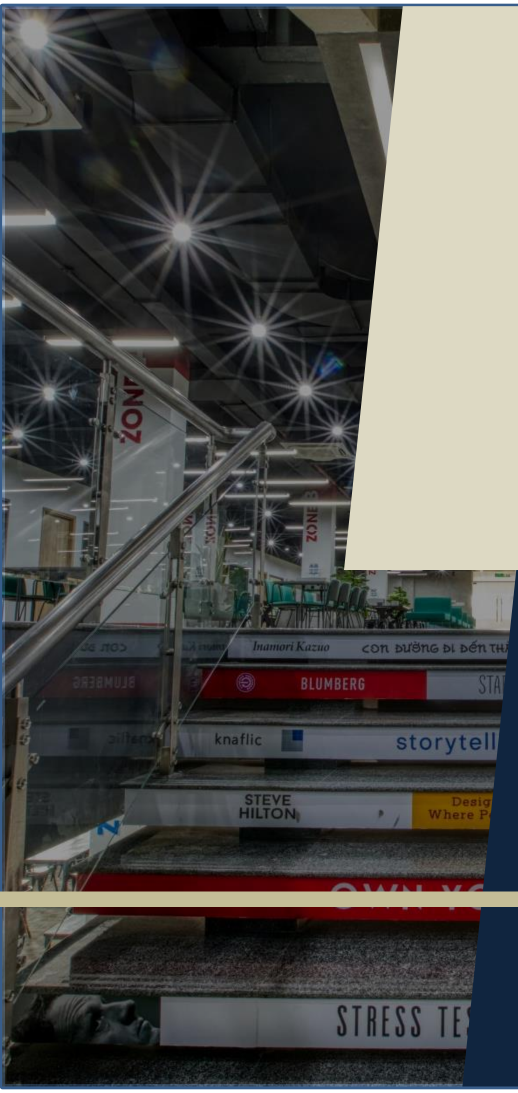


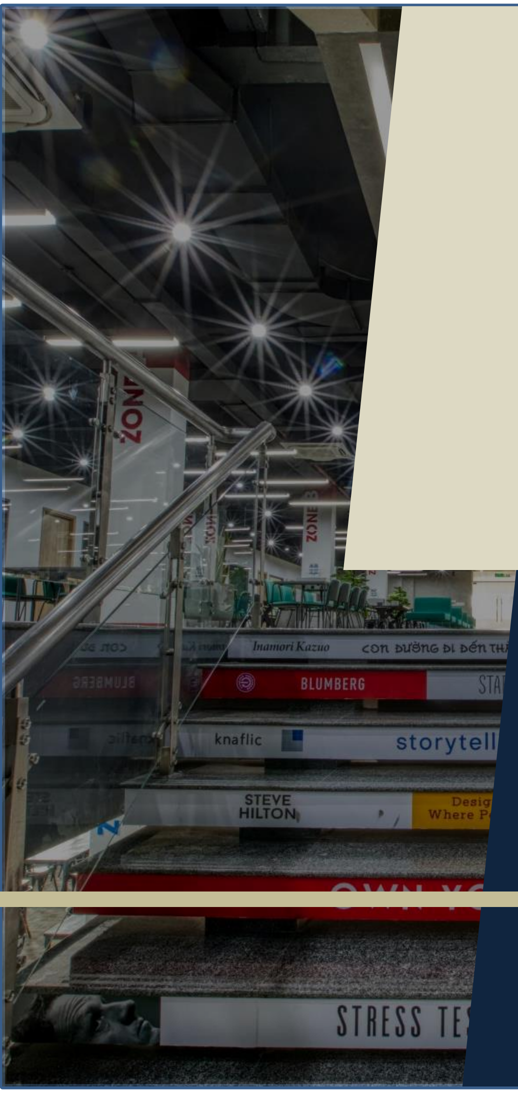
TRƯỜNG ĐẠI HỌC QUỐC TẾ MIỀN ĐÔNG
EASTERN INTERNATIONAL UNIVERSITY

Software Engineering Course's Code: CSE 305

TRƯỜNG ĐẠI HỌC QUỐC TẾ MIỀN ĐÔNG
EASTERN INTERNATIONAL UNIVERSITY

Chapter 6. Software Architecture and Design

6.1. Software Architecture
6.2. Design Concept
6.3. Modularity
6.4. Design Patterns



Software Architecture

- **What is Architectural design ?**

Architectural design is concerned with understanding how a software system should be organized and designing the overall structure of that system

Why :

- **What role does Architectural design play ?**

Architectural design is the first stage in the software design p rocess. It is the critical link between design and requirements e ngineering, as it identifies the main structural components in a system and the relationships between them.

"In agile processes, it is generally accepted that an early stag e of an agile development process should focus on designing a n overall system architecture. Incremental development of arch itectures is not usually successful"

Architectural Styles/Patterns

➢ Architectural styles represent a stylized description of different types of software architecture which has been tried and tested in different environments.

➢ **Types of Architectural Styles / Patterns**

- ❖ Model-View-Controller (MVC)
- ❖ Two Tier
- ❖ Three Tier
- ❖ Event Driven
- ❖ Microservices
- ❖ Peer-to-Peer

Model-View-Controller (MVC)

- **Separates presentation and interaction from the system data.**

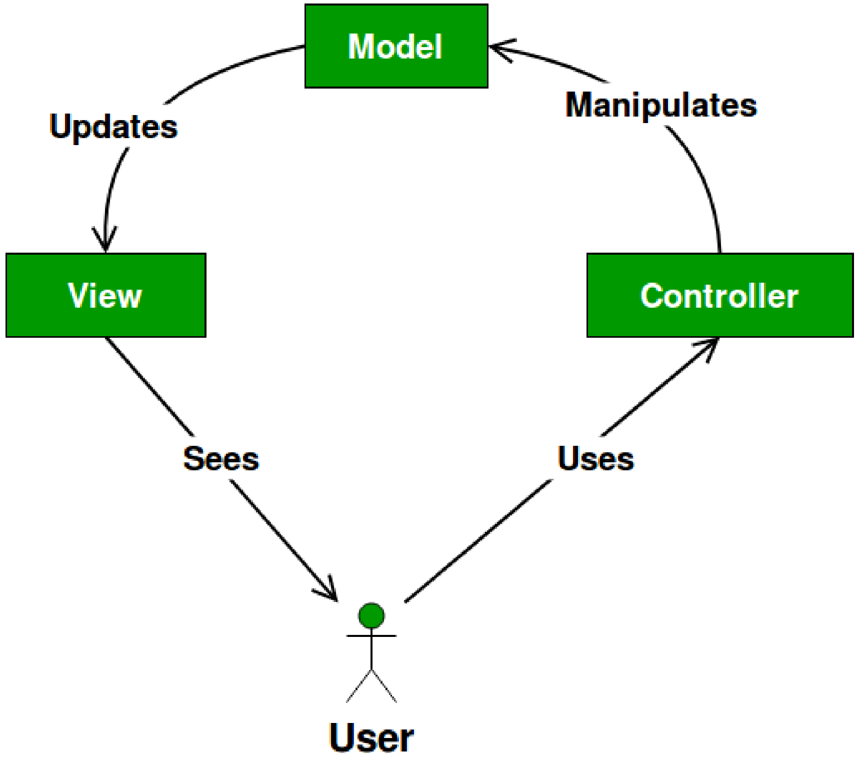
- The system is structured into three logical components that interact with each other.
  - **The Model component** manages the system data and associated operations on that data.
  - **The View component** defines and manages how the data is presented to the user.
  - **The Controller component** manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model

Model-View-Controller (MVC)

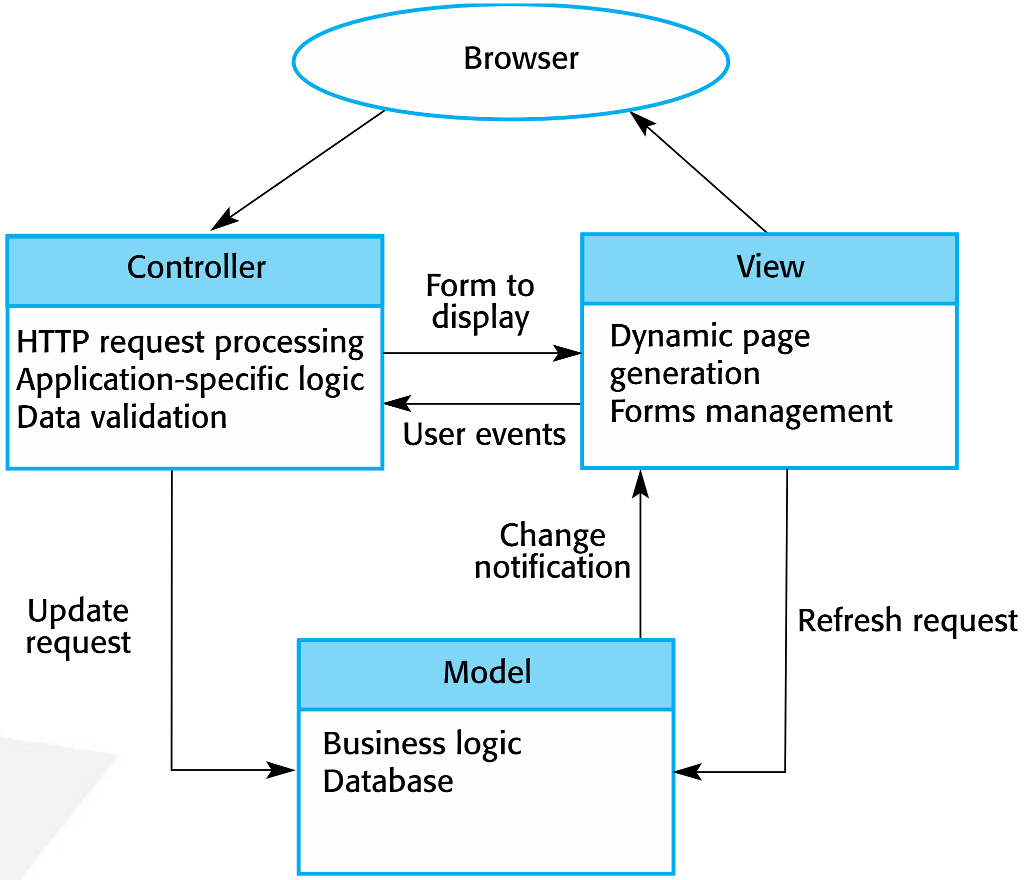
- The **Model** contains only the pure application data, it contains no logic describing how to present the data to a user.

- The **View** presents the model's data to the user. The view knows how to access the model's data, but it does not know what this data means or what the user can do to manipulate it.

- The **Controller** exists between the view and the model. It listens to events triggered by the view (or another external source) and executes the appropriate reaction to these events.



