Mini Project #2 & Final Report

Topic “Motorbike Showroom”

Student: Pazyl Yerbolat

Group: IS-1808K

2020

The purpose of system and its functionality

I decided to continue the Midterm Project.

My topic is about Motorbike Showroom.

The dealer sells motorcycles and buys from different manufacturers. When a new motorcycle arrives, the dealer notifies subscribers about the receipt of goods or discounts. The buyer can buy a motorcycle on credit.

In this project, I added:

1. Chain of Responsibility: used in the AuthService to process the request along the chain. I have two handlers ***BaseAuthHandler*** and ***RoleCheckHandler*** which are inherited from the abstract ***Handler*** class. First check whether there is such a user, if there is check the role of this user. If the user role is admin, send it to the admin panel. And If the customer then in the customer panel.

The "check" abcstarct function accepts two arguments password and login.

```
public abstract boolean check(String login, String password);
```

The "checkNext" function also accepts two arguments. If nextHandler is not truncated, it returns true because it has reached the end of the chain. If there is a next, it calls the next.check(login, password).

```
protected boolean checkNext(String login, String password) {
    if(next == null) {
        return true;
    }

    return next.check(login, password);
}
```

BaseAuthHandler check method: checks whether there is such a user.

```
@Override
public boolean check(String login, String password) {
    if (dbUser.checkUser(login, password)) {
        return checkNext(login, password);
    }

    return false;
}
```

RoleCheckHandler check method: defines the user role and sets the value in AuthService.

```
@Override
public boolean check(String login, String password) {
    if (dbUser.isAdmin(login)) {
        authService.setRole("Admin");
    } else {
        authService.setRole("Customer");
    }

    return checkNext(login, password);
}
```

***Note:** there is a change in the UML diagram. Added the authService fields to RoleCheckHandler class. I also added the MotoBikeShowroomApp and AuthService classes to the diagram.

2. Iterator Pattern: needed for sequentially traversing elements such as Order, Credit, Motorbike. I created three classes that implement the ***ICustomIterator*** interface such as ***DBBikeIterator***, ***DBCreditIterator***, ***DBOrderIterator***. ***IcustomIterableCollection*** interface has one method **createIterator()**.

```
package midka.iterators;

public interface ICustomIterableCollection {
    ICustomIterator createIterator();
}
```

DBBike, DBOrder, DBCredit implements **IcustomIterableCollection interface.**

DBBike:

```
@Override
public ICustomIterator createIterator() {
    return new DBBikeIterator(motorbikes);
}
```

DBOrder:

```
@Override
public ICustomIterator createIterator() {
    return new DBOrderIterator(orders);
}
```

DBCredit:

```
@Override
public ICustomIterator createIterator() {
    return new DBCreditIterator(credits);
}
```

The implementation of all iterators is similar only the element types differ.

DBBikeIterator

```java
package midka.iterators;

import midka.singleton.Pair;

import java.util.ArrayList;
import java.util.Map;

public class DBBikeIterator implements ICustomIterator {
    private ArrayList<Pair> bikes;
    private int position = 0;

    public DBBikeIterator(Map<String, Pair> bikes) {
        this.bikes = new ArrayList<>(bikes.values());
    }

    @Override
    public boolean hasNext() {
        return position < bikes.size();
    }

    @Override
    public Object next() {
        return bikes.get(position++);
    }
}
```

***Note:** I added void methods to the Iterator classes (UML diagram) where the createIterator() method is called().

3. Decorator Pattern: will send a message about a new receipt or discounts. By default, the message is sent to the customer's email, but you can add two more channels via SMS and via the customer's Facebook account. The customer can configure this at any time.

Interface ***Notifier*** one method send(String message).

```java
package midka.decorators;

public interface Notifier {
    void send(String message);
}
```

EmailNotifier and ***NotifierDecorator*** classes implements Notifier interface.

NotifierDecorator has two class ***FacebookDecorator*** and ***SMSDecorator***, which extends.

Decorator pattern call when action “add new motorbike” or “change price of motorbike”. AddMotorbikeListener and ChangePriceMotorbike listeners has method update().

