**Gebze Technical University**

**Computer Engineering**

# Object Oriented Analysis and Design

## CSE443

## Midterm Project

**Can BEYAZNAR**

**161044038**

# 1) Part 1

## A) Solution

For Part 1, the abstract factory design pattern was used. The features of the phone models and the features that vary according to the Region were separated. Thus, the fixed features of the phones and the changing features were separated from each other. First of all, I created an interface for each variable component. For example, components such as display, battery, camera. features like this in Turkey, I created for them to interface with different characteristics are sold on a global and Euro. Then I created subclasses for each component's interface that they might have. For example, the display is used as a 32-bit market in Turkey. That's why I created the display_32bit class by implementing the DisplayInterface interface. There is only toString () method in this class and it returns "32 bit" string. Thus, when a new market is added (such as the Asian market), we can easily add the features of this market with this method. I applied the example I gave according to the features of the phone. In addition, if there is any change in the markets, this problem will be solved with low maintenance costs. After creating our classes for features that vary according to the market, we need to create a structure to receive these variable properties.

First of all, I created the PhoneHardwareFactory interface as phone features change according to each market. Here, methods have been created for calling 6 features of the phone. Thus, classes that will derive from this interface will be able to access the phone features they want in their market by using this interface. Then I created classes for the regions that use the PhoneHardwareFactory interface. They return the phone features they want in their markets with these 6 methods. And so the structure of the changing properties is completed.

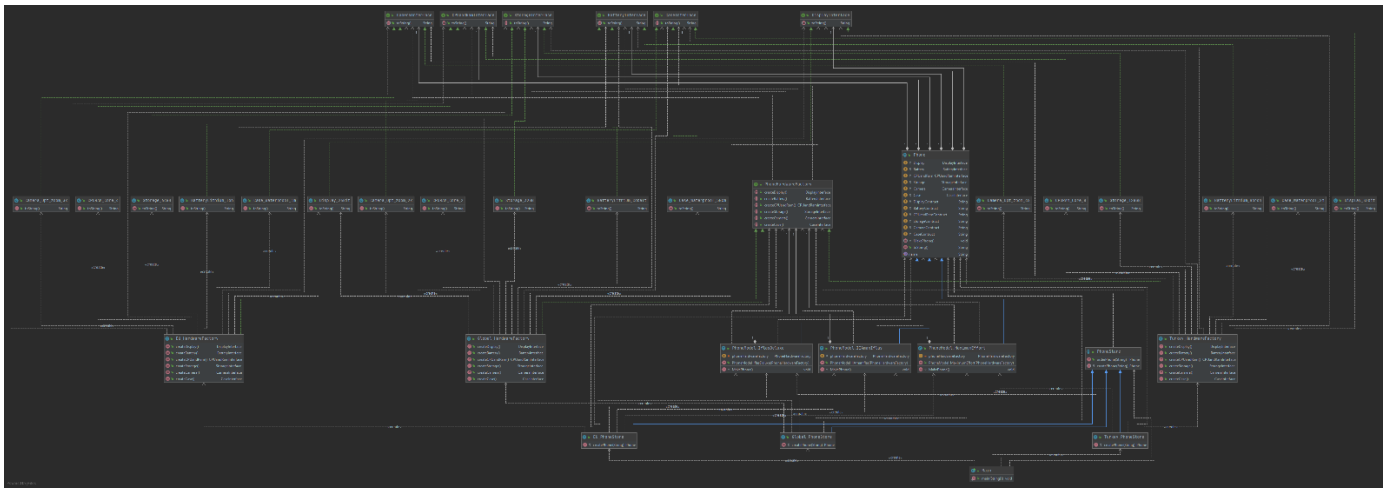
Then I created the Phone class to gather the features of these phones in one place. This is of type abstract class. It holds 13 protected parameters in total. 6 of them are features that vary by region, 6 of them are features that vary according to the phone model, and 1 of them is the name of the phone model. With this abstract class, we can get the phone we want to produce. The 3 phone models we have will write their methods using this class, and the phone will be created with the features it has. After creating this abstract class, I created the classes that we will use for our 3 phone models. They call the properties that vary according to the region they are in in the makephone () method, using the PhoneHardwareFactory interface. And it defines the features it has according to its model. And so the phone is produced.

