Design Pattern Summary/Review Notes

Singleton

A singleton pattern is an object creational pattern that allows our application to create one and only one instance of a particular class, no matter how many times that class is used in our application.

For example a PropertyReader class that can read the properties from a file and it is used multiple times in our application by different classes. We will create only one object of the PropertyReader and not multiple saving a lot of memory using the singleton pattern. So all application classes will use the same property reader object to perform the property reading from a file.

Another good example for singleton is a logger. Using logger's we can log different types of information errors debugging information and just general information from our application to a log file so that developers can read that log file later on to see what is happening with our application. So a logger can be a singleton the same logger object can be shared across our application classes.

Another good example in the JDBC world, the java database connectivity world is the data source class in the JDBC API the data source class is responsible for maintaining a connection pool and giving a connection from the pool to our application classes. We can have different classes in the application that need a database connection but there will be only one instance of the data source. So a DataSource is also an example of singleton.

UML

Sample Code:

```
public class DateUtil implements Serializable,Cloneable {

private static final long serialVersionUID = 1L;
```

```
private static volatile DateUtil instance;

private DateUtil() {

}

public static DateUtil getInstance() {

if (instance == null) {

synchronized (DateUtil.class) {

if (instance == null) {

instance = new DateUtil();

}

}

}

return instance;

}

protected Object readResolve() {

return instance;

}

@Override

protected Object clone() throws CloneNotSupportedException {

throw new CloneNotSupportedException();

}

}
```

Factory

A factory pattern is a creational pattern that abstracts or hides the object creation process. When you think of factory you can think of a car factory a chocolate factory or a toy factory.A car factory is responsible for manufacturing the cars .A car dealer need not worry about how the car is manufactured. He simply asks the car factory to deliver him some cars. The

car factory is responsible for manufacturing them and delivering them to the dealer.

Similarly the Chocolate Factory delivers different types of chocolates based on what the chocolate store asks them to deliver.

An example from the JDBC space when we use different databases like Oracle, MySql, SqlServer. In Java we use something called JDBC driver to connect to a database.

Driver is an interface in the JDBC API and the implementation for this driver is provided by different vendors. The responsibility of the driver is to connect to a particular database and execute sql statements against it.

To get a connection we need not remember each and every driver in how it works. For example if we want to connect to Oracle you need not deal with the Oracle driver and same for other databases.

You simply use DriverManager which is a class in the JDBC API to get connection and you pass in a connection string which is specific to a particular database where it is different for mysql, it's different for sql server.

The driver manager acts as a factory and it will return the connection for us by using the appropriate driver for Oracle it will use Oracle driver for my sql it will use the my sql driver and so on.

It hides all the details of finding a driver and creating a connection against the database for us. So give it a string it will give you a connection. So here the get connection is a static method on the factory class.

We need not create the instance of the driver manager to create objects of type connection.

We simply use the static method on the factory class and get the object we need.

UML:

Another example is a pizza store .A pizza store delivers different types of pizzas.We will a parent interface which is implemented by the veg pizza cheese pizza and meat pizza.The pizza store need not worry about how to create each of these pizzas.It simply asked the pizza factory to deliver the type of pizza it wants or to create a type of pizza it wants.so that it can give it

to the customer or it can deliver it to the customer.The pizza factory hides the complexity of creating the different types of pizzas from the pizza store.

Sample Code:

```
public class PizzaFactory {

public static Pizza createPizza(String type) {

Pizza p = null;

if (type.equals("cheese")) {

p = new CheesePizza();

} else if (type.equals("chicken")) {

p = new ChickenPizza();

} else if (type.equals("veggie")) {

p = new VeggiePizza();

}

return p;

}

}
```

Abstract Factory

Now that you have mastered the Factory design pattern learning and implementing the Abstract Factory pattern will be quite easy because an abstract factory is a factory of factories.That is a factory pattern was hiding the details of object creation and factory of factories or an abstract factory.hides the creation of the factory itself.