AddMotorbikeListener

```java
@Override
public void update(String eventType) {
    if(motorbikeName.equals(eventType)) {
        String message = "We have received a new motorcycle: " +
motorbikeName + ".";
        notifierService.sendMessage(email, message);
    }
}
```

ChangePriceMotorbike

```java
@Override
public void update(String eventType) {
    if(eventType.equals(motorbikeName)) {
        int newPrice = dbBike.getMotorBike(motorbikeName).getPrice();
        message = motorbikeName + "\nNew price: " + newPrice;
        notifierService.sendMessage(email, message);
    }
}
```

***Note:** I created enum “NotifierEnum” and NotifierService.

4. Façade Pattern: this pattern is responsible for proper initialization of the necessary components (classes) for correct work the system. **MotoBikeShowroomApp** class has many fields and methods. But have only one public method "run" to start the system. And this method calls other private methods at the user's need.

The “run” method calls the first initDataBases() to initialize the databases.

```java
private void initDataBases() {
    dbBike = DBBike.getInstance();
    dbUser = DBUser.getInstance();
    dbOrder = DBOrder.getInstance();
    dbCredit = DBCredit.getInstance();
}
```

After database initialization calls the initServices, initHandler, initEventManager methods.

```java
private void initServices() {
    notifierService = NotifierService.getInstance();
    authService = AuthService.getInstance();
}
```

```
private void initHandler() {
    BaseAuthHandler baseAuthHandler = new BaseAuthHandler();
    RoleCheckHandler roleCheckHandler = new RoleCheckHandler();
    baseAuthHandler.setNext(roleCheckHandler);
    authService.setHandler(baseAuthHandler);
}

private void initEventManager() {
    manager = new EventManager();
}
```

***Note:** in the first diagram there were no methods for initializing services and a handler and another method for exporting to JSON Format.

5. Visitor: I created a ***JSONExportVisitor*** class that converts objects of the Order and Credit classes to JSON format. There is a ***Visitor*** interface that has a single name for the method but different arguments:

```
package midka.visitor;

import midka.files.Credit;
import midka.files.Order;

public interface Visitor {
    String visitOrder(Order order);
    String visitCredit(Credit credit);
}
```

The **JSONExportVisitor** class contains implementations of these methods.

```
@Override
public String visitOrder(Order order) {
    return "{" + "\n" +
                tab + "\t\"customerEmail\": \"" + order.getCustomerEmail() +
"\"," +  "\n" +
                tab + "\t\"motorbikeId\": \"" + order.getMotorbikeId() +
"\"," + "\n" +
                tab + "\t\"cost\": " + order.getCost() + "\n" +
            tab + "}";
}
```

And added interface ***IFile***.

```
package midka.files;

import midka.visitor.Visitor;

public interface IFile {
    String accept(Visitor visitor);
}
```

Order and **Credit** class contains implementation of accept methods.

Order:

```
@Override
public String accept(Visitor visitor) {
    return visitor.visitOrder(this);
}
```

