SOFTWARE ENGINEERING

UNIT-3

Software Design:

Software design is a mechanism to transform user requirements into some suitable form, which helps the programmer in software coding and implementation. In software design, we consider the system to be a set of components or modules with clearly defined behaviors & boundaries.

It deals with representing the client's requirement, as described in SRS (Software Requirement Specification) document, into a form, i.e., easily implementable using programming language.

The software design phase is the first step in SDLC (Software Design Life Cycle), which moves the concentration from the problem domain to the solution domain.

Objectives of Software Design:

1. **Correctness:** Software design should be correct as per requirement.
2. **Completeness:** The design should have all components like data structures, modules, and external interfaces, etc.
3. **Efficiency:** Resources should be used efficiently by the program.
4. **Flexibility:** Able to modify on changing needs.
5. **Consistency:** There should not be any inconsistency in the design.
6. **Maintainability:** The design should be so simple so that it can be easily maintainable by other designers.

Software Design Principles:

1. Should not suffer from "Tunnel Vision":

While designing the process, it should not suffer from "tunnel vision" which means that is should not only focus on completing or achieving the aim but on other effects also.

2. Traceable to analysis model:

The design process should be traceable to the analysis model which means it should satisfy all the requirements that software requires to develop a high-quality product.

3. Should not "Reinvent the Wheel":

The design process should not reinvent the wheel that means it should not waste time or effort in creating things that already exist. Due to this, the overall development will get increased.

4. Minimize Intellectual distance:

The design process should reduce the gap between real-world problems and software solutions for that problem meaning it should simply minimize intellectual distance.

5. Exhibit uniformity and integration:

The design should display uniformity which means it should be uniform throughout the process without any change. Integration means it should mix or combine all parts of software i.e. subsystems into one system.

6. Accommodate change:

The software should be designed in such a way that it accommodates the change implying that the software should adjust to the change that is required to be done as per the user’s need.

7. Degrade gently:

The software should be designed in such a way that it degrades gracefully which means it should work properly even if an error occurs during the execution.

8. Assessed or quality:

The design should be assessed or evaluated for the quality meaning that during the evaluation, the quality of the design needs to be checked and focused on.

9. Review to discover errors:

The design should be reviewed which means that the overall evaluation should be done to check if there is any error present or if it can be minimized.

10. Design is not coding and coding is not design:

Design means describing the logic of the program to solve any problem and coding is a type of language that is used for the implementation of a design.

Software Design Methodologies:

Software design methodology provides a logical and systematic means of proceeding with the design process as well as a set of guidelines for decision-making. The design methodology provides a sequence of activities, and often uses a set of notations or diagrams.

Types of data methodologies:

1. Design Sprint methodology

Reducing risk before a new product hits the market or a new feature is introduced is more than welcome in virtually any industry. And this is when Design Sprint comes into play, offering a five-phase process encompassing stages like Understand, Diverge, Converge, Prototype, and Test.

Interestingly, this method of addressing strategic issues draws upon both design thinking and Agile methodologies – and time-boxed sprints as for the latter. Within the cycle, a high-fidelity interactive prototype gets tested by real users. The bottom line is that guesswork is eliminated and you may find out within a few days, and not months, if the product in question is worth developing, or not.

2. Lean Startup methodology

Eliminating uncertainty startups inevitably have to deal with is a hallmark of another design methodology – Lean Startup. It was born out of the willingness to engineer the startup success, by following the process suggested.

It all started in the year 2011 when the book „The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses" by Eric Ries was published. But it was based on the author's experience and failure to succeed with launching a new start-up enterprise.

This led him to the conclusion that more emphasis should be put on understanding the needs of prospective customers, and not on the act of product launch – one that people may actually don't want – per se. Moreover, by testing their visions continuously, new businesses may choose order instead of chaos.

3. Product discovery

This approach also puts attention to the greater importance of things like preparatory and preliminary actions and continuous learning about users. Researching and validating ideas upfront may be the new enterprise's salvation.

In short, users' true needs or problems should be identified first, and next, solutions to them should be designed, developed, and delivered collaboratively. Not the other way round.

Product discovery is to weaken risks involved that concern value, usability, feasibility, and business viability, and risk management, in fact, is product management.