A good example in the Java space is the JAXP API . JAXP stands for Java API for xml parsing . Using this API we can read or write and update the elements in a xml file the key class in this API is the document class that represents a xml document in memory to create a document class.We use that document builder.So this document builder is a factory class and there is one more class document.Build a factory which is responsible for creating the document

builder itself.So the document builder factory is a abstract factory because it is a factory of factories and the use case you are going to work on to give you

Another example is a DAO factory. DAO stands for data access object.We'll learn more about it in sections later on.It simply is a class that can read write create update data we can have different types of DAO'sDAO's that deal with xml data and DAO's that deal with DB data and within xml we can have employee information department information.

Similarly within that database we can have employ information and department information.So you can see that we can have a factory to deal with these separate DOA's we can have a DB DAO factory that can give us one of these classes here when our application needs them and we can have a xml DAO factory which can give one of the classes here.Now to get one of these factories themselves these factories will first implement a DAO abstract factory or they will extend DAO abstract factory class and we will have a producer which is responsible for creating one of these factories so abstract factory is a factory off factories.

It simply creates the factory we need. when we have multiple factories we see in our application.

UML:

Sample Code:

```
public class DaoFactoryProducer {

public static DaoAbstractFactory produce(String factoryType) {

DaoAbstractFactory daf = null;

if (factoryType.equals("xml")) {

daf = new XMLDaoFactory();

} else if (factoryType.equals("db")) {

daf = new DBDaoFactory();

}

return daf;

}
```

```
}
```

Flyweight

A flyweight design pattern can be used to save memory. A flyweight is a structural design pattern instead of creating a large number of similar objects.We can reduce the number of objects that are created by reusing the objects and saving memory. Memory is a huge concern especially when it comes to mobile applications with limited memory.

Let's say we are working on a paint app that allows users to draw different shapes that is circles and rectangles.

Here we have a shape interface with a draw method on it.

When we have a circle class and the rectangle classes that implements that interface using a circle.

The user can draw a circle using a rectangle of course he can try her rectangle to do that.

He will have to create a circle and assign a radius for each circle he wants to create.

Similarly for each rectangle he will how to set the length and the breadth by creating a separate object by using the flyweight pattern.

You will do all that by creating one single circle and one single rectangle he will see what the problem is he will implement the problem first and then you will resolve that problem of multiple object creation by using the fly design pattern in the next few lectures.

