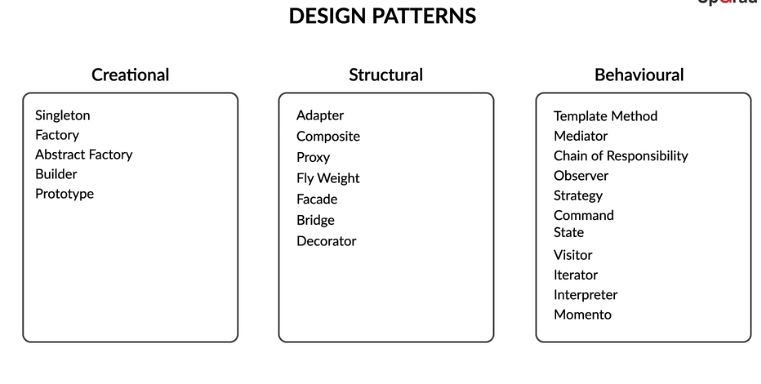
**Introduction to Design Patterns**

**Introduction:**

Welcome to the session on “Design Patterns”. Although programmers had been using Design Patterns long before even something like “Design Patterns” was formulated, they are one of the most essential building blocks of any software system. While creating the complex and multi-dimensional system, they help in visualising many recurring software development problems and help us in utilizing existing, battle-tested software design solution to our advantage.



**In this session:**

In the coming session, you will learn about a very commonly used and important principle in software development known as Design Patterns. You will understand what design patterns are, their usage and advantages. You will also learn how design patterns are categorised into three types: Creational, Behavioural and Structural design patterns. Through various demonstrative examples, you will learn what these types are and how they are further classified into various sub-types.

**Goals of the session:**

The session aims to give you a basic overview of design patterns and their application in software development.  The aim of the session will be achieved by first introducing design patterns and then analyzing its use and three types: Creational, Behavioural and Structural. There are a total of 23 design patterns, however, **as** **part of this session, we will be covering only the following as they are the most frequently used ones.**

1. **Creational design patterns:**
   1. Factory
   2. Builder
   3. Singleton
2. **Structural design patterns:**
   1. Adapter
   2. Proxy
   3. Facade
   4. Decorator
3. **Behavioural design patterns:**
   1. Iterator
   2. Observer
   3. Strategy
   4. Command

## People you will hear from in this session:-

**Teaching faculty**

[Anurag Varma](https://in.linkedin.com/in/anurag-varma-5bb5b421)  
Engineering Product Manager, UpGrad

Anurag is an Engineering Product Manager at UpGrad. Being a Tech Enthusiast and an Entrepreneur, Anurag has Co-founded Propeluss, led the Engineering team at Framebench (acquired by Freshworks) and the Frontend team at UpGrad. Anurag is a BITS Pilani Alumni.

**Industry Expert**  
[Varun Sehgal](https://www.linkedin.com/in/sehgalvarun)  
Director (Program Management), Zomato  
Varun has been working in the technology space for the last 10+ years. He has worked his way through various organisation setups — from IT services giants to small and medium-sized start-ups, from MNCs to Indian IT setups, and from very process-oriented to very execution-driven cultures. In the last few years, he has been part of India’s consumer internet growth story and has been privileged to watch it unfold from very close quarters. Varun holds a degree in Computer Science Engineering from PEC, Chandigarh and an MBA from IIT Madras, Chennai.

# Design patterns - Usability & Advantages

In scenarios which involve solving a complex, multi-dimensional problem, rather than building everything from scratch, it is often useful to use an already existing, well-tested template, which we can customize to suit our own needs. This is where design patterns come into play.

In the following let’s learn what design patterns are, their advantages and usability.

Speakers words: Imagine you are given a task of creating a website.now creating a web site is a huge task which involves in lot of things, creating interface creating databases, querying database…

You have 2 options

1. Either create everything from scratch like which would involve designing & planning the s layer, the n/w layer, the interface, interface creating databases, querying database…

If one would do write the entire code by himself, one would certainly become rigorous and master in program, but this code would certainly be prone to lot of errors, and it takes humongous amount of time to create a website.

1. The simpler and smarter way to solve this problem is to use already existing and time tested s/w solns to solve problems, there are many s/w libraries that enables us to build various components of a website easily and quickly, these libraries consists of 100 of methods & ways which provide ready to use and instant solns for some common problems that encounter while creating a website. These existing s/w libraries would have proper documentation and would be peer reviewed so chances of errors are very less.

DPs are similar to these libraries where in they are pieces of autonomous code chunks designed to reused in familiar situations, so that complex and layered problems would be solved in an easy and doablemanner.

We can customize the templates to suit our needs, this is where DP comes into picture.

A DP provides a general , well tested & reusable soln to a common design problem.

DP help in formalizing problems into concrete terms so that an appropriate already existing s/w solns can be applied on them.

Q> **Commonly used libraries and languages**

Can you share an example of a library or frameworks that you have used in the past for building software projects?

**Suggested Answer**

There are many commonly used tools, languages and software such as NodeJS, ReactJS, HTML, CSS which are extremely well-documented and robust enough to enable web development. You will learn and fiddle with some of these tools in the coming times.

Eg: We need to construct a house, this is a humongous task which involves diff. tasks- tiling, carpentry, sanitary….

We create a successful blueprint to solve them, or we can use already existing templates/blue prints of house architecture/planning. No need to plan from grasslevel planning. As 100s of houses are being constructed daily, this is a recurring problem and DP would come to our rescue.

ADVANTAGES;

Provide a verified soln to solve the issue related to s/w development

They make overall system easier to understand and maintain.

There are some good practices which must be followed while programming in any language. These practices not only ensure correctness of code, but also increase code readability, robustness, and maintainability. One of the important principles of Software Design is that we prefer Composition over Inheritance. Let us first revisit what Composition is and later on see why we should prefer it over Inheritance.

Inheritance is a mechanism by which one object acquires all the properties and behaviours of a parent object. An inherited class is called a subclass of its parent class or superclass. As an example, consider the base class Guitar defined like this:

// base class

**class** **Guitar**

{

**public** string type;

**public** string model;

}

Now we can define another class called ElectricGuitar which will be the derived from the Guitar class, and hence would share all the properties and behaviours of its parent class (i.e. Guitar class):

**class** **ElectricGuitar** **extends** Guitar {

// some more methods...

}

Why is composition preferred over inheritance? There are some advantages:-  
1. In inheritance, the connected classes are very tightly coupled. This essentially means that if the parent class changes even a little bit, then it would drastically affect the child classes as well.

Consider the following code:-

**public** **class** **A** {

**public** **void** **foo**() {

   }

}

Public Class B **extends** A {

**public** **void** **bar**(){

       foo();

   }

}

Class A has a function called foo(), and another class B has a function bar() which is calling the function foo(). The classes A and B are tightly couples. Suppose that in future, in class A, we declare a function such as this:-  
public string bar()

This change has come about in class A, but due to the same function declaration, the function bar() in class B would become unusable, and the code inside class B would not compile.

How does composition help to solve this problem? Composition means that the tight coupling isn't needed, and objects can be combined at runtime. Look at the following code to see how composition solves this problem:-

**public** **class** **A** {

**public** **void** **foo**(){

    }

}

**public** Class B {

    A a = **new** A();

**public** **void** **bar**(){

        x.foo();

    }

}

2. If class A inherits from class B, then class A must necessarily share and inherit all its attributes from class B. This means, that any method (say method X) must have the exact same implementation both in classes A and B. What are the implications of this? This would very tightly bound all the methods of class A to all the methods of class B, and we can no longer modify or customize any of the inherited methods! Suppose class A is a square which has an area method defined as (side\*side), and class B rectangle inherits from class A. Then, class A must necessarily have the area method in the exact same way as that of class B! This is a huge impediment when it comes to defining and declaring classes which are closely related to each other, but with subtle differences.

3. Additionally, if class A inherits from class B, then the public/protected methods and attributes of class A would be exposed to class B. This can lead to a lot of security issues especially in those systems (like networks and databases) which require a greater privacy when it comes to certain classes.

4. Moreover, suppose we are creating a simple MS Paint-like application and we have to create some toolbars which can be accessed from a menu. Now, if we create a parent class "Window", inside which we create an inherited class "Menu", inside which we create another class of "Toolbar", this would mean that whenever we have to render a simple Toolbar, it would place its parents objects also in the stack! This would make the rendering stack very heavy as there are a lot of nested subclasses inside a single class. Even rendering a simple object or toolbar would take a lot of time. Thus, it is not advisable to use inheritance here.  
 Q. **nheritance & composition**

Consider the following piece of code where various classes have been declared:-

**class** **Motherboard** {

// some methods

}

**class** **Processor**{

//some methods

}

**class** **Mouse**{

//some methods

}

**class** **Mousepad**{

//some methods

}

**class** **Keyboard**{

//some methods

}

**class** **Keypad** {

//some methods

}

**class** **Computer** {

**private** Motherboard m;

**private** Processor p;

**private** Mouse mo;

**private** Keyboard k;

}

**class** **Laptop** **extends** Computer {

}

Should we use inheritance in the Laptop class here?

**No**

**Feedback :**

*If Laptop inherits from the computer class, the public and protected methods and variables  of the Computer class will be inherited by the Laptop class, whether those methods or variables will be useful to the laptop class or not. However, a laptop doesn't have a mouse but it has a mousepad. Also, the Keypad is missing from the Laptop class as the laptop has a keypad, not a keyboard. Hence the use of inheritance is incorrect*

**Inheritance & composition**

In the below code, how do we overcome the limitation of the inheritance?

**Suggested Answer**

Instead of inheritance, we can use composition as follows:

**class** **Motherboard** {

// some methods

}

**class** **Processor**{

//some methods

}

**class** **Mouse**{

//some methods

}

**class** **Mousepad**{

//some methods

}

**class** **Keyboard**{

//some methods

}

**class** **Keypad** {

//some methods

}

**class** **Computer** {

**private** Motherboard m;

**private** Processor p;

**private** Mouse mo;

**private** Keyboard k;

}

**class** **Laptop** {

**private** Motherboard m;

**private** Processor p;

**private** Mousepad mp;

**private** Keypad kp;

}

Here the Laptop class doesn't inherit the properties of the Computer class. Thus coupling between the Laptop and Computer classes are reduced. In addition,  using composition, the Laptop has all the required components to be functional.

Now having learnt what design patterns are, let’s look at how they are characterised in the following video.

Design patterns are broadly categorised as follows:  
a. **Creational design patterns** are those which define HOW classes and objects will be created. Different classes of objects will follow different mechanisms for creating objects. Another major function of the creational design patterns is to hide the actual implementation of the classes from its usage. This pattern is useful is the actual implementation of the classes is not of much use to the user, and the user only cares about the final created product.

b. **Structural design patterns** are used to define the relationships between different classes. They are mainly concerned with how the different classes and subclasses are organized amongst each other & how classes and objects are composed to form larger objects and structures. They differ from the creational patterns in the way that while creational patterns focus on how classes and objects are created, while structural patterns focus on how these different classes are related to each other.

c. **Behavioural design patterns** are those design patterns which help us define how the relationship between different classes and objects, and how these classes and objects will communicate with each other. These patterns are concerned with how different responsibilities are shared amongst different classes. They help us define and streamline the complex flow of information between different classes.

**tructural and Behavioral patterns**

Which of the following statements about structural patterns and behavioural patterns are true?

Top of Form



**Structural patterns are mainly concerned with how classes and objects are organised amongst themselves.**

**Feedback :**

Structural patterns are mainly concerned with how classes and objects are organised amongst themselves.

**Behavioural patterns are mainly concerned with how classes and object communicate with each other.**

**Feedback :**

Behavioural patterns are mainly concerned with how classes and object communicate with each other.