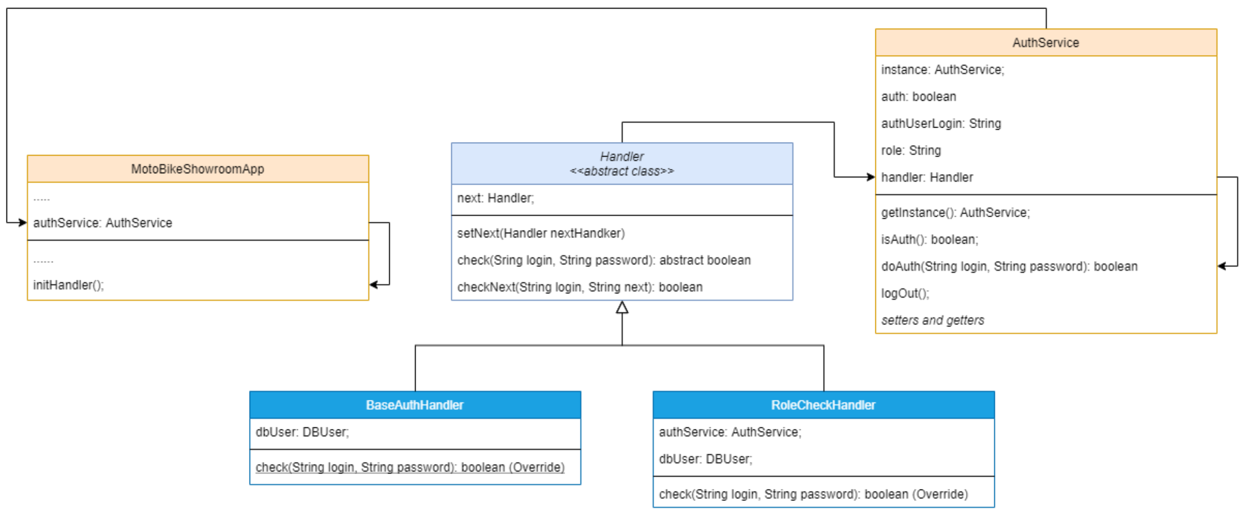
Credit:

```
@Override
public String accept(Visitor visitor) {
    return visitor.visitCredit(this);
}
```

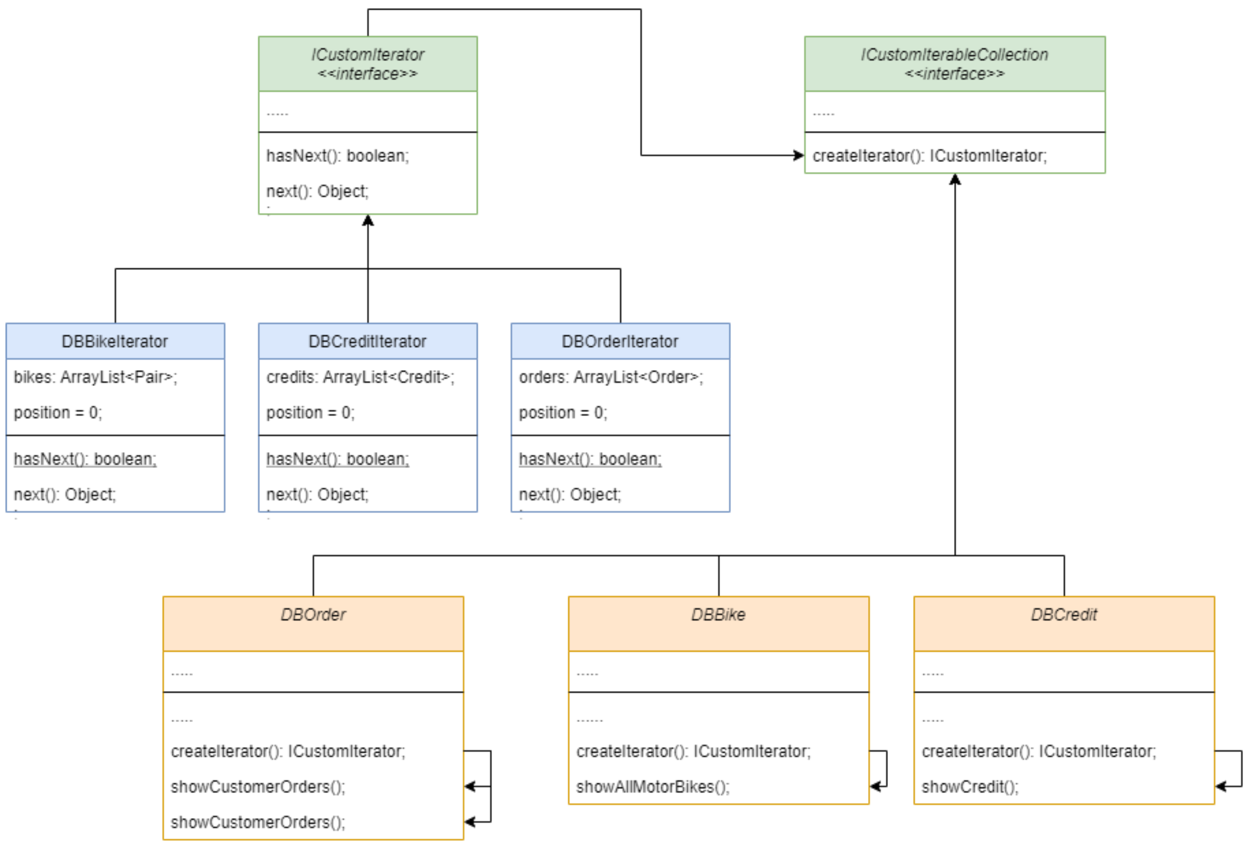
***Note:** I didn't plan to use this pattern in the beginning.

