From the vault of my engineering newsletter "Systems That Scale"

Saurav Prateek's

Low Level Design Handbook - Part 1

Explaining **Design Patterns** from scratch along with their **Code** implementations

Table of Contents

Chapter 1

Singleton Pattern

We will discuss the Singleton Pattern in-depth along with its Lazy Implementation. We will also discover the issue with Multithreading along with the performance overheads caused by the Synchronized implementation of the design pattern. Finally we will dive into the Eager Implementation of the design pattern and performance optimisations achieved by the same.

Definition

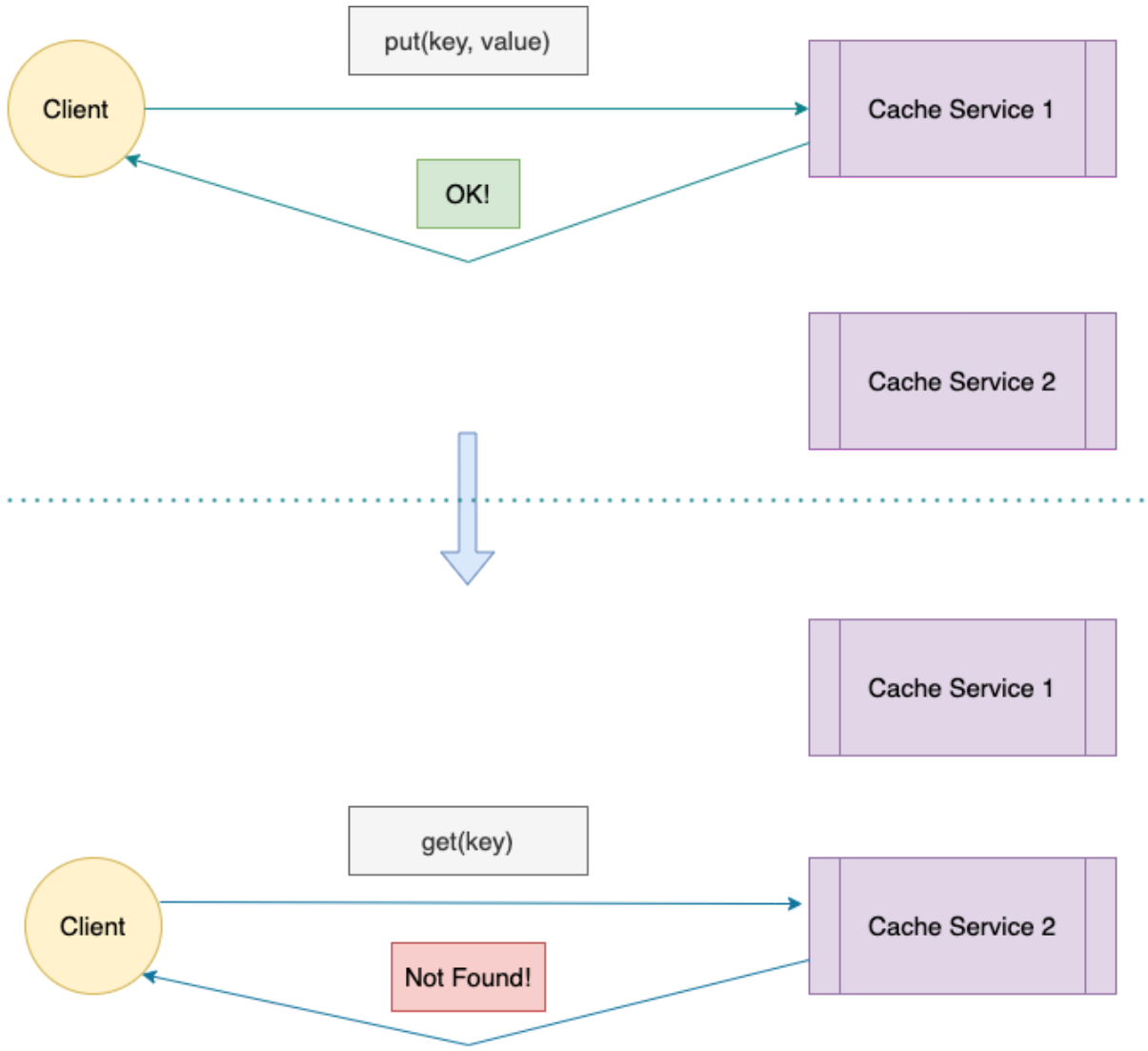
A Singleton Pattern is a creational design pattern that ensures that a class has only one instance and also provides a global access to it.

Introduction

As the definition states, the Singleton Pattern only allows its users to create a single instance of the class. Is it worth it to restrict a class implementation to have a single instance? The singleton pattern has some drawbacks but also has some interesting applications where we need only one single object, for eg. Thread Pools, Caches, Loggers etc.

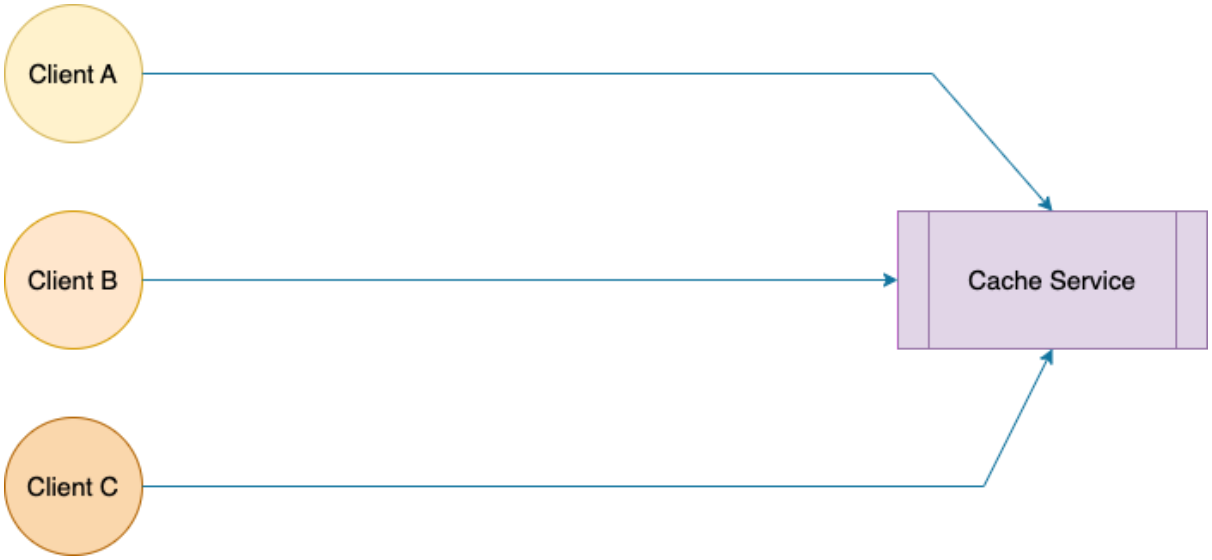
Let’s take the example of a Cache. We want a single cache object to be there throughout the runtime and all the clients should look for the cached data in that single object.

Having a multiple Cache instance can cause many unwanted problems. Suppose you instantiate two cache objects during runtime and a client puts a key in one cache instance and requests for the same key from a different instance, This can cause the issue of data unavailability in your cache.



You saw how the presence of multiple instances of the cache class caused the data unavailability issue in your code.

The Singleton Pattern can be used to deal with such issues by ensuring only one object of a class is created throughout the runtime and every client has a global access to it.



Lazy Initialization

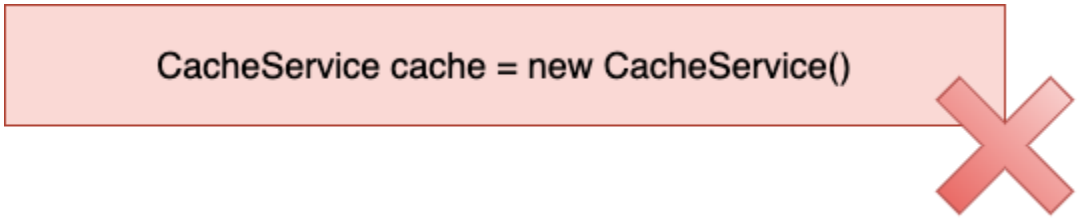
Let's dive into the implementation of the singleton pattern. We will discuss the Lazy Initialization method where the instance of the singleton class is only created when it is accessed for the first time.

It's called a Lazy initialization because we wait for a client to access the class or request for its first instance and only then creates one.

```
// Singleton Class for Cache Service - Lazy Implementation.
public class CacheService {
    private static CacheService service = null;
    private HashMap<Integer, String> cacheMap;

    private CacheService() {
        cacheMap = new HashMap<Integer, String>();
    }

    public static CacheService getCache() {
        if (service == null) {
            service = new CacheService();
        }
```

```
        return service;
    }

    public String getValue(int key) {
        return cacheMap.getOrDefault(key, "");
    }

    public void store(int key, String value) {
        cacheMap.put(key, value);
    }
}
```

In the above implementation we have kept the constructor of the class private so that no one could create the instance of the class from outside.