Bottom of Form

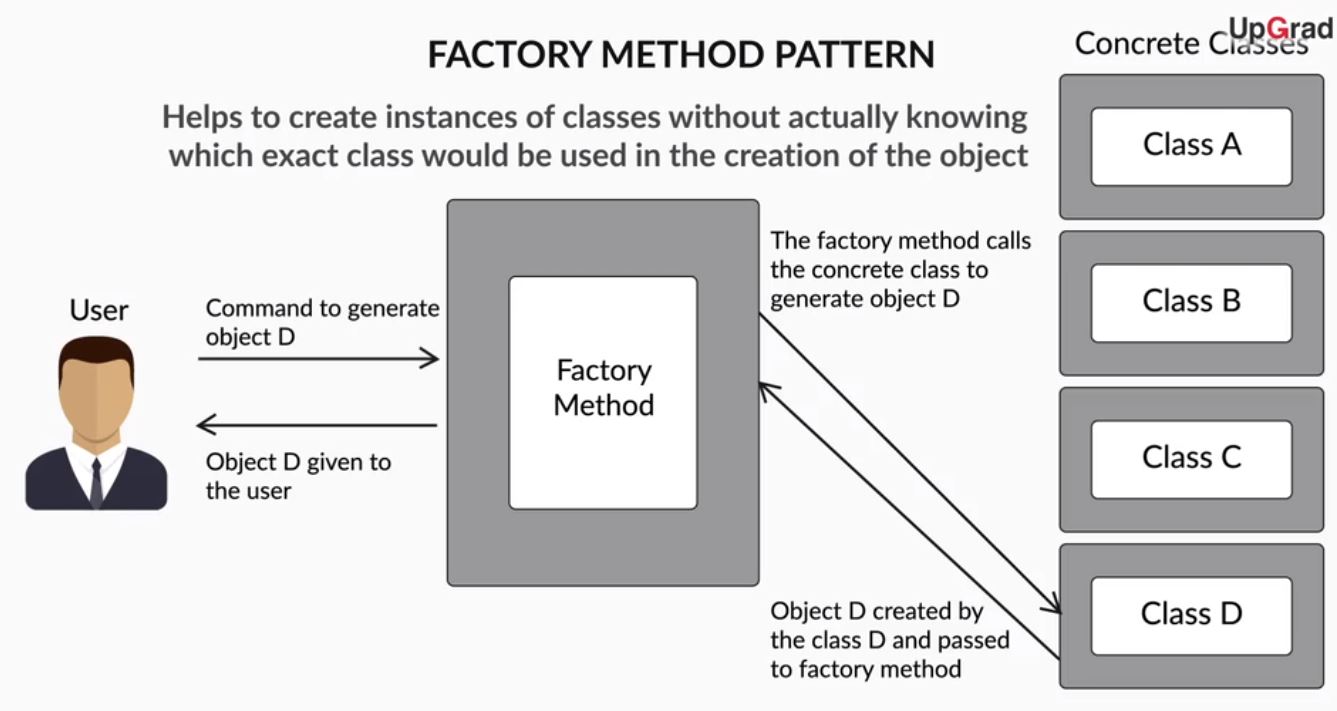
# Creational Design Patterns - Factory Method

In the previous segment, you learnt creational patterns and its usage. Specifically, Creational design patterns define HOW classes and objects will be created.

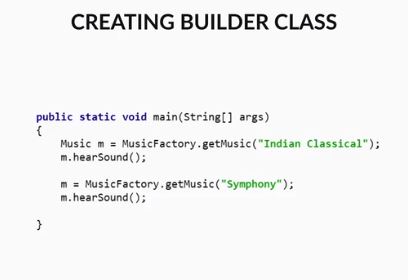
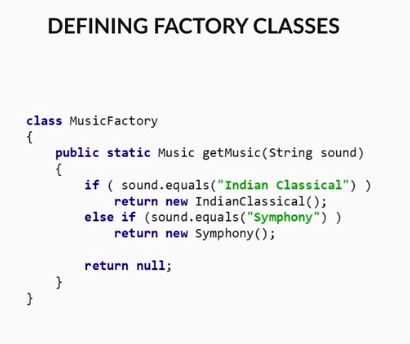
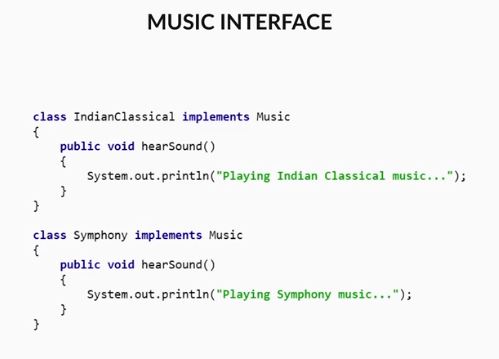
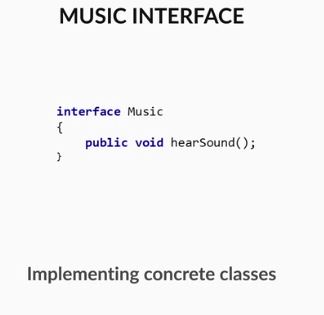
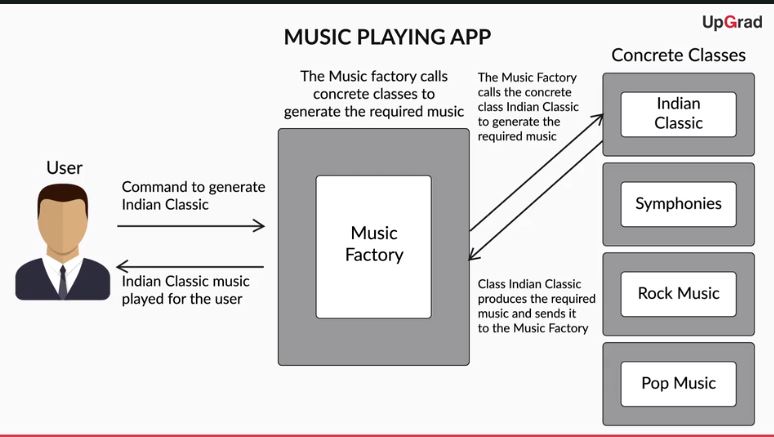
Now let’s imagine a scenario.

Suppose you are a tech lead in a company and you hire new employees via a hiring agency. Based on the availability, you tell the hiring agency your requirements and the role for which you wish to recruit. The hiring agency gives you the names of the candidates which they have hired. You provide your requirements and the number of vacancies, and you aren't exposed to the various underlying steps like creating job postings, looking up for candidates, verification, shortlisting the resumes, the interview process etc.

Factory pattern works exactly in situations like above, where the concern is only the final product and not the underlying steps. For example, like described, you are only interested in the final candidates and not how the list was created In the following video, we will learn more about the factory pattern and how is it used in different situations.



In the previous video, you learnt about the factory method pattern. The factory method pattern defines an interface for creating an object, but let subclasses decide which class to instantiate i.e., the factory method creates an object without exposing the creation. Now let’s see the implementation of the factory method pattern through an example in the following video.



# Creational Design Patterns - Builder

In the previous segment, you learnt about the factory method pattern and its implementation. Factory Method pattern helps to create instances of classes (i.e. objects) without actually knowing which exact class would be used in the creation of the object.

Let's again take the example that we discussed in the previous segment of recruiting candidates for your team. Instead of the hiring agency suppose you wish to recruit yourself. In order to recruit candidates for several positions you would have to undergo several steps in order to get the final recruited candidates like posting jobs for your company, collecting resumes, shortlisting the eligible candidates, conducting various rounds of interviews, salary negotiation and then you get the final recruited candidates. Notice here that the end product is the same (the recruited candidates) however, the difference lies in the process of recruitment. You cannot apply the factory method pattern here as it doesn't deal with the underlying methods and processes but only with the final product. Moreover, the factory method pattern is used in cases wherein object creation is a single step process. In situations like these where the object creation is multi-step and complex, we use another type of design pattern known as the builder design pattern.

In the following video, we will explore more about the builder pattern and how is it different from the factory method pattern

Q. How do you think the builder pattern is different from the factory method pattern?

**Suggested Answer**

The factory method pattern acts as a wrapper around a constructor and builds the entire object in a single method call. Thus the factory method pattern is used when the object creation is a single step process. The builder pattern, on the other hand, is used when the object creation process is a multi-step process. The builder pattern creates the complex objects by encapsulating the creation and assembling classes separately in a builder object. This allows the classes to delegate the object creation instead of directly creating the objects.

**Q. Creational Design Patterns**

Imagine that you are a furniture manufacturer who takes requests from customers and builds the required furniture for homes such as cupboards, doors, etc. A customer asks you to build a window for his bedroom. Which of the below design pattern would you use?

Factory Method

**Feedback :**

*The factory method is to create simple objects which can be created in a single step. Here you can use the factory method to create the window as it is a single step process.*

Q. **Creational Design Patterns**

Imagine that you are a furniture manufacturer who takes requests from customers and builds the required furniture for homes such as cupboards, doors, etc. A customer asks you to design his bedroom. Which of the below design pattern would you use?

