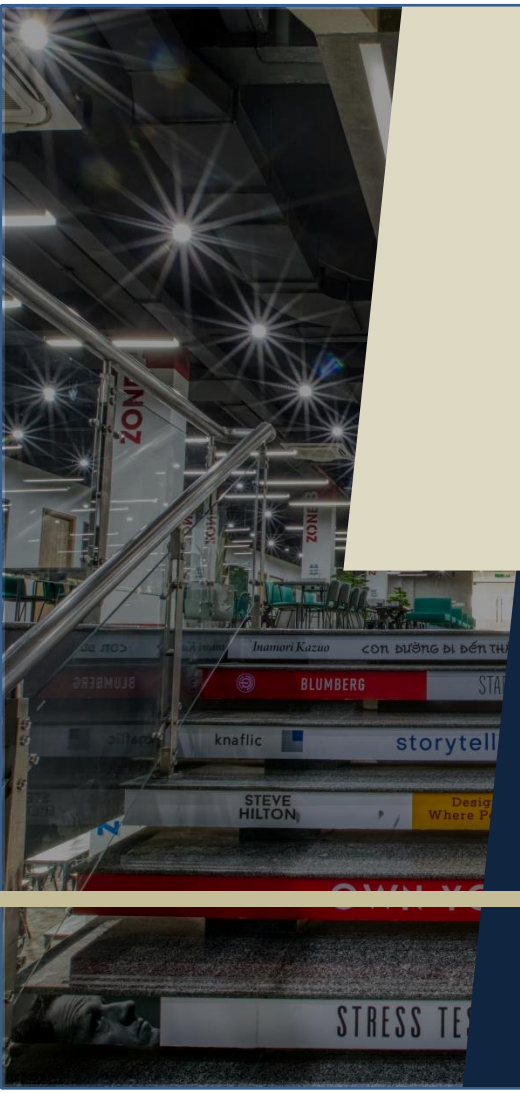
EIU
TRƯỜNG ĐẠI HỌC
QUỐC TẾ
MIỀN ĐÔNG
EASTERN
INTERNATIONAL
UNIVERSITY

Software Engineering
Course's Code: CSE 305



EIU
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 process. It is the critical link between design and requirements engineering, 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 stage of an agile development process should focus on designing an overall system architecture. Incremental development of architectures 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.**

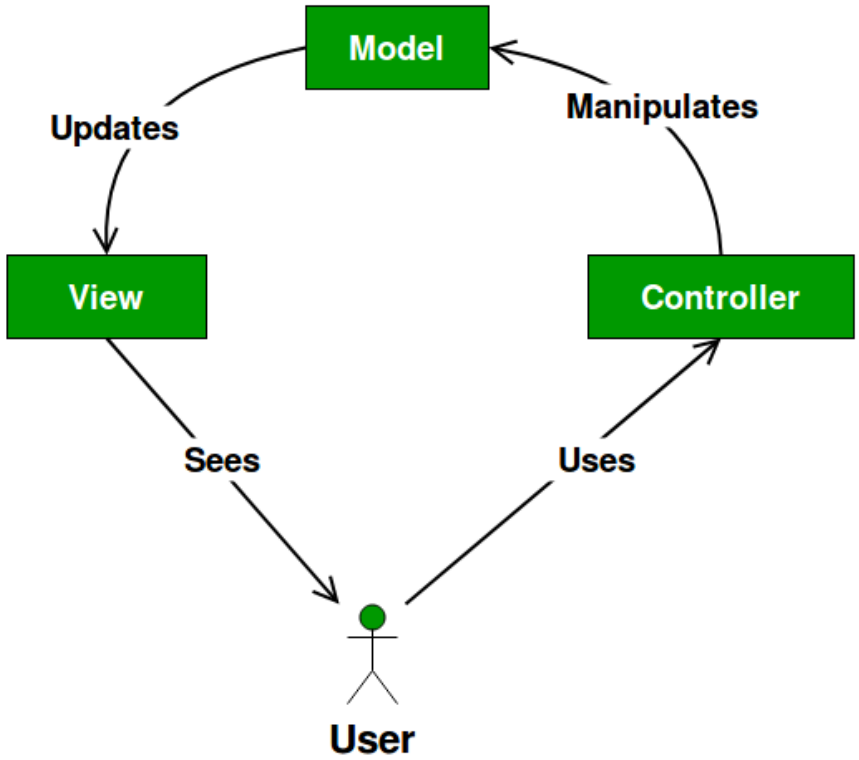
- The system is structured into three logical components that interact with each other.
  - **The Model component** manages the system data and associated operations on that data.
  - **The View component** defines and manages how the data is presented to the user.
  - **The Controller component** manages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

Model-View-Controller (MVC)

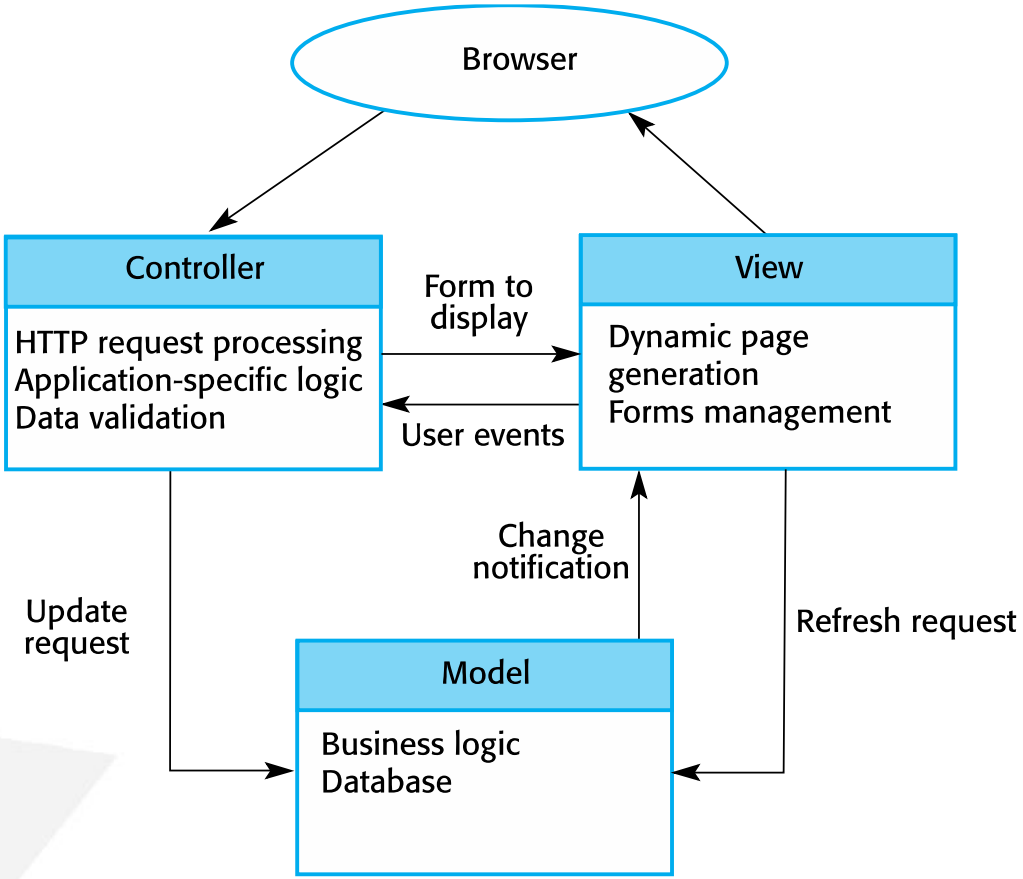
- The **Model** contains only the pure application data, it contains no logic describing how to present the data to a user.

