Software Architecture Design SAD :)

AT70.18

Asian Institute of Technology
School of Engineering and Technology

Design Pattern

Submitted to:
Dr. Chaklam Silpasuwanchai

Submitted by:
Abhishek Koirala (st120199)
Bibhuti regmi (st120320)
Shikshya Dahal (st120186)

Submitted Date: 8th March 2019

Role of each member:

We first discussed on what project we were going to implement design patterns on. We then decided to build an Android application and implement 4 design patterns. A project was created and added to Github. Each of us imported the same project and worked on each component of the application. The documentation and presentation part were also divided equally

Software Design pattern implemented :
Creational: Builder

The Builder design pattern is a creational design pattern and can be used to create complex objects step by step. The main intent is to separate the construction of a complex object from its representation so that the same construction process can create different representations.

Use Case(Builder)

-If there are multiple constructor having combinations of multiple parameter with nested objects.

-If there are large number of parameters.

Structural:Adapter, Facade

Adapter :
The adapter design pattern is a structural design pattern that allows two unrelated/uncommon interfaces to work together. In other words, the adapter pattern makes two incompatible interfaces compatible without changing their existing code. The main intent is to wrap an existing class with a new interface.

Use case (Adapter pattern) :

-If you want to create an intermediary abstraction that translates, or maps, one component to another system that are incompatible.

Facade :
Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. Facade provides 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 main intent is to wrap a complicated subsystem with a simpler interface.

Use case (Adapter pattern) :

- If you want to provide a simple interface to a complex subsystem. The application of design patterns often results in a lot of small classes which makes subsystems more flexible and customizable. A facade can provide a default view for clients which don't need to customize.
- If you want to reduce coupling between clients-subsystems or subsystems-subsystems.
- If you want to layer your subsystems. Use a facade to define an entry point to each subsystem level and make them communicate only through their facades, this can simplify the dependencies between them.

Behaviour: Command

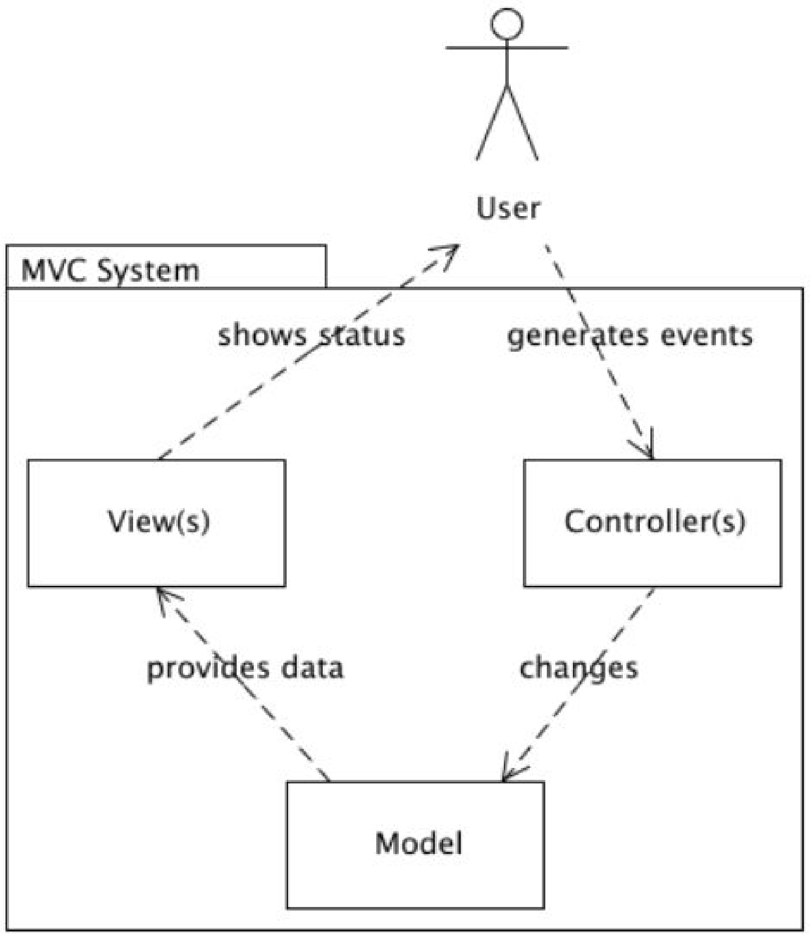
Command pattern is a data driven design pattern. A request is wrapped under an object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.The main intent is to encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations.

Use case (Adapter pattern) :

-If you want to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

UML :