Web application architecture using the MVC



Example of MVC in Java

```
class Student
{ private String rollNo;
 private String name;
public String getRollNo()
{ return rollNo;
}
public void setRollNo(String rollNo)
{ this.rollNo = rollNo;
}
public String getName()
{ return name;
}
public void setName(String name)
{ this.name = name;
}
}
```

Model

```
class StudentView
{
public void printStudentDetails(String st
udentName, String studentRollNo)
{ System.out.println("Student: ");
System.out.println("Name: " + studentN
ame);
System.out.println("Roll No: " + student
RollNo);
}
}
```

View

Example of MVC in Java

```
class StudentController
          { private Student model;
           private StudentView view;

              public StudentController(Student model, StudentView view)
              { this.model = model;  this.view = view; }

              public void setStudentName(String name)
              { model.setName(name);}

              public String getStudentName()
              { return model.getName();}

              public void setStudentRollNo(String rollNo)
              { model.setRollNo(rollNo);}

              public String getStudentRollNo()
              { return model.getRollNo();}

              public void updateView()
              {view.printStudentDetails(model.getName(), model.getRollNo());}
              }
```

Controller

EIU

Example of MVC in Java

```
class MVCPattern
{ public static void main(String[] args)
  { Student model = retriveStudentFromDatabase();
    StudentView view = new StudentView();
    StudentController controller = new StudentController(model, view);
    controller.updateView();
    controller.setStudentName("Vikram Sharma");
     controller.updateView();
   }
private static Student retriveStudentFromDatabase()
  { Student student = new Student();
     student.setName("Lokesh Sharma");
     student.setRollNo("15UCS157");
     return student;
  }
}
```

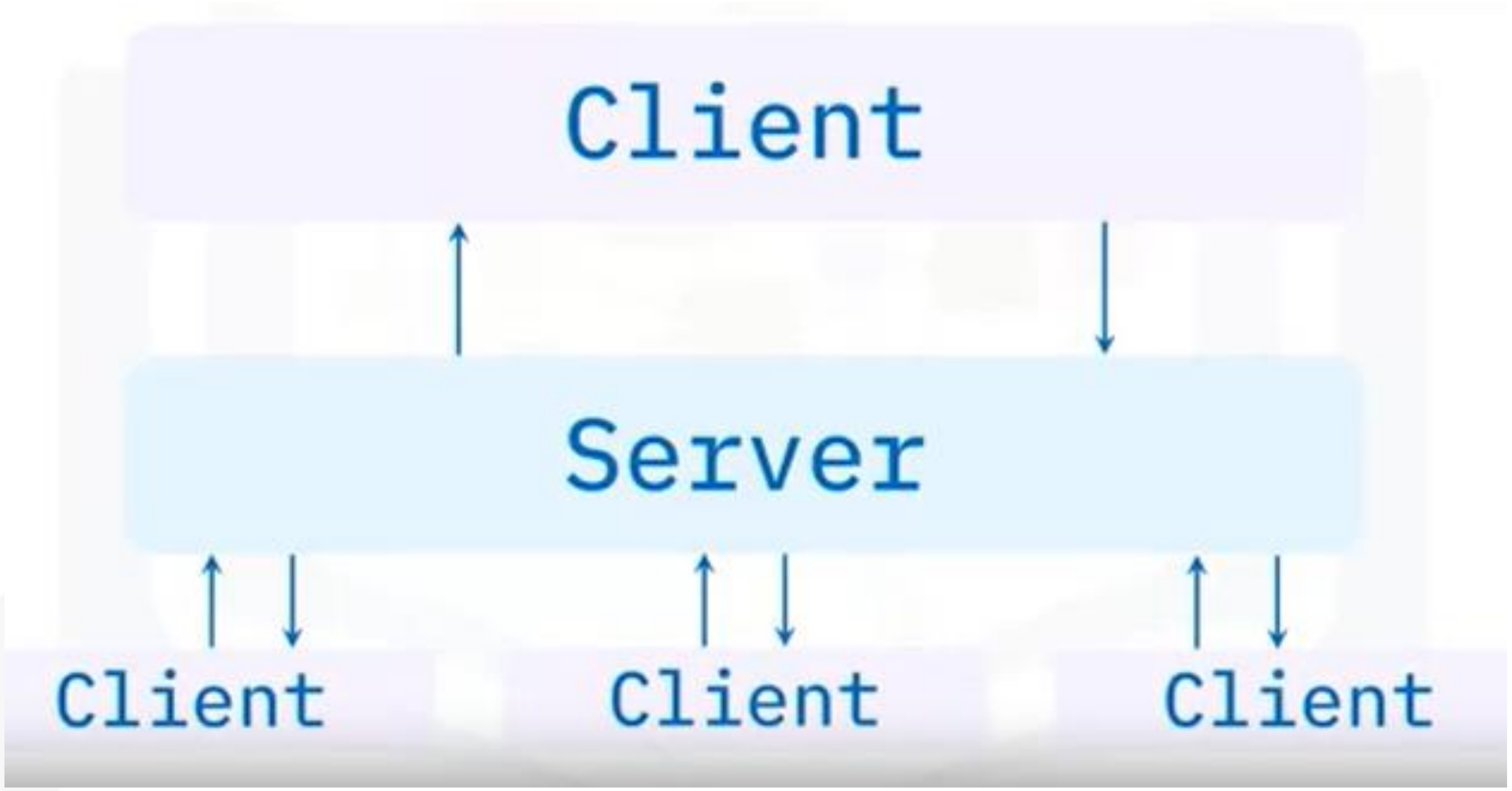
Output

```
Student:
Name: Lokesh Sharma
Roll No: 15UCS157
Student:
Name: Vikram Sharma
Roll No: 15UCS157
```

When to use MVC, Advantage and Disadvantage

When used	**Used when there are multiple ways to view and interact with data.** **Also used when the future requirements for interaction and presentation of data are unknown.**
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

2-tier Architecture



2-tier Architecture

- **Tiers** refer to components that are typically on different physical Machines. Sometimes “Tier” are also referred as layer

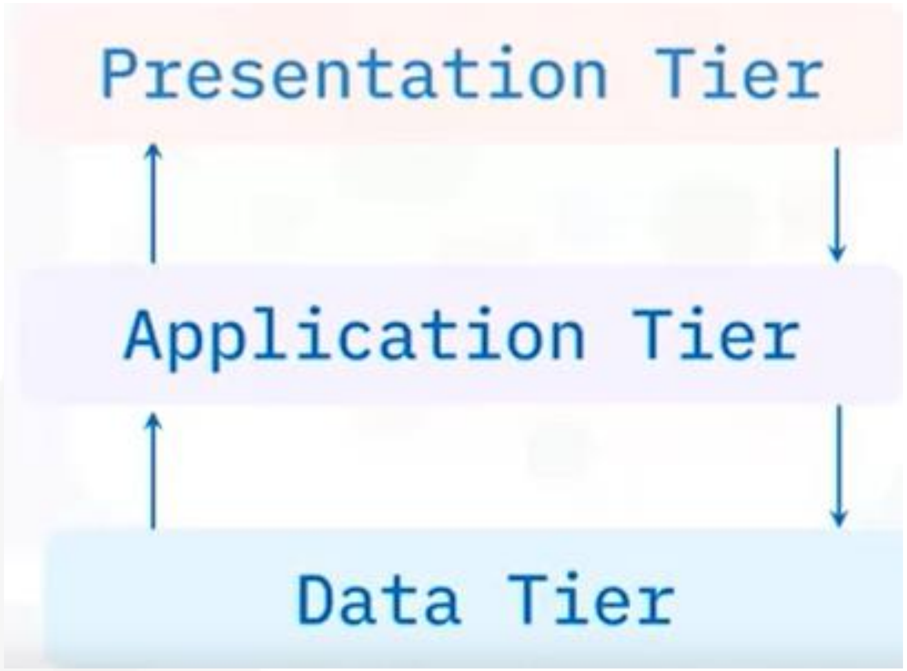
- In a 2-tier architecture, **the server** provides hosts, delivers, and manages most of the resources and services delivered to **the client**.

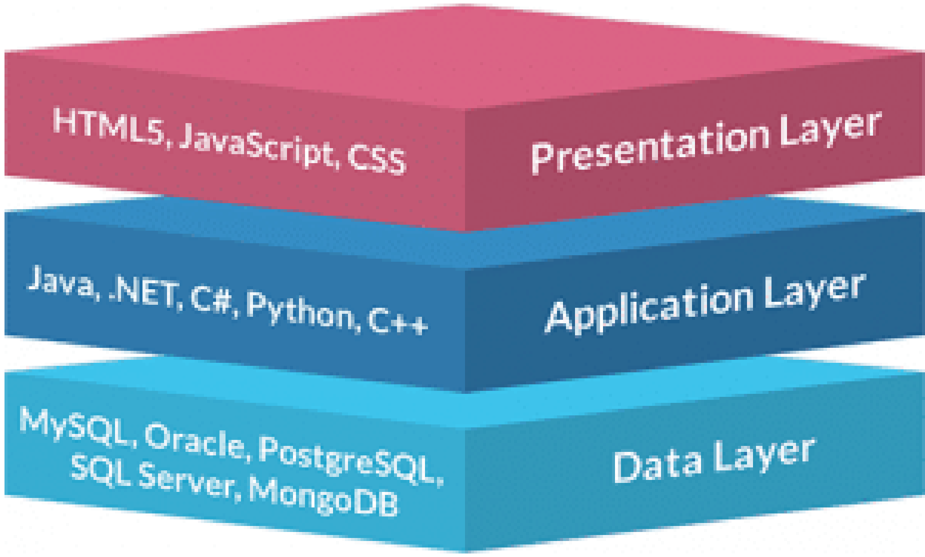
- This communication is known as the **request-response**.

- The interface resides on the client machine and makes requests to a server for data or services

- This type of architecture usually has more than one client computer connected to a server component **over a network connection**.

3-tier Architecture




HTML5, JavaScript, CSS
Presentation Layer
Java, .NET, C#, Python, C++
Application Layer
MySQL, Oracle, PostgreSQL, SQL Server, MongoDB
Data Layer

3-tier Architecture

- **A 3-tier architecture, or an n-tier architecture** when there are more than three layers, is the most common software architecture.

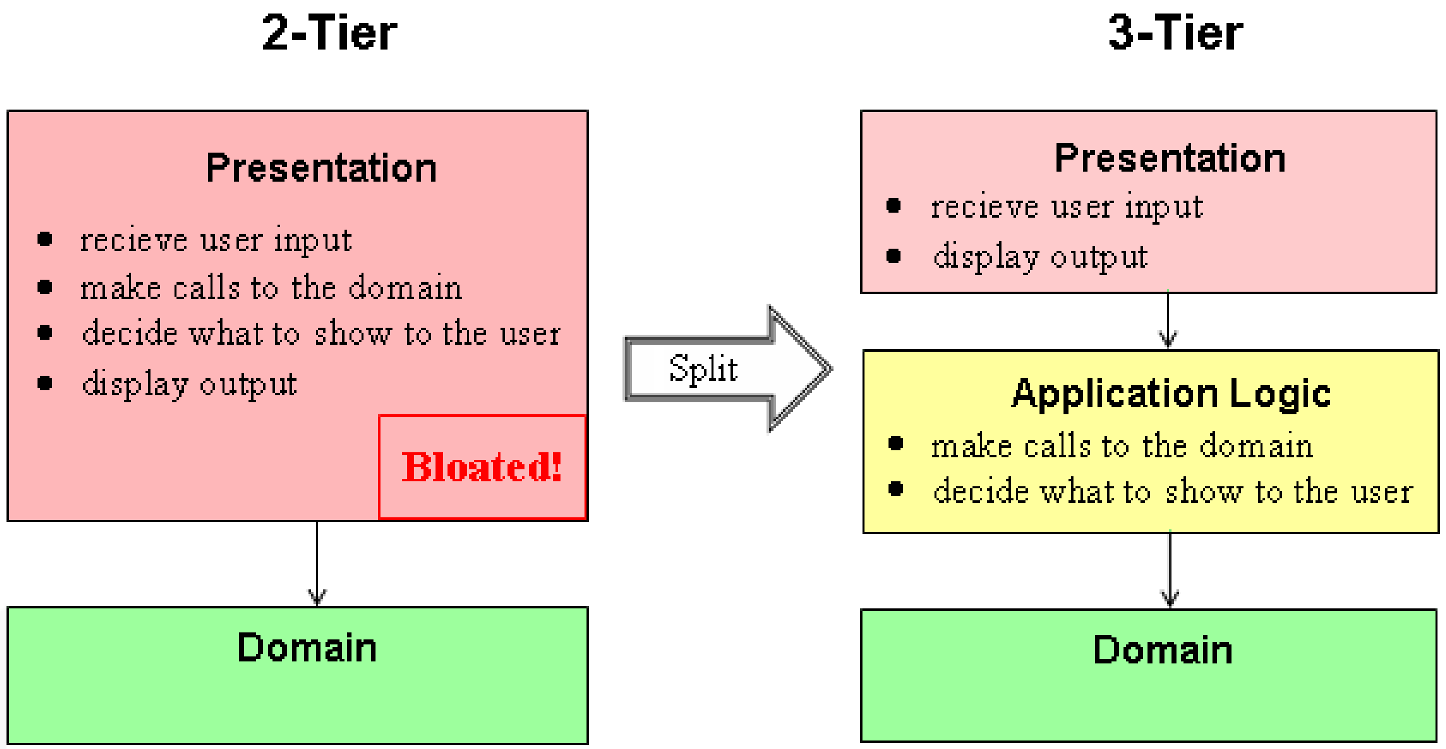
- **The 3-tier architecture** is composed of separate horizontal tiers that function together as a single unit of software.