- 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

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;
  }
}
```

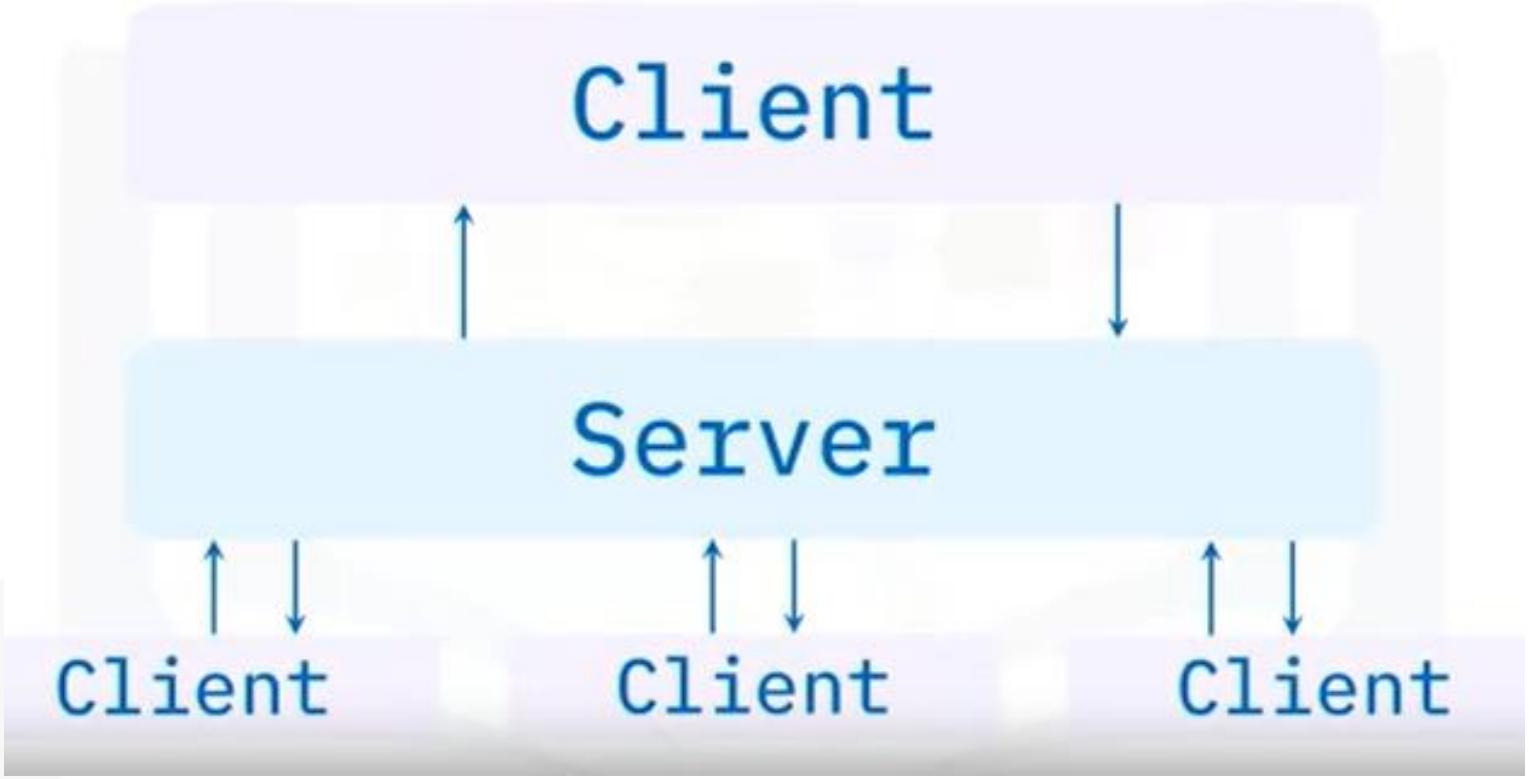
Output

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

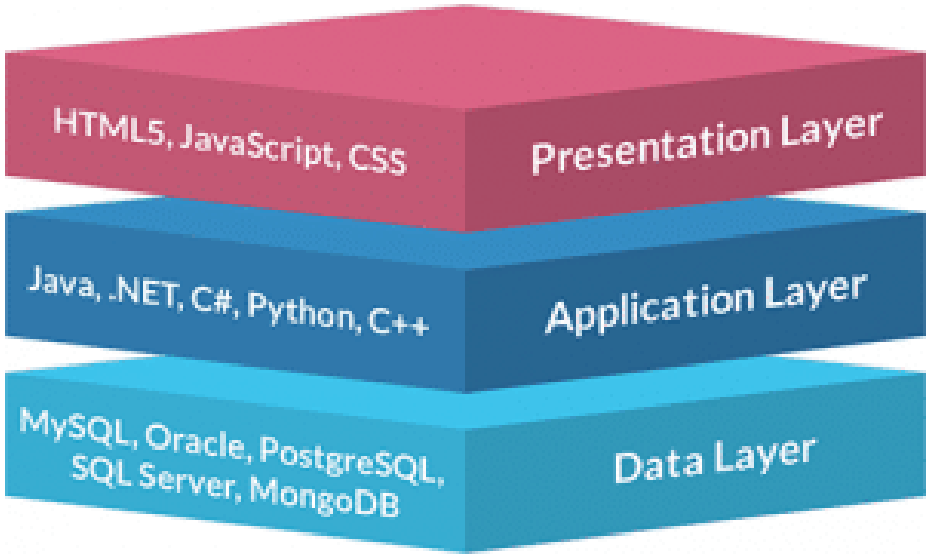
- In a 2-tier architecture, **the server** provides hosts, delivers, and manages most of the resources and services delivered to **the client**.

- 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



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.

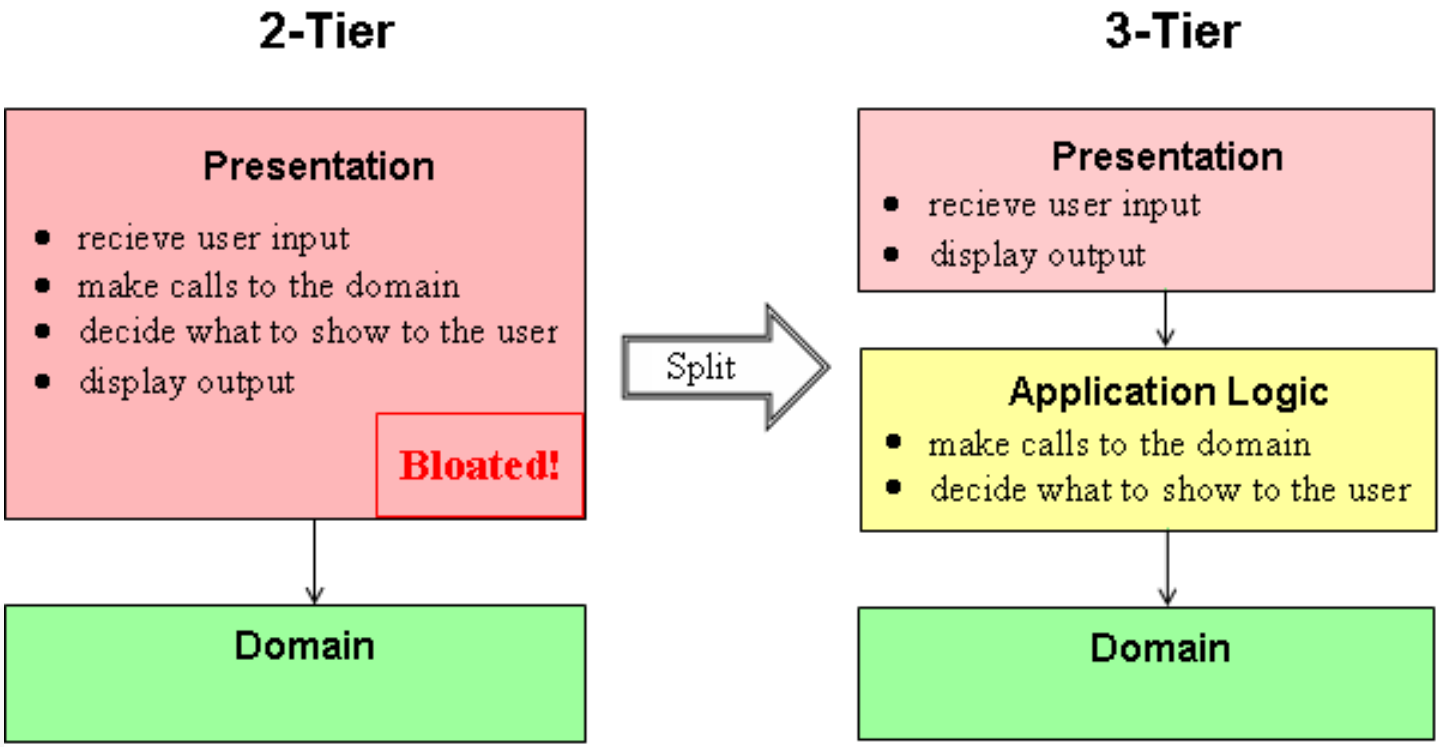
- **The 3-tier architecture** is composed of separate horizontal tiers that function together as a single unit of software.

- **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

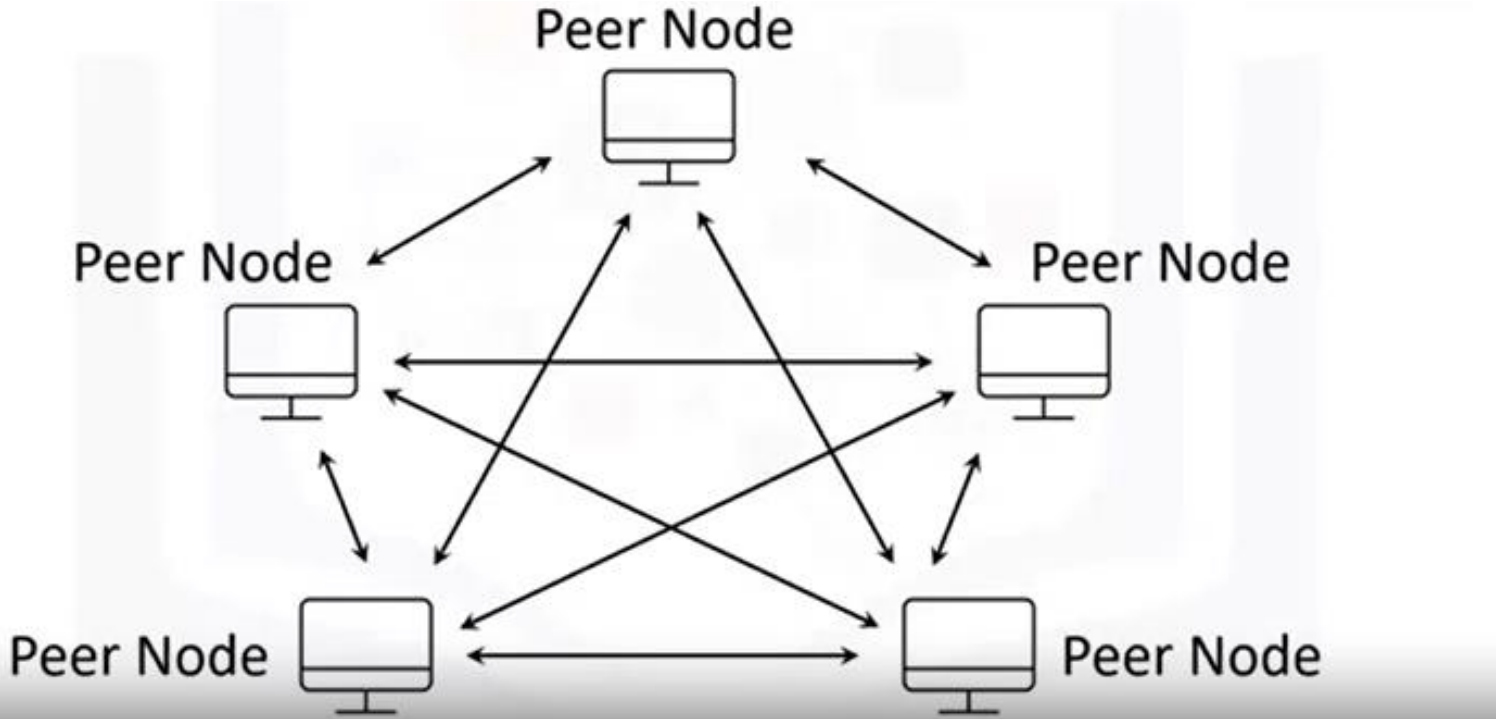
3-tier Architecture



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

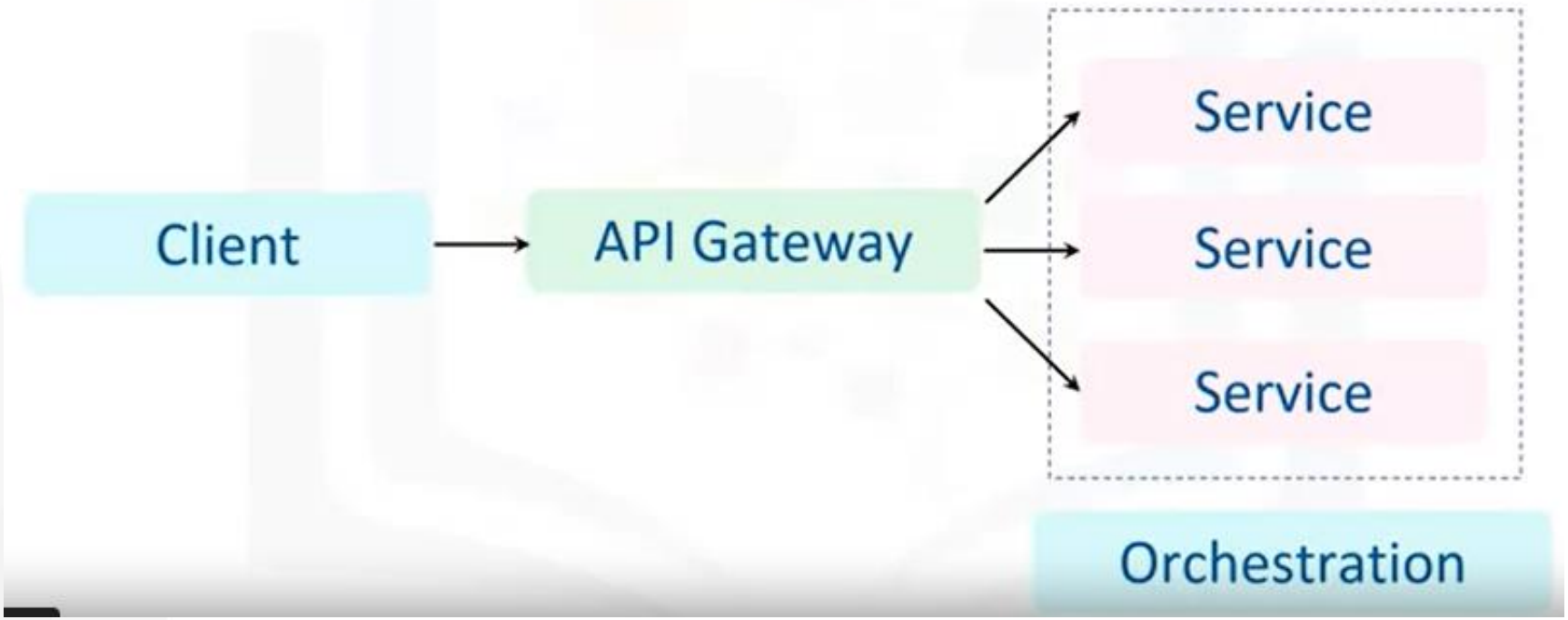
- **Event-driven architecture** focuses on producers and consumers of events. Producers listen for and react to triggers while consumers process an event.

- **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

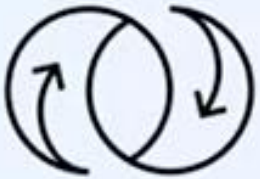
- Cryptocurrencies such as Bitcoin and Ethereum use a peer-to-peer pattern. Each computer in the blockchain acts as both server and client.

- 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

EIU
TRƯỜNG ĐẠI HỌC
QUỐC TẾ
MIỀN ĐÔNG
EASTERN
INTERNATIONAL
UNIVERSITY

Software Engineering
Course's Code: CSE 305

10/4/2024

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

10/4/2024

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

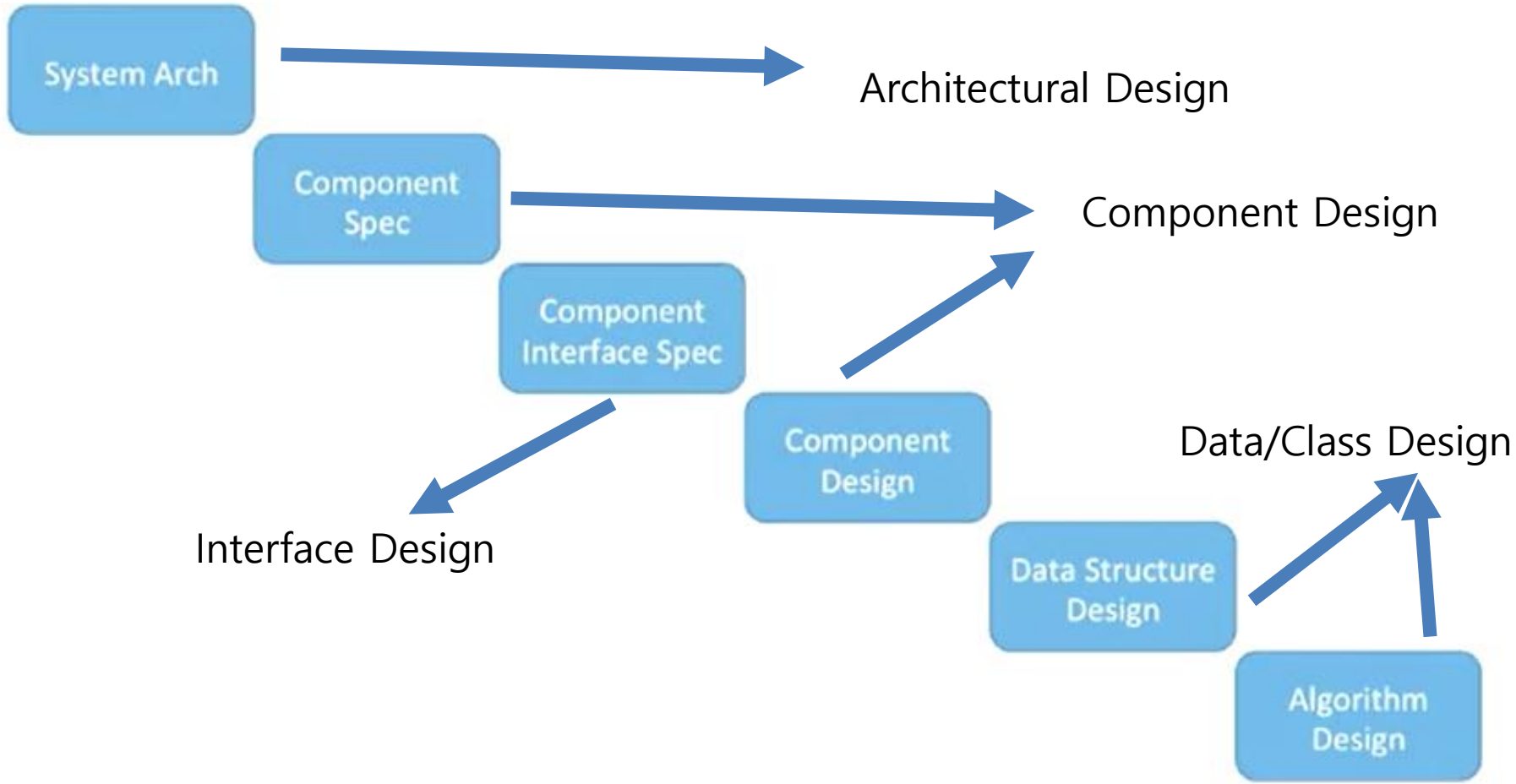
- The **architectural design** defines the relationship between major structural elements of the software, **the architectural styles and patterns**

- 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**.

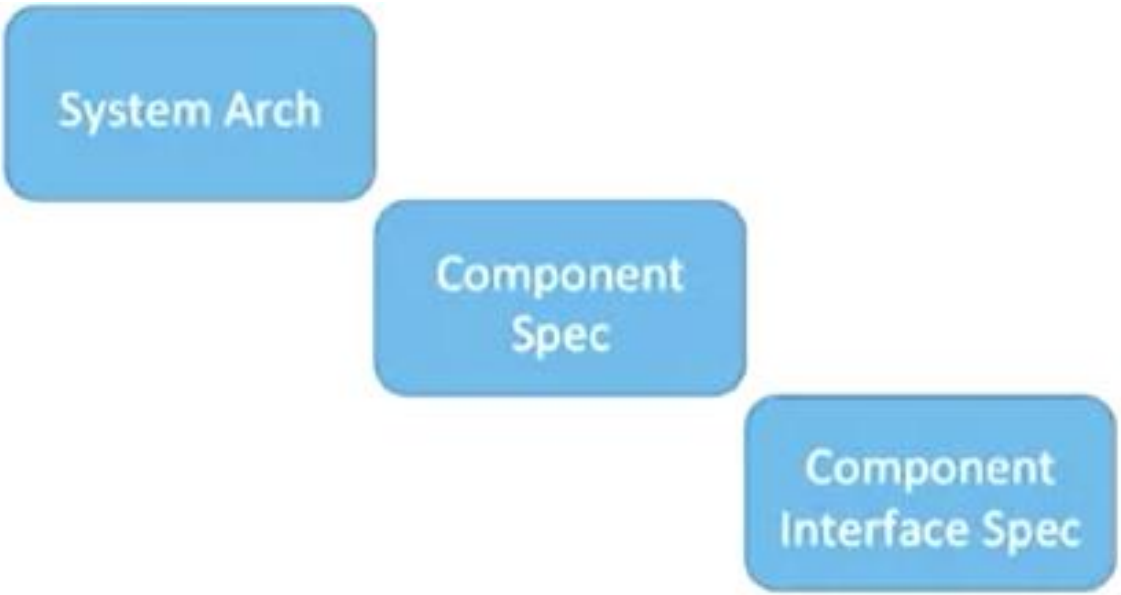
- The **data/class design** transforms class models into design class realizations and the **requisite data structures and algorithms** required to implement the software.

Stages of Design



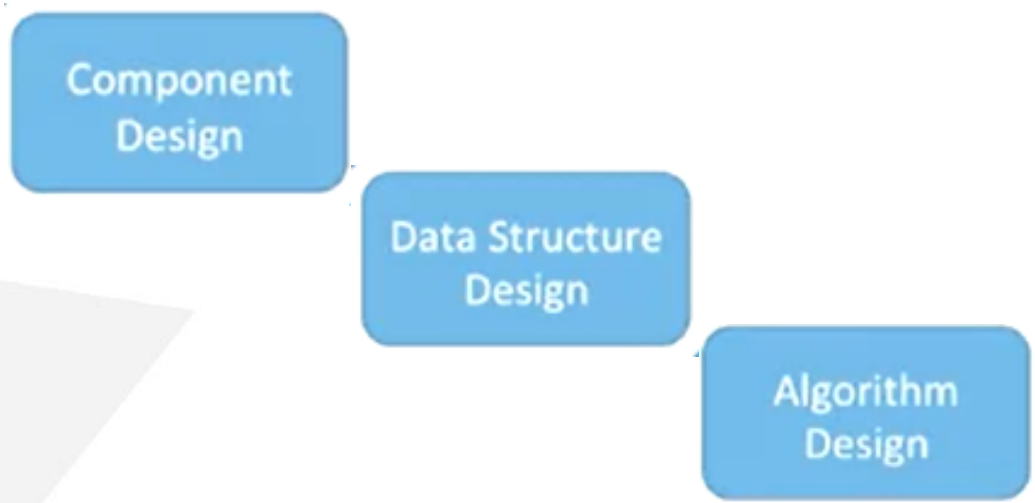
Stages of Design

- 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

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

10/4/2024

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

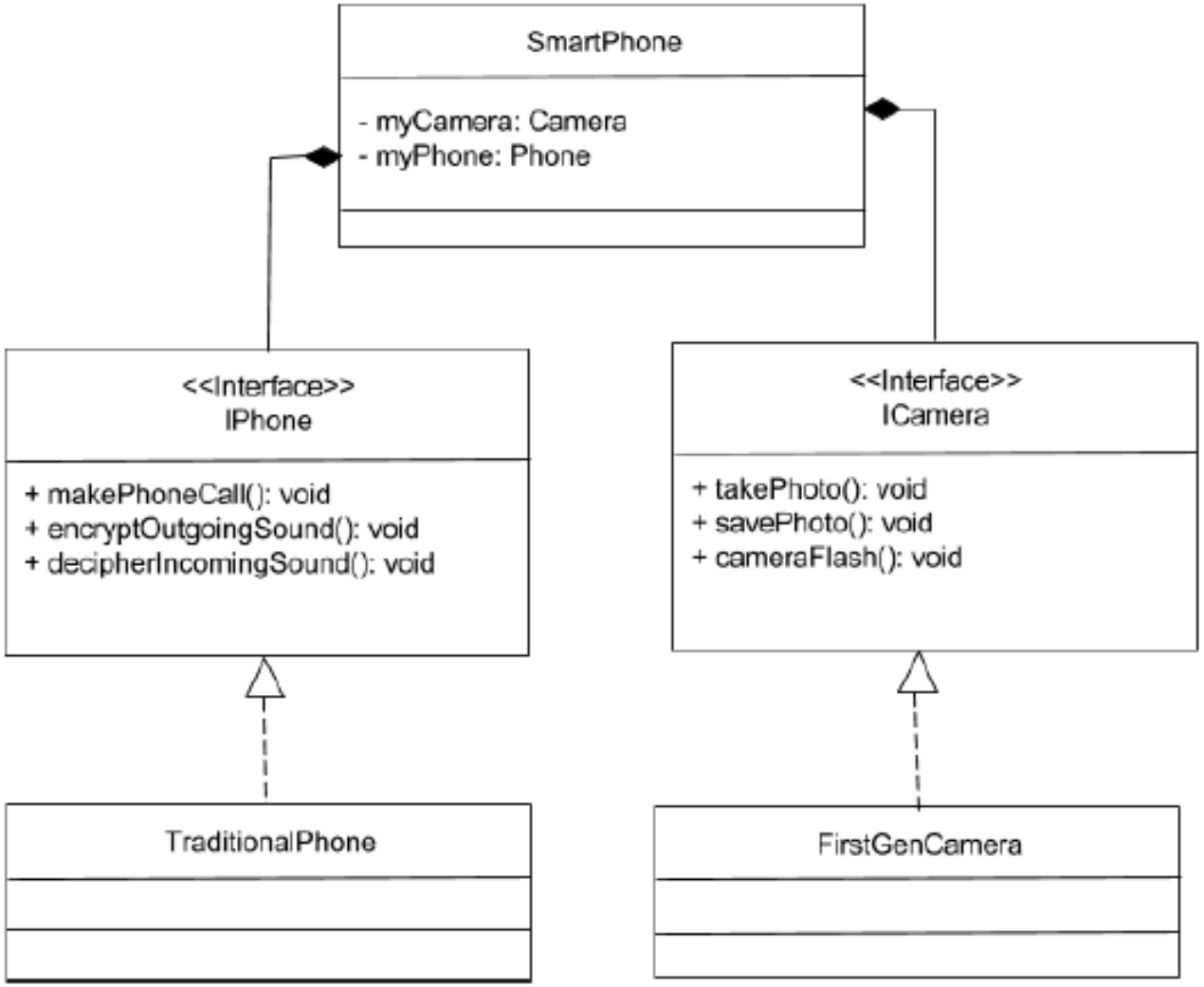
```
public class SmartPhone {
      private byte camera;
      private byte phone;

