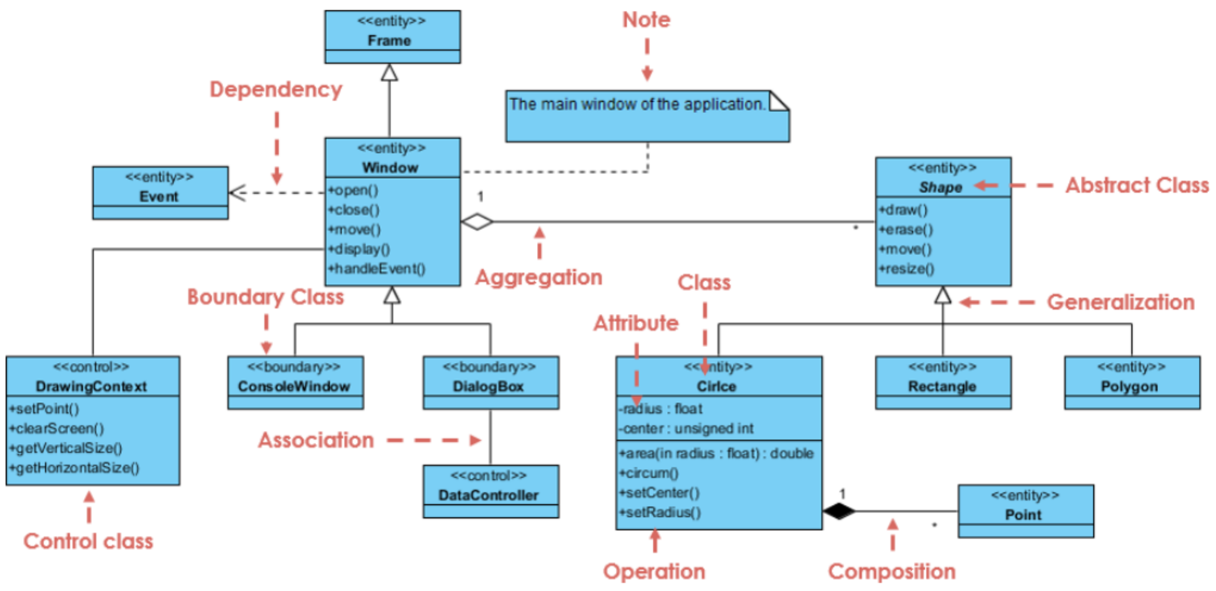
- Composition is a specialized form of aggregation.
- In composition, if the parent object is destroyed, then the child objects also cease to exist.
- Composition is actually a strong type of aggregation and is sometimes referred to as a "death" relationship.
- As an example, a house may be composed of one or more rooms. If the house is destroyed, then all of the rooms that are part of the house are also destroyed.
- Like aggregation, composition is also a whole/part or parent/child relationship.
- However, in composition the life cycle of the part or child is controlled by the whole or parent that owns it.
- It should be noted that this control can either be direct or transitive. That is, the parent may be directly responsible for the creation or destruction of the child or the parent may use a child that has been already created.
- Similarly, a parent object might delegate the control to some other parent to destroy the child object.

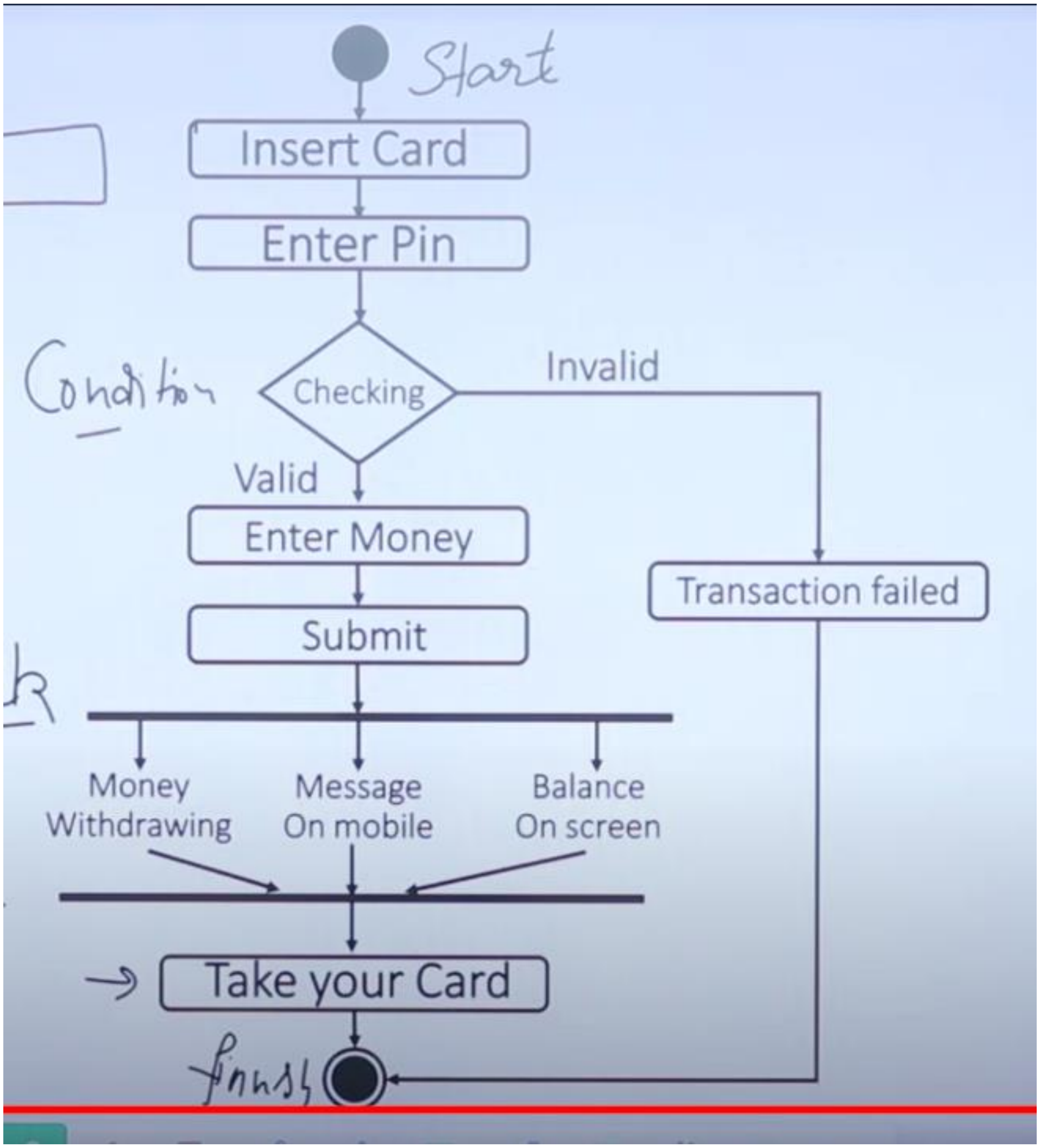
Dependency

An object of one class might use an object of another class in the code of a method. If the object is not stored in any field, then this is modeled as a dependency relationship.

The Person class might have a hasRead method with a Book parameter that returns true if the person has read the book (perhaps by checking some database).






Start
Insert Card
Enter Pin
Condition
Checking
Invalid
Valid
Enter Money
Transaction failed
Submit
Money Withdrawing
Message On mobile
Balance On screen
Take your Card
finish