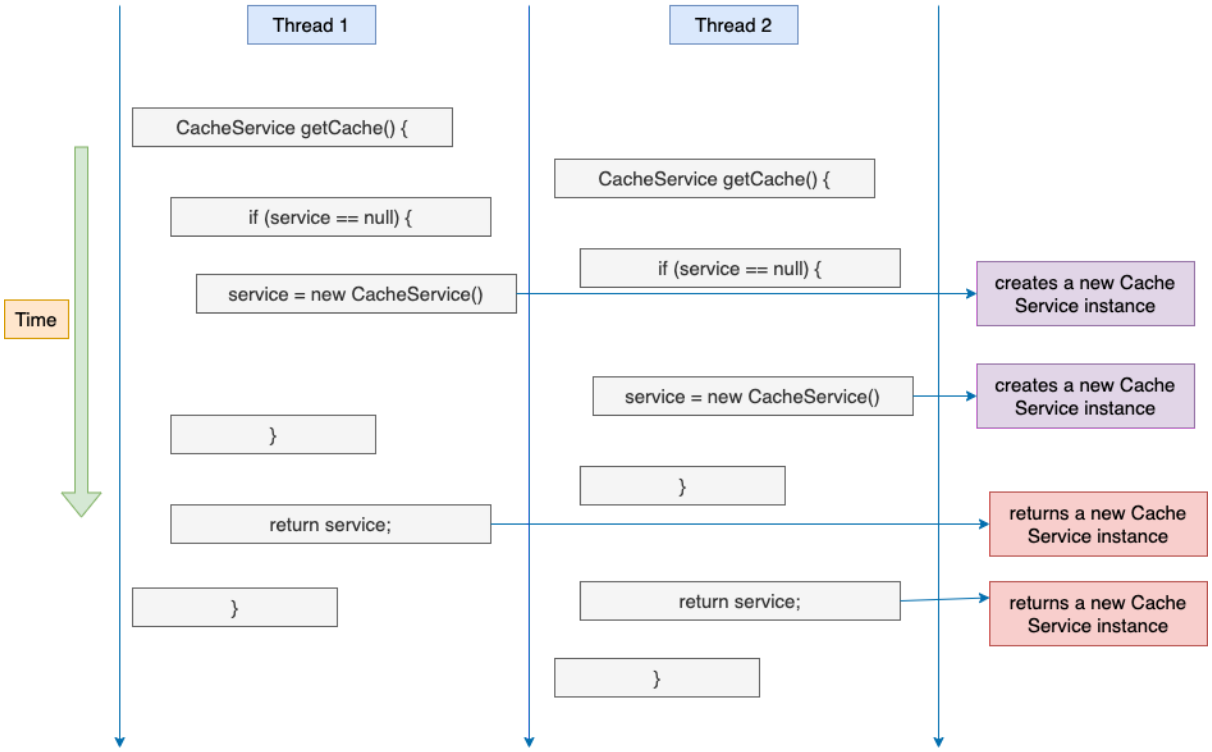
The above statement won't work.
Instead the clients will have to rely on the static method of the class to request for the Cache object.

Now, if a client requests for the object for the first time then a new instance of the class will be created and returned. Any subsequent calls to that static method post that will return the previously created instance.

Dealing with Multiple Threads

The previous implementation we dealt with is not thread safe. If multiple threads enter the getCache static method then more than once cache objects can get created and returned.

This will again start causing data unavailability issues in our code.



There are multiple ways to avoid the above scenario. Let's discuss the solutions.

Synchronized Implementation

We can fix the above issue of multi-threading by making the static getCache() a **synchronized** method.

By adding a synchronized keyword to getCache we force every thread to wait its turn before it can enter the method. This will avoid two threads accessing the static method at the same time.

```
// Singleton Class for Cache Service - Synchronized Implementation.
public class CacheService {
    private static CacheService service = null;
    private HashMap<Integer, String> cacheMap;

    private CacheService() {
        cacheMap = new HashMap<Integer, String>();
    }

    public static synchronized CacheService getCache() {
        if (service == null) {
            service = new CacheService();
        }
        return service;
    }

    public String getValue(int key) {
        return cacheMap.getOrDefault(key, "");
    }

    public void store(int key, String value) {
        cacheMap.put(key, value);
    }
}
```

Although we have solved the issue of Multi-threading, the above implementation can perform poorly when a huge traffic tries to access our cache service.

If we look carefully we only need the synchronized keyword for the first time to ensure multiple instances of the cache service are not created. Once a thread creates the first instance of the cache service, we don't need to synchronize anymore. Once the first instance is created it's perfectly fine for multiple threads to

access the static getCache() method at the same time. Since every time the already created instance will be blindly returned.

Hence the synchronized implementation will only create performance overheads after the first call to the static method.

Eager Initialization

In the previous section we saw the performance overhead caused by the synchronized method in multithreading. One way to improve the performance is to eagerly create the instance of our Cache service when the class is loaded.

```
// Singleton Class for Cache Service - Eager Implementation.
public class CacheService {
    private static CacheService service = new CacheService();
    private HashMap<Integer, String> cacheMap;

    private CacheService() {
        cacheMap = new HashMap<Integer, String>();
    }

    public static CacheService getCache() {
        return service;
    }

    public String getValue(int key) {
        return cacheMap.getOrDefault(key, "");
    }

    public void store(int key, String value) {
        cacheMap.put(key, value);
    }
}
```

It is guaranteed that the unique instance of the Cache service is created before any thread accesses the static getCache() method.

Summary

We discussed the **Singleton Pattern** in-depth along with its **Lazy Implementation**. We also discovered the issue with **Multithreading** along with the performance overheads caused by the **Synchronized** implementation of the design pattern. We also discussed the **Eager** Implementation of the design pattern and performance optimisations achieved by the same.

Code Implementation from "**Saurav's LLD Template**"

1. Lazy Implementation [View Sample Code]
2. Synchronized Implementation [View Sample Code]
3. Eager Implementation [View Sample Code]

Chapter 2

Decorator Pattern

We will discuss the Burger King problem which can be solved through the Decorator Pattern. We will also discuss the Initial Implementation to solve the problem which leads to Class Explosion. Further we will dive into yet another alternate implementation which violates the Open Closed design principle and finally discuss the optimal solution through Decorator Pattern in depth.

Definition

The **Decorator Pattern** attaches additional responsibilities to an object dynamically at the runtime. It provides a flexible alternative to subclassing for extending functionality.

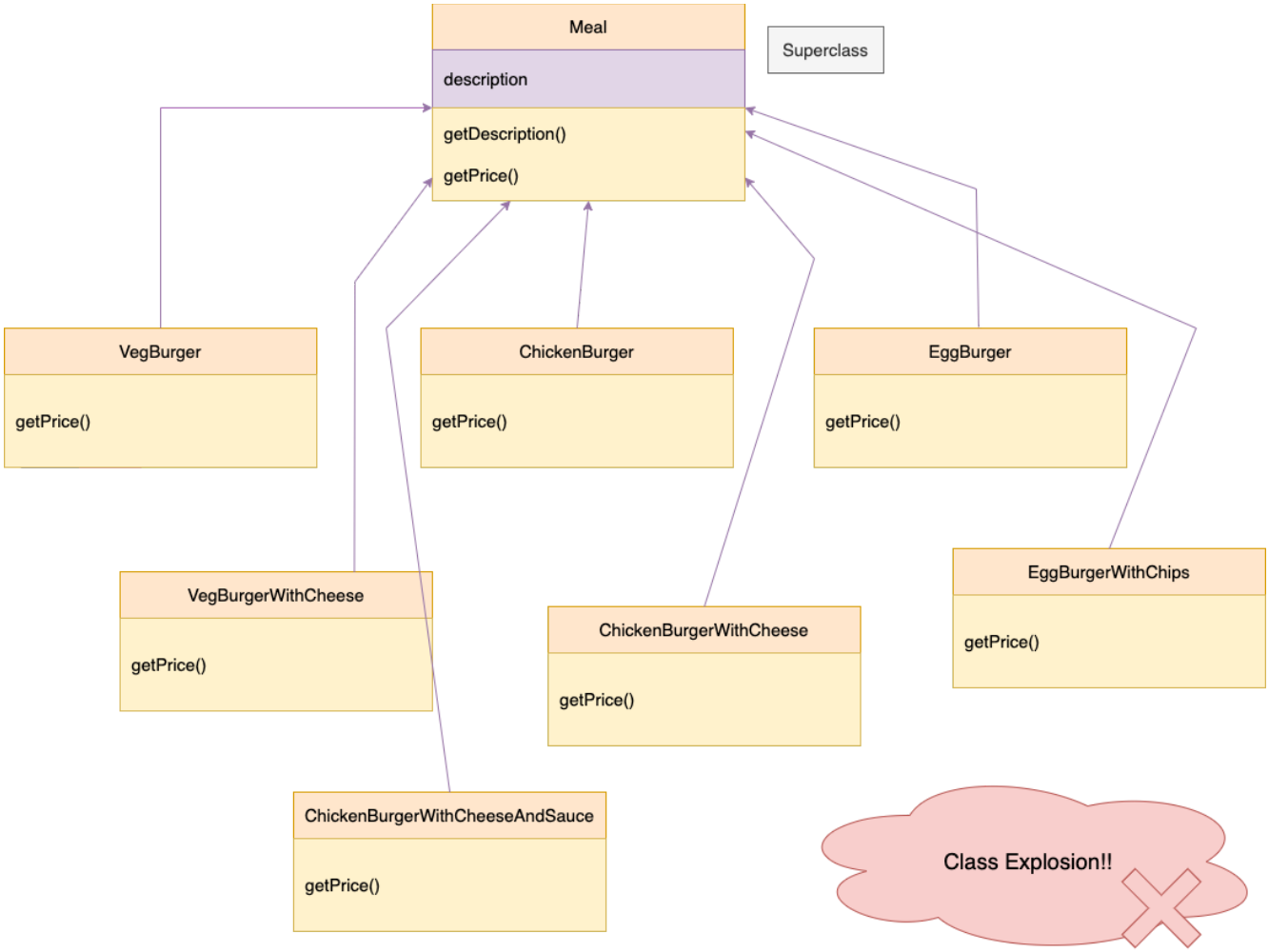
Introduction

As the definition above states that the **Decorator Pattern** adds additional features/responsibilities on top of your base class at the runtime. We don’t have to perform any changes in the source code to add any responsibilities, everything happens completely dynamically at the run-time.

Before moving on to the Decorator Pattern, let's first understand why we need such a pattern? What are the situations in which this pattern should be preferred?

Let’s understand this with the example of **Burger King** which has recently introduced a variety of burgers to choose from. Currently they have three base burgers i.e. **Veg**, **Chicken** and **Egg** along with multiple add-ons like **Cheese**, **Chips**, **Veggies** and **Sauce**. As a customer now I can combine my base burger with these add-ons to create my own burger meal preparation. For example: I can have a “Chicken burger with cheese and sauce”.