4. Wizard of Oz prototyping

Trying to figure out what customers might think about given products is in the center of the Wizard of Oz (WOZ) prototyping. This method involves some role-playing with the use of a prototype test – a physical product or working model.

There are two roles to be filled – of an end-user and the person that is to enact the completed product's performance. The motive behind those actions – which can only take one hour in some cases – is to test functionalities before they exist and find out which ones are necessary.

5. Concierge testing

This down-to-earth methodology is about immersing into potential customers' natural environment (like shops or cafes) for gathering their insights on hypothetical products. Participants' answers to directly asked questions as well as their emotional responses are collected. Concierge testing is sometimes confused with WOZ but the difference between them is that the former is designed for generating ideas, and the latter – for testing hypotheses.

Software Design Concepts:

1. Abstraction

An abstraction is a tool that enables a designer to consider a component at an abstract level without bothering about the internal details of the implementation. Abstraction can be used for existing element as well as the component being designed.

There are three commonly used abstraction mechanisms in software design, namely, functional abstraction, data abstraction and control abstraction. All these mechanisms allow us to control the complexity of the design process by proceeding from the abstract design model to concrete design model in a systematic manner.

(A) **Functional abstraction:**

This involves the use of parameterized subprograms. Functional abstraction can be generalized as collections of subprograms referred to as 'groups'. Within these groups there exist routines which may be visible or hidden. Visible routines can be used within the containing groups as well as within other groups, whereas hidden routines are hidden from other groups and can be used within the containing group only.

(B) **Data abstraction:**

This involves specifying data that describes a data object. For example, the data object window encompasses a set of attributes (window type, window dimension) that describe the window object clearly. In this abstraction mechanism, representation and manipulation details are ignored.

(C) Control abstraction:

This states the desired effect, without stating the exact mechanism of control. For example, if and while statements in programming languages (like C and C++) are abstractions of machine code implementations, which involve conditional instructions. In the architectural design level, this abstraction mechanism permits specifications of sequential subprogram and exception handlers without the concern for exact details of implementation.

2. Modularity

Modularity specifies to the division of software into separate modules which are differently named and addressed and are integrated later on in to obtain the completely functional software. It is the only property that allows a program to be intellectually manageable. Single large programs are difficult to understand and read due to a large number of reference variables, control paths, global variables, etc.

Advantages of Modularity

- It allows large programs to be written by several or different people
- It encourages the creation of commonly used routines to be placed in the library and used by other programs.
- It simplifies the overlay procedure of loading a large program into main storage.
- It provides more checkpoints to measure progress.
- It provides a framework for complete testing, more accessible to test
- It produced the well designed and more readable program.

Disadvantages of Modularity

- Execution time maybe, but not certainly, longer
- Storage size perhaps, but is not certainly, increased
- Compilation and loading time may be longer
- Inter-module communication problems may be increased
- More linkage required, run-time may be longer, more source lines must be written, and more documentation has to be done

3. Architecture:

Software architecture refers to the structure of the system, which is composed of various components of a program/ system, the attributes (properties) of those components and the relationship amongst them. The software architecture enables the software engineers to analyze the software design efficiently. In addition, it also helps them in decision-making and handling risks. The software architecture does the following.

- Provides an insight to all the interested stakeholders that enable them to communicate with each other.
- Highlights early design decisions, which have great impact on the software engineering activities (like coding and testing) that follow the design phase.
- Creates intellectual models of how the system is organized into components and how these components interact with each other.

Currently, software architecture is represented in an informal and unplanned manner. Though the architectural concepts are often represented in the infrastructure and the initial stages of a system configuration, the lack of an explicit independent characterization of architecture restricts the advantages of this design concept in the present scenario.

4. Patterns:

A pattern provides a description of the solution to a recurring design problem of some specific domain in such a way that the solution can be used again and again. The objective of each pattern is to provide an insight to a designer who can determine the following.

1. Whether the pattern can be reused
2. Whether the pattern is applicable to the current project
3. Whether the pattern can be used to develop a similar but functionally or structurally different design pattern.

Types of Design Patterns:

Software engineer can use the design pattern during the entire software design process. When the analysis model is developed, the designer can examine the problem description at different levels of abstraction to determine whether it complies with one or more of the following types of design patterns.

