Software Design

Ashima Tyagi
Assistant Professor
School of Computer Science & Engineering

Outline

- Software Design
- Software design concepts
- Effective modular design
- Cohesion and Coupling
- Architectural design
- Object Oriented Design
- User Interface Design
- Detailed Design Transaction Transformation
- Refactoring of designs

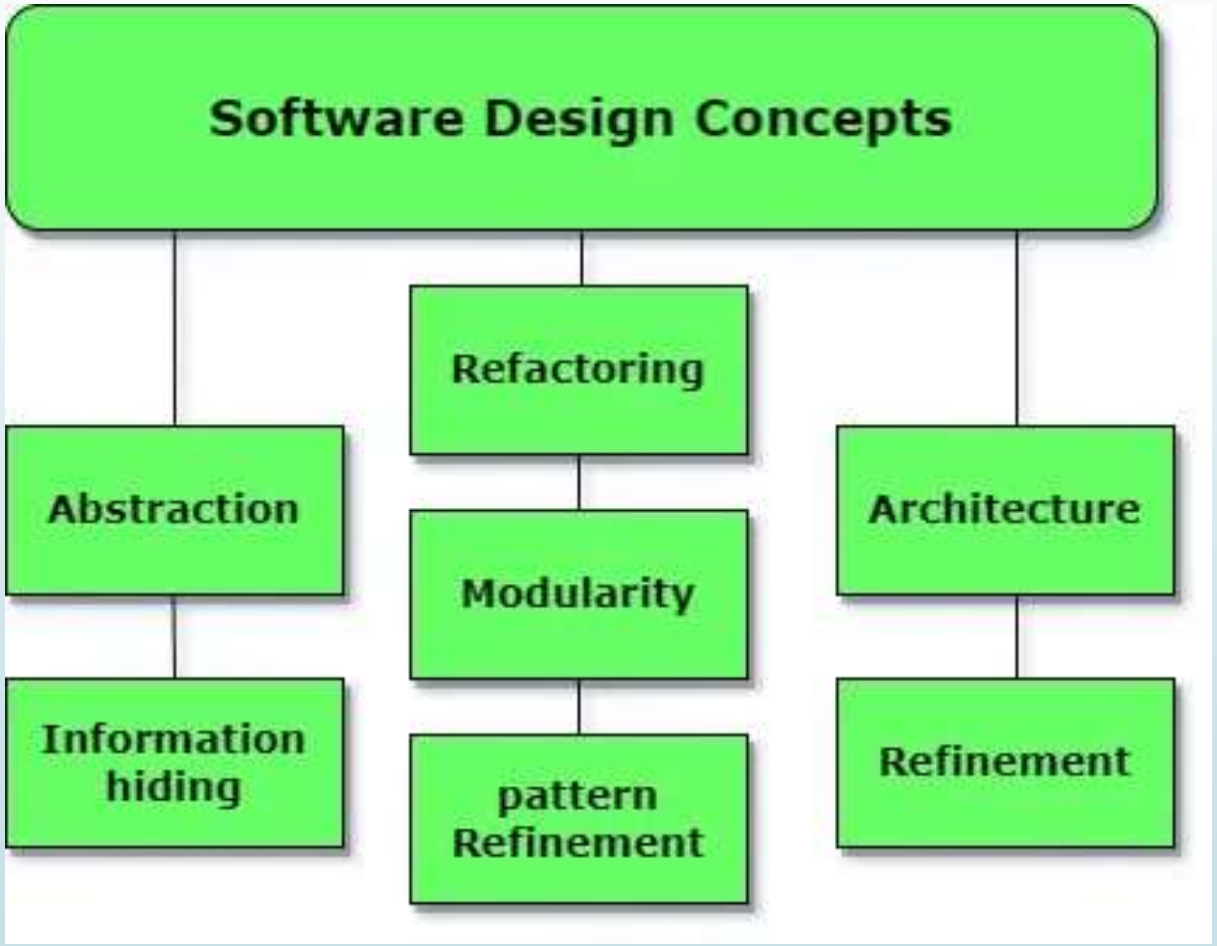
Software Design

The design phase of software development deals with transforming the customer requirements as described in the SRS documents into a form implementable using a programming language.

Software design challenges

1. Complexity
2. Maintaining Quality
3. Meeting Deadlines
4. Keeping up with new technologies
5. Managing changing requirements
6. Debugging
7. Balancing short-term and long-term goals

Software Design Concepts



Software design concepts

1. **Abstraction (Hide Irrelevant data):** Abstraction simply means to hide the details to reduce complexity and increase efficiency or quality. Abstraction is used to simplify complex systems by hiding unnecessary details and exposing only the relevant functionality. This makes software easier to develop, use, maintain, and scale. Abstraction allows developers to focus on what a module does rather than how it does it. By defining abstract methods or interfaces, the same code can be used in different scenarios without modification.
2. **Modularity (subdivide the system):** Modularity simply means dividing the system or project into smaller parts to reduce the complexity of the system or project. In the same way, modularity in design means subdividing a system into smaller parts so that these parts can be created independently and then use these parts in different systems to perform different functions. It is necessary to divide the software into components known as modules because nowadays, there are different software available like Monolithic software that is hard to grasp for software engineers. So, modularity in design has now become a trend and is also important. If the system contains fewer components then it would mean the system is complex which requires a lot of effort (cost) but if we can divide the system into components then the cost would be small.

3. **Architecture (design a structure of something)**: Architecture simply means a technique to design a structure of something. Architecture in designing software is a concept that focuses on various elements and the data of the structure. These components interact with each other and use the data of the structure in architecture.

4. **Refinement (removes impurities):** Refinement simply means to refine something to remove any impurities if present and increase the quality. The refinement concept of software design is a process of developing or presenting the software or system in a detailed manner which means elaborating a system or software. Refinement is very necessary to find out any error if present and then to reduce it.

5. **Pattern (a Repeated form):** A pattern simply means a repeated form or design in which the same shape is repeated several times to form a pattern. The pattern in the design process means the repetition of a solution to a common recurring problem within a certain context.