Finally, we need to sell these phones according to their region and phone model. To do this, we need to create a store class. By creating subclasses connected to this interface, we can sell our phones by region. Our PhoneStore abstact class receives a phone order from the customer and produces this phone. The createPhone method is defined as abstract. Thus, the stores that are specific to the regions can produce phones according to their different characteristics. By creating 3 classes that derive from the PhoneStore class, we can produce phones in every region. And the abstract factory design pattern we used is completed.

## B) UML Diagram

The UML diagram is as follows:

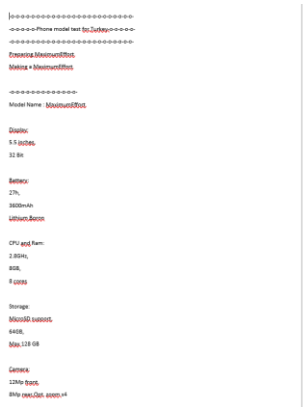
**NOTE**: The diagram may not appear clearly in the report as there are many classes. There is a screenshot of the uml diagram in the folder I created for Part 1.

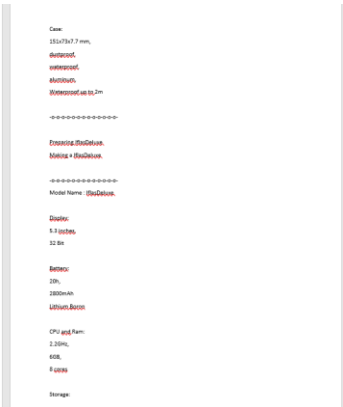


## C) Output

The printout of all phone models for each region is as follows:

The net version of Output is in the file abstractfactory_output.txt.





CPU and Ram:
2.2GHz,
4GB,
8 cores

Storage:
MicroSD,
support, 16GB,
Max 128 GB

Camera:
8Mp front,
5Mp rear,
Opt. zoom,x4

Case:
143x69x7.3 mm
waterproof,
plastic,
Watercroof up to 2m

-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-

-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-
-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-
-o-o-o-o-o-Phone model test for EU-o-o-o-o-o-
-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-
Preparing MaximumEffort
Making a MaximumEffort

-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-

Model Name : MaximumEffort

Display:
5.5 inches,
24 Bit

Battery:
27h,
3600mAh
Lithium Ion

CPU and Ram:
2.8GHz,
8GB,
4 cores

Storage:
MicroSD support,
64GB,
Max 64 GB

Camera:
12Mp front,
8Mp rear,Opt. zoom,x3

Case:
151x73x7.7 mm,
dustproof,
waterproof,
aluminum,
Watercroof up to 1m

-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-

-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-

Preparing IfixDeluxe,
Making a IfixDeluxe,

-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-
Model Name : IfixDeluxe,

Display:
5.3 inches,
24 Bit

Battery:
20h,
2800mAh
Lithium Ion

CPU and Ram:
2.2GHz,
6GB,
4 cores

Storage:
MicroSD support,
32GB,
Max 64 GB

Camera:
12Mp front,
5Mp rear,Opt. zoom,x3

Case:
149x73x7.7 mm
waterproof,
aluminum,
Watercroof up to 1m

-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-

Preparing I-I-Aman-Ifixs
Making a I-I-Aman-Ifixs

-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-
Model Name : I-I-Aman-Ifixs

Display:
4.5 inches,
24 Bit

Battery:
16h,
2000mAh,
Lithium Ion

CPU and Ram:
2.2GHz,
4GB,
4 cores

Storage:
MicroSD,
support, 16GB,
Max 64 GB

Camera:
8Mp front,
5Mp rear,
Opt. zoom,x3

Case:
143x69x7.3 mm
waterproof,
plastic,
Watercroof up to 1m

-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-

-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-
-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-
-o-o-o-o-o-Phone model test for Global-o-o-o-o-o-
-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-
Preparing MaximumEffort
Making a MaximumEffort

