
# Arba Minch University

# Faculty of Computing and Software Engineering

## Software Architecture and Design (SENG3141)

## Learners' Material

**Written By: Debebe Habte**

MAR, 2015

## Table of Contents

# Chapter One

# Introduction to Software Design

**Unit Objectives**

At the end of this chapter the students will be able to:

- Understand meaning software design.
- Understand objective of software design.
- Understand software design activities.
- Understand the mechanisms and methods of software design principles.

## 1.1 Software Design

Software design is a mechanism to transform user requirements into some suitable form, which helps the programmer in software coding and implementation. 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**,** which moves the concentration from the problem domain to the solution domain. In software design, we consider the system to be a set of components or modules with clearly defined behaviors & boundaries. It involves designing the entire system (network, databases, UI, data structures) and its elements, as well as includes overall architecture of the software.

The software design is like a blueprint of the software. This can be related to the construction of a building. Before constructing any building, a blueprint of that building is created and according to that blueprint, further construction is carried out. For a similar purpose, in the case of software development, we make software design. After the design phase, the software undergoes the coding and implementation phase. In this phase, different developers work on different modules. So, for any kind of reference or any help, they refer to this software design.

While designing any software, there are two different phases: -

1. Preliminary Design
2. Detailed Design
1. Preliminary Design

In the preliminary design, a rough design of the software is proposed by several designers according to the customer requirements. In the preliminary design, various designs of the software

are made, accepted, rejected and modified. After reaching a stage in which the design seems to fulfill all the customer requirements, a set of trained software designers make the final design of the software.

2. Detailed Design

The final design that is made at the end of the preliminary design phase is called the detailed design. After this design is made, no further changes are made in the design of the software, and the design is sent to the development team for further work. The design phase of the software is very crucial in the software development process. This is because if the design of the software is not accurate to fulfill all the requirements of the user, then the software that will be developed will also look those things because the software will follow the design only. So, for making good quality software, the design of the software should also be proper and accurate. So, each design in the preliminary stage must be evaluated in every manner before being finalized as the design for the software.

### 1.1.1 Software Design Levels

Software design yields three levels:

1. **Architectural Design -** The architectural design is the highest abstract version of the system. It identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of proposed solution domain.
2. **High-level Design-** The high-level design breaks the 'single entity-multiple component' concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other. High-level design focuses on how the system along with all of its components can be implemented in forms of modules. It recognizes modular structure of each sub-system and their relation and interaction among each other.
3. **Detailed Design-** Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs. It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules. It includes:
    - Decomposition of major system components into program units.
    - Allocation of functional responsibilities to units.
    - User interfaces, Network architecture
    - Unit states and state changes

- Data and control interaction between units
- Algorithms and data structures

**1.2 Objective of Software Design**

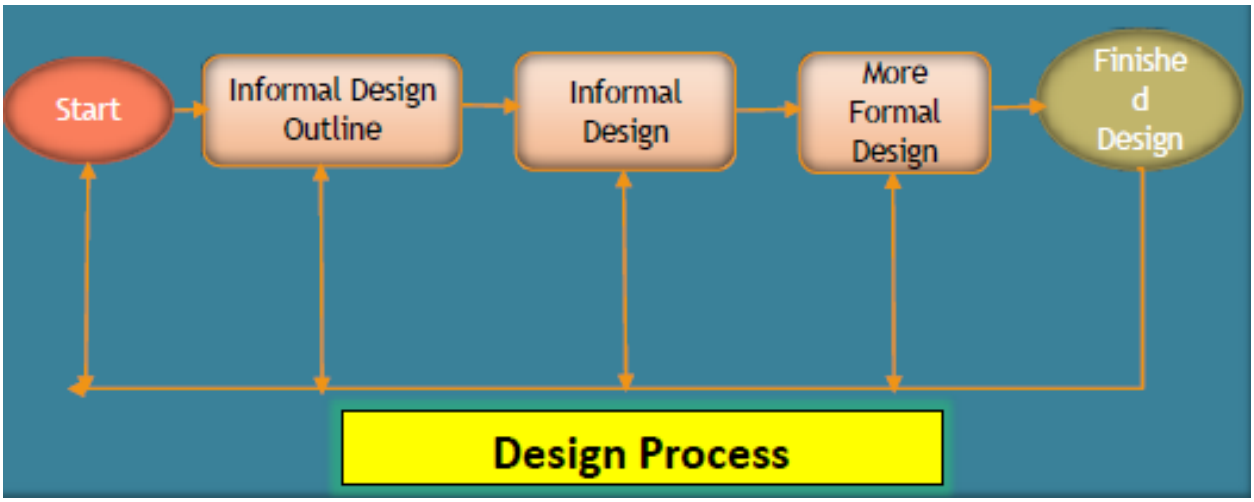
- Correctness: Software design should be correct as per requirement.
- Completeness: The design should have all components like data structures, modules, and external interfaces, etc.
- Efficiency: Resources should be used efficiently by the program.
- Flexibility: Able to modify on changing needs.
-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

**1.3 Stages of Design**

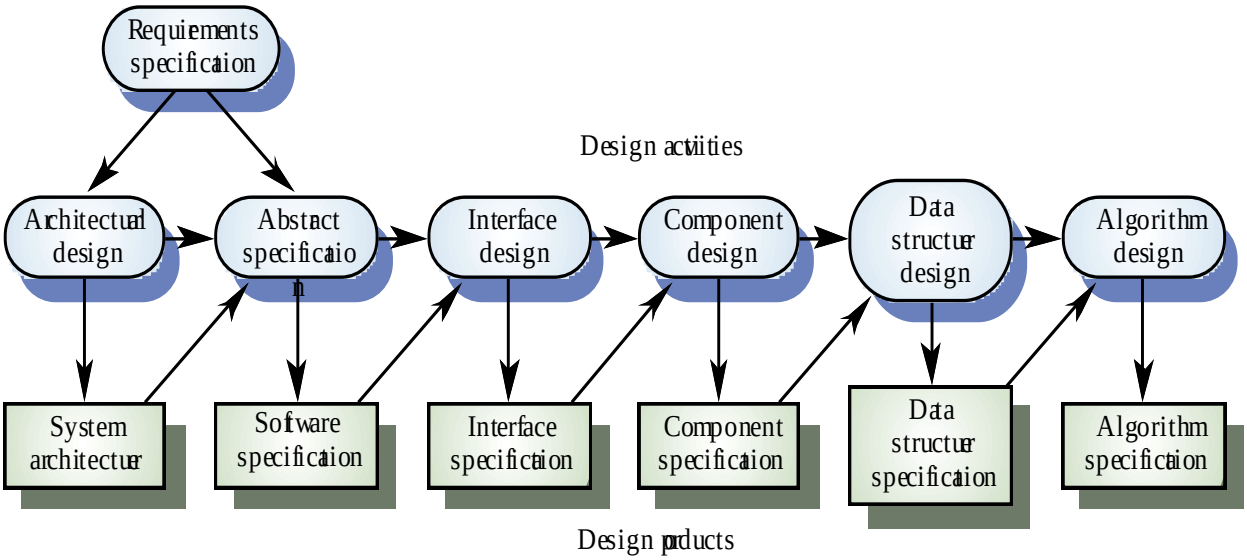
- Problem Understanding
  - Look at the problem from different angles to discover the design requirements.
- Identify one or more solutions
  - Evaluate possible solutions and choose the most appropriate depending on the designer's experience and available resources.
- Describe solution abstractions
  - Use graphical, formal or other descriptive notations to describe the components of the design.
- Repeat the process for each identified abstraction until the design is expressed in primitive terms.

**1.4 Design Process**

Software designers do not arrive at a finished design immediately. So, they develop design iteratively through a number of different versions.  starting point is informal design which is refined by adding information to make it consistent and complete design.




### 1.4.1 Phases in the Design Process




It includes the following software design activities:

- Architectural Design: Identify sub-systems/layouts.
- Abstract Specification: Specify sub-systems.
- Interface Design: Describe sub-system interfaces interaction.
- Component Design: Decompose sub-systems into components.
- Data Structure Design: Design data structures to hold problem data.
- Algorithm Design: Design algorithms for problem functions.

### 1.4.2 Software Design Activities

The fundamental activity that are common to all software process includes 7 steps in software design process activities: -

Step1: Determining the issues

Step2: Conducting extensive research

Step3: Coming up with a possible solution

Step4: Assessing and identifying a real solution

Step5: Developing and evaluating

Step6: Debugging and evaluating

Step7: Make changes to the finished product

### 1. 5 Software Design Concepts

Every software process is characterized by basic concepts along with certain practices or methods. Methods represent the manner through which the concepts are applied. As new technology replaces older technology, many changes occur in the methods that are used to apply the concepts for the development of software. However, the fundamental concepts underlining the software design process remain the same, some of which are described here.

#### Abstraction

Abstraction refers to a powerful design tool, which allows software designers to consider components at an abstract level, while neglecting the implementation details of the components. **IEEE** defines abstraction as 'a view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information.' The concept of abstraction can be used in two ways: as a process and as an entity. As a process, it refers to a mechanism of hiding irrelevant details and representing only the essential features of an item so that one can focus on important things at a time. As an entity, it refers to a model or view of an item.

Each step in the software process is accomplished through various levels of abstraction. At the highest level, an outline of the solution to the problem is presented whereas at the lower levels, the solution to the problem is presented in detail. For example, in the requirements analysis phase, a solution to the problem is presented using the language of problem environment and as we proceed through the software process, the abstraction level reduces and at the lowest level, source code of the software is produced.

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.

**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.

**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.

**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.

**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.

**Patterns**

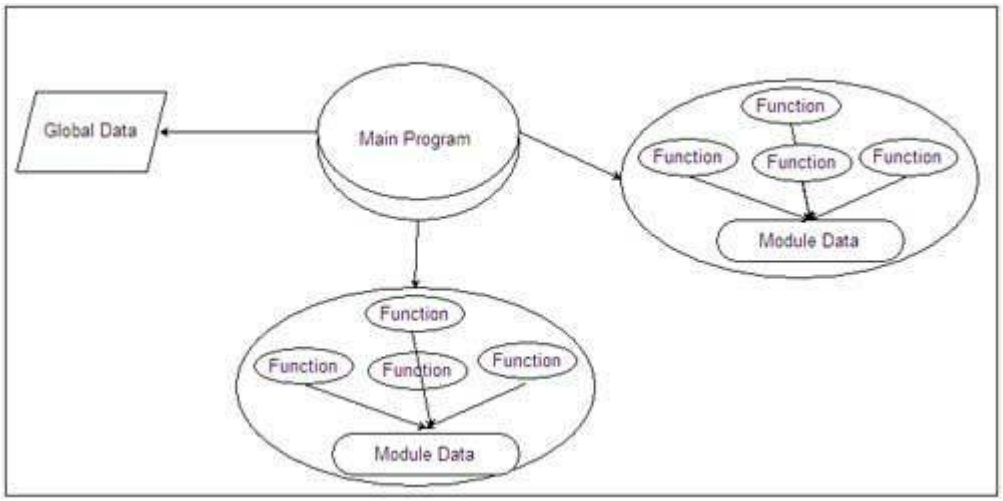
A pattern provides a description of the solution to a recurring design problem of some specific domain in such a way that the solution can be used again and again. The objective of each pattern is to provide an insight to a designer who can determine the following.

- Whether the pattern can be reused

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

**Modularity**

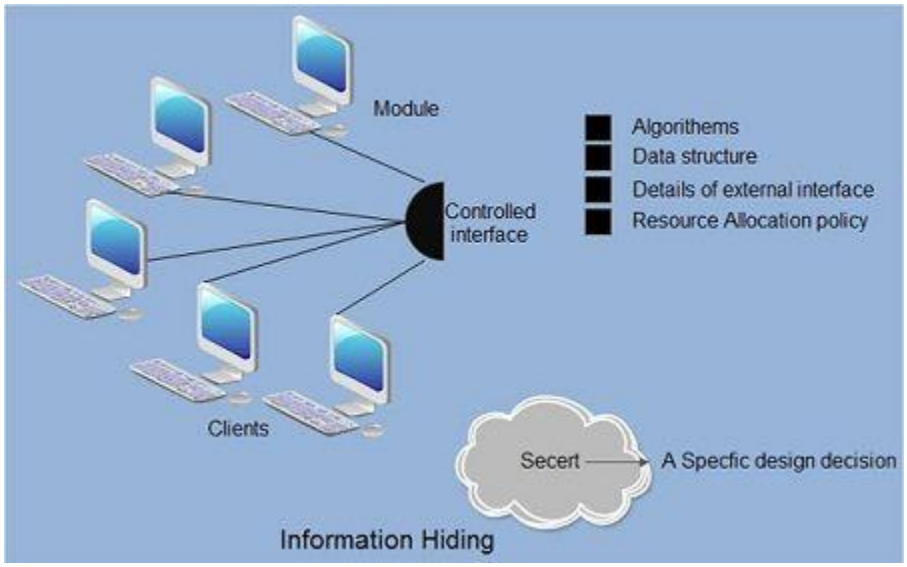
Modularity is achieved by dividing the software into uniquely named and addressable components, which are also known as **modules.** A complex system (large program) is partitioned into a set of discrete modules in such a way that each module can be developed independent of other modules. After developing the modules, they are integrated together to meet the software requirements. Note that larger the number of modules a system is divided into, greater will be the effort required to integrate the modules.




Modularizing a design helps to plan the development in a more effective manner, accommodate changes easily, conduct testing and debugging effectively and efficiently, and conduct maintenance work without adversely affecting the functioning of the software.

**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.

**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.

INPUT

Get user's name (string) through a prompt.

Get user's grade (integer from 0 to 100) through a prompt and validate.

PROCESS

OUTPUT

This is the first step in refinement. The input phase can be refined further as given here.

INPUT

Get user's name through a prompt.

Get user's grade through a prompt.

While (invalid grade)

Ask again:

PROCESS

OUTPUT

**Note:** Stepwise refinement can also be performed for PROCESS and OUTPUT phase.

**Refactoring**

Refactoring is an important design activity that reduces the complexity of module design keeping its behavior 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 (like a component performs four functions that have a limited relationship with one another). 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. Refactoring a software means examining the existing software design for:

- Redundancy
- Unused design elements
- Inefficient or unnecessary algorithms
- Poorly constructed or inappropriate data structures
- And any other design failure, in order to improve design or to get a better design.

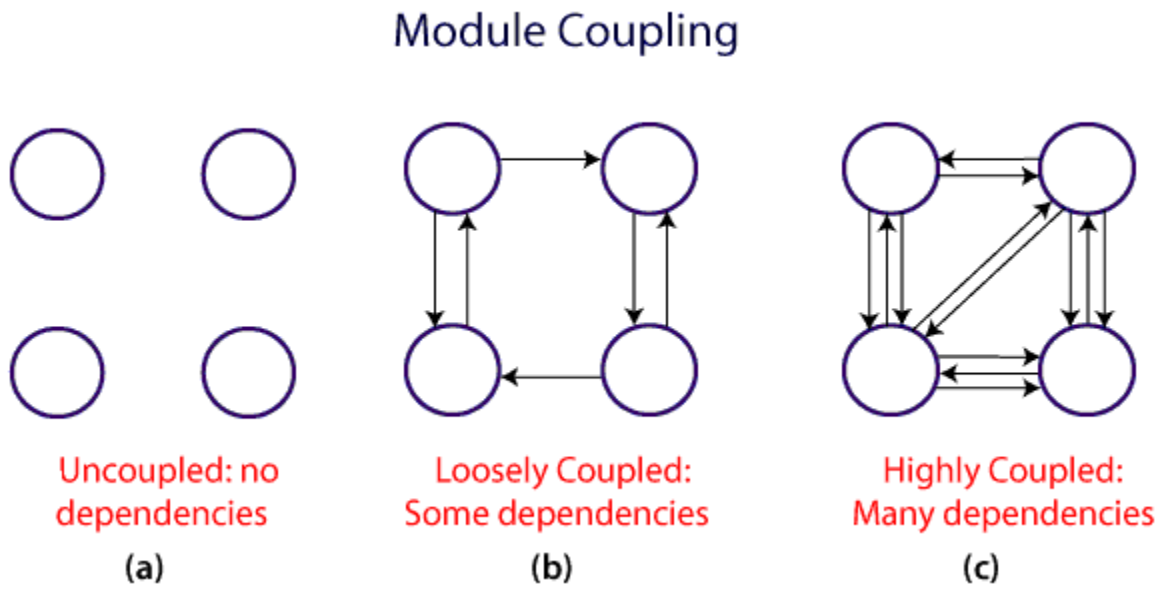
**Functional independence (Coupling and Cohesion)**

Module Coupling

In software engineering, coupling is the degree of interdependence between software modules. Two modules that are tightly coupled are strongly dependent on each other. However, two modules that are loosely coupled are not dependent on each other. Uncoupled modules have no interdependence at all within them. Two components can be dependent in many ways:

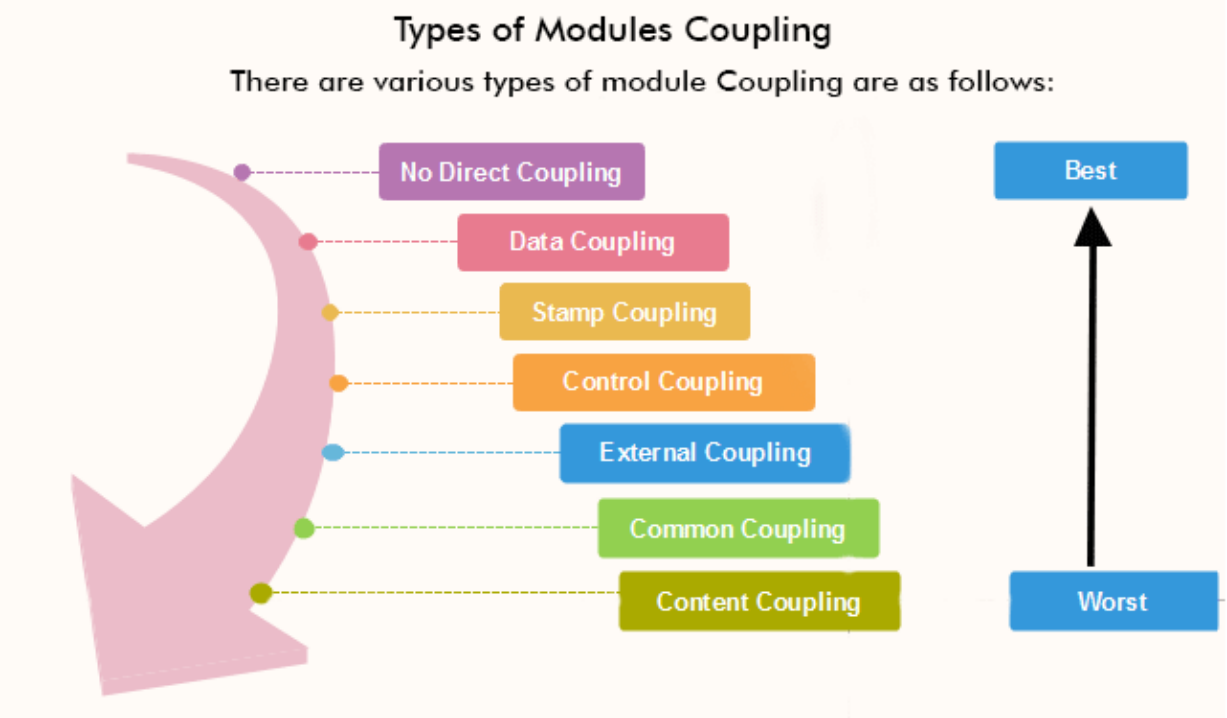
- References made from one to another
- Amount of data passed from one to another
- Amount of control one has over the other
- The complexity of the interface
- The type of information flow between modules

The various types of coupling techniques are shown in fig:

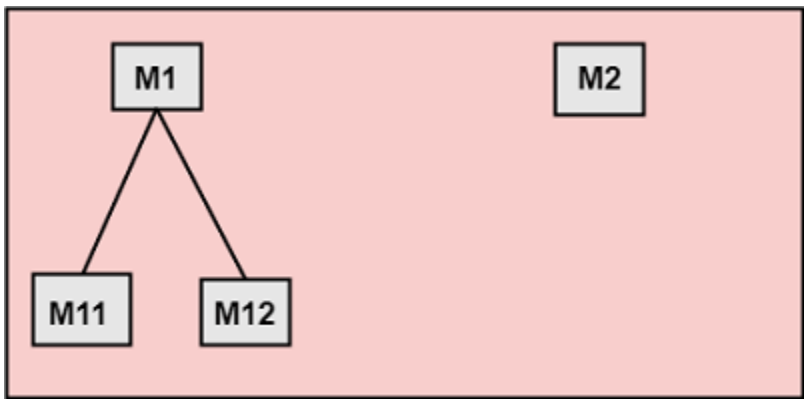



A good design is the one that has low coupling. Coupling is measured by the number of relations between the modules. That is, the coupling increases as the number of calls between modules increase or the amount of shared data is large. Thus, it can be said that a design with high coupling will have more errors.

Types of Module Coupling

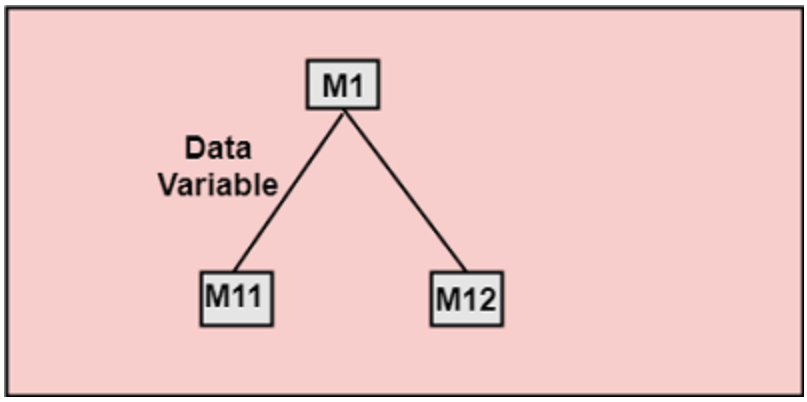



**1. No Direct Coupling:** There is no direct coupling between M1 and M2.




In this case, modules are subordinates to different modules. Therefore, no direct coupling.