6. **Information Hiding (Hide the Information**): Information hiding simply means to hide the information so that it cannot be accessed by an unwanted party. In software design, information hiding is achieved by designing the modules in a manner that the information gathered or contained in one module is hidden and can't be accessed by any other modules.

7. **Refactoring (Reconstruct something):** Refactoring simply means reconstructing something in such a way that it does not affect the behavior of any other features. Refactoring in software design means reconstructing the design to reduce complexity and simplify it without impacting the behavior or its functions. Fowler has defined refactoring as "the process of changing a software system in a way that it won't impact the behavior of the design and improves the internal structure".

Effective Modular Design

The process of breaking down a software into multiple independent modules where each module is developed separately is called Modularization.

Effective modular design can be achieved if the partitioned modules are separately solvable, modifiable as well as compilable. Here separate compilable modules means that after making changes in a module there is no need of recompiling the whole software system.

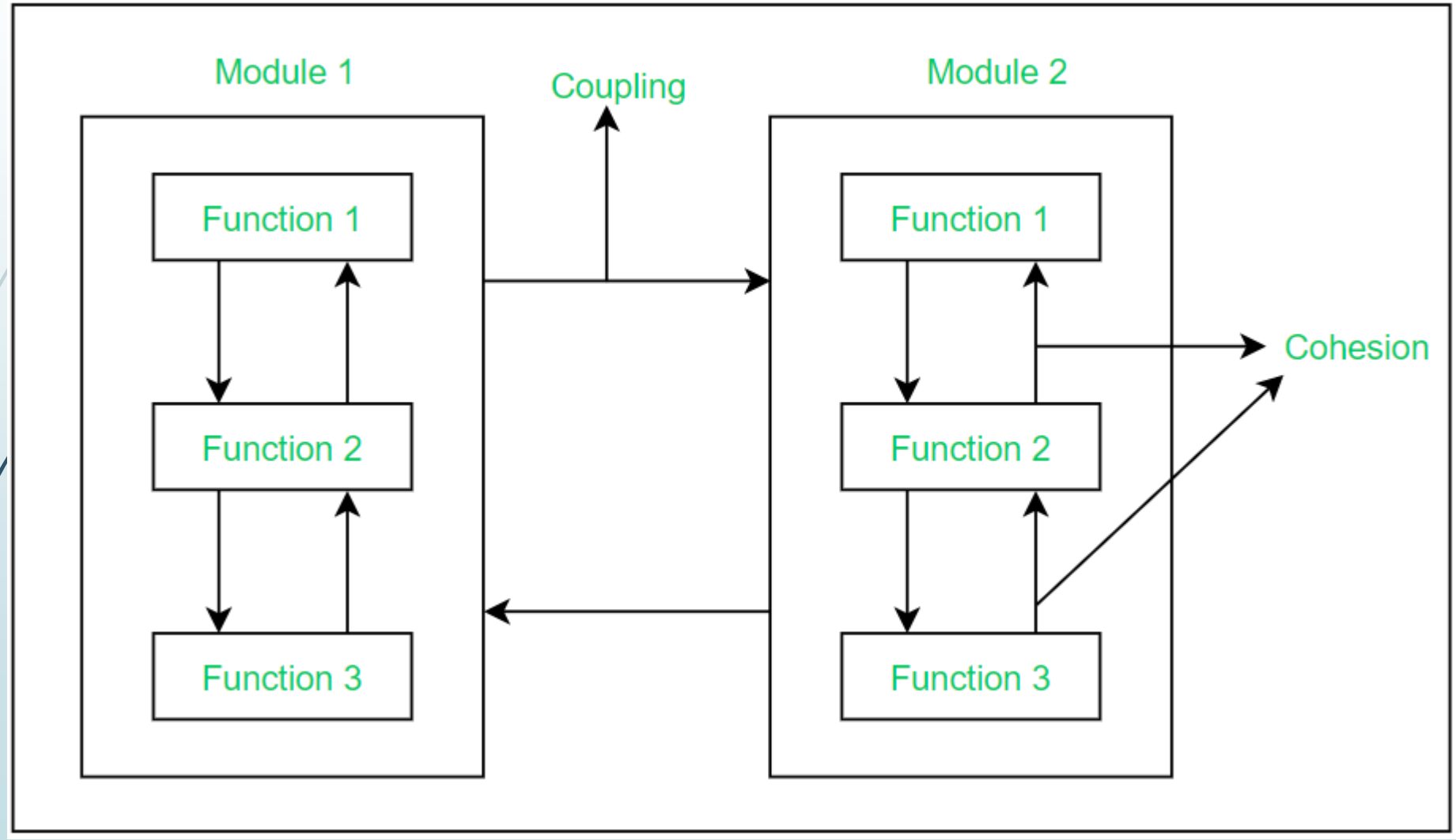
- In order to build a software with effective modular design there is a factor **"Functional Independence"** which comes into play. The meaning of Functional Independence is that a function is atomic in nature so that it performs only a single task of the software without or with least interaction with other modules. Functional Independence is considered as a sign of growth in modularity i.e., presence of larger functional independence results in a software system of good design and design further affects the quality of the software.

Benefits of Independent modules/functions in a software design:

- Since the functionality of the software have been broken down into atomic levels, thus developers get a clear requirement of each and every functions and hence designing of the software becomes easy and error free.