- **A tier only communicates with other tiers located directly above and below it**.

- Related components are placed within the same tier. Changes in one tier do not affect the other tier.

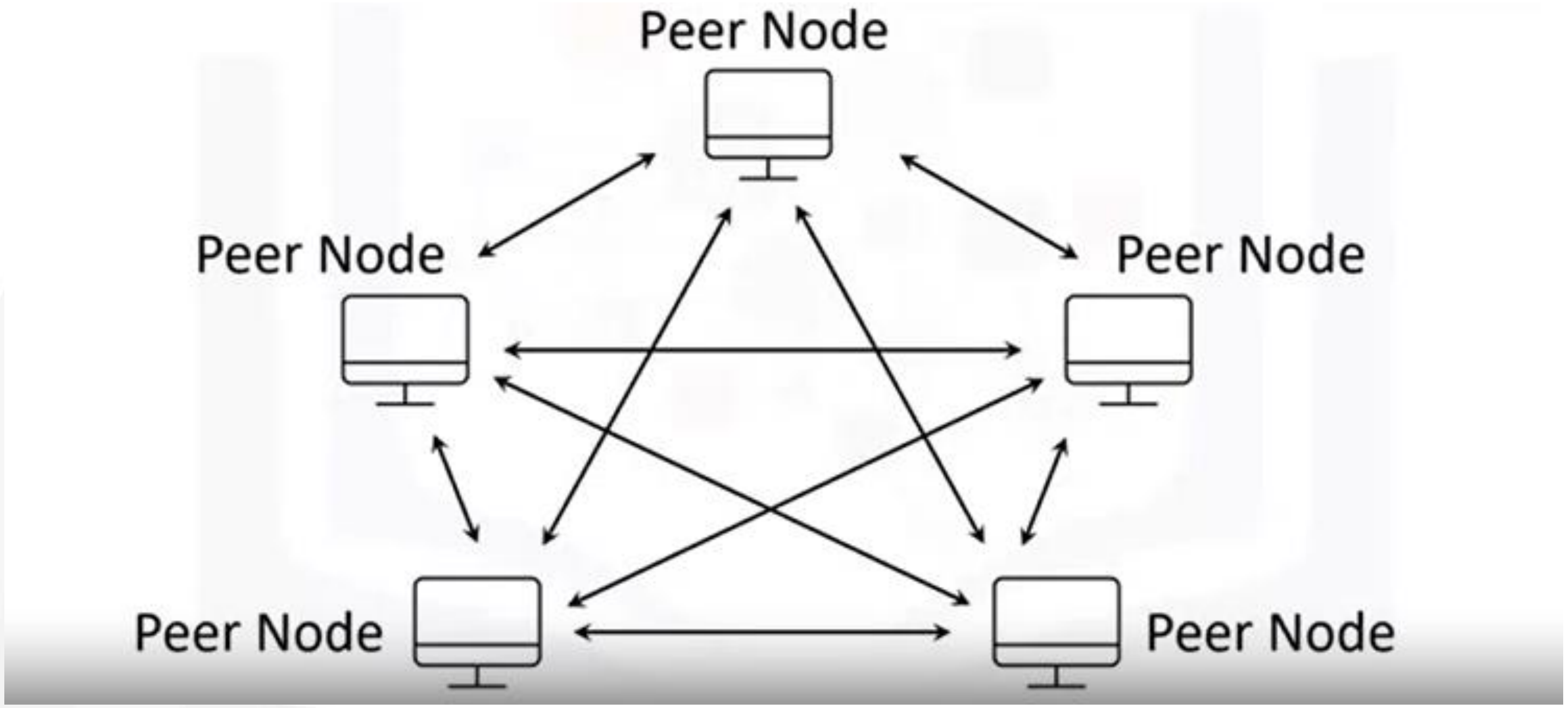
- The 3-tier architecture organizes applications into three logical and physical computing tiers: **the presentation tier**, or user interface; the middle tier which is usually **the application tier**, is where business logic is processed; **the data tier**, where the data is stored and managed

3-tier Architecture



2- tier: These responsibilities are rather vast and, as a syste m grows, may result in a bloated presentation layer.
3- tier: Three-tier architectures are more complex to design and implement than two-tier architectures.

Peer to Peer Architecture



Peer to Peer Architecture

- **The peer-to-peer architecture**, or **P2P for short**, consists of a decentralized network of **nodes that are both clients and servers**.

- The workload is partitioned among these nodes.

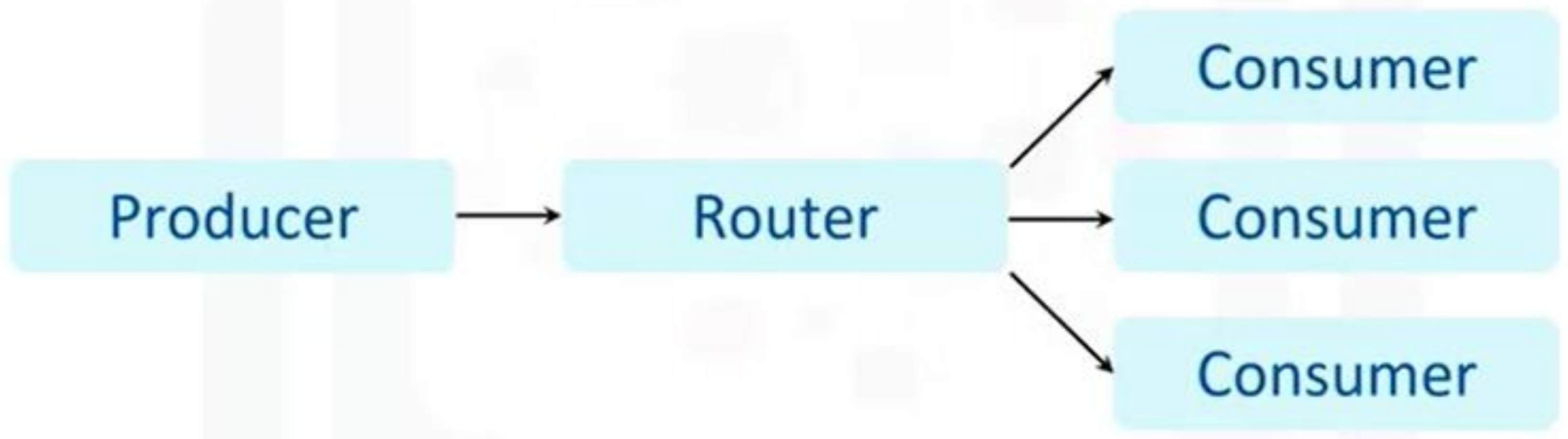
- Peers make a portion of their resources directly available to other network participants, **without the need for central coordination by servers**

- Resources are things like processing power, disk storage, or network bandwidth.

- **Peers both supply and consume resources**, in contrast to the traditional client-server architecture in which the consumption happens strictly by the client and the servers, supply the resources.

- **Peer-to-peer architecture is useful for file sharing, instant messaging, collaboration, and high-performance computing**

Event Driven Architecture



Event Driven Architecture

- **An event** is anything that results in a change of state. An event can be thought of as an action that is triggered by the end-user,
such as a mouse click

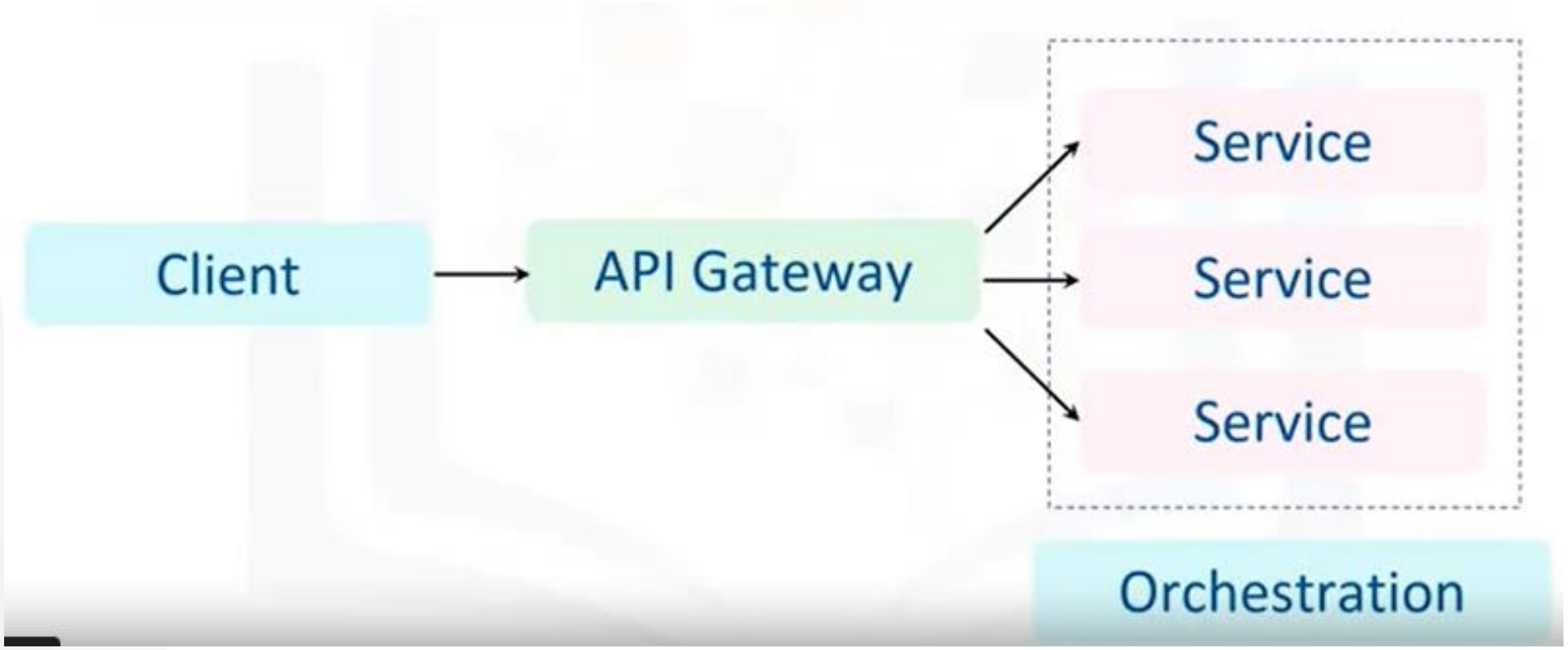
- **Event-driven architecture** focuses on producers and consumers of events. Producers listen for and react to triggers while consumers process an event.