Architectural pattern implemented :
Model View Controller pattern



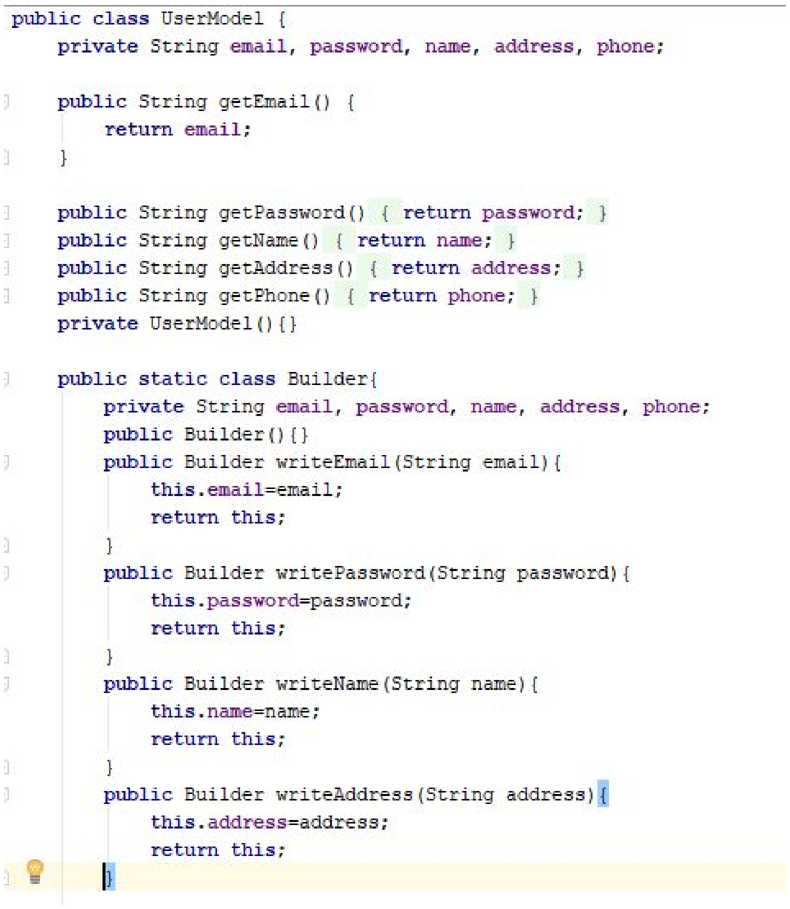
Are you satisfied with your code implementations? Discuss in terms of future maintainability.

- Yes, we are satisfied with our code implementations. Our operation logic or business logic is very much separated from client code. The design patterns that we have implemented(builder,adapter , facade and commands) are easily maintainable and scalable no matter how many users we grow in the future.

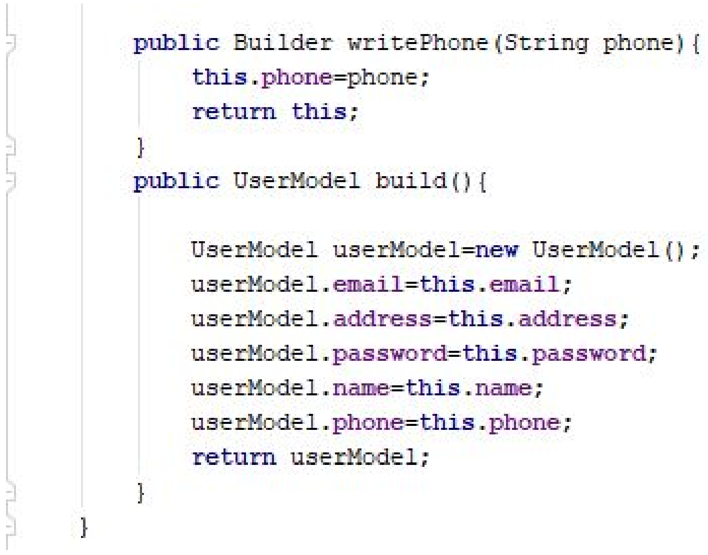
Implementation

Builder:
Two Model classes are used in the project which are implemented using Builder design pattern. The implementation is shown in figures below :

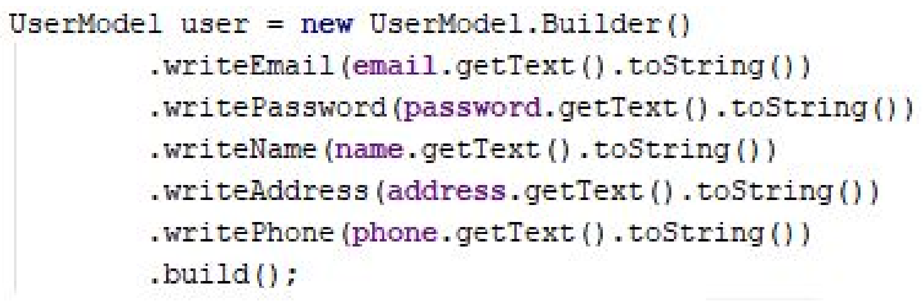
- **UserModel**



```java
public class UserModel {
    private String email, password, name, address, phone;

    public String getEmail() {
        return email;
    }

    public String getPassword() { return password; }
    public String getName() { return name; }
    public String getAddress() { return address; }
    public String getPhone() { return phone; }
    private UserModel(){}

    public static class Builder{
        private String email, password, name, address, phone;
        public Builder(){}
        public Builder writeEmail(String email){
            this.email=email;
            return this;
        }
        public Builder writePassword(String password){
            this.password=password;
            return this;
        }
        public Builder writeName(String name){
            this.name=name;
            return this;
        }
        public Builder writeAddress(String address){
            this.address=address;
            return this;
        }
```

