Design Patterns

E-commerce Mockup app

Presented by:

Abhishek Koirala
Bibhuti Regmi
Shikshya Dahal

Introduction to design patterns

A pattern is a solution to a problem in context

What advantages does design pattern provide

- Inspiration
  - *Patterns don't provide solutions, they **inspire** solutions.*
  - Patterns **explicitly** capture expert knowledge and design tradeoffs and make this expertise widely available.
  - Ease the transition to object-oriented technology.

- Patterns improve developer communication
  - Pattern names form a **vocabulary.**

- Help document the architecture of a system
  - Enhance understanding.

- Design patterns enable large-scale reuse of software architectures

What have we built

An e-commerce mock up app:

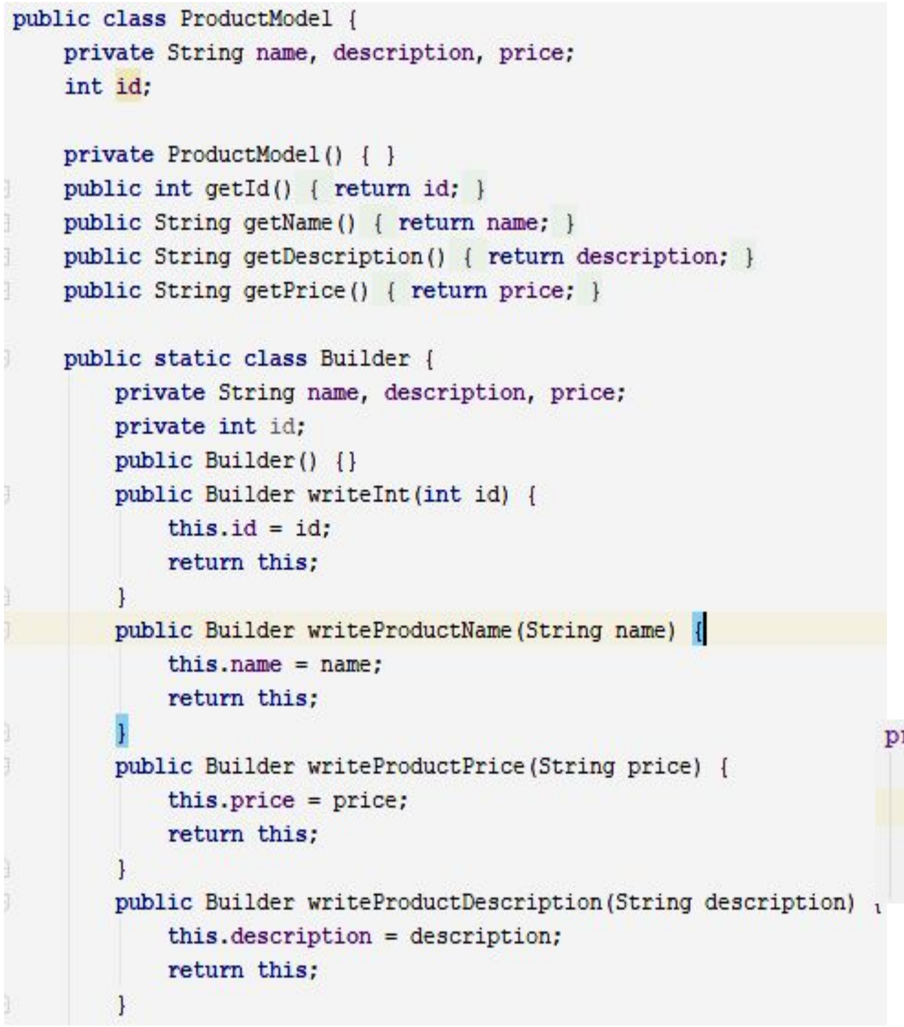
- UserModel
- ProductModel

3 features:

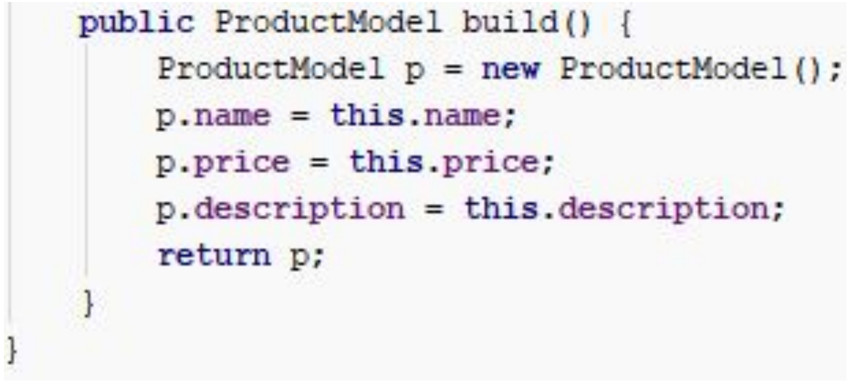
- User can register and sign in
- User can buy their product(payment and shipping service)
- User can add product to wishlist

Design Patterns implemented

Builder(Creational)

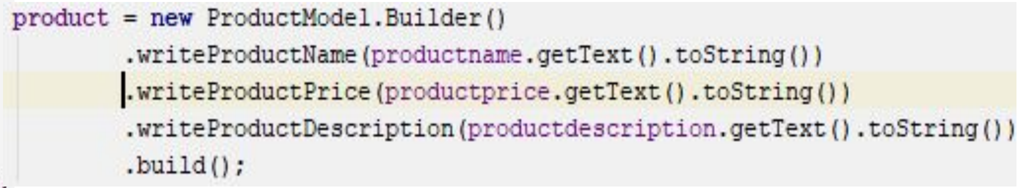


```java
public class ProductModel {
    private String name, description, price;
    int id;

    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;
        }
```



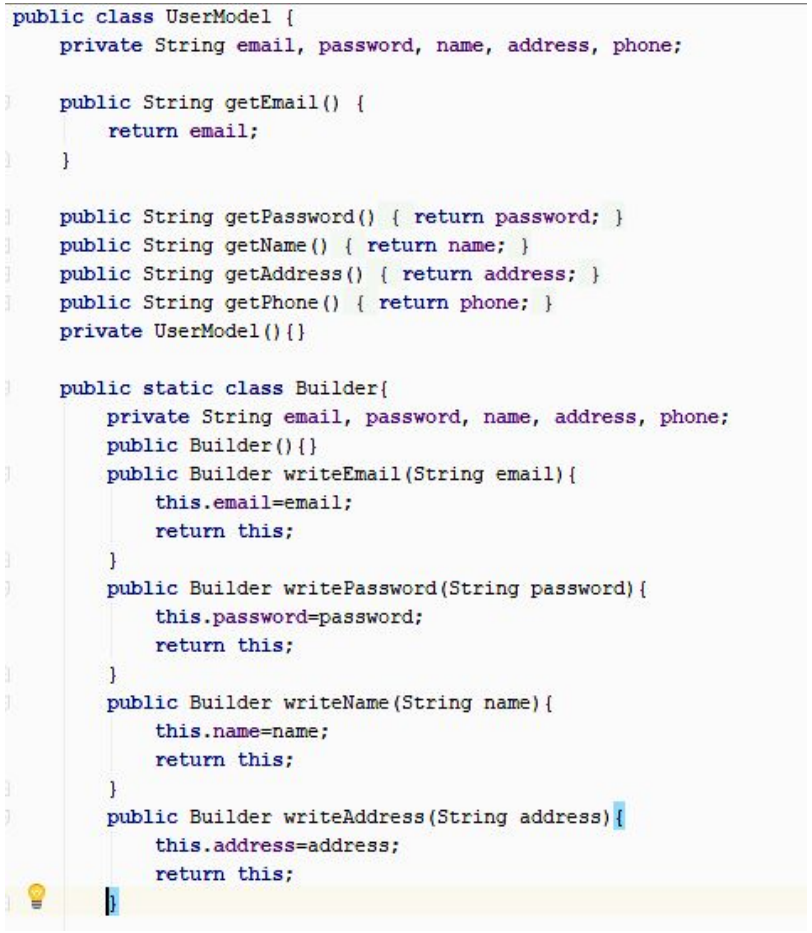
```java
    public ProductModel build() {
        ProductModel p = new ProductModel();
        p.name = this.name;
        p.price = this.price;
        p.description = this.description;
        return p;
    }
}
```