-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-
Model Name : MaximumEffort

Display:
5.5 inches,
24 Bit

Battery:
27h,
3600mAh
Lithium Cobalt

CPU and Ram:
2.8GHz,
8GB,
2 cores

Storage:
MicroSD support,
64GB,
Max 32 GB

Camera:
12Mp front,
8Mp rear,Opt. zoom,x2

Case:
151x73x7.7 mm,
dustproof,
waterproof,
aluminum,
Watercroof up to 50cm

-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-

Preparing IfixsDeluxe,
Making a IfixsDeluxe,

-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-

Model Name : IfixsDeluxe,

Display:
5.3 inches,
24 Bit

Battery:
20h,
2800mAh
Lithium Cobalt

CPU and Ram:
2.2GHz,
6GB,
2 cores

Storage:
MicroSD support,
32GB,
Max 32 GB

Camera:
12Mp front,
5Mp rear,Opt. zoom,x2

Case:
149x73x7.7 mm
waterproof,
aluminum,
Watercroof up to 50cm

-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-

Preparing I-I-Aman-Ifixs
Making a I-I-Aman-Ifixs

-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-
Model Name : I-I-Aman-Ifixs

Display:
4.5 inches,
24 Bit

Battery:
16h,
2000mAh,
Lithium Cobalt

CPU and Ram:
2.2GHz,
4GB,
2 cores

Storage:
MicroSD,
support, 16GB,
Max 32 GB

Camera:
8Mp front,
5Mp rear,
Opt. zoom,x2

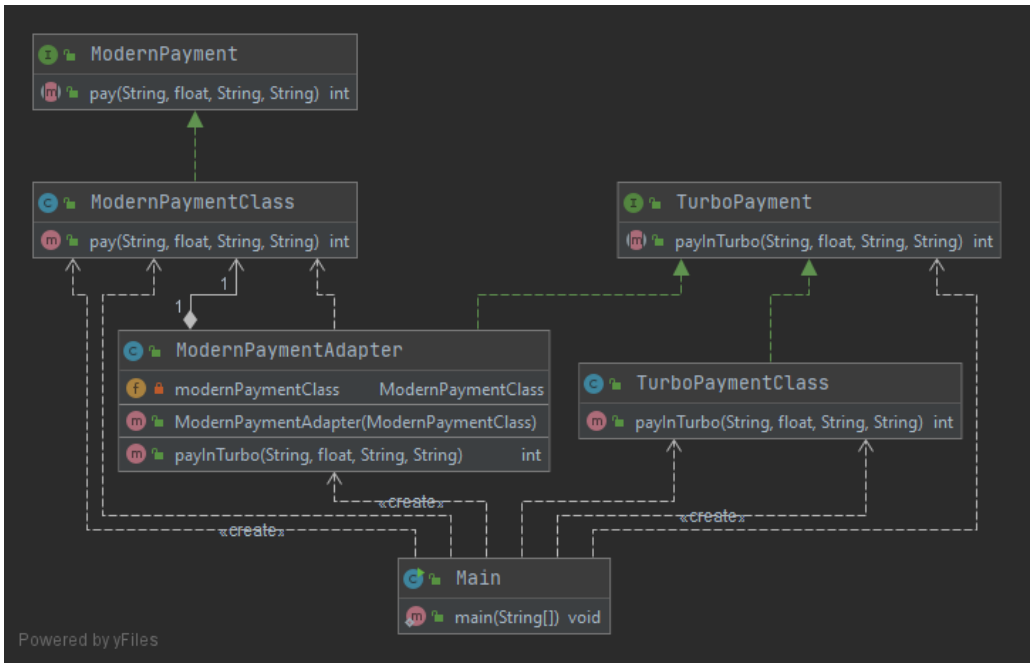
Case:
143x69x7.3 mm
waterproof,
plastic,
Watercroof up to 50cm
-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-
-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-