- **The producer** publishes the event to **an event router**. The router determines which consumer to push the event to.

- The triggering event generates a message, called an event notification, to **the consumer** which is listening for the event.

- The components in event-driven architectures are loosely coupled making the pattern appropriate for use with modern, distributed systems

Microservices architecture



Microservices architecture

- **Microservices** are an approach to building an application that breaks its functionality into modular components called services.

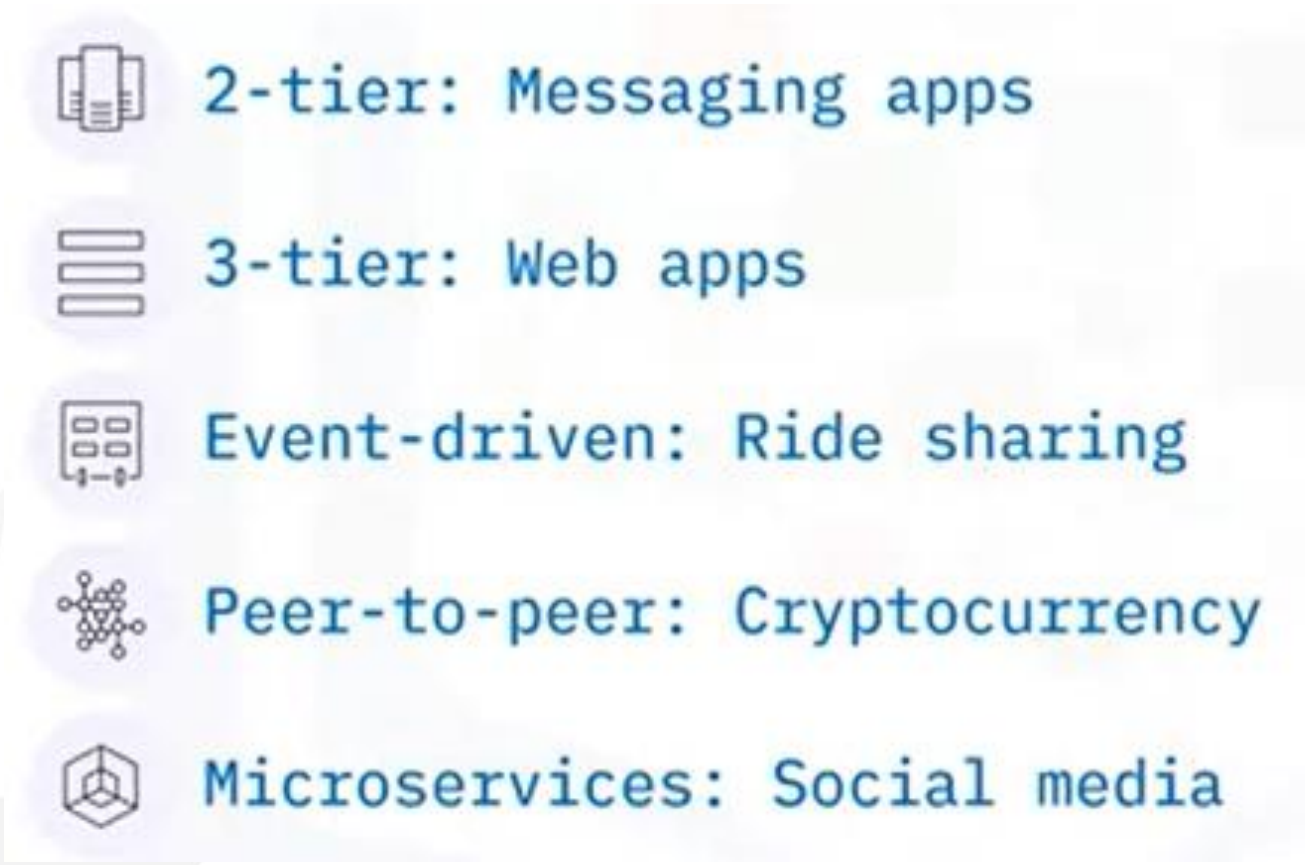
- **An application programming interface, also called an API**, is the part of an application that communicates with other applications.

- An API defines how two applications share and modify each other's data. APIs can be used to create a microservices-based architecture.

- The API Gateway routes the API from the client to a service.

- **Orchestration** handles communication between services.

Examples



- 2-tier: Messaging apps
- 3-tier: Web apps
- Event-driven: Ride sharing
- Peer-to-peer: Cryptocurrency
- Microservices: Social media

Examples

- A text messaging app is an example of a 2-tier pattern. The client initiates a request to send a text message through a server and the server responds by sending that message to another different client. Another example of the 2-tier pattern, is Database clients connecting with database servers.

- Many web apps use the 3-tier pattern. They use a web server to provide the user interface, an application server to process user inputs, and a database server that handles data management.

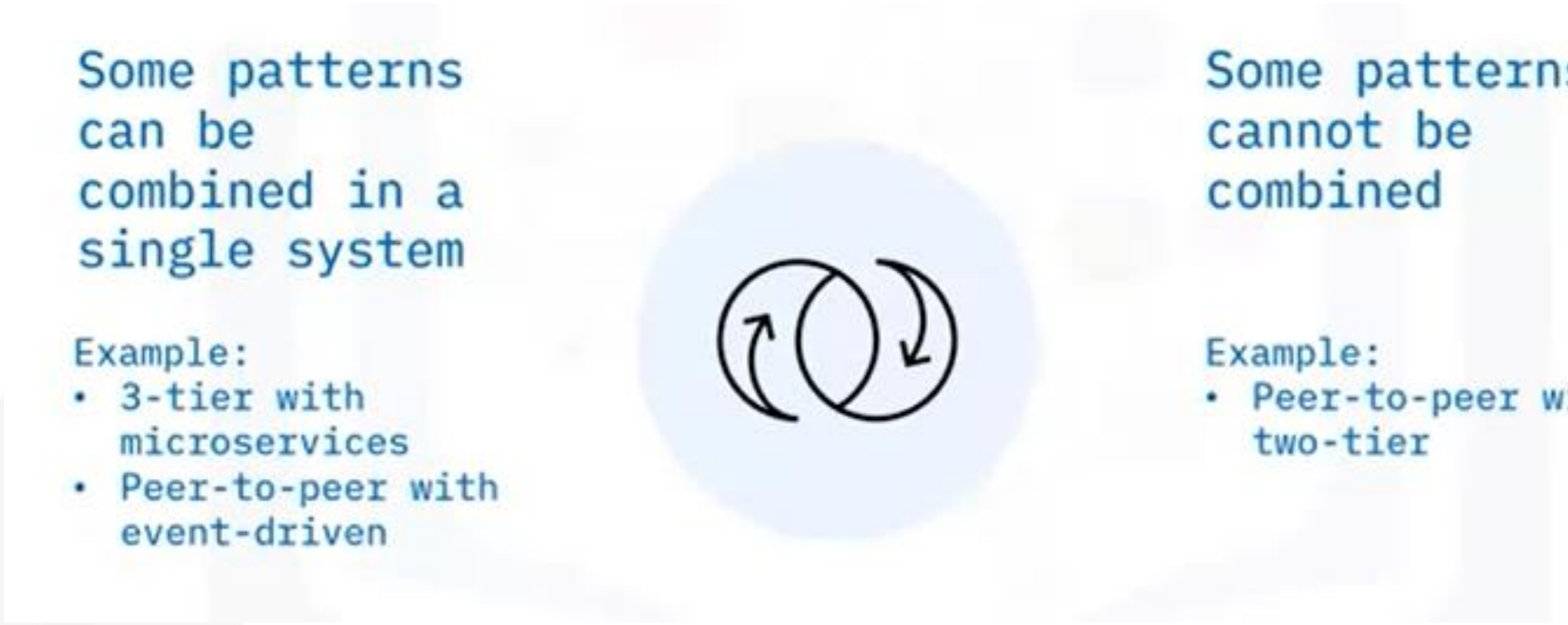
- Ride-sharing apps such as Grab and Uber are examples of event-driven patterns. The customer sends a notification that they need a ride from a particular location to another location, and that event is routed to a consumer.

Examples

- Cryptocurrencies such as Bitcoin and Ethereum use a peer-to-peer pattern. Each computer in the blockchain acts as both server and client.

- Social media sites are composed of microservices. A user has an account. That account can request different services such as adding friends, targeted ad recommendations, and displaying content.