Testing code

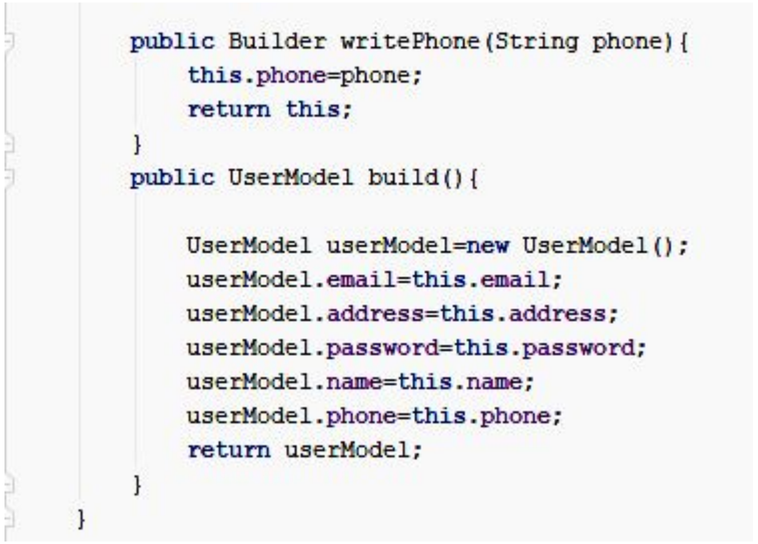


```java
product = new ProductModel.Builder()
        .writeProductName(productname.getText().toString())
        .writeProductPrice(productprice.getText().toString())
        .writeProductDescription(productdescription.getText().toString())
        .build();
```

Builder(Creational)

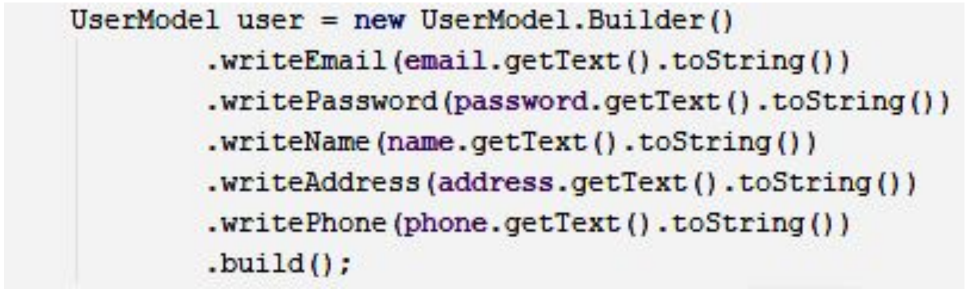


```
public class UserModel {
    private String email, password, name, address, phone;

    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;
        }
```



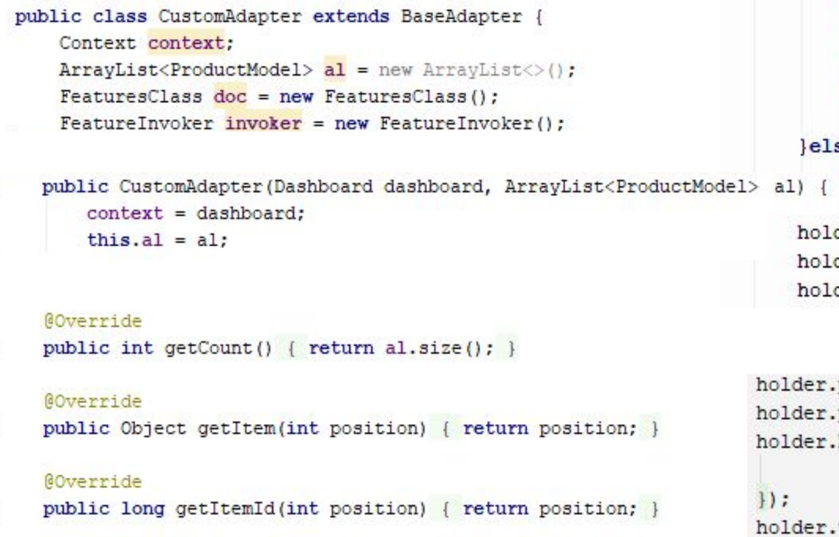
```
        public Builder writePhone(String phone){
            this.phone=phone;
            return this;
        }
        public UserModel build(){

            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;
        }
    }
```