Top of Form



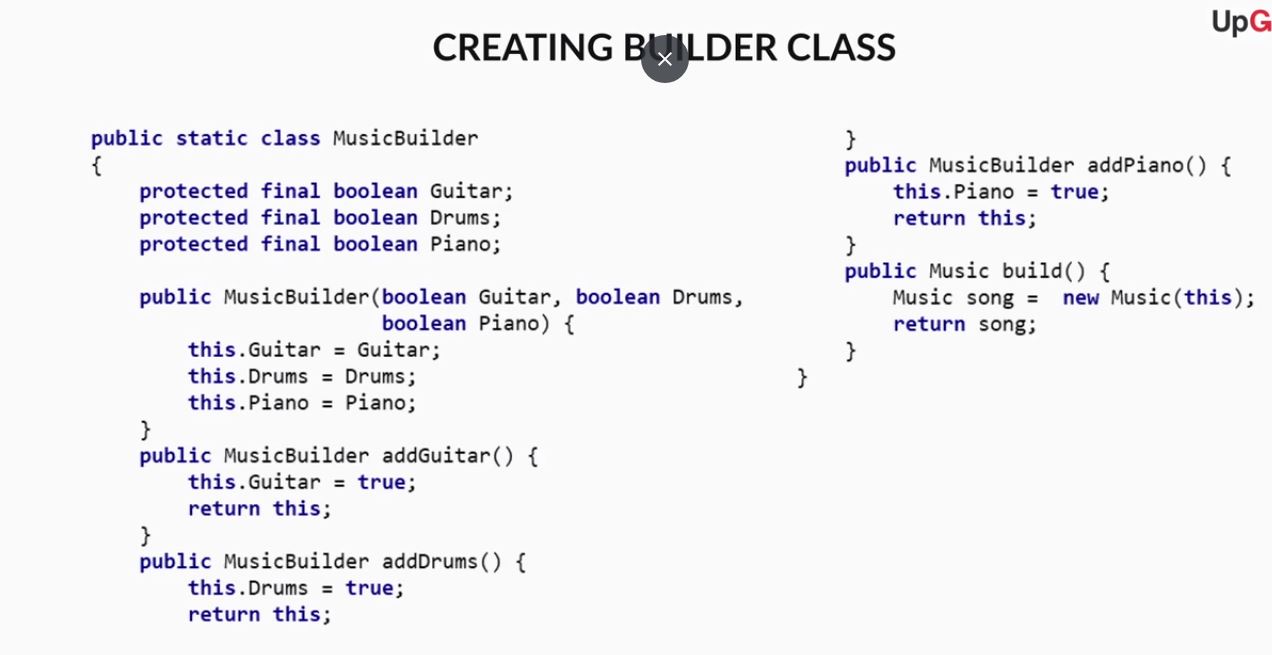
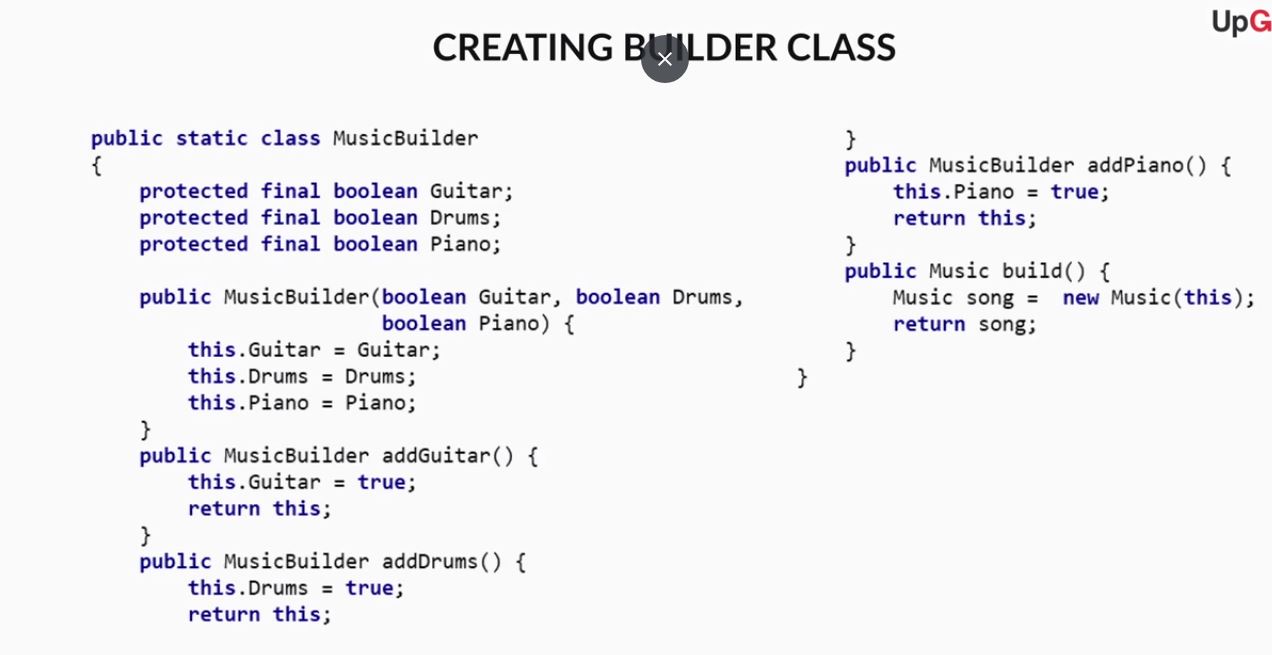
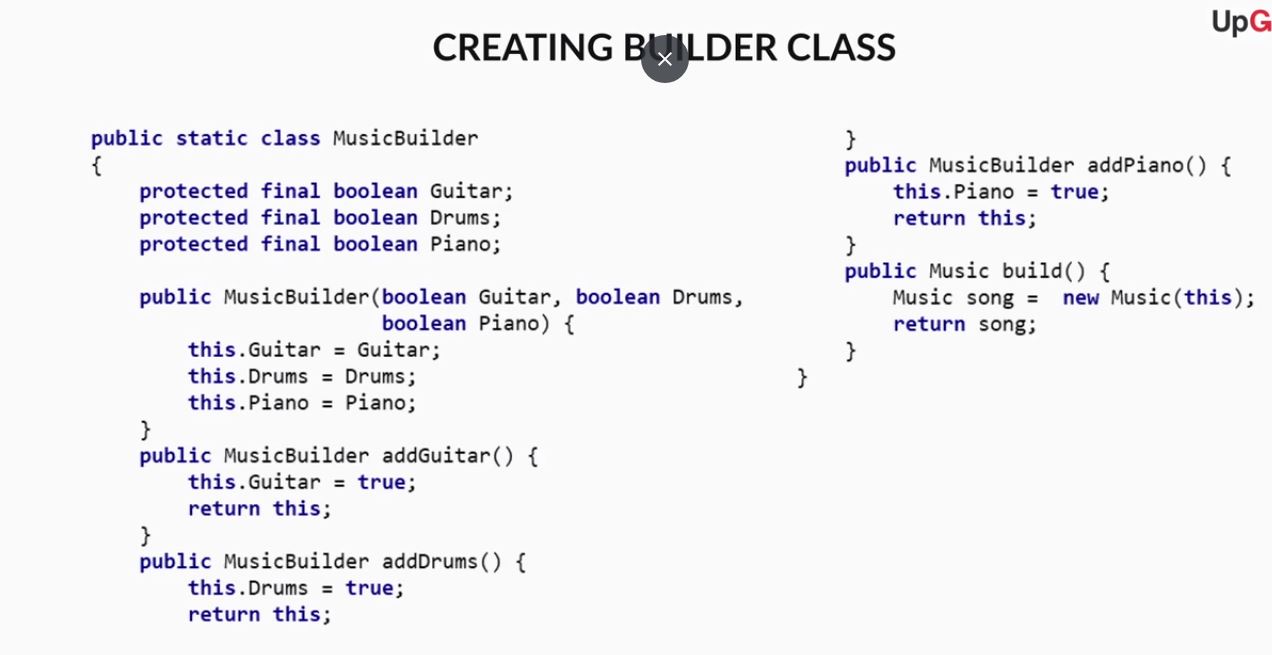
**Builder**

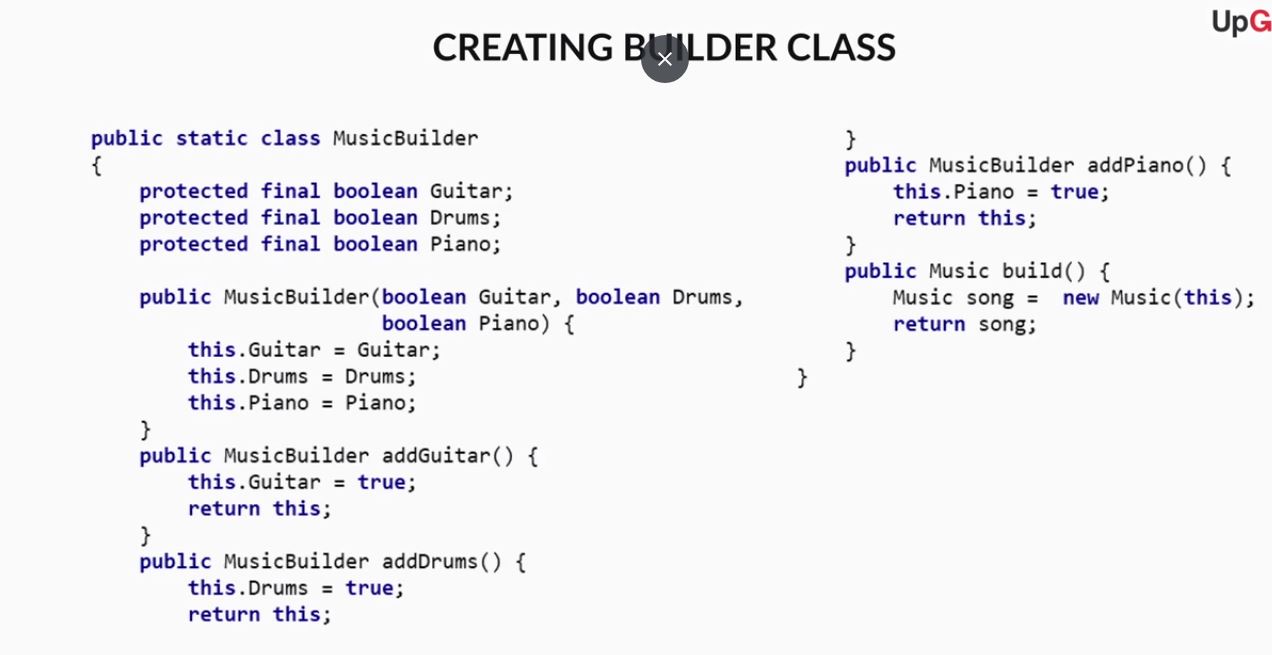
**Feedback :**

*The builder method is used to create objects when the creation process is a multi-step process. The bedroom consists of several entities such as cupboard, bed, door, window etc. To create objects which themselves comprise smaller sub-objects (such as a bedroom comprises of individual doors and windows), we need to use the builder pattern as there are many steps involved. Hence, here the builder pattern will be used which will create the bedroom.*

The builder pattern aims to enable us to create complex objects. The objects can be created through a step by step process with the final step, returning the object.  The factory method pattern is used when a simple object needs to be created in a simple step while as for the construction of complex objects in a step-by-step manner the builder pattern is used.

Now let’s look at how the builder pattern is implemented.

Bottom of Form



**package** **com.upgrad.assignment**;

**class** **Music** {

**private** **final** **boolean** guitar;

**private** **final** **boolean** drums;

**private** **final** **boolean** piano;

**private** **Music**(MusicBuilder builder) {

**this**.guitar = builder.guitar;

**this**.drums = builder.drums;

**this**.piano = builder.piano;

}

**@Override**

**public** String **toString**() {

**return** "Guitar : " + guitar + ", Drums : " + drums + ", Piano : " + piano;

}

**static** **class** **MusicBuilder** {

**private** **boolean** guitar;

**private** **boolean** drums;

**private** **boolean** piano;

Music **build**() {

**return** **new** Music(**this**);

}

MusicBuilder **addGuitar**() {

**this**.guitar = **true**;

**return** **this**;

}

MusicBuilder **addDrum**() {

**this**.drums = **true**;

**return** **this**;

}

MusicBuilder **addPiano**() {

**this**.piano = **true**;

**return** **this**;

}

}

}

**public** **class** **MusicMain** {

**public** **static** **void** **main**(String[] args) {

Music song = **new** Music.MusicBuilder().addGuitar().addDrum().addPiano().build();

System.out.println(song.toString());

}

}

The builder design pattern provides the following advantages:  
1. It allows the step by step building of products wherein each step adds or sets some property of the object under creation.  
2. Different representations of the object can be made using the builder pattern. Since the object creation is a multi-step process and we can leverage different classes at each step to create a new representation.   
3. The same code can be used for the construction of different objects.

# Creational Design Patterns - Singleton

In the previous segments, we discussed two important types of creational design patterns: Factory method and Builder.

Now let’s look at another scenario.

Suppose you work as a database manager and, to manage the connections to the database, you create a pool of database connections for your team to connect to the database. The database pool is expensive to create and destroy. Hence it should be created only once.

Anyone who wishes to connect to the database should use this database pool only. Here the pool should be global in order to be accessible to all in your team.

The factory method and builder patterns cannot be used in this case as they always create a new object whenever they are called, which is not desired in this case. Hence we need to use a new design pattern known as the singleton pattern to create this pool of database connections once and only once. In the following video let's learn what it is and how it is used.

Imagine we have an OS in which multiples threads and process are in action. In such scenario system is often prone to errors due to inconsistency in threads, deadlocks.. For consistent way to keep system is to maintain a log where all programs are registered, which acts aa a book keeping mechanism system, but due to some program malfunctioning this log will be erased again and system remains as inconsistent system.

Q. Which of the following statements about creational and singleton patterns are true? (Choose all that apply)

They are mainly concerned with how classes and objects are created.

**Feedback :**

Creational patterns help us to formulate how classes and objects will be created.