Initially the class design looks like this.



In the above design we have an abstract class named **Meal** which is subclassed by all the different Burger classes. Now, every Burger class has their own **getPrice()** method because every base burger and add-ons have different prices.

The superclass **Meal** looks like this.

```
public abstract class Meal {
    String description;

    public String getDescription() {
        return description;
    }
```

```
    public abstract double getPrice();
}
```

One of our burger class **ChickenBurgerWithCheeseAndSauce** will look like this.

```
public class ChickenBurgerWithCheeseAndSauce extends Meal {
    public ChickenBurgerWithCheeseAndSauce() {
        description = "Chicken Burger with Cheese and Sauce";
    }

    public double getPrice() {
        // Price of:
        //   1. Chicken Burger: Rs. 70
        //   2. Ceese: Rs. 20
        //   3. Sauce: Rs. 10
        return (70.0 + 20.0 + 10.0);
    }
}
```

In this way every preparation of a burger meal will have its own cost depending upon the base burger and the combination of add-ons.

The above design will lead to a **Class Explosion**. As there can be a lot of different combinations of Burger meals and hence a huge number of classes for every meal. The engineers at Burger King will have to maintain a huge number of different classes for different burger meals and might also have to add a lot more classes even if one more add-on is introduced to our meal catalog.

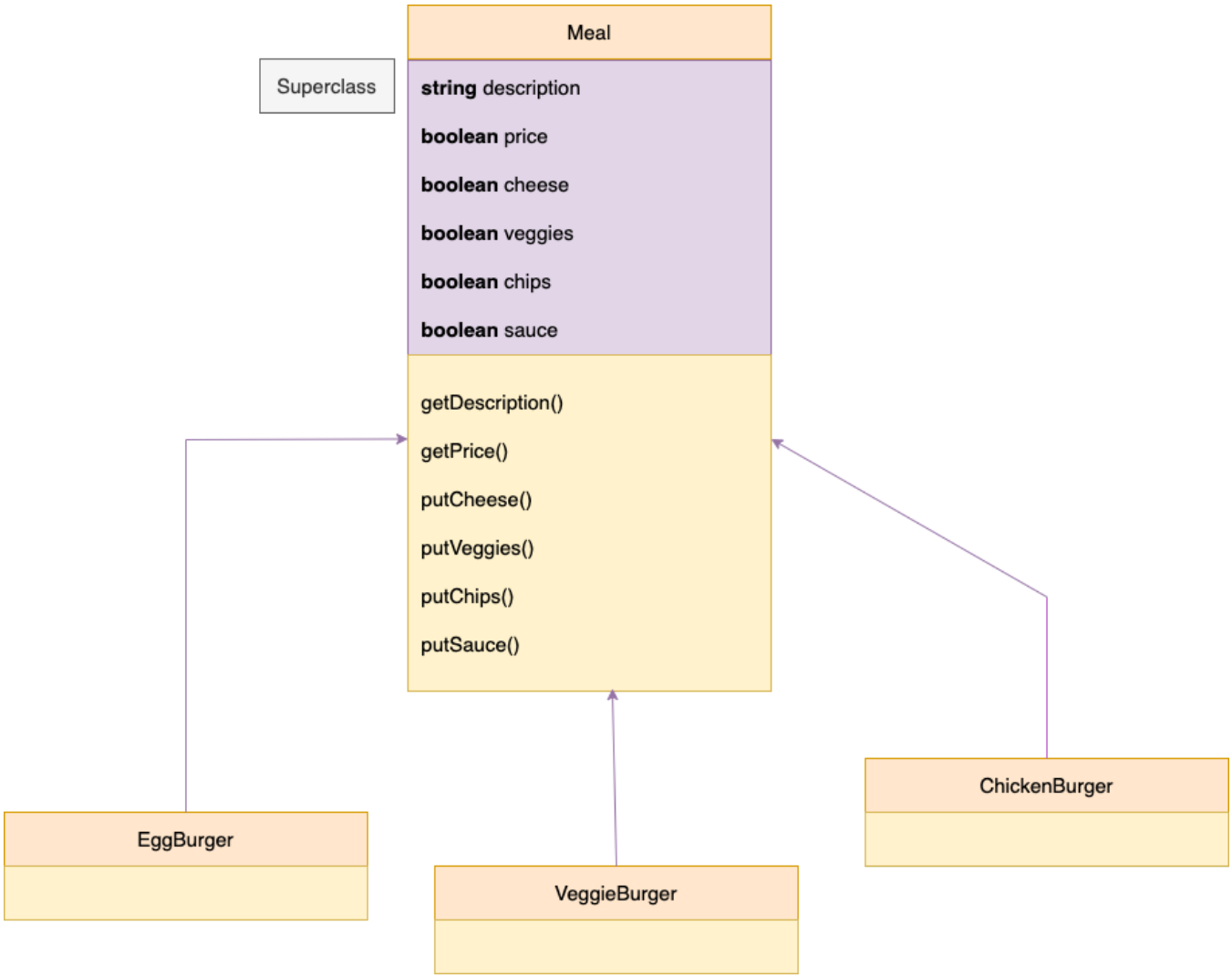
This is very complex and almost impossible to maintain or extend.

Alternate Approach

Let's discuss yet another alternative approach which can get us rid of the Class Explosion issue faced in our previous design.

In this design, instead of referring to every burger meal preparation as a separate class we can add a **boolean** flag for every add-ons in our super class and set them true/false depending upon whether that particular add-on is included in our burger meal or not.

Our class design will look like this.



The class implementation of the **Meal** superclass looks like this.

```
public class Meal {
    String description;
    double price;

    private boolean cheese;
    private boolean veggies;
    private boolean chips;
    private boolean sauce;

    public String getDescription() {
        if (cheese) {
            description += ", Cheese";
        }
        if (veggies) {
            description += ", Veggies";
        }
        if (chips) {
            description += ", Chips";
        }
        if (sauce) {
            description += ", Sauce";
        }
        return description;
    }

    public double getPrice() {
        if (cheese) {
            price += 20.0;
        }
        if (veggies) {
            price += 15.0;
        }
        if (chips) {
            price += 10.0;
        }
        if (sauce) {
            price += 8.0;
        }
        return price;
    }
```

```
    public void putCheese() {
        cheese = true;
    }

    public void putVeggies() {
        veggies = true;
    }

    public void putChips() {
        chips = true;
    }

    public void putSauce() {
        sauce = true;
    }
}
```

The base burger class **VegBurger** will inherit from the superclass **Meal** and will look like this.

```
public class VegBurger extends Meal {
    VegBurger() {
        description = "Veg Burger";
        price = 60.0;
    }
}
```

If a user wants to order a “Chicken burger with cheese and sauce”, it can be done this way.

```
public class BurgerKing {
    public static void main(String[] args) {
        // Order a Chicken Burger with Cheese and Sauce
        Meal burgerMeal = new ChickenBurger();
        burgerMeal.putCheese();
        burgerMeal.putSauce();

        System.out.println("Order placed: " + burgerMeal.getDescription());
```

```
        System.out.println("Cost: " + burgerMeal.getPrice());
    }
}
```

```
Order placed: Chicken Burger, Cheese, Sauce
Cost: 98.0
```

The above design avoids **Class Explosion**. We no longer have to maintain classes for every single burger meal preparation. But this design violates the **Open-Closed Principle**. The principle states that a class should be open for extension, but closed for modification. But in our design even if we have to add an extra Add-on, say "**Mint Sauce**", we have to change the code in our **Meal** base class.

Any new add-on will force us to add new methods and alter the existing **getDescription()** and **getPrice()** methods in the superclass (**Meal**).

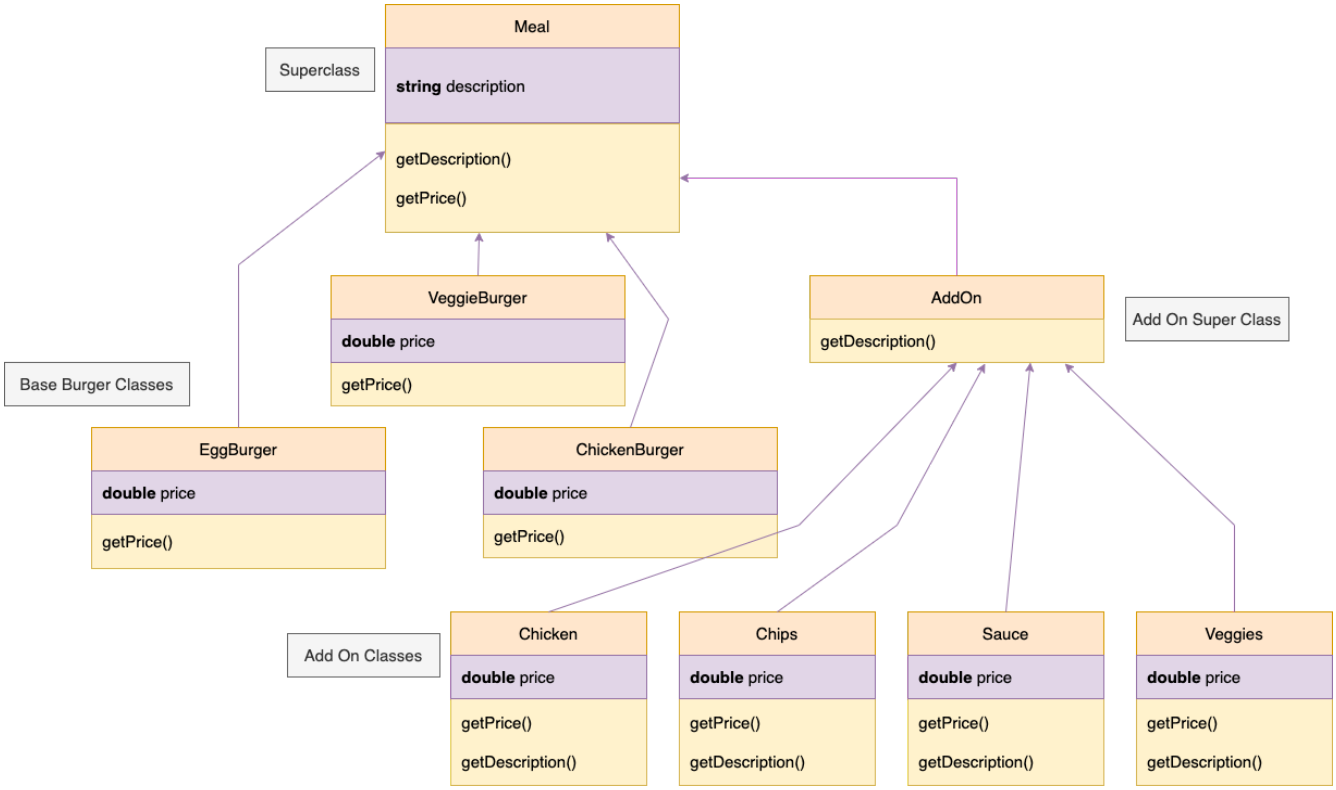
Decorator Pattern

We discussed the previous implementations and each of these had some potential design flaws. **Decorator Pattern** allows us to add requirements to the base class dynamically at the runtime.