## 2) Part 2

### A) Solution

I used Adapter design pattern method in Part 2. Since we need to integrate the previously used TurboPayment interface into the modern payment interface without making any changes, the adapter is a suitable method for us. First of all, the TurboPayment and ModernPayment interfaces given in the pdf of the homework are defined. No changes were applied to them. Then ModernPaymentClass was defined in order to implement the functions in the ModernPayment interface. Only the println method is called into the pay () method. Then the ModernPaymentAdapter class was created. This class will use the ModernPaymentClass class but will use the functions of the TurboPayment interface. Thus, the numerator method belonging to the ModernPayment interface is called in the payInTurbo method. And thus, the adapter design pattern has been implemented. Finally, the TurboPaymentClass class has been created in order to run tests in main. This class implements the old method TurboPayment interface.

### B) UML Diagram



### C) Output

```java
public class Main {
    public static void main(String[] args) {

        System.out.println("-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-");
        System.out.println("pay test with modernPaymentClass");
        ModernPaymentClass modernPaymentClass = new ModernPaymentClass();
        modernPaymentClass.pay( cardNo: "545", amount: 100, destination: "Isbank", installments: "EFT");
        System.out.println("-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-");

        System.out.println("payInTurbo test with turboPaymentClass");
        TurboPaymentClass turboPaymentClass = new TurboPaymentClass();
        turboPaymentClass.payInTurbo( turboCardNo: "700", turboAmount: 23, destinationTurboOfCourse: "AABANK", installmentsButInTurbo: "EFT");
        System.out.println("-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-");


        System.out.println("payInTurbo test with modernAdapter");
        TurboPayment modernAdapter = new ModernPaymentAdapter(modernPaymentClass);
        modernAdapter.payInTurbo( turboCardNo: "630 ", turboAmount: 29, destinationTurboOfCourse: "Vakif", installmentsButInTurbo: "EFT");
        System.out.println("-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-");
    }
}
```

```
-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-
pay test with modernPaymentClass
Paying with pay method (Location: ModernPaymentClass)
-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-
payInTurbo test with turboPaymentClass
Paying with payInTurbo method (Location : TurboPaymentClass)
-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-
payInTurbo test with modernAdapter
Converting turbo to modern payment with adapter class
Paying with payInTurbo method (Location: ModernPaymentAdapter)
Paying with pay method (Location: ModernPaymentClass)
-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-
```

## 3) Part 3

### A) Solution

In Part 3, we were asked to make a database engine. In order to apply various operations to the database, there must be commands such as SELECT, ALTER, UPDATE. A command design pattern should be used to gather these commands under a single roof. If it is necessary to explain the structure I used, first I created the classes of these commands so that these commands can be called by the user. There is no difference in the properties of these 3 command classes. For example, if it is necessary to mention the class of the select command (SelectClass), I have kept 2 private values in total. One of them is the code we will get from the user (such as "SELECT X in Y"), and the other is the value that comes to us from the database after running this code. I have given this value as an example for now, but if you need to think ahead, a database class can be created to better show this data in the database. Then I created a method named select_operation () to run this code that the user wants to implement. This method takes a code and a database name from the user. Thus, the label that the user wants to access is returned by applying the operations in it. And the code specified by the user runs on the database they want. These operations mentioned for the SelectClass class are the same for UPDATE and ALTER. These methods applied are quite simple. There