Combining Architectural Patterns

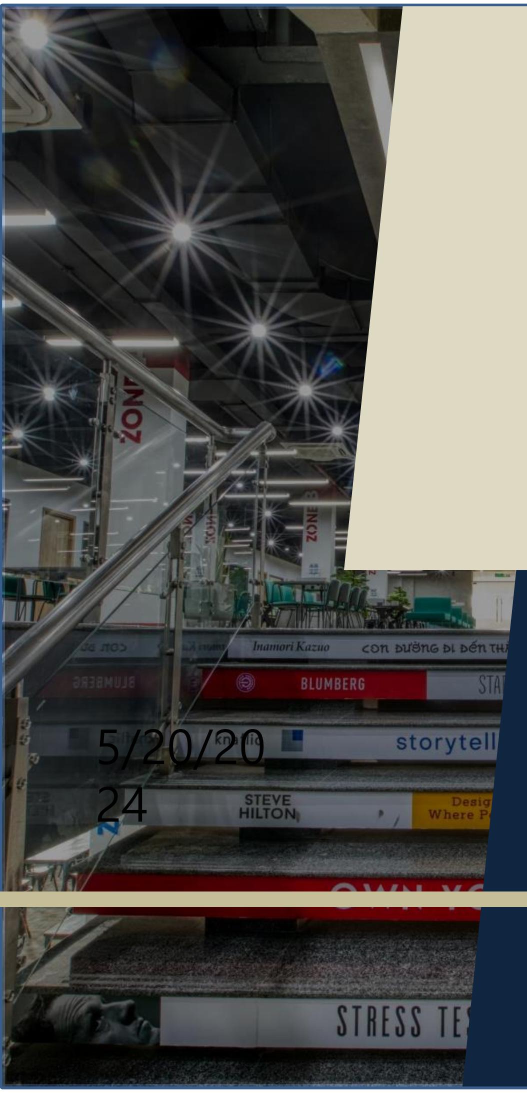


Some patterns can be combined in a single system

Example:
- 3-tier with microservices
- Peer-to-peer with event-driven

Some patterns cannot be combined

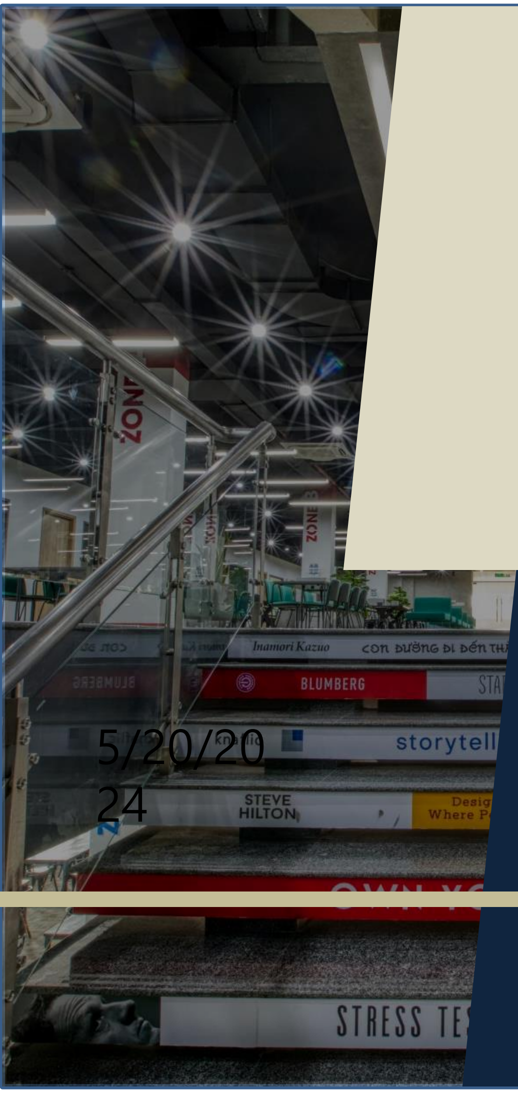
Example:
- Peer-to-peer with two-tier



TRƯỜNG ĐẠI HỌC QUỐC TẾ MIỀN ĐÔNG
EASTERN INTERNATIONAL UNIVERSITY

Software Engineering Course's Code: CSE 305

5/20/2024



5/20/2024

EIU
TRƯỜNG ĐẠI HỌC QUỐC TẾ MIỀN ĐÔNG
EASTERN INTERNATIONAL UNIVERSITY

Chapter 6. Software Architecture and Design

6.1. Software Architecture
6.2. Software Design
6.3. Modularity
6.4. Design Patterns

What is Software Design?

1. Requirements Gathering: The process begins with gathering and documenting the requirements for the software. This phase in volves understanding the needs of the stakeholders and defining what the software should do.

2. Software Architecture: Once the requirements are gathered, t he next step is to design the high-level structure and organization of the software system. Common architectural patterns and desig n principles are applied at this stage.

3. Software Design: Following the architectural phase, software design comes into play. Software design is a more detailed and fi ne-grained process that focuses on designing individual compone nts, modules, and classes within the system.

4.Implementation: After the software design is complete, the act ual coding and implementation of the software take place.

Object-oriented system design

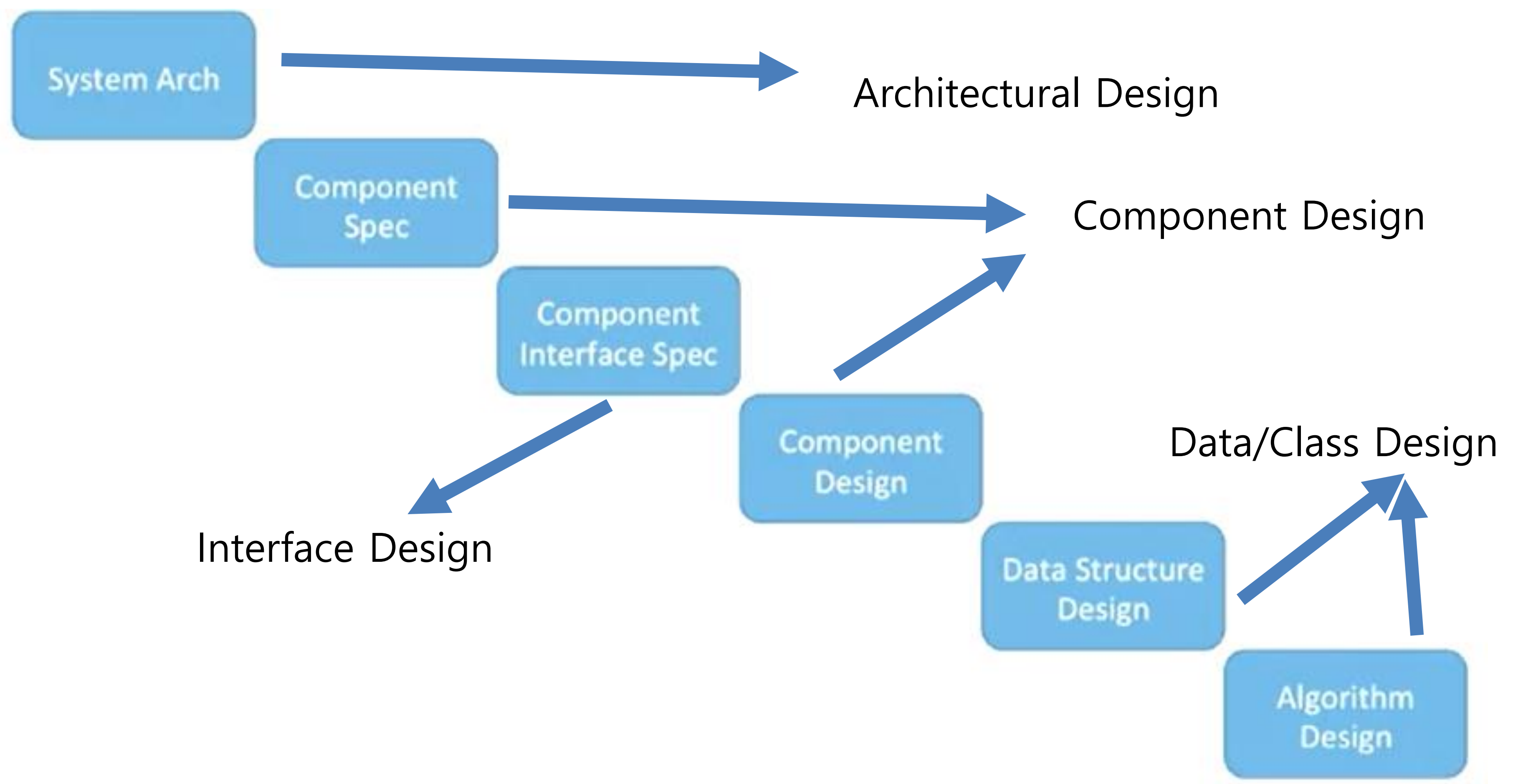
➢ The **architectural design** defines the relationship between major structural elements of the software, **the architectural styles and patterns**

➢ The **component-level design** transforms structural elements of the software architecture into **more detailed description of software components**

➢ The **interface design** describes how the **different components of a software communicates** with each other, how **a software communicate with another software** that interoperate with it, and **with end users who use it**.

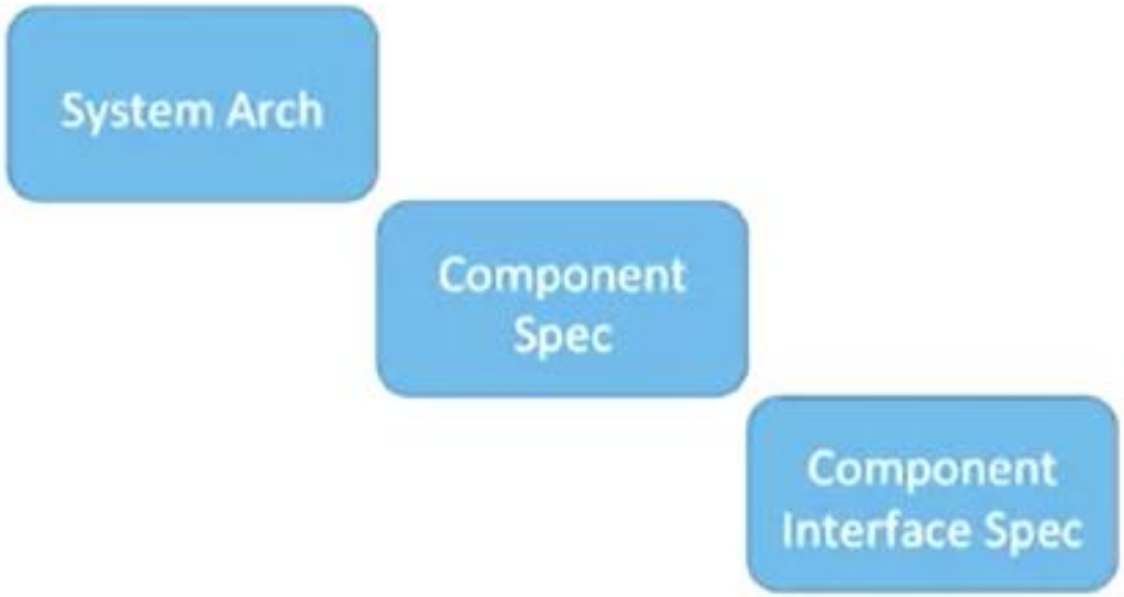
➢ The **data/class design** transforms class models into design class realizations and the **requisite data structures and algorithms** required to implement the software.

Stages of Design



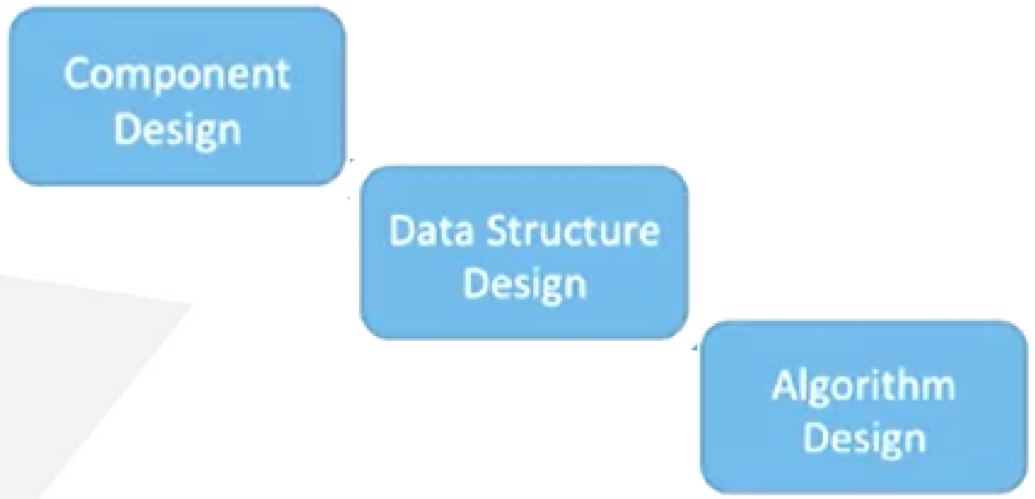
Stages of Design

- ➢ In **architecture and design, we follow these six stages**.
  - ❑ The first three are architectural and the last three are design.
- ➢ In the **first three steps we decide on a system architecture**, separate behavior responsibility into components, and determine how those components will interact through their interfaces, .



Stages of Design

- Then **we set out to design** the individual components
- Each component is designed in isolation
- Once each component is fully designed in isolation, any **data structures** which are inherently complex, important, or shared between the classes, or even shared between components, are then designed for efficiency.
- The same goes for **algorithms**. When the algorithm is particularly complex, novel, or important to the successful fulfillment of the components' required behavior, software designers rather, than the developers, are writing pseudo code to ensure that the algorithm is properly built



Question?

