CSE- 321
Software Engineering

Lecture : 12
Software design

Fahad Ahmed
Lecturer, Dept. of CSE
E-mail: fahadahmed@uap-bd.edu

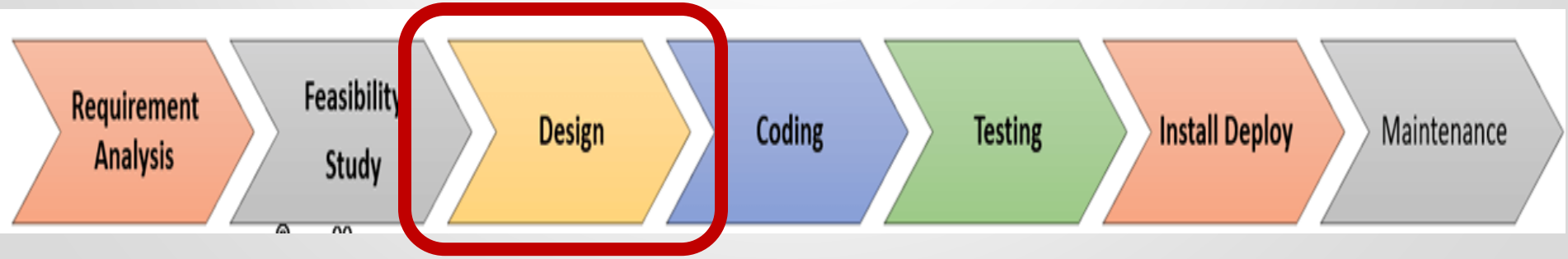
Lecture Outlines

- Software design
- Design Principles
- Strategy of Design
- Coupling and Cohesion
- Architectural Design
- Abstract machine (layered) Design
- Distributed Systems Architectures
- Client-Server Architecture
- Broker Architectural Style : CORBA
- Service-Oriented Architecture (SOA)

Software design

What is Software design ?

- Software design is a process to **transform user requirements** into some suitable form, which helps the programmer in **software coding and implementation.**
- Software design is the first step in SDLC (Software Design Life Cycle), which moves the concentration from **problem domain to solution domain.** It tries to specify how to fulfill the requirements mentioned in SRS.



Objectives of Software Design

Objectives of Software Design

Objectives of Software Design

Following are the purposes of Software design:

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

Software Design Levels

Software design yields three levels of results:

Architectural Design

- Highest abstract version of the system.
- It identifies the software as a system with many components interacting with each other.
- Designers get the idea of proposed solution domain.

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.

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

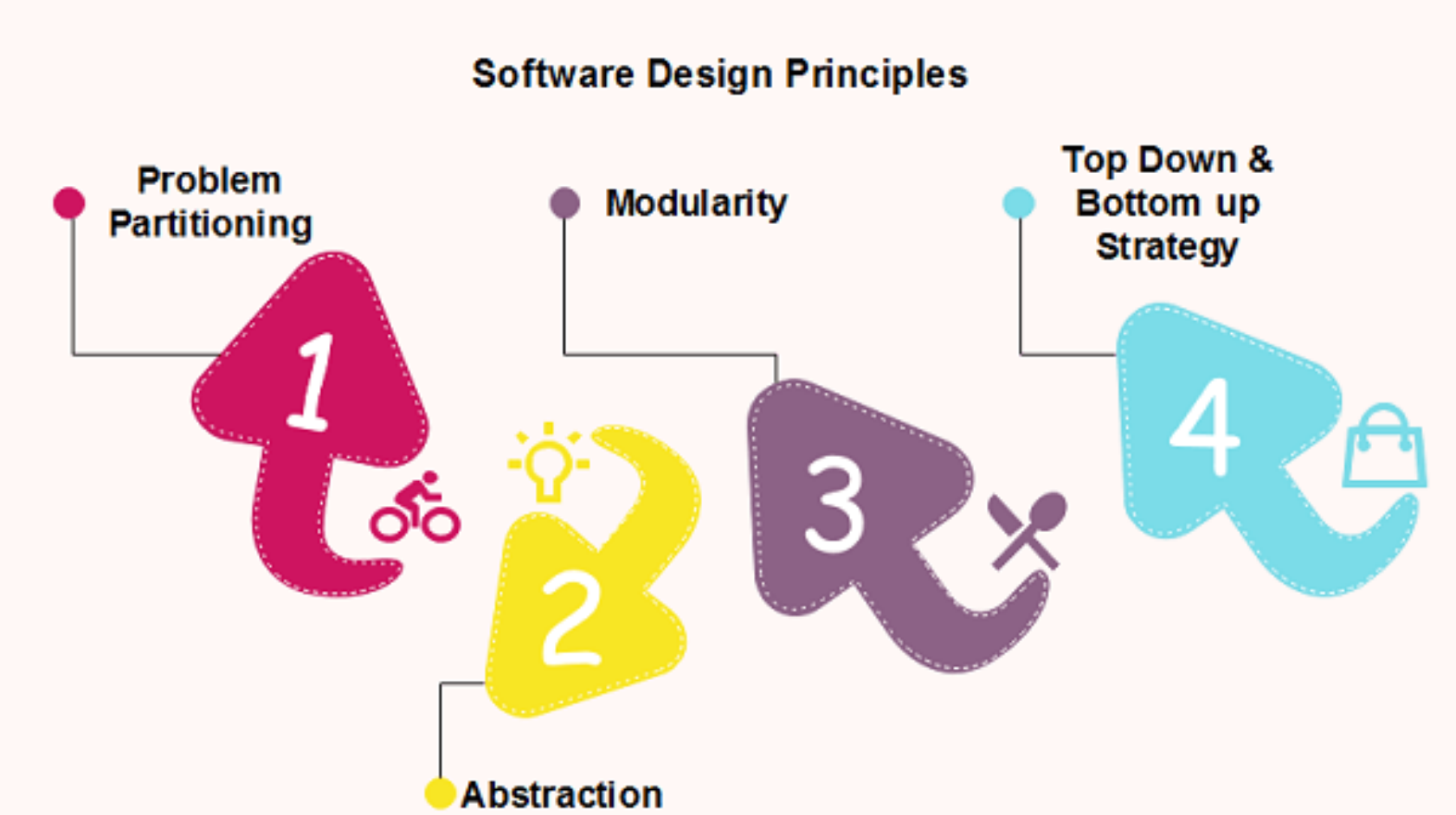
Software Design Principles

- Design is not coding, coding is not design.
- The design should be traceable to the analysis model.
- The design should not reinvent the wheel.
- The design should "minimize the intellectual distance" between the software and the problem as it exists in the real world.
- The design should exhibit uniformity and integration.
- The design should be structured to accommodate change.
- The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.

- The design should be assessed for quality as it is being created, not after the fact.
- The design should be reviewed to minimize conceptual (semantic) errors.

Software Design Principles

Software design principles are concerned with providing means to handle the complexity of the design process effectively.



Software Design Principles

Problem Partitioning

For small problem, we can handle the entire problem at once but for the significant problem, divide the problems and conquer the problem it means to divide the problem into smaller pieces so that each piece can be captured separately.