Testing code

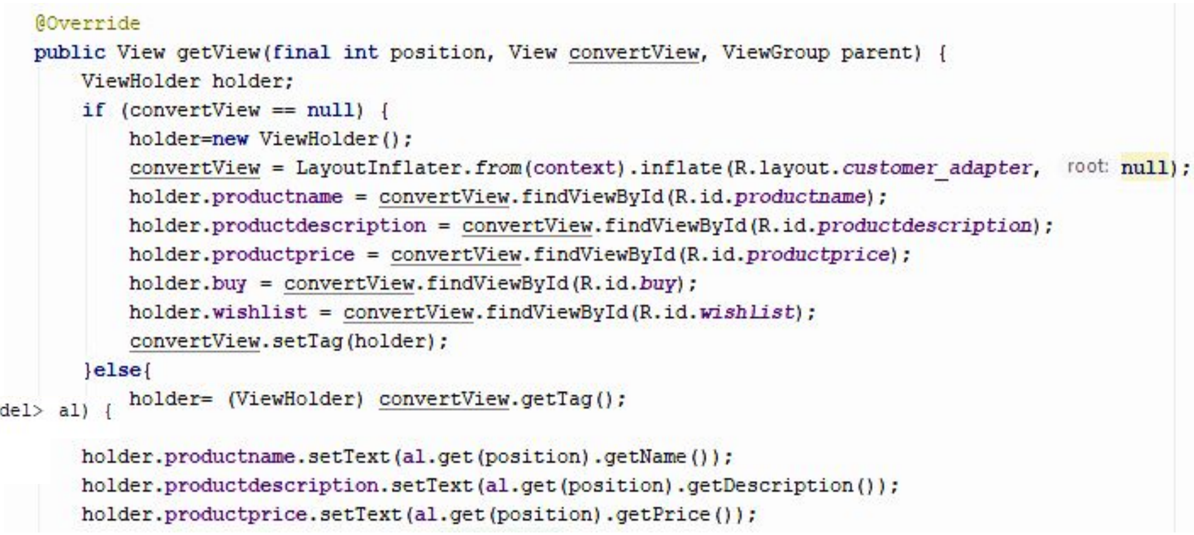


```
UserModel user = new UserModel.Builder()
        .writeEmail(email.getText().toString())
        .writePassword(password.getText().toString())
        .writeName(name.getText().toString())
        .writeAddress(address.getText().toString())
        .writePhone(phone.getText().toString())
       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

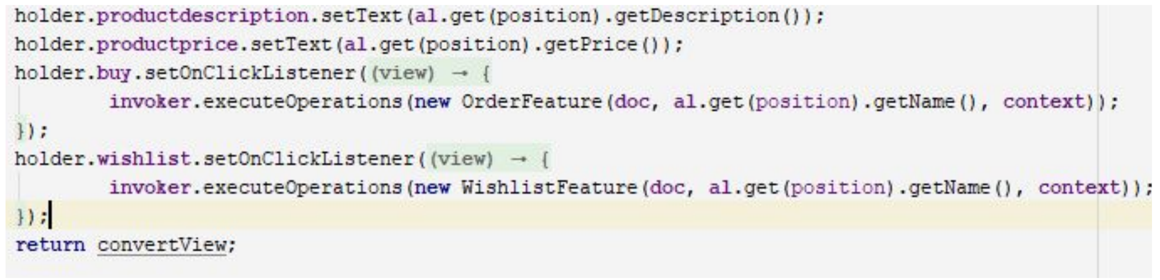
Adapter(Structural)



```
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; }
```

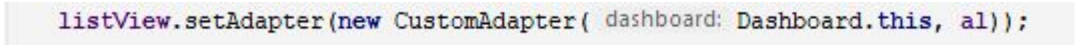