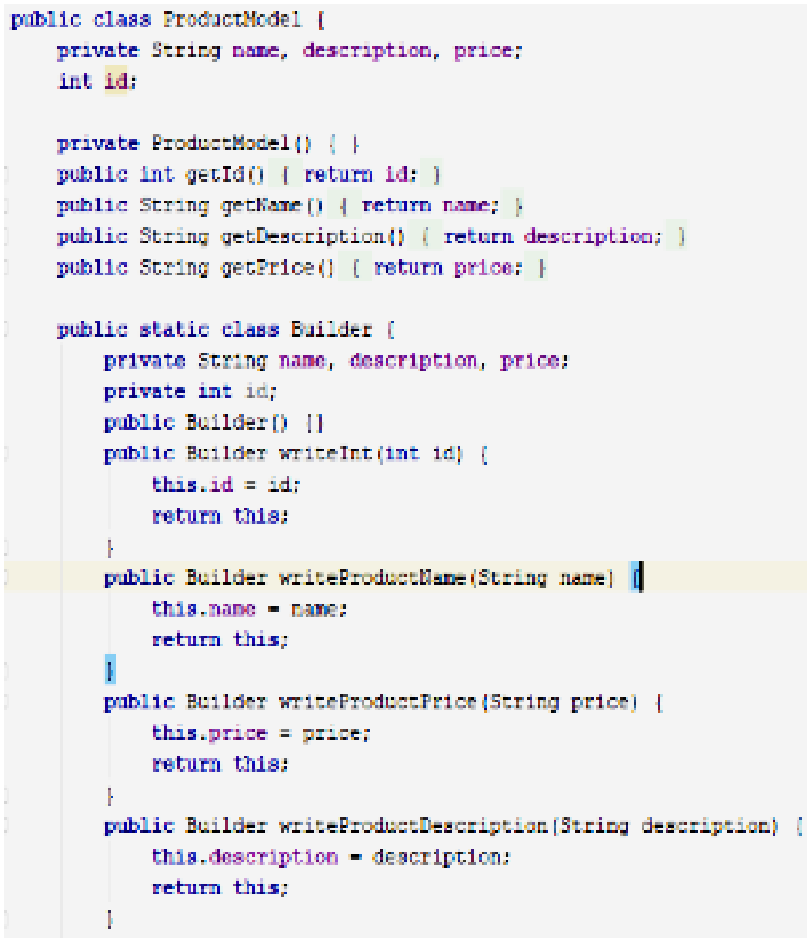


```java
        public Builder writePhone(String phone){
            this.phone=phone;
            return this;
        }
        public UserModel build(){

            UserModel userModel=new UserModel();
            userModel.email=this.email;
            userModel.address=this.address;
            userModel.password=this.password;
            userModel.name=this.name;
            userModel.phone=this.phone;
            return userModel;
        }
    }
```

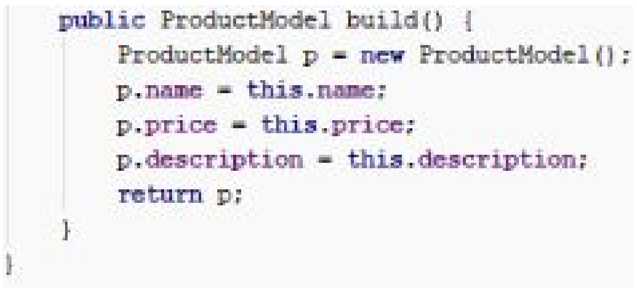


```java
UserModel user = new UserModel.Builder()
        .writeEmail(email.getText().toString())
        .writePassword(password.getText().toString())
        .writeName(name.getText().toString())
        .writeAddress(address.getText().toString())
        .writePhone(phone.getText().toString())
        .build();
```

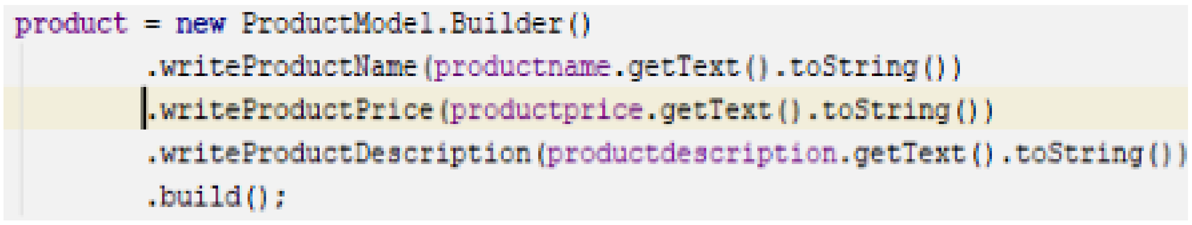
- **ProductModel**



```java
public class ProductModel {
    private String name, description, price;
    int id;

    private ProductModel() { }
    public int getId() { return id; }
    public String getName() { return name; }
    public String getDescription() { return description; }
    public String getPrice() { return price; }

    public static class Builder {
        private String name, description, price;
        private int id;
        public Builder() {}
        public Builder writeInt(int id) {
            this.id = id;
            return this;
        }
        public Builder writeProductName(String name) {
            this.name = name;
            return this;
        }
        public Builder writeProductPrice(String price) {
            this.price = price;
            return this;
        }
        public Builder writeProductDescription(String description) {
            this.description = description;
            return this;
        }
```



```java
    public ProductModel build() {
        ProductModel p = new ProductModel();
        p.name = this.name;
        p.price = this.price;
        p.description = this.description;
        return p;
    }
}
```

