**Used Design patterns:**

1. Abstract factory for (Service, Form, Handler):

We used the abstract factory pattern since its purpose is to create a family of different related objects, therefore we used it to create the needed service with its form and handler because they are different objects which are related to each other

**Note**: we made the factories to model the different types of services (Internet Factory, Mobile Factory, ….). we thought about making a factory for each specific service provider type.

For Example a factory for Vodafone mobile service and another for Orange mobile service.

But we found this design to be very explosive because we would have to add 3 new classes for each specific factory which brought the total of factory-related classes to nearly 40 classes.

We also took into consideration that type-related services nearly have the same handler and form, so we made abstract factories for the type of the services, if a new service was added it would go under one of them or a new factory type will be created, this also allows us to make service-specific forms in the future if wanted (in case its form was different from others under the same type)

1. Factory method for (transaction controller):

We used the factory method in the transaction controller and its children because it allows subclasses to alter the type of objects made, so we made it to override only certain parts of the creation algorithm to make different types of transaction controllers to handle different types of transactions (payment, refund, wallet).

So they override the makeTransaction ( ) , and createTransaction( ) functions but leave the rest of the processes the same as their parent.

**Note:** since the factory method is a specialization of template method, therefore this could be considered a template but since the goal of these controllers is to create different types of transactions, so we saw that it would be more fit for it to be a factory method due to its creation function

1. Strategy pattern:

We have used the Strategy pattern in landline receipt requirement, so if the user choosed to get receipt monthly, a new object will be created from type MonthlyReceipt in FactoryInterface and will be sent to setReceipt(Receipt r). So, using polymorphism, the receipt object from type Receipt in the Landline object that called the method will be set to be: receipt = new r(). And with this pattern, We can create any number of strategies to send the receipt to the user with out modifying the original code.

We have used the Strategy pattern in choosing the payment method, so the form will ask the user to choose a way to pay with, then a new object of this class will be created and returned to the handler (if the user choosed to pay with wallet, return new Wallet() ), then the handler will take this returned object of type PaymentController (the father of CreditCard, Wallet and cash if the service support paying cash), and the handler will call setTransaction, and every class shall implement this method with its own way (According to Open Closed principle).

1. Decorator pattern for (Discount):

We chose the decorator pattern to implement and model the discount.

Since either a service or a user has discounts on them added by the admin.

So we made added discount to wrap the existing discount on either of them .

taking into consideration that users and services are initialized with a discount of 0 amount $.

So we

GitHub link:- https://github.com/RamezTalaat/SDA-fawry-project

Project contributors:

Name: Khaled Ahmed Sayed Hashem ID : 20206019

Name: Hazim Emam Mohamed Ali ID : 20206015

Name: Omar Rafaat Ali ID : 20206041

Name: Ramez Ehab Talaat Riad ID : 20206025