**2. Data Coupling:** When data of one module is passed to another module, this is called data coupling.

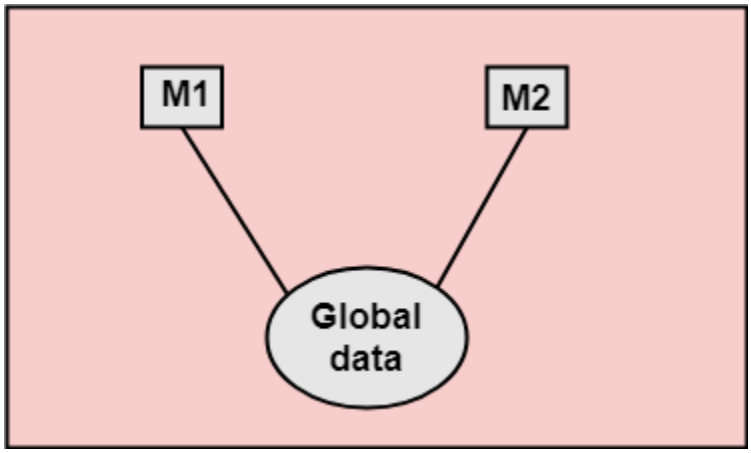



**3. Stamp Coupling:** Two modules are stamp coupled if they communicate using composite data items such as structure, objects, etc. When the module passes non-global data structure or entire structure to another module, they are said to be stamp coupled. For example, passing structure variable in C or object in C++ language to a module.

**4. Control Coupling:** Control Coupling exists among two modules if data from one module is used to direct the structure of instruction execution in another.

**5. External Coupling:** External Coupling arises when two modules share an externally imposed data format, communication protocols, or device interface. This is related to communication to external tools and devices.

**6. Common Coupling:** Two modules are common coupled if they share information through some global data items.

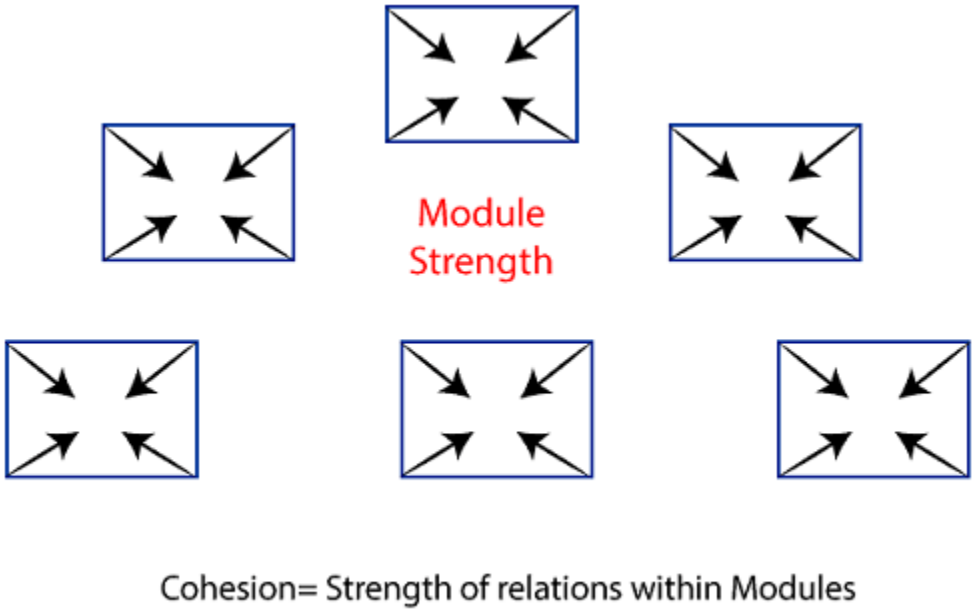



**7. Content Coupling:** Content Coupling exists among two modules if they share code, e.g., a branch from one module into another module.

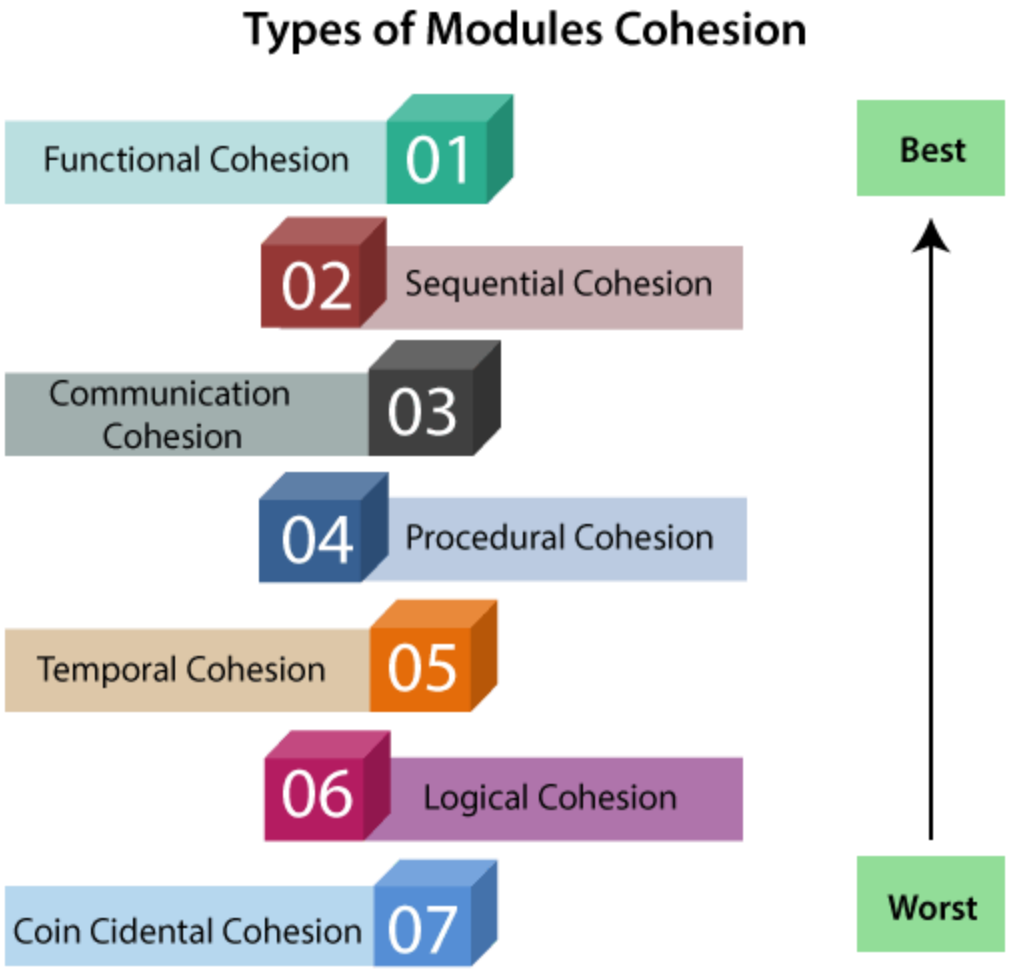
Module Cohesion

In computer programming, cohesion defines to the degree to which the elements of a module belong together. Thus, cohesion measures the strength of relationships between pieces of functionality within a given module. For example, in highly cohesive systems, functionality is strongly related.

Cohesion is an ordinal type of measurement and is generally described as "high cohesion" or "low cohesion."




Types of Modules Cohesion

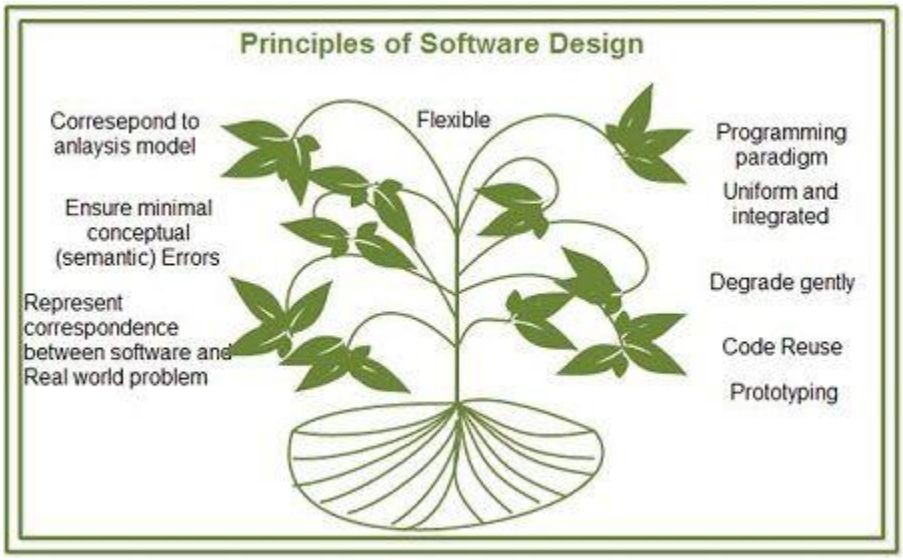



1. **Functional Cohesion:** Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.
2. **Sequential Cohesion:** A module is said to possess sequential cohesion if the element of a module forms the components of the sequence, where the output from one component of the sequence is input to the next.
3. **Communicational Cohesion:** A module is said to have communicational cohesion, if all tasks of the module refer to or update the same data structure, e.g., the set of functions defined on an array or a stack.
4. **Procedural Cohesion:** A module is said to be procedural cohesion if the set of purpose of the module are all parts of a procedure in which particular sequence of steps has to be carried out for achieving a goal, e.g., the algorithm for decoding a message.

5. **Temporal Cohesion:** When a module includes functions that are associated by the fact that all the methods must be executed in the same time, the module is said to exhibit temporal cohesion.
6. **Logical Cohesion:** A module is said to be logically cohesive if all the elements of the module perform a similar operation. For example, Error handling, data input and data output, etc.
7. **Coincidental Cohesion:** A module is said to have coincidental cohesion if it performs a set of tasks that are associated with each other very loosely, if at all.

### 1.6 Principles of Software Design

Developing design is a cumbersome process as most expansive errors are often introduced in this phase. Moreover, if these errors get unnoticed till later phases, it becomes more difficult to correct them. Therefore, a number of principles are followed while designing the software. These principles act as a framework for the designers to follow a good design practice.




Some of the commonly followed design principles are as following.

1. **Software design should correspond to the analysis model:** Often a design element corresponds to many requirements, therefore, we must know how the design model satisfies all the requirements represented by the analysis model.
2. **Choose the right programming paradigm:** A programming paradigm describes the structure of the software system. Depending on the nature and type of application, different programming paradigms such as procedure oriented, object-oriented, and prototyping

paradigms can be used. The paradigm should be chosen keeping constraints in mind such as time, availability of resources and nature of user's requirements.

3. **Software design should be uniform and integrated:** Software design is considered uniform and integrated, if the interfaces are properly defined among the design components. For this, rules, format, and styles are established before the design team starts designing the software.
4. **Software design should be flexible:** Software design should be flexible enough to adapt changes easily. To achieve the flexibility, the basic design concepts such as abstraction, refinement, and modularity should be applied effectively.
5. **Software design should ensure minimal conceptual (semantic) errors:** The design team must ensure that major conceptual errors of design such as ambiguousness and inconsistency are addressed in advance before dealing with the syntactical errors present in the design model.
6. **Software design should be structured to degrade gently:** Software should be designed to handle unusual changes and circumstances, and if the need arises for termination, it must do so in a proper manner so that functionality of the software is not affected.
7. **Software design should represent correspondence between the software and real-world problem:** The software design should be structured in such a way that it always relates with the real-world problem.
8. **Software reuse:** Software engineers believe on the phrase: 'do not reinvent the wheel'. Therefore, software components should be designed in such a way that they can be effectively reused to increase the productivity.
9. **Designing for testability:** A common practice that has been followed is to keep the testing phase separate from the design and implementation phases. That is, first the software is developed (designed and implemented) and then handed over to the testers who subsequently determine whether the software is fit for distribution and subsequent use by the customer. However, it has become apparent that the process of separating testing is seriously flawed, as if any type of design or implementation errors are found after implementation, then the entire or a substantial part of the software requires to be redone. Thus, the test engineers should be involved from the initial stages. For example, they should be involved with analysts to prepare tests for determining whether the user requirements are being met.
10. **Prototyping:** Prototyping should be used when the requirements are not completely defined in the beginning. The user interacts with the developer to expand and refine the requirements

as the development proceeds. Using prototyping, a quick ‘mock-up’ of the system can be developed. This mock-up can be used as a effective means to give the users a feel of what the system will look like and demonstrate functions that will be included in the developed system. Prototyping also helps in reducing risks of designing software that is not in accordance with the customer’s requirements.

Note that design principles are often constrained by the existing hardware configuration, the implementation language, the existing file and data structures, and the existing organizational practices. Also, the evolution of each software design should be meticulously designed for future evaluations, references and maintenance.

**SOLID Principles of Software Architecture:**

Each character of the word SOLID defines one principle of software architecture. This SOLID principle is followed to avoid product strategy mistakes. A software architecture must adhere to SOLID principle to avoid any architectural or developmental failure.

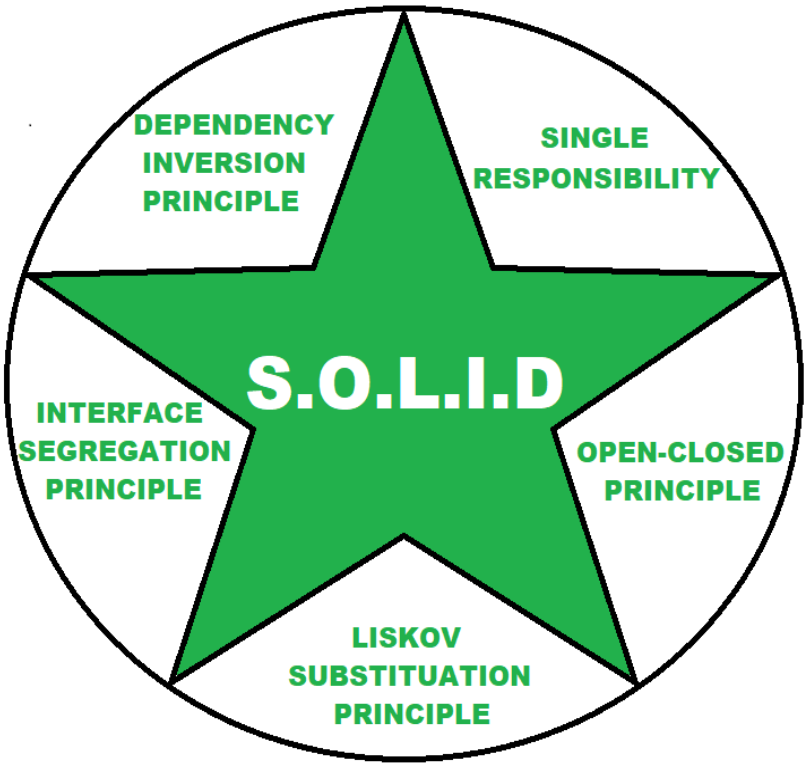



Fig: S.O.L.I.D Principle

1. **Single Responsibility:** Each service should have a single objective.
2. **Open-Closed Principle:** Software modules should be independent and expandable.
3. **Liskov Substitution Principle:** Independent services should be able to communicate and substitute each other.
4. **Interface Segregation Principle:** Software should be divided into such microservices there should not be any redundancies.
5. **Dependency Inversion Principle:** Higher-levels modules should not be depending on low-lower-level modules and changes in higher level will not affect to lower level.

## 1.7 User Interface Design

User interface design plays an important role in determining how people interact with a software product. Because, UI design has to do with how different visual elements, such as colors, typography, and images, work together to create a seamless user experience. The visual part of a computer application or operating system through which a client interacts with a computer or software. It determines how commands are given to the computer or the program and how data is displayed on the screen.

There are three main types of UI design

**Graphical user interfaces (GUI)**: Visual interfaces in computers and handheld devices, such as websites and app screens. GUI relies much more heavily on the mouse. A typical example of this type of interface is any versions of the Windows operating systems.

**Voice-controlled interfaces**: These interfaces require sounds, specifically voices, to initiate action. A classic example of this type of interface is Apple's Siri.

**Gesture-based interfaces**: These interfaces are motion-activated and used for applications such as virtual reality games.

### SELF-CHECK

1. Who designs and implement database structures?
   A. Programmers
   B. Project managers
   C. Technical writers
   D. Database administrator
2. In design phase, which is the primary area of concern?
   A. Architecture
   B. Data
   C. Interface
   D. All of the mentioned

3. The importance of software design can be summarized in a single word which is:
   A. Efficiency
   B. Accuracy
   C. Quality
   D. Complexity
4. Cohesion is a qualitative indication of the degree to which a module:
   A. Can be written more compactly
   B. Focuses on just one thing
   C. Is able to complete its function in a timely manner
   D. Is connected to other modules and the outside world
5. Coupling is a qualitative indication of the degree to which a module:
   A. Can be written more compactly
   B. Focuses on just one thing
   C. Is able to complete its function in a timely manner
   D. Is connected to other modules and the outside world
6. Which one of the following is **not true** about software design?
   A. It about where to place a component
   B. Iterative process
   C. Creative stage in software development
   D. It is about how to build a system
7. The inter dependency between modules in a system is:
   A. Cohesion
   B. Modularization
   C. Abstraction
   D. Coupling
8. A type of cohesion that has related elements by a reference to the same input or output data.
   A. Sequential
   B. Functional
   C. Communicational
   D. Procedural
9. Which one of the following is **not true** about information hiding?
   A. Emphasizes communication through controlled interfaces
   B. Limits the global impact of local design decisions
   C. Leads to encapsulation
   D. Side-effects due to module changes minimized
10. A design model which provides the detailed description of how structural elements of software will actually be implemented is___?

A. Component design
B. Data design
C. Architectural design
D. Interface design

**EXERCISE**

1. What are the fundamental activities of a software process?
2. Discuss about the principles for good software design?
3. What is software refactoring and why software refactoring is used in software design?
4. Differentiate between abstraction and information hiding.

# Chapter Two

# Design Patterns

**Unit Objectives**

At the end of this chapter the students will be able to:

- Understand meaning of design patterns
- Understand elements of design patterns
- Differentiate types of design patterns
- Apply the different types of design patterns in software development

## 2.1 Introduction

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it. Experienced object-oriented designers will tell you that a reusable and flexible design is difficult if not impossible to get "right" the first time. Before a design is finished, they usually try to reuse it several times, modifying it each time.

Yet experienced object-oriented designers do make good designs. Meanwhile new designers are overwhelmed by the options available and tend to fall back on nonobject-oriented techniques they've used before. It takes a long time for novices to learn what good object-oriented design is all about. Experienced designers evidently know something inexperienced ones don't. What is it? One thing expert designer know not to do is solve every problem from first principles. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, they use it again and again. Such experience is part of what makes them experts.

Consequently, you'll find recurring patterns of classes and communicating objects in many object-oriented systems. These patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience. A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them. An analogy will help

illustrate the point. Novelists and playwrights rarely design their plots from scratch. Instead, they follow patterns like "Tragically Flawed Hero" (Macbeth, Hamlet, etc.) or "The Romantic Novel" (countless romance novels). In the same way, object-oriented designers follow patterns like "represent states with objects" and "decorate objects so you can easily add/remove features." Once you know the pattern, a lot of design decisions follow automatically. We all know the value of design experience. How many times have you had design? —that feeling that you've solved a problem before but not knowing exactly where or how? If you could remember the details of the previous problem and how you solved it, then you could reuse the experience instead of rediscovering it. However, we don't do a good job of recording experience in software design for others to use. The purpose of this chapter is to record experience in designing object-oriented software as design patterns.

**2.2 Design Pattern**

In software engineering, a software design pattern is a general, reusable solution of how to solve a common problem when designing an application or system. Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [AIS+77, page x]. Even though Alexander was talking about patterns in buildings and towns, what he says is true about object-oriented design patterns. Our solutions are expressed in terms of objects and interfaces instead of walls and doors, but at the core of both kinds of patterns is a solution to a problem in a context.

Design patterns are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code. You can’t just find a pattern and copy it into your program, the way you can with off-the-shelf functions or libraries. The pattern is not a specific piece of code, but a general concept for solving a particular problem. You can follow the pattern details and implement a solution that suits the realities of your own program. Unlike a library or framework, which can be inserted and used right away, a design pattern is more of a template to approach the problem at hand.

The pattern typically shows relationships and interactions between classes or objects. The idea is to speed up the development process by providing well-tested, proven development/design paradigms. Design patterns are programming language independent strategies for solving a

common problem. That means a design pattern represents an idea, not a particular implementation. By using design patterns, you can make your code more flexible, reusable, and maintainable. It's not mandatory to always implement design patterns in your project. Design patterns are not meant for project development. Design patterns are meant for common problem-solving. Whenever there is a need, you have to implement a suitable pattern to avoid such problems in the future. To find out which pattern to use, you just have to try to understand the design patterns and their purposes. Only by doing that, you will be able to pick the right one. Each pattern is a three-part rule which expresses a relation between a certain context, a problem, and a solution. **i.e., Patterns = Problems + Solution pairs in a context**

Design patterns are granular and applied at different levels such as frameworks, subsystems, and sub-subsystems**.** Its expert solutions to recurring/ commonly occurring problems in a context and thus have been captured at many levels of abstraction and in numerous domains

Design patterns are used to support object-oriented programming (OOP), a paradigm that is based on the concepts of both objects (instances of a class; data with unique attributes) and classes (user-defined types of data). Or used to represent some of the best practices adapted by experienced object-oriented software developers. A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems.

**Goal:** Understand the purpose and usage of each design pattern in order to pick and implement the correct pattern as needed.

### 2.3 Elements of Design Pattern

In general, a pattern has four essential elements:

1. **The pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary. It lets us design at a higher level of abstraction. Having a vocabulary for patterns lets us talk about them with our colleagues, in our documentation, and even to ourselves. It makes it easier to think about designs and to communicate them and their trade-offs to others. Finding good names has been one of the hardest parts of developing our catalog. Example Adapter, Bridge, Flyweight, Private Class Data, and the like.
2. **The problem** describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design.

Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.

3. **The solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.
4. **The consequences** are the results and trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern. The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them.