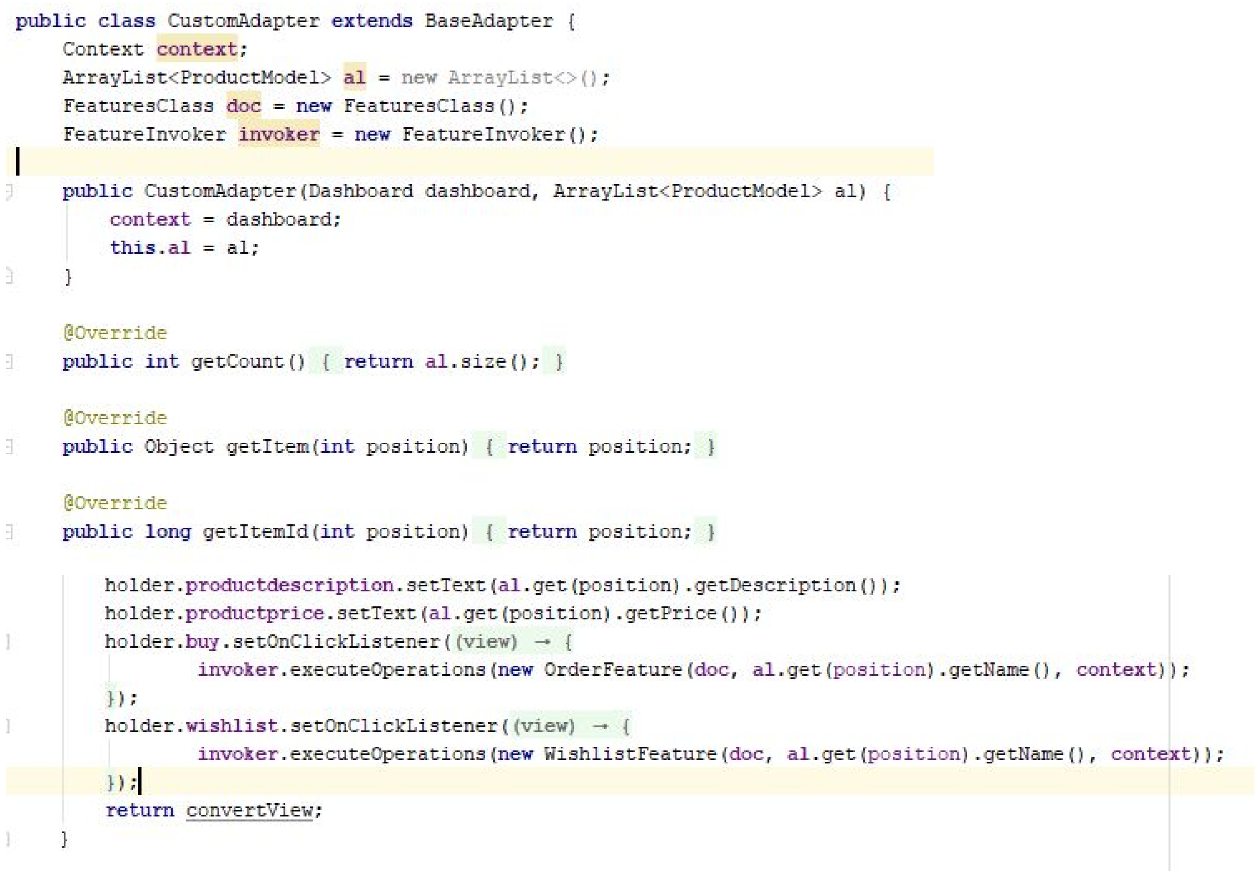


```java
product = new ProductModel.Builder()
        .writeProductName(productname.getText().toString())
        .writeProductPrice(productprice.getText().toString())
        .writeProductDescription(productdescription.getText().toString())
        .build();
```

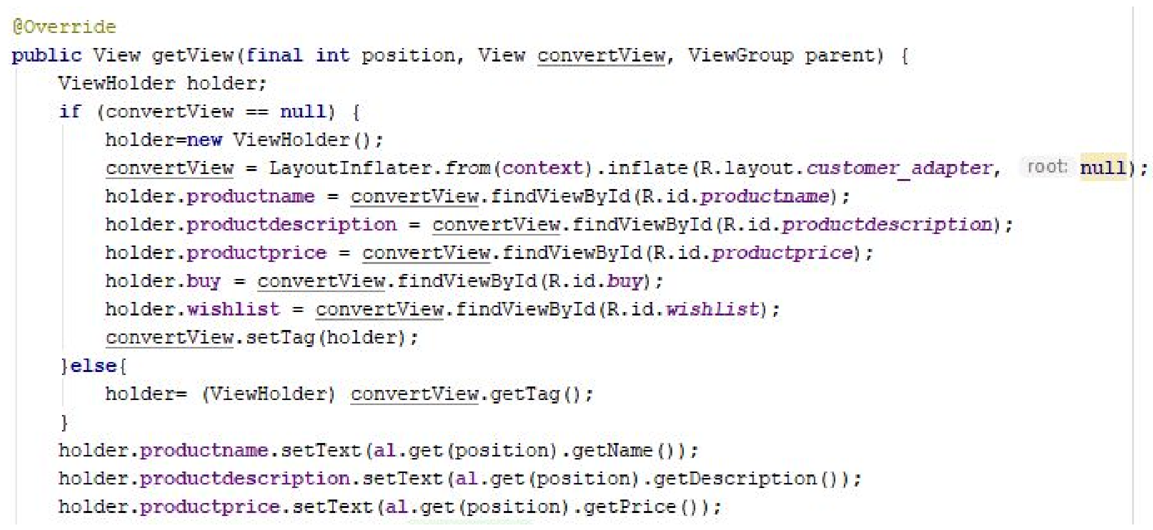
Adapter

BaseAdapter is an Android Component. BaseAdapter specifies an interface to use for programs that need adapters. In fact, this interface is called Adapter. This interface specifies the information needed to take an arbitrary list, and convert it to an arbitrarily long set of Views. BaseAdapter thus adapts custom input format to the Adapter interface. Thus, it is an Adapter design pattern.