      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 deciphereIncomingSound();
}

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 cert ain 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 mor e 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

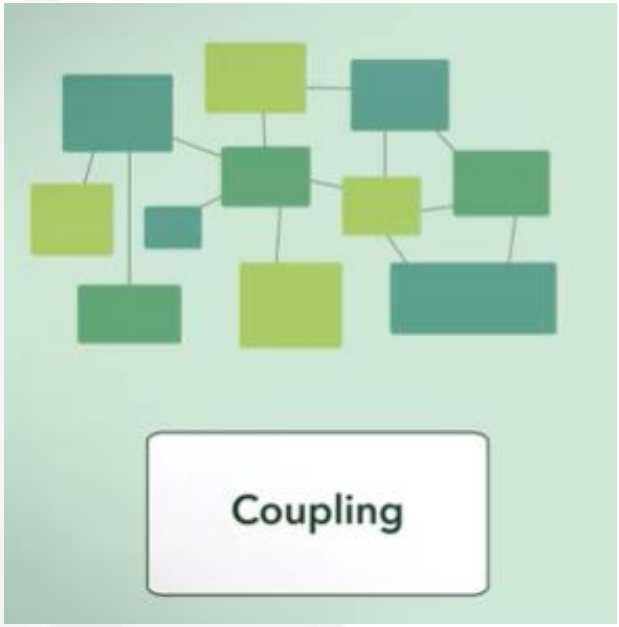
Coupling and Cohesion

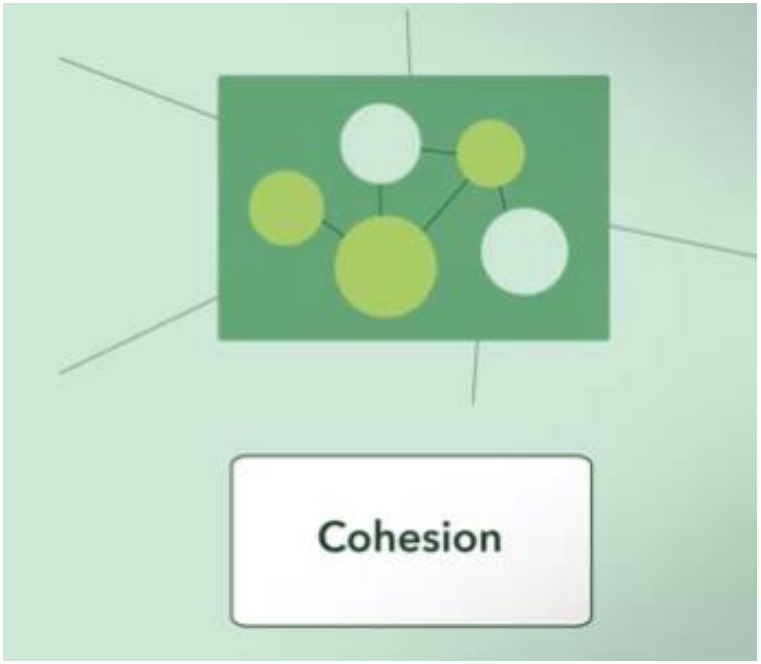
Cohesion is a measure of:

- 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

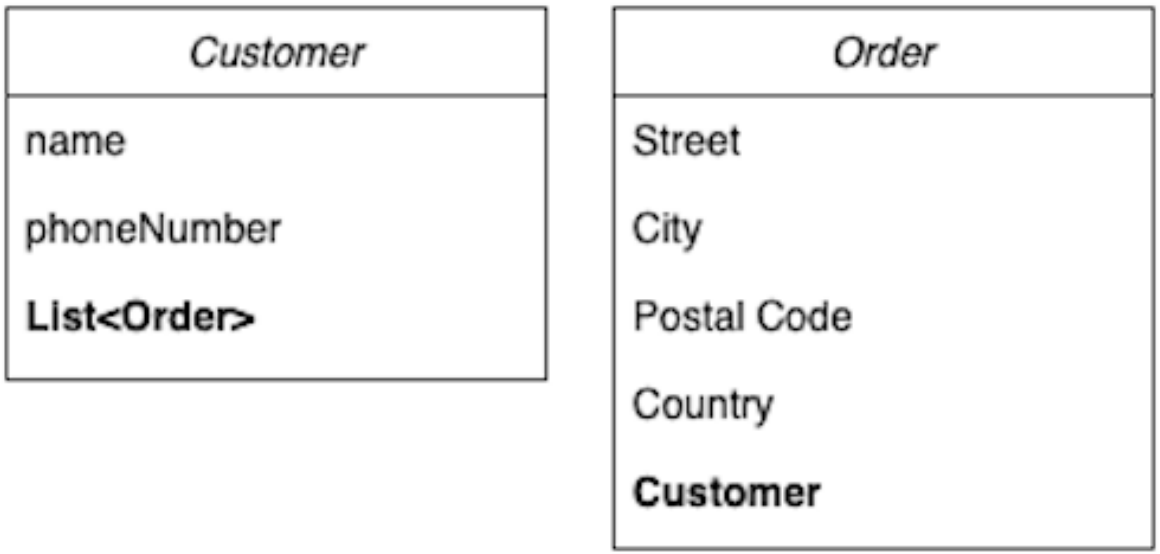
Goal of Good Modularity and also of Software Design is:

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

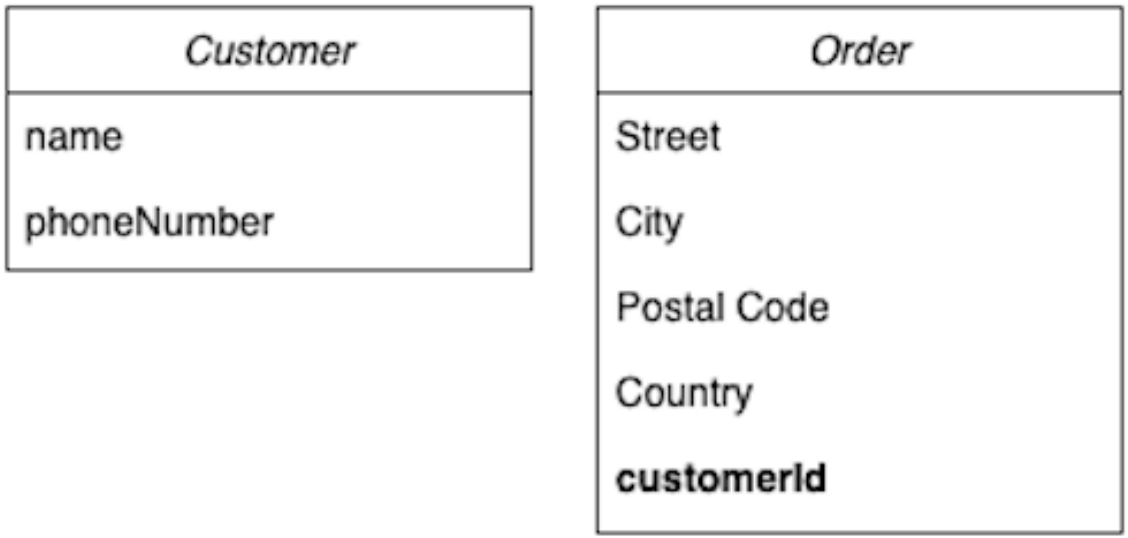
Two modules have high coupling (or tight coupling) if they are closely connected. For example, two concrete classes storing references to each other and calling each other's methods. As shown in the diagram below, *Customer* and *Order* are tightly coupled to each other. The *Customer* is storing the 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:

Low Cohesion

High Cohesion

EIU
TRƯỜNG ĐẠI HỌC
QUỐC TẾ
MIỀN ĐÔNG
EASTERN
INTERNATIONAL
UNIVERSITY

Software Engineering
Course's Code: CSE 305

EIU
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

What is Design Pattern

- Design Pattern **is** a description of the problem and the essence of its solution, so that the solution may be reused in different settings.
- Design Pattern **is not** a detailed specification
- Design Pattern can be described as a description of accumulated wisdom and experience, a well-tried solution to a common problem

What is Design Pattern

- Industry has some professionals v
  some best practices, there was a book published in 1994 by four authors:
  - ✓ These four authors are commonly known as "The Gang of Four" (GoF)
  - ✓ The design patterns provided by GoF can be applied in any object oriented language

Categories of GoF Design Pattern

23 design patterns are classified in these three categories and they are classic design patterns

- **Creational** Design Patterns are different way to create objects
- **Structural** Design Patterns are about relationships between these objects
- **Behavior** Design Patterns are about interaction or communication between these objects

Which design pattern should we use depends on software requirements

EIU

Benefits of using Design Pattern

Design Patterns can help us to **communicate** with other developers in more abstract level

- For example, you can tell your coworker, “Hey we can use this Façade Pattern to improve this code”
- You just need to name of the design pattern to communicate your idea
- You do not need to write a lots of code to express your idea

Benefits of using Design Pattern

It helps you to make a better designer

- Elements of reusable design: You can learn how to make reusable, extensible and maintainable software, no matter what kind of programming language you have used or what kind of application you have created
- This help you to learn any frameworks quickly, since same design patterns are used in various frameworks and libraries

EIU

Creational Design Pattern

In software design, creational design patterns are design patterns that deal with **object creation** mechanisms, trying to create objects in a manner suitable to the situation.