1. Architectural patterns

These patterns are high-level strategies that refer to the overall structure and organization of a software system. That is, they define the elements of a software system such as subsystems, components, classes, etc. In addition, they also indicate the relationship between the elements along with the rules and guidelines for specifying these relationships. Note that architectural patterns are often considered equivalent to software architecture.

2. Design patterns

These patterns are medium-level strategies that are used to solve design problems. They provide a means for the refinement of the elements (as defined by architectural pattern) of a software system or the relationship among them. Specific design elements such as relationship among components or mechanisms that affect component-to-component interaction are addressed by design patterns. Note that design patterns are often considered equivalent to software components.

3. Idioms

These patterns are low-level patterns, which are programming-language specific. They describe the implementation of a software component, the method used for interaction among software components, etc., in a specific programming language. Note that idioms are often termed as coding patterns.

5. Refactoring

Refactoring is an important design activity that reduces the complexity of module design keeping its behaviour or function unchanged. Refactoring can be defined as a process of modifying a software system to improve the internal structure of design without changing its external behavior.

During the refactoring process, the existing design is checked for any type of flaws like redundancy, poorly constructed algorithms and data structures, etc., in order to improve the design.

For example, a design model might yield a component which exhibits low cohesion. Software designers may decide to refactor the component into four different components, each exhibiting high cohesion. This leads to easier integration, testing, and maintenance of the software components.

6. Information Hiding

Modules should be specified and designed in such a way that the data structures and processing details of one module are not accessible to other modules. They pass only that much information to each other, which is required to accomplish the software functions. The way of hiding unnecessary details is referred to as information hiding.

IEEE defines information hiding as 'the technique of encapsulating software design decisions in modules in such a way that the module's interfaces reveal as little as possible about the module's inner workings; thus, each module is a 'black box' to the other modules in the system.

Information hiding is of immense use when modifications are required during the testing and maintenance phase. Some of the advantages associated with information hiding are listed below.

- Leads to low coupling
- Emphasizes communication through controlled interfaces
- Decreases the probability of adverse effects
- Restricts the effects of changes in one component on others
- Results in higher quality software.

7. Stepwise Refinement

Stepwise refinement is a top-down design strategy used for decomposing a system from a high level of abstraction into a more detailed level (lower level) of abstraction. At the highest level of abstraction, function or information is defined conceptually without providing any information about the internal workings of the function or internal structure of the data. As we proceed towards the lower levels of abstraction, more and more details are available.

Software designers start the stepwise refinement process by creating a sequence of compositions for the system being designed. Each composition is more detailed than the previous one and contains more components and interactions. The earlier compositions represent the significant interactions within the system, while the later compositions show in detail how these interactions are achieved.

To have a clear understanding of the concept, let us consider an example of stepwise refinement. Every computer program comprises input, process, and output.

8. Concurrency

Concurrency has limited resources and they must be utilized efficiently as much as possible. To utilize these resources efficiently, multiple tasks must be executed concurrently. This requirement makes concurrency one of the major concepts of software design. Every system must be designed to allow multiple processes to execute concurrently, whenever possible. For example, if the current process is waiting for some event to occur, the system must execute some other process in the meantime.

One way to achieve synchronization is mutual exclusion, which ensures that two concurrent processes do not interfere with the actions of each other. To ensure this, mutual exclusion may use locking technique. In this technique, the processes need to lock the data item to be read or updated. The data item locked by some process cannot be accessed by other processes until it is unlocked. It implies that the process, that needs to access the data item locked by some other process, has to wait.

9. Developing a Design Model

To develop a complete specification of design (design model), four design models are needed. These models are listed below.

1. **Data design:** This specifies the data structures for implementing the software by converting data objects and their relationships identified during the analysis phase. Various studies suggest that design engineering should begin with data design, since this design lays the foundation for all other design models.
2. **Architectural design:** This specifies the relationship between the structural elements of the software, design patterns, architectural styles, and the factors affecting the ways in which architecture can be implemented.
3. **Component-level design:** This provides the detailed description of how structural elements of software will actually be implemented.
4. **Interface design:** This depicts how the software communicates with the system that interoperates with it and with the end-users.