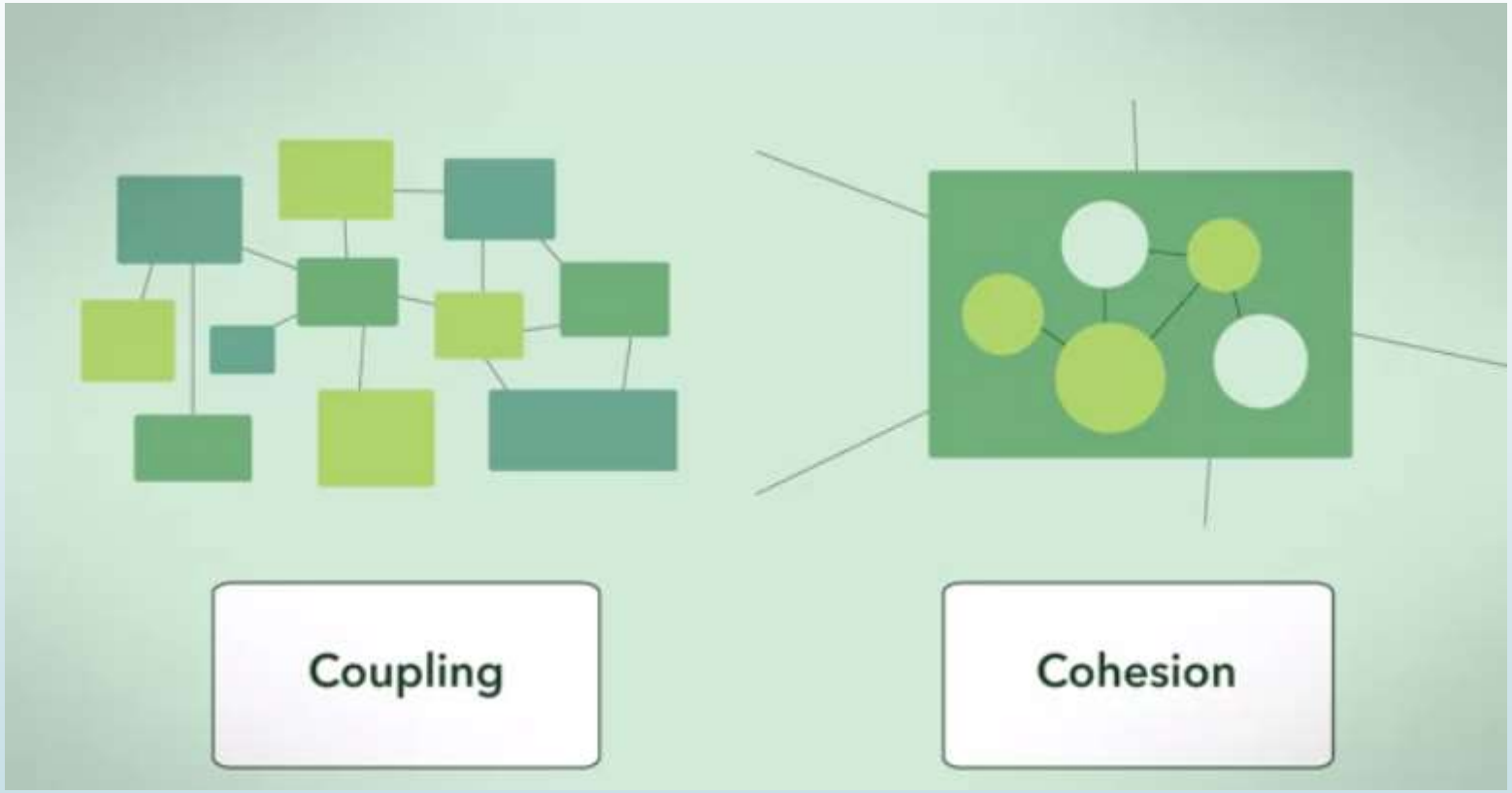
- As the modules are independent they have limited or almost no dependency on other modules. So, making changes in a module without affecting the whole system is possible in this approach.
- Error propagation from one module to another and further in whole system can be neglected and it saves time during testing and debugging.
- Independence of modules of a software system can be measured using 2 criteria : **Cohesion, and Coupling.**