**What does the pattern consist of?**

We describe design patterns using a consistent format. Each pattern is divided into sections according to the following template. The template lends a uniform structure to the information, making design patterns easier to learn, compare, and use. Here are the sections that are usually present in a pattern description:

**Intent** of the pattern briefly describes both the problem and the solution.

**Motivation** further explains the problem and the solution the pattern makes possible.

**Structure** of classes shows each part of the pattern and how they are related.

**Code example** in one of the popular programming languages makes it easier to grasp the idea behind the pattern.

**Pattern Name and Classification:** The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary.

**Applicability:** What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

**Participants:** The classes and/or objects participating in the design pattern and their responsibilities.

## 2.4 Usage of Design Pattern

Design Patterns have two main usages in software development.

1. Create common platform for developers

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

2. Best Practices

Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps unexperienced developers to learn software design in an easy and faster way.

## 2.5 Classification of Patterns

There are 23 classic design patterns, although there are at least 26 design patterns discovered to date. These design patterns gained popularity after the publication of Design Patterns: Elements of Reusable Object-Oriented Software, a 1994 book published by the “Gang of Four” (GoF): Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Due to this, the 23 design patterns are often known as the gang of four design patterns.

We classify design patterns by two criteria. The first criterion, called **purpose,** level of detail, complexity, and scale of applicability to the **entire system** being designed, design patterns can be broken down into three types, creational design patterns, structural design patterns, and behavioral design patterns**.**

Creational patterns concern the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility. The second criterion, called **scope,** specifies whether the pattern applies primarily to classes or to objects. Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static fixed at compile-time. Object patterns deal with object relationships, which can be changed at run-time and are more dynamic. Almost all patterns use inheritance to some extent. So, the only patterns labeled "class patterns" are those that focus on class relationships. Note that most patterns are in the object scope. Creational class patterns defer some part of object creation to subclasses,

while Creational object patterns defer it to another object. The Structural class patterns use inheritance to compose classes, while the Structural object patterns describe ways to assemble objects. The Behavioral class patterns use inheritance to describe algorithms and flow of control, whereas the Behavioral object patterns describe how a group of objects cooperate to perform a task that no single object can carry out alone.

Table 1. **Type of design patterns**

| | | Purpose | | |
|---|---|---|---|---|
| | | Creational | Structural | Behavioral |
| Scope | Class | Factory Method | Adapter (class) | Interpreter<br>Template Method |
| | Object | Abstract Factory<br>Builder<br>Prototype<br>Singleton | Adapter (object)<br>Bridge<br>Composite<br>Decorator<br>Facade<br>Flyweight<br>Proxy | Chain of Responsibility<br>Command<br>Iterator<br>Mediator<br>Memento<br>Observer<br>State<br>Strategy<br>Visitor |

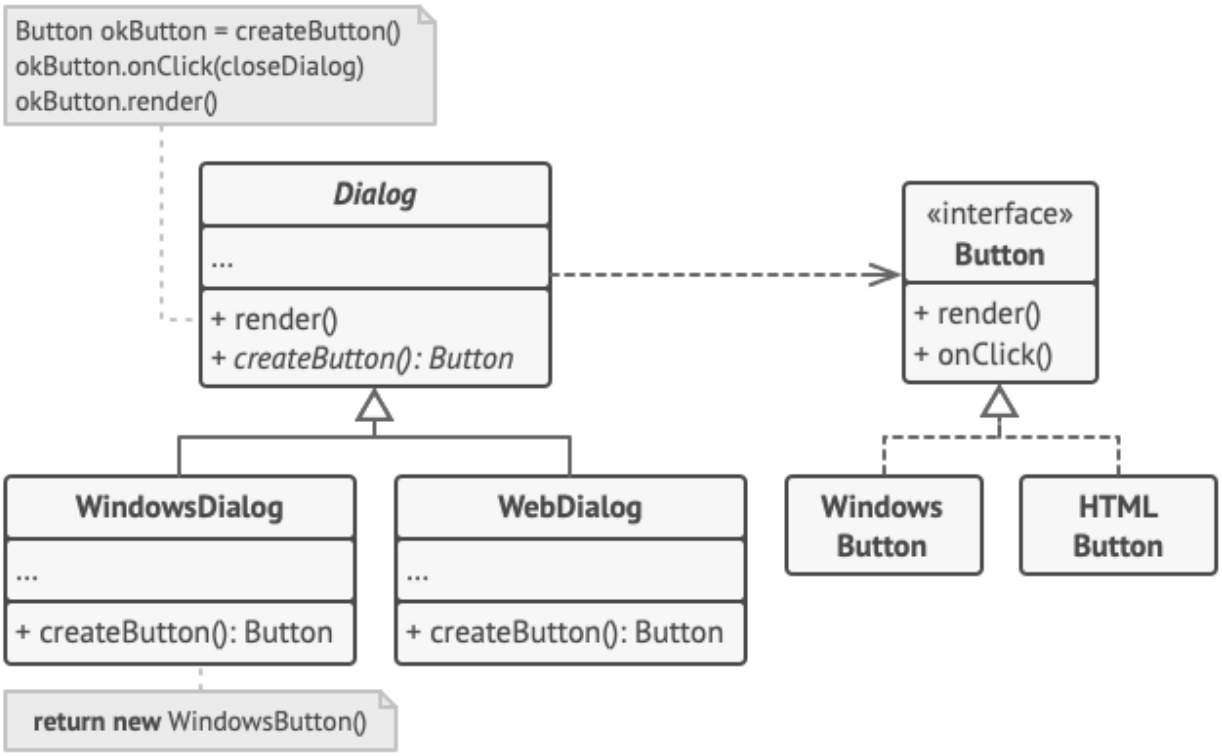
### 2.5.1 Creational Design Patterns

A creational design pattern deals with object creation and initialization, providing guidance about which objects are created for a given situation. These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. Creational design patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented. A class creational pattern uses inheritance to vary the class that's instantiated, whereas an object creational pattern will delegate instantiation to another object.

Creational patterns become important as systems evolve to depend more on object composition than class inheritance. As that happens, emphasis shifts away from hardcoding a fixed set of behaviors toward defining a smaller set of fundamental behaviors that can be composed into any number of more complex ones. Thus, creating objects with particular behaviors requires more than

simply instantiating a class. There are two recurring themes in these patterns. First, they all encapsulate knowledge about which concrete classes the system uses. Second, they hide how instances of these classes are created and put together. All the system at large knows about the objects is their interfaces as defined by abstract classes. Consequently, the creational patterns give you a lot of flexibility in what gets created, who creates it, how it gets created, and increase code reusability. Creational design patterns are the Factory Method, Builder, Singleton, Prototype, Abstract Factory, and Object Pool. Let's discuss the first 4.

**Factory Method**: Creates objects with a common interface and lets a class defer instantiation to subclasses. it is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. This example illustrates how the **Factory Method** can be used for creating cross-platform UI elements without coupling the client code to concrete UI classes.



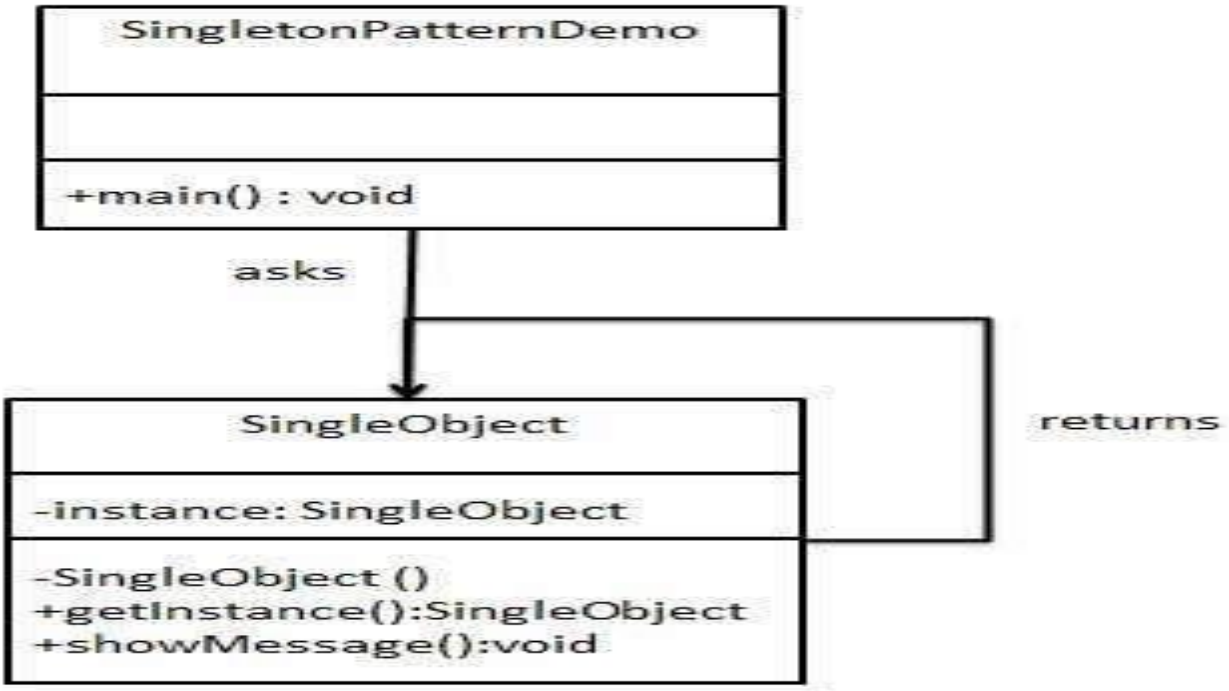

**Singleton**: This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object. The pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way

to access its only object which can be accessed directly without need to instantiate the object of the class and restricts object creation for a class to only one instance. All implementations of the Singleton have these two steps in common:

- Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.
- Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.

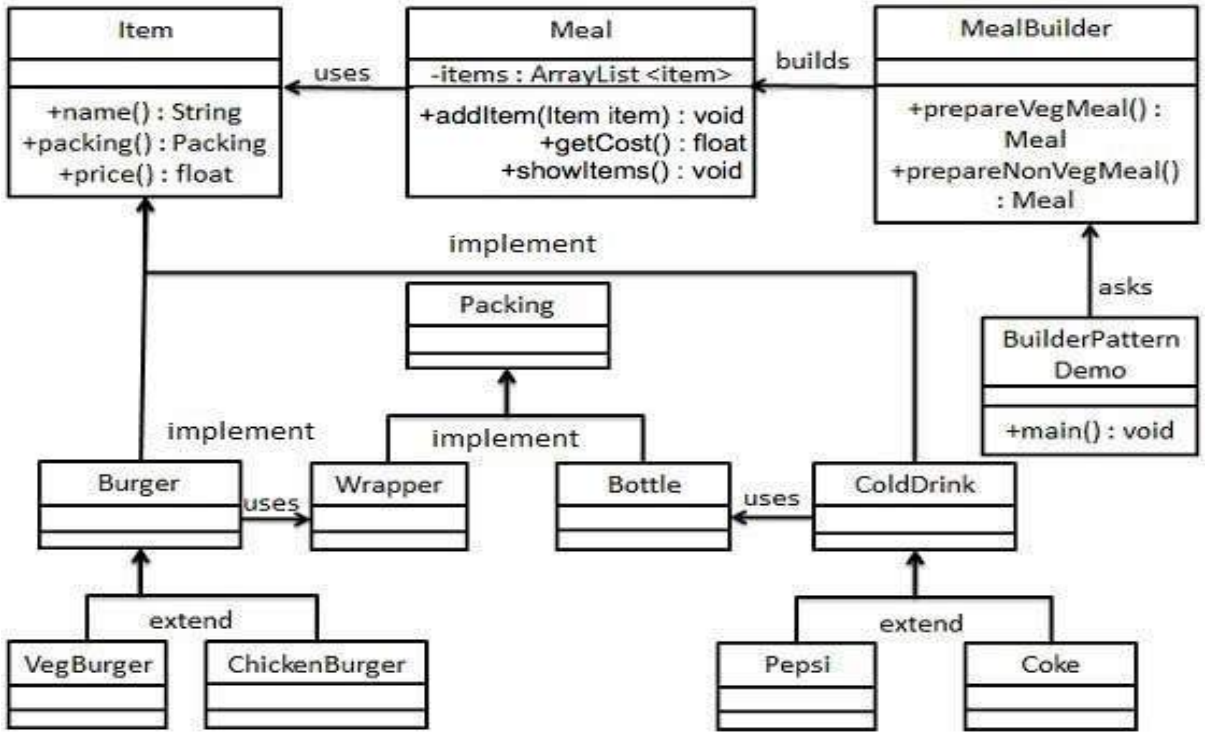
If your code has access to the Singleton class, then it's able to call the Singleton's static method. So, whenever that method is called, the same object is always returned.

Example: SingleObject class provides a static method to get its static instance to outside world. SingletonPatternDemo, our demo class will use SingleObject class to get a SingleObject object.

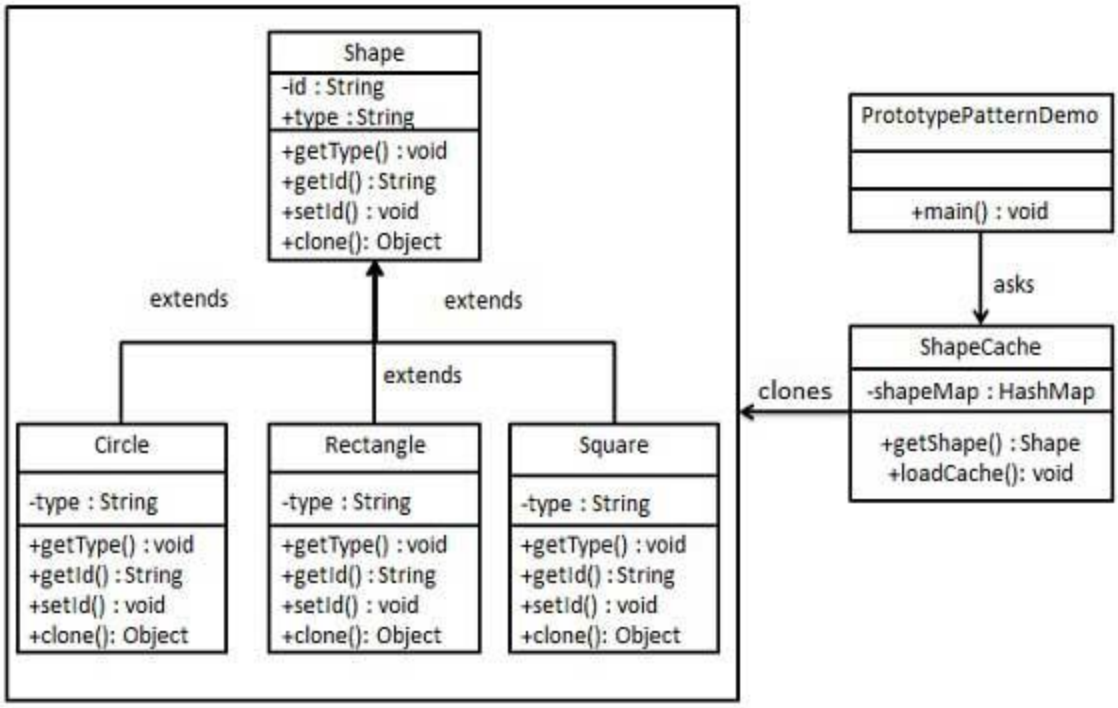



**Builder Pattern:** Builds a complex object using simple objects using a step-by-step approach and the final step will return the object. The process of constructing an object should be generic so that it can be used to create different representations of the same object. Builder pattern aims to "Separate the construction of a complex object from its representation so that the same construction process can create different representations." Builder design pattern also helps in

minimizing the number of parameters in the constructor and thus there is no need to pass in null for optional parameters to the constructor. The parameters of the constructor are reduced and are provided in highly readable method calls. and object is always instantiated in a complete state. Example: Immutable objects can be built without much complex logic in the object-building process.




**Prototype Pattern:** refers to creating duplicate object while keeping performance in mind. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object. The pattern involves implementing a prototype interface which tells to create a clone of the current object. Prototype pattern is used when creation of object directly is costly. For example, an object is to be created after a costly database operation. We can cache the object, returns its clone on next request and update the database as and when needed thus reducing database calls. Example: We're going to create an abstract class Shape and concrete classes extending the Shape class. A class ShapeCache is defined as a next step which stores shape objects in a Hashtable and returns their clone when requested. PrototypPatternDemo, our demo class will use ShapeCache class to get a Shape object.



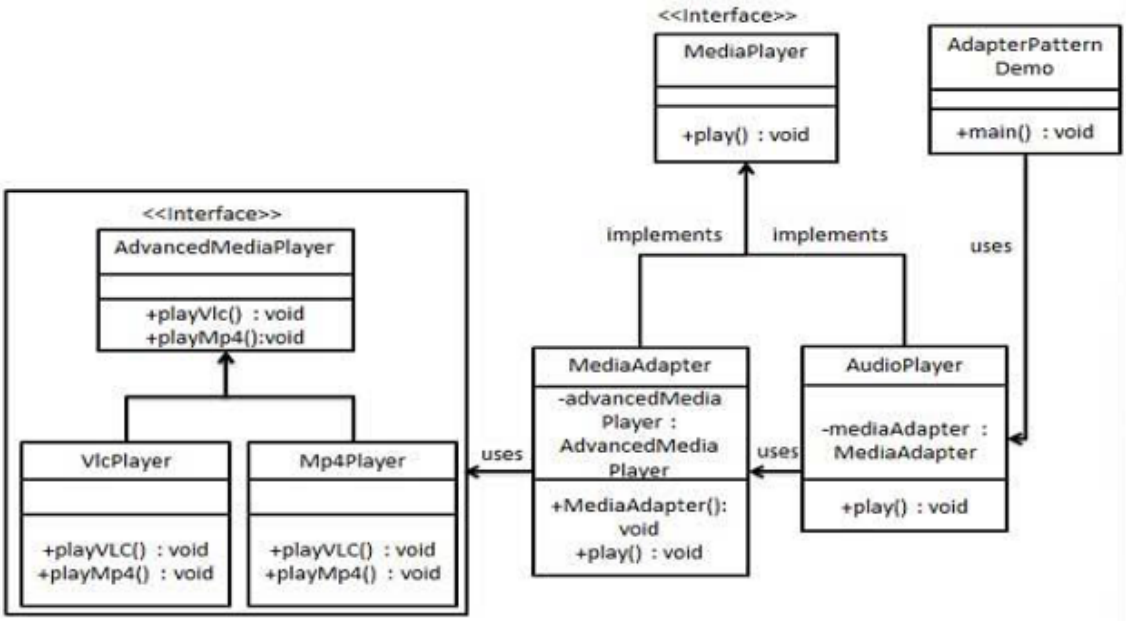

### 2.5.2 Structural Design Patterns

A structural design pattern deals with class and object composition, or how to assemble objects and classes into larger structures. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities. Structural design patterns are Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data, and Proxy. Let's discuss the first 4.

**Adapter**: Adapter pattern works as a connector between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces. Adapter pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces. For example, consider a USB to Ethernet adapter. We need this when we have an Ethernet interface on one end and USB on the other. Since they are incompatible with each other. we use an adapter that converts one to other. This example is pretty analogous to Object Oriented Adapters. In design, adapters are used when we have a class (Client) expecting some type of object and we have an object (Adaptee) offering the same features but exposing a different interface.

To use an adapter:

- The client makes a request to the adapter by calling a method on it using the target interface.
- The adapter translates that request on the adaptee using the adaptee interface.
- Client receives the results of the call and is unaware of adapter's presence.
- Example:
  - We have a MediaPlayer interface and a concrete class Audio Player implementing the Media Player interface.
  - Audio Player can play mp3 format audio files by default.
  - We are having another interface AdvancedMediaPlayer and concrete classes implementing the AdvancedMediaPlayer interface.
  - These classes can play vlc and mp4 format files.
  - We want to make AudioPlayer to play other formats as well.
  - To attain this, we have created an adapter class MediaAdapter which implements the MediaPlayer interface and uses AdvancedMediaPlayer objects to play the required format.
  - AudioPlayer uses the adapter class MediaAdapter passing it the desired audio type without knowing the actual class which can play the desired format.
  - AdapterPatternDemo, our demo class, will use AudioPlayer class to play various formats.

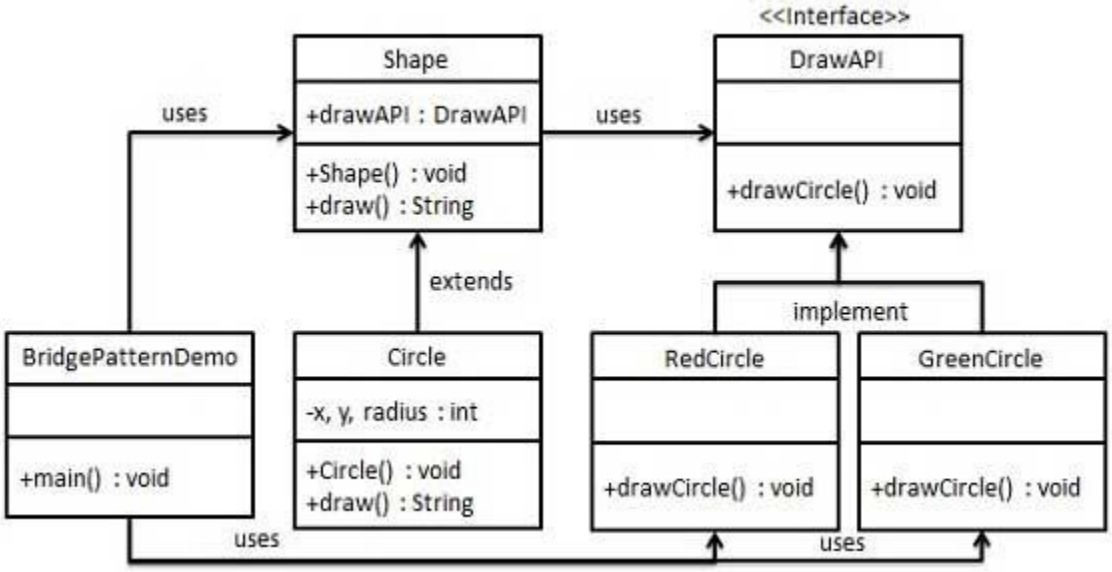



**Facade Design:** The facade design pattern is a "structural" design pattern that helps provide one interface (class) for access to a large body of code / various objects. A facade hides complexities of various sub-systems (often organized into a class) with a simple interface. For example, an E - commerce customer only wants one point of interacting with a brand, rather than individually communicating (interfacing) with each system to support the sale such as product inventory, authentication, security, payment processing, order fulfillment, etc. In this case, the facade has encapsulated all the "order" activities and systems to provide a single interface, the customer remains completely unaware of what's going on behind the scenes.