Diagrams

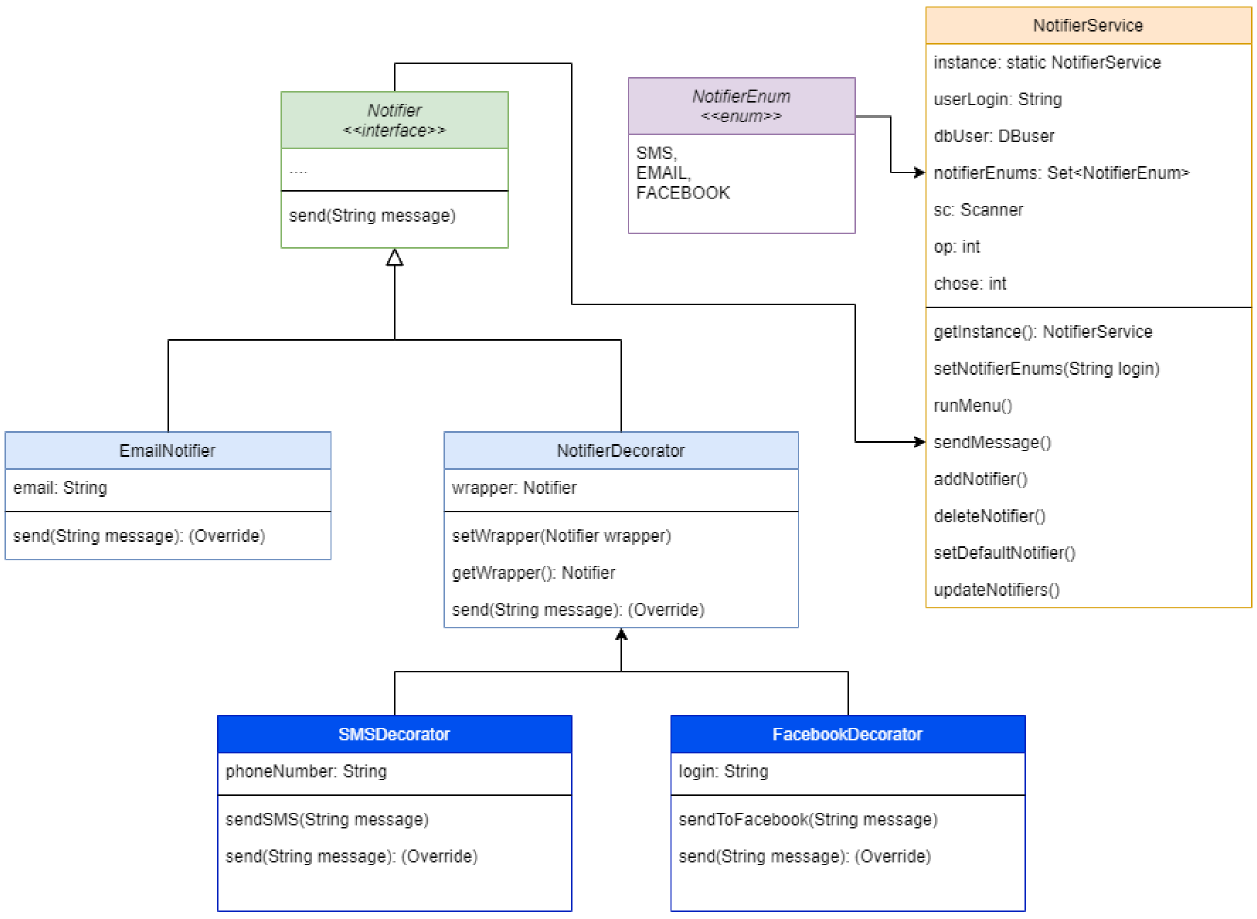
1. Chain of Responsibility.