Cohesion Vs Coupling



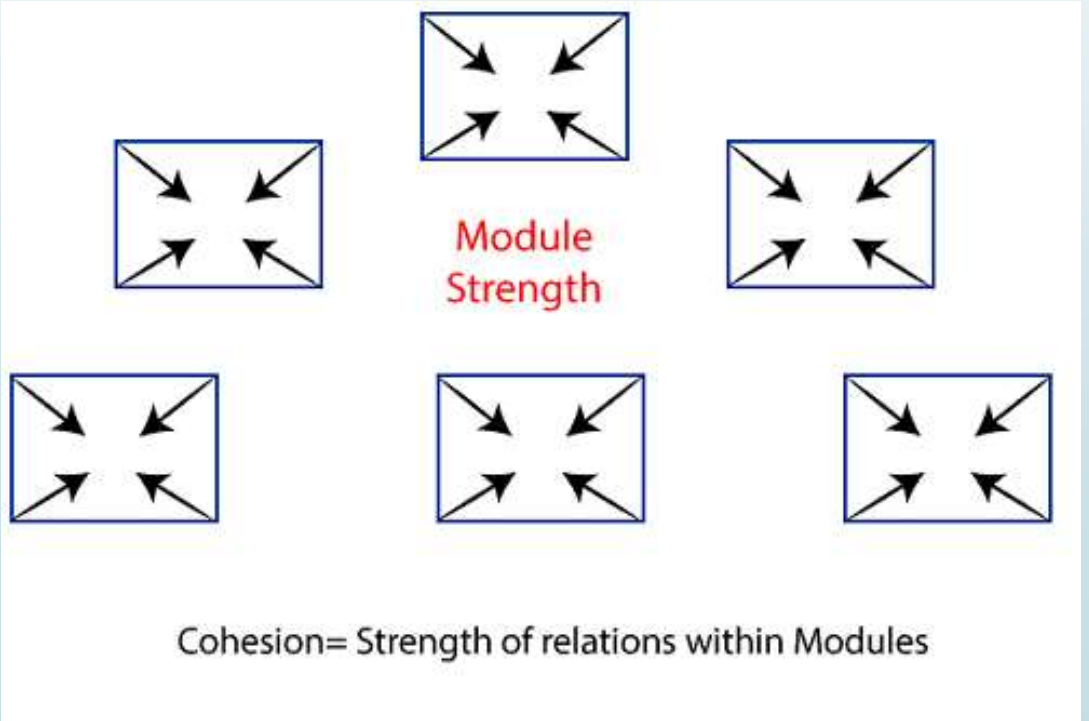
Cohesion Vs Coupling

- **We want: High cohesion and low coupling**

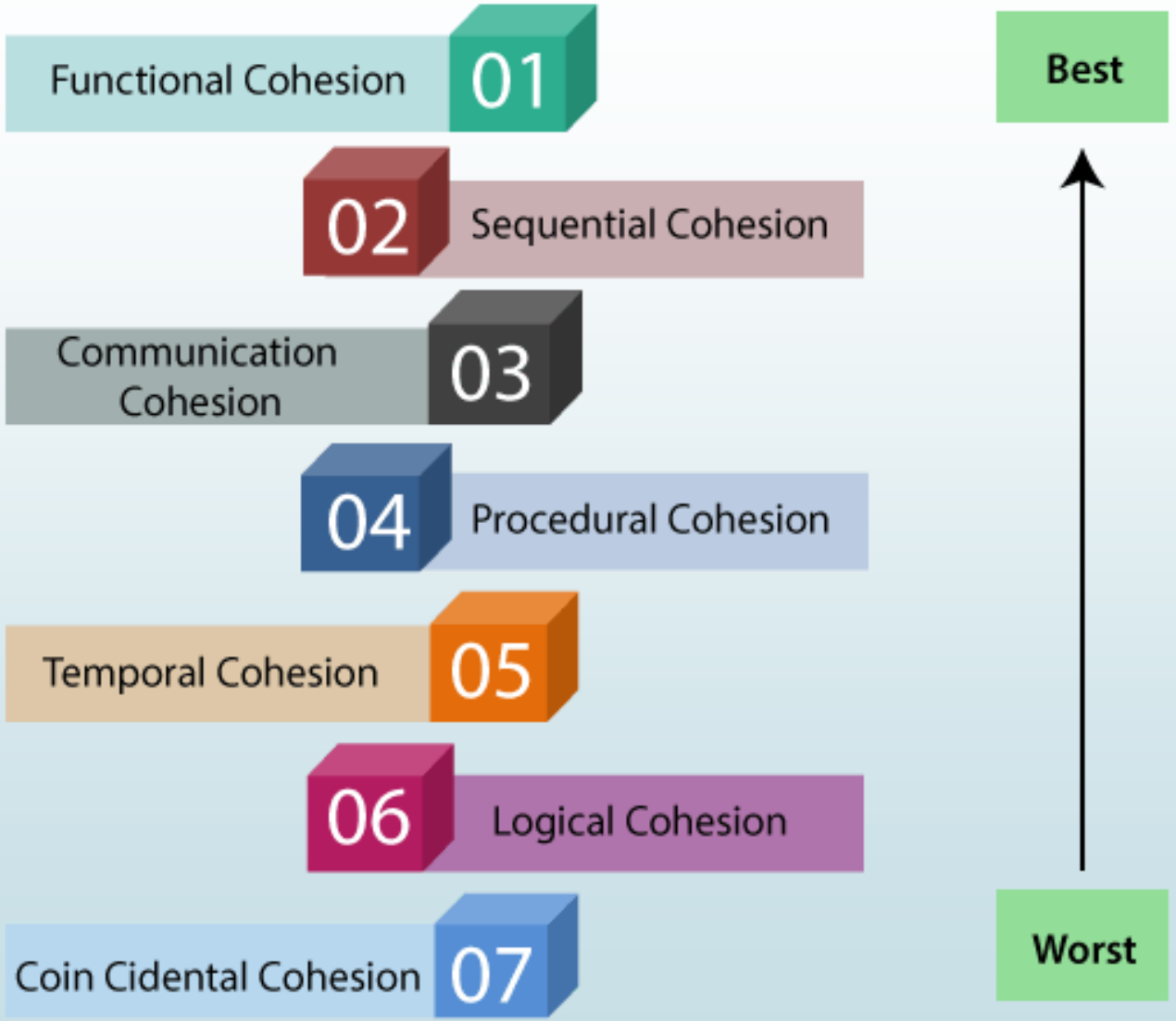


Cohesion

- 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.