**Bridge**: A method to decouple an interface from its implementation. Bridge is used when we need to decouple an abstraction from its implementation so that the two can vary independently. This type of design pattern comes under structural pattern as this pattern decouples implementation class and abstract class by providing a bridge structure between them. The pattern involves an interface which acts as a bridge which makes the functionality of concrete classes independent from interface implementer classes. Both types of classes can be altered structurally without affecting each other.

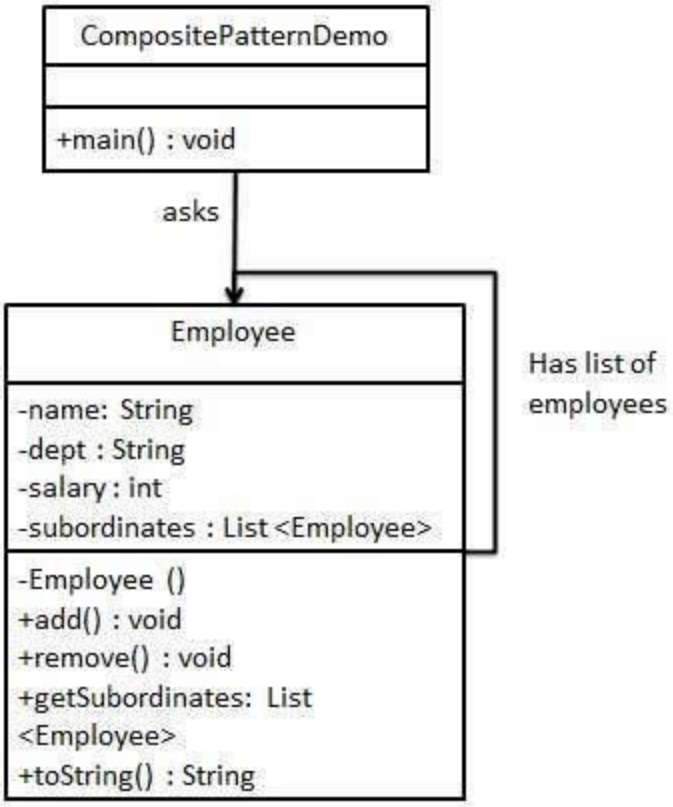
We are demonstrating use of Bridge pattern via following example in which a circle can be drawn in different colors using same abstract class method but different bridge implementer classes. We have a DrawAPI interface which is acting as a bridge implementer and concrete

classes RedCircle, GreenCircle implementing the DrawAPI interface. Shape is an abstract class and will use object of DrawAPI. BridgePatternDemo, our demo class will use Shape class to draw different colored circle.




**Composite**: Leverages a tree structure to support manipulation as one object. Composite pattern is used where we need to treat a group of objects in similar way as a single object. Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchy. The pattern creates a class that contains group of its own objects. This class provides ways to modify its group of same objects.

The following example show employees hierarchy of an organization. We have a class Employee which acts as composite pattern actor class. CompositePatternDemo, our demo class will use Employee class to add department level hierarchy and print all employees.



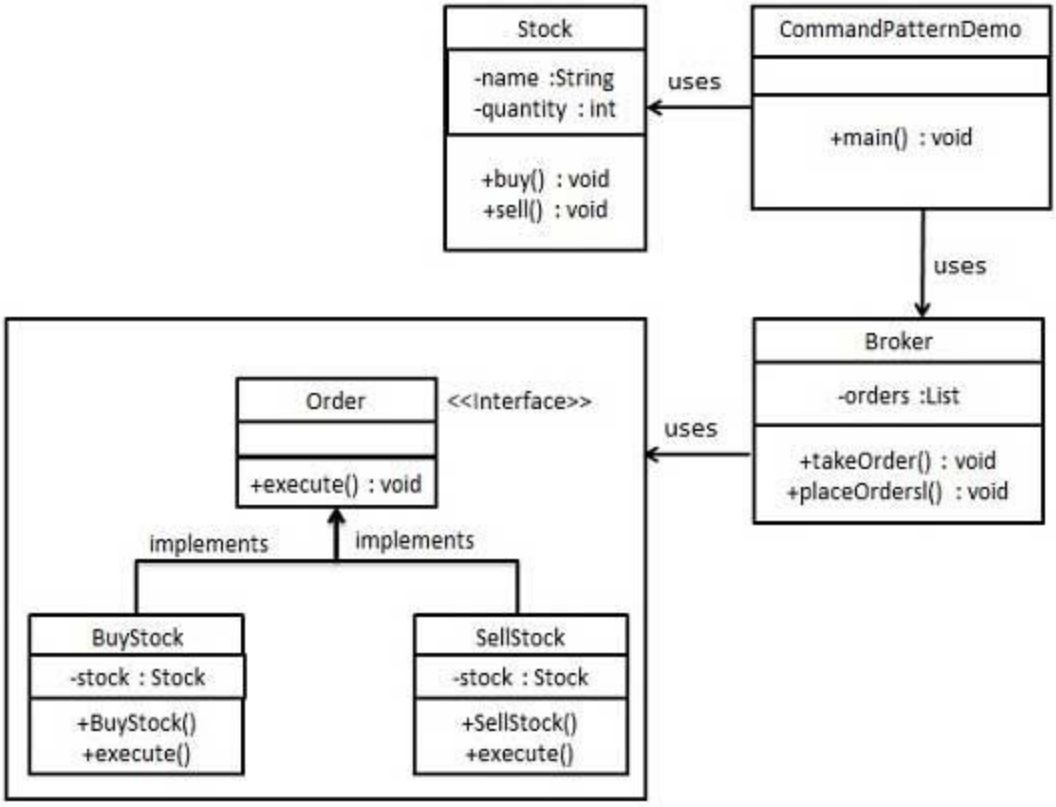

### 2.5.3. Behavioral Design Patterns

Behavioral patterns are about identifying common communication patterns between objects and realizing these patterns or how responsibilities are assigned between objects. Behavioral patterns are Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template method, Visitor. Let's discuss the first 5.

**Command Pattern** is a data driven design pattern and falls under behavioral pattern category. A request is wrapped under an object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

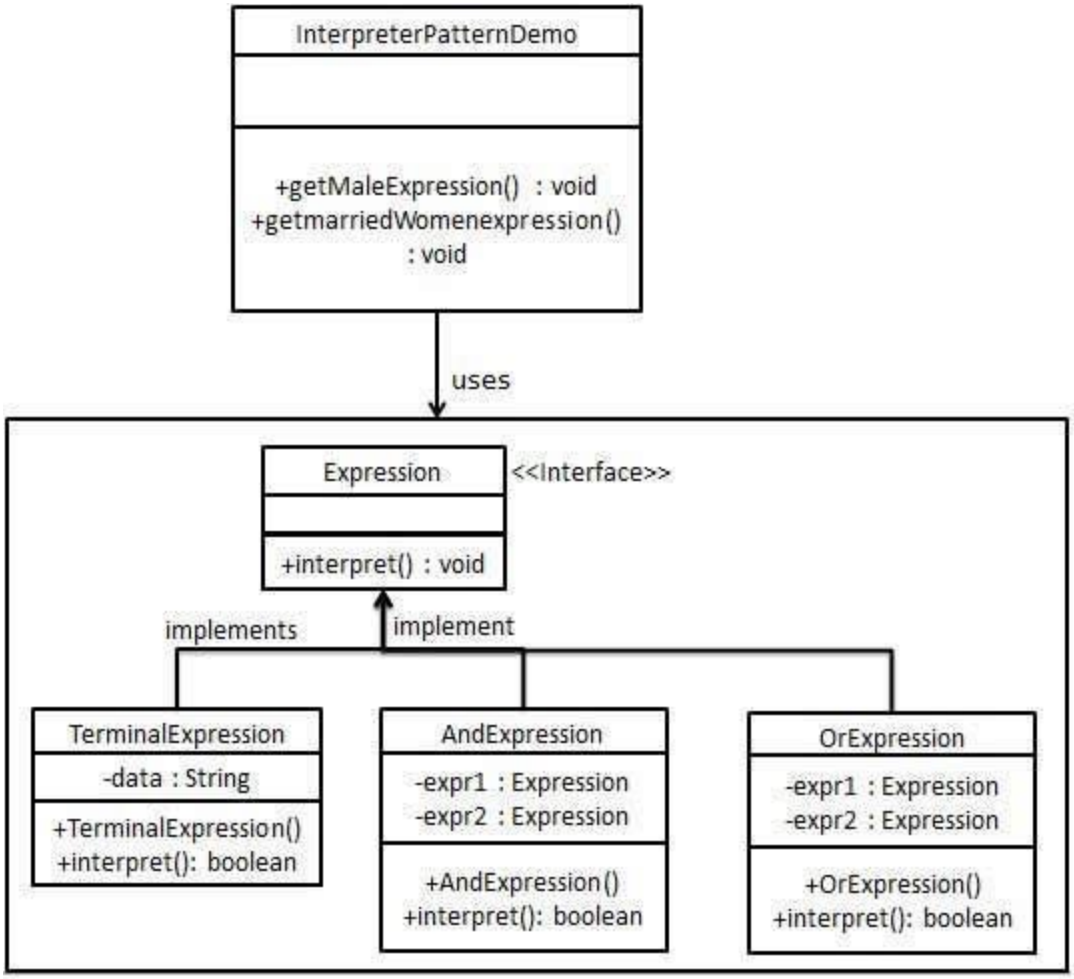
Example: We have created an interface Order which is acting as a command. We have created Stock class which acts as a request. We have concrete command

classes BuyStock and SellStock implementing Order interface which will do actual command processing. A class Broker is created which acts as an invoker object. It can take and place orders. Broker object uses command pattern to identify which object will execute which command based on the type of command. CommandPatternDemo, our demo class, will use Broker class to demonstrate command pattern.



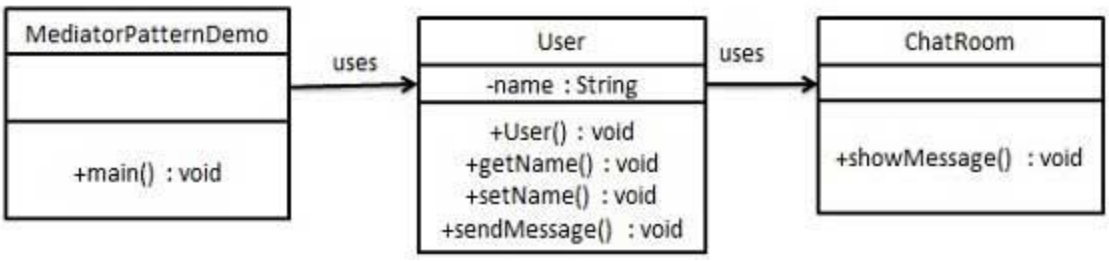

**Interpreter Pattern**: Provides a way to evaluate language grammar or expression. This type of pattern comes under behavioral pattern. This pattern involves implementing an expression interface which tells to interpret a particular context. This pattern is used in SQL parsing, symbol processing engine etc.

For example: To create an interface Expression and concrete classes implementing Expression interface. A class TerminalExpression is defined which acts as a main interpreter of context in question. Other classes OrExpression, AndExpression are used to create combinational expressions. InterpreterPatternDemo, our demo class, will use Expression class to create rules and demonstrate parsing of expressions.



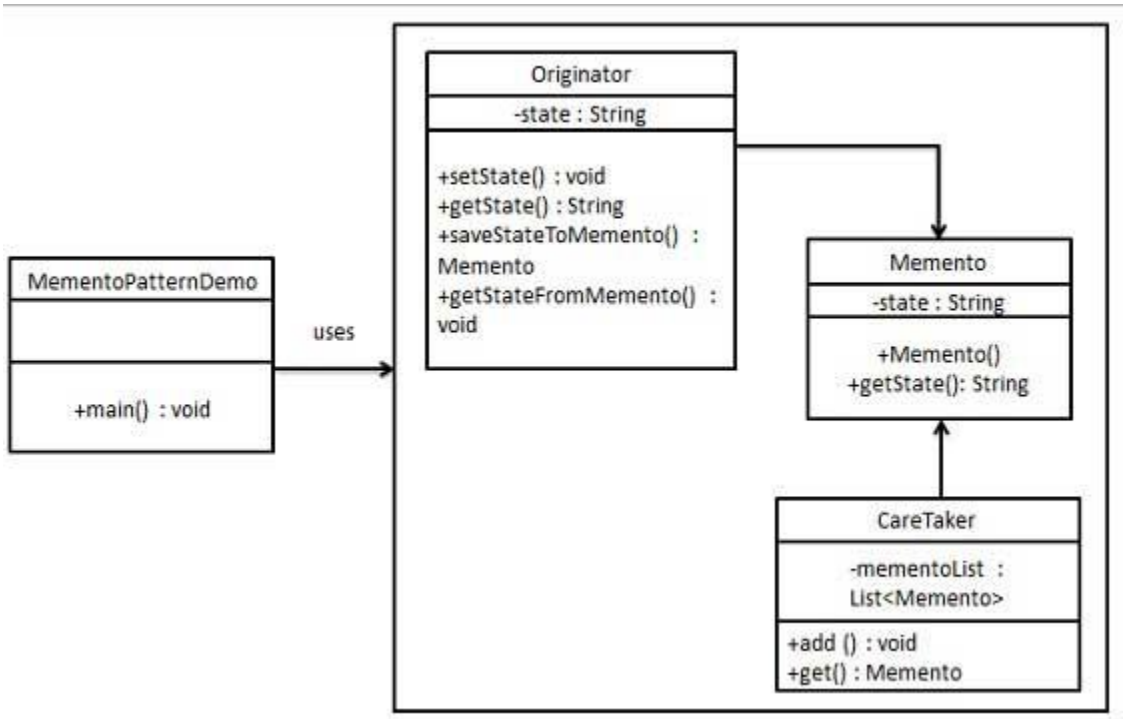

**Mediator**: Articulates simple communication between classes. Mediator pattern is used to reduce communication complexity between multiple objects or classes. This pattern provides a mediator class which normally handles all the communications between different classes and supports easy maintenance of the code by loose coupling.

Example: A chat room where multiple users can send message to chat room and it is the responsibility of chat room to show the messages to all users. We have created two classes ChatRoom and User. User objects will use ChatRoom method to share their messages. MediatorPatternDemo, our demo class, will use User objects to show communication between them.




**Observer:** The observer design pattern is "behavioral," linking an object (subject) to dependents (observers) in a one-to-many pattern. When any of the observers change, the subject is notified. The observer design pattern is useful in any kind of event-driven programming such as notifying a user of a new comment on Facebook, sending an email when an item ships, etc.

**Memento**: A process to save and restore the internal/original state of an object. Memento pattern is falls under behavioral pattern category and used to restore state of an object to a previous state. Memento pattern uses three actor classes. Memento contains state of an object to be restored. Originator creates and stores states in Memento objects and Caretaker object is responsible to restore object state from Memento. Example, we have created classes Memento, Originator and CareTaker. MementoPatternDemo, our demo class, will use CareTaker and Originator objects to show restoration of object states.



**Why do we need design patterns?**

Design patterns offer a best practice approach to support object-oriented software design, which is easier to design, implement, change, test and reuse. These design patterns provide best practices and structures.

- **Proven solution**

Design patterns provide a proven, reliable solution to a common problem, meaning the software developer does not have to "reinvent the wheel" when that problem occurs.

- **Reusable**

Design patterns can be modified to solve many kinds of problems – they are not just tied to a single problem.

- **Expressive**

Design patterns are an elegant solution.

- **Prevent the need for refactoring code**

Since the design pattern is already the optimal solution for the problem, this can avoid refactoring.

- **Lower the size of the codebase**

Each pattern helps software developers change how the system works without a full redesign. Further, as the "optimal" solution, the design pattern often requires less code.

**SELF-CHECK**

1. Which of the following is true about design patterns?
   A. Design patterns represent the best practices used by experienced object-oriented software developers.
   B. Design patterns are solutions to general problems that software developers faced during software development.
   C. Design patterns are obtained by trial and error by numerous software developers over quite a substantial period of time.
   D. All of the above.
2. Which of the following is correct about creational design patterns.
   A. These design patterns are specifically concerned with communication between objects.
   B. These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator.

C. These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.

D. None of the above.

3. Which of the following is correct about structural design patterns.
   A. These design patterns are specifically concerned with communication between objects.
   B. These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator.
   C. These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
   D. None of the above.
4. Which of the following is correct about behavioral design patterns.
   A. These design patterns are specifically concerned with communication between objects.
   B. These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator.
   C. These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
   D. None of the above.
5. Which of the following is correct about singleton design pattern.
   A. This type of design pattern comes under creational pattern.
   B. This pattern involves a single class which is responsible to create an object while making sure that only single object gets created.
   C. Singleton class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.
   D. All of the above.
6. Which of the following describes the builder pattern correctly?
   A. It builds a complex object using simple objects and using a step-by-step approach.
   B. It refers to creating duplicate object while keeping performance in mind.
   C. It is used when creation of object directly is costly.

D. It is used when we need to decouple an abstraction from its implementation so that the two can vary independently.

7. Which of the following describes the bridge pattern correctly?
   A. This pattern builds a complex object using simple objects and using a step-by-step approach.
   B. This pattern refers to creating duplicate object while keeping performance in mind.
   C. This pattern is used when creation of object directly is costly.
   D. This pattern is used when we need to decouple an abstraction from its implementation so that the two can vary independently.
8. Which of the following describes the prototype pattern correctly?
   A. It builds a complex object using simple objects and using a step-by-step approach.
   B. It refers to creating duplicate object while keeping performance in mind.
   C. It works as a bridge between two incompatible interfaces.
   D. It is used when we need to decouple an abstraction from its implementation so that the two can vary independently.
9. Which of the following pattern builds a complex object using simple objects and using a step-by-step approach?

   A Builder Pattern
   C Adapter Pattern
   B Bridge Pattern
   D Filter Pattern

10. Which of the following pattern refers to creating duplicate object while keeping performance in mind?

    A - Builder Pattern
    C - Prototype Pattern
    B - Bridge Pattern
    D - Filter Pattern

11. Which of the following pattern works as a bridge between two incompatible interfaces?

    A - Builder Pattern
    C - Prototype Pattern
    B - Adapter Pattern
    D - Filter Pattern

12. Which one of the following is used under class creation or object creation pattern?

    A. Builder
    C. Bridge
    B. Adapter
    D. All

13. Which of the following pattern is used when we need to decouple an abstraction from its implementation so that the two can vary independently?

    A - Bridge Pattern
    C - Prototype Pattern
    B - Adapter Pattern
    D - Filter Pattern

14. Which of the following pattern is used when creation of object directly is costly?

    A - Bridge Pattern
    C - Prototype Pattern
    B - Adapter Pattern
    D - Filter Pattern

## EXERCISE

1. List benefits of design patterns in Software Engineering?
2. Why do we need design patterns?
3. What are the constraints of the system?
4. What does the pattern consist of?
5. Can we create a clone of a singleton object?
6. Is design pattern a code?
7. Discuses essential elements of design patterns?

# Chapter Three

# Envisioning Architecture

Unit Objectives

At the end of this chapter the students will be able to:

- Understand meaning of software architecture.
- Understand architectural styles
- Understand common tools and terminology related to software architecture.
- Understand the role of the software architect with a development project.
- Use methods for constructing and evaluating architectures.

### 3.1 Architectural Business Cycle

Software architecture is a result of technical, business, and social influences. Its existence in turn affects the technical, business, and social environments that subsequently influence future architectures. This is known as cycle of influences, from the environment to the architecture and back to the environment is called the Architecture Business Cycle (ABC).

- The organization goals of architecture business cycle are begetting requirements, which beget an architecture, which begets a system. The architecture flows from the architect's experience and the technical environment of the day.
- Three things required for ABC are as follows:
  - Case studies of successful architectures crafted to satisfy demanding requirements, so as to help set the technical playing field of the day.
  - Methods to assess an architecture before any system is built from it, so as to mitigate the risks associated with launching unprecedented designs.
  - Techniques for incremental architecture-based development, so as to uncover design flaws before it is too late to correct them.

Where do architectures come from?

- Architectures are influenced by:
  - System stakeholders: The underlying problem, is that each stakeholder has different concerns and goals, some of which may be contradictory. The reality is that the architect often has to fill in the blanks and mediate the conflicts.

- The developing organization: The structure and nature of organization.
- The background and experience of the architects: Architects for a system have had good results using a particular architectural approach, chances are that they will try that same approach on a new development effort, or may be reluctant to try it again. Architectural choices may also come from an architect's education and training, exposure to successful architectural patterns, or exposure to systems that have worked particularly poorly or particularly well.
- The technical environment: Architect's background and experience is reflected by the technical environment. Current environment will influence the architecture– standard industry practices or software engineering techniques.

### 3.2 Software Architecture

A software architecture is a set of principles that define the way software is designed and developed. An architecture defines the structure of the software system and how it is organized. An architecture can be used to define the goals of a project, or it can be used to guide the design and development of a new system. Architecture serves as a blueprint for a system. It provides an abstraction to manage the system complexity and establish a communication and coordination mechanism among components. Software architecture is, simply, the organization of a system. This organization includes all components, the environment in which they operate, how they interact with each other / constraints between them, connectors, and the principles used to design the software. In many cases, it can also include the evolution of the software into the future. The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

"Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution." [ANSI/IEEE 1471- 400]. Software architecture defines a structured solution to meet all the technical and operational requirements, while optimizing the common quality attributes like performance and security. Further, it involves a set of significant decisions about the organization related to software development and each of these decisions can have a considerable impact on quality, maintainability, performance, and the overall

success of the final product. Software architecture defines a list of things which results in making many things easier in the software development process.