Benefits of Problem Partitioning

1. Software is easy to understand
2. Software becomes simple
3. Software is easy to test
4. Software is easy to modify
5. Software is easy to maintain
6. Software is easy to expand

As the number of partition increases = Cost of partition and complexity increases

Modularization

Modularization is a technique to divide a software system into multiple discrete and **independent modules**, which are expected to be capable of carrying out task(s) **independently.** These modules may work as basic constructs for the entire software.

Advantage of modularization:

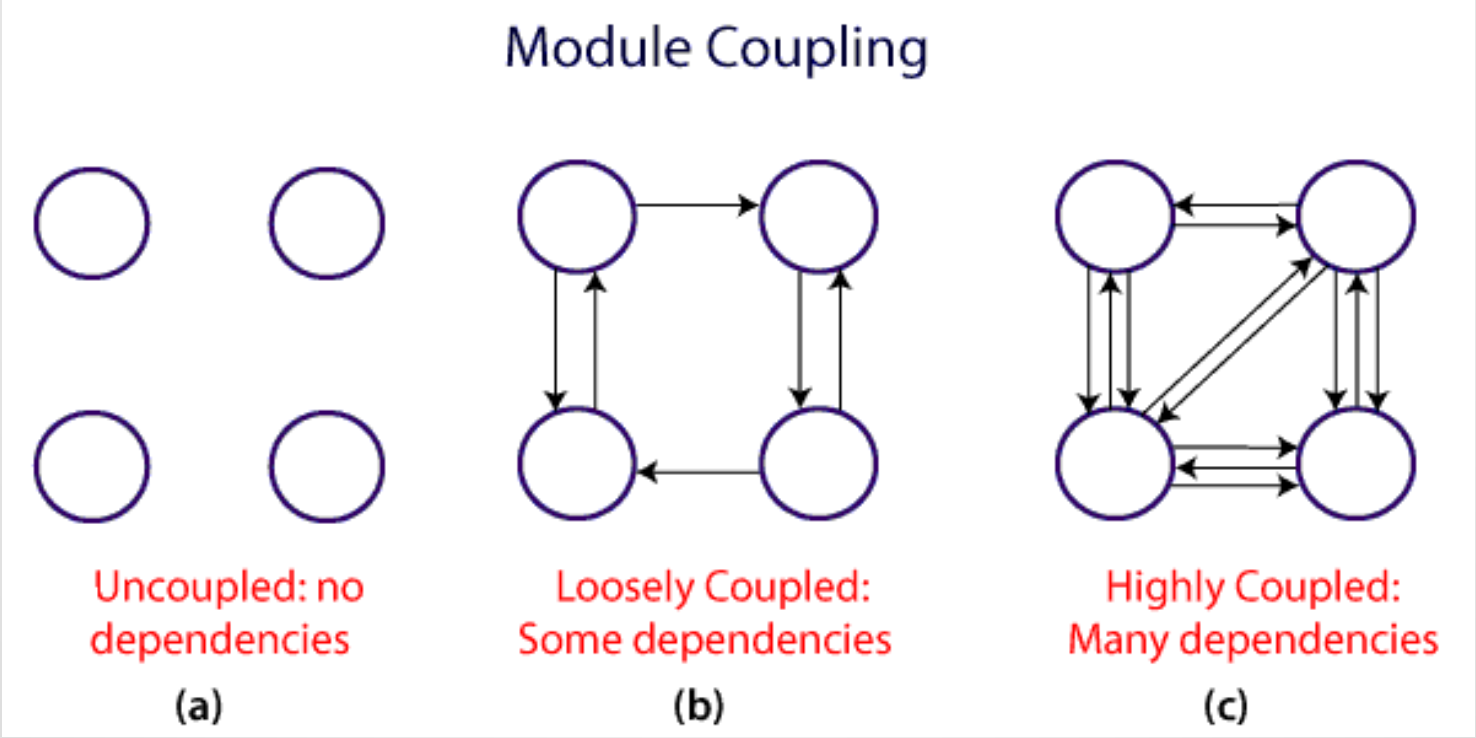
- Smaller components are easier to maintain
- Program can be divided based on functional aspects
- Desired level of abstraction can be brought in the program
- Components with high cohesion can be re-used again
- Concurrent execution can be made possible
- Desired from security aspect

Coupling and Cohesion

Module Coupling

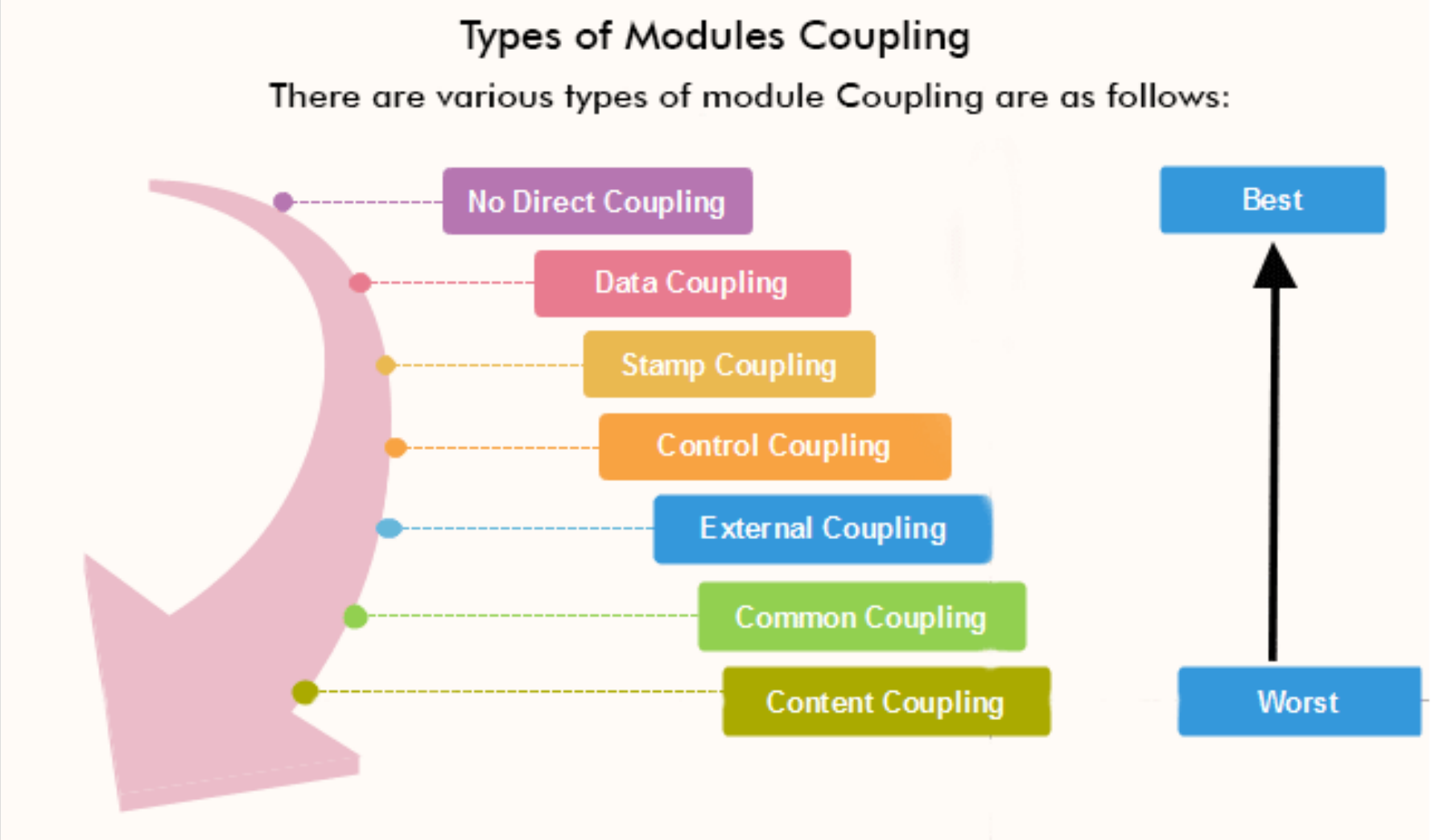
In software engineering, Coupling is measured by the number of relations between the modules. That means, the coupling is the degree of interdependence between software modules.