```
@Override
public View getView(final int position, View convertView, ViewGroup parent) {
    ViewHolder holder;
    if (convertView == null) {
        holder=new ViewHolder();
        convertView = LayoutInflater.from(context).inflate(R.layout.customer_adapter,  root: null);
        holder.productname = 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());
```



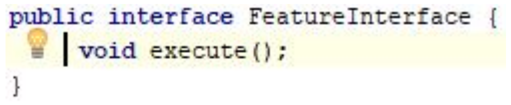
```
holder.productdescription.setText(al.get(position).getDescription());
holder.productprice.setText(al.get(position).getPrice());
holder.buy.setOnClickListener((view) → {
        invoker.executeOperations(new OrderFeature(doc, al.get(position).getName(), context));
});
holder.wishlist.setOnClickListener((view)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

<u>Testing code</u>

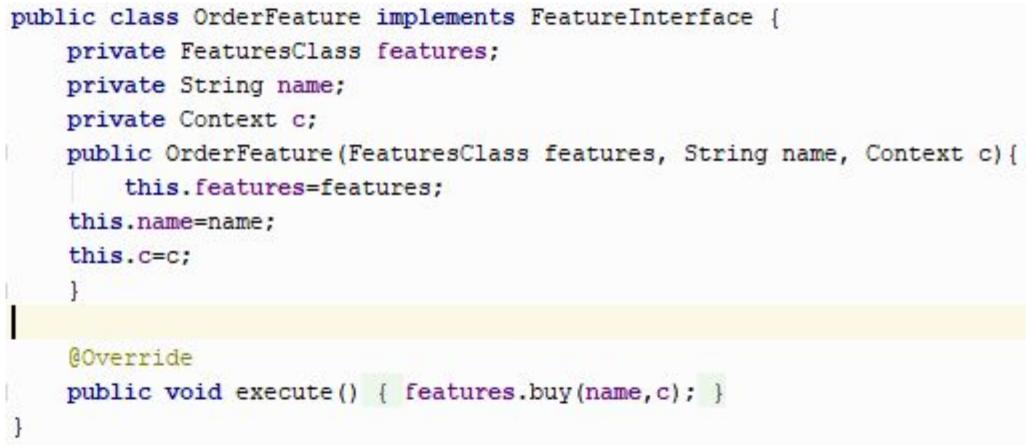


```
listView.setAdapter(new CustomAdapter( dashboard: Dashboard.this, al));
```

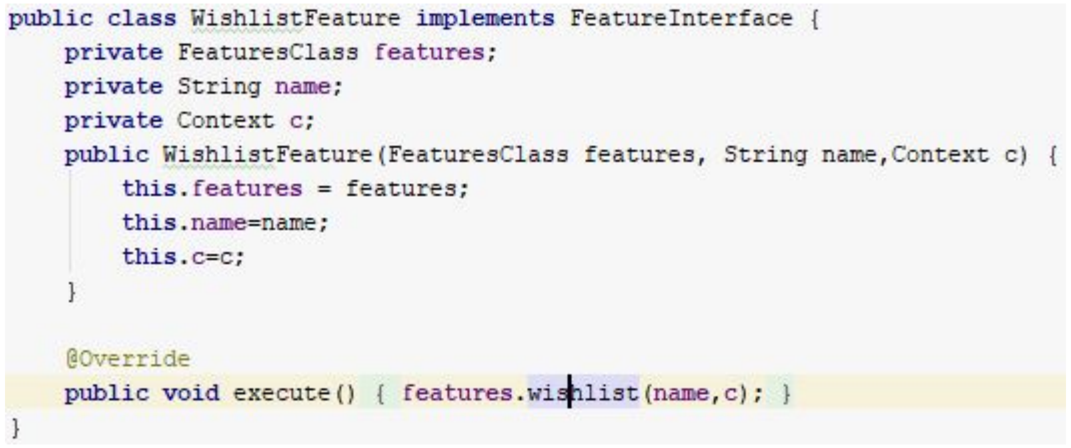
Command (Behavioural)