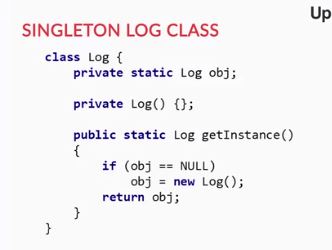
**Correct**You missed this!



**Singleton pattern is used in cases when we need to create a static and global object.**

**Feedback :**

Singleton pattern is used in cases when we need to create a static and global object in our system.



Singleton pattern is a software design pattern which is used in cases where only ONE class object is used to perform some specific actions in the system. It is used to restrict the instantiation of a class to exactly one object. The intent behind using the singleton pattern is to ensure a class only has one instance and provide a global point of access to it.

The use of the singleton pattern provides the following advantages:  
1. It ensures that a class has only a single instance  
2. It that the single instance of the class can be accessed global access by other parts of the program

Q. **Creational patterns**

In your project, you use github for software development. You wish to create a generic account that provides access to all the developers with the same privileges. The generic user will have single configuration and provides equal access to all the developers working in the code development. And this master user once created cannot be created again. Which of the below design patterns would you leverage to implement this?

**Singleton**

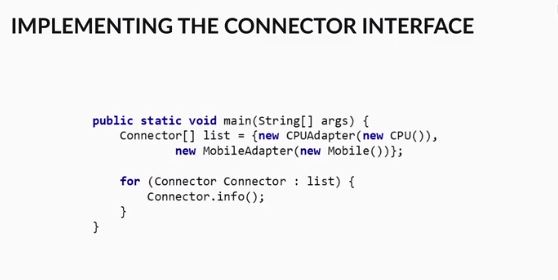
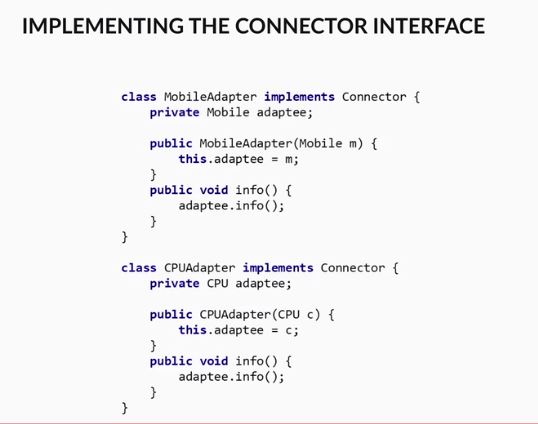
**Feedback :**

*The problem requires a design pattern that will return a single instance of the resource (generic user in this case) whenever the method is called. The singleton pattern is used where only ONE class object is used to perform some specific actions in the system. In case the generic user is created with the help of the singleton pattern, the singleton pattern will restrict the instantiation of this class to exactly one object which is what is required in the given design. Hence, this is the correct design pattern.*

# Structural Design Patterns - Adapter and Decorator

Recall in the first segment; you learnt that the structural design patterns are used to define the relationships between different classes. In this and the next segments, we will discuss the types of structural design patterns.

Suppose you have a CD that you wish to play on your TV how you would play it? Here the TV and the CD are two incompatible classes which cannot interact directly. The design patterns that you studied previously, i.e., the creational design patterns will not be used in this case as we aren't creating an object here but wish to provide a communication link between two incompatible objects or classes. In such scenarios, we use a structural design pattern known as the adapter design pattern. Let's discuss the adapter pattern.



An adapter design pattern is used to link two interfaces or classes which are otherwise incompatible with each other. It does so by wrapping these incompatible objects in a wrapper, such that they can then be linked to a common interface.

The use of the adapter pattern provides the following advantages:  
1. It allows two otherwise incompatible classes to work together to achieve some design goal, which provides the benefit of increasing design flexibility and code reuse in your application.  
2. It hides the unnecessary code implementation details of data and interfaces from the client

Opinions: **Coming back to the CD and TV example, how do you play the CD in your TV? The answer is a CD player. The CD player connects to the TV and runs the CD to play its contents. Using the adapter pattern, can you write down the code how this can be achieved? You would need to define two classes first the TV class and the CD class, and then create a common connector interface. Using the CD player as a wrapper for both the classes (TV and CD), link the classes via the common interface.**

**rakesh malhotra**

4 months ago

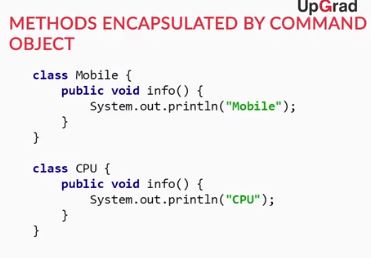
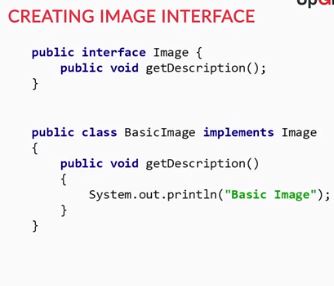
class TV { public void play(){ System.out.println("Playing TV");} }  
class CD { public void play(){ System.out.println("Playing CD");} }  
public interface connector { public void play() ;}  
  
class TVadapter implement connector {  
private TV tvadaptee;  
public TVadapter(TV tv){ this.tvadaptee = tv};  
public void play(){ tvadaptee.play()};  
  
class CDadapter implement connector {  
private CD cdadaptee;  
public CDadapter(CD cd){ this.cdadaptee = cd};  
public void play(){ cdadaptee.play()};

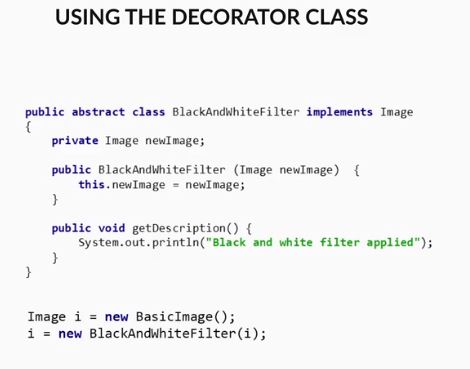
In the previous video, you learnt about the adapter pattern. Now consider a scenario wherein you own a newspaper website. The newspaper website displays news articles written in your newspaper to your viewers. The website has two types of viewers:  
- The subscribed viewer who has paid a monthly fee to your newspaper  
- The unsubscribed viewer who has not paid a monthly fee to your newspaper  
You want your website to dynamically adjust how much of a newspaper article will be shown to the user at runtime  
- The subscribed viewer gets to see the entire news article  
- The unsubscribed viewer gets to see only the first paragraph of the news article

Here the creational design patterns cannot be used as there is no object creation needed.   
Instead, the news articles are already created, and there is a need to regulate the content at the program runtime. Specifically,  we need to display different news article content for different categories of users.

The adapter pattern will also not be useful in this case because there are no incompatible classes or objects to connect here. In such scenarios, we can use another type of design pattern known as the decorator design pattern. Let's learn the decorator pattern in the upcoming video

Example of Decorator, before you post images on insta you can edit the images that were captured in insta before uploading dynamically.





The Decorator pattern is used to attach additional responsibilities to an object dynamically. It is used when we need to dynamically change the behaviour of an object at run-time or add some additional behaviour to it. In simpler words, the decorator pattern will allow you to add extra functionalities to an object at runtime without changing the object’s interface or underlying implementation.

The use of decorator pattern provides the following advantages:  
1. It allows adding and removing behaviours at runtime. Thus you gain the flexibility of writing simple classes and add functionalities to these simple classes as needed during runtime. Versus having to write large classes that implement all possible functionalities. This makes your code easier to maintain and easier to test.

2. It is more flexible than inheritance because inheritance adds responsibility at compile time, but decorator pattern adds responsibilities at runtime.

More specifically, with the decorator pattern, you gain the flexibility of dynamically adding or removing functionalities to your object while the application is executing.

On the other hand, if you want to add functionalities to a specific class through inheritance, you need to create a new subclass for every new responsibility that you want to add to the class. Thus, you may end up creating many different subclasses if you need to add many functionalities, and thus increasing the complexity of your application.

3. It extends the functionality of the object without impacting any other objects of the same class.

Q. **Creational patterns**

Suppose you have an antenna which only reads/transmits radio signals and a television which only reads/transmits AC/DC signals. You need to establish a system in place so that you can connect these two incompatible electronic types of equipment (antenna and the TV) so that the radio signals can be translated into AC/DC signals. You wish to establish a communication between them using a design pattern. Which of the below would you use?

**Adapter**

**Feedback :**

*The antenna and tv are two incompatible classes and there is a need for an interface or link that would enable them to communicate with each other. The design pattern should define the relationship between the two classes if you recall the adapter pattern is used to link two interfaces or classes which are otherwise incompatible with each other. Here the antenna and tv are incompatible with each other and the adapter will act as an intermediary to enable them to communicate with each other. Hence, the adapter pattern would be the correct design to use in this case.*

**Creational Patterns**

You have your blogging website wherein you share your thoughts on the latest developments in the technology. The length of each post varies, and you wish to have a dynamic scrolling, i.e., the scroll bar will adjust according to the size of the post once you publish a post.  Note that the behaviour of the scroll bar will change at run-time depending upon the how much the user has scrolled or written on the blog. Which of the below design patterns would you use for this purpose?

**Decorator**

**Feedback :**

*The length of the scroll is a dynamic function that should be added at the runtime. The design pattern used here should be able to add this additional function depending on the requirement after the object is created. The decorator pattern is used to dynamically change the behaviour of an object at run-time or add some additional behaviour to it. Using the decorator pattern here will fulfil our requirement as once the blog is created the decorator pattern will dynamically change the behaviour of the scroll when the user is using it, i.e., at the runtime. Hence, this is the correct answer.*

# Structural Design Patterns - Facade and Proxy

In the previous segment, you learnt about the adapter and the decorator patterns. Now let’s look at other types of structural patterns in this segment.

Suppose you plan on going a vacation to Andaman and look up for a travel package via an agent. Here there are many subsystems involved such as flight bookings, cab transfers, hotel bookings which are taken care of by the travel package. Thus you need not concern yourself with them.

Specifically, the agent provides you with the aggregate details of all the subsystems such as the flight schedule, the hotel bookings, the cab details and many more. This reduces your overhead of searching the flights, hotels. cabs etc and booking the same for the correct duration. In a way, you only interact with a high-level interface or specifically the travel agent in this case.

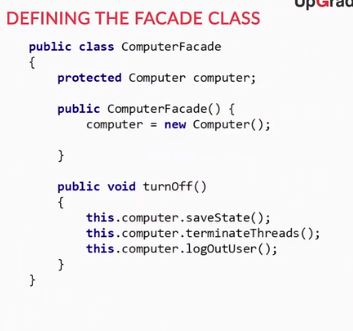
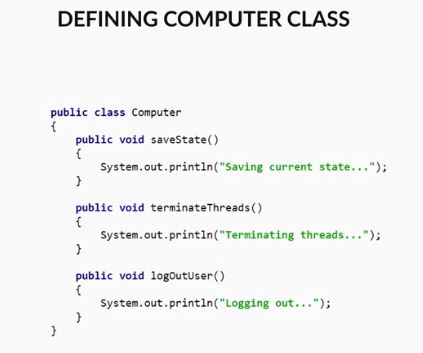
In case you were to design such a high-level interface which makes the subsystems easier to use, which design pattern would you use? Clearly, the creational design patterns cannot be used as we aren't creating any object here. The adapter pattern also would not be the correct one as there is no need to define relationships between two incompatible classes. The decorator pattern also is not the correct choice as there is no need to dynamically modify any parts of the system at the runtime.

In such scenarios, there is another type of design pattern used known as the facade pattern which defines the high-level interface to interact with the subsystems and hide their complexities. Let's learn more about the facade pattern in the following video

Façade Pattern provides a minimalistic and simple interface to a complex body of code.