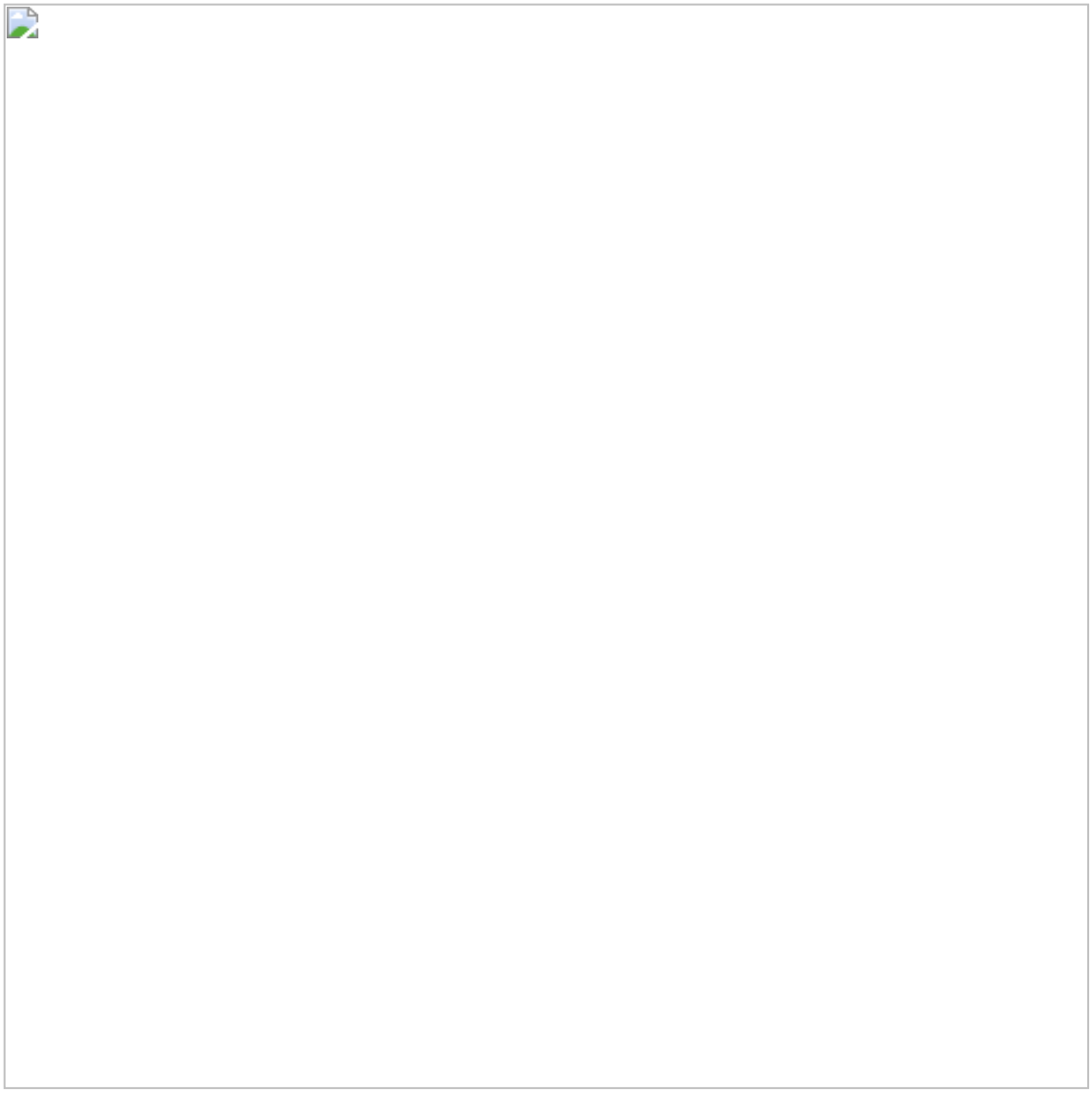
UML:

Sample Code:

```
public class ShapeFactory {

private static Map<String, Shape> shapes = new HashMap<>();
```

```
public static Shape getShape(String type) {

Shape shape = null;

if (shapes.get(type) != null) {

shape = shapes.get(type);

} else {

if (type.equals("circle")) {

shape = new Circle();

} else if (type.equals("rectangle")) {

shape = new Rectangle();

}

shapes.put(type, shape);

}

return shape;

}

}
```

Template Method:



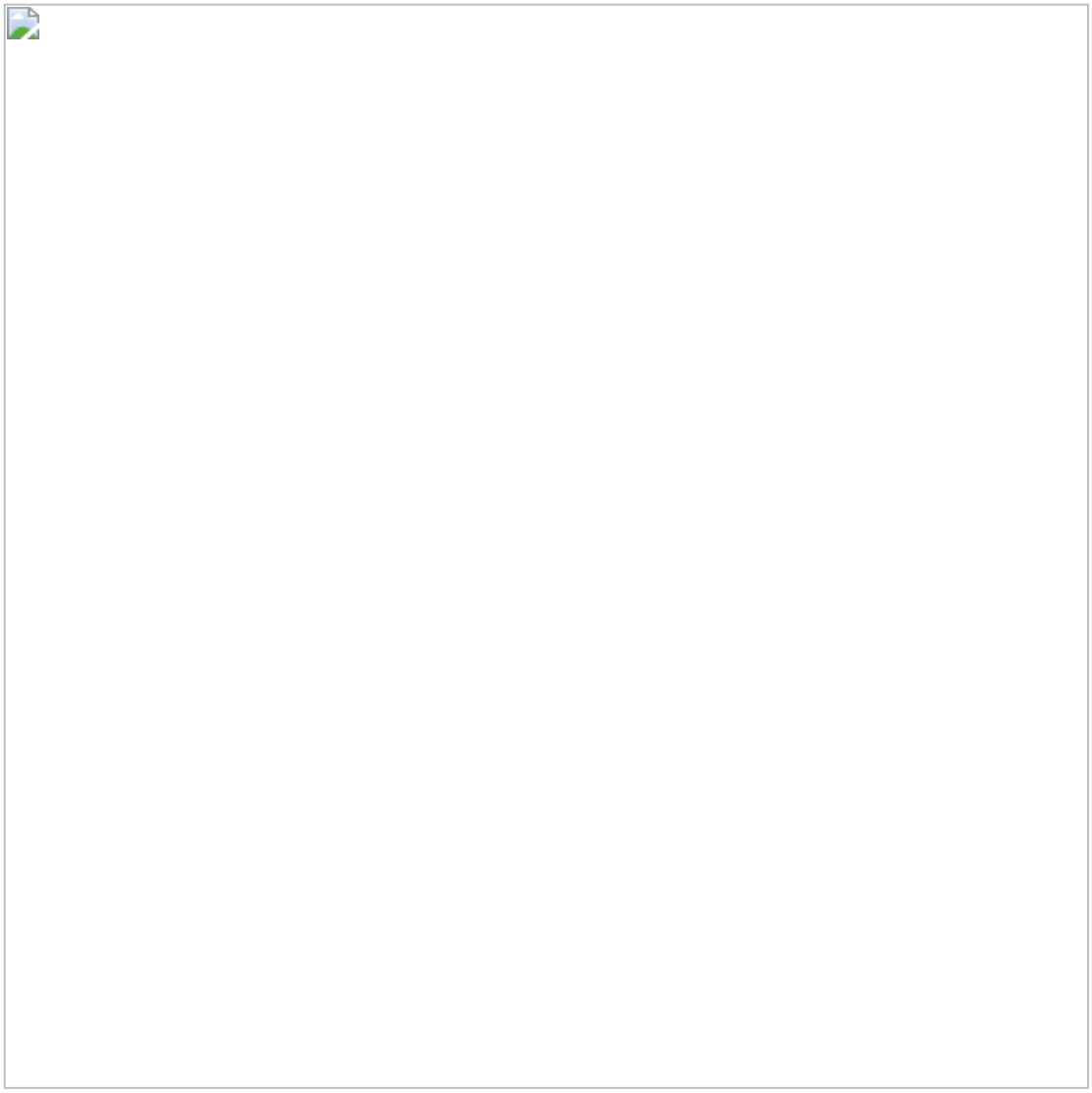
The template method pattern is a behavioural pattern. And as the name itself says it provides a base template method. When we are working with inheritance in our applications we provide a base template method that should be used by the child classes. The child classes can override certain methods but they should use the base template method as is.

For example we have a data renderer class which can read the data, process the data and then render or display that data to the end user.But in our application we want to render the data in the same way no matter in which format the data is coming in that is if it is xml data or if it is CSV data. We want to render it using the render method in the base class reading the data and processing that data is up to the child classes. The child classes can override the readData and the processData.But we want to provide a base template method with all the implementation in it in that data renderer superclass. This pattern is called template method ,as we are providing a template for a particular method from the parent class that should be used by the child classes.

Code Sample:

```
public abstract class DataRenderer{
public void render(){
String data = readData();
String processedData = processData(data);
System.out.println(processedData);
}
public abstract String readData();
public abstract String processData(String data);
}
```

Adapter



If you have used a power adapter then you already know what an adapter pattern is .The job of a power adapter is to adapt it to a particular location and a particular switchboard.For example the same laptop plug pins that work in USA will not work in UK and in India.We will have to use appropriate power adapter that can take our laptop pins into it and on the other side of it it will have pins that can go into the local countries switchboard and it can also adapt to the appropriate range in that country.