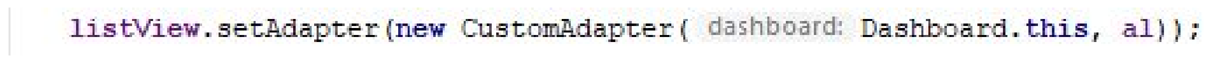
We have used BaseAdapter indirectly to implement our view logic for each product on the product list



```java
public class CustomAdapter extends BaseAdapter {
    Context context;
    ArrayList<ProductModel> al = new ArrayList<>();
    FeaturesClass doc = new FeaturesClass();
    FeatureInvoker invoker = new FeatureInvoker();

    public CustomAdapter(Dashboard dashboard, ArrayList<ProductModel> al) {
        context = dashboard;
        this.al = al;
    }

    @Override
    public int getCount() { return al.size(); }

    @Override
    public Object getItem(int position) { return position; }

    @Override
    public long getItemId(int position) { return position; }

        holder.productdescription.setText(al.get(position).getDescription());
        holder.productprice.setText(al.get(position).getPrice());
        holder.buy.setOnClickListener((view) → {
                invoker.executeOperations(new OrderFeature(doc, al.get(position).getName(), context));
        });
        holder.wishlist.setOnClickListener((view) → {
                invoker.executeOperations(new WishlistFeature(doc, al.get(position).getName(), context));
        });
        return convertView;
    }
```



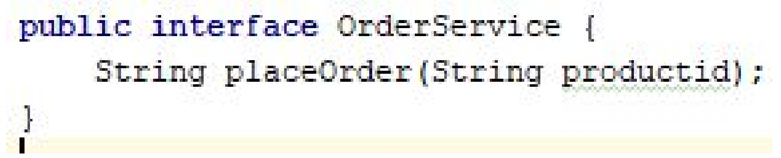
```
@Override
public View getView(final int position, View convertView, ViewGroup parent) {
    ViewHolder holder;
    if (convertView == null) {
        holder=new ViewHolder();
        convertView = LayoutInflater.from(context).inflate(R.layout.customer_adapter,  root: null);
        holder.productname = convertView.findViewById(R.id.productname);
        holder.productdescription = convertView.findViewById(R.id.productdescription);
        holder.productprice = convertView.findViewById(R.id.productprice);
        holder.buy = convertView.findViewById(R.id.buy);
        holder.wishlist = convertView.findViewById(R.id.wishlist);
        convertView.setTag(holder);
    }else{
        holder= (ViewHolder) convertView.getTag();
    }
    holder.productname.setText(al.get(position).getName());
    holder.productdescription.setText(al.get(position).getDescription());
    holder.productprice.setText(al.get(position).getPrice());
```



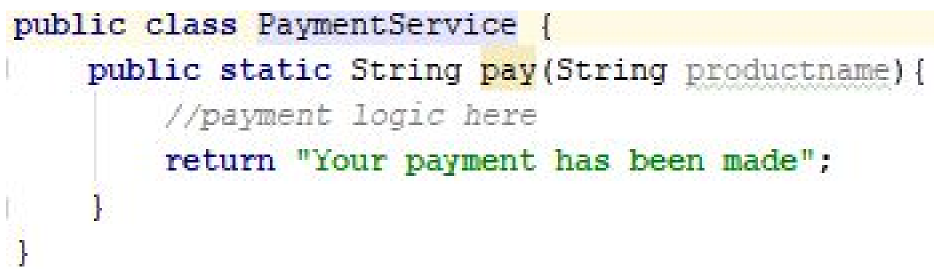
```
    listView.setAdapter(new CustomAdapter( dashboard: Dashboard.this, al));
```

Facade

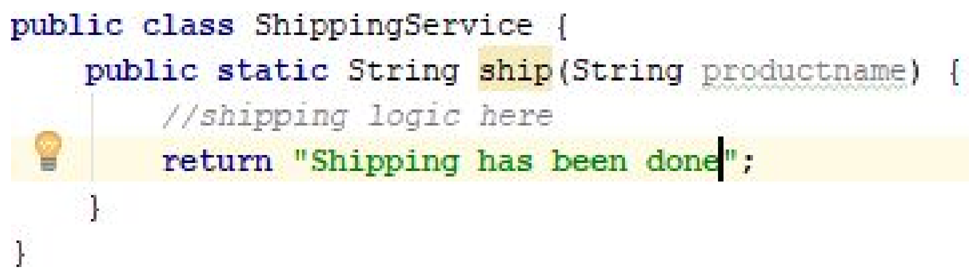
Our application has a combined payment and shipping logic when user presses buy button on the app interface. This would trigger a method on **OrderServiceImplementation.java** which will then trigger respective methods in **PaymentService.java** and **ShippingService.java** .