Types of Modules Cohesion

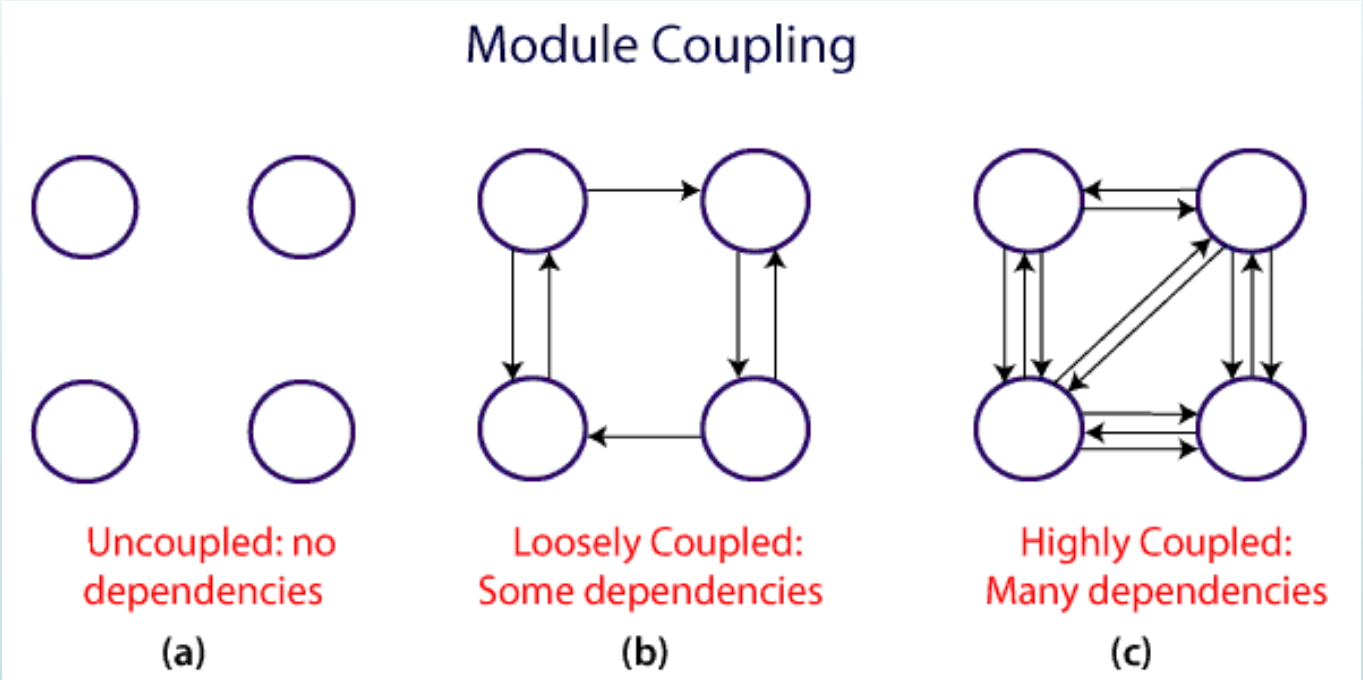


1. **Functional Cohesion:** Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.
2. **Sequential Cohesion:** A module is said to possess sequential cohesion if the element of a module form the components of the sequence, where the output 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.

Coupling

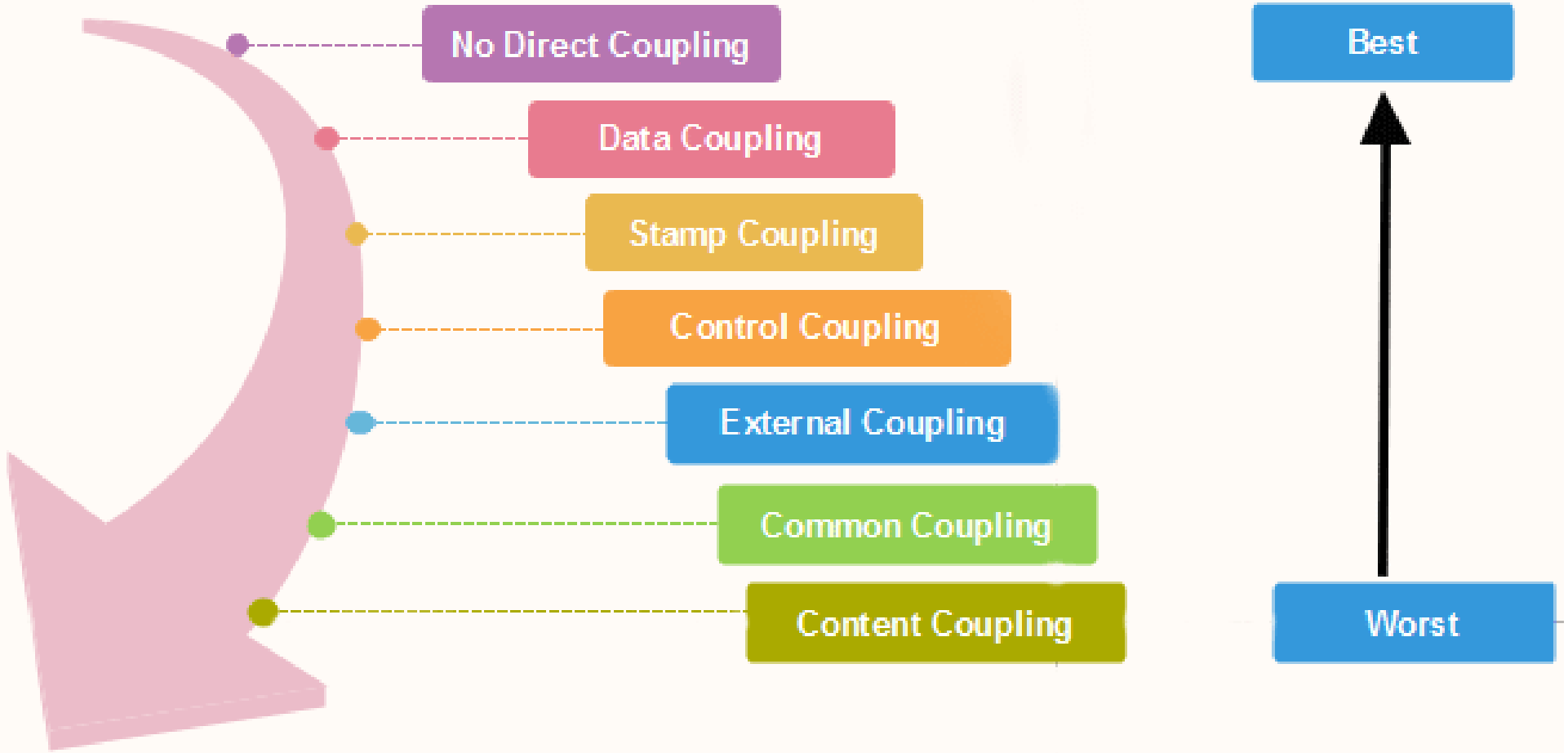
- The coupling is the degree of interdependence between software modules.
- Two modules that are tightly coupled are strongly dependent on each other.
- However, two modules that are loosely coupled are not dependent on each other.
- Uncoupled modules have no interdependence at all within them.

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 Modules Coupling

There are various types of module Coupling are as follows:



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 is one module controlling the flow of another, by passing it information on what to do (e.g., passing a what-to-do flag).

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.

Coupling Vs Cohesion

Coupling	Cohesion
Coupling is also called Inter-Module Binding.	Cohesion is also called Intra-Module Binding.
Coupling shows the relationships between modules.	Cohesion shows the relationship within the module.
Coupling shows the relative **independence** between the modules.	Cohesion shows the module's relative **functional** strength.
While creating, you should aim for low coupling, i.e., dependency among modules should be less.	While creating you should aim for high cohesion, i.e., a cohesive component/ module focuses on a single function (i.e., single-mindedness) with little interaction with other modules of the system.
In coupling, modules are linked to the other modules.	In cohesion, the module focuses on a single thing.