In our example of **Burger King** we can dynamically put add-ons over the base burger the way we like. For this we neither have to deal with the class explosion nor have to change the class implementation in case of any modification/addition.

The Decorator design pattern allows the current design to be extended to add more add-ons if required. It allows the add-on classes to wrap over the base class and **recursively** calculate the price or description of the meal preparation.

The class design looks like this.



Our **Meal** superclass looks like this.

```
public abstract class Meal {
    String description;

    public String getDescription() {
        return description;
    }

    public abstract double getPrice();
}
```

Our base burger **ChickenBurger** class extends the **Meal** class and looks like this.

```
public abstract class AddOn extends Meal {
    public abstract String getDescription();
}
```

An add-on **Chips** class looks like this.

```
public class Chips extends AddOn {
    private double PRICE = 10.0;
    private Meal meal;

    public Chips(Meal meal) {
        this.meal = meal;
    }

    public String getDescription() {
        return meal.getDescription() + ", Chips";
    }

    public double getPrice() {
        return PRICE + meal.getPrice();
    }
}
```

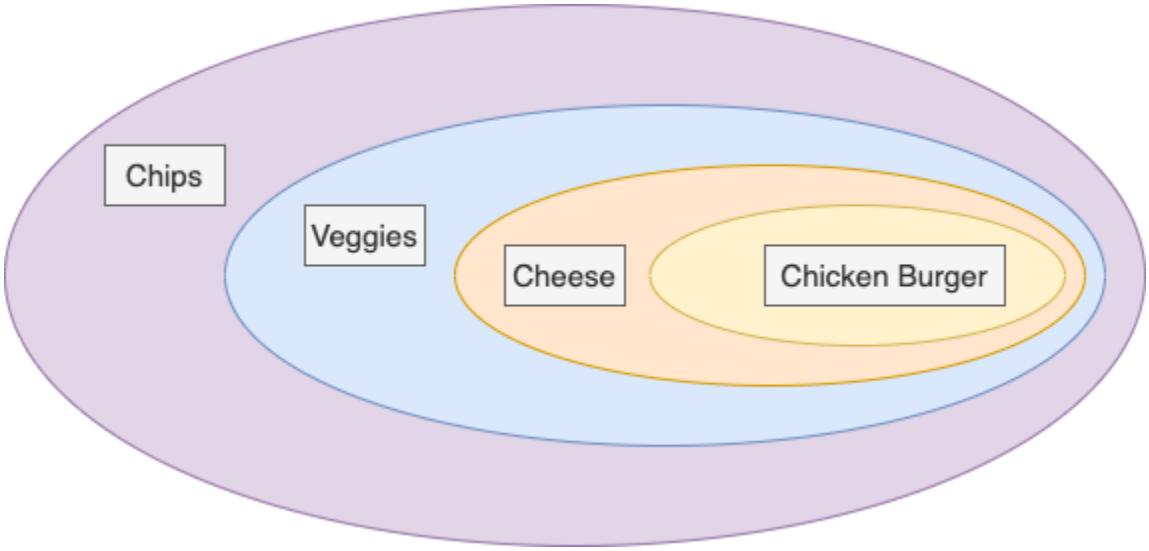
In the above class design we can see that the add-on **Cheese** class wraps a **Meal** class. In this way we can wrap add-ons over a prepared meal. Hence if a user wants a “Chicken burger with cheese, veggies and chips” then they can add them dynamically at runtime in this way.

```
public class BurgerKing {
    public static void main(String[] args) {
        // Order: Chicken Burger with Cheese, Veggies and Chips
        Meal burgerMeal = new ChickenBurger();
        burgerMeal = new Cheese(burgerMeal);
        burgerMeal = new Veggies(burgerMeal);
        burgerMeal = new Chips(burgerMeal);

        System.out.println("Order placed: " + burgerMeal.getDescription());
        System.out.println("Price: " + burgerMeal.getPrice());
    }
}
```

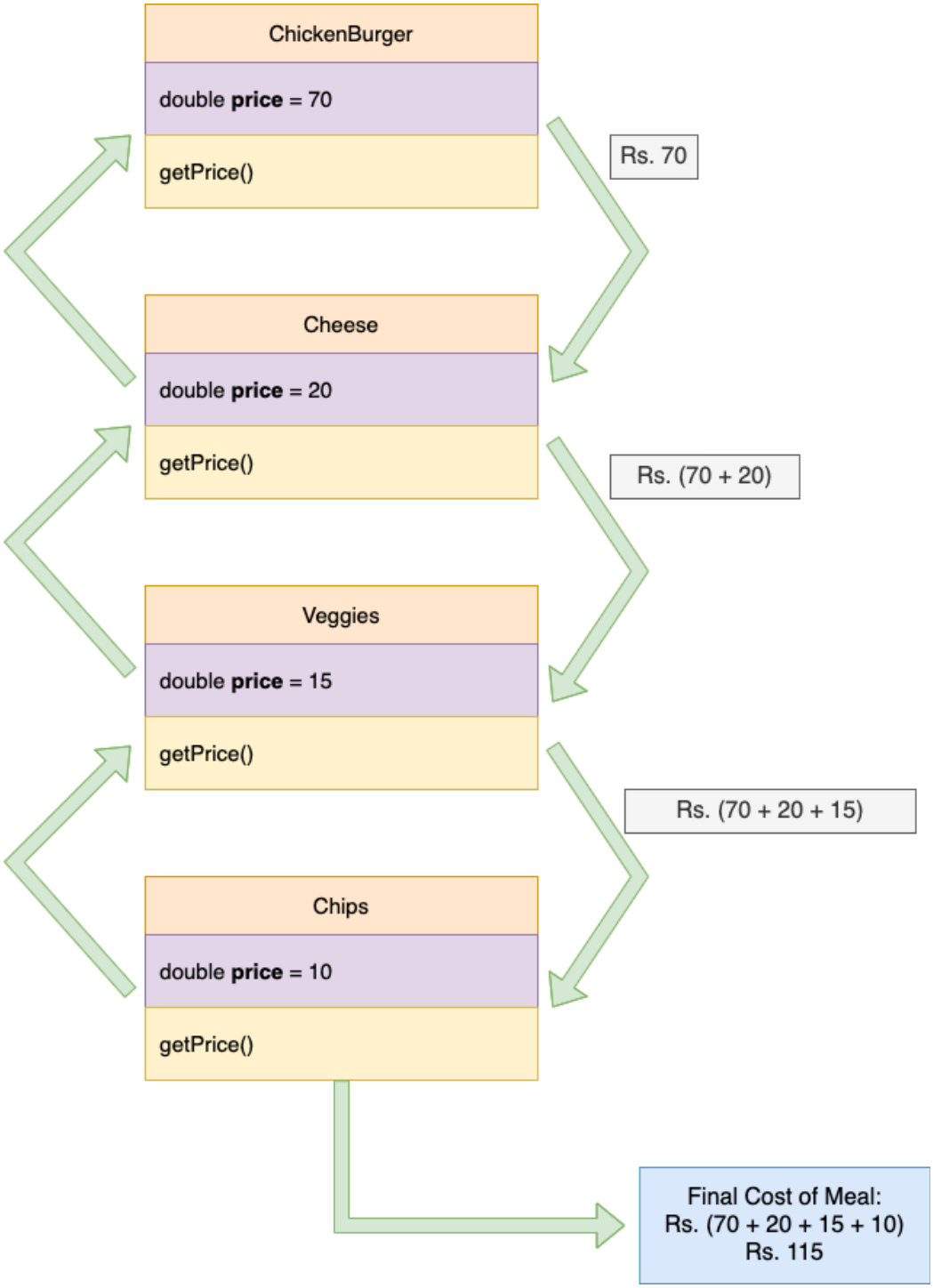
```
Order placed: Chicken Burger, Cheese, Veggies, Chips
Price: 115.0
```

As we wrap the add-ons over the prepared meal, the final meal gets its price and description by recursively calling all the wrapped meals.

The wrapped class design looks like this.



The process of computation of **Price** recursively for the order “Chicken burger with cheese, veggies and chips” looks like this.



The similar method applies for computing the **Description** of a prepared meal.

Summary

We discussed the **Decorator Pattern** in detail along with the initial implementation which caused **Class Explosion**. We also discussed yet another alternative implementation that violated the **Open Closed** design principle. We finally solved the Burger King design problem effectively with the decorator pattern.

Code Implementation from "**Saurav's LLD Template**"

1. Class Explosion Implementation [View Sample Code]
2. Alternate Implementation [View Sample Code]
3. Effective Implementation [View Sample Code]

Chapter 3

Factory Pattern

We will discuss the initial Inefficient Implementation of concrete Object creation. We will also look around the drawbacks of this implementation. We will finally discuss the Simple Factory implementation where we decoupled the concrete object creation logic and at the end we discuss the Factory Method design pattern where we define an interface to create an object but allow their subclasses to create concrete products.