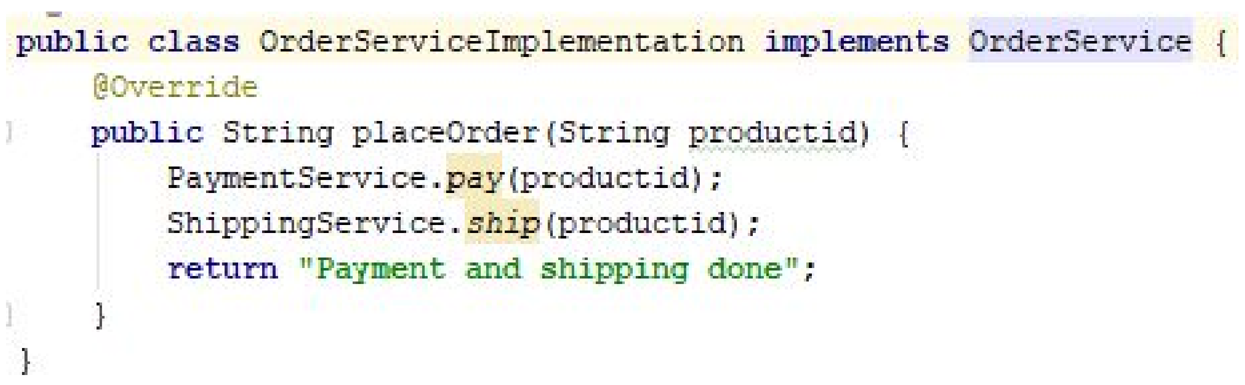
```
public interface OrderService {
    String placeOrder(String productid);
}
```



```
public class PaymentService {
    public static String pay(String productname){
        //payment logic here
        return "Your payment has been made";
    }
}
```



```
public class ShippingService {
    public static String ship(String productname) {
        //shipping logic here
        return "Shipping has been done";
    }
}
```

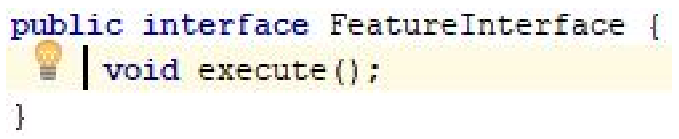


```java
public class OrderServiceImplementation implements OrderService {
    @Override
    public String placeOrder(String productid) {
        PaymentService.pay(productid);
        ShippingService.ship(productid);
        return "Payment and shipping done";
    }
}
```

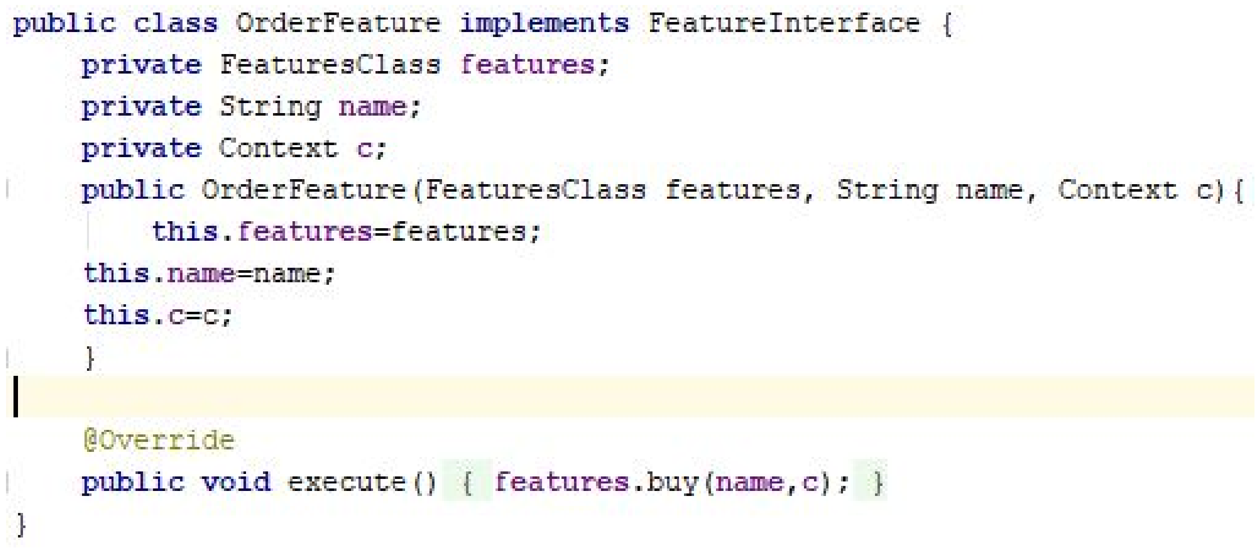
Command

The app has 2 features for user. Users can buy a product(implemented with facade logic) and add the product to wishlist. Both these features are implemented using Command Design pattern.