The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

Different design patterns included in this category are:

- **Abstract Factory**: Creates an instance of several families of classes
- **Builder**: Separates object construction from its representation
- **Factory Method**: Creates an instance of several derived classes
- **Object Pool**: Avoid expensive acquisition and release of resources by recycling objects that are no longer in use
- **Prototype**: A fully initialized instance to be copied or cloned
- **Singleton**: A class of which only a single instance can exist

EIU

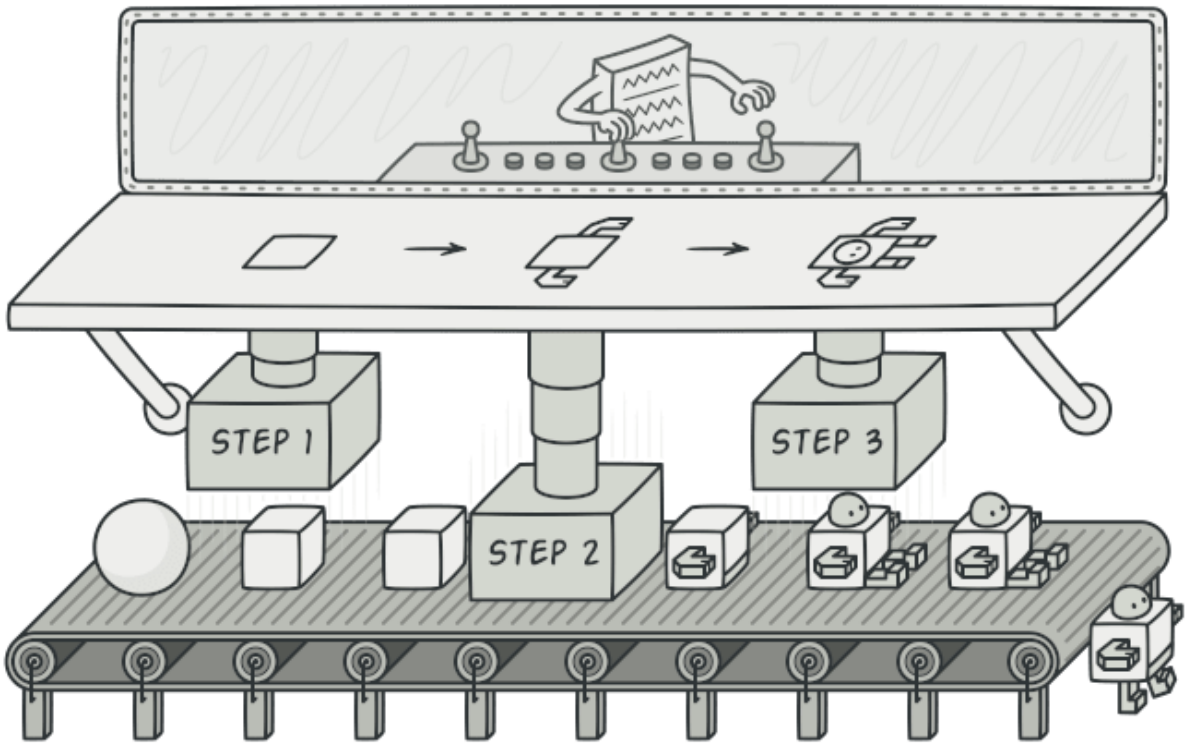
Builder Pattern

Builder

Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

EIU

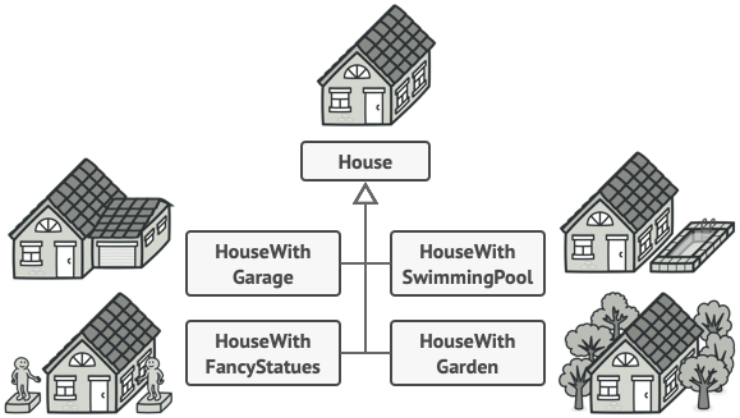
Builder Pattern



Builder Pattern

Problem

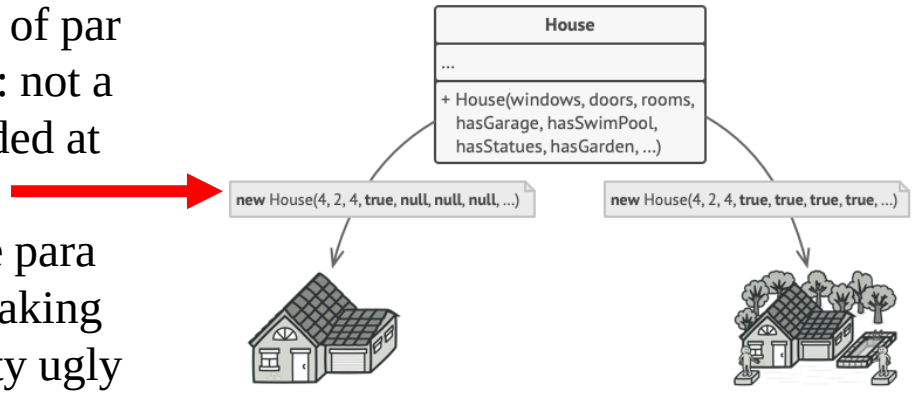
Imagine a complex object that requires laborious, step-by-step initialization of many fields and nested objects. Such initialization code is usually buried inside a monstrous constructor with lots of parameters. Or even worse: scattered all over the client code.



Builder Pattern

- The simplest solution is to extend the base House class and create a set of subclasses to cover all combinations of the parameters. But eventually you'll end up with a considerable number of subclasses. Any new parameter, such as the porch style, will require growing this hierarchy even more.
- There's another approach that doesn't involve breeding subclasses. You can create a giant constructor right in the base House class with all possible parameters that control the house object. While this approach indeed eliminates the need for subclasses, it creates another problem.

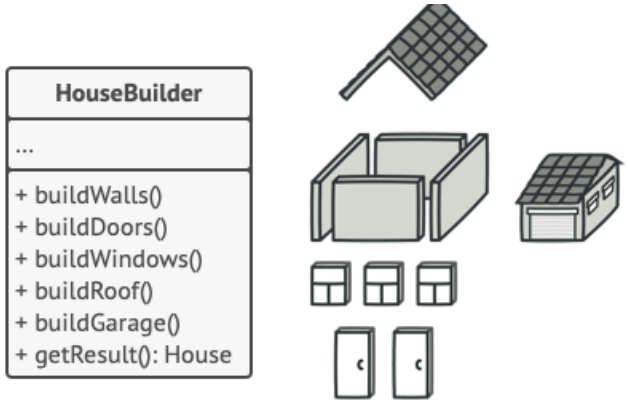
- The constructor with lots of par ameters has its downside: not a ll the parameters are needed at all times.
- In most cases most of the para meters will be unused, making the constructor calls pretty ugly .



Builder Pattern

Solution

The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called *builders*.



The pattern organizes object construction into a set of steps (buildWalls, build Door, etc.). To create an object, you execute a series of these steps on a builder object. The important part is that you don't need to call all of the steps. You can call only those steps that are necessary for producing a particular configuration of an object.

EIU

Builder Pattern

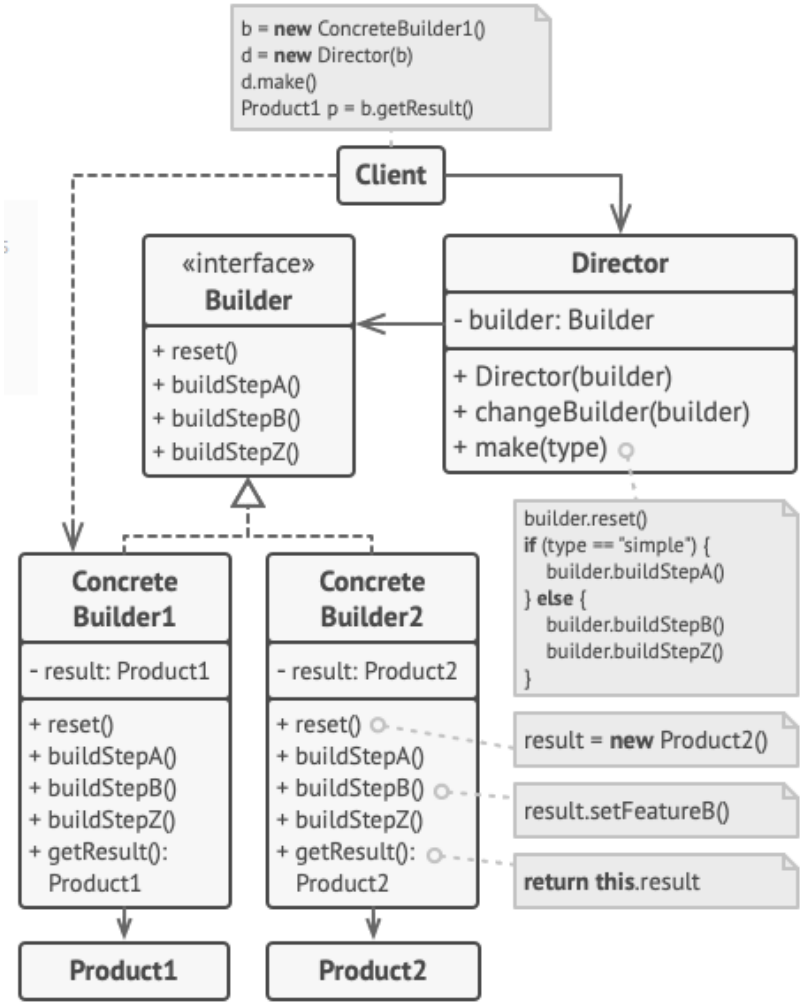
- Some of the construction steps might require **different implementation** when you need to build various representations of the product. For example, walls of a cabin may be built of wood, but the castle walls must be built with stone.
- In this case, you can create several different builder classes that implement the same set of building steps, but in a different manner. Then you can use these builders in the construction process (i.e., an ordered set of calls to the building steps) to produce different kinds of objects

Builder Pattern

- **The Builder** interface declares product construction steps that are common to all types of builders
- **Concrete Builders** provide different implementations of the construction steps. While adhering to the common interface, Concrete Builders may produce products that vary in their internal representations or structures.
- **Products** are resulting objects. Products constructed by different builders don’t have to belong to the same class hierarchy or interface.
- **The Director** class defines the order in which to call construction steps, so you can create and reuse specific configurations of products.
- **The Client** must associate one of the builder objects with the director.

EIU

Builder Pattern



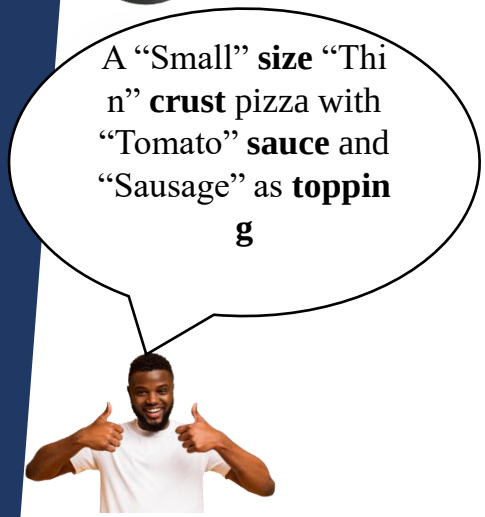
EIU

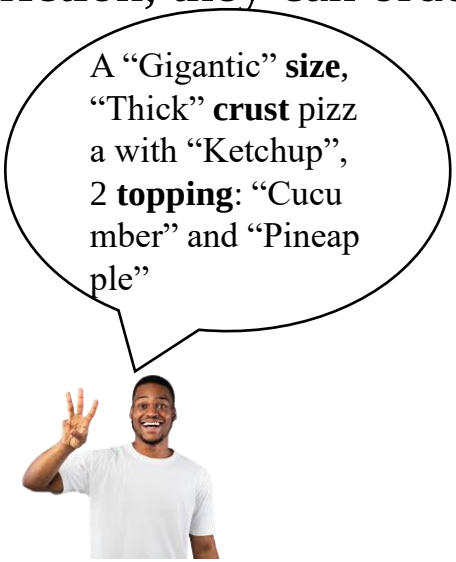
Builder Pattern

LET MAKE A PIZZA ORDERING SYSTEM

Your system let the user custom their pizza as they desire: from the **Size** of their Pizza, what type of **Sauce** they like, how thick is the **Crust,** what kind of **Topping** they want to put on (their pizza).

There is no restriction, they can order anything





EIU

Builder Pattern

To meet the requirement, you Pizza class should be s ome something like this

```
class Pizza {
private String size;
private String crust;
private String sauce;
private List<String> toppings;
@Override
public String toString() {
return "Pizza [size=" + size + ", crust=" + crust + ", sauce=" + sauce
+ ", toppings=" + toppings + "]";
}
}
```

EIU

Builder Pattern

In normal practice, if we want to make an immutable class, then we must pass all pieces of information as parameters to the constructor. It will look like this:

```
public Pizza(String size, String crust, String sauce,List<String> toppings)
{
 this.size = size;
 this.crust = crust;
 this.sauce = sauce;
 this.toppings = toppings;
}
```

New pizza will be created like:

```
 public class PizzaOrder {
 public static void main(String[] args) {
 Pizza order1 = new Pizza("Small","Thin","Tomato",Arrays.asList("Sausage"
 ));
 System.out.print(order1);
 // TODO Auto-generated method stub
 }
 }
```

only firstName and lastName are **mandatory**

Builder Pattern

Very good. Now what if only Size and topping are **mandatory**, the rest 2 fields are **optional**.

Problem !! We need more constructors. This problem is called the telescoping constructors problem.

```
public Pizza(String size, String crust, String sauce,List<String> toppings) {…}
public Pizza(String size,String crust, List<String> toppings) {…}
public Pizza(String size, List<String> toppings, String sauce) {…}
public Pizza(String size,List<String> toppings) {…}
```

We will need some more like the above

```
public static void main(String[] args) {
Pizza order1 = new Pizza("Small","Thin","Tomato",Arrays.asList("Sausage"));
Pizza order2 = new Pizza("Small","Thin",Arrays.asList("Sausage"));
Pizza order3 = new Pizza("Small",Arrays.asList("Sausage"));
System.out.print(order1);
}
```

EIU

Builder Pattern

Still can manage? Now let's introduce our **fifth optio nal attribute** i.e. Spice**.**

Now it is a problem !!!!

One way is to create **more constructors**, and another is to l se the immutability and introduce **setter methods**. You cho se any of both options, and you lose something, right?

Here, the builder pattern will help you consume additional attributes while retaining the immutability of the User class.

EIU

Builder Pattern

Builder: interface declares product construction steps that a re common to all types of builders

```
package Builder;
import Product.Pizza;


public interface PizzaBuilder {
void setSize(String size);
void setCrust(String crust);
void setSauce(String sauce);
void addTopping(String topping);
Pizza build();
}
```

EIU

Builder Pattern

Concrete Builders: provide different implementations of the construction steps

```
public class PizzaConcreteBuilder1 implements PizzaBuilder
{
private String size;
private String crust;
private String sauce;
private List <String> toppings;
@Override
public void setSize(String size) {
this.size = size;
}
@Override
public void setCrust(String crust) {
this.crust = crust;
}
@Override
public void setSauce(String sauce) {
this.sauce = sauce;
}
@Override
public void addTopping(String topping) {
if (this.toppings == null) {
this.toppings = new ArrayList<String>();
}
this.toppings.add(topping);
}
@Override
public Pizza build() {
return new Pizza(size,crust,sauce,toppings);
}
}
```

EIU

Builder Pattern

Final Product

```
package Client;

import Product.Pizza;
import ConcreteBuilder.PizzaConcre
teBuilder;

public class Client {
public static void main(String[] a
rgs) {
Pizza order1 = new PizzaConcreteBu
ilder()
.setSize("Big")
.setCrust("Thin")
.setSauce("Tomato")
.addTopping("Banana")
.build();
System.out.println(order1);
}
}
```

EIU

Builder Pattern

What about **Director ?**

```
class PizzaDirector {
private PizzaBuilder builder;
public PizzaDirector(PizzaBuilder builder) {
this.builder = builder;
}
public Pizza buildCustomPizza() {
builder.setSize("Large");
builder.setCrust("Thin");
builder.setSauce("Tomato");
builder.addTopping("Mushrooms");
builder.addTopping("Pepperoni");
return builder.build();
}
}
public class BuilderPatternWithDirectorDemo {
public static void main(String[] args) {
PizzaBuilder builder = new CustomPizzaBuilder();
PizzaDirector director = new PizzaDirector(builde
r);
Pizza pizza = director.buildCustomPizza();
System.out.println(pizza);
}
}
```

Builder Pattern

Readability:

- The code is more readable, and it's clear what each parameter represents.
- No need to remember the order of parameters; you only set what you need.

Flexibility:

- You can create a object with only the required parameters.
- You can set optional parameters in any order.

Avoid Telescoping Constructors:

- Avoid having multiple constructors with different parameter combinations (telescoping constructors).

Immutable Objects:

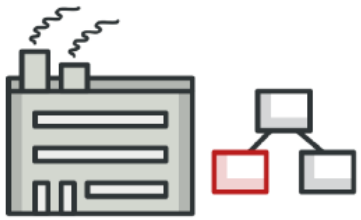
- The Person class can be made immutable by not providing setters for its fields.

Easy to Extend:

- Adding new optional parameters is straightforward; you just need to add a new method in the builder.

EIU

Factory Method



Factory Method

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

❖ This design pattern is also known as Virtual Constructor

EIU

Factory Method

☹ Problem

Imagine that you're creating a logistics management application. The first version of your app can only handle transportation by trucks, so the bulk of your code lives inside the `Truck` class.