Eg: If you want to shut down a computer, which does saving the state of programs, terminating the running threads and many internal operations.

By just pressing the dumb button (acts as facade)



**Facade pattern** a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. The facade pattern acts as a simple gateway between the user and a complex set of functions.

The use of the facade pattern provides the following advantages:  
1. It decreases coupling between the client and underlying subsystem packages/classes because the facade patterns collate all the related complex method calls and related code blocks through a single facade class. Hence, the client’s interaction with the system can be simplified as it only needs to interact with the facade class. Also, any change in the packages/classes doesn't impact the client as any underlying complexity of the packages/classes are encapsulated by the facade class.  
2. Since the system becomes loosely coupled, it results in the reduction of the dependencies between libraries or other packages. Thus making the application easier to use and maintain creating a more structured environment.

In the previous video, you learnt about the facade pattern. Now consider the example when you connect to a Wi-Fi of your company from your computer or laptop. While you are connected to the WiFi network, the company wish for all the web requests to be routed through central server, which acts as an intermediary between the employees and the rest of the world wide web.

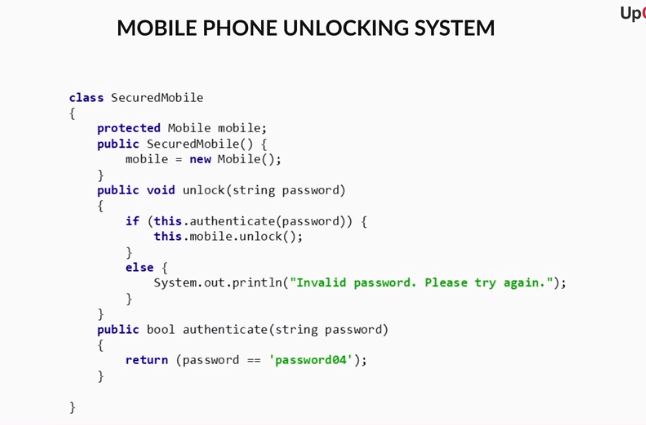
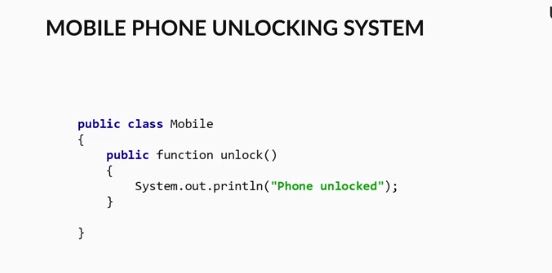
The server gives the company more control because the company can regulate the internet browsing and restrict the access of the certain website via this central server.

Which design pattern would you use to create such an intermediary system? The creational design patterns cannot be used as we aren't creating any object here. The adapter pattern also would not be the correct one as there is no need to define relationships between two incompatible classes. The decorator pattern also is not the correct choice as there is no need of modification at the runtime. The facade pattern is also not a good choice as it acts as a simple gateway to a complicated set of functionalities

Instead, here we need an intermediary that will provide us with the control to access the objects internet browsing in this case as the company would like to control the access via the central server. In such a scenario we use another type of design pattern known as the proxy pattern. Let's learn what it is and how it's used.

Proxy pattern acts as a placeholder for another object to control access to it.

Eg: If we want to design a pwd lock for a mobile phone, Here proxy does additional task like validating the pwd before mobile gets turn on



A proxy pattern is a surrogate class functioning as a simpler interface to another class. It provides an interface similar to the class for which it is acting as a proxy, but the proxy class can also perform additional functions to control direct access to that class.

For example, a proxy class usually provides additional functionality to a class that the user should not care or need to know the detail about. Thus, it does the job of merely acting as an interface, while performing some additional functions.

The use of proxy pattern provides the following advantages:  
1. It can act as a placeholder class for resources that might be huge in size or memory intensive and delay the allocation of the resources until the of is needed.   
2. It can provide additional security by restricting a user from directly accessing the class that the proxy class is substituting for

**Q. Structural Patterns**

There is a need to design a switchboard to enable the user to switch on light, fan and an AC. There are separate switches for each appliance in the switchboard. The user is only exposed to the switch for the appliances without knowing the complex operations of how the appliance is being switched on. The switchboard acts as an interface between the user and underlying complex operations. Which of the below design patterns should be used to design the switchboard?

**Facade pattern**

**Feedback :**

*The facade pattern is used when there is a need to hide the system complexities. Using the facade pattern will provide the required functionality as the facade pattern will provide the interface which will act as a gateway between the user and operations of the electronic appliances and hide*

**Structural Patterns**

You are asked to develop software for an ATM. The user is asked to enter their username and password. Once the user inputs are verified, the ATM will act as an intermediary between the bank and the user and enable the user to transact money. Which of the below design patterns would you use for implementing the ATM software?

**Proxy**

**Feedback :**

*The proxy pattern is used as a substitute for an object, and it implements additional functionality to regulate the access to the object. Using the proxy pattern would provide the required functionality as the given problem needs an intermediary which can provide the user with the access to the bank. The proxy pattern acts as a substitute to the bank and gives the access only when the user is verified.*

# Behavioral Patterns - Command design and Iterator

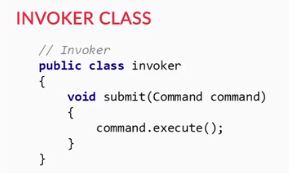
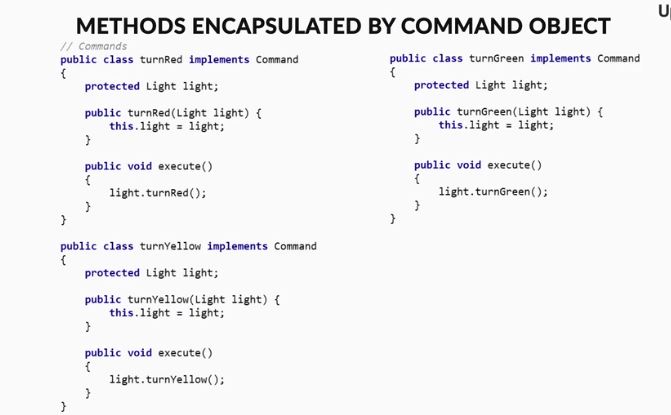
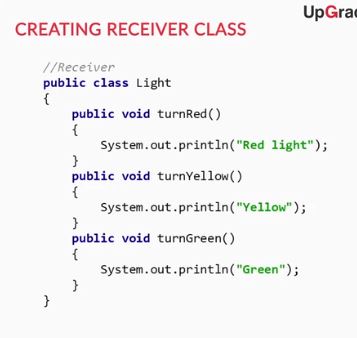
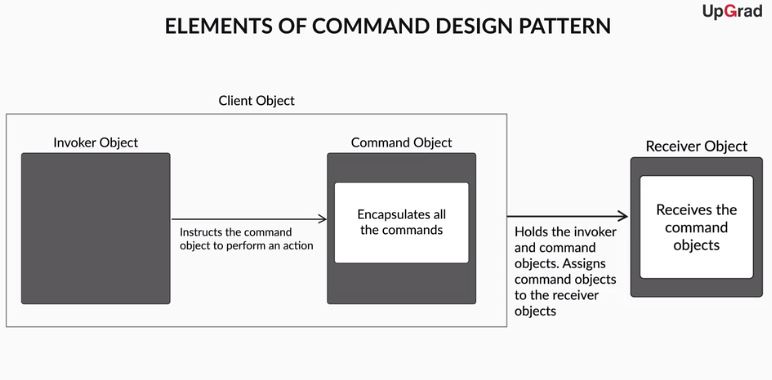
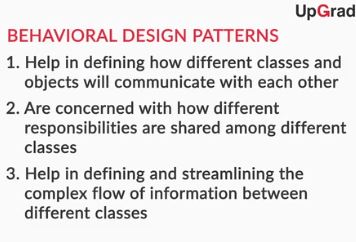
So far you have looked at creational as well as structural design patterns in this session. Now let’s look at another important type of design pattern: Behavioural design pattern. Recall that Behavioural design patterns are design patterns which help us define how different classes and objects will communicate with each other.

In this segment, you learn about two important types of behavioural design patterns: Command design and Iterator.

Let’s look at an example. Whenever you search for anything on Google, Google displays all of its results on the results page. If you click on the Google logo, you are redirected to google homepage. In case you wish to implement the same function wherein a click on the logo is captured and redirected to a predefined page, which design pattern would you use to create such a system?

Here you might want to encapsulate the commands and run them anytime the logo is clicked with commands instructing the browser to go to the home page. We cannot apply creational design patterns, e.g., factory method or singleton methods here as we aren't creating an object. Structural design patterns will also not be applicable because we don't need to define the relationship between classes. There is a need to have a design pattern that encapsulates the commands and then runs them whenever required. In such scenarios, we use another type of design pattern known as the command design pattern. In the following video, we will explore this

Eg: When should a traffic signal should turn red/yellow/green based on what commands from user?

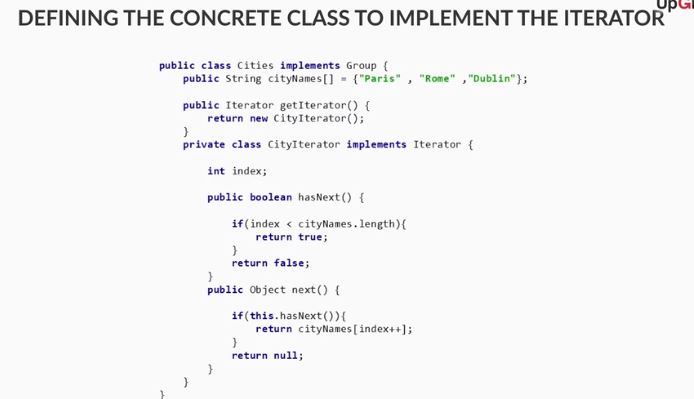
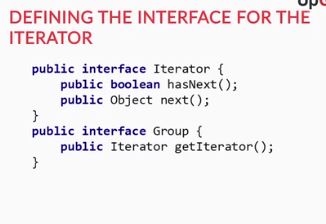


Command pattern is used to encapsulate a request as an object, thereby letting you parameterize clients with different requests. There are four primary elements involved in a Command Design pattern:-

1. A **command object** encapsulates all the commands which are needed to be executed at a later time. It also stores the parameters required to execute all the stored commands.  
2. An **invoker object** instructs the command object to act.  
3. A **receiver object**, which receives the command objects and knows what to actually do with the commands encapsulated in the command object.  
4. A **client object** holds the invoker objects, command objects and invoker objects. The client then assigns command objects to the receiver objects.

The use of the command design pattern provides the following advantages:  
1. It makes the system extensible as new command can be added without changing the existing code  
2. It allows deferred execution of commands  
3. It allows assembling simple commands into larger ones  
4. It also allows performing of undo or reversal of operations

We all like to listen to music and most of are familiar with the MP3 players that let us browse through our song collection. In older MP3 players, the song collection was traversed through simple forward and backward buttons. In case you wish to design a system that enables you to browse to through a collection of objects one by one without knowing the underlying representation as in how are the songs stored in the mp3 player, which design pattern would you use to create such a system?. We cannot apply the creational design patterns, i.e., factory method, singlet here as we aren't creating an object here. The structural design patterns will also not be applicable because we don't need to define the relationship between classes. The command design pattern is also not a good choice as there is are no predefined commands over here that need to run. There is a need for a design pattern that would allow you to traverse through a collection of objects without bothering about their underlying representation. In such scenarios, we use the iterator pattern. Let' learn it



The Iterator pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. The iterator lets us traverse a group of objects, without revealing the identity of the members of that group of objects.