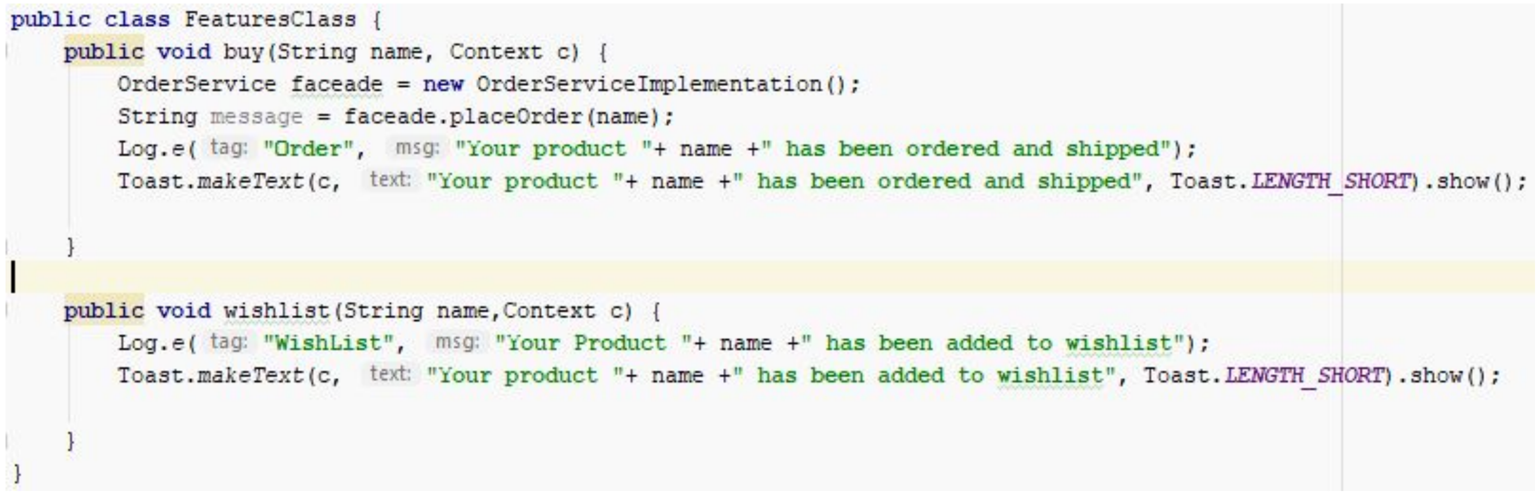
```java
public interface FeatureInterface {
    void execute();
}
```



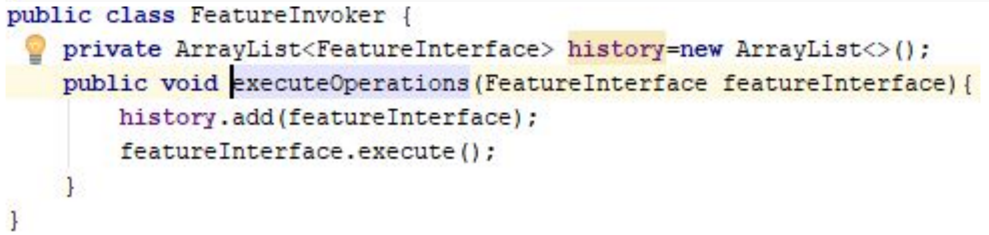
```java
public class OrderFeature implements FeatureInterface {
    private FeaturesClass features;
    private String name;
    private Context c;
    public OrderFeature(FeaturesClass features, String name, Context c){
        this.features=features;
    this.name=name;
    this.c=c;
    }

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



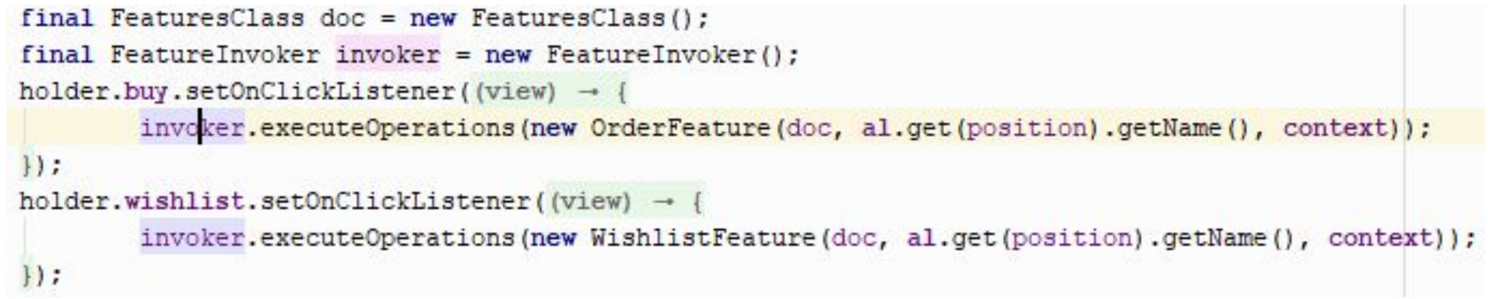
```java
public class WishlistFeature implements FeatureInterface {
    private FeaturesClass features;
    private String name;
    private Context c;
    public WishlistFeature(FeaturesClass features, String name,Context c) {
        this.features = features;
        this.name=name;
        this.c=c;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