Design Process

Software design process can be perceived as series of well-defined steps. Though it varies according to design approach (function oriented or object oriented), yet It may have the following steps involved:

- A solution design is created from requirement or previous used system and/or system sequence diagram.
- Objects are identified and grouped into classes on behalf of similarity in attribute characteristics.
- Class hierarchy and relation among them is defined.
- Application framework is defined.

Software Design Approaches:

Here are two generic approaches for software designing:

Top-Down Design

We know that a system is composed of more than one sub-system and it contains a number of components. Further, these sub-systems and components may have they're on set of sub-system and components and creates hierarchical structure in the system.

Top-down design takes the whole software system as one entity and then decomposes it to achieve more than one sub-system or component based on some characteristics. Each sub-system or component is then treated as a system and decomposed further. This process keeps on running until the lowest level of system in the top-down hierarchy is achieved.

Top-down design starts with a generalized model of system and keeps on defining the more specific part of it. When all components are composed the whole system comes into existence.

Top-down design is more suitable when the software solution needs to be designed from scratch and specific details are unknown.

Bottom-up Design

The bottom-up design model starts with most specific and basic components. It proceeds with composing higher level of components by using basic or lower-level components. It keeps creating higher level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased.

Bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.

Both, top-down and bottom-up approaches are not practical individually. Instead, a good combination of both is used.

Data design:

Data design is the first design activity, which results in less complex, modular and efficient program structure. The information domain model developed during analysis phase is transformed into data structures needed for implementing the software. The data objects, attributes, and relationships depicted in entity relationship diagrams and the information stored in data dictionary provide a base for data design activity. During the data design process, data types are specified along with the integrity rules required for the data. For specifying and designing efficient data structures, some principles should be followed.

These principles are listed below:

1. The data structures needed for implementing the software as well-as the operations that can be applied on them should be identified.
2. A data dictionary should be developed to depict how different data objects interact with each other and what constraints are to be imposed on the elements of data structure.
3. Stepwise refinement should be used in data design process and detailed design decisions should be made later in the process.
4. Only those modules that need to access data stored in a data structure directly should be aware of the representation of the data structure.
5. A library containing the set of useful data structures along with the operations that can be performed on them should be maintained.
6. Language used for developing the system should support abstract data types.

The structure of data can be viewed at three levels, namely, program component level, application level, and business level. At the program component level, the design of data structures and the algorithms required to manipulate them is necessary, if high-quality software is desired. At the application level, it is crucial to convert the data model into a database so that the specific business objectives of a system could be achieved. At the business level, the collection of information stored in different databases should be reorganized into data warehouse, which enables data mining that has an influential impact on the business.

Architectural design:

The software needs the architectural design to represents the design of software. IEEE defines architectural design as “the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.” The software that is built for computer-based systems can exhibit one of these many architectural styles.

An architectural design performs the following functions:

1. It defines an abstraction level at which the designers can specify the functional and performance behaviour of the system.
2. It acts as a guideline for enhancing the system (whenever required) by describing those features of the system that can be modified easily without affecting the system integrity.
3. It evaluates all top-level designs.
4. It develops and documents top-level design for the external and internal interfaces.
5. It develops preliminary versions of user documentation.
6. It defines and documents preliminary test requirements and the schedule for software integration.
7. The sources of architectural design are listed below.
8. Information regarding the application domain for the software to be developed
9. Using data-flow diagrams
10. Availability of architectural patterns and architectural styles.

Types of Architectural styles:

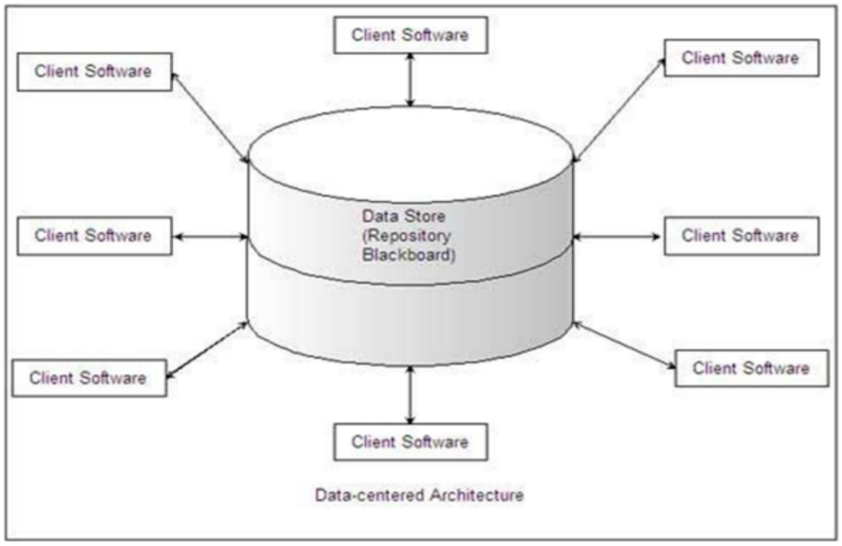
1. Data centered architectures:

A data store will reside at the center of this architecture and is accessed frequently by the other components that update, add, delete or modify the data present within the store.

The figure illustrates a typical data centered style. The client software accesses a central repository. Variation of this approach are used to transform the repository into a blackboard when data related to client or data of interest for the client change the notifications to client software.

This data-centered architecture will promote integrability. This means that the existing components can be changed and new client components can be added to the architecture without the permission or concern of other clients.

Data can be passed among clients using blackboard mechanism.



Advantage of Data centered architecture

- Repository of data is independent of clients
- Client work independent of each other
- It may be simple to add additional clients.
- Modification can be very easy

2. Data flow architectures:

This kind of architecture is used when input data to be transformed into output data through a series of computational manipulative components.

The figure represents pipe-and-filter architecture since it uses both pipe and filter and it has a set of components called filters connected by pipes.

Pipes are used to transmit data from one component to the next.

Each filter will work independently and is designed to take data input of a certain form and produces data output to the next filter of a specified form. The filters don't require any knowledge of the working of neighboring filters.

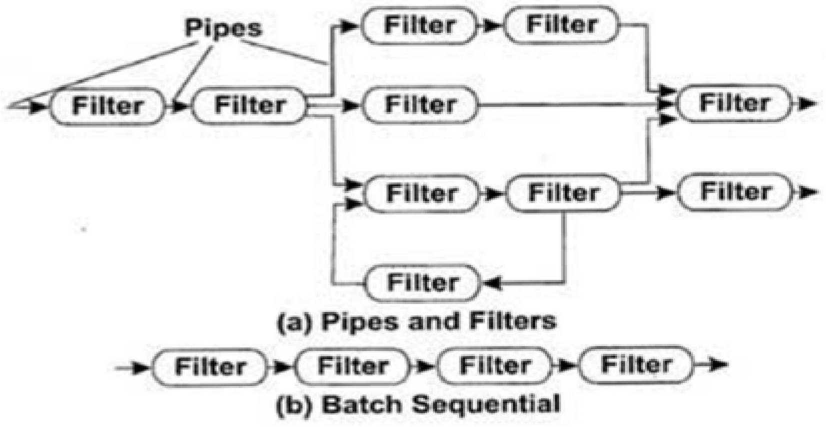
If the data flow degenerates into a single line of transforms, then it is termed as batch sequential. This structure accepts the batch of data and then applies a series of sequential components to transform it.

Advantage of Data Flow architecture

- It encourages upkeep, repurposing, and modification.
- With this design, concurrent execution is supported.

Disadvantage of Data Flow architecture

- It frequently degenerates to batch sequential system
- Data flow architecture does not allow applications that require greater user engagement.
- It is not easy to coordinate two different but related streams

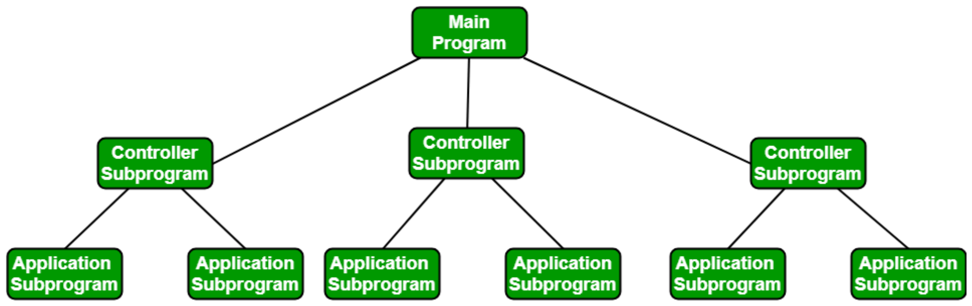


Data-flow Architecture

3. Call and Return architectures:

It is used to create a program that is easy to scale and modify. Many sub-styles exist within this category. Two of them are explained below.