After a while, your app becomes pretty popular. Each day you receive dozens of requests from sea transportation companies to incorporate sea logistics into the app.

Factory Method

Great news, right? But how about the code? At present, most of your code is coupled to the `Truck` class. Adding `Ships` into the app would require making changes to the entire codebase. Moreover, if later you decide to add another type of transportation to the app, you will probably need to make all of these changes again.

Adding a new class to the program isn't that simple if the rest of the code is already coupled to existing classes.

EIU

Factory Method

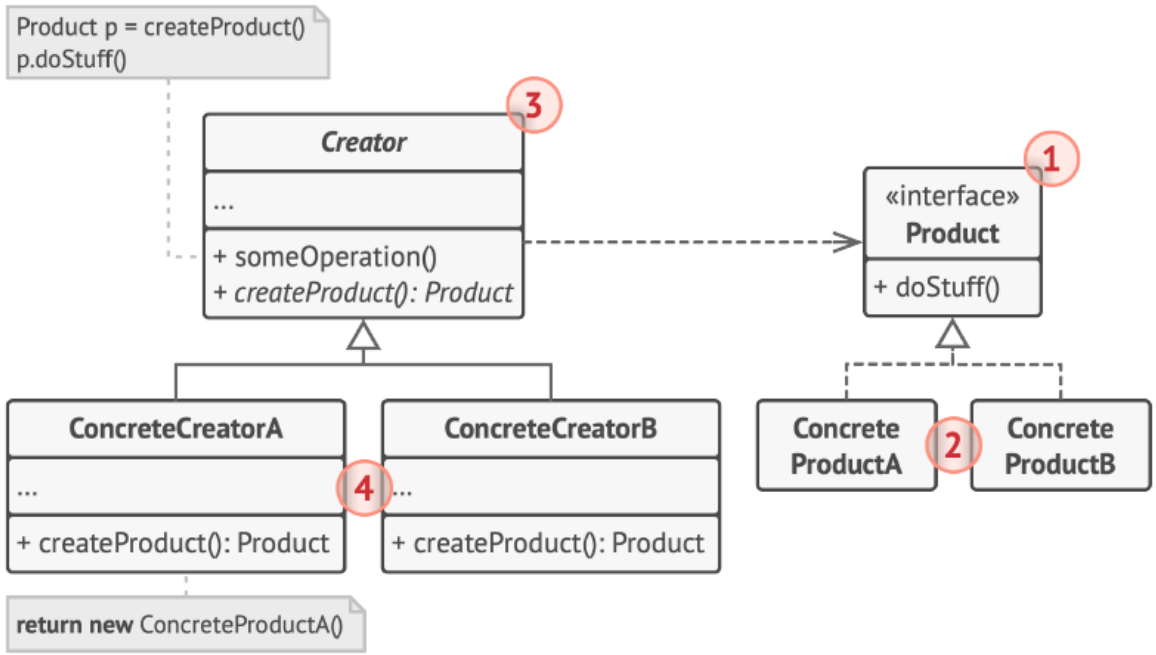
😊 Solution

The Factory Method pattern suggests that you replace direct object construction calls (using the `new` operator) with calls to a special *factory* method. Don't worry: the objects are still created via the `new` operator, but it's being called from within the factory method. Objects returned by a factory method are often referred to as *products*.

EIU

Factory Method

Structure



EIU

Factory Method

1. The **Product** declares the interface, which is common to all objects that can be produced by the creator and its subclasses.

2. **Concrete Products** are different implementations of the product interface.

3. The **Creator** class declares the factory method that returns new product objects. It's important that the return type of this method matches the product interface.

4. **Concrete Creators** override the base factory method so it returns a different type of product.

EIU

Factory Method

LET MAKE A PIZZA ORDERING SYSTEM

Your system let the user pick from the list of pre-defined type of Pizza but also let customer customize as their preference.

EIU

Factory Method

Again, you Pizza class should be some something like this

```
class Pizza {
private String size;
private String crust;
private String sauce;
private List<String> toppings;
public Pizza(String size, String crust, String sauce, List<S
tring> toppings) {
this.size = size;
this.crust = crust;
this.sauce = sauce;
this.toppings = toppings;
}
@Override
public String toString() {
return "Pizza [size=" + size + ", crust=" + crust + ", sauce
=" + sauce + ", toppings=" + toppings + "]";
}
}
```

Factory Method

Creator: let us first create the interface Pizza - an abstracts of the proces s of creating Pizza objects, his separation of concerns allows you to create d ifferent variations of Pizza objects by changing the factory implementation

```
interface PizzaFactory {
Pizza createPizza(String size, String crust, String sauce
, List<String> toppings);
}
```

EIU

Factory Method

Concrete Creator: allows you to create different variations of Pizza objects by changing the factory implementation

- Subclass 1 - **Standard Pizza:** crispy "crust", ketchup "sauce", bacon "Topping", customer may customize only size

```
class StandardPizzaFactory implements PizzaFactory {
@Override
public Pizza createPizza(String size, String crust, String sauce, List<S
tring> toppings) {
return new Pizza(size, crust, sauce, toppings);
}
}
```

- Subclass 2 - **PepperoniPizza:** pepperoni as "Topping", customer may customize the rest

```
class PepperoniPizza implements PizzaFactory {
@Override
public Pizza createPizza(String size, String crust, String sauce, List<Strin
g> toppings) {
toppings.add("Pepperoni");
return new Pizza(size, crust, sauce,toppings );
}
```

Factory Method

- **Concrete Creator:** allows you to create different variations of Piz za objects by changing the factory implementation
  - Subclass : **Pepperoni Pizza**

```
class PepperoniPizza implements PizzaFactory {
@Override
public Pizza createPizza(String size, String crust, String sauce, Li
st<String> toppings) {
toppings.add("Pepperoni");
return new Pizza(size, crust, sauce,toppings );
}
```

- All subclasses provide their own implementation of the createPizza () m ethod.

Factory Method Design Pattern

```
public class FactoryMethodDemo {
public static void main(String[] args) {
PizzaFactory factory = new StandardPizzaFactory();
List<String> toppings1 = new ArrayList<>(Arrays.asList("Mushrooms", "Pepperc
Pizza pizza1 = factory.createPizza("Large", "Thin", "Tomato", toppings1);
List<String> toppings2 = new ArrayList<>(Arrays.asList("Chicken"));


PizzaFactory factory2 = new PepperoniPizzaFactory();
Pizza pizza2 = factory2.createPizza("Medium", "Thick", "BBQ", “”);
System.out.println(pizza1);
System.out.println(pizza2);
}
}
```

Factory Method Design Pattern

- **Abstraction of Object Creation**: The Factory Method pattern abstracts the process of creating Pizza objects. This separation of concerns allows you to create different variations of Pizza objects by changing the factory implementation.

- **Consistent Interface**: The factory provides a consistent interface for creating Pizza objects, regardless of the specific pizza variations or implementations.

EIU

- Structural Design Pattern

- Structural patterns, describe how objects are connected to each other.
- Previously, we examined the major design principles like Association, Aggregation, Composition and Generalization, and how they are expressed in UML class diagrams
- There are many different ways you can structure objects depending on the relationship you'd like between them.
- Not only do structural patterns describe how different objects have relationships, but also how subclasses and classes interact through inheritance.
- Structural Patterns use these relationships, and describe how they should work to achieve a particular design goal.

EIU

- Structural Design Pattern

- The Structural Design Patterns are classified as
  - Adapter. Allows for two incompatible classes to work together by wrapping an interface around one of the existing classes.
  - Bridge. Decouples an abstraction so two classes can vary independently.
  - Composite. Takes a group of objects into a single object.
  - Decorator. Allows for an object's behavior to be extended dynamically at run time.
  - Facade. Provides a simple interface to a more complex underlying object.
  - Flyweight. Reduces the cost of complex object models.
  - Proxy. Provides a placeholder interface to an underlying object to control access, reduce cost, or reduce complexity.

EIU

- Structural Design Pattern

- If software system become larger, they naturally become more complex.
- This can be potentially confusing for the client classes in your system to use.
- System complexity is not always a sign of poor design, though.
- The scope of the problem you're trying to solve may be so large, it requires a complex solution.
- Client classes, however, would prefer a simpler interaction.
- The facade design pattern provides a single simplified interface for client classes to interact with the subsystem.

- What is Facade?

➢ If you've ever gone shopping at a retail store, eaten at a restaurant, or ordered anything online, you've had experience interacting with the real world facade design pattern.

➢ Imagine yourself walking down the street looking for a place to have dinner. What do you look for when you're at street? Naturally, you look for a sign on the face of a building that indicates that the contents within the structure can provide you with the dining service.

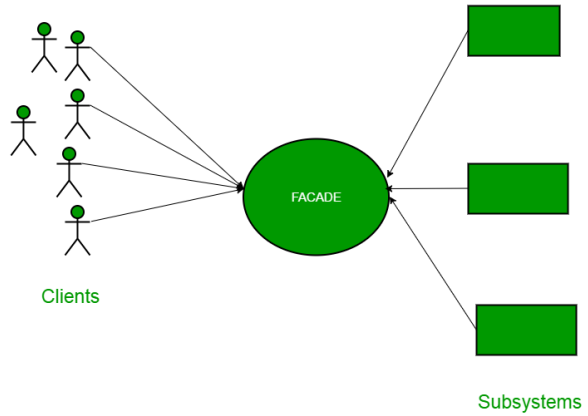
EIU

- What is Facade?

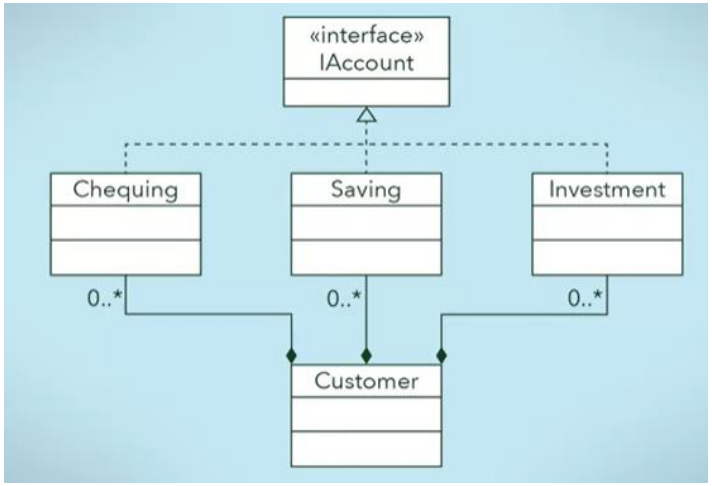
- Similarly, when you're shopping online, you look for some outward indication that the website you're on has a virtual storefront. These outward facing indicators are used to communicate to the public what types of services are available.
- When a waiter takes and places your order, or when an online sales platform sends your order to be fulfilled, they are acting as part of a facade by hiding away all the extra work that needs to be done
- You are able to purchase goods and services without having to know how the request is processed. The entire construction of the buildings, wait-staff, menus, websites and many other components are behind facades.
- Keep in mind that a facade does not actually add more functionality, a facade simply acts as a point of entry into your subsystem.

- What is Facade?

- In software, the facade design pattern does exactly what a waiter or salesper would do in real life.
- A facade is a wrapper class that encapsulate the subsystem in order to hide the subsystem's complexity.
- This wrapper class will allow a client class to interact with the subsystem through a façade
- Let's take a look at how client code would interact with the subsystem without a facade class for a simple banking system



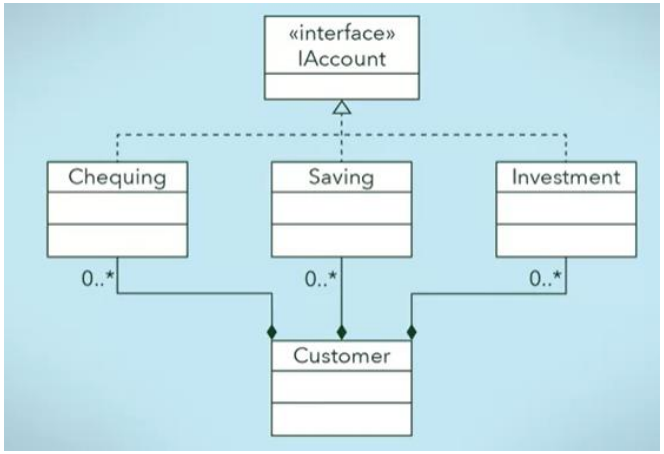
- Client – Subsystem Interaction without Facade

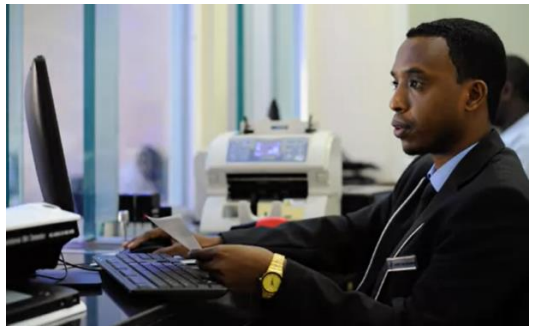


➢ Without a facade class, the customer class would contain instances of the checking, saving and investment classes.

➢ This means that the customer is responsible for properly instantiating each of these constituent classes and knows about all their different attributes and methods.

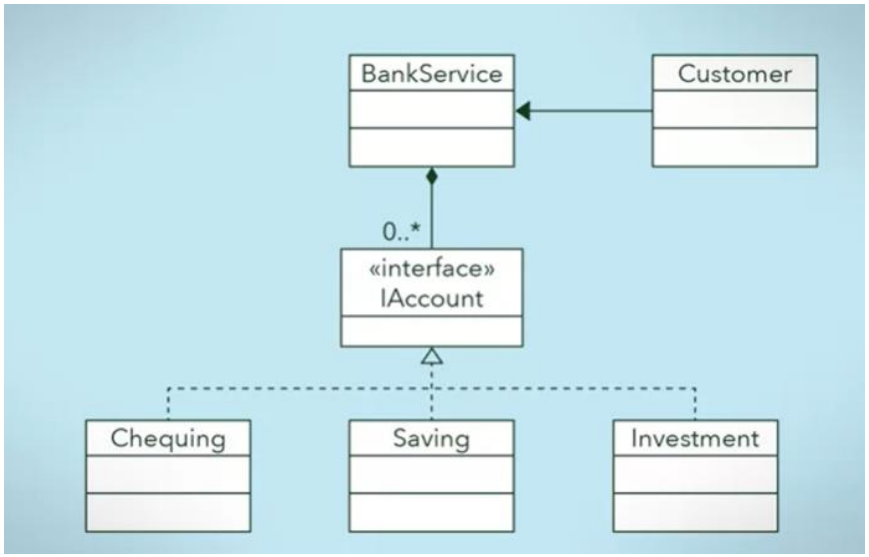
- Client – Subsystem Interaction without a Façade class





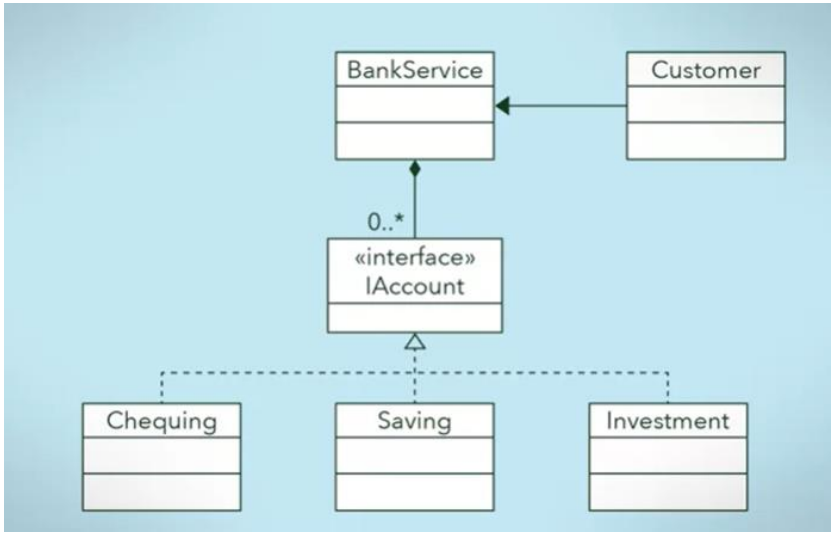
➢ This is like having to manage all of your own financial accounts in real life, which can be complex with lots of accounts, instead of letting a financial institution do it for you.