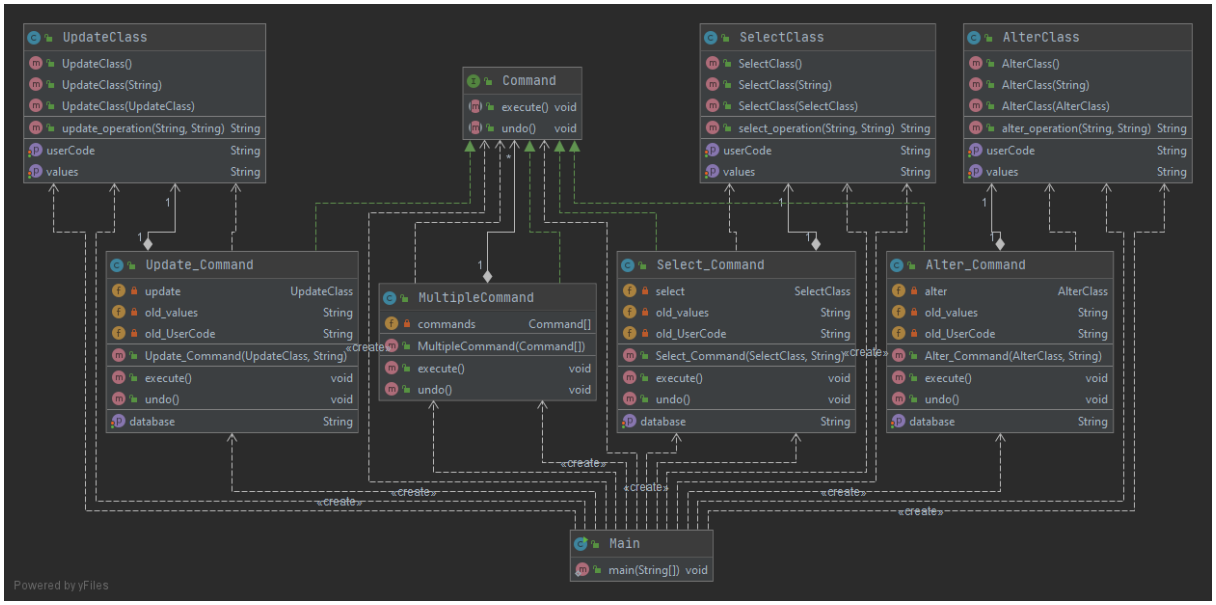
are no obvious procedures in any way. The main idea is to create the framework of the project to explain the solution to the problem in a simple way. We created our classes to call the desired action in the user section.

Next is to collect and run these processes in a common place. For this, we need to run these processes with the same methods, regardless of their type. To do this, we create the Command interface. There are only execute () and undo () methods in this interface. Thus, we can run our operations in a similar way by calling them in classes derived from this interface, and undo our operations. Here, too, there are no major differences between these 3 classes derived from the Command interface. If it is necessary to explain the Select_Command class, it has 4 values in total. One is the SelectClass class from the user. Thus, when the user wants to implement this process, we can take this information in the Select_Command class and apply this process in the execute () method. The second is old_values and old_UserCode. These parameters will be used if the user wishes to undo the action they have applied. Our old data are backed up in these parameters and these old values can be restored with the undo () method. old_UserCode is the old command that the user applied, and old_values is the database data the user obtained in the old transaction. And finally the Database parameter is kept. Thus, the user can edit and change the database to which he / she wants to apply the transaction at any time. As for the operations that I apply in the methods, in the execute () method, I first back up the old values and apply the select_operation () method according to the database specified by the user. In the undo () operation, I assign the old data as current data and terminate the method. These operations I explained for the Select_Command class are the same for the remaining 2 classes. With what I have explained so far, the user can apply his database operations as he wishes.

If the user writes an incorrect code, all the actions he has done must be undone. In order for this to be achieved, I need a class that collects all commands in one place. That's why I created the MultipleCommand class. The MultipleCommand class uses the methods of the Command interface. And it holds an array of Command type inside. Thus, using this array in the execute () method, it runs all the commands in the array from beginning to end. In the undo () method, it applies the undo () method to the first command on the last command calling the execute () method. Thus, the sequence does not break and the program is not damaged.

I applied this solution by making minor changes in the Command Design pattern. As mentioned earlier, I created the framework of the program to better explain the solution applied to the problem.