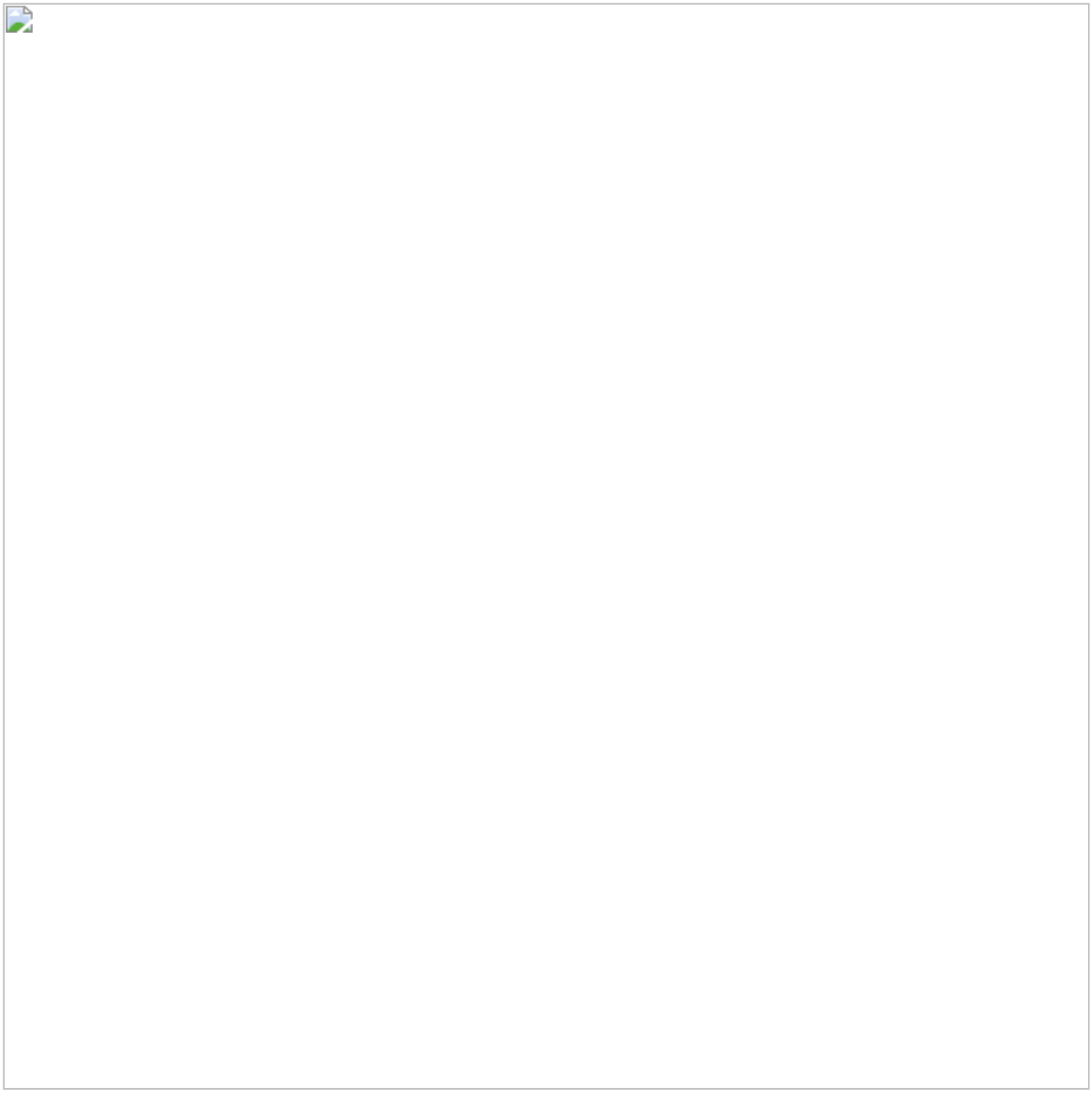
Similarly in the world of programming when we have two applications communicating with each other or two objects using each other and one object invokes the method of another object.Then we have to adapt in some cases.