Software design is the process of transforming the stated problem into a ready-to-use implementation.

a) True

b) False

Answer is False

While a solution coming from software design does not include implementation details, there are still common cases where pseudo code may be provided to correctly capture the sense of a complex algorithm.

a) True

b) False

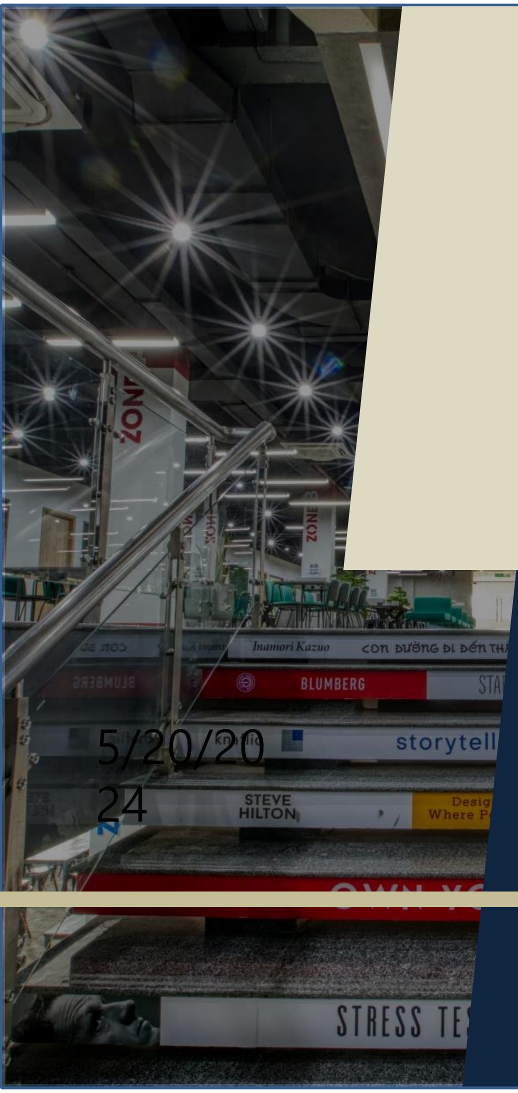
Answer is True

Where does software design fit in the traditional waterfall software development lifecycle?.

a) Between architecture and implementation

b) Between specification and architecture

Answer is a



TRƯỜNG ĐẠI HỌC QUỐC TẾ MIỀN ĐÔNG
EASTERN INTERNATIONAL UNIVERSITY

Chapter 6. Software Architecture and Design

6.1. Software Architecture
6.2. Software Design
6.3. Modularity
6.4. Design Patterns

5/20/2024

Object Oriented Design (OOD): Modularity

Modularity is a fundamental attributes of any good design. It is based on Divide and conquer principle.

Complex systems MUST be broken down into smaller parts

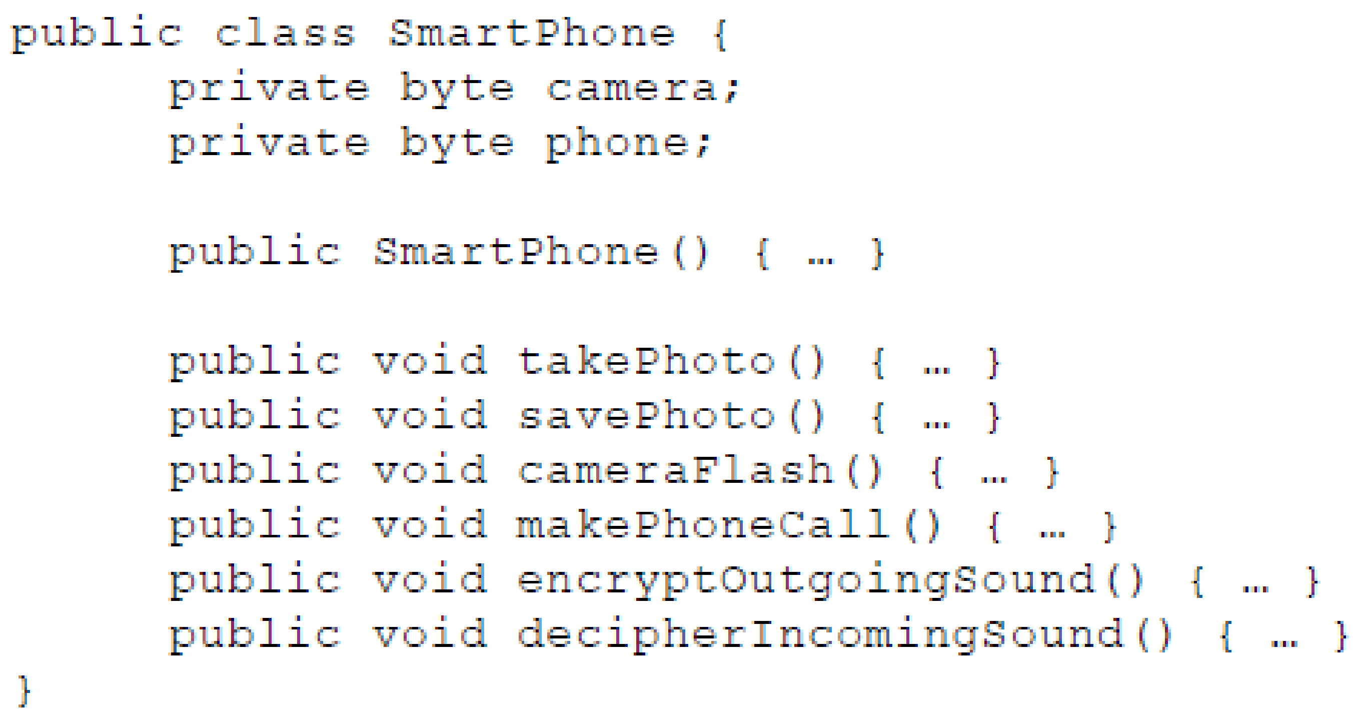
Three primary goals:

1. Decomposability When the problem is too large and complex to get a proper handle on it, breaking it down into smaller parts until you can solve the smaller part. Then you just solve all the smaller parts

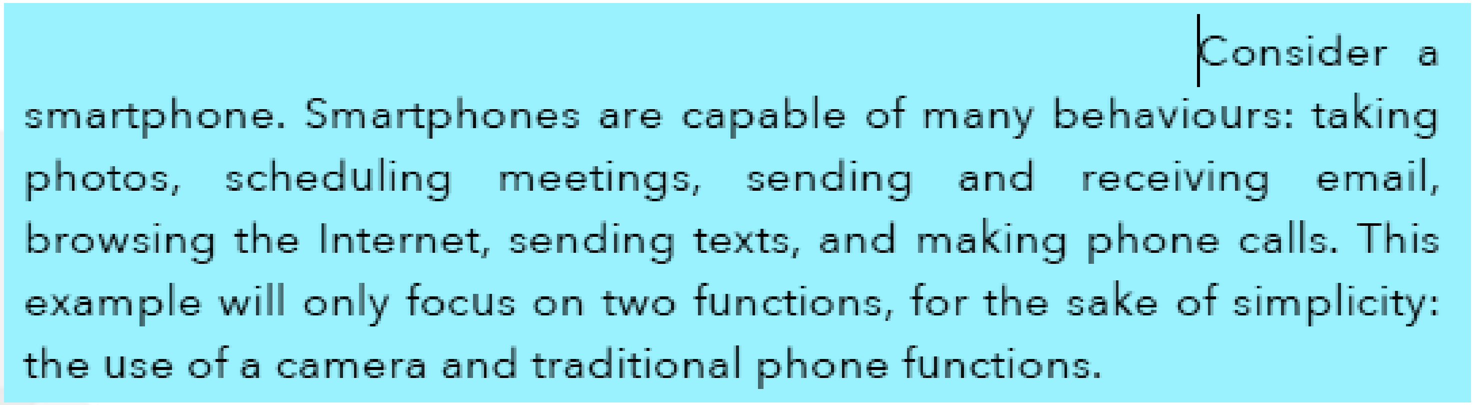
2. "Composability" But then we have to put all those smaller parts back together and that's where composability comes into play.

3. Ease of understanding By breaking down the components we hope to provide an ease of understanding, which will then hopefully lead to an ease of communication

Example of Modularity

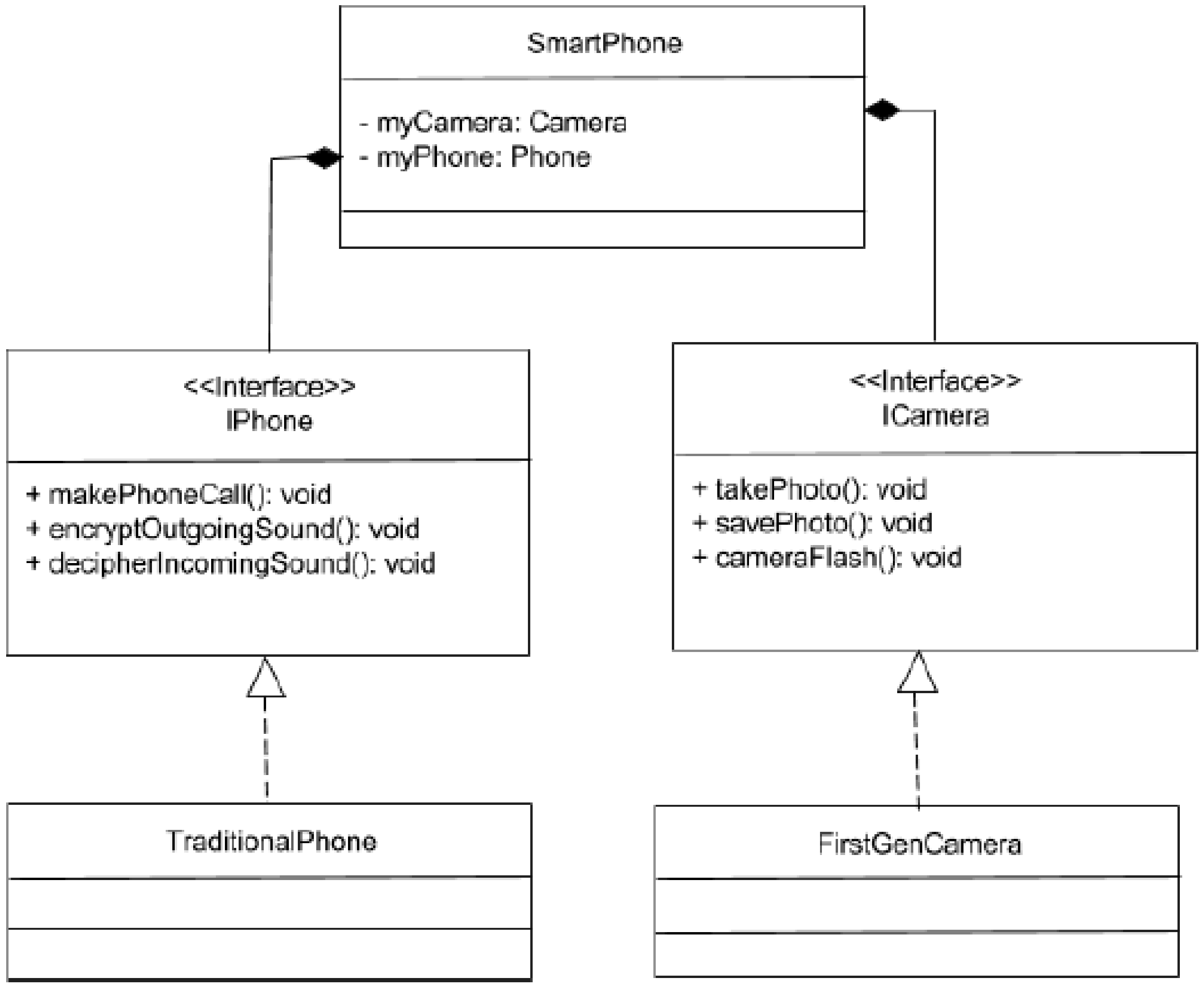


```
public class SmartPhone {
         private byte camera;
         private byte phone;

         public SmartPhone() { … }

         public void takePhoto() { … }
         public void savePhoto() { … }
         public void cameraFlash() { … }
         public void makePhoneCall() { … }
         public void encryptOutgoingSound() { … }
         public void decipherIncomingSound() { … }
}
```