A good design is the one that has low coupling.



Coupling and Cohesion

Types of Module Coupling



https://www.javatpoint.com/software-engineering-coupling-and-cohesion

Coupling and Cohesion

Data Coupling: If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent to each other and communicating through data. Module communications don't contain tramp data. Example-customer billing system.

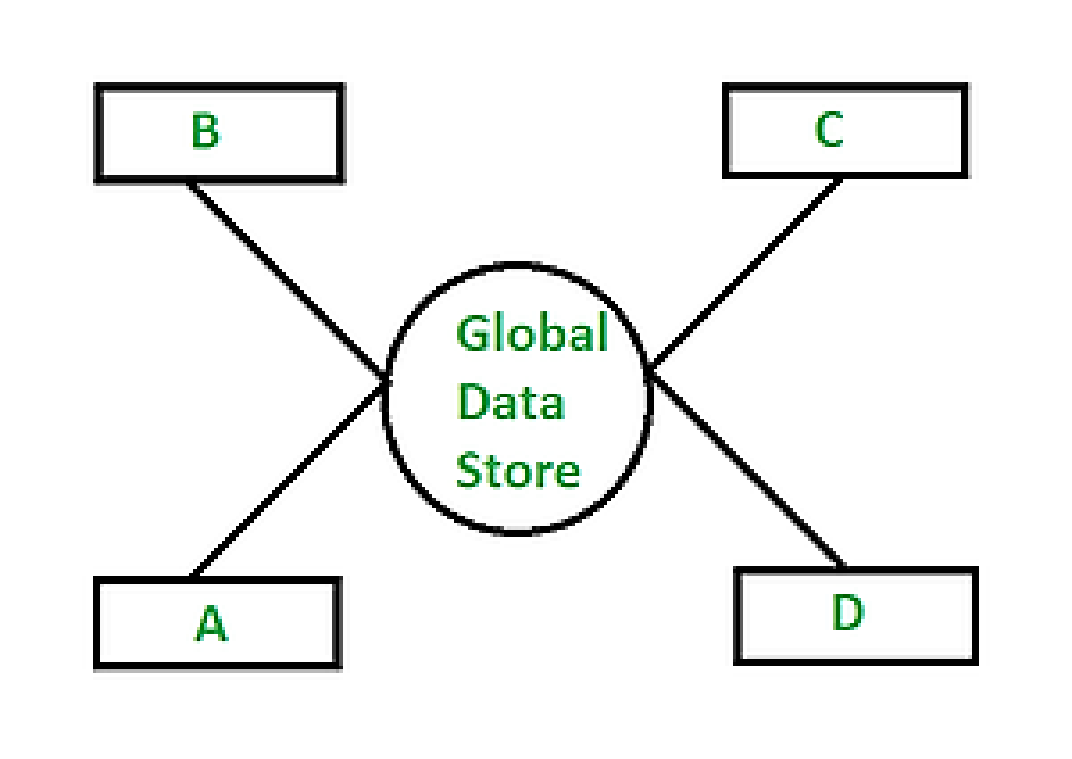
Stamp Coupling: Two modules are stamp coupled if they communicate using composite data items such as structure, objects, etc. When the module passes non-global data structure or entire structure to another module, they are said to be stamp coupled. For example, passing structure variable in C or object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

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

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.

Coupling and Cohesion

Common Coupling: The modules have shared data such as global data structures. The changes in global data mean tracing back to all modules which access that data to evaluate the effect of the change. So it has got disadvantages like difficulty in reusing modules, reduced ability to control data accesses and reduced maintainability.



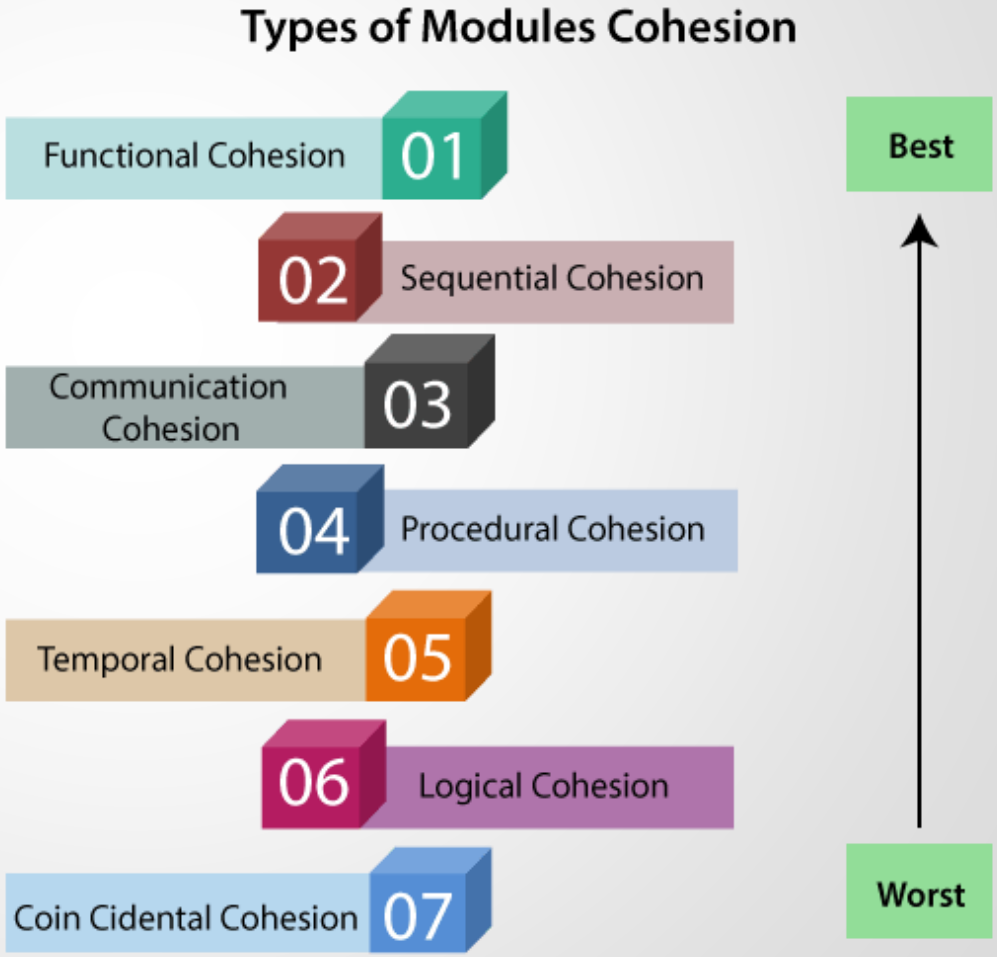
Coupling and Cohesion

Module 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.