For example here we have a WeatherFinder class which has a findWeather. By passing in a city you can get the weather and we have an implementation of it which will return the weather back and there is a UI class that wants to use the weather finder . But the UI only knows the zip code of the city. It does not have the city information it only has the zip code but it wants to get the weather of it.

That is where an adapter comes in. We will implement an adapter which will take the zip code. The weather UI will invoke the findTemperature Method on the WeatherAdapter it will pass in the zip code. The weather adapter is responsible for looking up for the appropriate city that matches the zip code and hen invoke the weather finder, take the results and return the results back to the weather UI. So it exactly acts like a power adapter. It takes the inputs from the class that wants to use another class because the inputs here are different from what the other side of the relationship expects.

```
public class WeatherAdapter {

public int findTemperature(int zipCode) {

String city = null;

if (zipCode == 19406) {

city = "King Of Prussia";

}

WeatherFinder finder = new WeatherFinderImpl();

int temperature = finder.find(city);

return temperature;

}

}
```

Command



A command design pattern is a behavioural design pattern from that gang of four patterns. It is used to encapsulate a request as an object and pass it to an invoker the invoker doe not know how to service the request from the client. It will take the command and pass it to a receiver who knows how to perform the action typically.

There are five actors in the command design pattern. They are the **command** itself. The **client** the invoker the concrete command that implements the command and a **receiver** who knows how to perform all the actions.

Let's take a look at an **example** to see all these five actors in action. Let's consider a person who is using a television or is watching a television and he

uses a remote control typically to perform several operations. But let's simply take the on and off operations and let's see how the command pattern fits in here.Here the person is the client who wants to execute the on and off command on it television, the remote control is the invoker so he uses remote control to invoke a particular command by pressing a button and the commands themselves are the on command and off command that implement a interface called Command which has a execute method.

So the person will wrap this on command passes it to the remote control the remote control will send that command to the television and the television knows how to perform that action based on the command that comes in when it is on.

It will execute the command which is passed in which is the on command execute method. And when it is off it will perform the Execute method of the off command to switch off the television.

Here person is the client remote control is the Invoker command is the command interface on command and off command are the concrete command classes. And finally television is the receiver who knows how to perform the action. The huge advantage of the command pattern is that the invoker which is the client and the remote control are completely decoupled from the receiver.

The person need not touch the television or he need not know how to perform the on and off command he simply uses the remote control and presses the button the remote control also doesn't care how the actions are performed. It simply passes the command to the television. That way they are completely decoupled the invoker does not know the details of the action that needs to be performed.