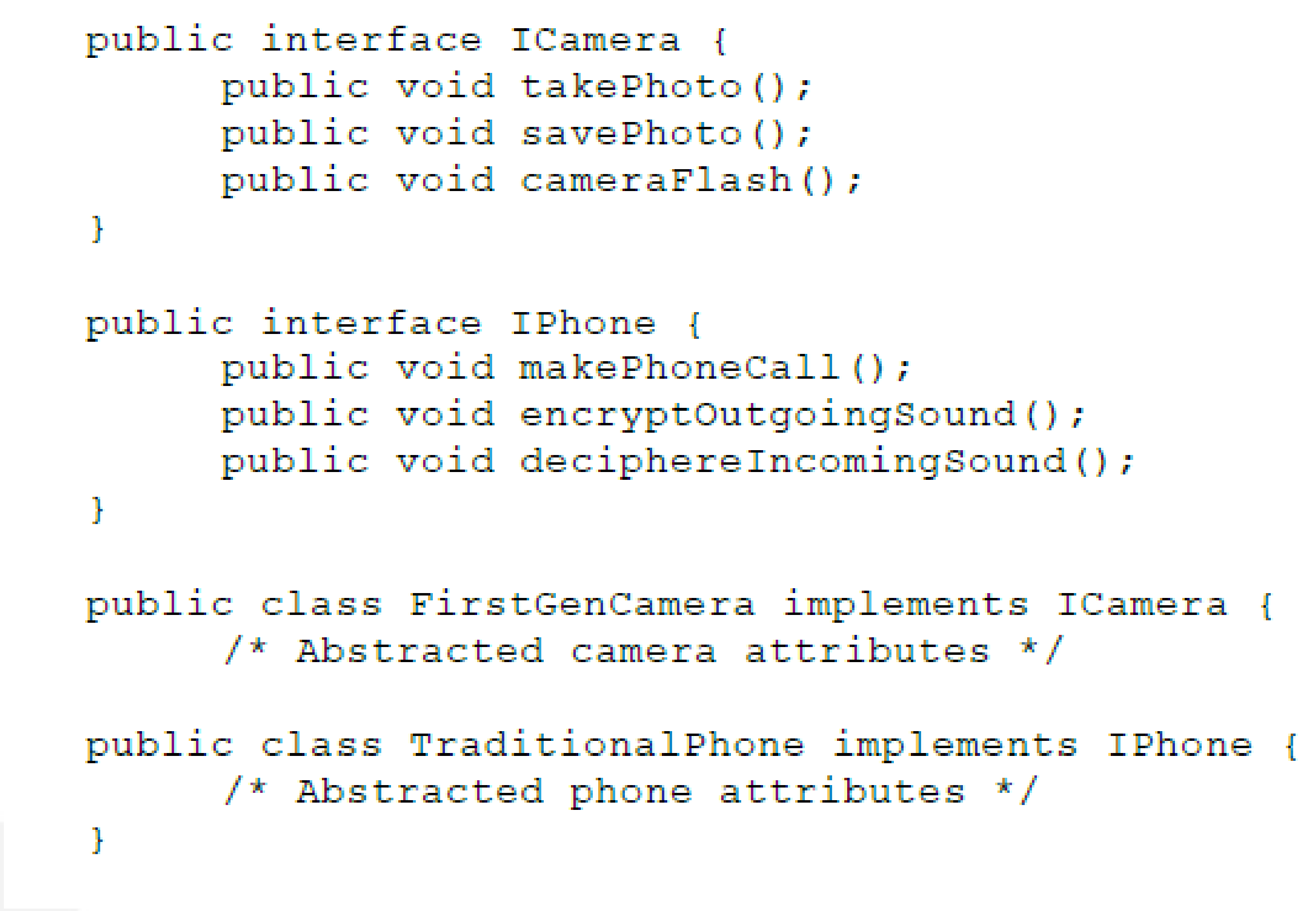


Consider a smartphone. Smartphones are capable of many behaviours: taking photos, scheduling meetings, sending and receiving email, browsing the Internet, sending texts, and making phone calls. This example will only focus on two functions, for the sake of simplicity: the use of a camera and traditional phone functions.

Example of Modularity



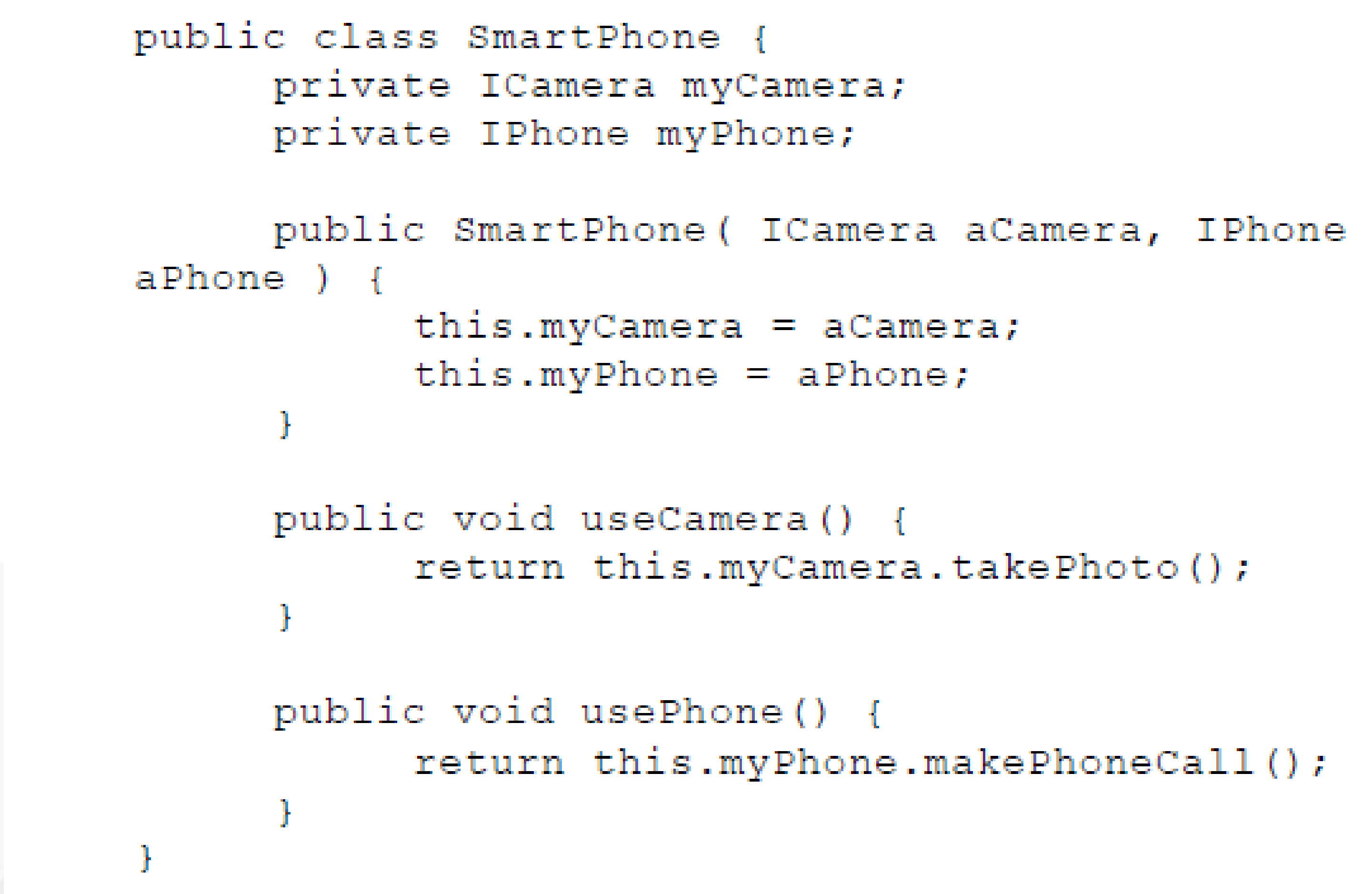
Example of Modularity



```
public interface ICamera {
     public void takePhoto();
     public void savePhoto();
     public void cameraFlash();
}

public interface IPhone {
     public void makePhoneCall();
     public void encryptOutgoingSound();
     public void decipherеIncomingSound();
}

public class FirstGenCamera implements ICamera {
     /* Abstracted camera attributes */

public class TraditionalPhone implements IPhone {
     /* Abstracted phone attributes */
}
```

- Icamera is one module.
- Iphone is another module

Example of Modularity



```
public class SmartPhone {
        private ICamera myCamera;
        private IPhone myPhone;

        public SmartPhone( ICamera aCamera, IPhone
aPhone ) {
                this.myCamera = aCamera;
                this.myPhone = aPhone;
        }

        public void useCamera() {
                return this.myCamera.takePhoto();
        }

        public void usePhone() {
                return this.myPhone.makePhoneCall();
        }
}
```

- SmartPhone is another Module

Aspects/ Features of Modularity

Coupling	Coupling are measures of how well modules work together
Cohesion	Cohesion are measures how well each individual module meets a certain single well-defined task
Information Hiding	Information hiding describes our ability to abstract away information and knowledge in a way that allows us to complete complex work in parallel without having to know all the implementation details concerning how the task will be completed eventually
Data Encapsulation	Data Encapsulation refers to the idea that we can contain constructs and concepts within a module allowing us to much more easily understand and manipulate the concept when we're looking at it in relative isolation.

Information Hiding

Hide complexity in a “black box”

Examples: Functions, macros, classes, libraries

An example of information hiding

```
void sortAscending (int *array, int length)
```

Don’t know which sort is uses

Don’t really need to

Know how to use it

Data Encapsulation

Encapsulate the data

Protecting the data from unauthorized access and m aintaining integrity is a key point.

Helps find where problems are

The developer of a module has the best idea of how and when the attributes should be modified, and then we try to allow them to maintain as much co ntrol as is possible.

Nobody else is allowed to mess with that data. If it gets corrupted, it must have been done by the Module.

Makes designs more robust

This means chances are that new additions aren't going to break the current design.

Question?