Basically, cohesion **is the internal glue that keeps the module together**.

A good software design will have high cohesion.



https://www.javatpoint.com/software-engineering-coupling-and-cohesion

Coupling and Cohesion

Functional Cohesion: Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.

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.

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.

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.

Design Pattern

- A design pattern provides a general reusable solution for the common problems occurs in software design.
- The patterns typically show relationships and interactions between classes or objects.
- Design Patterns establishes solutions to common problems which helps to keep code maintainable, extensible and loosely coupled.
- It is popularized by Gang Of Four(1994) book.

Design Pattern

- A Design Pattern is neither a static solution nor is it an algorithm, No hard rule of the coding standard.

- Software Architecture is not a Design Pattern. Software Architecture dictates what's going to implemented and where it will be put. While Design Patterns states how it should be done.

Design Pattern

Why You Should Learn Design Patterns?

If you boil-down the definition of Object-Oriented Design, it combining data and its operation into a context-bound entity(i.e. class/struct). And it stands true while designing an individual object.

But when you are designing complete software you need to take into the account that

- **Creational Design Patterns:** How do those objects going to be instantiated/created?
- **Structural Design Patterns:** How do those objects combine with other object and formalize bigger entity? which should also be scalable in future.
- **Behavioural Design Patterns:** You also need to think in terms of communication between those objects which can anticipate future changes easily and with fewer side effects.

Design Pattern

Types of Design Patterns

Creational Design Patterns

1. Factory
2. Builder
3. Prototype
4. Singleton

Structural Design Patterns

1. Adapter
2. Bridge
3. Composite
4. Decorator
5. Facade
6. Flyweight
7. Proxy

Behavioural Design Patterns

1. Chain of responsibility
2. Command
3. Interpreter
4. Iterator
5. Mediator
6. Memento
7. Observer
8. State
9. Strategy
10. Template Method
11. Visitor

Design Pattern

For example, Assume we are designing an app for a company that provides online service of household equipments.

In our applications core we have this method *processServiceRequest()*, the purpose of which is to create instance of a *OnlineService* class based on the *serviceType* and process the request.

Design Pattern

```
1  public void processServiceRequest(String serviceType) {
2      OnlineService service;
3      if (service.equals("AirConditioner"))
4          service = new ACService();
5      else if (service.equals("WashingMachine"))
6          service = new WMService();
7      else if (service.equals("Refrigerator"))
8          service = new RFService();
9      else
10         service = new GeneralService();
11
12     service.getinfo();
13     service.assignServiceRequest();
14     service.assignAgent();
15     service.processRequest();
16 }
```

Here, the type of service is a functionality which is bound to change at anytime. We might remove some services or add new and every such change in implementations would require to change this piece of code.

We can remove the code that creates instances and create a class which would work as a factory class that is only there to provide required type of instances. We are going to follow the Factory design pattern here and we will be having only one method *getOnlineService()* in this class which would do our job

Design Pattern

```
1  public class OnlineServiceFactory {
2      public OnlineService getOnlineService(String type) {
3          OnlineService service;
4          if(service.equals("AirConditioner"))
5              service = new ACService();
6          else if(service.equals("WashingMachine"))
7              service = new WMService();
8          else if(service.equals("Refrigerator"))
9              service = new RFService();
10         else
11             service = new GeneralService();
12         return service;
13     }
14 }
```

now we can refactor our previous code as,

```
public void processServiceRequest(String serviceType) {
    OnlineService service = new OnlineServiceFactory().getOnlineService(serviceType);
    service.getinfo();
    service.assignServiceRequest();
    service.assignAgent();
    service.processRequest();
}
```

Now any changes with the service types wont affect the rest of the code.

Design Principles : SOLID

SOLID is one of the most popular sets of design principles in object-oriented software development introduced by Robert C. Martin, popularly known as Uncle Bob.

The SOLID principles comprise of these five principles:

- **S**RP -- Single Responsibility Principle : the responsibilities of a class should be limited
- **O**CP -- Open/Closed Principle : design should be open for extension but closed for modifications
- **L**SP -- Liskov Substitution Principle :should always be able to substitute subtypes for their base class
- **I**SP -- Interface Segregation Principle : Any component that is not related to each other must be separated and segregated
- **D**IP -- Dependency Inversion Principle : the high level modules should not depend on the low level modules. Instead both should depend on abstraction

https://bootcamp.uxdesign.cc/software-design-principles-every-developers-should-know-23d24735518e

Design Patterns vs Design Principles

Design Patterns vs Design Principles

Design Principles are general guidelines that can guide your class structure and relationships.

On the other hand, Design Patterns are proven solutions that solve commonly reoccurring problems.

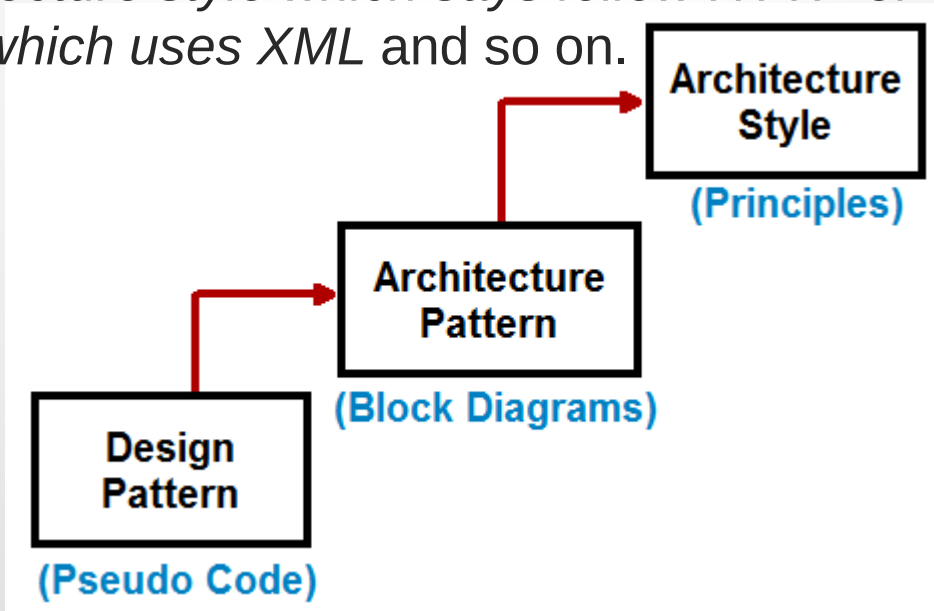
Having said that, most of the practical implementations of these design principles are mostly done using one or more design patterns.

Design Pattern

•In design pattern you know the **source code** , you know some **pseudo-code**. For example in singleton pattern you know you have to make class as static , constructor has private. In iterator pattern you know you have to use enumerable , enumerator as so on.