## B) UML Diagram




## C) Output

## Main Code

```
System.out.println("-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-");
System.out.println("Select command test with undo");
System.out.println("Operations : ");
String User_command = "SELECT Accounts";
SelectClass selectClass_v1 = new SelectClass(User_command);
Select_Command select_command = new Select_Command(selectClass_v1, Database: "BANK-A");
select_command.execute();
select_command.undo();
select_command.execute();
System.out.println("-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-");

System.out.println("MultipleCommand Class Test");
int command_listsize = 3;
Command[] command_list = new Command[command_listsize];
selectClass_v1 = new SelectClass(User_command);
command_list[0] = new Select_Command(selectClass_v1, Database: "BANK-A");

User_command = "UPDATE Accounts Accounts+1";
UpdateClass updateClass = new UpdateClass(User_command);
command_list[1] = new Update_Command(updateClass, Database: "BANK-B");

User_command = "ALTER Accounts newAccounts";
AlterClass alterClass = new AlterClass(User_command);
command_list[2] = new Alter_Command(alterClass, Database: "BANK-C");

MultipleCommand multipleCommand = new MultipleCommand(command_list);
multipleCommand.execute();
multipleCommand.undo();
```

### Terminal

```
-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-
Select command test with undo
Operations :
Getting the label's values (SELECT)
Your database name is : BANK-A
Your code is : SELECT Accounts in BANK-A
Making undo in Select_Command class
Your values before undo (Select_Command) : SELECT Accounts( -> values after SELECT )
Your current values after undo (Select_Command) : EMPTY_VALUES
Getting the label's values (SELECT)
Your database name is : BANK-A
Your code is : SELECT Accounts in BANK-A
-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-o-
MultipleCommand Class Test
Getting the label's values (SELECT)
Your database name is : BANK-A
Your code is : SELECT Accounts in BANK-A
Updating the label's values (UPDATE)
Your database name is : BANK-B
Your code is : UPDATE Accounts Accounts+1 in BANK-B
Alter the label's values (ALTER)
Your database name is : BANK-C
Your code is : ALTER Accounts newAccounts in BANK-C
Making undo in Update_Command class
Your values before undo (Update_Command) : ALTER Accounts newAccounts( -> values after ALTER )
Your current values after undo (Update_Command) : EMPTY_VALUES
Making undo in Update_Command class
Your values before undo (Update_Command) : UPDATE Accounts Accounts+1( -> values after UPDATE )
Your current values after undo (Update_Command) : EMPTY_VALUES
Making undo in Select_Command class
Your values before undo (Select_Command) : SELECT Accounts( -> values after SELECT )
Your current values after undo (Select_Command) : EMPTY_VALUES

Process finished with exit code 0
```

## 4) Part 4

### A) Solution

In Part 4, we are asked for a program that implements DFT and DCT methods. First, the inputs should be taken from the file, then the method desired by the user should be applied, and finally the results should be written to an output file specified by the user. While performing these processes, all the steps followed are the same. Therefore, it would be appropriate to apply a template design pattern for this problem. First of all, an abstract class named DiscreteTransformTemplate is created to gather these two methods in one place. In this class, there are solve, readInputFromFile, makeTransform, writeToFile, Display_TimeofExecution and userWants_Display_TimeofExecution methods. The solve method is a method that calls all other methods. Thus, the user will only call the solve method and this class will do the remaining operations. When implementing the remaining methods, methods common to DFT and DCT are implemented here (eg for reading and writing from a file). Different methods for DFT and DCT were created abstractly. Thus, DFT and DCT classes can write these operations according to themselves. For example, there are differences between the two classes because there are calculation operations in the makeTransform method. So DFT and DCT classes should write them differently.

In the DFT class, the abstract methods specified are implemented. Also, if the user wants to learn the time of execution value, he / she receives input from the user at the end of the program. This value is calculated with the time from the beginning to the end of the makeTransform method. This value is written to the terminal in milliseconds. I cast it to double because it was a very low value. Then, if the user requests this value, I print it on the screen. If we come to the calculation part, the method was applied according to the formula found on the wiki page in the pdf. And this method returns ArrayList. ArrayList was used to get the values, calculate and write them to the file.