Definition

The **Factory Method Pattern** defines an interface for creating an object, but lets the subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Introduction

There can be many scenarios where you might have to instantiate a group of concrete classes on the basis of a logic. The most basic way to instantiate a concrete class is through a **new** keyword. But many times creating the concrete classes directly through the “new” keyword in your code might cause multiple issues.

We will understand how we can use **Factory Method Pattern** to separate out the creation logic of the concrete classes away from your main code. This will allow us to use the encapsulated creation logic at multiple places. In this design pattern we will code against interfaces which will help us insulate from a lot of changes down the lane.

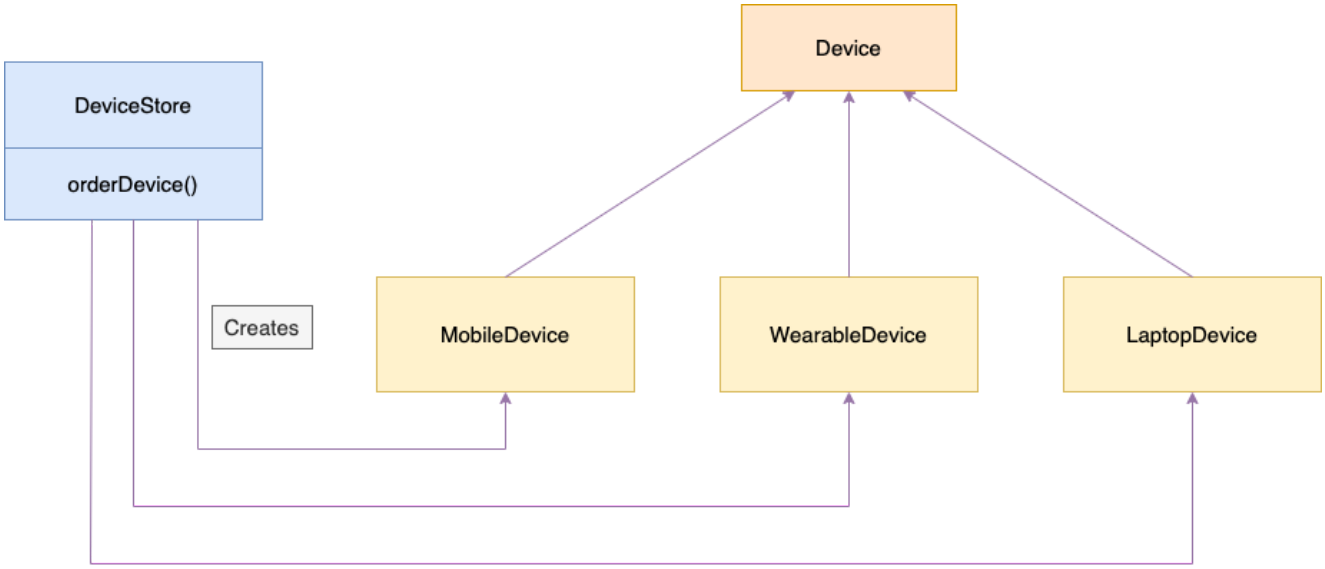
Let’s take an initial example of a **Device Store** which sells multiple types of electronic devices such as **Mobile**, **Wearables** and **Laptops**. Now a customer can order any one of the devices by providing an appropriate **Device Type** to the device store.

Initially the implementation of Device Store is messy which can cause a lot of problems.

```
public class DeviceStore {
    enum DeviceType {
        MOBILE,
        WEARABLE,
        LAPTOP
    }
```

```
    Device orderDevice(DeviceType deviceType) {
        Device device;

        switch (deviceType) {
            case MOBILE:
                device = new MobileDevice();
                prepareDevice(device);
                return device;
            case WEARABLE:
                device = new WearableDevice();
                prepareDevice(device);
                return device;
            case LAPTOP:
                device = new LaptopDevice();
                prepareDevice(device);
                return device;
        }

        throw new AssertionError("Invalid or Unknown Device type");
    }

    private void prepareDevice(Device device) {
        device.viewConfiguration();
        device.box();
        device.ship();
    }
}
```

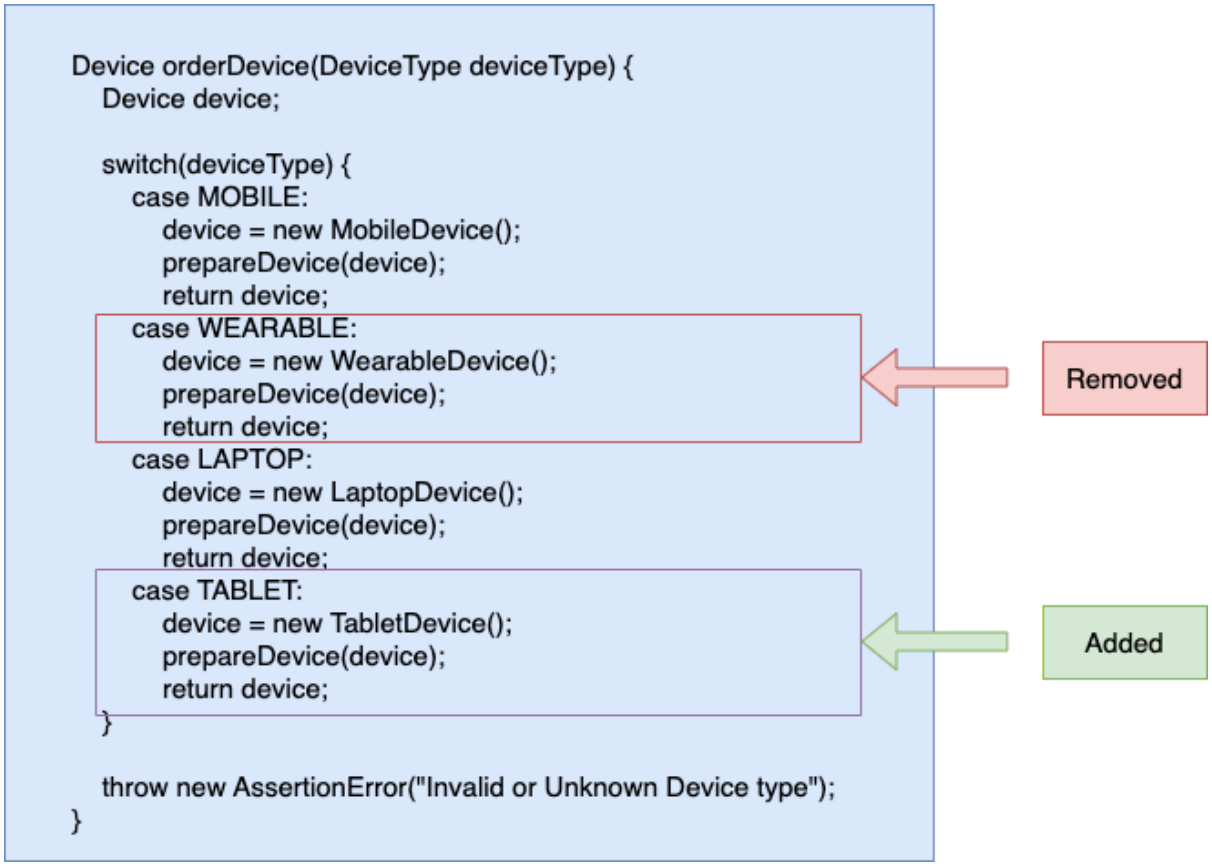
In our previous implementation we can observe that our **DeviceStore** is dependent upon all the concrete devices like **MobileDevice**, **WearableDevice** and **LaptopDevice**.



But if we think about the future aspects, we don't want our **DeviceStore** to be strongly coupled with all the concrete Devices. Device store does not need to know the creation process of all the available concrete devices.

1. We want to add a new Device type called "**TabletDevice**" to our device catalog. To support this we have to change our **DeviceStore** class code and add the TabletDevice creation logic there.
   Hence, everytime a new Device type is added, we have to change our DeviceStore class implementation.
   This makes our DeviceStore class open for modification and hence violates the design principle.

2. The sale of wearable devices is declining and as a result we want to remove **WearableDevice** from our device catalog. Even to support this we have to change our **DeviceStore** class implementation.



Simple Factory

In our previous implementation, we saw how the **DeviceStore** class was strongly coupled with the concrete Device classes (**MobileDevice**, **WearableDevice** and **LaptopDevice**). We also discussed the potential issues due to the strong coupling.

In this section we will use a Simple Device Factory to create the concrete Devices. The idea is to move the device creation logic away from the **DeviceStore** class and put them in a separate **SimpleDeviceFactory** class.

```
public class SimpleDeviceFactory {
    enum DeviceType {
        MOBILE,
        WEARABLE,
        LAPTOP
```

```
    }

    public Device createDevice(DeviceType deviceType) {
        Device device = null;

        switch (deviceType) {
            case MOBILE:
                device = new MobileDevice();
                break;
            case WEARABLE:
                device = new WearableDevice();
                break;
            case LAPTOP:
                device = new LaptopDevice();
                break;
            default:
                break;
        }

        return device;
    }
}
```