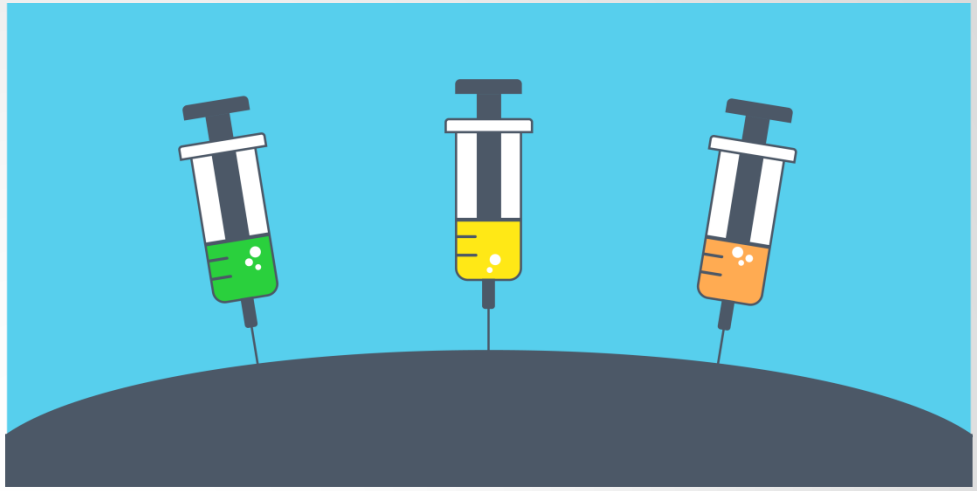
•In architecture pattern you know **only blocks** like MVC , MVVM , MVP and so on. So you know only layers and not the actual code.

•Architecture style is all about **only principles**. It just has one liners like *REST is architecture style which says follow HTTP* or *SOA is a architecture style which uses XML* and so on.



Dependency injection

In software engineering, dependency injection is a technique in which an object receives other objects that it depends on, called dependencies.



Classes often require references to other classes. These required classes are called dependencies. Typically, the receiving object is called a client and the passed-in ('injected') object is called a **service**. The code that passes the service to the client is called the **injector**.

Instead of the client specifying which service it will use, the injector tells the client what service to use. The 'injection' refers to the passing of a dependency (a service) into the client that uses it.

https://developer.android.com/training/dependency-injection#java

Dependency injection

A client who wants to call some services should not have to know how to construct those services. Instead, the client delegates to external code (the injector). The client is not aware of the injector. The injector passes the services, which might exist or be constructed by the injector itself, to the client. The client then uses the services.

The service is made part of the client's state. Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

The intent behind dependency injection is to achieve separation of concerns of construction and use of objects. This can increase readability and code reuse.

Dependency injection solves the following problems:

- How can a class be independent of how the objects on which it depends are created?
- How can the way objects are created be specified in separate configuration files?
- How can an application support different configurations?

Dependency injection

Dependency injection involves four roles:

- the **service** objects to be used
- the **client** object, whose behaviour depends on the services it uses
- the **interfaces** that define how the client may use the services
- the **injector**, which constructs the services and injects them into the client

There are at least three ways a **client can receive a reference** to an external module:

Constructor injection: The dependencies are provided through a client's class constructor.

Setter injection: The client exposes a setter method that the injector uses to inject the dependency.

Interface injection: The dependency's interface provides an injector method that will inject the dependency into any client passed to it.

Dependency injection

Use Hilt in your Android app

Hilt is Jetpack's recommended library for dependency injection in Android. Hilt defines a standard way to do DI in your application by providing containers for every Android class in your project and managing their lifecycles automatically for you.

Hilt is built on top of the popular DI library Dagger to benefit from the compile time correctness, runtime performance, scalability, and Android Studio support that Dagger provides.

In Kotlin

```
buildscript {
    ...
    dependencies {
        ...
        classpath("com.google.dagger:hilt-android-gradle-plugin:2.38.1")
    }
}
```

Alternatives to dependency injection

Alternatives to dependency injection

An alternative to dependency injection is using a **service locator**. The service locator design pattern also improves decoupling of classes from concrete dependencies. You create a class known as the service locator that creates and stores dependencies and then provides those dependencies on demand.

Architectural Design

Architectural Design

- Architectural design is a process for identifying the sub-systems making up a system and the framework for sub-system control and communication.
- The output of this design process is a description of the **software architecture**.
- It represents the link between specification and design processes
- Often carried out in parallel with some specification activities.
- It involves identifying major system components and their communications.

Architectural Design

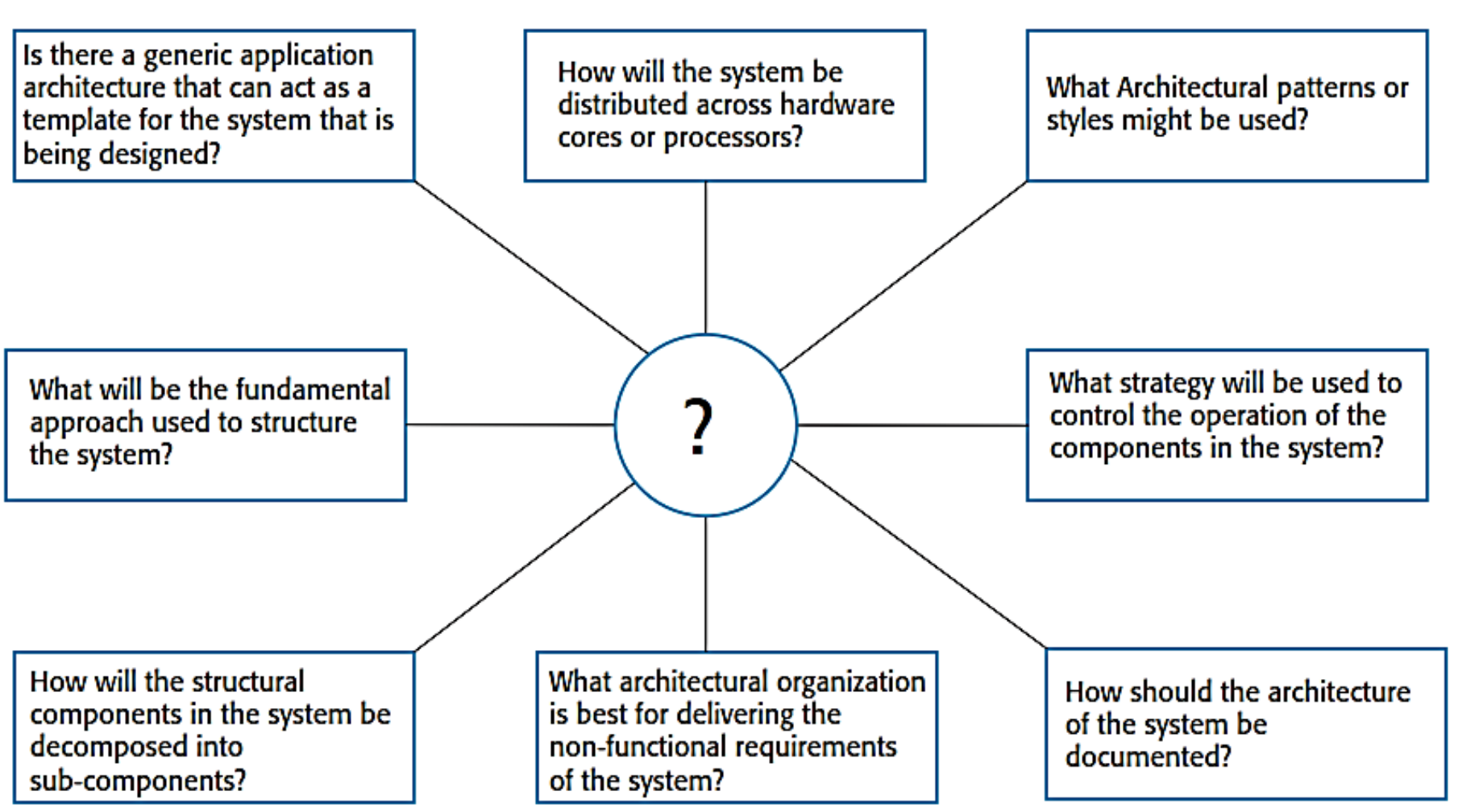
- A software architecture is a description of how a **software system is organized**.
- Architectural design decisions include decisions on the type of application, the distribution of the system, the architectural styles to be used.
- Architectures may be **documented from several different perspectives** or views such as a conceptual view, a logical view, a process view, and a development view.
- Architectural patterns are a means of reusing knowledge about generic system architectures.
- Architectural patterns are similar to software design pattern but have a broader scope.