The use of the iterator pattern provides the following advantages:  
1.  It promotes encapsulation by allowing you to traverse a collection of objects without the need to know how the collection of objects are implemented.   
2. It provides a unified interface for traversing different but similar collection of objects. For example, in Java, you can use the List iterator to traverse a collection LinkedList objects, and you can also use List iterator to traverse a collection of ArrayList objects.   
 3. It allows parallel traversing of a collection of objects.

**Behavioral patterns**

Suppose you have a music playing software having a lot of songs stored in the database. Whenever the user wishes to play a song, the name of the song or name of the artist or genre is captured in a variable "userInput" and then compared with the all the songs and artists in the database one by one. Once the song is found, it is then played for the user. Which of the following design patterns can be used to compare the input song with each & every song in the database one by one??



**Iterator**

**Feedback :**

*The iterator design pattern is used when there a need to traverse a group of objects, without revealing the identity of the members of that group of objects. The iterator pattern lets us traverse a group of objects, without revealing the identity of the members of that group of objects. Here the input can be anything ranging from song name, artist name or genre name, which means the input data types can vary. The iterator pattern supports different data types and traverses through them without exposing the internal structure as the user isn't aware how the songs are stored in the database. Also, the songs in the database may or may not be ordered according to the artist name, genre or song name, without knowing the actual data-type of the object or their order, the Iterator pattern can help to traverse such a list of objects. Hence, the use of the iterator pattern is the right answer.*

**Behavioral Patterns**

You are implementing a Microsoft word like application, and you need to implement an undo function. The undo function should restore the document back to its previous state. Here the previous states need to be stored in an undo stack, and whenever the undo function is called, we would execute a set of instructions to revert the document back to its previous state. Which design pattern would you use to implement this?

Top of Form



**Command Design**

**Feedback :**

*The command design pattern is used when there a need to encapsulate or contain different commands that are needed to perform at a later time. Using a command design here would provide the required functionality, as in the given problem there is a need to encapsulate the previous state of the document and restoring the document back to its previous state whenever the undo function is called. Hence, the command design pattern is the right design pattern to use in this case.*

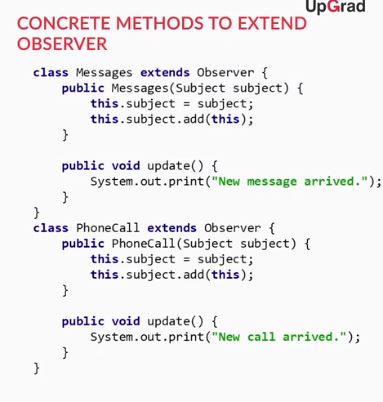
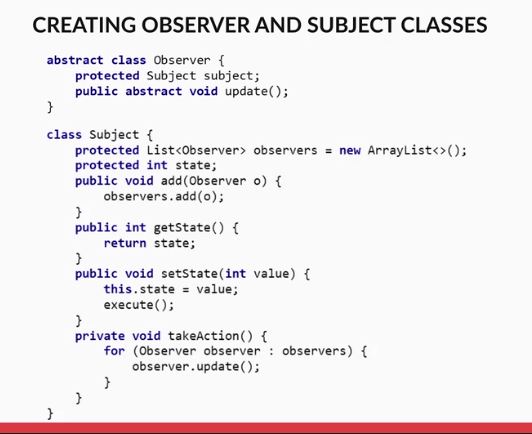
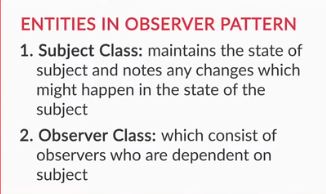
# Behavioral Patterns - Observer and Strategy

Continuing our discussion on the behavioural pattern from the previous segment, let’s look at its next type: Observer pattern.

Let's consider a case wherein software is being developed. Once the code for a particular module is deployed, all the developers on the project will be notified to go ahead with the coding of the next module. Note, to have an efficient system all the developers should be informed immediately which design pattern would you use to create such a system?

We cannot apply the creational design patterns, i.e., factory method or singleton method here as we aren't creating an object here. The structural design patterns will also not be applicable because we don't need to define the relationship between classes. The command design pattern is also not a good choice as there is are no predefined commands over here that need to run. The iterator is also not applicable as there is no need to traverse through a collection of objects. In such scenarios, we use another type of design pattern known as the observer pattern. Let's learn this design pattern

Eg: Phone notification system



Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. In the Observer pattern, there is a subject class, which maintains the state of the subject and notes any changes which might happen in the state of the subject. There is also an Observer class, which consist of observers who are dependent on the subject. Whenever the state of the subject changes, the subject class notifies all the members of the observer about the change in the state, and accordingly, an action can be taken later on.

The use of the observer pattern provides the following advantages:  
1. It decreases code coupling as the observers and subjects are independent from each other, and there is minimal coordination or dependence between the observer and the subject   
2. It increase application robustness as subscribers can be added to the system without any changes in the objects

Now let's consider a scenario wherein a bank is approving home loans for the applicants. The limit of the loan depends on the applicant's income. Below are the rules for the loan eligibility of the applicants:  
1. If the applicant's income is less than five lakhs pa, he/she is eligible for a loan of 1 lakh.  
2. If the applicant's income is more than five lakhs pa but less than ten lakhs pa, he/she is eligible for a loan of 5 lakh.  
3. If the applicant's income is more than ten lakhs pa but less than 20 lakhs pa, he/she is eligible for a loan of 10 lakh.

The bank wishes to devise a software system wherein a customer applied for a loan and based on his/her income the loan eligibility is calculated and the loan gets approved or rejected. Here there is a need to change the rule applied for the loan eligibility at the runtime depending on the user income. We cannot apply the creational design patterns, i.e., factory method, singlet here as we aren't creating an object here. The structural design patterns will also not be applicable because we don't need to define the relationship between classes. The command design pattern is also not a good choice as there is are no predefined commands over here that need to run. The iterator is also not applicable as there is no need to traverse through a collection of objects. The observer again will not be applicable as we aren't observing the change in the state of anything to inform the dependent components. In such scenarios, we use the strategy design pattern that you will learn in the upcoming video.

EG: take emp salary as input and produce tax calculation

**Strategy pattern**

Which of the following do you think is the difference between the strategy pattern and factory method pattern?

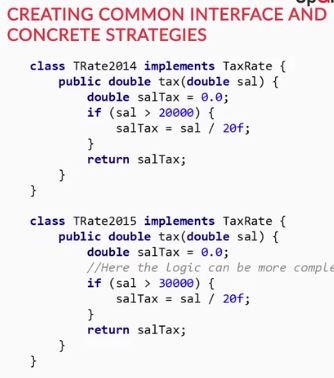
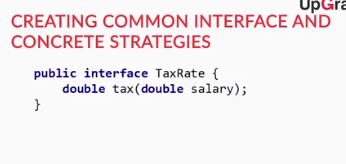
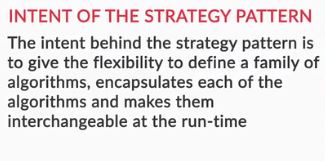
Top of Form

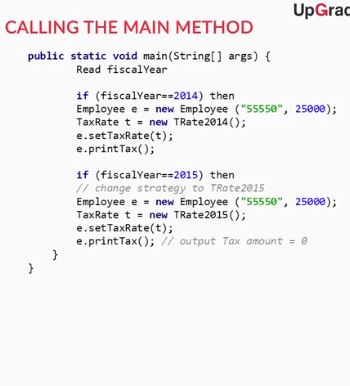
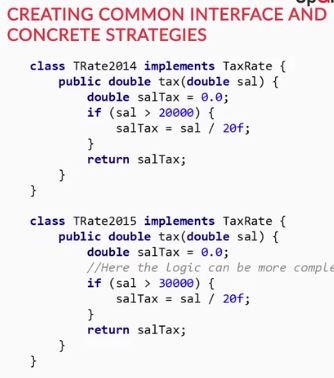
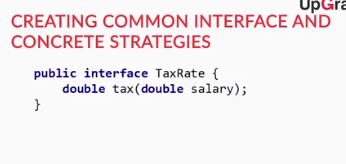
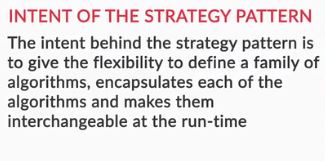


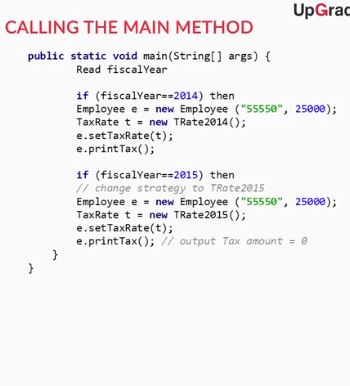
**The factory pattern is used to create the objects while as the strategy patterns is used to select different algorithms for a task or situation**

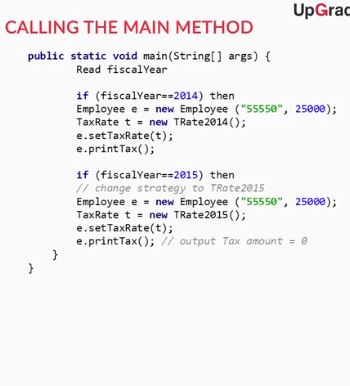
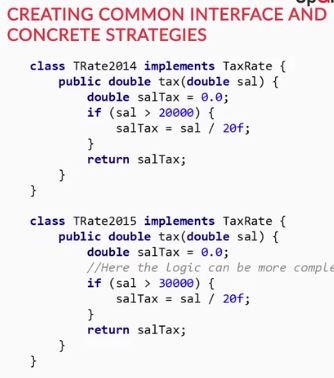
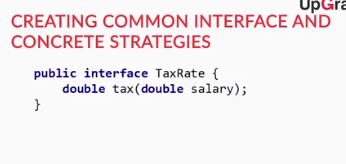
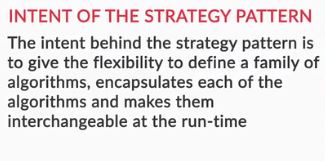
**Feedback :**

*Yes, the factory method pattern is used to create objects by acting as an interface between the user and concrete classes and delegates the task of the object creation to the concrete classes without exposing them to the user. While as, scenarios where there might be multiple Algorithms / Methods to solve a problem, but only one appropriate for a particular situation must be used at one time. It helps us to maintain flexibility in choosing the best or the most appropriate algorithm/method of calculation at runtime. Hence, this is the correct answer.*



Bottom of Form



Bottom of Form

Bottom of Form

# Introduction to Design Patterns Industry Demonstration

In the previous sections, you have seen what are design patterns, what are their advantages and what are the different categories of design patterns.

In the coming sections, you are going to read and learn about how design patterns are put to use in actual software development and how you can create your design patterns depending upon the situation.

**Design patterns**

Can you explain, in your own words, what do you mean by design patterns?

**Suggested Answer**

**A design pattern provides a general, well-tested & reusable solution to a common design problem. These are peer-reviewed software solutions to common problems and issues we run into in software development. It can be understood as a “blueprint” of how to solve a complex, multi-dimensional, and recurring problem.**



# An Example through Observer Pattern

In the coming video, you are going to see what design patterns are and how they can be put to use in a practical circumstance. Imagine a scenario wherein you need to create a notification system for a software. How would you go about doing it? Which design pattern would you use?