- Client – Subsystem Interaction using a Facade class



- Instead, we introduce the bank service class to act as a facade for the checking, saving, and investment classes.
- The customer class no longer needs to handle instantiation or deal with any of the other complexities of financial management.

- Client – Subsystem Interaction using a Façade class



- Since the three different accounts all implement the IAccount interface, the bank's service class is effectively wrapping the account interfacing classes, and presenting a simpler front to them for the customer client class to use..
- The facade design pattern is simple to apply in our bank accounts example.
- It combines interface implementation by one or more classes which then gets wrapped by the facade class.

- Question?

What are the two conditions required for you to use the facade design pattern?

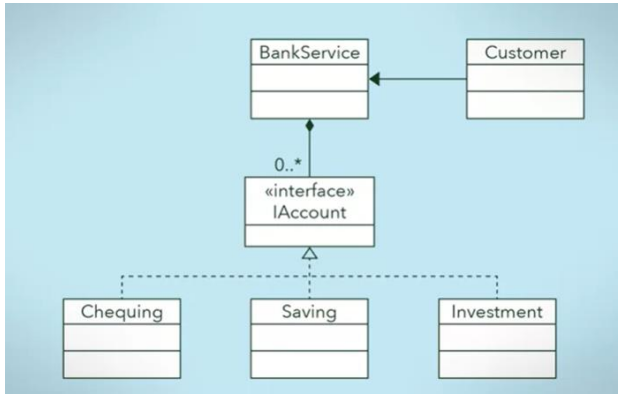
a) You need a class that will instantiate other classes within your system and provide these instances to a client class.

b) You need a class to translate messages between two existing subsystem s because one is expecting a specific interface to use but is provided with an interface that is incompatible.

c) You need a class to act as an interface between your subsystem and a client class.

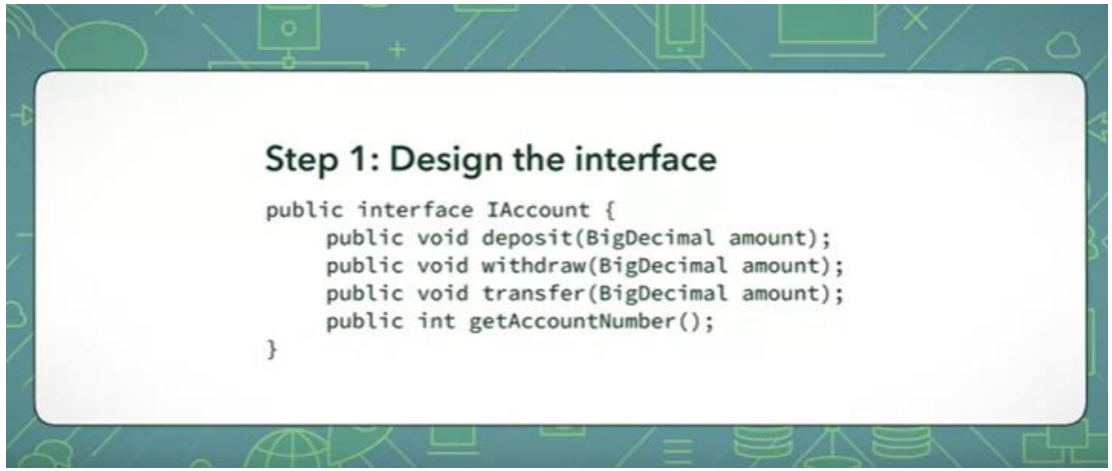
d) You need to simplify the interaction with your subsystem for client classes.

- Example of Facade Design Pattern

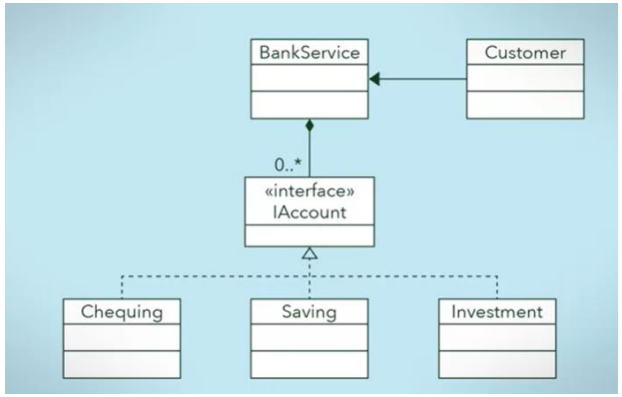


- This Java interface is the one that wil
  l

be implemented by the different account classes and will not be known to the customer class

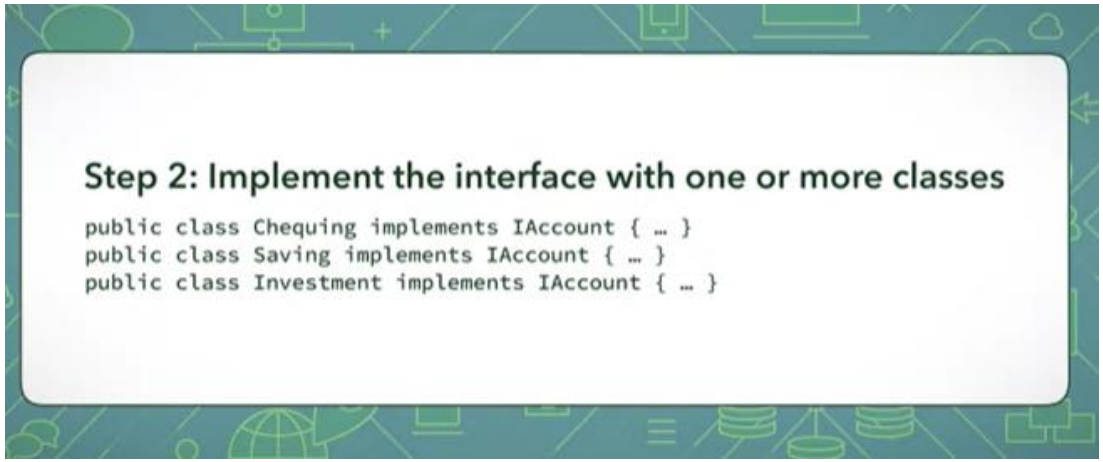


- Example of Facade Design Pattern

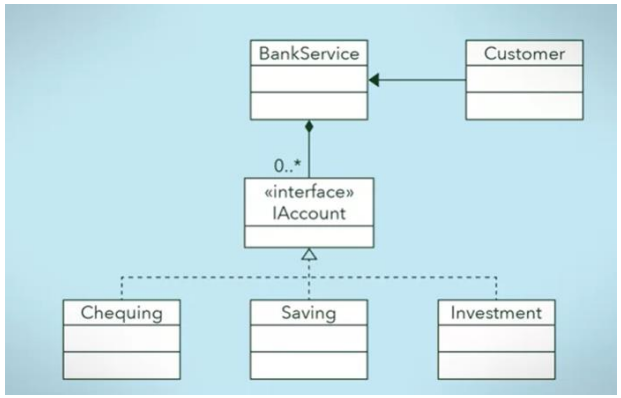


- Remember that interfaces allow us to create subtypes which means that
- checking, saving, and investment are subtypes of i-Account and are

expected to behave like an account type



- Example of Facade Design Pattern



Step 3: Create the facade class and wrap the classes that implement the interface

- In this example, we are only implementing and hiding one interface, but in practice, a facade class can be used to wrap all the interfaces and classes for a subsystem.
- It is our decision as to what we want to wrap
- In this example, BankService class is the facade

- Example of Facade Design Pattern

```
public class BankService {
      private Hashtable<int, IAccount> bankAccounts;
      public BankService() {
            this.bankAccounts = new Hashtable<int, IAccount>;
      }
      public int createNewAccount(String type, BigDecimal initAmount) {
            IAccount newAccount = null;
            switch (type) {
                  case "chequing":
                        newAccount = new Chequing(initAmount);
                        break;
                  case "saving":
                        newAccount = new Saving(initAmount);
                        break;
                  case "investment":
                        newAccount = new Investment(initAmount);
                        break;
                  default:
                        System.out.println("Invalid account type");
                        break;
            }
          if (newAccount != null) {
                this.bankAccounts.put(newAccount.getAccountNumber(), newAccount);
                return newAccount.getAccountNumber();
          }
          return -1;
    }
    public void transferMoney(int to, int from, BigDecimal amount) {
          IAccount toAccount = this.bankAccounts.get(to);
          IAccount fromAccount = this.bankAccounts.get(from);
          fromAccount.transfer(toAccount, amount);
    }
}
```

- Example of Facade Design Pattern

- Notice that its public methods are simple to use and show no hint of the underlying interface and implementing classes.
- Another thing to note is that we set the access modifiers for each account to be private.
- Since the entire point of the facade design pattern is to hide complexity, we use the information hiding design principle to prevent all client classes from seeing the account objects and how these accounts behave.

```
public class BankService {
    private Hashtable<int, IAccount> bankAccounts;
    public BankService() {
        this.bankAccounts = new Hashtable<int, IAccount>;
    }
```

- Example of Facade Design Pattern

Step 4: Use the facade class to access the subsystem

```
public class Customer {

    public static void main(String args[]) {
        BankService myBankService = new BankService();

        int mySaving = myBankService.createNewAccount("saving",
            new BigDecimal(500.00));

        int myInvestment = myBankService.createNewAccount(
            "investment", new BigDecimal(1000.00));

        myBankService.transferMoney(mySaving, myInvestment, new
            BigDecimal(300.00));
    }

}
```

- Example of Facade Design Pattern

➢ Client classes can access the functionalities of the different accounts through the methods of the BankService class.

➢ The BankService class will tell the client what type of actions it will allow the client to call upon, and then will delegate that action to the appropriate account objects.

➢ Now that we have our facade in place, our client class can access its accounts through the BankService.

➢ customer class does not need to worry about creating and managing its ow accounts.

➢ The customer simply needs to know about the BankService and the set of behaviors the BankService is capable of performing.

➢ We have effectively hidden the complexity of account management from the customer using the BankService facade class.

- Summary: Facade Design Pattern

The facade design pattern:

- Is a means to hide the complexity of a subsystem by encapsulating it behind a unifying wrapper called a facade class.
- Removes the need for client classes to manage a subsystem on their own, resulting in less coupling between the subsystem and the client classes.
- Handles instantiation and redirection of tasks to the appropriate class within the subsystem.
- Provides client classes with a simplified interface for the subsystem.
- Acts simply as a point of entry to a subsytem and does not add more functionality to the subsystem.

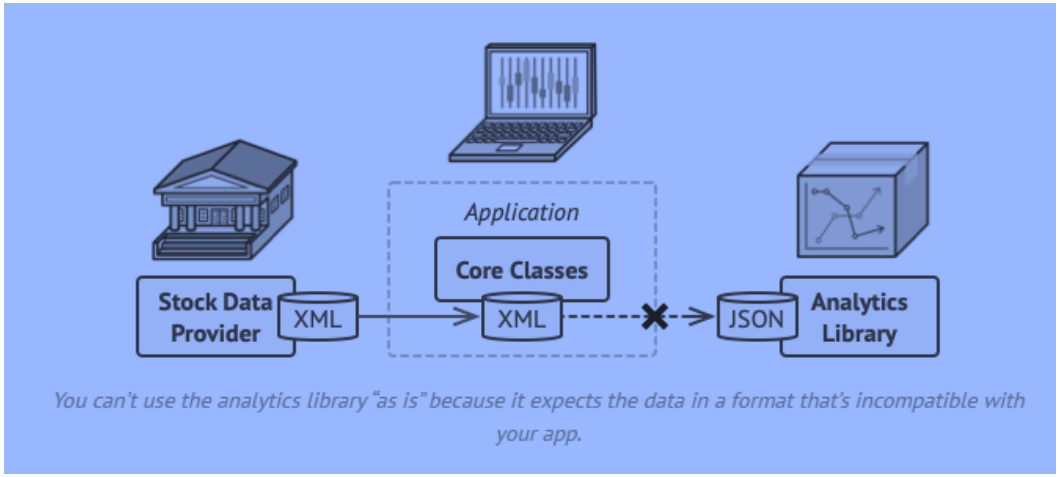
- Adapter Pattern

- Physically, an adapter is a device that is used to connect pieces of equipment that cannot be connected directly.
- Software systems may also face similar issues: not all systems have compatible software interfaces.
- In other words, the output of one system may not conform to the expected input of another system.
- This frequently happens when a pre-existing system needs to incorporate third-party libraries or needs to connect to other systems.
- The adapter design pattern facilitates communication between two existing systems by providing a compatible interface.
- It is a structural design pattern.

- Adapter Pattern

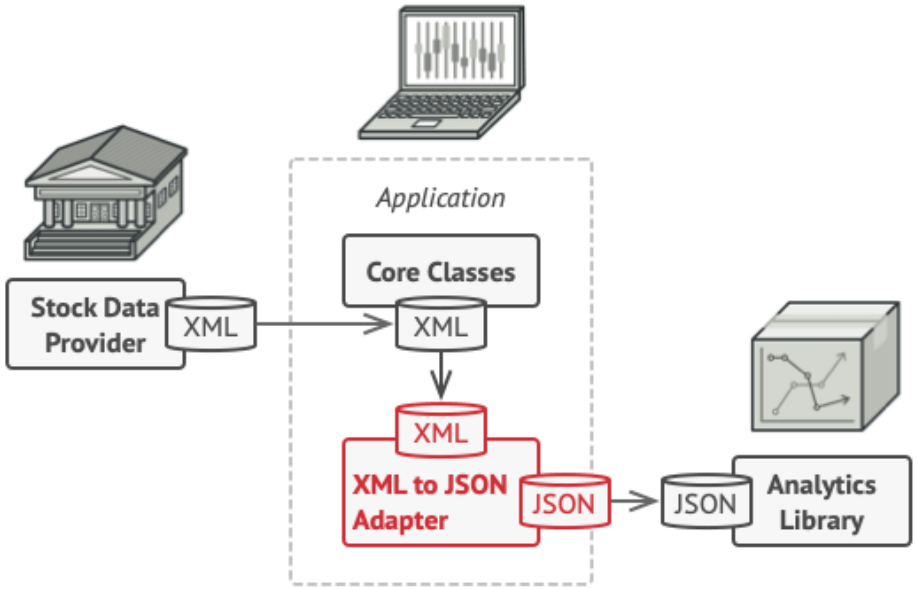
Imagine that you're creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.

At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in JSON format.

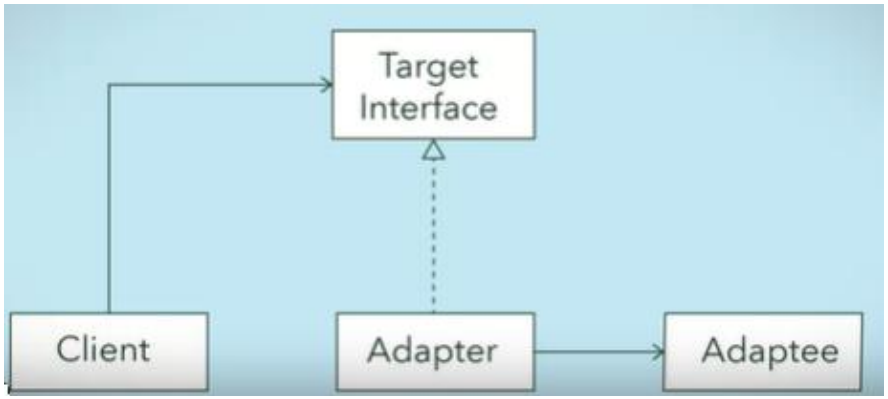


You can't use the analytics library "as is" because it expects the data in a format that's incompatible with your app.