- A software architecture defines structure of a system.
- A software architecture defines behavior of a system.
- A software architecture defines component relationship.
- A software architecture defines communication structure.
- A software architecture balances stakeholder's needs.
- A software architecture influences team structure.
- A software architecture focuses on significant elements.
- A software architecture captures early design decisions.
- Must be designed to meet the specific requirements and constraints of the application it is intended for.

### 3.2.1 Characteristics of Software Architecture

Architecture characteristics define the software's requirements and what it is expected to do. Some of the characteristics shared by software architectures include:

**A description of the overall system setup:** This includes the structure of the software you want to build. To make it easier for stakeholders to understand, you might want to create a visual representation with diagrams and charts. Visuals are a great way to show relationships between components and subsystems. They give everybody involved insight into the architecture and give your perspective as you analyze the structure and look for ways to improve your structure or plan an expansion to an existing system.

**A definition of fundamental elements:** Software architecture defines the core set of elements and properties that are required to build the system. It does not document every element in detail. It simply identifies the structures that are required to build the software's core functionality. For example, a web browser and a web server describe the core elements needed for a user to interact with the internet.

**A description of high-level structures:** The development teams need to make decisions about the high-level structure that describe things like the system availability, performance, ability to scale, system reliability and fault tolerance, configuration and support, and monitoring and maintenance.

**A description of what is being built:** You are likely building software or a system to address the needs and requirements of stakeholders. But you can't always fully develop everything that the stakeholders ask for. A description of what you are building can help you manage stakeholder expectations. Use diagrams, flowcharts, and process documents to keep stakeholders informed and to avoid feature and scope creep.

Software design vs. software architecture

Software architecture and design are two separate parts of one process that depend on each other for success. Software architecture focuses on developing the skeleton and high-level infrastructure of software. Software design, on the other hand, focuses on how the software will be built. Software design is one of the initial phases of the software development life cycle. In this phase you analyze and identify the methods that your developers will use. Additionally, you define how the software will be built according to stakeholder and customer requirements.

Table 2: Difference between software design and software architecture

| S.No. | Software design | Software architecture |
|---|---|---|
| 01. | Software design is about designing individual modules/components. | Software architecture is about the complete architecture of the overall system. |
| 02. | Software design defines the detailed properties. | Software architecture defines the fundamental properties. |
| 03. | In general, it refers to the process of creating a specification of software artifact which will help to developers to implement the software. | In general, it refers to the process of creating high level structure of a software system. |
| 04. | It helps to implement the software. | It helps to define the high-level infrastructure of the software. |
| 05. | Software design avoids uncertainty. | Software architecture manages uncertainty. |

| | | |
|---|---|---|
| 06. | Software design is more about on individual module/component. | Software architecture is more about the design of entire system. |
| 07. | It is considered as one initial phase of Software Development Cycle (SSDLC) and it gives detailed idea to developers to implement consistent software. | It is a plan which constrains software design to avoid known mistakes and it achieves one organizations business and technology strategy. |
| 08. | Some of software design patterns are creational, structural and behavioral. | Some of software architecture patterns are microservice, server less and event driven. |
| 09. | In one word the level of software design is implementation. | In one word the level of software architecture is structure. |
| 10. | How we are building is software design. | What we are building is software architecture. |

### 3.2.2 Goals of Architecture

The primary goal of the architecture is to identify requirements that affect the structure of the application. A well-laid architecture reduces the business risks associated with building a technical solution and builds a bridge between business and technical requirements. Some of the other goals are as follows:

- Expose the structure of the system, but hide its implementation details.
- Realize all the use-cases and scenarios.
- Try to address the requirements of various stakeholders.
- Handle both functional and quality requirements.
- Reduce the goal of ownership and improve the organization's market position.
- Improve quality and functionality offered by the system.
- Improve external confidence in either the organization or system.

### 3.2.3 Benefits of Using System Architecture

The following are benefits associated with system architecture.

**Communication among stakeholders:**

The system architecture diagram is a visual representation of the software architecture. It depicts the system architecture, including its context, components, relationships, and dependencies. The key to a good system architecture is to clearly communicate its requirements to your stakeholders and developers, and to have a well-defined system architecture from the beginning. A well-defined system architecture enables you to focus on the development of the software, and avoid long-term problems with integration and operational issues.

Customer satisfaction provides the key to a great architecture. If you do not hear the customers' voice in your decision-making, you will not know what direction to take. It does not matter how great the architect is if the customers do not buy the product. The same applies to your software architecture. It does not matter how good the developer is if the end user does not feel comfortable with the software. In order to know what to build and how to build it, you need to know what the customer values and how to keep them happy. The diagram shows the customer value that a supplier's product or service adds to the customer's business. It also shows the value of the supplier's market position to potential customers. As a supplier's value increases, so does the diagram's importance. For a high-value supplier, the diagram is often expanded to show the customer value the supplier's solution adds to the customer's business.

**Software architecture helps to make early design decisions:**

Software architecture manifests the earliest design decisions about a system, which influence the system's remaining development, its deployment, and maintenance life. The architecture defines constraints on implementation. The architecture dictates the organizational structure. The architecture inhibits or enables a system's quality attributes. e.g., If your system requires high performance, you need to manage the time-based behavior of elements and the frequency and volume of inter-element communication.

**Software architecture provides a transferable, re-usable model:**

Software architecture constitutes a relatively small, intellectually graspable model for how a system is structured and how its elements work together, and this model is transferable across systems. In particular, it can be applied to other systems exhibiting similar quality attributes and functional requirements and can promote large-scale re-use.

### 3.2.4 Terminology in Software Architecture

**Components**

- An individual software component is a software package, a web service, a web resource, or a module that encapsulates a set of related functions (or data). It is an architectural entity that encapsulates a subset of the system's functionality and/or data.
- All system processes are placed into separate components so that all of the data and functions inside each component are semantically related.

**Types of components**

- **Computational**: does a computation of some sort. E.g., function, filter.
- **Memory**: maintains a collection of persistent data. E.g., database, file system, symbol table.
- **Manager**: contains state + operations. State is retained between invocations of operations. E.g., server.

**Controller**

- Controller: governs time sequence of events. E.g., control module,
- **Constraints** are limitations on the design, may include a constraint that the system must use predefined hardware or software, use of a particular algorithm and etc.

**Connector**

- A software connector is an architectural building block, that regulates interactions among components. Connectors typically provide application-independent interaction facilities. In many software systems, connectors are usually simple procedure calls or shared data accesses. Much more sophisticated and complex connectors are possible!

**Types of connectors:**

- Procedure call connectors
- Shared memory connectors
- Message passing connectors
- Streaming connectors
- Distribution connectors
- Wrapper/adaptor connectors

**Sub-System**

- A sub-system is a system in its own right whose operation is independent of the services provided by other sub-systems.

**A module**

- A module is a system component that **provides services** to other components but would not normally be considered as a separate system.

**Configuration/Topology**

- An **architectural configuration**/topology, is a set of specific associations between the components and connectors of a software system's architecture.
- Components and connectors are composed in a specific way in a given system's architecture to accomplish that system's objective.

**Middleware**

- Middleware is connectivity software that consists of a set of enabling services that allow multiple processes running on one or more machines to interact with each other.
- Middleware is the "**glue**" that connects diverse computer systems. Typically, legacy systems store information in proprietary formats, use propriety protocols to communicate, and may even be running on hardware that's no longer manufactured or supported.

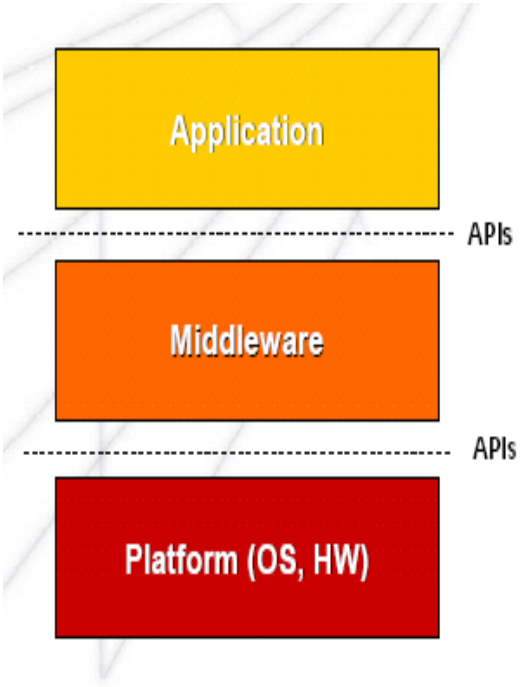



Fig: Middleware

- Layer between OS and distributed applications
- Hides complexity and heterogeneity of distributed system
- Bridges gap between low-level OS communications and programming language abstractions.

- Provides common programming abstraction and infrastructure for distributed applications
- Middleware Types: -
    - Transaction Processing (TP) monitors
    - Remote Procedure Calls (RPC)
    - Message Oriented Middleware (MOM)
    - Object Request Brokers (ORBs)
    - Web Service
    - Event-Based Middleware

### 3.3 Role of Software Architect

A Software architect provides a solution that the technical team can create and design for the entire application.

- Convert customer requirements into a technical design.
- Lead the problem domain analysis team.
- Ensure that the technical design meets quality requirement.
- Perform continuous risk assessment, and develop risk mitigation strategies.
- Perform early prototyping aimed at mitigating major risks.
- Communicate with stakeholders through detailed technical presentations.
- Listen to stakeholders and build consensus.
- Review developer code and ensure conformance to the architecture and good coding practices.
- Serve as a mentor for analysts, designers, and developers.
- Meet with clients to determine objectives and requirements for structures.
- Give preliminary estimates on cost and construction time.
- Prepare structure specifications.
- Direct workers who prepare drawings and documents.
- Prepare scaled drawings, either with computer software or by hand.

### 3.4 Architecture Style

Software architecture patterns are related to software architecture styles, but they are unique concepts. Architecture patterns are essentially organizational software templates in which software elements can be placed, while architecture styles function like different formats for the different elements. Software architecture styles are set of rules, constraints, or patterns of how to structure a system into a set of components and

connectors. According to Garlan and Shaw, architectural style defines a family of systems in terms of a pattern of structural organization. More specifically, an architectural style defines a vocabulary of components and connector types and a set of constraints on how they can be combined. These can include topological constraints on architectural descriptions (e.g., no cycles). Other constraints- say, having to do with execution semantics—might also be part of the style definition.

An architectural style/ pattern is a set of principles. You can think of it as a coarse-grained pattern that provides an abstract framework for a family of systems. It is a general, reusable solution to a commonly occurring problem in software architecture within a given context. An architectural style improves partitioning and promotes design reuse by providing solutions to frequently recurring problems. Architectural patterns are similar to software design pattern but have a broader scope. Key architectural styles in Software Engineering are:

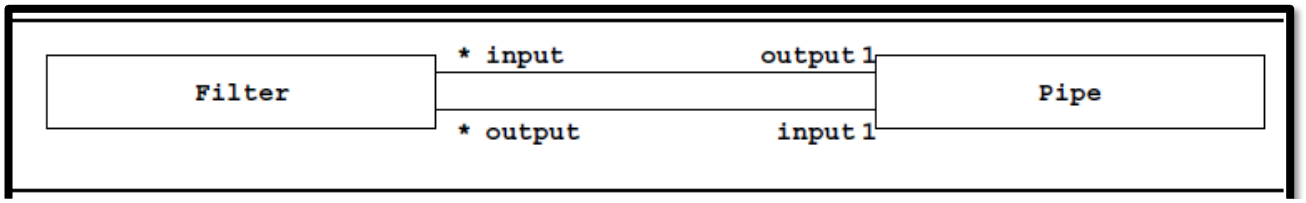
1. Pipes-and-Filter
2. Client-Server
3. Peer-to-Peer
4. Event Based
5. Layering
6. MVC (Model-View-Controller Pattern)
7. SOA (Service Oriented Architecture)

### 3.4.1 Pipe-Filter

This pattern can be used to structure systems which produce and process a stream of data. Each processing step is enclosed within a filter component. In this architecture subsystems process data received from a set of inputs and send results to other subsystems via a set of outputs. The subsystems are called filters, and the associations between the subsystems are called pipes. These pipes can be used for buffering or for synchronization purposes.

Data is processed by being separated or filtered between different elements or pipelines, which conjoin their results into an output viewed by the software user, significantly reducing the power necessary to complete the computing task. In other words, data from multiple sources are put into discrete sections of a software program that all concurrently perform the same pattern of tasks on each data set. The end result is multiple datasets, all in the same format, that have been produced using the same tasks. These sets are then supplied to the user and can be analyzed, making it a

fairly reliable software pattern, but it is also very complex. There is also a small risk for data redundancy. A filter can have many inputs and outputs. A Pipe connects one of the outputs of a filter to one of the inputs of another filter as shown in below:




"The **Pipes and Filters** architectural pattern provides a structure for systems that **process a stream of data**. Each processing step is encapsulated in a filter component. Data [are] passed through pipes between adjacent filters. Recombining filters allows you to build families of related filters."

Implementation steps:

- Divide the functionality of the problem into a sequence of processing steps.
- Define the type and format of the data to be passed along each pipe.
- Determine how to implement each pipe connection.
- Design and implement the filters.

Note: The design of a filter is based on the nature of the task to be performed and the nature of the pipes to which it can be connected.

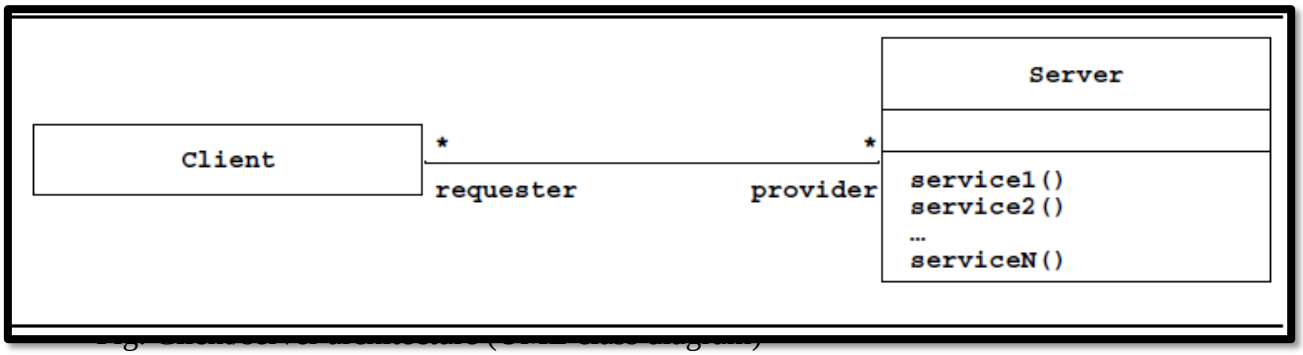
- Drawbacks
  - Encourages batch processing
  - Not good for handling interactive application
  - Duplication in filters functions.

### 3.4.2 Client-Server

The client-server architecture refers to a system that hosts, delivers, and manages most of the resources and services that the client requests. In this model, all requests and services are delivered over a network, and it is also referred to as the networking computing model or client server network. Client-server architecture, alternatively called a client-server model, is a network application that breaks down tasks and workloads between clients and servers that reside on the same system or are linked by a computer network.

Client-server architecture typically features multiple users' workstations, PCs, or other devices, connected to a central server via an Internet connection or other network. The client sends a request for data, and the server accepts and accommodates the request, sending the data packets back to the user who needs them. A client-server architecture distributes application logic and services respectively to a number of client and server sub systems, each potentially running on a different machine and communicating through the network (e.g, by RPC).

- Two types of components:
  - Server components offer services (Response/Reply)
  - Clients sends request and receive response from server.
- Client may send the server a request and server reply a response to the request. It is a kind of request and response/reply mechanism.
- In the client/server architecture a subsystem, the **server,** provides services to instances of other subsystems called the **clients,** which are responsible for interacting with the user.
- The request for a service is usually done via a remote procedure call mechanism or a common object broker.
- The client/server architecture is a generalization of the repository architecture.




**Advantages**

- Distribution of data is straightforward
- Makes effective use of networked systems.
- May require cheaper hardware
- Easy to add new servers or upgrade existing servers

**Disadvantages**

- Redundant management in each server.

- May require a central registry of names and services : it may be hard to find out what servers and services are available.

### 3.4.3 Peer-to-Peer

In this pattern, individual components are known as **peers**. Peers may function both as a **client**, requesting services from other peers, and as a **server**, providing services to other peers. A peer may act as a client or as a server or as both, and it can change its role dynamically with time.

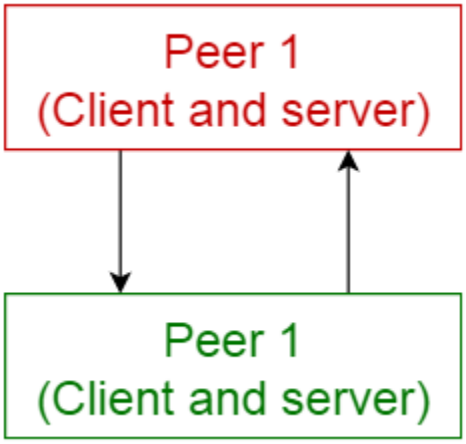



Fig. Peer-to-peer architecture

- It is a generalization of the client/server architecture in which subsystems can act both as client or as servers, in the sense that each subsystem can request and provide services.
- The control flow within each subsystem is independent from the others except for synchronizations on requests.
- Each component acts as its own process and acts as both a client and a server to other peer components.
- Any component can initiate a request to any other peer component.
- Characteristics
  - Scale up well
  - Increased system capabilities
  - Peers are distributed in a network, can be heterogeneous, and mutually independent.
  - Robust in face of independent failures.
  - Highly scalable

- This differs from client/server_architectures, in which some computers are dedicated to serving others.
- Components do not offer the same performance under heavy loads.

Client-Server vs. Peer-to-Peer

Peer-to-peer networks, also called P2P networks, consist of groups of computers (also called nodes or peers) linked together in a network, where peers act as both a client and a server. Peers have equal responsibilities and permissions to work with data. This setup radically differs from the client-server model, which has very defined groups of users and servers. Here are the main differences between the two network models:

- Client-server networks need a central file server and consequently cost more to implement; peer-to-peer doesn't have that server.
- Client-server networks delineate between users and providers; peers act as both consumers and providers.
- Client-server networks offer more levels of security, making them safer. The end-users are responsible for peer-to-peer network security.
- The more active nodes in a peer-to-peer network, the more its performance suffers. Client-server networks offer better stability and scalability. The ideal range for P2P networks is two to eight users.
- Peer-to-peer users can share files faster and easier than on a client-server network.
- If a client-server network server crashes, everything comes to a halt, but if a single node in a P2P network fails, the rest remains operational.

### 3.4.4 Event-Based Architecture

This pattern primarily deals with events and has 4 major components; event source, event listener, channel and event bus. Sources publish messages to particular channels on an event bus. Listeners subscribe to particular channels. Listeners are notified of messages that are published to a channel to which they have subscribed before.

- Components interact by broadcasting and reacting to events
  - Component expresses interest in an event by subscribing to it

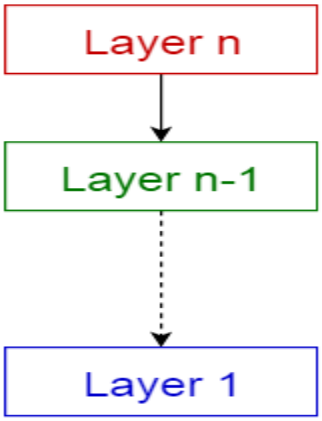
- When another component announces (publishes) that event has taken place, subscribing components are notified
- Implicit invocation is a common form of publish-subscribe architecture
- Registering: subscribing component associates one of its procedures with each event of interest (called the procedure)

➢ Characteristics

- Strong support for evolution and customization
- Easy to reuse components in other event-driven systems
- Difficult to test

### 3.4.5 Layered Style

This pattern can be used to structure programs that can be decomposed into groups of subtasks, each of which is at a particular level of abstraction. Each layer provides services to the next higher layer.




Layered pattern

➢ Layers are hierarchical

- Each layer provides service to the one outside it and acts as a client to the layer inside it.
- Layer bridging: allowing a layer to access the services of layers below its lower neighbor.
- Each layer has two interfaces.
- Upper interface provides services and the lower interface requires services.

➢ The design includes protocols

- Explain how each pair of layers will interact

➢ Advantages

- High levels of abstraction
- Relatively easy to add and modify a layer

➢ Disadvantages

- Not always easy to structure system layers
- System performance may suffer from the extra coordination among layers

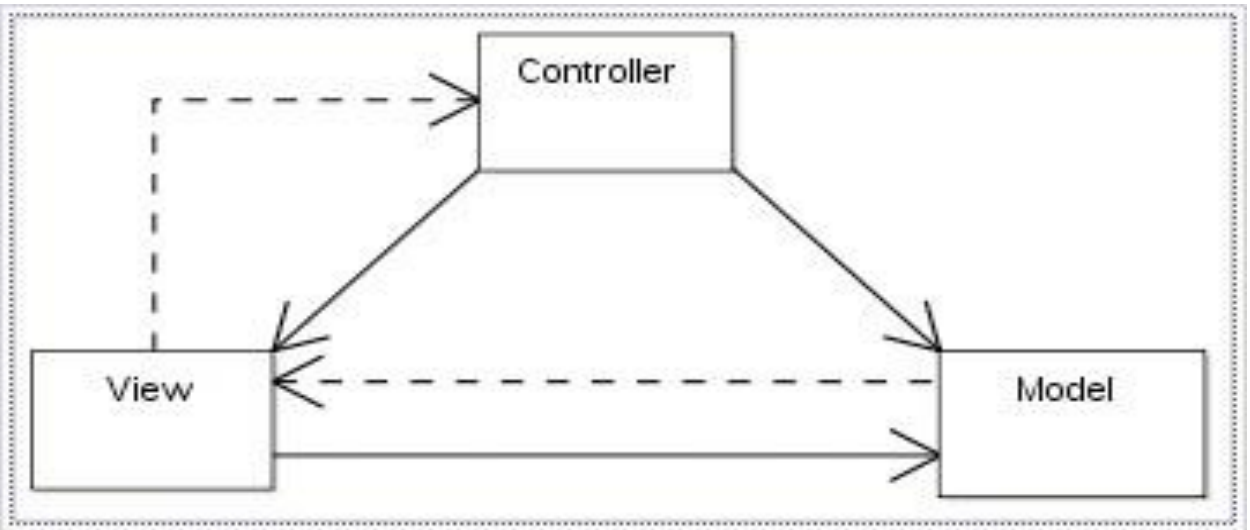
### 3.4.6 Model-View-Controller

In the MVC paradigm, the user input, the modeling of the external world, and the visual feedback to the user are explicitly separated and handled by three types of objects, each specialized for its task.