Which of the four aspects of modularity is defined as: How well modules work together.

a) Coupling
b) Cohesion
c) Information Hiding
d) Data Encapsulation

Answer is a

Which of the four aspects of modularity can be described as: Containment of constructs and concepts within a module.

a) Coupling
b) Cohesion
c) Information Hiding
d) Data Encapsulation

Answer is d

The ability to use a built-in function of a programming language to generate a random number is an example of which of the following?

a) Coupling
b) Modularity
c) Information hiding
d) Cohesion

Answer is c

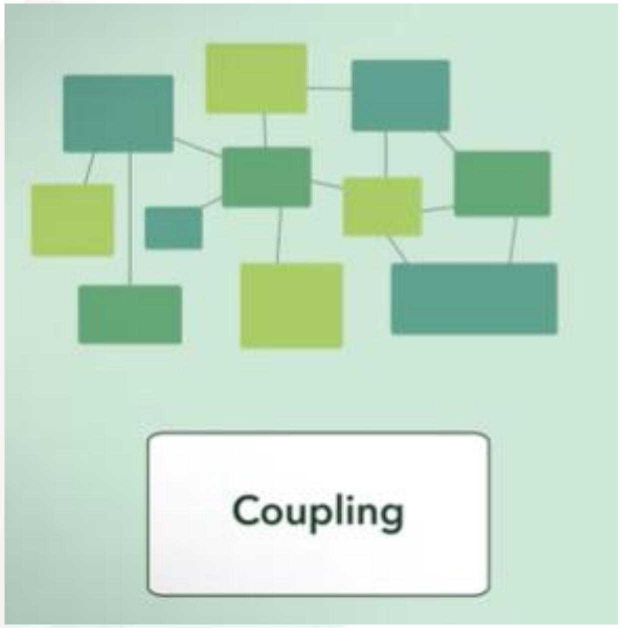
Coupling and Cohesion

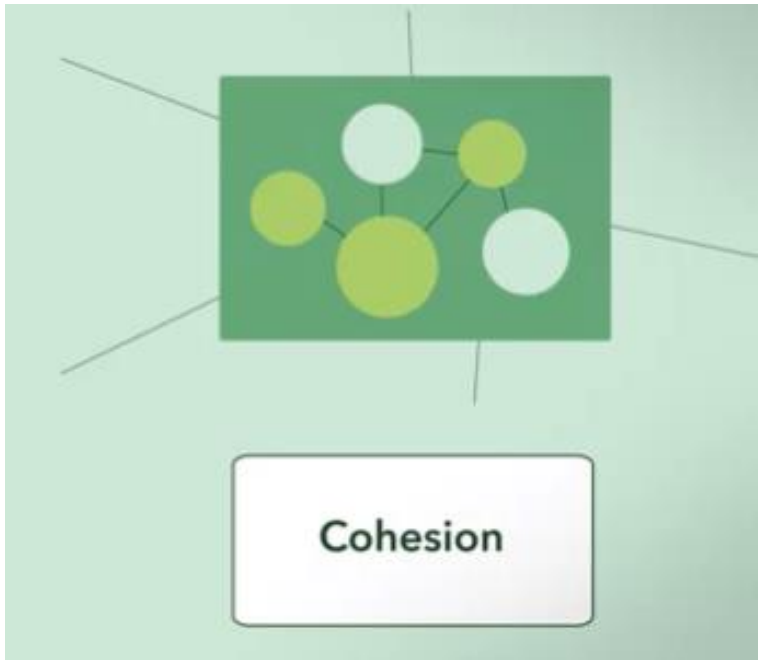
Cohesion is a measure of:

- functional strength of a module.
- A cohesive module performs a single task or function.

Coupling between two modules:

- A measure of the degree of the interdependence or interaction between the two modules.





Coupling and Cohesion

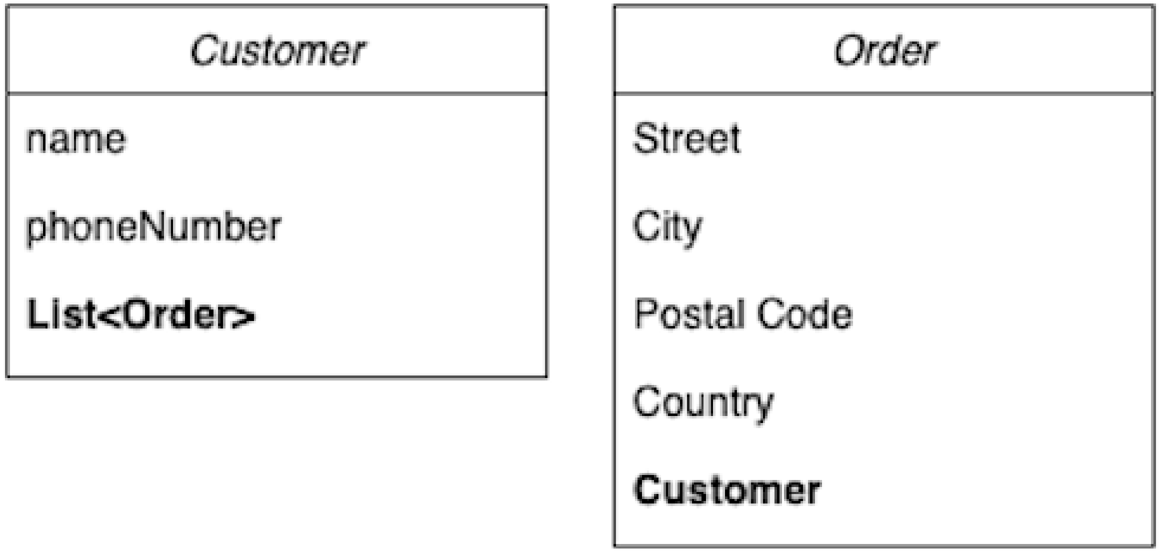
Goal of Good Modularity and also of Software Design is:

A module should have high cohesion and low coupling:

- functionally independent of other modules:
  - A functionally independent module has minimal interaction with other modules.

Coupling

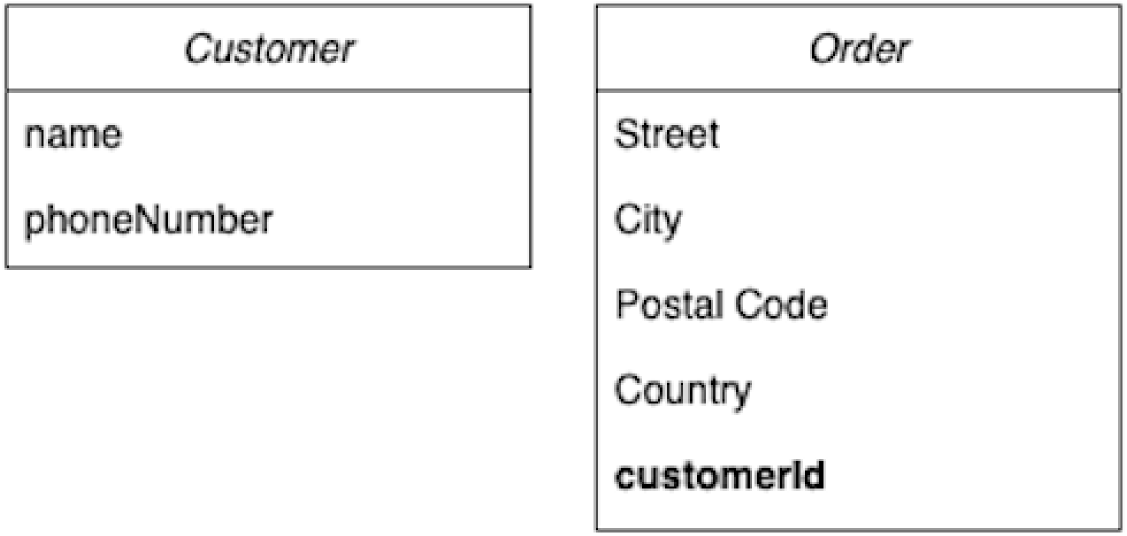
Two modules have high coupling (or tight coupling) if they are closely connected. For example, two concrete classes storing references to each other and calling each other's methods. As shown in the diagram below, *Customer* and *Order* are tightly coupled to each other. The *Customer* is storing the list of al the orders placed by a customer, whereas the *Order* is storing the reference to the *Customer* object.



Tight Coupling

Coupling

Every time the customer places a new order, we need to add it to the order list present inside the *Customer*. This seems an unnecessary dependency. Also, *Order* only needs to know the customer identifier and does need a reference to the *Customer* object. We could make the two classes loosely coupled by making these changes:



Loose Coupling

Question?

Module A relies directly on local data of module B. This is an example of what type of coupling?

a) Tight Content Coupling
b) Tight Common Coupling
c) Tight External Coupling

Answer: a

Cohesion

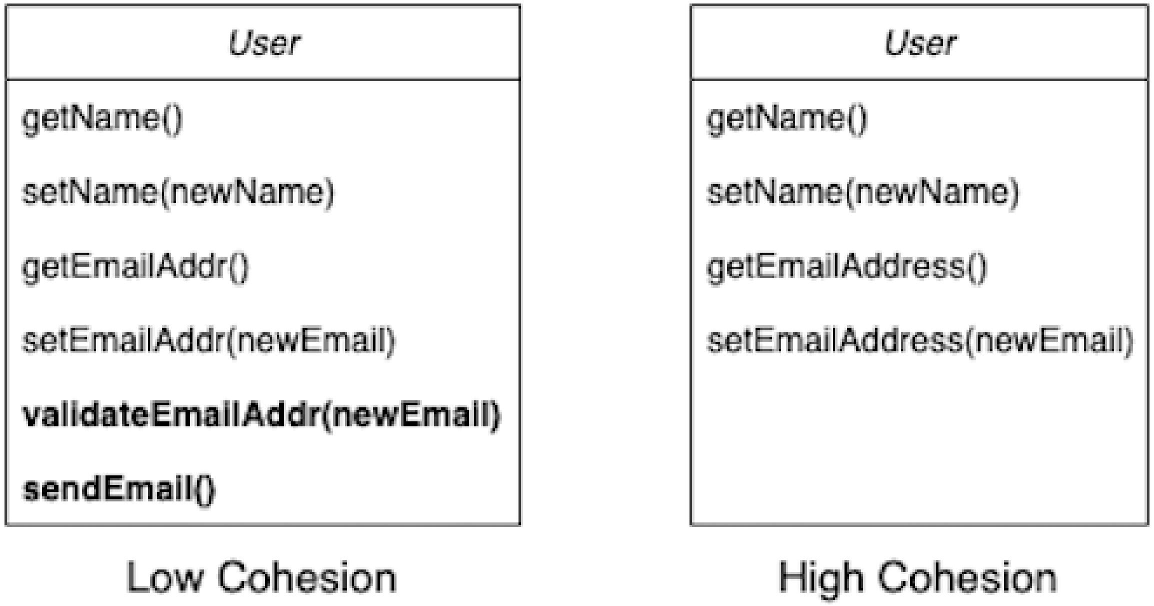
Measures how well a module's components 'fit together'

Implements a single logical entity or function

Divides into various levels of strength

Cohesion

A module is said to have low cohesion if it contains unrelated elements. For example, a *User* class containing a method on how to validate the email address. *User* class can be responsible for storing the email address of the user but not for validating it or sending an email:


User
getName()
setName(newName)
getEmailAddr()
setEmailAddr(newEmail)
validateEmailAddr(newEmail)
sendEmail()
Low Cohesion
User
getName()
setName(newName)
getEmailAddress()
setEmailAddress(newEmail)
High Cohesion