Architectural design decisions

Architectural design is a creative process so the process differs depending on the type of system being developed.
However, a number of common decisions span all design processes.

- Is there a generic application architecture that can be used?
- How will the system be distributed?
- What architectural styles are appropriate?
- What approach will be used to structure the system?
- How will the system be decomposed into modules?
- What control strategy should be used?
- How will the architectural design be evaluated?
- How should the architecture be documented?

Architectural design decisions



Architectural views

4 + 1 view model of software architecture

· A **logical view,** which shows the key abstractions in the system as objects or object classes.

· A **process view**, which shows how, at run-time, the system is composed of interacting processes.

· A **development view**, which shows how the software is decomposed for development.

· A **physical view**, which shows the system hardware and how software components are distributed across the processors in the system.

· ***Related using use cases or scenarios (+1)***

Architectural patterns

- Patterns are a means of representing, sharing and reusing knowledge.
- An architectural pattern is a **stylized description** of good design practice, which has been tried and tested in different environments.
- Patterns should include information about when they are and when they are not useful.
- Patterns may be represented using tabular and graphical descriptions.

The Model-View-Controller (MVC) pattern

- **Separation of Concerns** (**SoC**) is a design principle for separating a computer program into distinct sections. Each section addresses a separate concern, a set of information that affects the code of a computer program.

- **MVC** (Model-View-Controller) is a pattern in software design commonly used to implement user interfaces, data, and controlling logic.

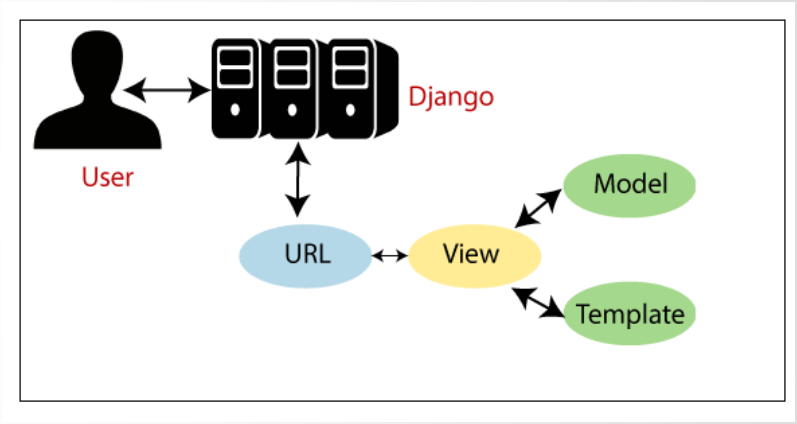
- It emphasizes a separation between the software's business logic and display.

- This "separation of concerns" provides for a better division of labour and improved maintenance.

- MVC is popular as it isolates the application logic from the user interface layer and different parts of the application can be developed separately.

The Model-View-Controller (MVC) pattern

- Some other design patterns followed by MVC, such as
  - **MVVM (Model-View-Viewmodel),**
  - MVP (Model-View-Presenter)
  - MVW (Model-View-Whatever).
  - Model View Template



It is a collection of three important components **Model View and Template**. The Model helps to handle **database**. It is a data access layer which handles the data. **In an MVT, the controller part is taken care of by the framework itself.**

The Model-View-Controller (MVC) pattern

•Serves as a basis of interaction management in many web-based systems.

•Decouples three major interconnected components:

- The **model** is the central component of the pattern that directly manages the data, logic and rules of the application. It is the application's dynamic data structure, independent of the user interface.

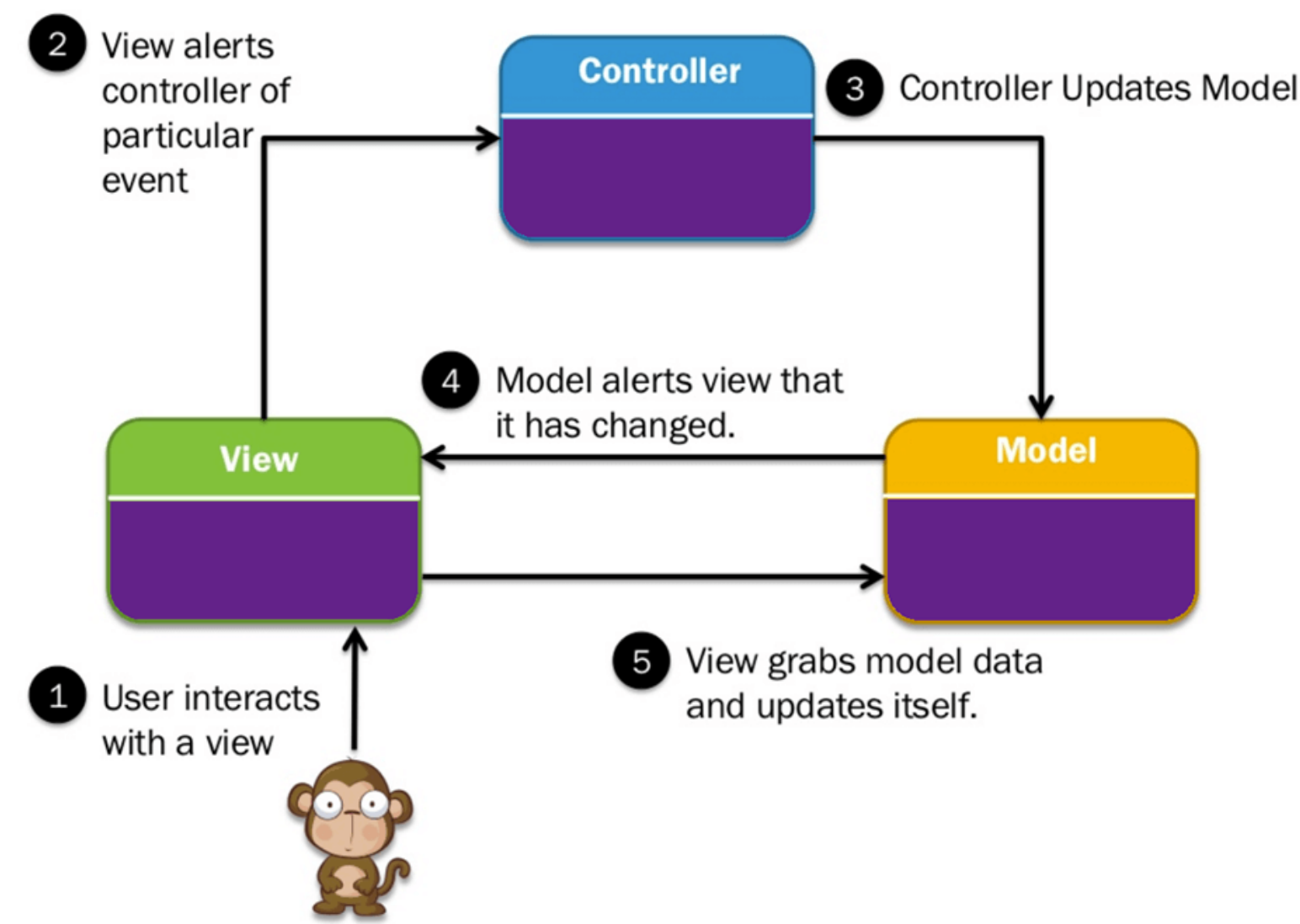
- A **view** can be any output representation of information, such as a chart or a diagram. Multiple views of the same information are possible.