- Adapter Pattern



- Adapter Pattern



The adapter design pattern consists of several parts.

- A client class. This class is the part of your system that wants to use a third-party library or external systems.
- An adaptee class. This class is the third-party library or external system that is to be used.
- An adapter class. This class sits between the client and the adaptee.
- The adapter conforms to what the client is expecting to see, by implementing a target interface. The adapter also translates the client request into a message that the adaptee will understand, and returns the translated request to the adaptee. The adapter is a kind of wrapper class.
- A target interface. This is used by the client to send a request to the adapter

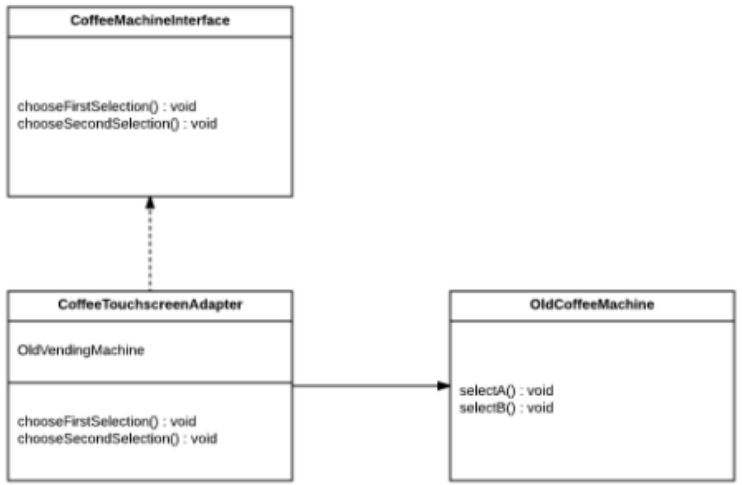
- Adapter Pattern

Implementation of an adapter design pattern can also be broken down into ste
s

1. Design the target interface
2. Implement the target interface with the adapter class
3. Send the request from the client to the adapter using the target interface

- Example:
  Adapter Pattern

You are working in an office with an old coffee machine that dispenses two different coffee flavours. However, the new boss wants to add a new coffee machine with a touchscreen that can also connect to the old coffee mac hine. Add an adapter so that the new touchscreen will to work with the old co ffee machine. Use the following UML class diagram for a guide:



- Adapter Pattern

1. Design the target interface

CoffeeMachineInterface.java

```
public interface CoffeeMachineInterface {
    public void chooseFirstSelection();
    public void chooseSecondSelection();
}
```

- Adapter Pattern

```
OldCoffeeMachine.java

public class OldCoffeeMachine {

        public void selectA() {
                System.out.println(“A - Selected”);
        }
        Public void selectB() {
                System.out.println(“B - Selected”);
        }
}
```

- Adapter Pattern

2. Implement the target interface with the adapter class

CoffeeTouchscreenAdapter.java

```
public class CoffeeTouchscreenAdapter implements CoffeeMachin
eInterface {

    OldCofffeeMachine theMachine;

    public CoffeeTouchscreenAdapter(OldCoffeeMachine newMachi
ne) {
        theMachine = newMachine;
    }
    public void chooseFirstSelection() {
        theMachine.selectA();
    }
    public void chooseSecondSelection() {
        theMachine.selectB();}}
```

- Adapter Pattern

3. . Send the request from the client to the adapter using the target interface

```
public class NewCoffeeMachine {

public static void main(String[] args) {
OldCoffeeMachine ocf=new OldCoffeeMachine();
CoffeeTouchScreenAdapter adapter=new CoffeeTouchScreenAdapter(ocf)
;
adapter.chooseFirstSelection();
adapter.chooseSecondSelection();

// TODO Auto-generated method stub

}

}
```

- Behavioral Design Pattern

- Behavioral Design Pattern focus on how objects distribute work.
- They describe how each object does a single cohesive function.
- Behavioral patterns also focus on how independent objects work towards a common goal.
- Think of a behavioral pattern like a race car pit crew at a track.
- In the pit crew, the roles of the members describe how the team is able to achieve victory, which is their common goal.

- Behavioral Design Pattern

- Each member has a specific responsibility, their role in the race.
- Some members change the tires, others un mount, and mount the wheel nuts, others refuel the car, but all must work together to win.
- Like a game plan, a Behavioral Pattern lays out the overall goal and the purpose for each of the objects.

- Behavioral Design Pattern

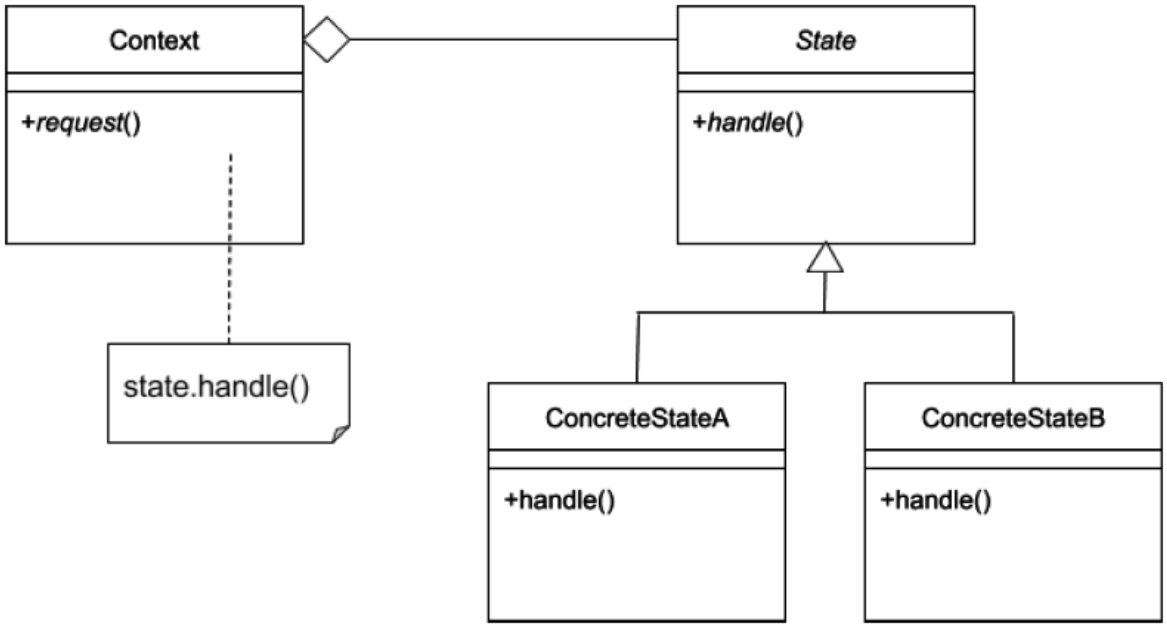
- The Behavioural Design Patterns are classified as
  - Command. Creates objects which encapsulate actions and parameters.
  - Iterator. Accesses the elements of an object sequentially without exposing its underlying representation.
  - Mediator. Allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
  - Observer. Is a publish/subscribe pattern which allows a number of observer objects to see an event.
  - State. Allows an object to alter its behavior when its internal state changes.
  - Strategy. Allows one of a family of algorithms to be selected on-the-fly at run-time.
  - Template Method. Defines the skeleton of an algorithm as an abstract class, allowing its sub-classes to provide concrete behavior.
  - Visitor. Separates an algorithm from an object structure by moving the hierarchy of methods into one object.

- State Pattern

Objects in your code are aware of their current state. They can choose an appropriate behaviour based on their current state. When their current state changes, this behaviour can be altered. This is the **state design pattern**.

This pattern should be primarily used when you need to change the behaviour of an object based upon changes to its internal state or the state it is in at run-time. This pattern can also be used to simplify methods with long conditionals that depend on the object's state.

- State Pattern



- State Pattern

The state design pattern might be best explained through an example. A vending machine has several states, and specific actions based on those states. For example, imagine you want to purchase a chocolate bar from the vending machine, that costs one dollar.

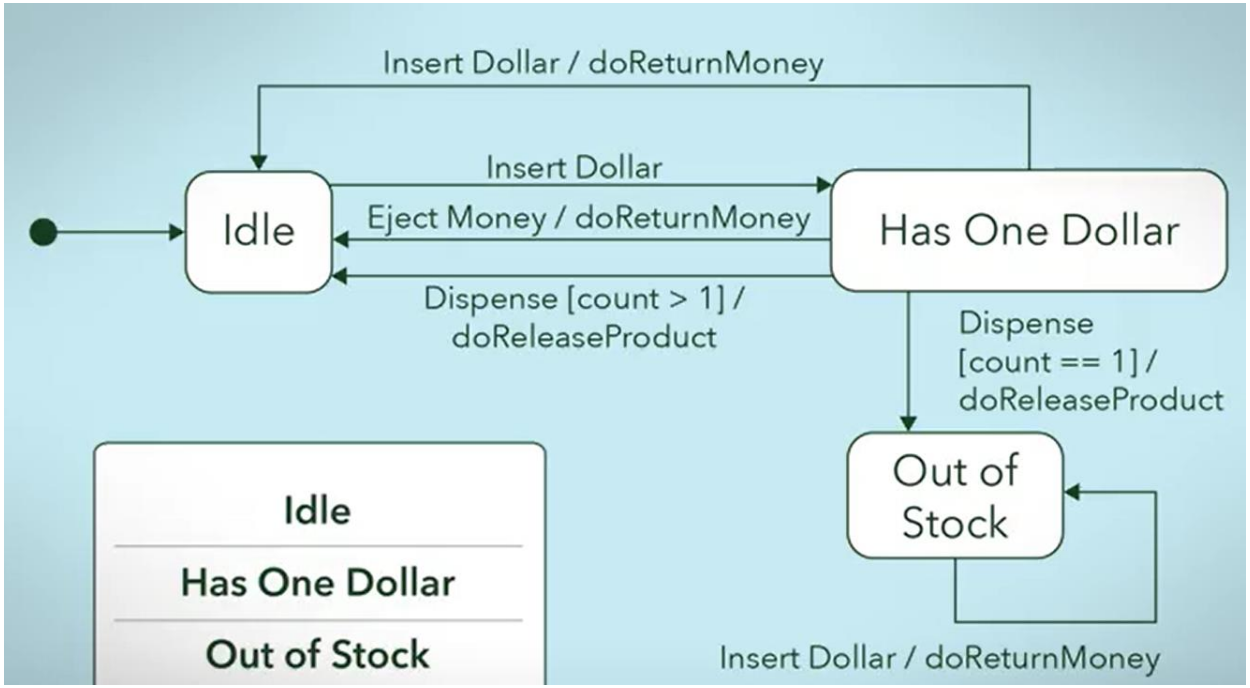
If you insert the dollar into the machine, several things might happen. You could make your selection, the machine would dispense a chocolate bar, and you would collect the bar and leave. You could decide you no longer want the chocolate bar, press the eject money button, and the machine would return the dollar. You could make your selection, the machine could be out of chocolate bars, and could notify you of this.

- State Pattern

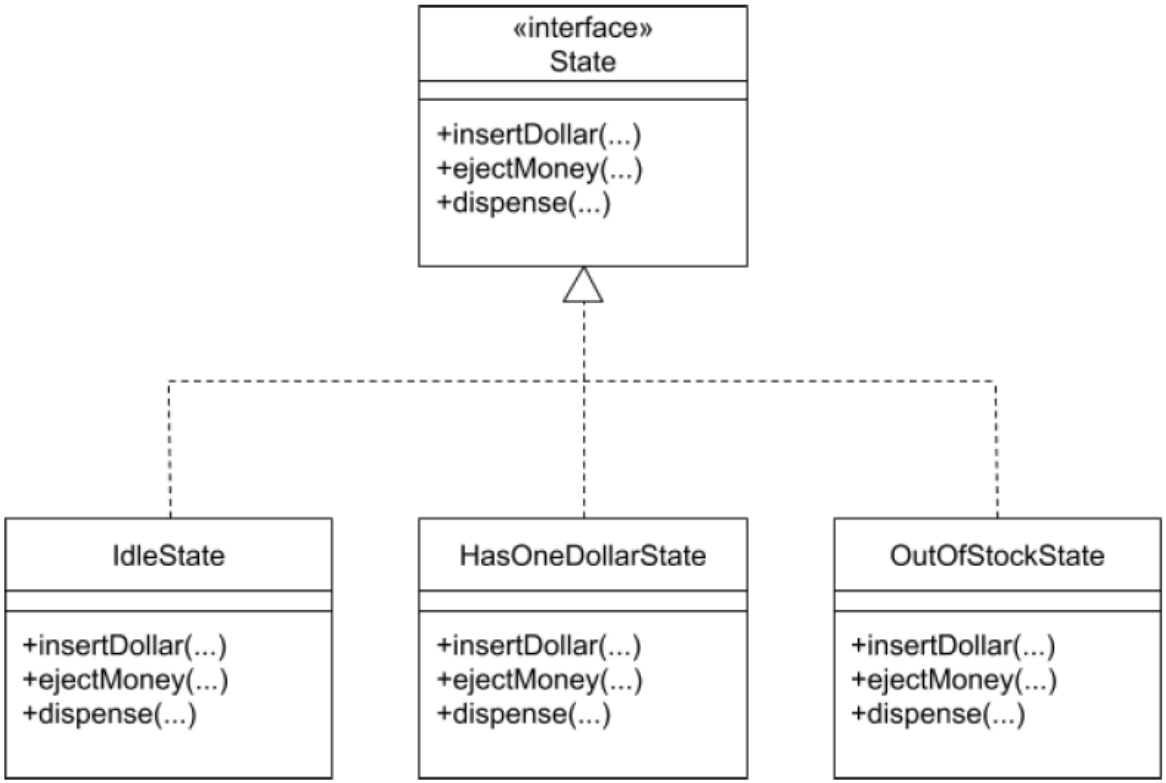
There are three different states that the vending machine can be in within this scenario. Before approached, the machine is in an idle state. When the dollar is inserted, the state of the machine changes, and could either dispense a product or return the money upon receiving an eject money request. The third state the machine could be in would be if the machine is out of stock.

Each of these states could be represented as state objects in the code.