MotoBikeShowroomApp
.....
authService: AuthService
......
initHandler();
Handler
<<abstract class>>
next: Handler;
setNext(Handler nextHandker)
check(Sring login, String password): abstract boolean
checkNext(String login, String next): boolean
AuthService
instance: AuthService;
auth: boolean
authUserLogin: String
role: String
handler: Handler
getInstance(): AuthService;
isAuth(): boolean;
doAuth(String login, String password): boolean
logOut();
setters and getters
BaseAuthHandler
dbUser: DBUser;
check(String login, String password): boolean (Override)
RoleCheckHandler
authService: AuthService;
dbUser: DBUser;
check(String login, String password): boolean (Override)

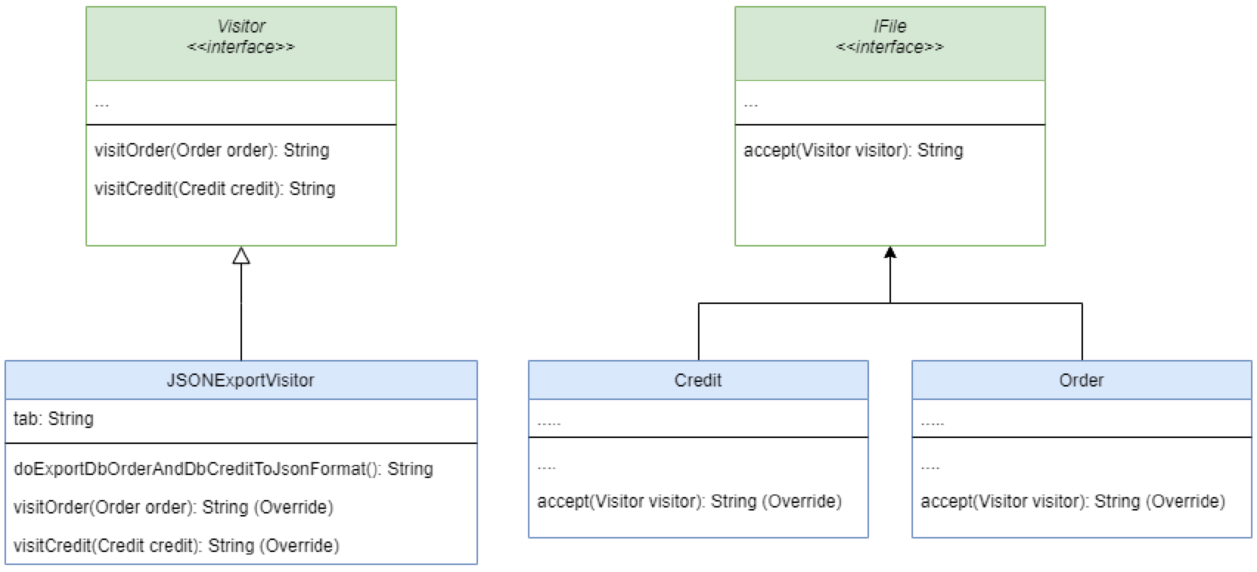
2. Iterator.