➢ The **view** manages the graphical and/or textual output to the portion of the bitmapped display that is allocated to its application.

➢ The **controller** interprets the mouse and keyboard inputs from the user, commanding the model and/or the view to change as appropriate.

➢ The **model** manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller). The pattern isolates business logic from input and presentation, permitting independent development, testing and maintenance of each.



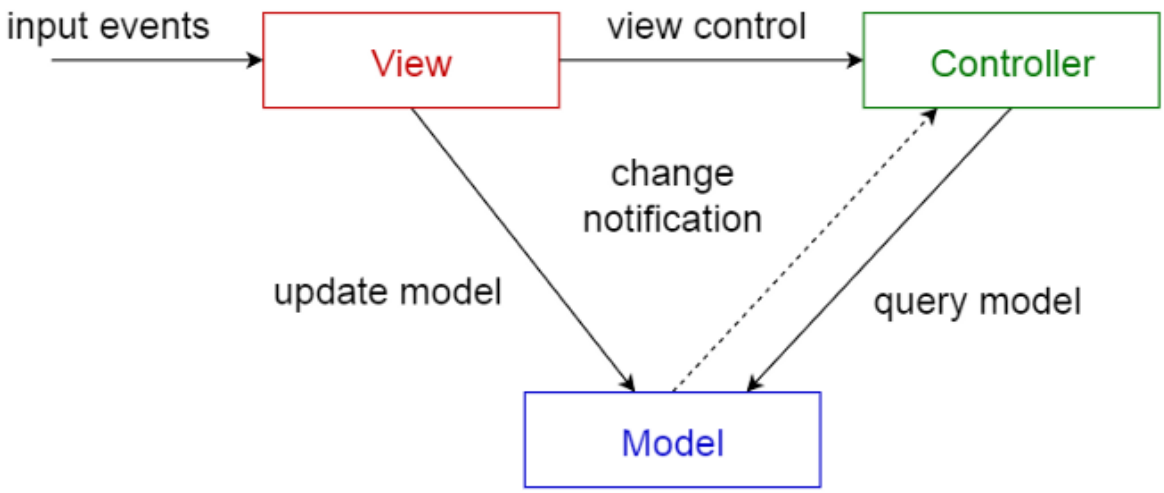

Model-View-Controller concept. Note: The solid line represents a direct association, the dashed an indirect association via an observer (for example).

The MVC paradigm is a way of breaking an application, or even just a piece of an application's interface, into three parts:

1. **Model**: contains the core functionality, data, the business logic, rules, and strategies.
2. **View**: displays the information to the user (more than one view may be defined) or displays the model more than one view and usually has components that allow user to edit or change the model.
3. **Controller**: handles the input from the user.

- Allows data to flow between the view and the model
- The controller mediates between the view and model

This is done to separate internal representations of information from the ways information is presented to, and accepted from, the user. It decouples components and allows efficient code reuse.




Model-view-controller pattern

MVC Benefits

- Clarity of design
  - easier to implement and maintain
- Modularity
  - changes to one don't affect the others
  - can develop in parallel once you have the interfaces

**Combining architectural styles**

The architecture of a software system is almost never limited to a single architectural style but is often a combination of architectural styles that form the complete system. For example, you might have an SOA design composed of services developed using a layered architecture approach and an object-oriented architecture style.

- Use different styles at different layers (e.g., overall client-server architecture with server component decomposed into layers)
- Use a mixture of styles to model different components or types of interaction (e.g., client components interact with one another using publish-subscribe communications
- If architecture is expressed as a collection of models, documentation must be created to show the relation between models. The following examples show how combinations of some of these styles can be used to build applications:
- **Example 1:** A combination of architecture styles will be useful if you are building a public-facing Web application. Employ effective separation of concerns by using the layered architecture style. This will separate your presentation logic from your business logic and your data access logic. Your organization's security requirements might make you deploy the application using either the 3-tier deployment or a deployment of more than three tiers. The presentation tier may be deployed in the perimeter network, which sits between an organization's internal network and an external network. On your presentation tier (Web server), you might decide on a separated presentation architecture style for your interaction model, such as MVC. From a communication standpoint between your Web server and application server, you might choose an SOA architecture style and implement message-based communication.
- **Example 2:** If you are building a desktop application (Windows Forms application), you might prefer to have a client that sends a request to a program on the server. Deploy the client and server using the client/server architecture style. Use the component-based architecture to decompose the design further into independent components that expose the appropriate communication interfaces.

Summary of architectural style

| Architectural Style | Description |
|---|---|
| Client-Server | Segregates the system into two applications, where the client makes a service request to the server. |
| Layered Architecture | Partitions the concerns of the application into stacked groups (layers) such as presentation layer, business layer, data layer, and services layer. |
| N-tier/3-tier | Segregates functionality into separate segments in much the same way as the layered style, but with each segment being a tier located on a physically separate computer. |
| Service-Oriented Architecture (SOA) | Refers to Applications that expose and consume functionality as a service using contracts and messages |

Choosing an architectural style

The definition and choice of software architecture is completed during the design phase. The architecture of a software tool represents the way that the elements will be organized and arranged – and how they will interconnect with each other. The architect generally has the role of designing the architecture. To be sure that the chosen software architecture will meet the needs of the software that will be developed, it is very important that the architect fully grasps and understands the business needs that will be served. These should be as equally understood as the technical and functional expectations of the application. There are several factors that influence the architectural styles that you choose. Some factors include the capacity of your organization for design and implementation; the capabilities and experience of developers; and the hardware and deployment scenarios available or consider these criteria:

- Security and performance requirements
- Client and vendor expectations
- The type of hosting
- Operating systems
- Technologies

To sum up, a good architecture is characterized by the following capacities:

- Evolution
- Level of simplicity
- Maintainability

- Interconnectivity

If you have incorporated these four aspects into your software architecture design then you are on the right track.

### 3.5 Architectural View Models

Architecture view model represents the functional and non-functional requirements of software application.

A model is a complete, basic, and simplified description of software architecture which is composed of multiple views from a particular perspective or viewpoint.

A view is a representation of an entire system from the perspective of a related set of concerns. It is used to describe the system from the viewpoint of different stakeholders such as end-users, developers, project managers, and testers.

#### 3.5.1 4+1 View Model

The 4+1 View Model was designed by Philippe Kruchten to describe the architecture (design) of a software intensive system using several, concurrent views. It is a multiple view model that addresses different features and concerns of the system. It standardizes the software design documents and makes the design easy to understand by all stakeholders. End-users, developers, system engineers, clients, architects, and project managers all have unique views on the system, hence the viewpoints are used to describe it from their perspectives.

The reason behind the name: 4+1


Five views in the 4+1 model

**Model Description**

**Logical View:** The logical view is concerned with the system's functionality as it pertains to end-users. The logical view focuses mostly on achieving the functional requirements of a system. Class diagrams and state diagrams are examples of UML diagrams that are used to depict the logical view.
**(Object-oriented Decomposition)**
**Viewer:** End-user
**Considers:** Functional requirements- What the system should provide in terms of services to its users.
**Notation:** object and dynamic models
**Process View:** The process view focuses on the system's run-time behavior and deals with the system's dynamic elements. It explains the system processes and how they communicate. This focuses on describing the concurrency and communications elements of an architecture. The process view focuses on achieving nonfunctional requirements which specify the desired qualities for the system. Nonfunctional requirements like concurrency, distribution, integrator, performance, and scalability are all addressed in the process view. The sequence diagram, communication diagram, and activity diagram are all UML diagrams that can be used to describe a process view.
**(The process decomposition) viewer:** Integrators

**Considers:** non-functional requirements (concurrency, performance, scalability)

**Development View:** The development view depicts a system from the standpoint of a programmer and is concerned with software administration. This captures the internal organization of the software components, typically as they are held in a development environment or configuration management tool. The implementation view is another name for this view. It describes system components using the UML Component diagram. The Package diagram is one of the UML diagrams used to depict the development view.

**(Subsystem decomposition)**

**Viewer:** Programmers and Software Managers

**Considers:** software module organization

(Hierarchy of layers, software management, reuse, constraints of tools)

**Notation:** module, subsystem, layer.

**Physical View:** The physical view portrays the system from the perspective of a system engineer. The physical layer, it is concerned with the topology of software components as well as the physical connections between these components. This depicts how the major processes and components are mapped on to the application's hardware. It might show, for example, how the database and web servers for an application are distributed across a number of server machines. The deployment view is another name for this view. The deployment diagram is one of the UML diagrams used to depict the physical perspective.

(Mapping the software to the Hardware) Viewer: System Engineers

Considers Non-functional req. regarding to underlying hardware (Topology, Communication)

**Scenarios:** These views are tied together by the architecturally significant use cases (often called scenarios).

These basically capture the requirements for the architecture or used to illustrate the description of architecture and hence are related to more than one particular view. By working through the steps in a particular use case, the architecture can be "tested", by explaining how the design elements in the architecture respond to the behavior required in the use case. Sequences of interactions between objects and processes are described in the scenarios. They are used to identify architectural aspects as well as to demonstrate and assess the design of the architecture. They can also be used as a starting point for architecture prototype testing. The use case view is another name for this view.

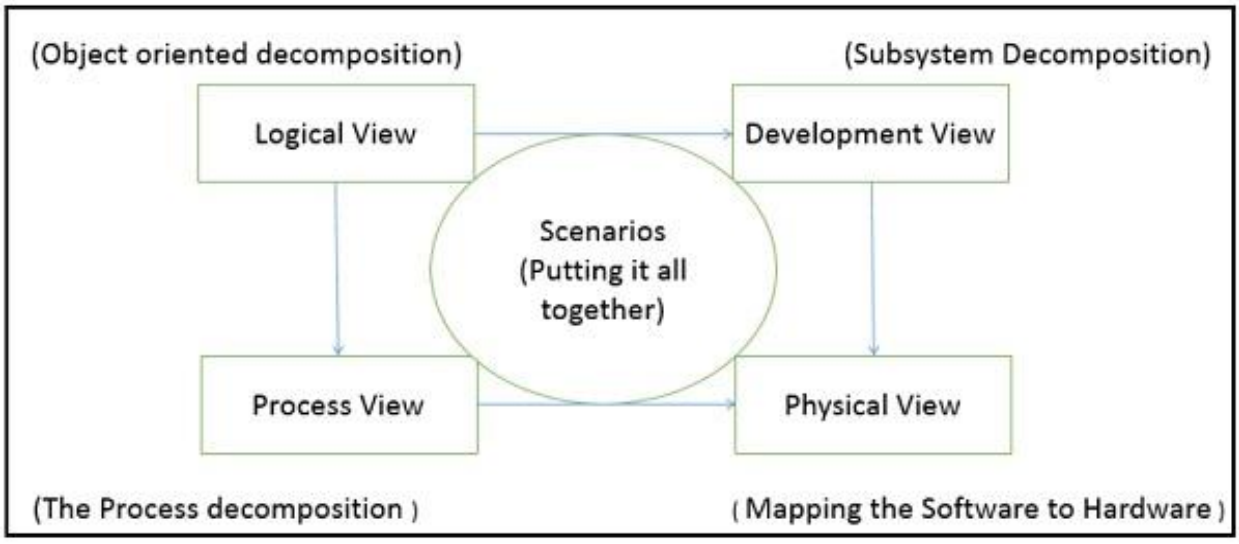
(Putting it all together)

Viewer: All users of other views and evaluators.

Considers: System consistency, validity

Notation: almost similar to the logical view

The following figure describes the software architecture using five concurrent views (4+1) model.




The following table shows the 4+1 view in detail:

| | **Logical** | **Process** | **Development** | **Physical** | **Scenario** |
|---|---|---|---|---|---|
| Description | Shows the component (Object) of system as well as their interaction | Shows the processes / Workflow rules of system and how those processes communicate, focuses on dynamic view of system | Gives building block views of system and describe static organization of the system modules | Shows the installation, configuration and deployment of software application | Shows the design is complete by performing validation and illustration |

| | | | | | |
|---|---|---|---|---|---|
| Viewer / Stake holder | End-User, Analysts and Designer | Integrators & developers | Programmer and software project managers | System engineer, operators, system administrators and system installers | All the views of their views and evaluators |
| Consider | Functional requirements | Non-Functional Requirements | Software Module organization (Software management reuse, constraint of tools) | Nonfunctional requirement regarding to underlying hardware | System Consistency and validity |
| UML – Diagram | Class, State, Object, sequence, Communication Diagram | Activity Diagram | Component, Package diagram | Deployment diagram | Use case diagram |

### 3.6 Enterprise Architectures

An enterprise architecture (EA) is a conceptual blueprint that defines the structure and operation of organizations. The intent of enterprise architecture is to determine how an organization can effectively achieve its current and future objectives. Enterprise architecture involves the practice of analyzing, planning, designing and eventual implementing of analysis on an enterprise. Concepts of enterprise architecture are variable, so it will not look the same for each organization. Different parts of an organization may also view EA differently. For example, programmers and other technical IT professionals regard enterprise architecture strategies in terms of the infrastructure, application and management components under their control. However, enterprise architects are still responsible for enacting business structure analysis. There are four types of architecture from the viewpoint of an enterprise and collectively, these architectures are referred to as enterprise architecture.

- Business architecture: Defines the strategy of business, governance, organization, and key business processes within an enterprise and focuses on the analysis and design of business processes.
- Application (software) architecture: Serves as the blueprint for individual application systems, their interactions, and their relationships to the business processes of the organization.
- Information architecture: Defines the logical and physical data assets and data management resources.
- Information technology (IT) architecture: Defines the hardware and software building blocks that make up the overall information system of the organization.

**SELF-CHECK**

1. Which one of the following is not the source of architecture requirement?
   A. Stakeholder needs
   B. Functional requirement documents
   C. Architectural styles
   D. None
2. Which element of the system is chosen to be designed for existing systems which needs enhancement?
   A. The whole system should be chosen
   B. The part of the system to be added should be chosen
   C. Both whole and part of the system should be chosen
   D. All
3. _____ is a type of software architecture in which subsystems can act both as a client or as servers that each subsystem can request and provide services.
   A. Client/ Server Architecture
   B. Central Repository Architecture
   C. Pipe and Filter Architecture
   D. Peer-to-Peer Architecture
4. What is the purpose of a component in software architecture?
   A. To provide a high-level view of the software system
   B. To represent a single, isolated piece of functionality
   C. To define the relationships between software elements
   D. To document the structure and behavior of a software system

5. In which type of client-server architecture the user interface is stored on the client's side and the database is stored on the server, while database logic and business logic are preserved either on the client's side or on the server's side?
   A. N-tier architecture
   B. 1-tier architecture
   C. 3-tier architecture
   D. 2 tier architecture
6. What is the purpose of the Model-View-Controller (MVC) architectural style?
   A. To provide a way to modularize code and increase maintainability
   B. To improve the performance of a software system.
   C. To separate the presentation layer from the business logic and data storage layers.
   D. To simplify the testing of a software system
7. Which of the following factors are influenced on the architect?
   A. Background and experience of the architects
   B. Developing an organization
   C. Customers and end users
   D. All of the above
8. Open system interconnection model is an example of
   A. MVC
   B. Pipe and Filter
   C. Client-Server
   D. Layered
9. Architect is responsible for
   A. Project planning
   B. comprising system designers to produce the architecture blue print
   C. All
   D. Done
10. The architectural style where computation is achieved by cooperating peers that request service from and provide service to one another across a network is
   A. Client server
   B. Peer to peer
   C. Pipe and filter
   D. MVC
11. A set of rules, constraints, or patterns of how to structure a system into a set of components and connectors.
   A. Architectural design
   B. Architectural style
   C. Design pattern
   D. B & C

12. In which architecture of client-server architecture, a middleware lies between the client and the server, and every request made by the client is first received by middleware?

    A. 1-tier architecture
    B. 3-tier architecture
    C. 2 tier architecture
    D. N-tier architecture

## EXERCISE

1. What are the qualities that a software architect needs to be successful?
2. Why is software architecture important?
3. What is software architecture?
4. Discuss the difference and the similarity of software design and software architecture?
5. Discus architectural structures and views?
6. How are software architecture and design related?
7. What is the role of a software architect in a software development team and how do they work with other stakeholders such as developers, product owners, and designers?
8. When to choose a layered architecture pattern?
9. What are software architecture patterns?
10. How to choose the best architecture, considering the context?
11. Why architecture view model called 4+1 instead of 5?
12. What mean by three-tier architecture?

# Chapter Four

# Quality Attributes

## 4.1 Understanding Quality Attributes

Quality attributes describe externally visible properties of a software system and the expectations for that system's operation. Quality attributes define how well a system should perform some action. Sometimes called quality requirements. Here is a list of some common quality attributes from Software Architecture in Practice [BCK12].

| Design Time Properties | Runtime Properties | Conceptual Properties |
|---|---|---|
| Modifiability<br>Maintainability<br>Reusability<br>Testability<br>Buildability or Time-to-Market | Availability<br>Reliability<br>Performance<br>Scalability<br>Security | Manageability<br>Supportability<br>Simplicity<br>Teachability |

### 4.1.1 Functionality and quality attributes

Functionality and quality attributes are orthogonal. What is functionality? It is the ability of the system to do the work for which it was intended. A task requires that many or most of the system's elements work in a coordinated manner to complete the job. If the elements have not been assigned the correct responsibilities or have not been endowed with the correct facilities for coordinating with other elements. Functionality may be achieved through the use of any of a number of possible structures. In fact, if functionality were the only requirement, the system could exist as a single monolithic module with no internal structure at all. Instead, it is decomposed into modules to make it understandable and to support a variety of other purposes. In this way, functionality is largely independent of structure. Software architecture constrains its allocation to structure when other quality attributes are important.

### 4.1.2 Architecture and Quality Attributes

According to Bass et al., 2013, a key in designing software architectures is the satisfaction of quality attribute requirements. Achieving quality attributes must be considered throughout

design, implementation, and deployment. Architecture is critical to the realization of many qualities of interest in a system, and these qualities should be designed in and can be evaluated at the architectural level. Architecture, by itself, is unable to achieve qualities. It provides the foundation for achieving quality, but this foundation will be to no avail if attention is not paid to the details. So, every architecture decision promotes or inhibits at least one quality attribute.

Types of quality attributes:

1. Qualities of the system
2. Business qualities
   - Time to market.
   - Cost and benefit.
   - Projected lifetime of the system.
   - Targeted market.
   - Rollout schedule.
3. Qualities that are about the architecture itself

### 4.1.3 System Quality Attributes

Software Quality Attributes Are Invisible, Until Things Go Wrong.

- Software quality attributes are one of the two types of non-functional requirements (user invisible and user visible).
- Commonly identified as the software "ilities" (security, availability, scalability, and more), are often considered part of the work that isn't visible for the users but provides positive value for them. Sure, it can be classified as invisible when it's working correctly, but when it isn't, your users will surely notice.
- Much of a software architect's life is spent designing software systems to meet a set of quality attribute requirements.
- Quality attribute requirements are part of an application's nonfunctional requirements, which capture the many facets of how the functional requirements of an application are achieved.

**Finding a Balance Between Quality Attributes is Critical**

Like many things in life, when it comes to software quality attributes, you can't have it all. Several tactics can help in the goal of achieving a desired quality level, but they will inevitably conflict with another attribute. A typical example is security. Many times, trying to make a system more

secure means encrypting data, which consequently translates into slower processing times (affecting performance). Sometimes it also means adding controls and verifications that make a process that otherwise would be simple, cumbersome by affecting usability.

**The Right Attributes, at the Right Level**

There are quality attributes that are common to most applications, like performance, security, or availability. Others may only apply to specific scenarios, like interoperability, cost, time to market, or safety. Any organization needs to identify the right attributes for the application, so they can use resources efficiently.

Moreover, not all quality attributes will have the same priority; we need to think of them as different scenarios refined at the right level of detail so we can plan accordingly. A scenario describes a system response to a specific stimulus under particular conditions and the desired response. Quality is, by nature, a subjective concept. Expressing it in objective and succinct terms makes it easier to achieve.

**Quality attribute scenarios**

A quality attribute scenario is a quality-attribute-specific requirement. It consists of six parts.

1. Source of stimulus. This is some entity (a human, a computer system, or any other actuator) that generated the stimulus.
2. Stimulus. The stimulus is a condition that needs to be considered when it arrives at a system.
3. Environment. The stimulus occurs within certain conditions. The system may be in an overload condition or may be running when the stimulus occurs, or some other condition may be true.
4. Artifact. Some artifact is stimulated. This may be the whole system or some pieces of it.
5. Response. The response is the activity undertaken after the arrival of the stimulus.
6. Response measure. When the response occurs, it should be measurable in some fashion so that the requirement can be tested.

Quality attribute parts

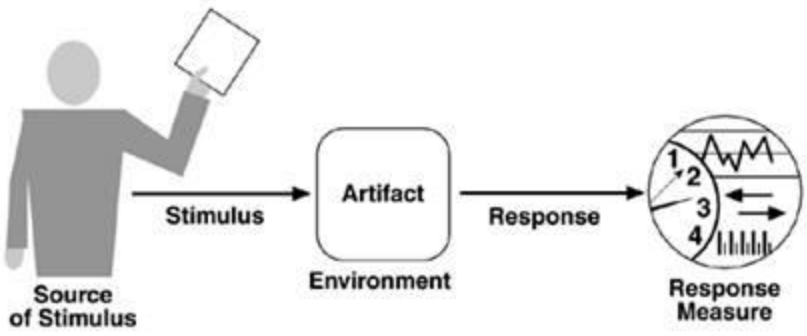



Figure: shows the parts of a quality attribute scenario

Quality Attribute Scenarios in Practice

**Availability**

- Availability is concerned with system failure and its associated consequences.
- A system failure occurs when the system no longer delivers a service consistent with its specification.
- Such a failure is observable by the system's users—either humans or other systems.