Architectural Design

- Software architecture is the high-level structure of a software system that defines how different components interact.
- It serves as a blueprint that guides developers in designing and implementing software systems efficiently.

The software needs an architectural design to represent the design of the software.

IEEE defines architectural design as "the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system."

The software that is built for computer-based systems can exhibit one of these many architectural styles.

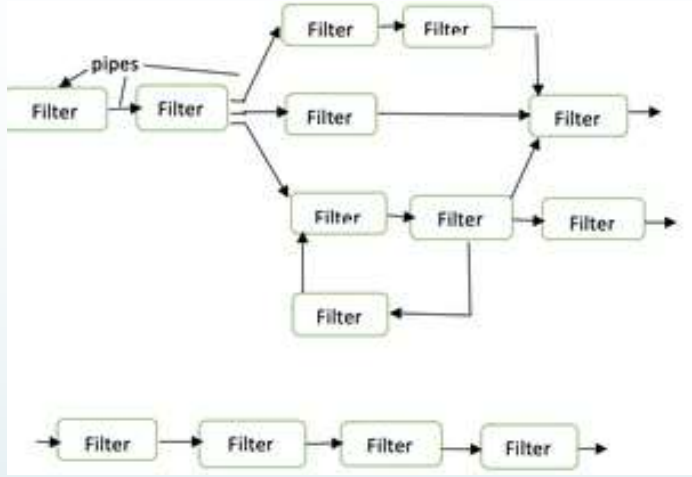
System Category Consists of

- A *set of components* (eg: a database, computational modules) that will perform a function required by the system.
- The *set of connectors* will help in coordination, communication, and cooperation between the components.
- *Condition*s that defines how components can be integrated to form the system.
- *Semantic models* that help the designer to understand the overall properties of the system.
- The *use* of architectural styles is to establish a structure for all the components of the system.

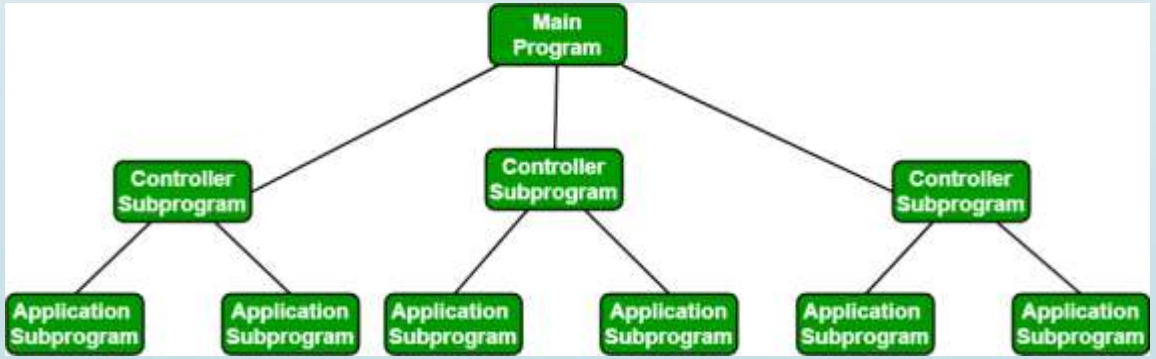
Different Architecture Styles:

- Data centered architectures
- Data flow architectures
- Call and Return architectures
- Object Oriented architecture
- Layered architecture