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



```
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();

    }
}
```



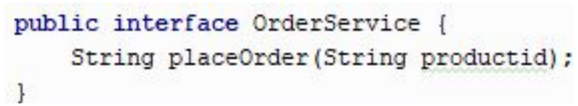
```
public class FeatureInvoker {
    private ArrayList<FeatureInterface> history=new ArrayList<>();
    public void executeOperations(FeatureInterface featureInterface){
        history.add(featureInterface);
        featureInterface.execute();
    }
}
```



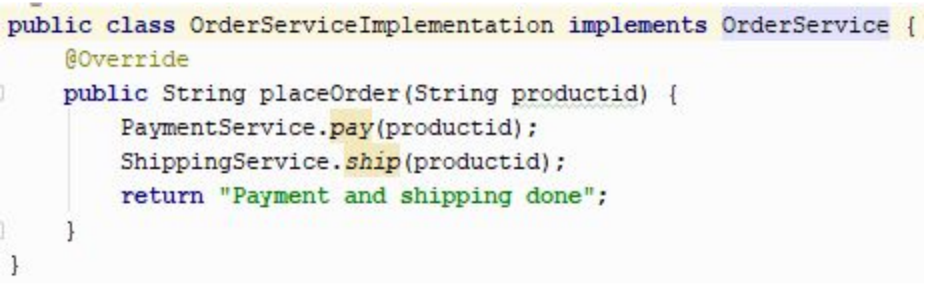
```
final FeaturesClass doc = new FeaturesClass();
final FeatureInvoker invoker = new FeatureInvoker();
holder.buy.setOnClickListener((view) → {
        invoker.executeOperations(new OrderFeature(doc, al.get(position).getName(), context));
});
holder.wishlist.setOnClickListener((view) → {
        invoker.executeOperations(new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

Testing code

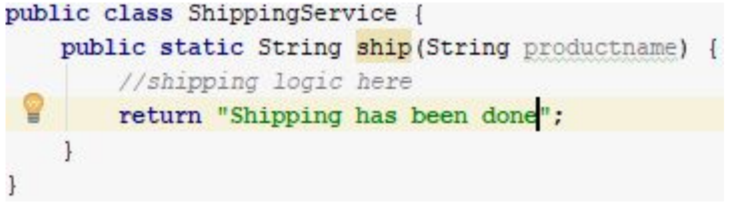
Facade(Structural)



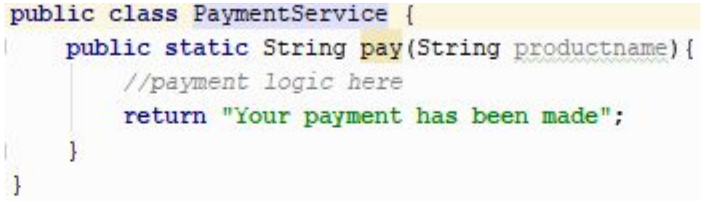
```java
public interface OrderService {
    String placeOrder(String productid);
}
```



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

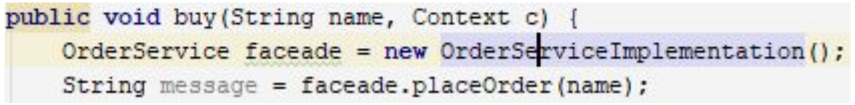


```java
public class ShippingService {
    public static String ship(String productname) {
        //shipping logic here
        return "Shipping has been done";
    }
}
```



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

Testing code



```java
public void buy(String name, Context c) {
    OrderService faceade = new OrderServiceImplementation();
    String message = faceade.placeOrder(name);
```

A little about anti-patterns

- patterns in software development that are considered bad programming practices.
- 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.

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() {}
	}

THANK YOU