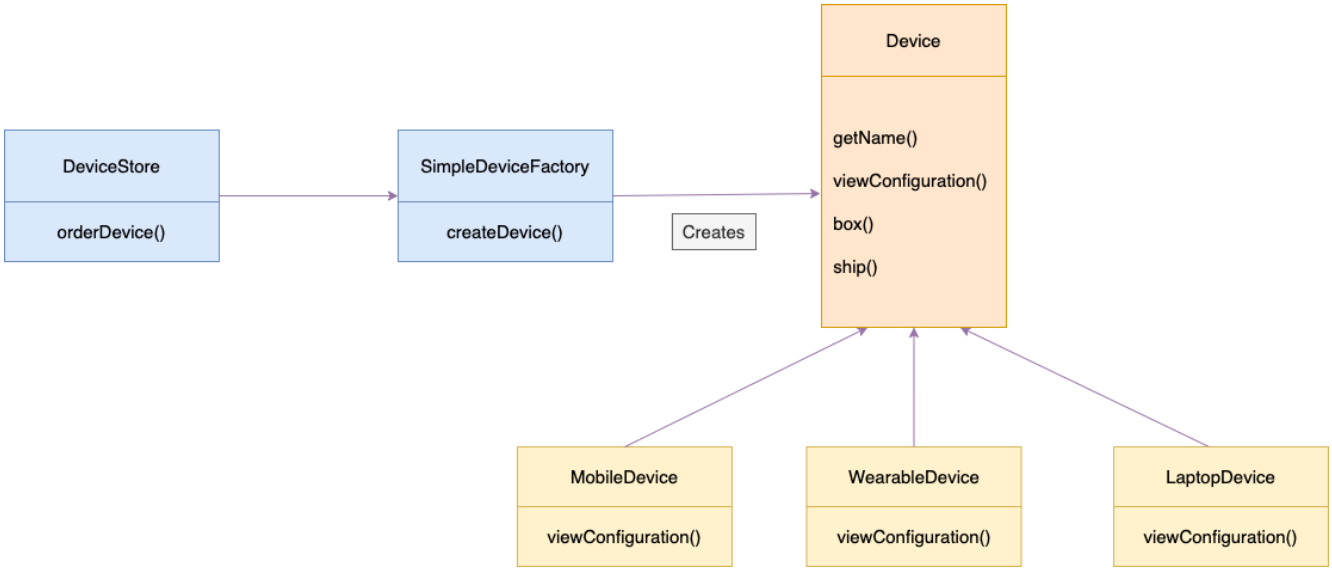
In the above implementation we have isolated the creation logic of concrete devices in the **SimpleDeviceFactory** class. Now we will use this class in our **DeviceStore** to order devices.

```
public class DeviceStore {
    SimpleDeviceFactory factory;

    public DeviceStore(SimpleDeviceFactory factory) {
        this.factory = factory;
    }

    Device orderDevice(DeviceType deviceType) {
        Device device = this.factory.createDevice(deviceType);

        device.viewConfiguration();
        device.box();
```

```
        device.ship();

        return device;
    }
}
```

We can observe that now our **DeviceStore** class is independent of the concrete Devices. It only depends on the **Device** interface. Any modification in our Device catalog will no longer require the DeviceStore class to be modified.

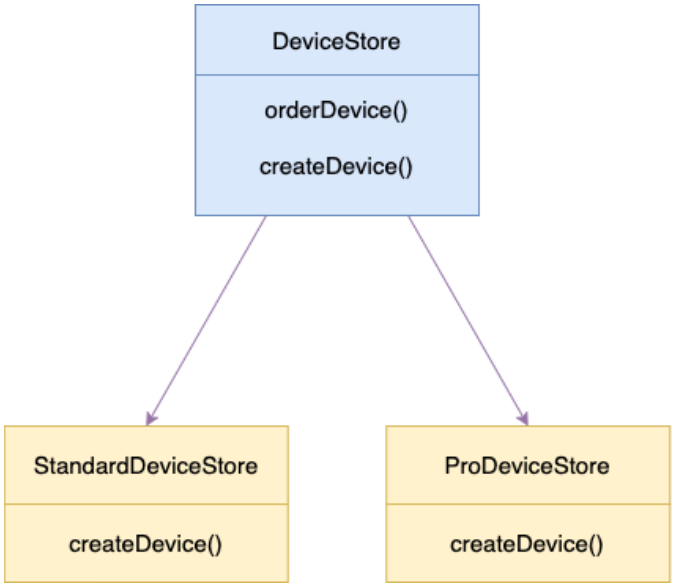
Also if we want to use the device creation logic at other places then we can simply re-use the **SimpleDeviceFactory** class without introducing any code-duplication or additional dependencies.



Factory Method Pattern

In our previous section we used Simple Factory to isolate the creation of concrete Devices. Now our Device Store wants to introduce two different variants for each Device type i.e. **Standard** and **Pro**.

In order to support this our abstract **DeviceStore** class will have a **StandardDeviceStore** and **ProDeviceStore** as the subclasses. We will also move our **createDevice** method that will take care of the concrete Device creation process.



Our abstract **DeviceStore** class will look like this.

```
public abstract class DeviceStore {
    enum DeviceType {
        MOBILE,
        WEARABLE,
        LAPTOP
    }

    public Device orderDevice(DeviceType deviceType) {
        Device device = createDevice(deviceType);

        device.viewConfiguration();
        device.box();
        device.ship();
```

```
        return device;
    }

    protected abstract Device createDevice(DeviceType deviceType);
}
```

The subclasses **StandardDeviceStore** and **ProDeviceStore** will inherit the **createDevice()** method and are responsible for creating concrete devices. Standard Device Store is responsible for creating Standard devices while Pro Device Store is responsible for creating Pro Devices.

One of our subclass **ProDeviceStore** looks like this.

```
public class ProDeviceStore extends DeviceStore {

    public Device createDevice(DeviceType deviceType) {
        switch (deviceType) {
            case MOBILE:
                return new ProMobileDevice();
            case WEARABLE:
                return new ProWearableDevice();
            case LAPTOP:
                return new ProLaptopDevice();
        }

        throw new AssertionError("Invalid or Unknown Device type");
    }
}
```

We can observe that the **ProDeviceStore** creates the Pro Device concrete objects i.e. **ProMobileDevice**, **ProWearableDevice** and **ProLaptopDevice**.

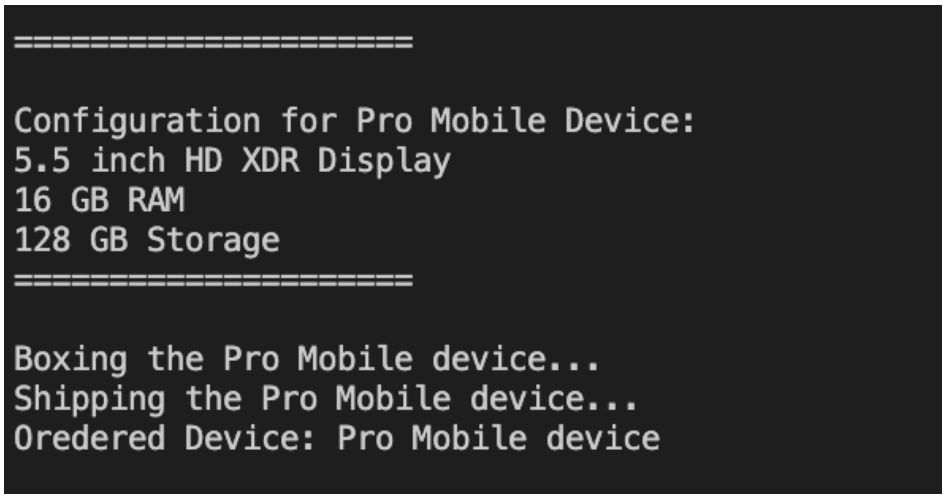
Our Device catalog looks somewhat like this.

All the six concrete device classes inherit the **Device** abstract class.

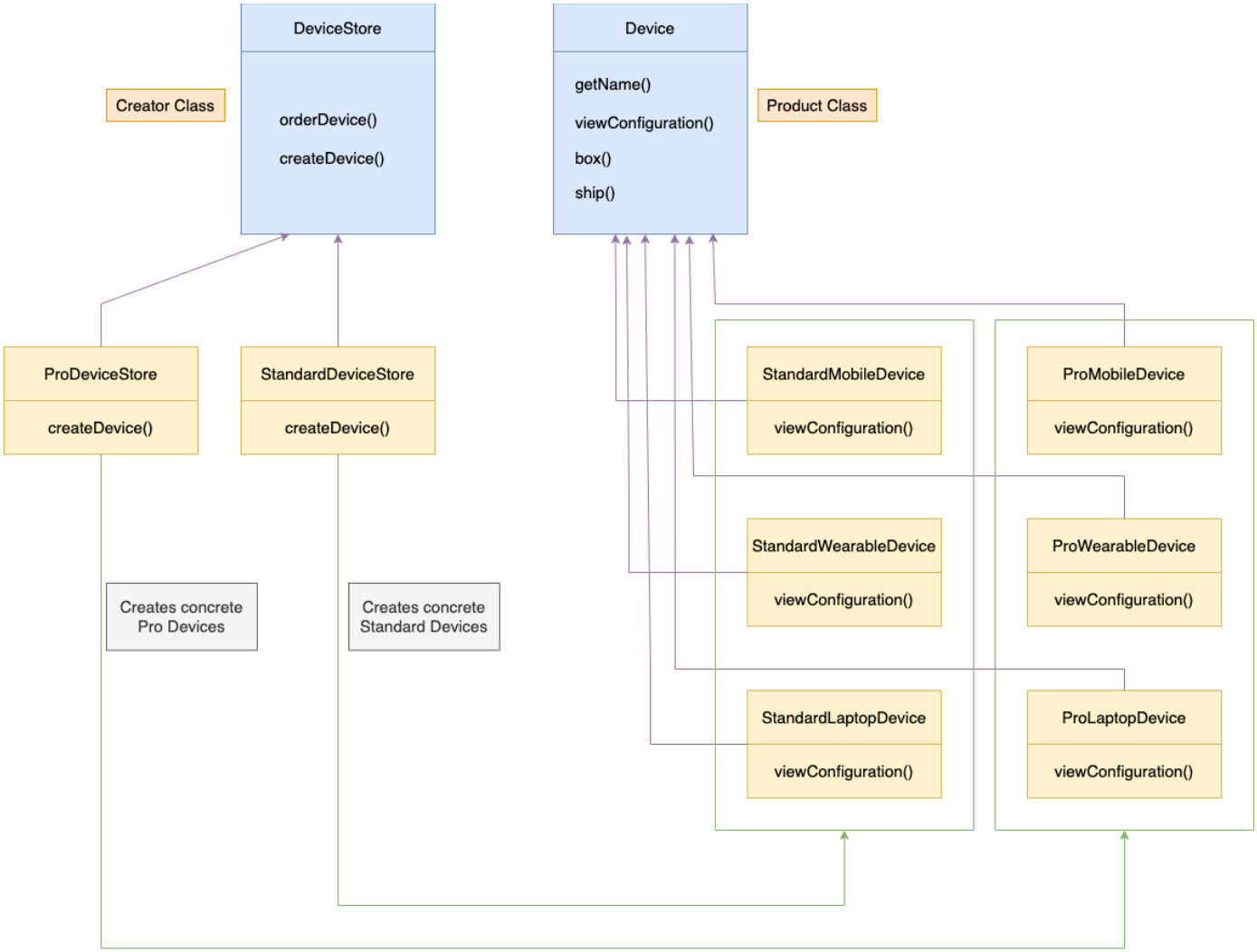
Let's understand how we can now order a Pro Mobile Device from the **DeviceStore**.

```
// Ordering a Pro Mobile Device
ProDeviceStore proDeviceStore = new ProDeviceStore();
Device proMobileDevice = proDeviceStore.orderDevice(DeviceType.MOBILE);
System.out.println("Ordered Device: " + proMobileDevice.getName());
```

The above implementation orders a Pro Mobile Device and we get the following output.