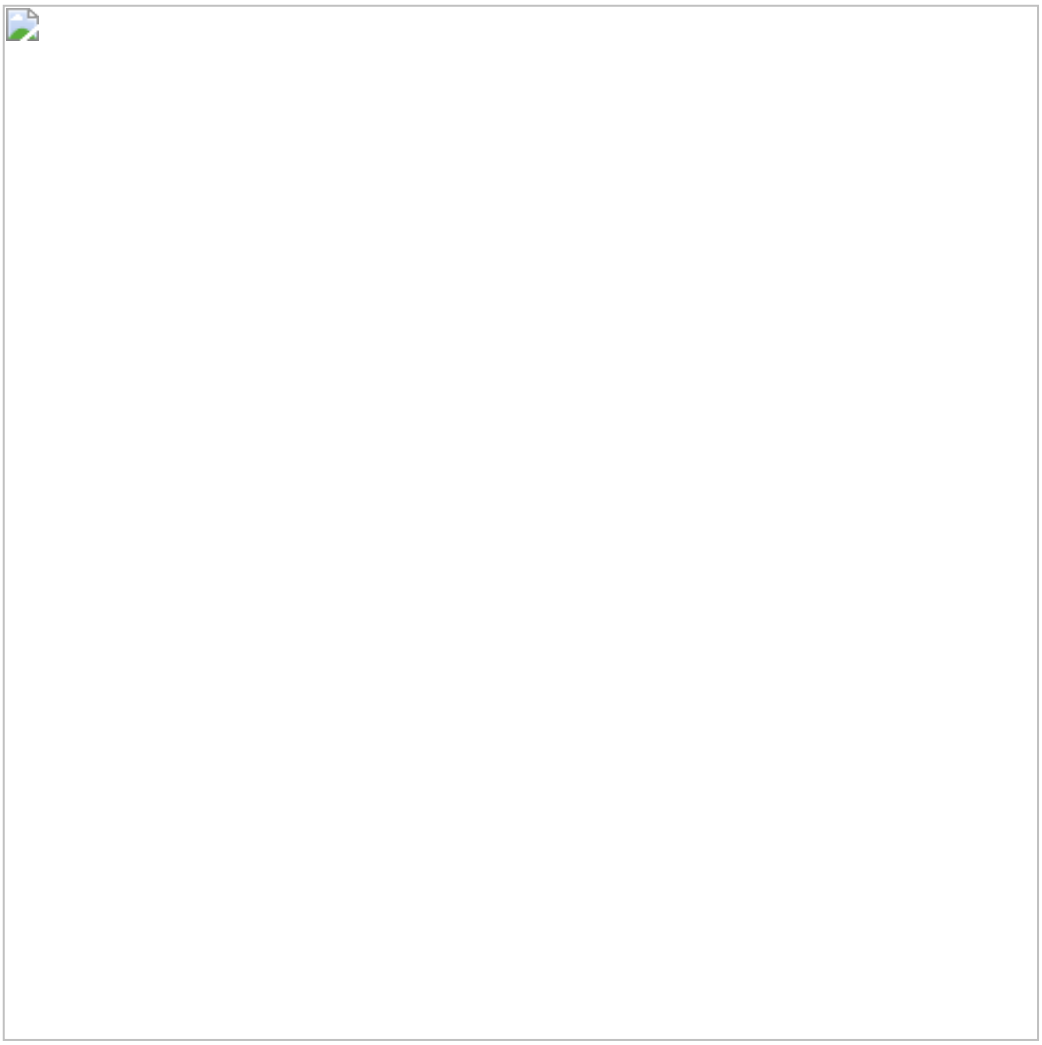
The receiver here the television can change the implementation of how the on and off should be performed without impacting the remote control and the person.

Sample Code:

```
public class RemoteControl{

private Command command;

public void pressButton(){
```

```
command.execute();

}

}
```

Decorator



A decorator pattern is a behavioural pattern that adds additional functionality to an object dynamically at runtime. A decorator wraps an object with additional behaviour without affecting other objects of the same type. The classes in the input output streams in Java use the decorator pattern to read and write files.

For example lets consider a pizza shop. We have a pizza and we have a base pizza. A plain pizza, pizza by itself doesn't mean anything it is very abstract. A plain pizza probably it is just the dove without any cheese or veggies or any

meat on it. And now when the client wants a plain pizza or he can ask for a veggie pizza or a cheese pizza or meat pizza

At runtime we can dynamically add all these toppings using a pizza decorator as required. A pizza decorator will be implemented by veggie pizza decorator and a cheese pizza decorator. Each of these bring in additional functionality. So if the client asks for a veggie pizza we are going to use the veggie pizza decorator at run time.

The client ask for a cheese pizza. We can ask the cheese pizza decorator to decorate the plain base pizza with cheese and with veggies as required. The pizza here is the component the plain pizza is a concrete or a base component. The decorator is the pizza decorator and these two are concrete decorators. Both the veggie pizza decorator and the cheese pizza decorator are called concrete decorators so you are going to implement all of that in the next few lectures.

Samle Code:

```
public class PizzaShop{

public static void main(String args[]){

Pizza pizza = new VeggiePizzaDecorator(new CheesePizzaDecorator(new
PlainPizza()));

}

}
```