- **Remote procedure call architecture:** This component is used to present in a main program or sub program architecture distributed among multiple computers on a network.
- **Main program or Subprogram architectures:** The main program structure decomposes into number of subprograms or function into a control hierarchy. Main program contains number of subprograms that can invoke other components.



4. Object Oriented architecture:

The components of a system encapsulate data and the operations that must be applied to manipulate the data. The coordination and communication between the components are established via the message passing.

Characteristics of Object-Oriented architecture

- Object protect the system's integrity.
- An object is unaware of the depiction of other items.

Advantage of Object-Oriented architecture

- It enables the designer to separate a challenge into a collection of autonomous objects.
- Other objects are aware of the implementation details of the object, allowing changes to be made without having an impact on other objects.

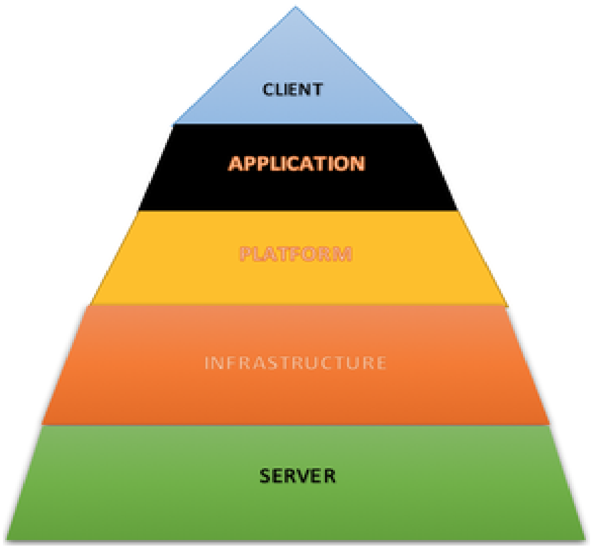
5. Layered architecture:

A number of different layers are defined with each layer performing a well-defined set of operations. Each layer will do some operations that becomes closer to machine instruction set progressively.

At the outer layer, components will receive the user interface operations and at the inner layers, components will perform the operating system interfacing (communication and coordination with OS)

Intermediate layers to utility services and application software functions.

One common example of this architectural style is OSI-ISO (Open Systems Interconnection-International Organization for Standardization) communication system.



Procedural Design:

In software design, Procedural Design (SPD) converts and translates structural elements into procedural explanations. SPD starts straight after data design and architectural design.

Procedural design is used to model programs that have an obvious flow of data from input to output.

It represents the architecture of a program as a set of interacting processes that pass data from one to another.

Procedural design is also called component design. It is completely based on process and control specifications.

The “state transition diagram” of the requirements analysis model is also used in component design.

Component design is usually done after user interface design.

The component-level design depicts the software at a level of abstraction that is very close to the code.

At the component level, the software engineer must represent data structures, interfaces, and algorithms in sufficient detail to guide the generation of programming language source code.

Object-oriented Concepts/Design:

Object oriented design works around the entities and their characteristics instead of functions involved in the software system. This design strategies focuses on entities and its characteristics. The whole concept of software solution revolves around the engaged entities.

Let us see the important concepts of Object-Oriented Design:

Objects

All entities involved in the solution design are known as objects. For example, person, banks, company and customers are treated as objects. Every entity has some attributes associated to it and has some methods to perform on the attributes.

Classes

A class is a generalized description of an object. An object is an instance of a class. Class defines all the attributes, which an object can have and methods, which defines the functionality of the object.

In the solution design, attributes are stored as variables and functionalities are defined by means of methods or procedures.

Encapsulation

In OOD, the attributes (data variables) and methods (operation on the data) are bundled together is called encapsulation. Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world. This is called information hiding.

Inheritance

OOD allows similar classes to stack up in hierarchical manner where the lower or subclasses can import, implement and re-use allowed variables and methods from their immediate super classes. This property of OOD is known as inheritance. This makes it easier to define specific class and to create generalized classes from specific ones.

Polymorphism

OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending upon how the function is invoked, respective portion of the code gets executed.