Availability General Scenarios

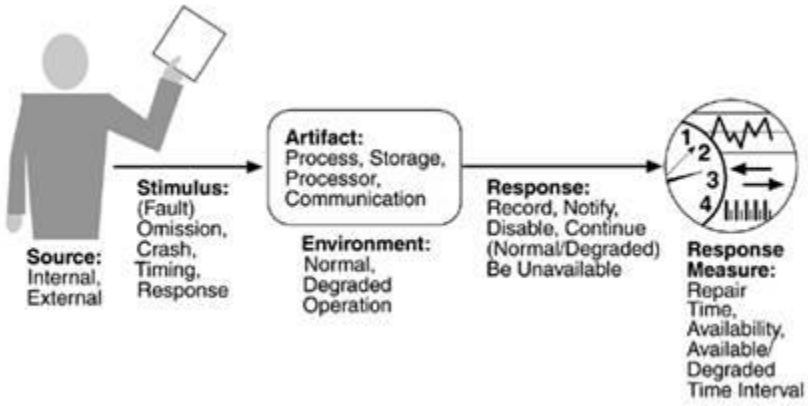
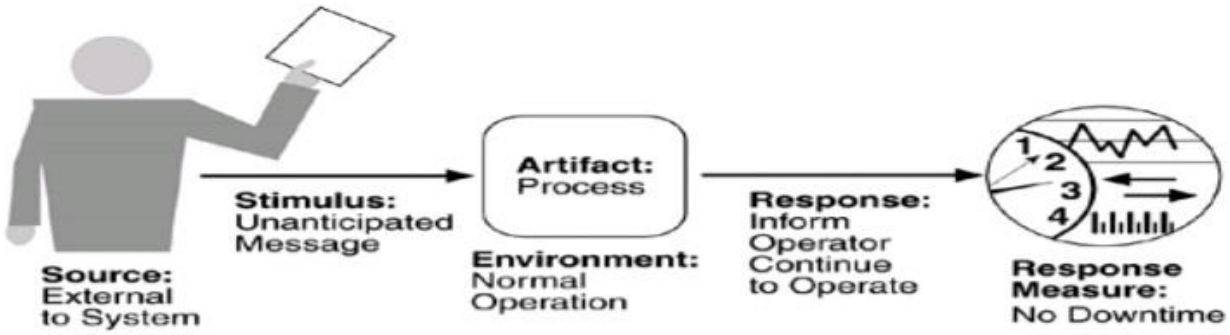



Fig. Sample availability scenario

Example: Identify the Availability quality attribute scenario from the following: - An unanticipated external message is received by a process during normal operation. The process

informs the operator of the receipt of the message and continues to operate with no downtime.




**Modifiability**

Modifiability is about the cost of change.

For example, a modifiability general scenario is: The platform on which the system depends is changed. The system must be modified to continue to provide current functionality. The platform change may be a change in hardware including input and output hardware, it could be a change in operating system or it could be a change in COTS middleware included in the system. Existing functionality of the system should remain unchanged.

Modifiability General Scenario Generation

| Portion of Scenario | Possible Values |
|---|---|
| Source | End user, developer, system administrator |
| Stimulus | Wishes to add/delete/modify/vary functionality, quality attribute, capacity |
| Artifact | System user interface, platform, environment; system that interoperates with target system |
| Environment | At runtime, compile time, build time, design time |
| Response | Locates places in architecture to be modified; makes modification without affecting other functionality; tests modification; deploys modification |

Modifiability General Scenario Generation

| Portion of Scenario | Possible Values |
|---|---|
| Response Measure | Cost in terms of number of elements affected, effort, money; extent to which this affects other functions or quality attributes |

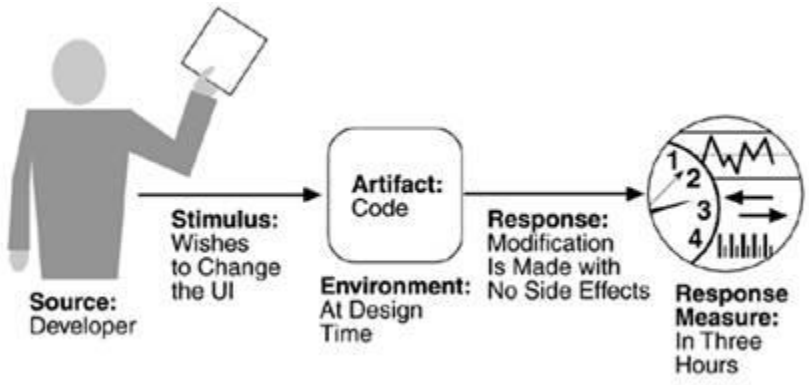



Fig. Sample modifiability scenario

**Performance**

- Performance is about timing. Events (interrupts, messages, requests from users, or the passage of time) occur, and the system must respond to them.
- One of the things that make performance complicated is the number of event sources and arrival patterns.
- A performance scenario begins with a request for some service arriving at the system. Satisfying the request requires resources to be consumed. While this is happening, the system may be simultaneously servicing other requests.
- An arrival pattern for events may be characterized as either periodic or stochastic.

Performance General Scenario Generation

| Portion of Scenario | Possible Values |
|---|---|
| Source | One of a number of independent sources, possibly from within system |
| Stimulus | Periodic events arrive; sporadic events arrive; stochastic events arrive |

Performance General Scenario Generation

| Portion of Scenario | Possible Values |
|---|---|
| Artifact | System |
| Environment | Normal mode; overload mode |
| Response | Processes stimuli; changes level of service |
| Response Measure | Latency, deadline, throughput, jitter, miss rate, data loss |

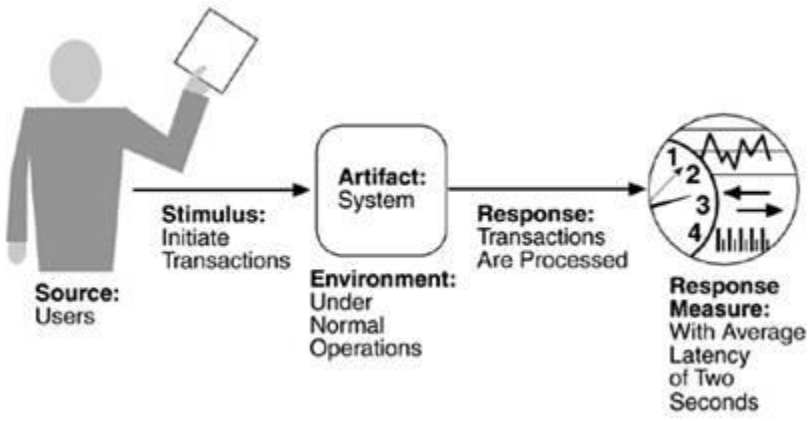



Fig: Sample performance scenario

Example: User initiates 1,000 transactions per minute stochastically under normal operations, and these transactions are processed with an average latency of two seconds.

- Performance is about timing.
- Events (interrupts, messages, requests from users, or the passage of time) occur, and the system must respond to them.
- There are a variety of characterizations of event arrival and response but basically performance is concerned with how long it takes the system to respond when an event occurs.
- Stimulus. The stimuli are the event arrivals. The arrival pattern can be characterized as periodic, stochastic/random, or sporadic/irregular. In our example, the stimulus is the stochastic initiation of 1,000 transactions per minute.
- Artifact. The artifact is always the system's services, as it is in our example.

- Environment. The system can be in various operational modes, such as normal, emergency, or overload. In our example, the system is in normal mode.
- Response. The system must process the arriving events. This may cause a change in the system environment (e.g., from normal to overload mode). In our example, the transactions are processed.
- Response measure. The response measures are the time it takes to process the arriving events (latency or a deadline by which the event must be processed), the variation in this time (jitter), the number of events that can be processed within a particular time interval (throughput), or a characterization of the events that cannot be processed (miss rate, data loss). In our example, the transactions should be processed with an average latency of two seconds.

**Security**

- Security is a measure of the system's ability to resist unauthorized usage while still providing its services to legitimate users.
- Security can be characterized as a system providing nonrepudiation, confidentiality, integrity, assurance, availability, and auditing.

1. Nonrepudiation is the property that a transaction (access to or modification of data or services) cannot be denied by any of the parties to it.
2. Confidentiality is the property that data or services are protected from unauthorized access.
3. Integrity is the property that data or services are being delivered as intended.
4. Assurance is the property that the parties to a transaction are who they purport to be.
5. Availability is the property that the system will be available for legitimate use.
6. Auditing is the property that the system tracks activities within it at levels sufficient to reconstruct them.

Security General Scenario Generation

| Portion of Scenario | Possible Values |
|---|---|
| Source | Individual or system that is correctly identified, identified incorrectly, of unknown identity who is internal/external, authorized/not authorized with access to limited resources, vast resources |
| Stimulus | Tries to display data, change/delete data, access system services, reduce availability to system services |
| Artifact | System services; data within system |
| Environment | Either online or offline, connected or disconnected, firewalled or open |
| Response | Authenticates user; hides identity of the user; blocks access to data and/or services; allows access to data and/or services; grants or withdraws permission to access data and/or services; records access/modifications or attempts to access/modify data/services by identity; stores data in an unreadable format; recognizes an unexplainable high demand for services, and informs a user or another system, and restricts availability of services |
| Response Measure | Time/effort/resources required to circumvent security measures with probability of success; probability of detecting attack; probability of identifying individual responsible for attack or access/modification of data and/or services; percentage of services still available under denial-of-services attack; restore data/services; extent to which data/services damaged and/or legitimate access denied |

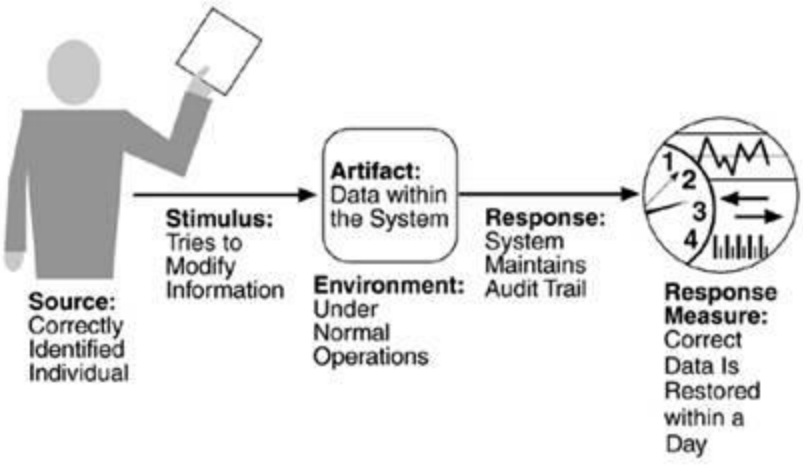



Figure: Shows the Parts of Security Scenario

**4.2 Business** Qualities

Time to market

- If there is competitive pressure or a short window of opportunity for a system or product, development time becomes important. This in turn leads to pressure to buy or otherwise re-use existing elements.

Cost and benefit

- The development effort will naturally have a budget that must not be exceeded. Different architectures will yield different development costs. For instance, an architecture that relies on technology (or expertise with a technology) not resident in the developing organization will be more expensive to realize than one that takes advantage of assets already in-house.
- An architecture that is highly flexible will typically be more costly to build than one that is rigid (although it will be less costly to maintain and modify).

Projected lifetime of the system

- If the system is intended to have a long lifetime, modifiability, scalability, and portability become important. On the other hand, a modifiable, extensible product is more likely to survive longer in the marketplace, extending its lifetime.

Targeted market

- For general-purpose (mass-market) software, the platforms on which a system runs as well as its feature set will determine the size of the potential market. Thus, portability and

functionality are key to market share. Other qualities, such as performance, reliability, and usability also play a role.

Rollout schedule

- If a product is to be introduced as base functionality with many features released later, the flexibility and customizability of the architecture are important. Particularly, the system must be constructed with ease of expansion and contraction in mind.

  Integration with legacy systems.
- If the new system has to integrate with existing systems, care must be taken to define appropriate integration mechanisms. This property is clearly of marketing importance but has substantial architectural implications.

### 4.3 Architecture Qualities

- Conceptual integrity is the underlying theme or vision that unifies the design of the system at all levels. The architecture should do similar things in similar ways.
- Correctness and completeness are essential for the architecture to allow for all of the system's requirements and runtime resource constraints to be met. A formal evaluation is the architect's best hope for a correct and complete architecture.
- Buildability allows the system to be completed by the available team in a timely manner and to be open to certain changes as development progresses.

Summary

Architects must expend a lot of effort in precisely understanding quality attributes so that a design can be conceived to address them. That’s why an architect must be associated with the requirements gathering exercise for the system. Understanding the quality attribute requirements is merely a necessary prerequisite to designing a solution to satisfy them. Creating solutions that choose a point in the design space that adequately satisfies these requirements is remarkably difficult, both technically and socially. The latter involves communications with stakeholders to discuss design tolerances, discovering scenarios when certain quality requirements can be safely relaxed, and clearly communicating design

**SELF-CHECK**

1. Which one of the following is **not** business qualities?
   A. Scalability
   B. Time to market
   C. Cost and benefit
   D. Projected lifetime of the system
   E. Targeted market
2. When the user presses the green button, the options dialog appears" this is an example of?
   A. Quality requirement
   B. None functional requirement
   C. Functional requirement
   D. None
3. An availability tactic that used to detect faults
   A. State resynchronization
   B. Sanity checking
   C. Exception handling
   D. All
4. Which one of the following is NOT true?
   A. Achieving quality attributes considered throughout design, implementation, and deployment.
   B. Architecture is critical to the realization of many qualities of interest in a system
   C. The achievement of quality attributes will always have a positive effect
   D. Flexibility of the architecture is important for a software product to be introduced as base functionality with many features released later.
5. The quality attributes can be calculated under which of the following measures?
   A. Observable
   B. Non observable
   C. All of the mentioned
   D. None
6. Which of the following is a correct statement?
   A. A highly modifiable system produces correct results
   B. A highly modifiable system does not produce correct results
   C. A highly modifiable system may or may not produces correct results
   D. None
7. Which of the following is considered incorrect with respect to the quality?
   A. Architecture is critical to the realization of many of the qualities of interest in a system.
   B. All qualities are architecturally sensitive

C. Architecture provides the foundation for achieving quality.

D. Quality attribute requirements are part of an application's nonfunctional requirements.

8. What can be stated about modifiability?
    A. Modifiability cannot be considered largely architectural
    B. Modifiability can never be determined by how functionality is divided
    C. Modifiability is about the cost of change.
    D. All of the mentioned
    E. None
9. What can be stated with regards to performance?
    A. Performance is an example of architectural and non-architectural dependencies
    B. Performance depends partially on how much communication is necessary between the components
    C. All of the mentioned
    D. None
10. What is the main technique for achieving portable software?
    A. The main technique for achieving portable software is to isolate system dependency.
    B. The main technique for achieving portable software is to increase the overall performance.
    C. The main technique for achieving portable software is to have independent platform dependency.
    D. None
11. By what methods quality attributes can be judged?
    A. Qualities that can be discerned by observing the system execute
    B. Qualities directly attributable to the system.
    C. Qualities about the architecture itself that are important
    D. All
12. The main concern of performance quality attributes is

    A. Fault recovery
    C. Resource management
    B. Latency
    D. Reducing coupling

## EXERCISE

1. What are quality attribute scenarios?
2. What are quality attributes of a good software explain those along with examples?
3. What is the relationship between a use case and a quality attribute scenario?
4. What are the main categories of quality attributes?
5. How do you identify critical quality attributes?
6. Why software quality attributes are important?
7. What are five of the most important attributes of software quality?
8. From the following identify the general software quality attribute that is addressed and the quality attribute scenarios. The e-commerce store's virtual space, including its servers, databases, and website, experiencing downtime because of server hardware failure. The developer tries to bring the servers back online as soon as possible to customers by repairing or replacing the faulty hardware.

# Chapter Five

# Architecture In the Life Cycle

Unit Objectives

At the end of this chapter the students will be able to:

- Understand meaning architecture in the agile projects.
- Understand Architecture and requirements
- Understand designing and documentation

## 5.1 Architecture in the Agile Projects

Software architectural design is the process of applying various techniques and principle for the purpose of defining a module, a process, or a system in sufficient detail to permit its physical coding. The conventional approach to the software design process focuses on partitioning a problem and its solution into detailed pieces up front before proceeding to the construction phase. These up-front software architecture efforts are critical and leave no room to accommodate changing requirements later in the development cycle. Some of the issues faced by organizations involved in up front software design efforts are:

- Requirements evolve over time due to changes in customer and user needs, technological advancement and schedule constraints.
- Changes to requirements systematically involves modifying the software design, and in turn, the code.
- Accommodating changing software design is an expensive critical activity in the face of rapidly changing requirements.
- Clear specification of activities in the agile software design process is missing and there is a lack of a set of techniques for practitioners to choose from.

### 5.1.1 Agile Development Methods

- The goal of agile methods is to allow an organization to be agile, but what does it mean to be Agile. Agile means being able to “Deliver quickly”; “Change quickly and often”.
- While agile techniques vary in practices and emphasis, they follow the same principles behind the agile manifesto.

- Working software is delivered frequently (weeks rather than months).
- Working software is the principal measure of progress.
- Customer satisfaction by rapid, continuous delivery of useful software.
- Late changes in software requirements are accepted.
- Close daily cooperation between business people and software developers.
- Face-to-face conversation is the best form of communication.
- Projects are built around motivated individuals who should be trusted.
- Continuous attention to technical excellence and good design.

### 5.1.2 Software Architectural Design in Agile Environments

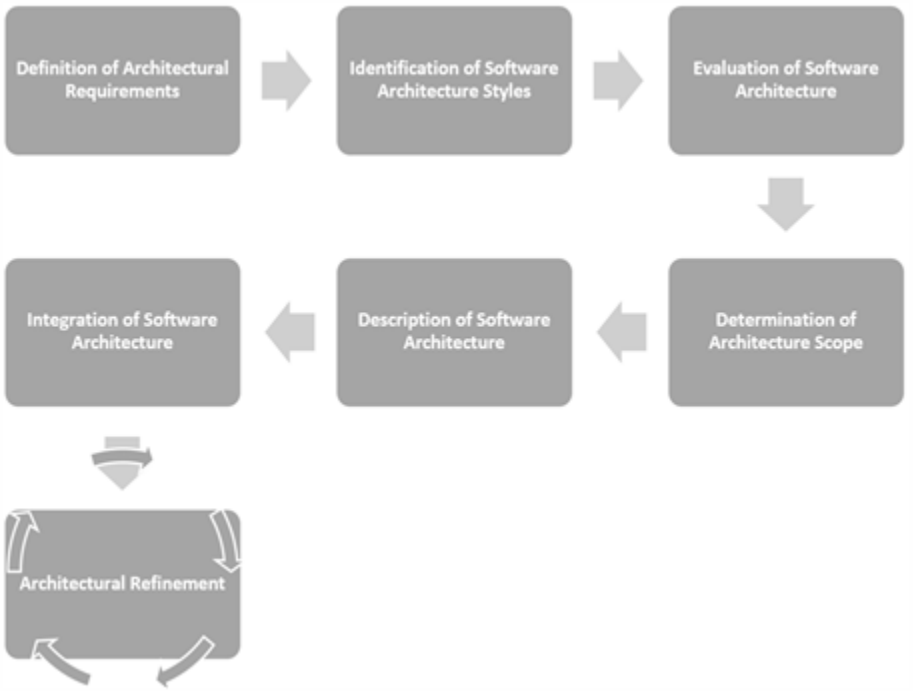



Fig: Software design methodology in agile environment.

**Step 1: Definition of Architectural Requirements**

Establishing the driving architectural requirements: Driving architectural requirements are obtained by analyzing the business drivers and system context as well as the issues deemed critical to system success by the product stakeholders. The goal is a specification for the architecture directing the architects to create a structure for the system that is sufficient to ensure success in the

eyes of the stakeholders. These requirements prevent creation of an architecture that is overly complex or that strives for unnecessary elegance at the expense of critical system properties.

**Step 2: Identification of Software Architecture Styles**

Architectural structures and coordination strategies are developed to satisfy the driving architectural requirements. Alternative architecture solutions may be proposed and analyzed to identify an optimal solution for the product or product line being developed. When product lines are involved, adaptation of the product line architecture to specific product requirements or fully develop the architecture for an individual product. The identification of software architecture styles aims to precise the associated elements, forms, and rationales

**Step 3: Evaluation of Software Architecture**

Software architecture evaluation determines when and what methods of architecture evaluation are appropriate. The results of such evaluation are then analyzed and measures are determined and applied to improve the developing architecture. A formal software architecture evaluation should be a standard part of our software architecture methodology in agile environments. Software architecture evaluation is a cost-effective way of mitigating the substantial risks associated with this highly important artifact. The achievement of a software system's quality attribute depends much more on the software architecture than on code-related issues such as language choice, fine-grained design, algorithms, data structures, testing, and so forth. Most complex software systems are required to be modifiable and have good performance. They may also need to be secure, interoperable, portable, and reliable. Several software architecture evaluation methods can be used like Architecture Tradeoff Analysis Method, Software Architecture Analysis Method, Active Reviews for Intermediate Designs.

**Step 4: Determination of Architecture Scope**

Before defining an architecture, the developers determine how many of the system-design decisions should be established by the architecture of the system. This scope delimits the activities of application developers, allowing them to concentrate on what they do best. Software architecture scope is a reflection of system requirements and trade-offs that made to satisfy them.

**Step 5: Description of Software Architecture**

An architecture must be described in sufficient detail and in an easily accessible form for developers and other stakeholders. The architecture is one of the major mechanisms that allow stakeholders to communicate about the properties of a system. Architecture documentation

determines what views of software are useful for the stakeholders, the amount of detail required, and how to present the information efficiently. Agile methods agree strongly on a central point: "If information is not needed, do not document it". All documentation should have an intended use and audience in mind, and be produced in a way that serves both. One of the fundamental principles of technical documentation is "Write for the reader". Another central idea to remember is that documentation is not a monolithic activity that holds up all other progress until it is complete. With that in mind, the following is the suggested approach for describing software architecture using agile-like principles:

- Create a skeleton document (document outline) for a comprehensive view-based software architecture document using the standard organization schemes;
- Decide which architectural views should be to produced, given the software architecture scope (step 4) with respect to available resources.