Here we defined **DeviceStore** as an interface to create an object, but allowed its subclasses i.e. **StandardDeviceStore** to create Standard Devices and **ProDeviceStore** to create Pro Devices. This is the definition of **Factory Method Pattern** that we discussed at the start.


DeviceStore
orderDevice()
createDevice()
Creator Class
Device
getName()
viewConfiguration()
box()
ship()
Product Class
ProDeviceStore
createDevice()
StandardDeviceStore
createDevice()
StandardMobileDevice
viewConfiguration()
ProMobileDevice
viewConfiguration()
StandardWearableDevice
viewConfiguration()
ProWearableDevice
viewConfiguration()
StandardLaptopDevice
viewConfiguration()
ProLaptopDevice
viewConfiguration()
Creates concrete Pro Devices
Creates concrete Standard Devices

Summary

We discussed the **Factory Method Pattern** in detail. We looked around an inefficient implementation to create a family of concrete objects. We also understood the **Simple Factory Pattern** and how it isolates the object creation logic. Finally we discussed the **Factory Method Pattern** in depth around how a family of creator classes creates a family of concrete products.

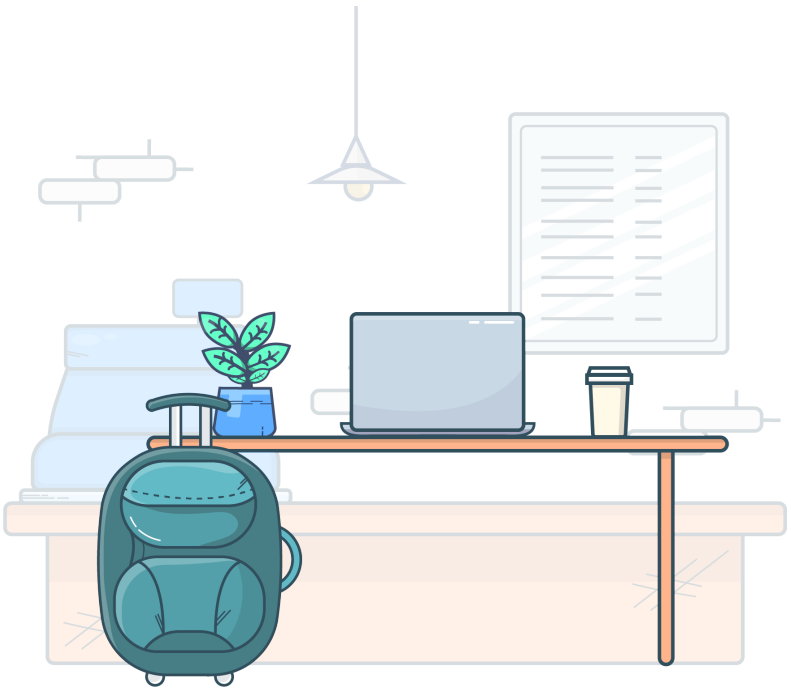
Code Implementation from "**Saurav's LLD Template**"

1. Inefficient Implementation [View Sample Code]
2. Simple Factory Implementation [View Sample Code]
3. Factory Method Implementation [View Sample Code]

Chapter 4

Command Pattern

We leverage the Command Pattern to design a Scheduler that schedules commands which can be executed at a later point of time. We also discuss how Command Pattern decouples the Command/Request execution logic away from the Invoker/Executer. We finally discuss the concept of Macro Command which can be used to execute multiple concrete commands at once.



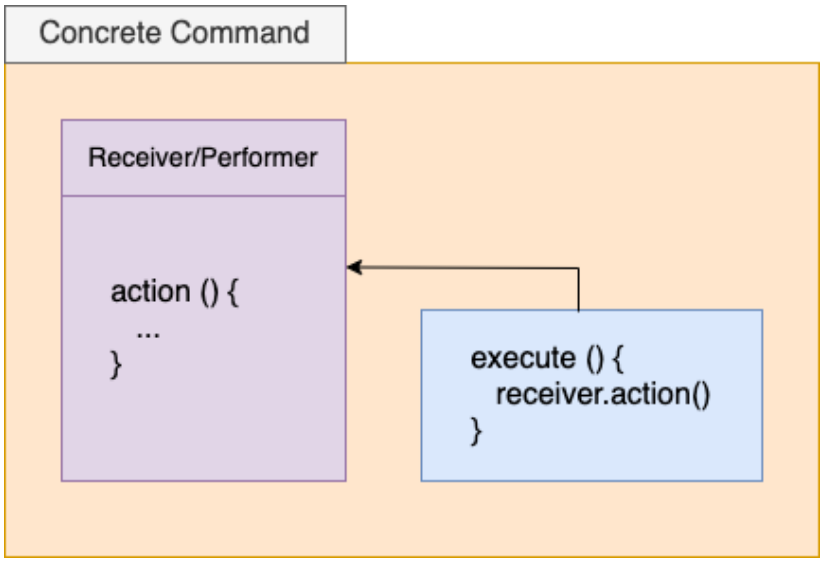
Definition

The **Command Pattern** encapsulates a request as an object, thereby letting us parameterise other objects with different requests, queue or log requests.

Introduction

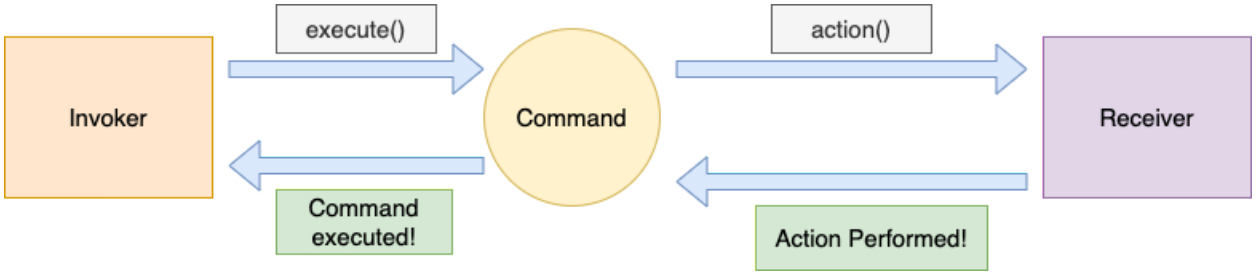
As the definition suggests, the Command Pattern encapsulates a request into an object by binding together an **Action/Command** with the **Performer/Receiver**.

The Command Pattern encapsulates the entire business logic of command execution into these request objects also known as **Concrete Command** objects. For an **Invoker** (an entity executing these concrete commands), they have no idea on how these commands are being executed or who (receiver) is actually executing the commands. The design pattern achieves this by making the concrete command objects to provide a **public** method (**execute()**) through which an invoker can execute them.



From the figure above, we can understand that the Concrete Command object encapsulates the Receiver and provides a **public execute()** method that invokes the Receiver to perform a desired action.

If we look from the **Invoker's** point of view, then we can assume that an Invoker does not need to know the internals of a request/command it has to execute. They can simply hit the publically available **execute()** method of the command and let the command be executed.



Using **Command Pattern** can actually allow us to decouple the command execution logic from its invoker. As long as a command has a publicly executable method, the invoker can execute it anytime.

This behavior makes Command Pattern a right fit in scheduling multiple commands or queuing requests and logging requests to recover from failure.

Using Command Pattern to Schedule requests

In our previous section we saw how commands give us a way to package a piece of computation and pass it around as a first-class object. Sometimes these requests might get executed at a later point of time from the moment they got created by a client.

In this section we will design a **Scheduler** that will have a queue to schedule multiple incoming requests. Before designing a scheduler. Let's first understand what a request/command looks like.

Every **Concrete Command** will implement a **Command** interface. The Command interface will look somewhat like this.

```
public interface Command {
    public void execute();
}
```

Every concrete command will override the execute method to perform the computation.

We will use our scheduler to schedule three requests.

1. **Request 1**: Collect email-ids of users who are subscribed to a newsletter.
2. **Request 2**: Send them a welcome mail.
3. **Request 3**: Send an Acknowledged message to the newsletter author.

All the above three requests/commands can be treated as a Concrete Command and will implement Command interface.

Let's look at the first request.

```
public class CollectSubscriberEmails implements Command {
    private UserService userService;

    public CollectSubscriberEmails(UserService userService) {
        this.userService = userService;
    }

    @Override
    public void execute() {
        userService.readUsersFromDb();
        userService.filterSubscribers();
        userService.snedSubscribersDataToMailService();
    }
}
```

The above concrete command binds a **UserService** which is a **Receiver** in this case and actually performs all the above mentioned activities. Let's look at the **UserService** class.

```
public class UserService {
    public void readUsersFromDb() {
        System.out.println("Collect user rows from "
            + "the User table in database");
    }

    public void filterSubscribers() {
        System.out.println("Filter the unsubscribed "
            + "users from the collected user rows");
    }

    public void snedSubscribersDataToMailService() {
        System.out.println("Send filtered subscriber details "
            + "to the mail service");
    }
}
```

The above explained **UserService** read user details from the database, filter those details to remove unsubscribed users and finally send the subscribed user details to their Mailing service.