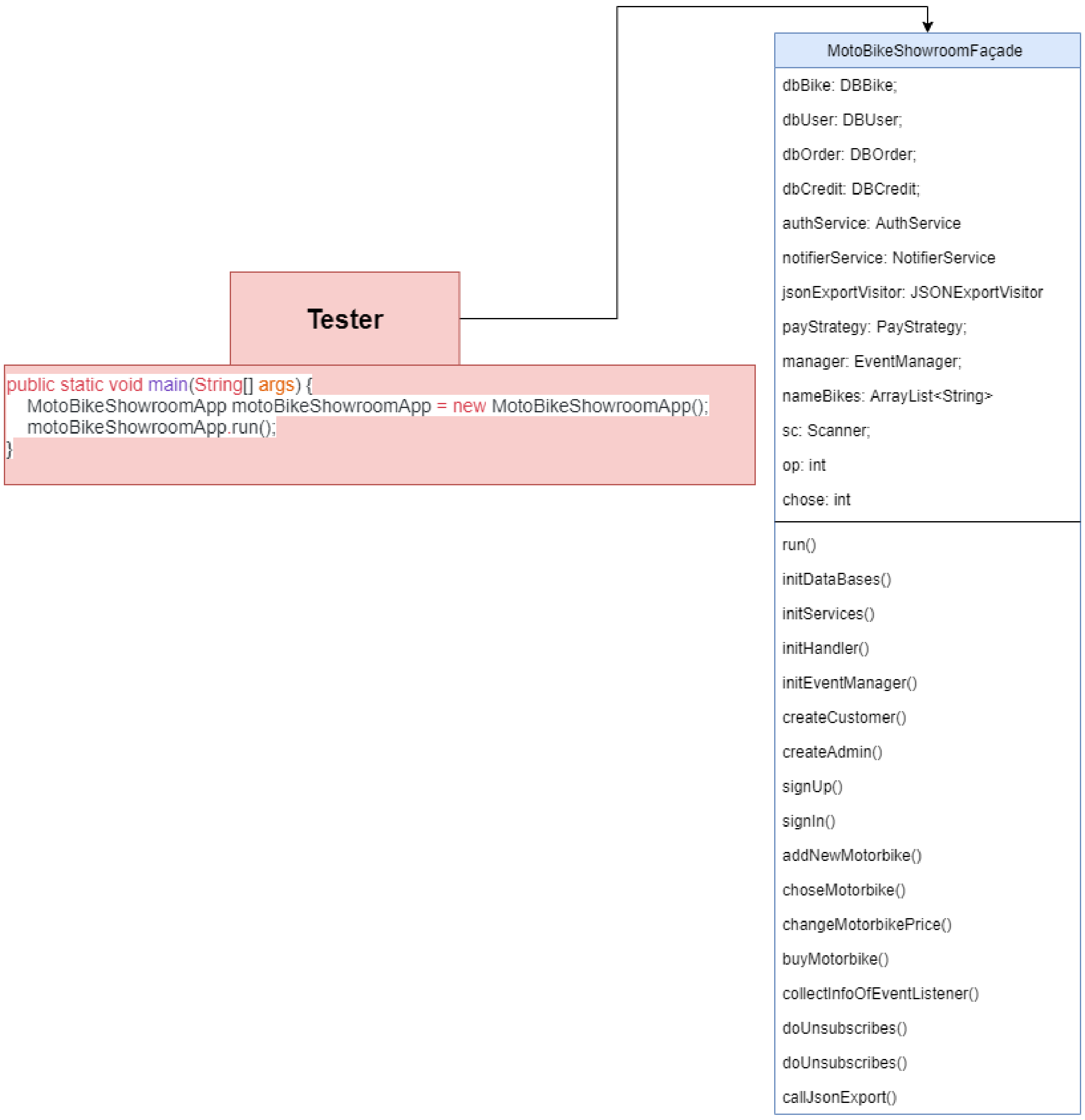
3. Decorator.



4. Visitor


Visitor
<<interface>>
...
visitOrder(Order order): String
visitCredit(Credit credit): String
IFile
<<interface>>
...
accept(Visitor visitor): String
JSONExportVisitor
tab: String
doExportDbOrderAndDbCreditToJsonFormat(): String
visitOrder(Order order): String (Override)
visitCredit(Credit credit): String (Override)
Credit
.....
....
accept(Visitor visitor): String (Override)
Order
.....
....
accept(Visitor visitor): String (Override)

5. Facade



```
public static void main(String[] args) {
    MotoBikeShowroomApp motoBikeShowroomApp = new MotoBikeShowroomApp();
    motoBikeShowroomApp.run();
}
```