**Step 6: Integration of Software Architecture**

The software architecture integration process is a set of procedures used to combine software architectural components into larger components, subsystems or final software architecture. Software architecture integration enables the organization to observe all important attributes that a software will have; functionality, quality and performance. This is especially true for software systems as the integration is the first occurrence where the full result of the software development effort can be observed. Consequently, the integration activities represent a highly critical part of the software development process in agile environments. Usually, Architecture Analysis and Design Language are used in order to build integrated software-reliant systems

**Step 7: Continuous Architectural Refinement**

Architectural refinement aims to help provide the degree of architectural stability required to support the next iterations of development. This stability is particularly important to the successful operation of multiple parallel Scrum teams. Making architectural dependencies visible allows them to be managed and for teams to be aligned with them. The architecture refinement supports the team decoupling necessary to allow independent decision-making and reduce communication and coordination overhead. During the preparation phase, agile teams identify an architecture style of infrastructure sufficient to support the development of features in the near future. Product development using an architectural refinement most likely occurs in the preservation phase. Architectural refinement is one of the key factors to successfully scale agile. Describing and

maintaining (through refinement) software architectural design enables a system infrastructure sufficient to allow incorporation of near-term high-priority features from the product backlog. The proposed software architecture methodology in agile environments allows the software architecture and design to support the features without potentially creating unanticipated rework by destabilizing refactoring. Larger software systems (and teams) need longer architectural refinements. Building and re-architecting software takes longer than a single iteration or release cycle. Delivery of planned functionality is more predictable when the architecture for the new features is already in place. This requires looking ahead in the planning process and investing in architecture by including design work in the present iteration that will support future features and customer needs. The architectural refinement is not complete. The refinement process intentionally is not complete because of an uncertain future with changing technology orientations and requirement engineering. This requires continuously extending the architectural refinement to support the development teams.

### 5.2 Architecture and requirements

Architecturally significant requirements are those requirements that have a measurable effect on system's architecture. This can comprise both software and hardware requirements. They are a subset of requirements, the subset that affects the architecture of a system in measurably identifiable ways.

Architecturally significant requirements can be characterized from the following aspects: -

**Descriptive characteristics:** Architecturally significant requirements are often hard to define and articulate, tend to be expressed vaguely, tend to be initially neglected, tend to be hidden within other requirements, and are subjective, variable, and situational. Other requirements could also demonstrate these descriptive characteristics. However, architecturally significant requirements' significance made these manifestations unique and challenging.

**Indicators:** A requirement that has wide effect, targets trade-off points, is strict (constraining, limiting, non-negotiable), assumption breaking, or difficult to achieve is likely to be architecturally significant.

Indicators for architectural significance that have been reported in the literature include:

- The requirement is associated with high business value and/or technical risk.
- The requirement is a concern of a particularly influential stakeholder.

- The requirement has a first-of-a-kind character, e.g., none of the responsibilities of already existing components in the architecture addresses it.
- The requirement has QoS/SLA characteristics that deviate from all ones that are already satisfied by the evolving architecture.
- The requirement has caused budget overruns or client dissatisfaction in a previous project with a similar context.
- Like all non-functional requirements and quality attribute requirements, architecturally significant requirements should be specified in a SMART way.

**Software Architecture Activities**

Software architecture is comprised of a number of specific architecting activities (covering the entire architectural lifecycle) and a number of general architecting activities (supporting the specific activities). The specific software architecture activities are composed of five items:

- Architectural Analysis (AA) defines the problems an architecture must solve. The outcome of this activity is a set of architecturally significant requirements (ASRs).
- Architectural Synthesis (AS) proposes candidate architecture solutions to address the ASRs collected in AA, thus this activity moves from the problem to the solution space.
- Architectural Evaluation (AE) ensures that the architectural design decisions made are the right ones, and the candidate architectural solutions proposed in AS are measured against the ASRs collected in AA.
- Architectural Implementation (AI) realizes the architecture by creating a detailed design.
- Architectural Maintenance and Evolution (AME) is to change an architecture for the purpose of fixing faults and architectural evolution is to respond to new requirements at the architectural level.

**5.3 Designing and Documentation**

There are many good reasons why we want to document our architectures, for example:

- Others can understand and evaluate the design. This includes any of the application stakeholders, but most commonly other members of the design and development team.
- We can understand the design when we return to it after a period of time.
- Others in the project team and development organization can learn from the architecture by digesting the thinking behind the design.

- We can do an analysis of the design, perhaps to assess its likely performance, or to generate standard metrics like coupling and cohesion.

Documenting architectures is problematic though, because:

- There's no universally accepted architecture documentation standard.
- Architecture can be complex, and documenting it in a comprehensible manner is time-consuming and non-trivial.
- Architecture has many possible views. Documenting all the potentially useful ones is time-consuming and expensive.
- An architecture design often evolves as the system is incrementally developed and more insights into the problem domain are gained. Keeping the architecture documents current is often an overlooked activity, especially with time and schedule pressures in a project

Probably the most crucial element of the "what to document" equation is the complexity of the architecture being designed. A two-tier client-server application with complex business logic may actually be quite simple architecturally. It might require no more than an overall "architecture" diagram describing the main components, and perhaps a structural view of the major components (maybe it uses a model-view-controller architecture) and a description of the database schema, no doubt generated automatically by database tools. This level of documentation is quick to produce and routine to describe. So architectural documentation becomes of vital importance for describing design elements such as:

- Component interfaces;
- Subsystems constraints;
- Test scenarios;
- Third-party component purchasing decisions;
- Team structure and schedule dependencies
- External services to be offered by the application.

Architecture documentation is both prescriptive and descriptive and user to produce different documents for different stakeholders, write from the point of view of the reader.

The C4 model was introduced by Simon Brown, and it's the best idea about software architecture documentation. The idea is to use 4 different granularity (or zoom) levels for documenting software architecture:

Level 1: System Context diagram

Level 2: Container diagram

Level 3: Component diagram

Level 4: Code diagram

Level 1: System Context diagram

This is the highest granularity diagram. It has little detail but its main goal is to describe the context in which the application is. So, it will be composed by one single box for the whole application, and it will be surrounded by other boxes that refer to the external systems and users the application interacts with.

Level 2: Container diagram

At this level of granularity, we will see the containers of the application**,** where a container is any independent technical piece of the application, for example a mobile app, an API or a database. It also documents the major technologies used and how the containers communicate**.**

Level 3: Component diagram

The component diagram shows us the components inside one container. In this context, each component is a module of the application, not restricted to domain wise modules, but also including purely functional modules.

Level 4: Code

The most fine-grained diagram, aimed at describing the code structure inside a component**.** For this level, we use an UML diagram with class level artefacts.

### 5.3.1 Types of Technical documentation

Technical documentation in software engineering is the umbrella term that encompasses all written documents and materials dealing with software product development. All software development products, whether created by a small team or a large corporation, require some related documentation. And different types of documents are created through the whole software development lifecycle (SDLC).

The main goal of effective documentation is to ensure that developers and stakeholders are headed in the same direction to accomplish the objectives of the project. To achieve them, plenty of documentation types exist.

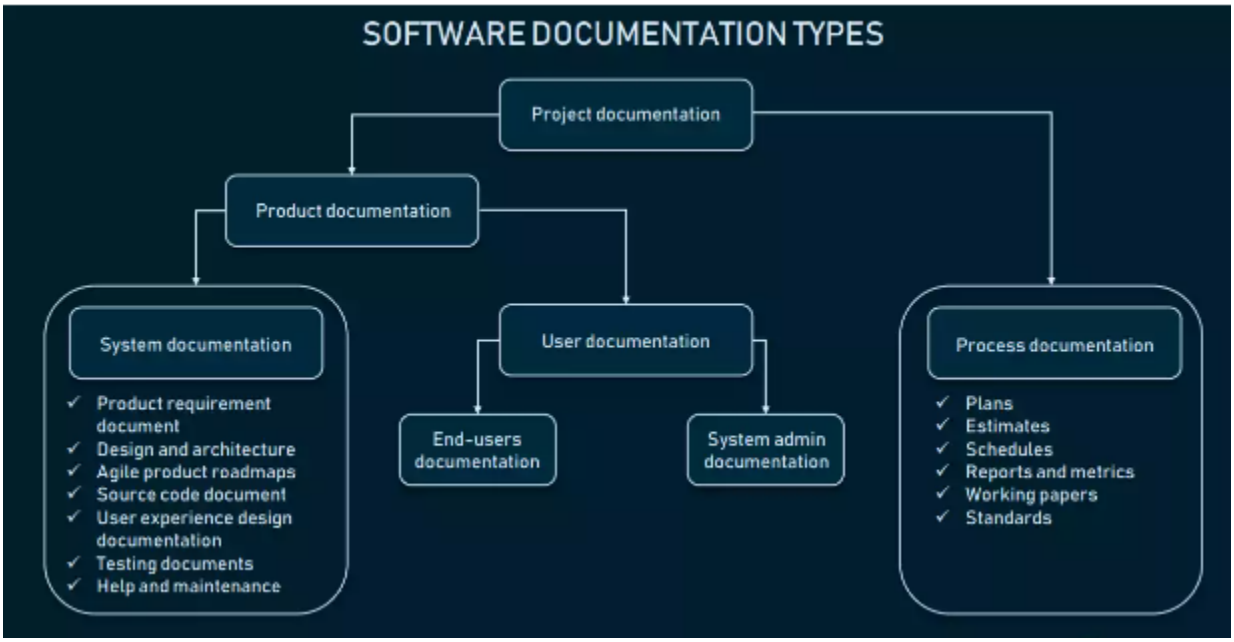



Fig: Software documentation classification

All software documentation can be divided into **two main categories:**

- Product documentation
- Process documentation

**Product documentation** describes the product that is being developed and provides instructions on how to perform various tasks with it. In general, product documentation includes requirements, tech specifications, business logic, and manuals. There are two main types of product documentation:

- System documentation represents documents that describe the system itself and its parts. It includes requirements documents, design decisions, architecture descriptions, program source code, …...
- User documentation covers manuals that are mainly prepared for end-users of the product and system administrators. User documentation includes tutorials, user guides, troubleshooting manuals, installation, and reference manuals.

**Process documentation** represents all documents produced during development and maintenance that describe… well, the process. The common examples of process-related documents are standards, project documentation, such as project plans, test schedules, reports, meeting notes, or even business correspondence. The main difference between process and product documentation

is that the first one records the process of development and the second one describes the product that is being developed.

**Software architecture design document**

It is one of system documentation under product documentation of software documentation, sometimes also called technical specifications, include the main architectural decisions made by the solution architect. Unlike the product requirement document mentioned above that describes what needs to be built, the architecture design documentation is about how to build it. It has to describe in what way each product component will contribute to and meet the requirements, including solutions, strategies, and methods to achieve that. So, the software design document gives an overview of the product architecture, determines the full scope of work, and sets the milestones, thus, looping in all the team members involved and providing the overall guidance. An effective design and architecture document comprises the following information sections:

**Overview and background.** Briefly describe the main goals of the project, what problems you are trying to solve, and the results you want to achieve.

**Architecture & Design Principles**. Underline the guiding architecture and design principles with which you will engineer the product. For instance, if you plan to structure your solution using microservices architecture, don't forget to specifically mention this.

**User Story description.** Connect user stories with associated business processes and related scenarios. You should try to avoid technical details in this section.

**Solution details.** Describe the contemplated solution by listing planned services, modules, components, and their importance.

**Diagrammatic representation of the solution.** Provide the diagrams and/or other graphic materials to help understand and communicate the structure and design principles.

**Milestones**. Include the overall timeline, deadlines for completion, and/or functional milestones, i.e., independent modules of the application developed. That will help organize the work process and provide a clear metric to monitor progress. This section can be very brief as it's closely related to the process documentation described below.

## 5.4 Architecture in Advance

### 5.4.1 Cloud Definition

Cloud architecture is a key element of building in the cloud. It refers to the layout and connects all the necessary components and technologies required for cloud computing. Cloud architecture is the way technology components combine to build a cloud, in which resources are pooled through virtualization technology and shared across a network. Cloud computing architecture is a combination of service-oriented architecture and event-driven architecture. Migrating to the cloud can offer many business benefits compared to on-premises environments, from improved agility and scalability to cost efficiency. While many organizations may start with a "lift-and-shift" approach, where on-premises applications are moved over with minimal modifications, ultimately it will be necessary to construct and deploy applications according to the needs and requirements of cloud environments.

Cloud architecture dictates how components are integrated so that you can pool, share, and scale resources over a network. Think of it as a building blueprint for running and deploying applications in cloud environments. Cloud architecture refers to how various cloud technology components, such as hardware, virtual resources, software capabilities, and virtual network systems interact and connect to create cloud computing environments. It acts as a blueprint that defines the best way to strategically combine resources to build a cloud environment for a specific business need. The components of a cloud architecture include:

- A front-end platform (the client or device used to access the cloud)
- A back-end platform (servers and storage)
- A cloud-based delivery model
- A network (internet, intranet, or intercloud)

In cloud computing, frontend platforms contain the client infrastructure-user interfaces, client-side applications, and the client device or network that enables users to interact with and access cloud computing services. For example, you can open the web browser on your mobile phone and edit a Google Doc. All three of these things describe frontend cloud architecture components.

- On the other hand, the back end refers to the cloud architecture components that make up the cloud itself, including computing resources, storage, security mechanisms, management, and more.
- Below is a list of the main backend components:

- **Application:** The backend software or application the client is accessing from the front end to coordinate or fulfill client requests and requirements.
- **Service:** The service is the heart of cloud architecture, taking care of all the tasks being run on a cloud computing system. It manages which resources you can access, including storage, application development environments, and web applications.
- **Runtime cloud:** Runtime cloud provides the environment where services are run, acting as an operating system that handles the execution of service tasks and management. Runtimes use virtualization technology to create hypervisors that represent all your services, including apps, servers, storage, and networking.
- **Storage:** The storage component in the back end is where data to operate applications is stored. While cloud storage options vary by provider, most cloud service providers offer flexible scalable storage services that are designed to store and manage vast amounts of data in the cloud. Storage may include hard drives, solid-state drives, or persistent disks in server bays.
- **Infrastructure:** Infrastructure is probably the most commonly known component of cloud architecture. In fact, you might have thought that cloud infrastructure *is* cloud architecture. However, cloud infrastructure comprises all the major hardware components that power cloud services, including the CPU, graphics processing unit (GPU), network devices, and other hardware components needed for systems to run smoothly. Infrastructure also refers to all the software needed to run and manage everything.
- Cloud architecture, on the other hand, is the plan that dictates how cloud resources and infrastructure are organized.
- **Management:** Cloud service models require that resources be managed in real time according to user requirements. It is essential to use management software, also known as middleware, to coordinate communication between the backend and frontend cloud architecture components and allocate resources for specific tasks. Beyond middleware, management software will also include capabilities for usage monitoring, data integration, application deployment, and disaster recovery.
- **Security:** As more organizations continue to adopt cloud computing, implementing cloud security features and tools is critical to securing data, applications, and platforms.

It’s essential to plan and design data security and network security to provide visibility, prevent data loss and downtime, and ensure redundancy. This may include regular backups, debugging, and virtual firewalls.

All these technologies create a cloud computing architecture on which applications can run, providing end-users with the ability to leverage the power of cloud resources. Cloud computing architecture enables organizations to reduce or eliminate their reliance on on-premises server, storage, and networking infrastructure. Organizations adopting cloud architecture often shift IT resources to the public cloud, eliminating the need for on-premises servers and storage, and reducing the need for IT data center real estate, cooling, and power, and replacing them with a monthly IT expenditure.

## 5.4.2 Cloud architecture layers

A simpler way of understanding how cloud architecture works is to think of all these components as various layers placed on top of each other to create a cloud platform. Basic cloud architecture layers are: -

1. Hardware: The servers, storage, network devices, and other hardware that power the cloud.
2. Virtualization: An abstraction layer that creates a virtual representation of physical computing and storage resources. This allows multiple applications to use the same resources.
3. Application and service: This layer coordinates and supports requests from the frontend user interface, offering different services based on the cloud service model, from resource allocation to application development tools to web-based applications.

### 5.4.3 Benefits of cloud architecture

Reasons to adopt a cloud architecture are:

- Accelerate the delivery of new apps
- Take advantage of cloud-native architecture such as Kubernetes to modernize applications and accelerate digital transformation.
- Ensure compliance with the latest regulations
- Deliver greater transparency into resources to cut costs and prevent data breaches

- Enable faster provisioning of resources
- Utilize hybrid cloud architecture to support real-time scalability for applications as business needs change
- Meet service targets consistently
- Leverage cloud reference architecture to gain insight into IT spending patterns and cloud utilization

The fundamental components of cloud architecture include:

- **Virtualization**: Clouds are built upon virtualization of servers, storage, and networks. Virtualized resources are a software-based, or virtual, representation of a physical resource such as servers or storage. This abstraction layer enables multiple applications to utilize the same physical resources, thereby increasing the efficiency of servers, storage, and networking throughout the enterprise.
- **Infrastructure**: Yes, there are real servers. Cloud infrastructure includes all the components of traditional data centers including servers, persistent storage and networking gear including routers and switches.
- **Middleware**: As in traditional data centers, these software components such as databases and communications applications enable networked computers, applications and software to communicate with each other.
- **Management**: These tools enable continuous monitoring of a cloud environment's performance and capacity. IT teams can track usage, deploy new apps, integrate data and ensure disaster recovery, all from a single console.
- **Automation software**: The delivery of critical IT services through automation and pre-defined policies can significantly ease IT workloads, streamline application delivery, and reduce costs. In a cloud architecture, automation is used to easily scale up system resources to accommodate a spike in demand for compute power, deploy applications to meet fluctuating market demands, or ensure governance across a cloud environment.

### 5.4.4 Cloud Architect

A cloud architect is an IT expert responsible for developing, implementing, and managing an organization's cloud architecture. As cloud strategies continue to become more complex, the skills and expertise of cloud architects are becoming more vital for helping companies navigate the

complexities of cloud environments, implement successful strategies, and keep cloud systems running smoothly.

## SELF-CHECK

1. _____is concerned with identifying a candidate and constrained architecture?
   A. Architecture analysis
   B. Architecture design
   C. Architecture pattern
   D. Architecture rules
2. What do you mean by software requirements?
   A. Software requirement is what software is
   B. Software requirement is a condition needed by a user to solve a problem or achieve an objective
   C. Software requirement is a representation of a condition of software
   D. All of the above
3. _____ is the process of applying various techniques and principle for the purpose of defining a module, a process, or a system in sufficient detail to permit its physical coding.
   A. Architecture rules
   B. Software architectural design
   C. User interface design
   D. Architecture style
4. Which one of the following cloud concepts is related to sharing and pooling the resources?
   A. Polymorphism
   B. Virtualization
   C. Abstraction
   D. None of the mentioned
5. What is agile methodology?
   A. Agile methodology is a sequential approach to software development.
   B. Agile methodology is an iterative approach to software development.
   C. Agile methodology is a circular approach to software development.
   D. Agile methodology is a prototype approach to software development
6. Which of the following consist of the rapid explanation of all the functionalities desired in the product?
   A. User manual
   B. Documentation
   C. Product backlog
   D. All of the above

7. _____is a framework for developing a computer system that is defined by assembling hardware and software components and defining their interfaces.
   A. Architecture style
   B. Architecture design
   C. Architecture pattern
   D. All
8. In which one of the following phases, IT Architecture Development came?
   A. Strategy Phase
   B. Planning Phase
   C. Deployment Phase
   D. Development Phase
9. Cloud computing architecture is a combination of:
   A. service-oriented architecture and grid computing
   B. Utility computing and event-driven architecture.
   C. Service-oriented architecture and event-driven architecture.
   D. Virtualization and event-driven architecture.
10. Through which, the backend and front-end are connected with each other?
   A. Browser
   B. Database
   C. Network
   D. A and B
11. Which of the following provides the Graphic User Interface (GUI) for interaction with the cloud?

A. Client
B. Client Infrastructure
C. Application
D. Server

## EXERCISE

1. What are the characteristics of architecturally significant requirements?
2. What does architecturally significant mean?
3. What are the requirements with architecture?
4. How do you document software architecture?
5. List elements of software architecture documentation?
6. What are the challenges of cloud architecture?
7. What are steps in software architectural design in agile environments?
8. Discus C4 different model (or zoom) levels for documenting software architecture?

9. Why adopt cloud architecture?
10. What are the five important components required by the cloud architecture?

**References**


1. Bachmann, F., Bass, L., Klein, M., 2003b. Moving from Quality Attribute Requirements to Architectural Decisions.
2. How do architecture patterns and tactics interact? A model and annotation.
3. https://doi.org/10.1016/j.jss.2010.04.067
4. Partha ,Kuchana, "Software architecture design patterns in Java", AUERBACH PUBLICATIONS, 2004.
5. Len Bass, Paul Clements, Rick Kazman, -Software Architecture in Practice,2nd edition Addison-Wesley, 2003.
6. Booch G, Rumbaugh J, Jacobson I, "The Unified Modeling Language User Guide", Addison-Wesley, 1999.
7. Taylor R.N, Medvidovic N, Dashofy E. M, "Software Architecture: Foundations, Theory, and Practice", Wiley, 2009
8. https://www.ou.nl/documents/40554/791670/IM0203_03.pdf/30dae517-691e-b3c7-22ed-a55ad27726d6
9. "Design Patterns in Java – Javatpoint." Www.javatpoint.com, Available here.
10. "Software Design." Wikipedia, Wikimedia Foundation, 11 Apr. 2019, Available here.
11. "Software Architecture." Wikipedia, Wikimedia Foundation, 5 Apr. 2019, Available here.
12. ."What Is Event-Driven Architecture (EDA)? – Definition from WhatIs.com." SearchMicroservices,
13. Len Bass, Paul Clements, Rick Kazman, ―Software Architecture in Practice‖, 3rd edition Pearson, 2013.
14. Mary Shaw, David Garlan, ―Software Architecture: Perspectives on an Emerging Discipline‖, Prentice Hall, 1996.
15. https://sourcemaking.com/design_patterns
16. https://www.cs.unb.ca/~wdu/cs6075w10/sa2.htm
17. Design Patterns: Elements of Reusable Object-Oriented Software
18. https://sourcemaking.com/design_patterns
19. https://www.ics.uci.edu/~andre/ics223w2006/kruchten3.pdf