The second request actually sends a welcome email to the subscribers. The request looks like this.

```
public class SendWelcomeMail implements Command {
    private MailService mailService;

    public SendWelcomeMail(MailService mailService) {
        this.mailService = mailService;
    }
```

```
    @Override
    public void execute() {
        mailService.prepareWelcomeMailTemplate();
        mailService.sendWelcomeMail();
    }
}
```

The above concrete command **SendWelcomeMail** binds a **MailService** which acts as a receiver. Let's take a look at the MailService class.

```
public class MailService {
    public void prepareWelcomeMailTemplate() {
        System.out.println("Prepare welcome mail template");
    }

    public void sendWelcomeMail() {
        System.out.println("Send welcome mail to the subscribers");
    }
}
```

The above-mentioned **MailService** prepares a welcome mail template and then sends that email to all the subscribers of the newsletter.

Our third and final request sends an acknowledgement message to the author of the newsletter confirming that the welcome mails are sent to the subscribers. The request looks like this.

```
public class SendAcknowledgementMessage implements Command {
    private MessageService messageService;

    public SendAcknowledgementMessage(MessageService messageService) {
        this.messageService = messageService;
    }
```

```
    @Override
    public void execute() {
        messageService.prepareAcknowledgementMessage();
        messageService.sendAcknowledgement();
    }
}
```

The above concrete command **SendAcknowledgementMessage** binds a **MessageService** which acts as a receiver. Let's take a look at the MessageService.

```
public class MessageService {
    public void prepareAcknowledgementMessage() {
        System.out.println("Prepare acknowledgement message "
            + "mentioning the welcome mails are sent to the "
            + "subscribers");
    }

    public void sendAcknowledgement() {
        System.out.println("Send acknowledgement to the Author");
    }
}
```

We looked around our concrete command object in detail. Now let's design how the scheduler will look like.

```
public class Scheduler {
    private Queue<Command> scheduler;

    public Scheduler() {
        scheduler = new LinkedList<Command>();
    }
```

```
    public void scheduleRequest(Command command) {
        scheduler.add(command);
    }

    public void executeScheduledRequest() {
        if (scheduler.isEmpty()) {
            System.out.println("No requests to execute!");
            return;
        }

        Command command = scheduler.remove();
        command.execute();
    }
}
```

The **Scheduler** schedules the requests in a queue and executes them accordingly.

Everything looks set, now let's use our **Scheduler** to schedule our three requests.

```
public class SchedulerTest {
    public static void main(String[] args) {
        UserService userService = new UserService();
        MailService mailService = new MailService();
        MessageService messageService = new MessageService();

        CollectSubscriberEmails collectEmailCommand
            = new CollectSubscriberEmails(userService);
        SendWelcomeMail sendWelcomeMail
            = new SendWelcomeMail(mailService);
        SendAcknowledgementMessage sendAcknowledgementMessage
            = new SendAcknowledgementMessage(messageService);

        Scheduler scheduler = new Scheduler();

        scheduler.scheduleRequest(collectEmailCommand);
        scheduler.scheduleRequest(sendWelcomeMail);
        scheduler.scheduleRequest(sendAcknowledgementMessage);
```

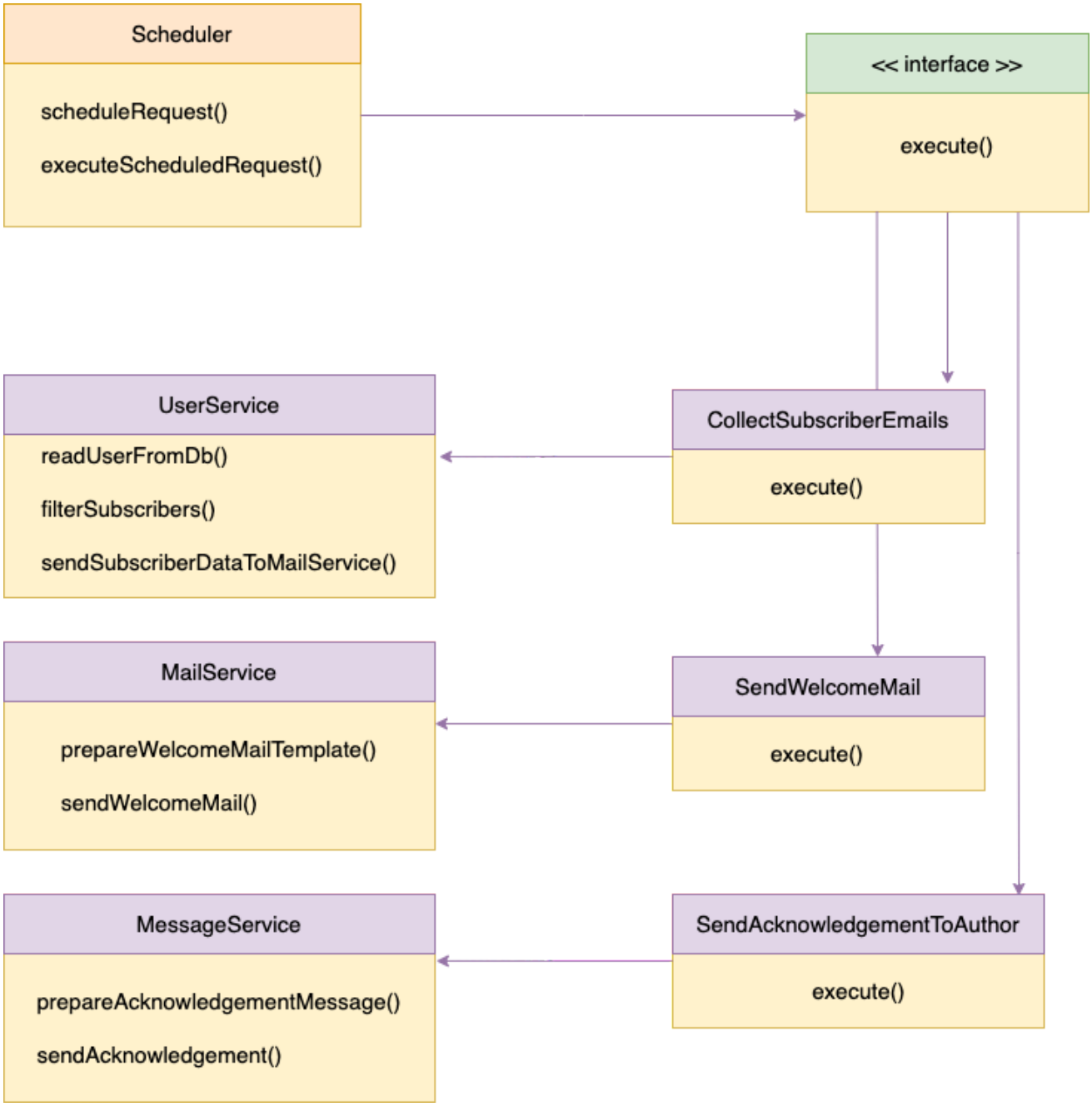
```
        scheduler.executeScheduledRequest();
        scheduler.executeScheduledRequest();
        scheduler.executeScheduledRequest();
    }
}
```

The scheduler output looks like this.

```
==========================
Collect user rows from the User table in database
Filter the unsubscribed users from the collected user rows
Send filtered subscriber details to the mail service
==========================


==========================
Prepare welcome mail template
Send welcome mail to the subscribers
==========================


==========================
Prepare acknowledgement message mentioning the welcome mails are sent to the subscribers
Send acknowledgement to the Author
==========================
```

We saw how the Scheduler scheduled our three requests and executed them one-by-one. We can also witness how the three requests are decoupled from the scheduler class and scheduler only calls their public **execute()** method to get those commands executed.

The **UML** design for our Scheduler example will look like this.


Scheduler
scheduleRequest()
executeScheduledRequest()
<< interface >>
execute()
UserService
readUserFromDb()
filterSubscribers()
sendSubscriberDataToMailService()
CollectSubscriberEmails
execute()
MailService
prepareWelcomeMailTemplate()
sendWelcomeMail()
SendWelcomeMail
execute()
MessageService
prepareAcknowledgementMessage()
sendAcknowledgement()
SendAcknowledgementToAuthor
execute()

Macro Command

A **Macro Command** is a special type of command which can execute a set of concrete commands.

Suppose we want to perform our previous three requests in one go. We can achieve that through Macro Command.

```
public class MacroCommand implements Command {
    Command[] commands;

    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }

    @Override
    public void execute() {
        for(Command command : this.commands) {
            command.execute();
        }
    }
}
```

Let's schedule our previous three commands at once using Macro Command.

```
Command[] commands = {
  collectEmailCommand,
  sendWelcomeMail,
  sendAcknowledgementMessage
};
MacroCommand macroCommand = new MacroCommand(commands);

scheduler.scheduleRequest(macroCommand);
scheduler.executeScheduledRequest();
```

The above code-block uses macro command concept to execute our previous three requests at once. This is the output printed by our scheduler.

```
==========================
Collect user rows from the User table in database
Filter the unsubscribed users from the collected user rows
Send filtered subscriber details to the mail service
Prepare welcome mail template
Send welcome mail to the subscribers
Prepare acknowledgement message mentioning the welcome mails are sent to the subscribers
Send acknowledgement to the Author
==========================
```

Summary

We discussed the concept of **Command Pattern** in detail by designing a Scheduler to schedule multiple requests/commands. We also looked around how a command pattern isolates the execution of a command from the invoker executing it.

At the end we discussed the concept of **Macro Command** to schedule multiple commands at once.

Code Implementation from "**Saurav's LLD Template**"

1. Command Pattern Implementation [View Sample Code]

Thank You! ❤️

Hope this handbook helped you out in clearing out the Low Level Design Pattern concepts

I will be soon coming up with **Part-2** of this Handbook where we will discuss a few more essential Design Patterns from scratch along with their code implementations.

Follow me on Linkedin
Do subscribe to my engineering newsletter "Systems That Scale"