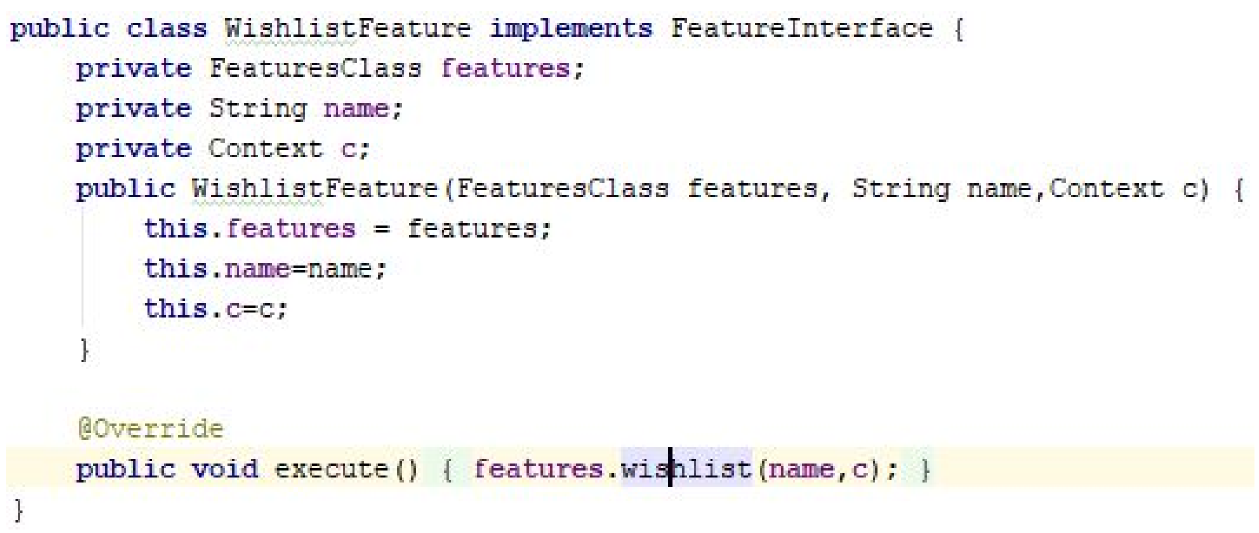
The **OrderFeature.java** and **WishListFeature.java** implements FeatureInterface.java which consists of execute( ) method.The client command is passed to the invoker class **FeatureInvoker.java** which then executes respective feature in **FeatureClass.java**



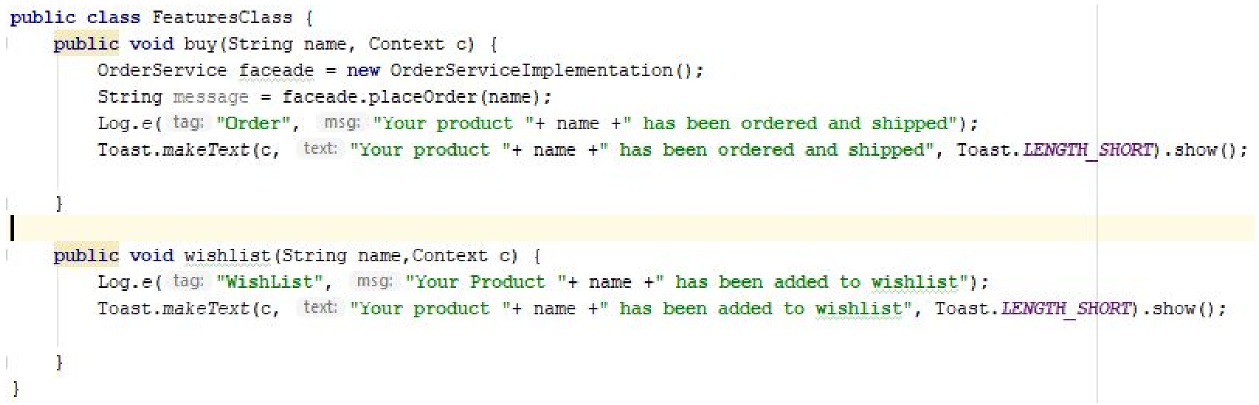
```java
public interface FeatureInterface {
    void execute();
}
```



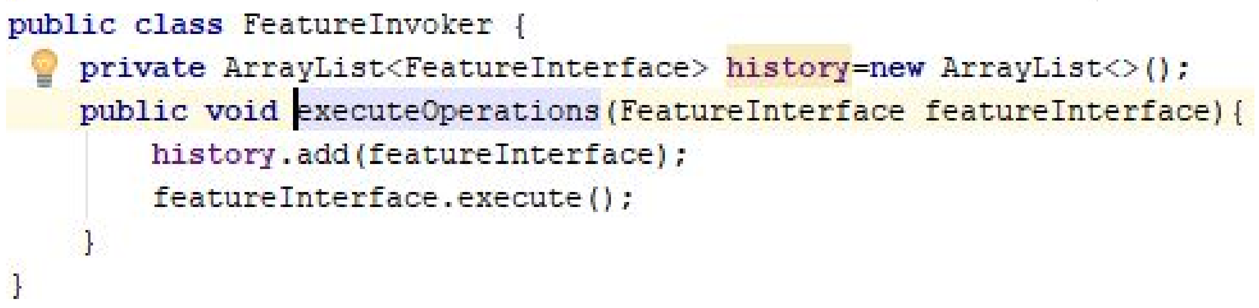
```java
public class OrderFeature implements FeatureInterface {
    private FeaturesClass features;
    private String name;
    private Context c;
    public OrderFeature(FeaturesClass features, String name, Context c){
        this.features=features;
    this.name=name;
    this.c=c;
    }

    @Override
    public void execute() { features.buy(name,c); }
}
```