Let us see in the coming video

Q. **Observer pattern**

**Explain the logic behind observer pattern in your own words.**

**Suggested Answer**

**The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.**

**Q. Inefficiencies in the system**

**Under which of the following scenarios will the above implementation be rendered inefficient for the Facebook system?**

**A new user "User2" is created in the Facebook system**

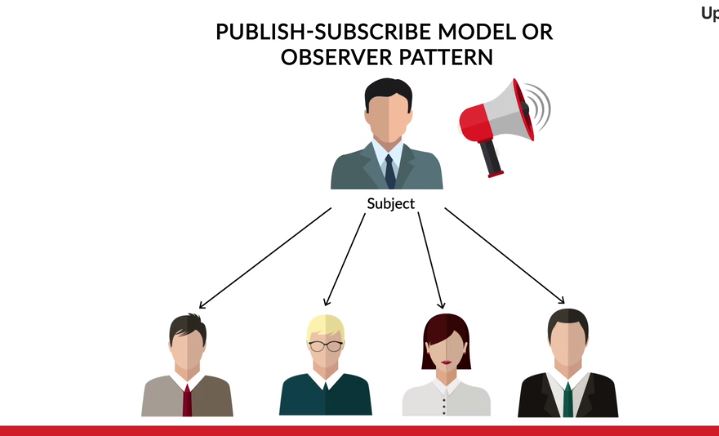
**Feedback :**

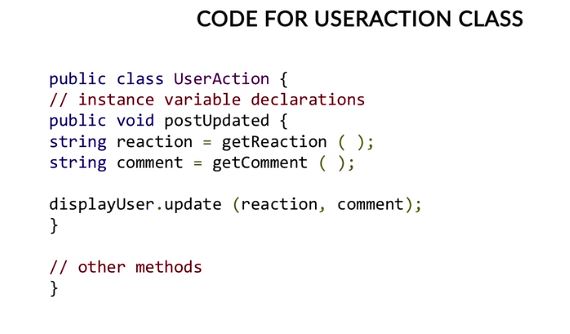
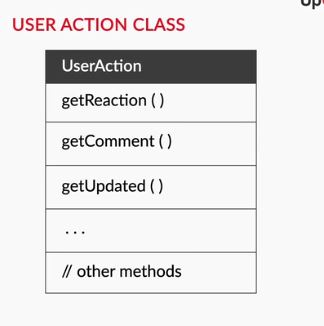
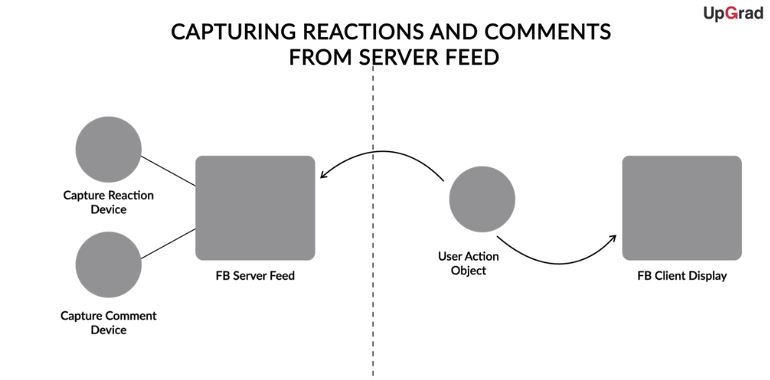
**When a new user comes in the system, we would have to write update functions for all such new users, which is a cumbersome task.**

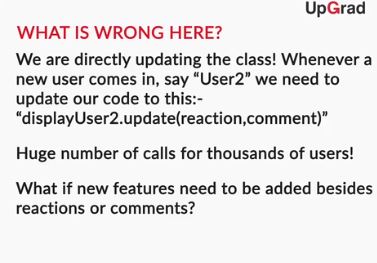
**A new element called "Feeling" is introduced in the system.**

**Feedback :**

**If a new element called “Feeling” would be introduced, we would have to modify the structure of our UserAction to incorporate that. It would require us to change the internal structure of the class, which is certainly not desirable and efficient.**



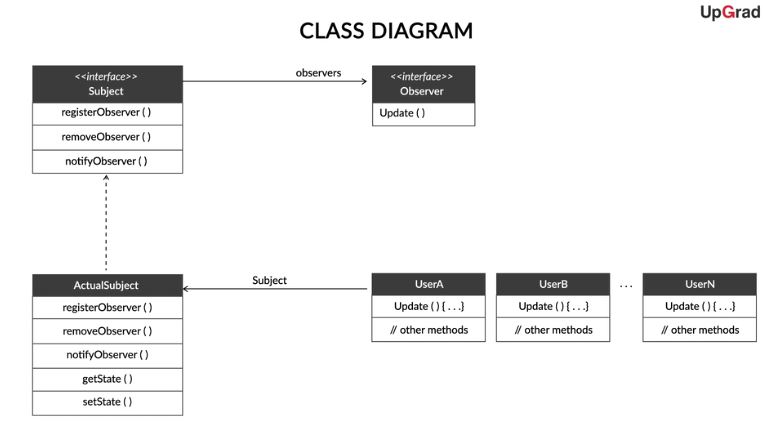
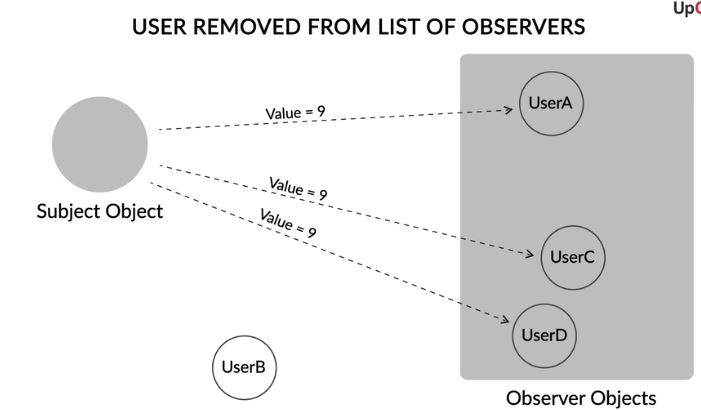
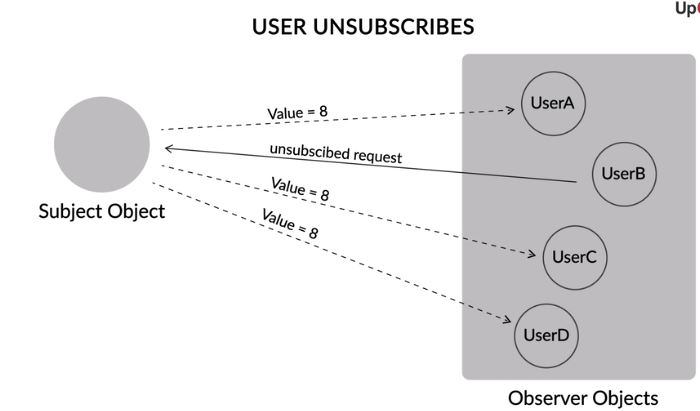
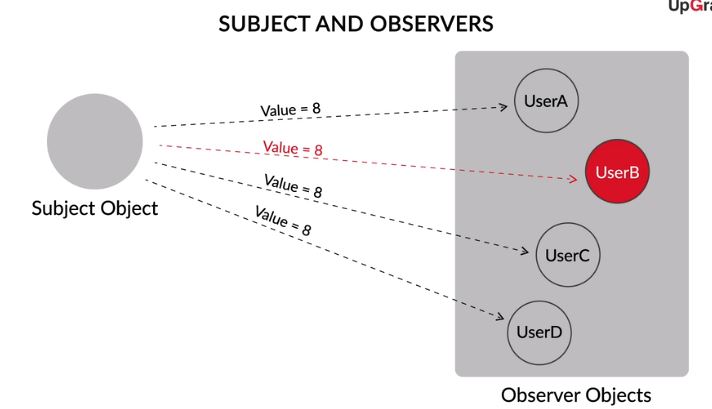
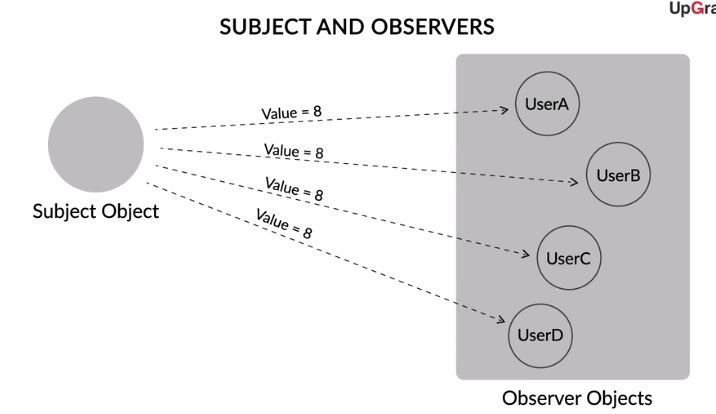
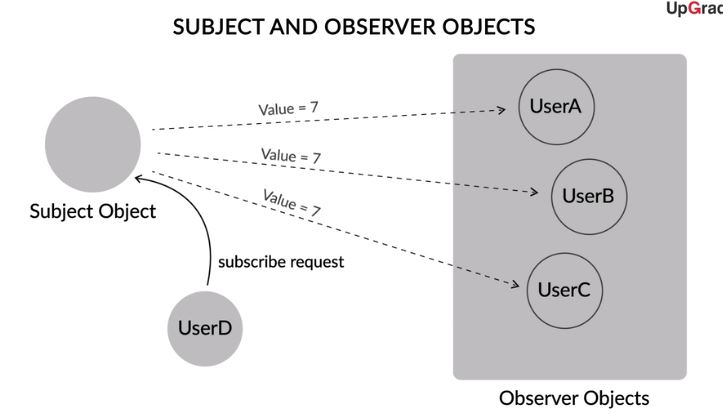
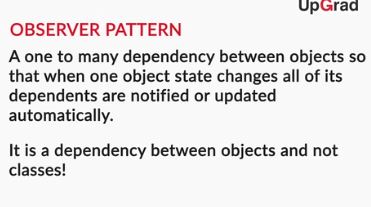




# Subject-Object Model

In the previous video, you saw briefly how we could go about implementing the notification system we wanted for Facebook. We also saw that the simplest implementation which we came up with had two major problems in it. Can we do anything about these problems and make our solution more efficient?

Let us see in the coming video.



**Design**

Which of the following statements about the above design are true?

Top of Form

**The actual implementation of the methods happen in the ActualSubject class**

**Feedback :**

The actual implementation of methods must happen inside the class which is implementing the interface. Hence, the actual implementation of the methods happen in the ActualSubject class.

**Correct**



**The Subject interface can only consist of the names of methods and not the implementation of any method.**

**Feedback :**

An interface can only consist of function and method names’s declaration and not its implementation.

**Observer pattern implementation**

Which of the following logical statements while designing the Observer pattern for our scenario are true?

Top of Form



**The job of registering or removing observer objects is done by the Subject interface.**

**Feedback :**

The Subject interface is responsible for registering or removing observer objects.

**Correct**

The job of updating the list of observer objects is done by the Observer interface.

**Feedback :**

The Observer interface is is responsible for updating the list of observer objects

**Correct**You missed this!

Now that you have seen how we have implemented our Observer pattern, let us see in the next video how does Java SDK implement the same observer pattern. We will also analyze what are the differences between the two kinds of implementation, and which one is more appropriate for our scenario.

# Java Observer Pattern

In the previous video, you saw our own implementation of the Observer pattern. How is it different from the Observer Pattern implementation defined by default in Java libraries? Let's learn this in the video below



Thus, in the above video, we have seen that although we had created our own version of Observer pattern, Java SDK has its own pre-defined version of the Observer pattern. We are free to use either the Java version, or define our own, as per our own needs. It is better if we learn to code our own design patterns since that would give us a direct hands-on experience on how to fiddle around with and apply design patterns.