- State Pattern



- State Pattern



- State Pattern

```
public interface State {
      public void insertDollar( VendngMachine
vendingMachine );
      public void ejectMoney( VendingMachine
vendingMachine );
      public void dispense( VendingMachine
vendingMachine );
}
```

- State Pattern

```
public class IdleState implements State {

	public void insertDollar( VendingMachine
vendingMachine ) {
		System.out.println( "dollar inserted" );

		vendingMachine.setState(
			vendingMachine.getHasOneDollarState()
		);
	}

	public void ejectMoney( VendingMachine
vendingMachine ) {
		System.out.println( "no money to return" );
	}
```

- State Pattern

```
        public void dispense( VendingMachine
 vendingMachine ) {

                System.out.println( "payment required" );
        }
}
```

- State Pattern

```
public class HasOneDollarState implements State {

    public void insertDollar( VendingMachine vendingMachine ) {
        System.out.println( "already have one dollar" );

        vendingMachine.doReturnMoney();
        vendingMachine.setState(
            vendingMachine.getIdleState()
        );
    }

    public void ejectMoney( VendingMachine vendingMachine ) {
        System.out.println( "returning money" );

        vendingMachine.doReturnMoney();
        vendingMachine.setState(
            vendingMachine.getIdleState()
        );
    }
```

- State Pattern

```
public class VendingMachine {

    private State idleState;
    private State hasOneDollarState;
    private State outOfStockState;

    private State currentState;
    private int count;

    public VendingMachine( int count ) {
        // make the needed states
        idleState = new IdleState();
        hasOneDollarState = new HasOneDollarState();
        outOfStockState = new OutOfStockState();

        if (count > 0) {
            currentState = idleState;
            this.count = count;
        } else {
            currentState = outOfStockState;
            this.count = 0;
        }
    }
```

EIU

- State Pattern

```
public void insertDollar() {
    currentState.insertDollar( this );
}

public void ejectMoney() {
    currentState.ejectMoney( this );
}

public void dispense() {
    currentState.dispense( this );
}
```

EIU

- Observer Pattern

- Imagine that you have a favorite blog. Every day you visit it multiple times per day to check for new blog posts.
- After a while, you get fed up with this routine. There must be a better way you think to yourself.
- The first solution that crosses your mind is to write a script to check the blog for new posts every fraction of a second.
- But upon further consideration, you realize most sites would not appreciate this barrage of requests, and would block your IP address.
- To avoid this, you instead write a script to check the blog for new posts once per hour.

- Observer Pattern

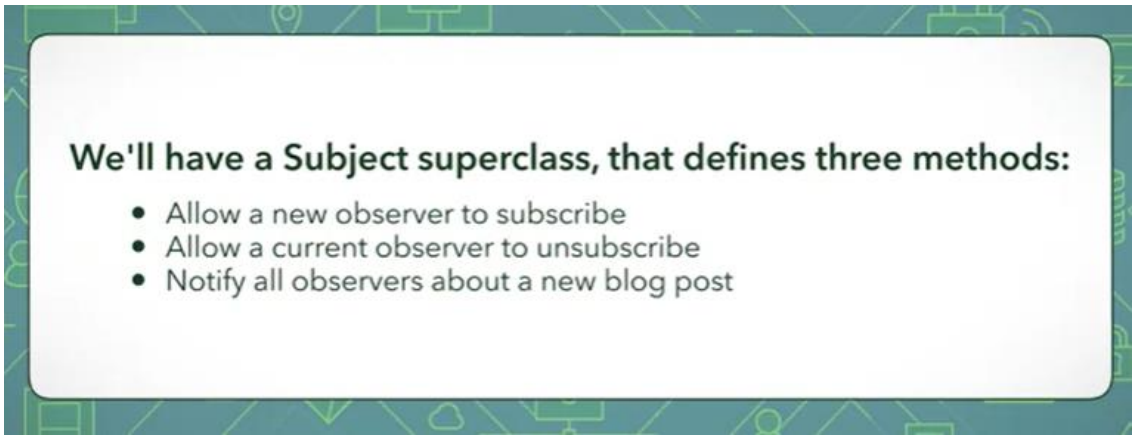
- But to your disappointment, this means that you're now missing out on blog posts that provide live changes, such as real time posts about your favorite TV show.
- A better solution to this problem is for the blog to notify you every time a new post is added.
- You would subscribe to the blog, every time a new post was published, the blog would notify each subscriber, including you.
- This way, you would be notified of the new content when it is created, rather than having to pull for this information at some interval

- Observer Pattern

- The observer design pattern is a pattern where a subject keeps a list of observers.
- Observers rely on the subject to inform them of changes to the state of the subject.
- In an observer design pattern, there is generally a Subject superclass, which would have an attribute to keep track of all the observers.
- There is also an Observer interface with a method so that an observer can be notified of state changes to the subject.
- The Subject superclass may also have subclasses that implement the Observe interface.
- These elements create the relationship between the subject and observer.

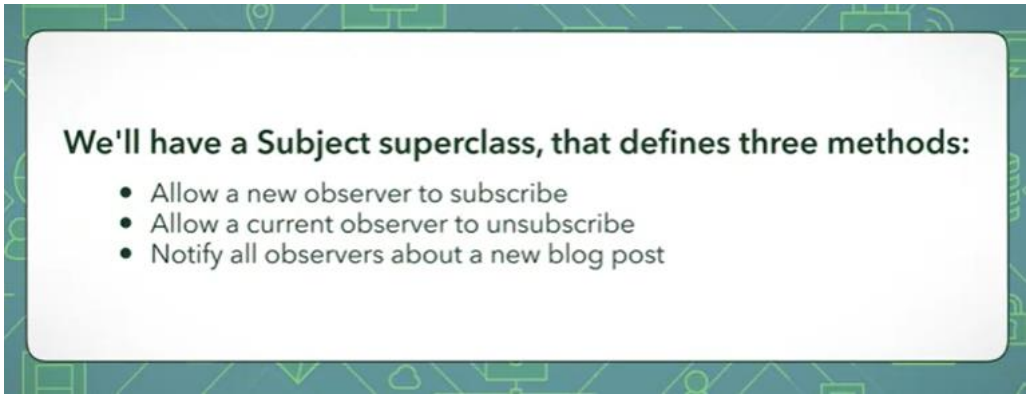
- Observer Pattern

- The subject in our example, the blog, would keep a list of observers.
- In this example, we might think of these observers as subscribers to the blog
- The observers rely on the blog to inform them of any changes to the state of the blog such as, when a new blog post is added.
- how we could apply this idea to solve this problem related to the blog?
- First:



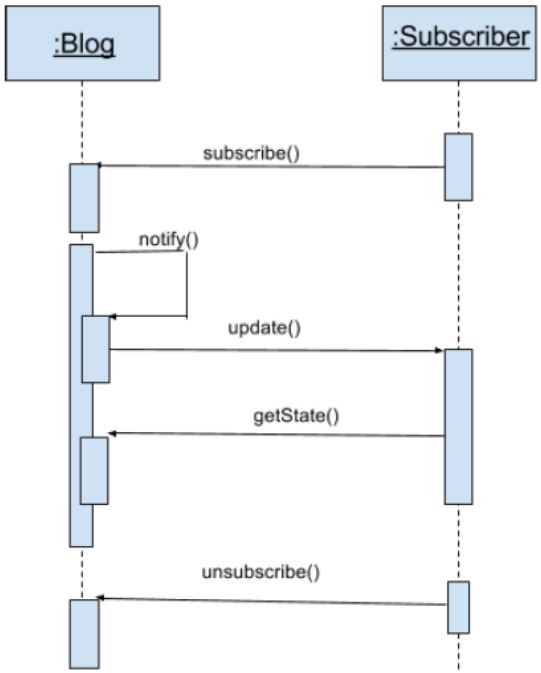
- Observer Pattern

- This superclass would also have an attribute to keep track of all the observer and will make an observer interface with methods that an observer can be notified to update itself
- Next, the blog Class will be a subclass of the Subject superclass, and the Subscriber Class will implement the observer interface.
- Observer design elements are essential to forming a Subject and Observer relationship. For example, for the blog and subscriber



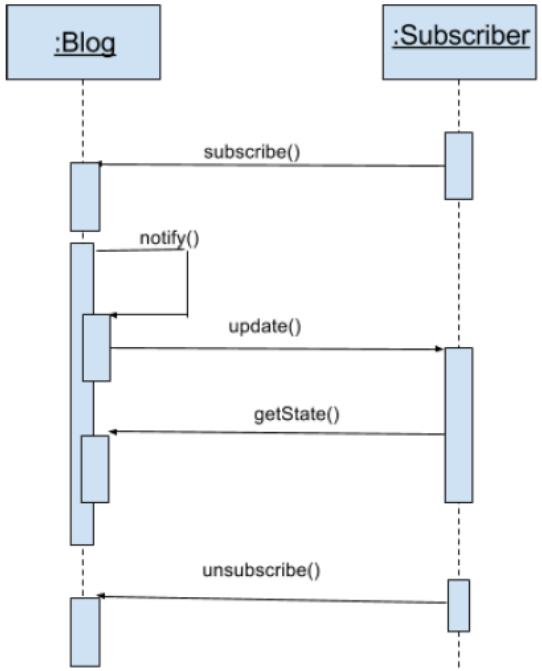
- Observer Pattern

- Using Sequence Diagram, we can describe this kind of relationship
- A sequence diagram for observe patterns will have two major roles: the subject (the blog) and the observer (a subscriber).
- In order to form the subject and observer relationship, a subscriber must subscribe to the blog.
- The blog then needs to be able to notify subscribers of a change.
- The notify function keeps subscribers consistent, and is only called when a change has been made to the blog.



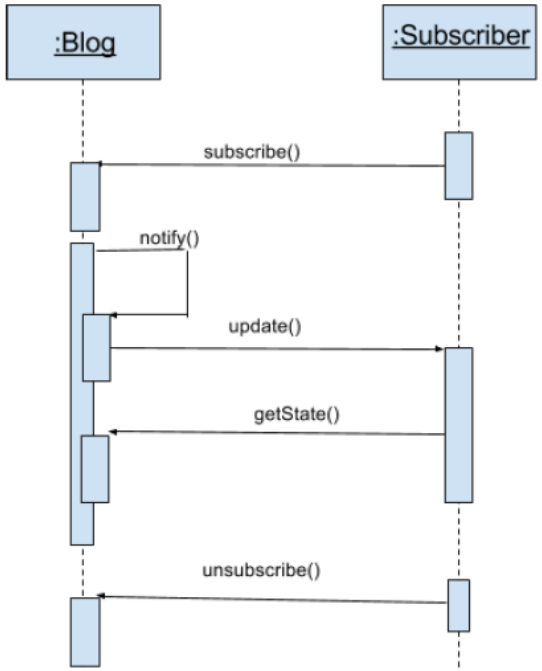
- Observer Pattern

- If a change is made, the blog will make an update call to update subscribers.
- Subscribers can get the state of the blog through a getState call.
- It is up to the blog to ensure its subscribers get the latest information.
- To unsubscribe from the blog, sub scribers could use the last call in the sequence diagram.
- unsubscribe() originates from the subscriber and lets the blog know the subscriber would like to be removed from the list of observers.



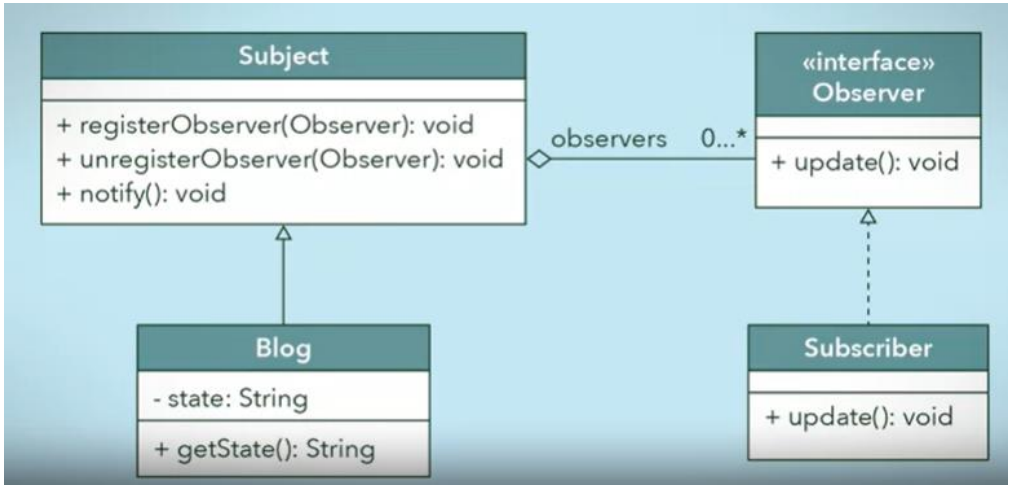
- Observer Pattern

- If a change is made, the blog will make an update call to update subscribers.
- Subscribers can get the state of the blog through a getState call.
- It is up to the blog to ensure its subscribers get the latest information.
- To unsubscribe from the blog, sub scribers could use the last call in the sequence diagram.
- unsubscribe() originates from the subscriber and lets the blog know the subscriber would like to be removed from the list of observers.



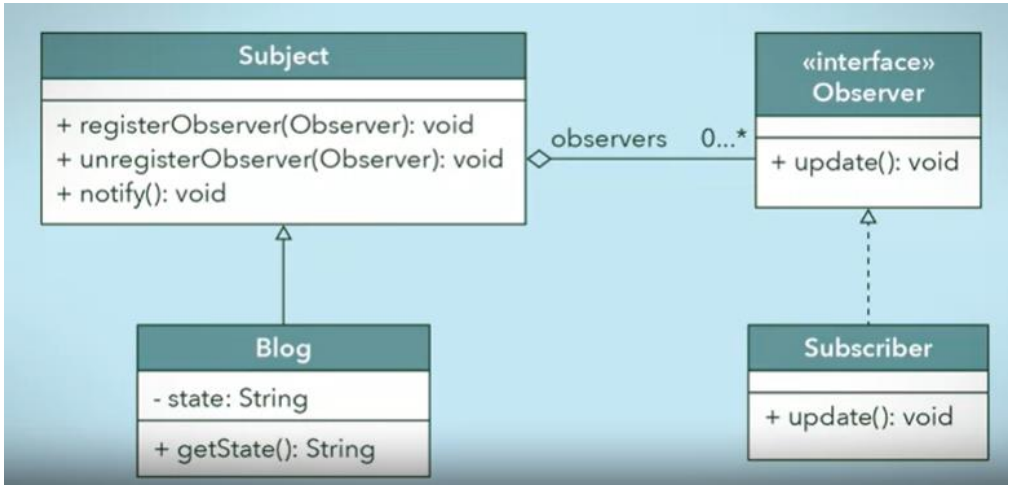
- Observer Pattern

- Let's take a look, in the UML diagram of this example of Observer Pattern
- The Subject superclass has three methods: register observer, unregister observer, and notify. These are essential for a subject to relate to its observers.
- A subject may have zero or more observers registered at any given time.



- Observer Pattern

- The Blog subclass would inherit these methods.
- The Observer interface only has the update method. An observer must have some way to update itself.
- The Subscriber class implements the Observer interface, providing the body of an update method so a subscriber can get what changed in the blog.



- Observer Pattern: Java Code

➢ Let's see the java code for subject superclass.

```
public class Subject {
    private ArrayList<Observer> observers = new ArrayList<Observer>();

    public void registerObserver(Observer observer) {
        observers.add(observer);
    }
    public void unregisterObserver(Observer observer) {
        observers.remove(observer);
    }
    public void notify() {
        for (Observer o : observers) {
            o.update();
        }
    }
}
```

- Observer Pattern: Java Code

- Next the blog class is ta subclass of the subject.
- which will inherit the register observer, unregister observer and notify methods and have the other responsibilities of managing a blog and posting messages.

```
public class Blog extends Subject {

    private String state;

    public String getState() {
        return state;
    }

    // blog responsibilities
    …
}
```

- Observer Pattern: Java Code

- The Observer interface makes sure all observer objects behave the same way.
- There is only a single method to implement, update(), which is called by the subject.
- The subject makes sure when a change happens, all its observers are notified to update themselves.

```
public interface Observer {
        public void update();
}
```

- Observer Pattern: Java Code

➢ The subject makes sure when a change happens, all its observers are notified to update themselves. In this example, there is a class Subscriber the implements the Observer interface.

```
class Subscriber implements Observer {

      public void update() {
            // get the blog change
            …
      }
}
```

- Observer Pattern: Java Code

- The observer design pattern can save you a lot of time when you're implementing your system.
- If you know that you have many objects that rely on the state of one, the value of the observer pattern becomes more pronounced.
- Instead of managing all the observer objects individually, you can have your subject manage them and make sure the observers are updating themselves as needed.
- There are many different ways and situations you can apply the observer design pattern.
- As a behavioral pattern, it makes it easy to distribute and handle notifications of changes across systems, in a manageable and controlled way.