Data centered architectures



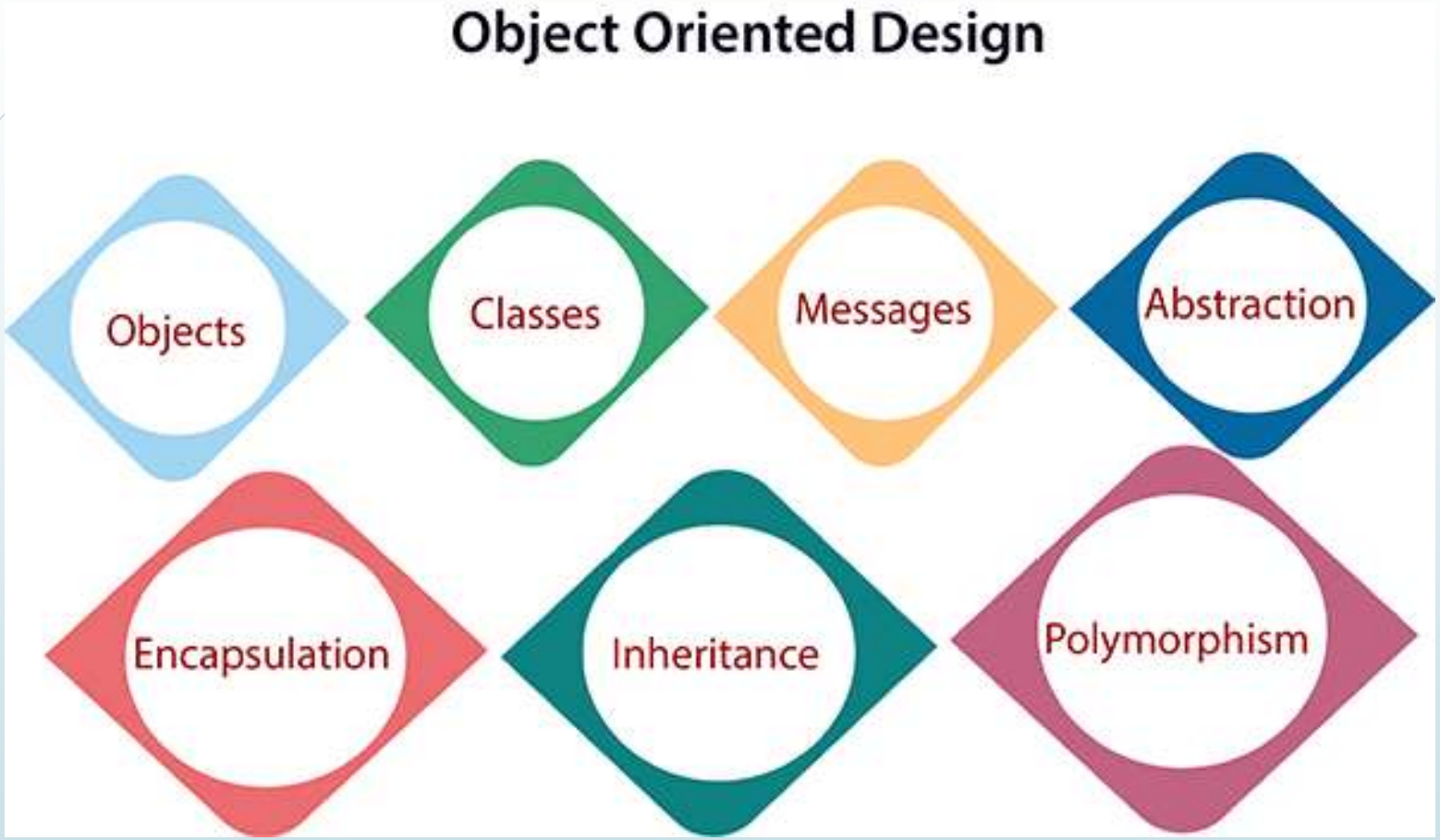
Data flow architectures



Call and Return architectures

Object-Oriented Design

- In the object-oriented design method, the system is viewed as a collection of objects (i.e., entities).
- The state is distributed among the objects, and each object handles its state data.
- For example, in a Library Automation Software, each library representative may be a separate object with its data and functions to operate on these data.
- In other words, each object is a member of some class. Classes may inherit features from the superclass.


Object Oriented Design
Objects
Classes
Messages
Abstraction
Encapsulation
Inheritance
Polymorphism

Advantages:

1. **Modularity:** OOD promotes modularity by organizing software into objects that represent real-world entities. This makes it easier to maintain and update software components independently.
2. **Reusability:** Objects can be reused in different parts of an application or in different applications altogether, which can significantly reduce development time and effort.
3. **Encapsulation:** Wrapping data (variables) and methods (functions) that operate on the data into a single unit (a class).
4. **Abstraction:** OOD allows developers to create abstract representations of real-world entities, which can simplify the design and implementation of complex systems.
5. **Inheritance:** Inheritance allows new classes to inherit attributes and methods from existing classes, enabling code reuse and promoting the creation of hierarchies of related classes.
6. **Polymorphism:** Polymorphism allows objects to be treated as instances of their parent class, which can simplify code and make it more flexible and adaptable to change.

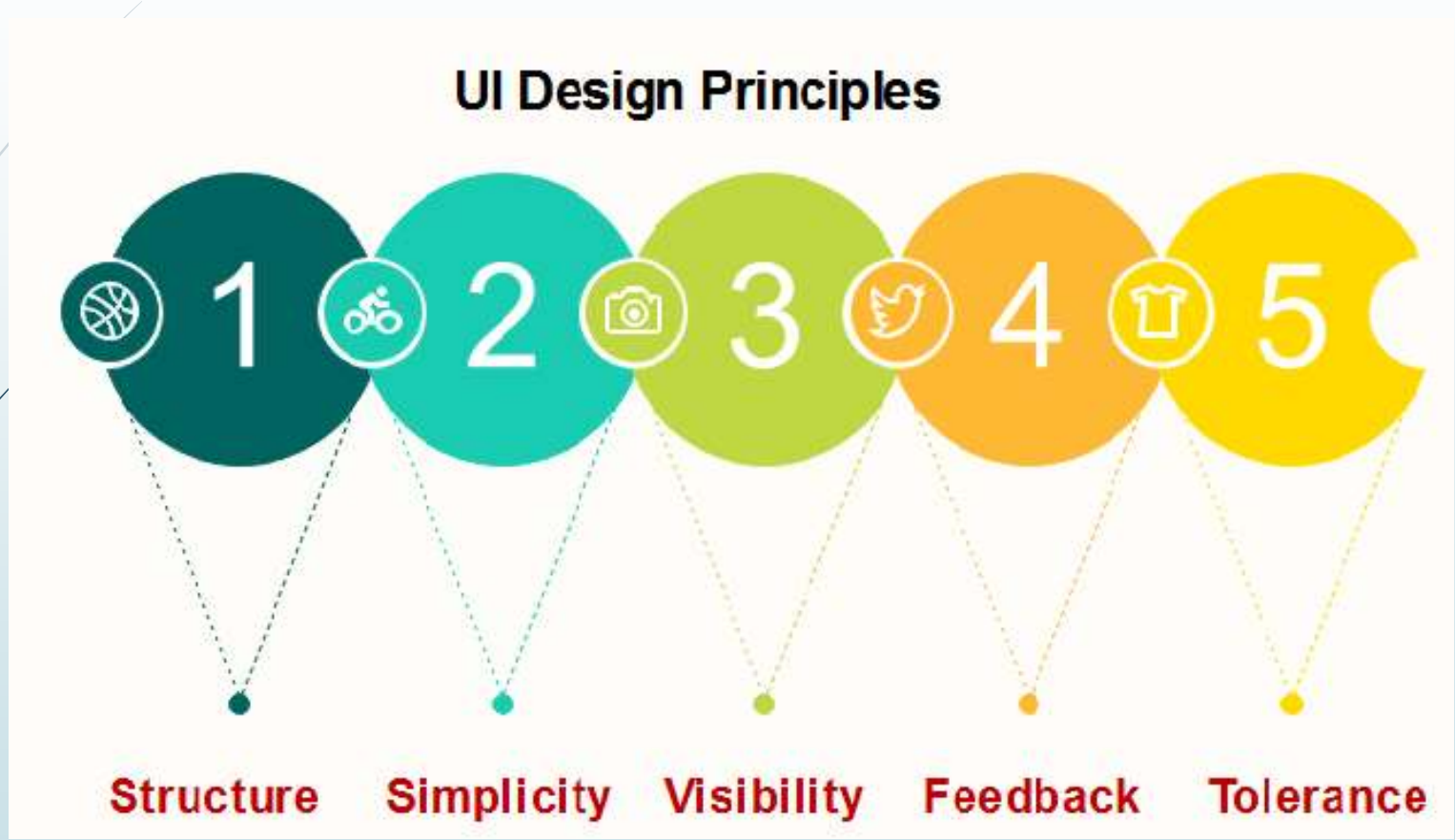
Challenges:

1. **Complexity:** Object-oriented designs can become complex, especially in large-scale systems with many interacting objects. Managing this complexity can be challenging and require careful design and planning.
2. **Performance Overhead:** Object-oriented programming languages often introduce a performance overhead compared to procedural languages due to features like dynamic dispatch and memory allocation for objects.
3. **Learning Curve:** Object-oriented concepts such as inheritance, polymorphism, and encapsulation can be challenging for developers new to OOD, leading to longer learning curves.
4. **Design Patterns:** Choosing the right design patterns for a given problem can be challenging, and using them incorrectly can lead to overly complex or inefficient designs.
5. **Testing and Debugging:** Object-oriented designs can be more difficult to test and debug compared to procedural designs, especially when dealing with complex interactions between objects.
6. **Maintenance:** While OOD promotes modularity and reusability, poorly designed object-oriented systems can be difficult to maintain and extend, especially if the original design did not anticipate future changes.

User-Interface Design

The user interface is the front-end application view to which the user interacts to use the software. The software becomes more popular if its user interface is:

- Attractive
- Simple to use
- Responsive in a short time
- Clear to understand
- Consistent on all interface screens


UI Design Principles
1
2
3
4
5
Structure
Simplicity
Visibility
Feedback
Tolerance

Elements:

- **Layout**
- **Grid systems** help align elements properly.
- Responsive design ensures adaptability on different devices.
- **Typography**
- Use readable fonts like **Sans-serif (e.g., Roboto, Open Sans)**.
- Maintain **consistent font sizes** for headings, body text, and buttons.
- **Color Scheme**
- Use a **primary color palette** and a few accent colors.
- Ensure **contrast** for readability (e.g., black text on a white background).
- **Buttons & Icons**
- Use **clear labels and recognizable icons**.
- Ensure buttons have a **hover/press state**.
- **Navigation**
- Keep navigation **simple and intuitive**.
- Use a **hamburger menu** for mobile devices.
- **Forms & Inputs**
- Provide **clear labels** and placeholders.
- Show error messages when users enter incorrect data.

Refactoring of designs

- Refactoring in software design refers to the process of modifying an existing system's structure without changing its external behavior.
- The goal is to improve code maintainability, readability, and performance while reducing technical debt.

Refactoring improves software by:

- Enhancing readability (simplifies complex code).
- Improving maintainability (reduces dependencies and code duplication).
- Boosting performance (removes unnecessary computations).
- Reducing complexity (eliminates redundant logic).
- Ensuring scalability (better modularity for future changes).

Refactoring should be done when:

- ✓ Code is difficult to understand or modify.
- ✓ The system has repeated code (duplication).
- ✓ Adding new features takes too much time due to tight coupling.
- ✓ The code has long methods or large classes.
- ✓ The system experiences performance bottlenecks.

Refactoring **is NOT done** when software is close to a deadline or when major redesigns are required (in such cases, rewriting may be a better option).

Types of Refactoring in Design

1. **Code-Level Refactoring:** Refactoring the structure of code without changing functionality.

- Extract Method: Break large functions into smaller, reusable functions.✓ Improves readability and maintainability.
- Reduce Nested Loops:✓ Improves performance.

2. Architectural Refactoring: Involves restructuring software components for better scalability and performance.

- Reduce coupling
- Layered Architecture Refactoring

3. Performance-Based Refactoring: Refactors inefficient code to improve performance.

- Optimize Loops ✓ Avoid redundant calculations inside loops.

4. UI/UX Design Refactoring: Refactoring user interfaces to improve usability.

- Reduce Clutter: Remove unnecessary UI elements.
- Improve Navigation: Simplify menu structures.
- Optimize Load Times: Reduce API calls and optimize image sizes.

Example:

- Before Refactoring – A long form with 10 fields on one page.
- After Refactoring – Break into multi-step forms to improve user experience.

Detailed Design Transaction Transformation

- Transaction transformation is a software design approach that deals with processing input transactions into output results by applying a structured sequence of operations.
- It is commonly used in business applications, banking systems, and real-time data processing.

What is Transaction Transformation?

It refers to the structured way of handling transactions (user requests or system inputs) that trigger a series of operations to produce a required output.

Transaction Processing Flow

A transaction typically follows these steps:

1. Input Transaction Processing

- The system receives an input request from a user or another system. Example: A customer places an online order.

2. Validation & Pre-Processing

- Checks if the input data is valid (e.g., username/password, stock availability). Example: Verify if an item is in stock before confirming the order.

3. Transformation & Computation

- The system processes the transaction according to business rules. Example: Calculate total cost including taxes and discounts.

4. Database Update

- The transaction modifies relevant data records. Example: Update inventory, deducting purchased quantity.

5. Output Generation

- The system generates results, which could be: UI feedback (e.g., "Order Confirmed!"), A receipt or invoice, A message or notification

Example – Online Banking Transaction

Let's consider an online fund transfer:

1. Input Transaction Processing

- User initiates transfer (inputs account details, amount).

2. Validation

- Check if account exists.
- Verify sufficient balance.

3. Processing

- Deduct amount from sender's account.
- Add amount to receiver's account.

4. Database Update

- Commit changes to ensure consistency.

5. Output Generation

- Show confirmation message (e.g., "Transfer Successful!").
- Send notification via SMS/email.

Error Handling & Recovery

Robust systems handle transaction failures using:

✅ **Rollback Mechanism** → Undo changes if a failure occurs.
✅ **Compensation Transactions** → Reverse incorrect transactions (e.g., refund processing).
✅ **Retry Logic** → Reattempt transaction if it fails due to temporary issues.

Thank You