**Observer pattern**

**Which of the following questions about Observer pattern model is true?**

**The object which changes its state is called a "subject"**

**Feedback :**

**The subject changes its state in the observer pattern system, and its changes are being observed.**

**Correct**



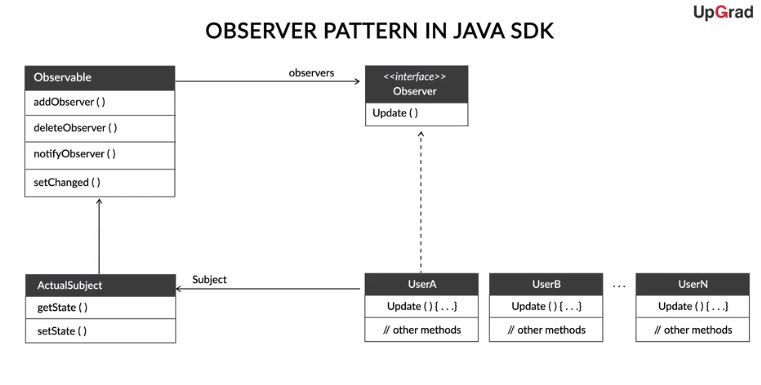
**The object which observes the change in state of another is called a "dependent" object**

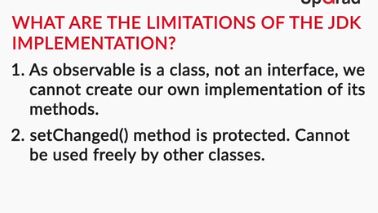
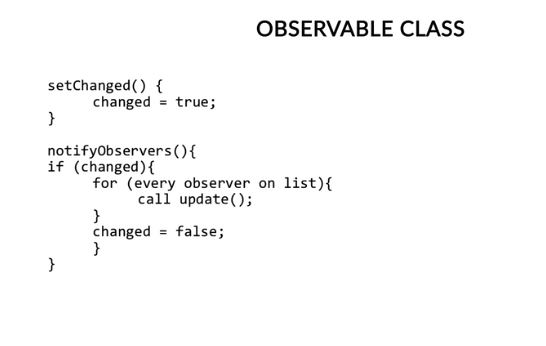
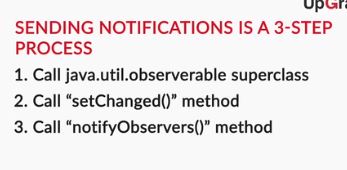
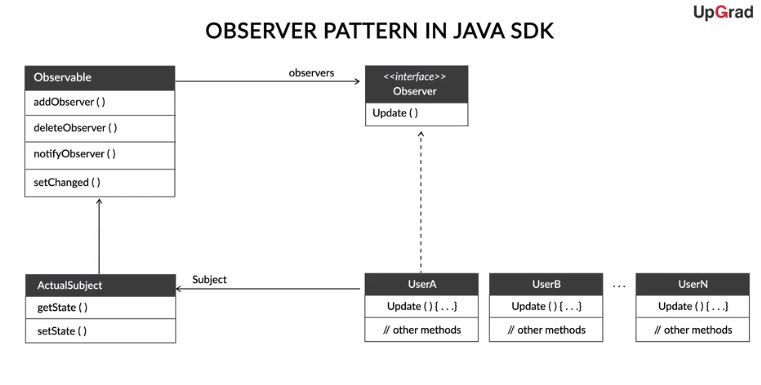
**Feedback :**

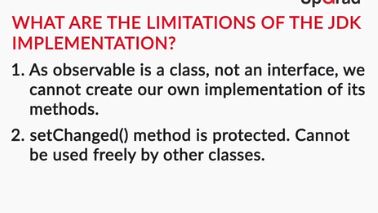
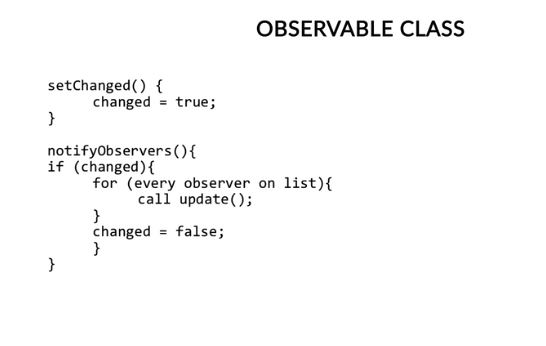
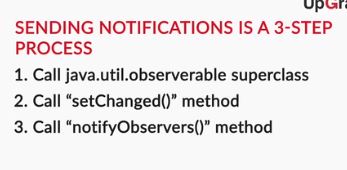
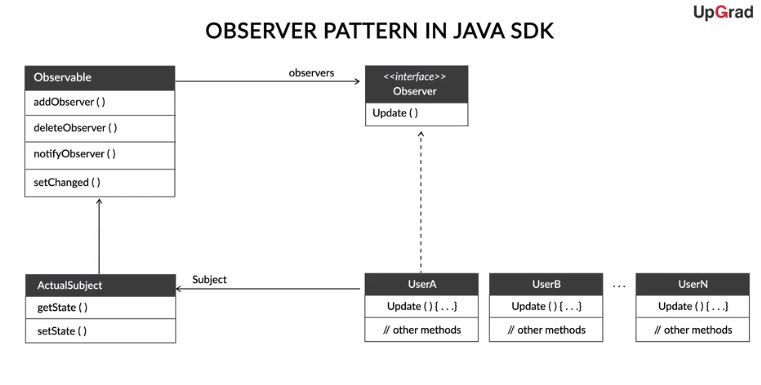
**The subject changes its state, and this change of state is observed by the dependent objects i.e. the observers.**

# Differences between Our Observer Pattern and Java SDK

Let us now look in detail inside the Java SDK Observer pattern and see how does it differ from our implementation of the Observer pattern.



Bottom of Form



Bottom of Form

Thus, we have seen, that we may not always use the default JDK implementation of any design pattern, and must create our own design pattern to suit our needs. We must know that design patterns provide us with an easy way to implement commonly encountered problems in software development, and we can create our design patterns to solve problems specific to our situation

# Summary

In this session, you learnt the basics of design patterns.   
A design pattern provides a general, well-tested & reusable solution to a common design problem. These are peer-reviewed software solutions to common problems and issues we run into in software development. The use of design patterns provides advantages such as:  
1. They provide a way to solve issues related to software development using a proven solution.  
2. They make the overall system easier to understand and maintain

You learnt the three categories of the design patterns as:  
Creational design patterns are those who define HOW classes and objects will be created. Different classes of objects will follow different mechanisms for creating objects.   
Structural design patterns are used to define the relationships between different classes. They are mainly concerned with how the different classes and subclasses are organized amongst each other & how classes and objects are composed to form larger objects and structures.   
Behavioural design patterns are those design patterns which help us define how different classes and objects will communicate with each other. These patterns are concerned with how different responsibilities are shared amongst different classes.

After discussing the categories of the design patterns, we learnt their types through various implementation examples.

The Creational design patterns have various types, and we discussed the following in our session:  
a. Factory Method pattern: It defines an interface for creating an object, but let subclasses decide which class to instantiate. The factory method acts as an interface which deals with the concrete classes without burdening the user of knowing all the underlying functions.

b. Builder Pattern: It enables us to create objects which are “layered” or “complex”, and which cannot be implemented through a simple factory pattern. Builder pattern is generally used in those cases, where creation is a multi-step process, and cannot be done in just a single step.

c. Singleton Pattern: It is used in cases where only ONE class object is used to perform some specific actions in the system. It is used to restrict the instantiation of a class to exactly one object.

After discussing the creational design patterns, you learnt about following Structural design patterns:

a. Adapter pattern: The adapter design pattern is used to link two interfaces or classes which are otherwise incompatible with each other.

b. Decorator pattern: It is used when we need to dynamically change the behaviour of an object at run-time or add some additional behaviour to it.

c. Facade pattern: The facade pattern acts as a simple gateway between the user and a complex set of functions.

d. Proxy pattern: A proxy pattern is a class functioning as a simpler interface to something else. It provides an interface similar to the class for which it is acting as a proxy, but it can also perform some additional functions to control direct access to that class.

Having learnt about the structural design patterns, we discussed the last category of design patterns: Behavioural design patterns and their following types:  
a. Command Design pattern: The Command pattern is used to encapsulate a request as an object, thereby letting you parameterize clients with different requests. In this, a singular object called the “command object” encapsulates all the information which is needed to act at a later time.

b. Iterator pattern: The Iterator pattern is used to provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

c. Observer pattern: The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

d. Strategy pattern: The strategy pattern gives the flexibility to define a family of algorithms, encapsulates each of the algorithms and makes them interchangeable at the runtime.

Then you moved on to the Industry demonstration where you learnt about how design patterns are put to use in actual software development and how you can create your design patterns depending upon the situation.

Q1. **Choose the correct design pattern**

There are many applications on Android mobile devices. You as an android developer wish to provide common configuration settings to these applications. For keeping the device configuration consistent throughout all of the applications, the configuration class should have a single instance that should be globally accessible, so all the required applications can read the global configuration file. Which pattern do you think would be useful in such a scenario wherein I need to create something which must be static and global?

**Singleton**

**Feedback :**

*Here, we need only ONE class object which would perform some specific actions in the system. We want to restrict the instantiation of a class to exactly one object. For this, we can create a single instance of our configurations using singleton pattern, and this single instance of our configurations can never be overwritten, created more than once, or erased. Singleton pattern seems like the perfect choice for this as it is used for exactly this purpose*

**Choose the correct design pattern**

Imagine that you are running a door making broker business. You don't create the doors on your own rather delegate the task of door creation to the right door maker by taking the door requirements from your clients, and, based on the client's needs, pass their requirements to the appropriate door maker to make the requested door. For example, if the requirement calls for a steel door, the final output will be a steel door object. Similarly, if the door calls for a wooden door, you would give the door to the wooden door maker to make the door. After the door maker has created and given you the desired door, you would then give the created door back to your client. The user is not concerned with the door creation mechanism but rather the user merely delegates the system to produce the final door object of the user's liking. WHich design pattern should be used to produce such door objects?

**Factory**

**Feedback :**

*Rather than constructing a door from scratch every time you needed to install a door, you can choose to delegate this work to a “factory”, which can create whichever doors you want for your house, based on your needs. Factory pattern would be useful in this case*

**Choose the correct design pattern**

Imagine an everyday process like shutting down a computer. Programmatically, shutting down a computer involves saving the state of programs, terminating running threads and many other internal operations. Now, if the user were to see the complex steps which go behind something as simple as “shutting down”, he would clearly be taken aback! Thus, in order to make life simple for the user, we can create a simple interface between the computer and the user. This would enable the user to perform the shut down operation in a simple way. Which design pattern do you think would be useful for creating such an interface?

**Facade**

**Feedback :**

*We need to provide a minimalistic and simple interface to a complex body of code. This interface would act as a simple gateway between the user and a complex set of functions. Facade pattern helps us in doing this.*

**Choose the correct design pattern**

Imagine that you have to implement a sorting algorithm for a dataset. In the beginning the dataset was small, thus we were using insertion sort for our sorting. But now, when the dataset started growing, insertion sort became slow and thus we had to shift to quicksort. But we then realized that quicksort is very inefficient for smaller datasets. Hence, we need to carefully choose which sorting algorithm to use when, depending upon the size of the dataset. Which design pattern do you think would be useful in such a case?

**Strategy**

**Feedback :**

*We need to select an appropriate sorting algorithm at runtime. Strategy design pattern helps us to do so. Strategy pattern is useful in scenarios where there might be multiple ways to do a certain thing, but only one of such suitable ways(algorithms) must be selected at any given time.*