- The c**ontroller** accepts input and converts it to commands for the model or view.

•Supported by most language frameworks.

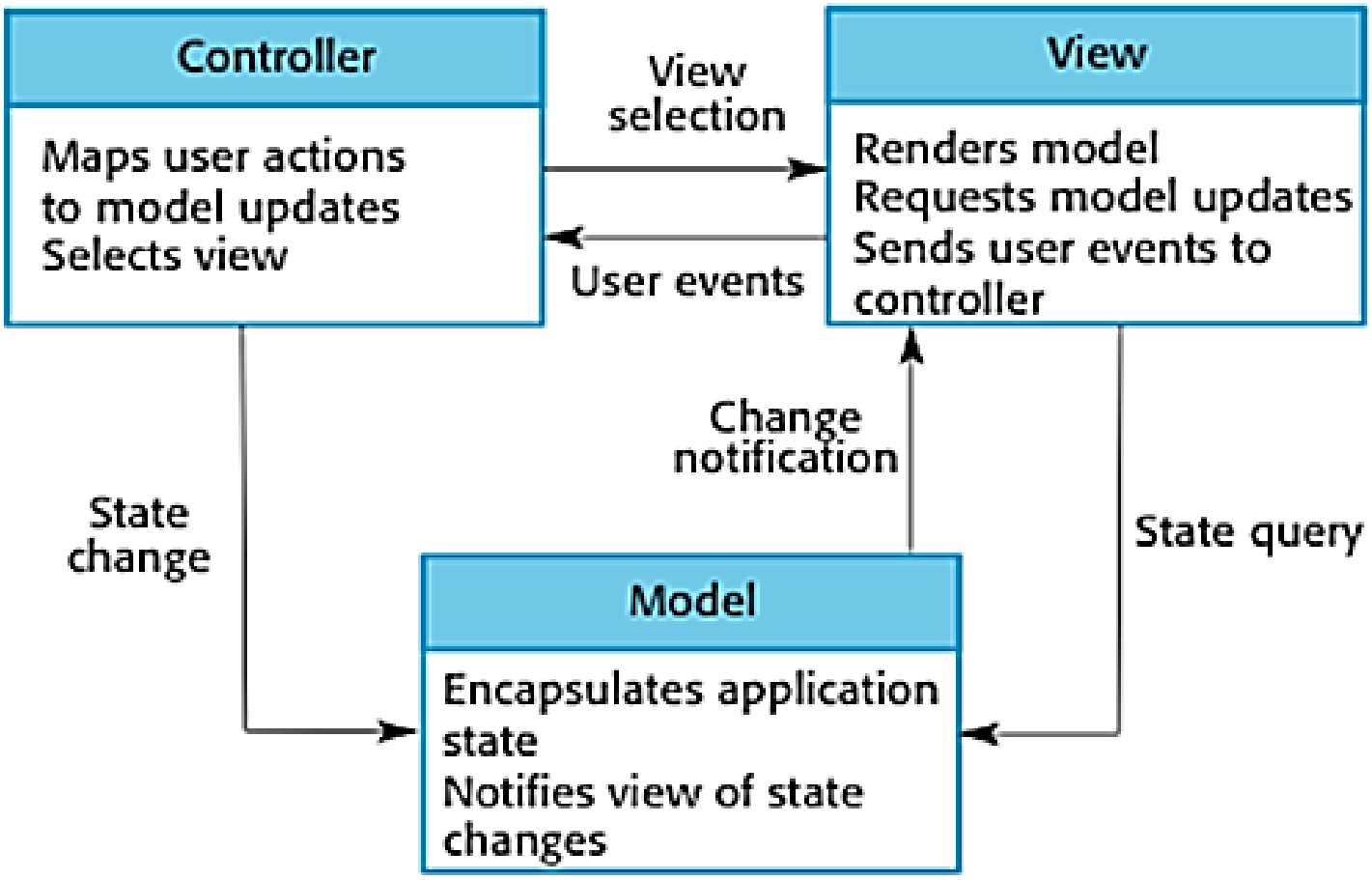
The Model-View-Controller (MVC) pattern

MVC architectural pattern



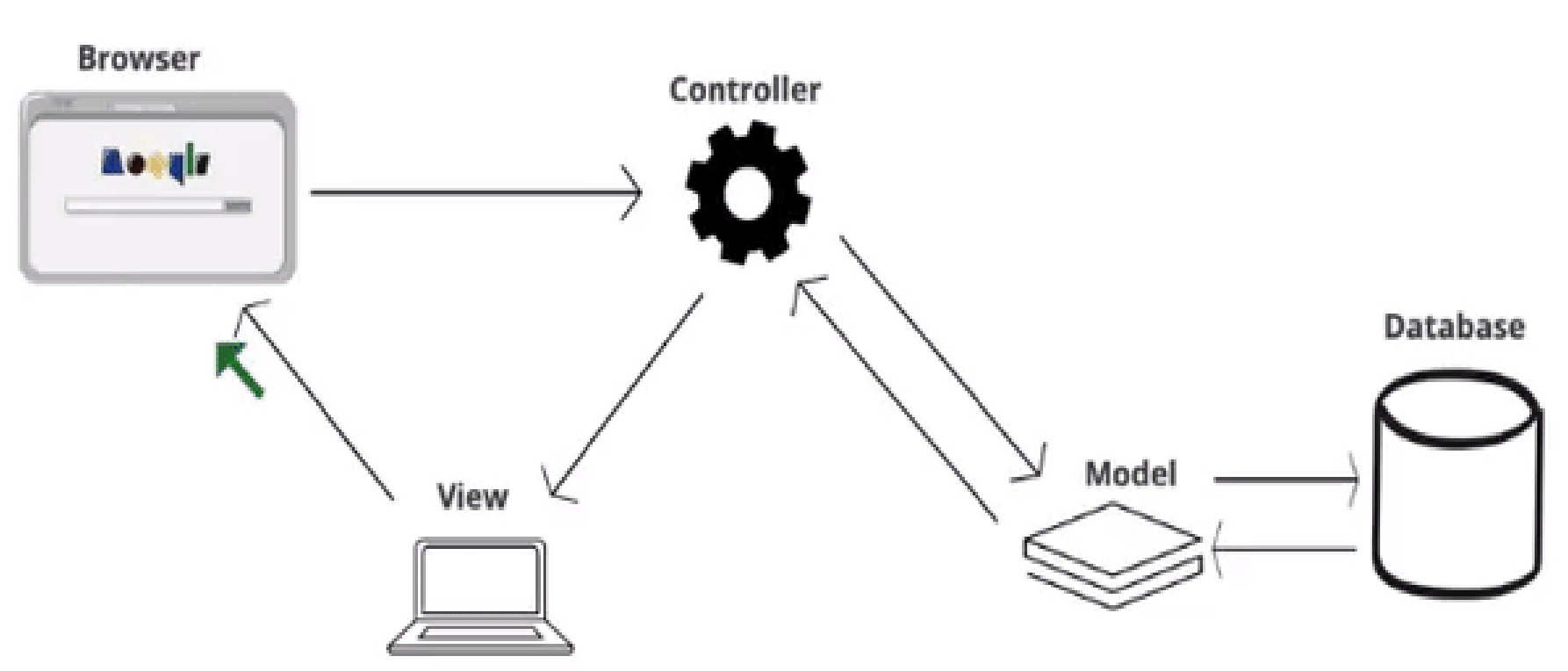
The Model-View-Controller (MVC) pattern

MVC architectural pattern



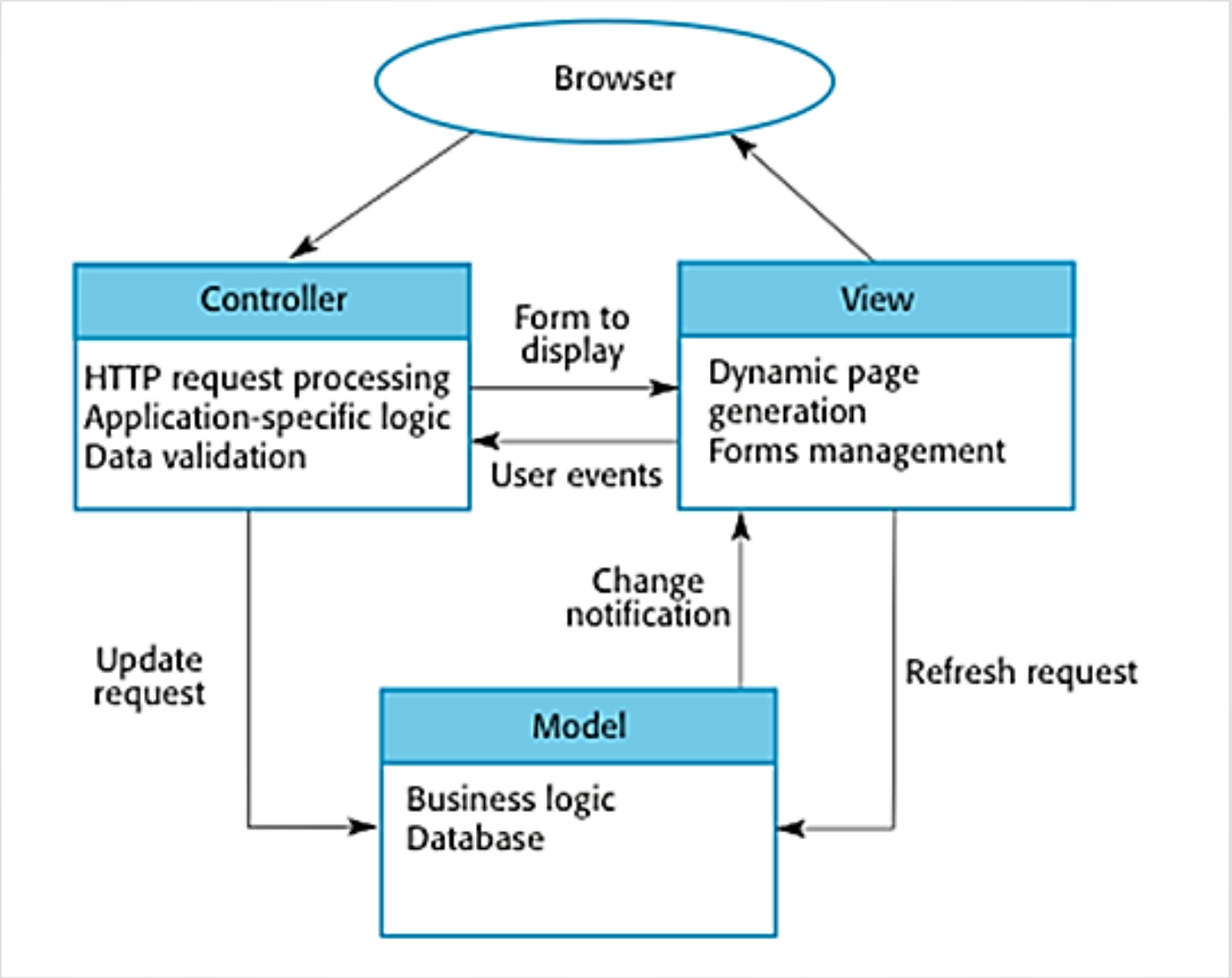
The Model-View-Controller (MVC) pattern

MVC architectural pattern



The Model-View-Controller (MVC) pattern

Web application architecture using the MVC pattern



The Model-View-Controller (MVC) pattern

Case- Study : restaurant

1. Let's assume you go to a restaurant. You will not go to the kitchen and prepare food which you can surely do at your home. Instead, you just go there and wait for the waiter to come on.
2. Now the waiter comes to you, and you just order the food. The waiter doesn't know who you are and what you want he just written down the detail of your food order.

3. Then, the waiter moves to the kitchen. In the kitchen waiter not prepare your food.
4. The cook prepares your food. The waiter is given your order to him along with your table number.
5. Cook then prepared food for you. He uses ingredients to cooks the food. Let's assume that your order a vegetable sandwich. Then he needs bread, tomato, potato, capsicum, onion, bit, cheese, etc. which he sources from the refrigerator

6. Cook final hand over the food to the waiter. Now it is the job of the waiter to moves this food outside the kitchen.
7. Now waiter knows which food you have ordered and how they are served.

The Model-View-Controller (MVC) pattern

Case- Study : **restaurant**

In this case,

You= View
Waiter= Controller
Cook= Model
Refrigerator= Data

Thanks to All