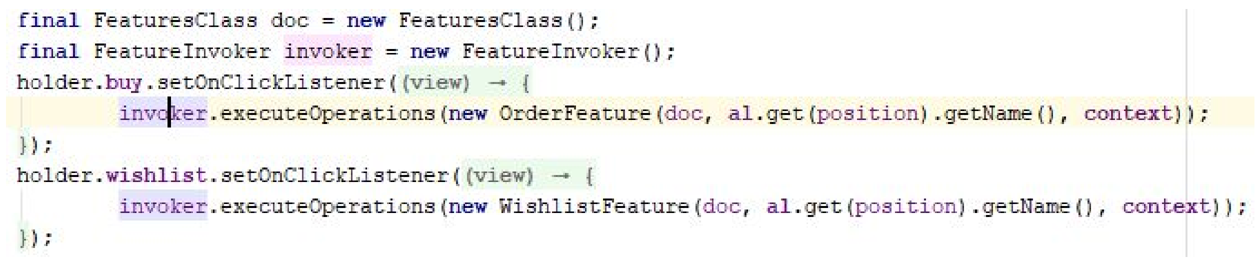
```java
public class WishlistFeature implements FeatureInterface {
    private FeaturesClass features;
    private String name;
    private Context c;
    public WishlistFeature(FeaturesClass features, String name,Context c) {
        this.features = features;
        this.name=name;
        this.c=c;
    }

    @Override
    public void execute() { features.wishlist(name,c); }
}
```



```java
public class FeaturesClass {
    public void buy(String name, Context c) {
        OrderService faceade = new OrderServiceImplementation();
        String message = faceade.placeOrder(name);
        Log.e( tag: "Order",  msg: "Your product "+ name +" has been ordered and shipped");
        Toast.makeText(c,  text: "Your product "+ name +" has been ordered and shipped", Toast.LENGTH_SHORT).show();

    }

    public void wishlist(String name,Context c) {
        Log.e( tag: "WishList",  msg: "Your Product "+ name +" has been added to wishlist");
        Toast.makeText(c,  text: "Your product "+ name +" has been added to wishlist", Toast.LENGTH_SHORT).show();

    }
}
```



```java
public class FeatureInvoker {
    private ArrayList<FeatureInterface> history=new ArrayList<>();
    public void executeOperations(FeatureInterface featureInterface){
        history.add(featureInterface);
        featureInterface.execute();
    }
}
```



```java
final FeaturesClass doc = new FeaturesClass();
final FeatureInvoker invoker = new FeatureInvoker();
holder.buy.setOnClickListener((view) → {
        invoker.executeOperations(new OrderFeature(doc, al.get(position).getName(), context));
});
holder.wishlist.setOnClickListener((view) → {
        invoker.executeOperations(new WishlistFeature(doc, al.get(position).getName(), context));
});
```

When should one apply or not apply Design Patterns?
Design patterns should be applied when :

- Solutions to problems that recur with variations
    - No need for reuse if problem only arises in one context
- Solutions that require several steps:
    - Not all problems need all steps
    - Patterns can be overkill if solution is a simple linear set of instructions
- Solutions where the solver is more interested in the existence of the solution than its complete derivation
    - Patterns leave out too much to be useful to someone who really wants to understand

Design pattern should not be applied when the solution is simple. The use of patterns is not appropriate when it's overdone.

Anti pattern:
Anti-patterns are certain patterns in software development that are considered bad programming practices.

As opposed to design patterns which are common approaches to common problems which have been formalized and are generally considered a good development practice, anti-patterns are the opposite and are undesirable.

For example, in object-oriented programming, the idea is to separate the software into small pieces called objects. An anti-pattern in object-oriented programming is a God object which performs a lot of functions which would be better separated into different objects.

Eg:

Anti pattern	**Design Pattern**

class GodObject {
        function PerformInitialization() {}
        function ReadFromFile() {}
        function WriteToFile() {}
        function DisplayToScreen() {}
        function PerformCalculation() {}
        function ValidateInput() {}
        // and so on... //
}
	class FileInputOutput {
	   function ReadFromFile() {}
	   function WriteToFile() {}
	}
	
	class UserInputOutput {
	   function DisplayToScreen() {}
	   function ValidateInput() {}
	}
	
	class Logic {
	   function PerformInitialization() {}
	   function PerformCalculation() {}
	}

The example above has an object that does *everything*. In object-oriented programming, it would be preferable to have well-defined responsibilities for different objects to keep the code less coupled and ultimately more maintainable.

Examples of anti-pattern:

1.Swiss Army Knife:

A Swiss Army Knife, also known as Kitchen Sink, is an excessively complex class interface. The designer attempts to provide for all possible uses of the class. In the attempt, he or she adds a large number of interface signatures in a futile attempt to meet all possible needs.

Real-world examples of Swiss Army Knife include from dozens to thousands of method signatures for a single class. The designer may not have a clear abstraction or purpose for the class, which is represented by the lack of focus in the interface.

This AntiPattern is problematic because it ignores the force of managing complexity, that is, the complicated interface is difficult for other programmers to understand, and obscures how the class is intended to be used, even in simple cases. Other consequences of complexity include the difficulties of debugging, documentation, and maintenance.

2. Useless (Poltergeist) Classes

Useless classes with no real responsibility of their own, often used to just invoke methods in another class or add an unneeded layer of abstraction.Poltergeist classes add complexity, extra code to maintain and test, and make the code less readable—the reader first needs to realize what the poltergeist does, which is often almost nothing, and then train herself to mentally replace uses of the poltergeist with the class that actually handles the responsibility.

REFERENCE:
https://sourcemaking.com/design_patterns/builder

https://sourcemaking.com/antipatterns/software-architecture-antipatterns
https://sahandsaba.com/nine-anti-patterns-every-programmer-should-be-aware-of-with-examples.html