In DCT class, calculations are made according to the formula specified in pdf. The DCT-II formula on the Wikipedia page has been applied. In this class, the ArrayList is used as in DFT.

**Input and Output Format**

When taking values from the file coming from the user, only the first line is taken. Therefore, only the first line of a file with more than one line will be accepted as input. The file should contain numbers only (no characters). And there should be tabs between numbers. Data received in DFT must contain real number and imaginary number. Therefore, when the input file is taken for DFT, the data are taken by two in the calculation section.

For example, if 1 2 3 4 in the input file, our data for DFT will be 1 + 2i and 3 + 4i. Please edit your input file according to this format. For DCT, it must contain only real number and must be tab between numbers.

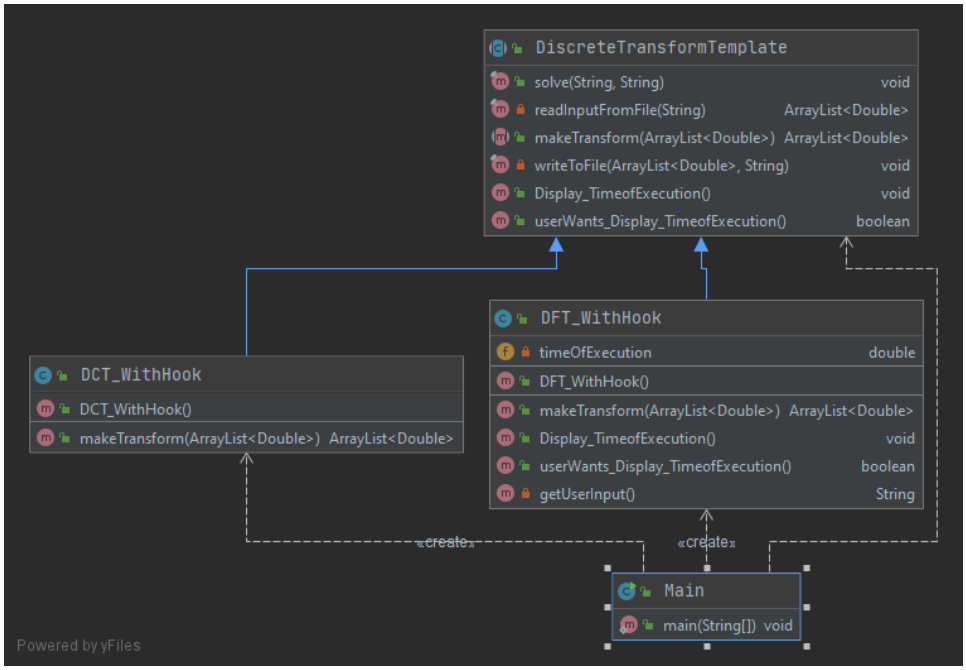
In the output file, the output will only consist of numbers. The result for DFT will be like the format mentioned in the input. In other words, since numbers are complex, they must be accepted in pairs. For example, get the numbers 1 2 3 4 in the result output. The result should be considered as 1 + 2i, 3 + 4i.

You need to give the input file path from the terminal.

**Output file for DFT -> “DFT_Output.txt”**

**Output file for DCT -> “DCT_Output.txt”**

## B) UML Diagram




## C) Output

## İnput file

```
0.672957 -0.453061 -0.835088 0.980334 0.972232 0.640295 0.791619 -0.042803 0.282745 0.153629 0.939992 0.588169 0.189058 0.461301 -0.667901 -0.314791
```

## DFT Result

```
Do you want to display time of execution? (y/n) y
Time of Execution : 0.2305 ms

Process finished with exit code 0
```

```
2,346  2,013  -1,248  -1,082  1,721  -1,382  2,968  0,315  1,888  -0,409  2,387  -1,698  -2,132  -1,42  -2,546  0,038
```

## DCT Result

```
4,359  0,913  -2,764  0,142  -2,498  1,317  2,635  1,919  1,624  1,899  1,386  -0,59  0,461  -2,432  -0,07  0,417
```