CSE443 - Object Oriented Analysis and Design

Midterm Report

Melih Yabaş

161044072

Teacher: Erchan Aptoula

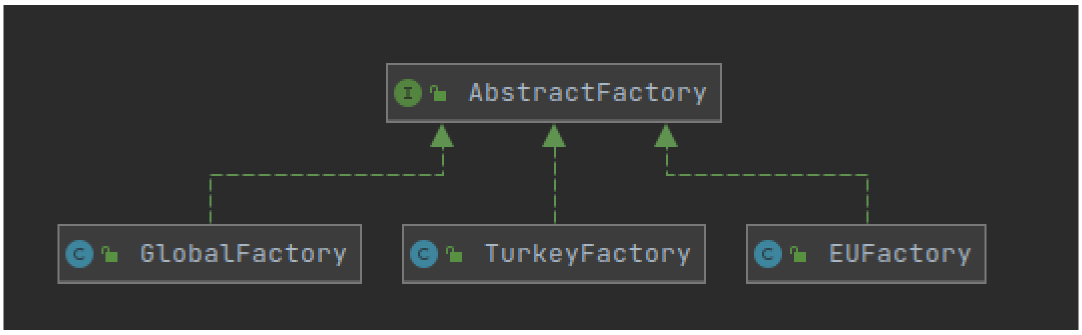
Asistant: Abdullah Salih Bayraktar

Part 1

There is a company named as Iflas-Technologies Ltd
There are 3 kinds of phone model in the company (MaximumEffort, IflasDeluxe and I-I-Aman-Iflas). And these models have different specifications to different markets.

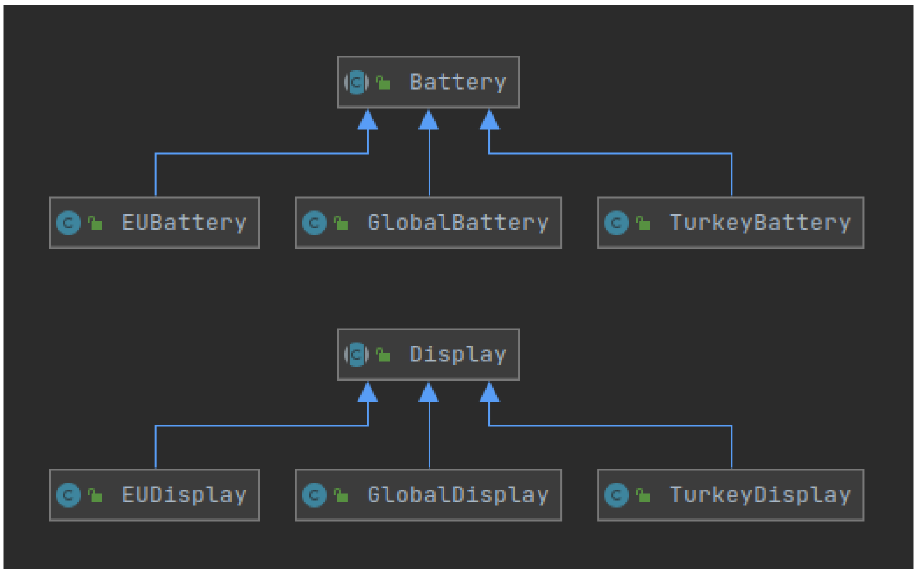
There are 3 regions where the company operates: Turkey, Europe, Global.

We need to create an Abstract Factory Design Pattern.
First, we need to create an Abstract Factory Interface. It keeps component attach methods that mandatory for all factories.
These components are; CPU, Ram, Display, Battery, Storage, Camera and Case.



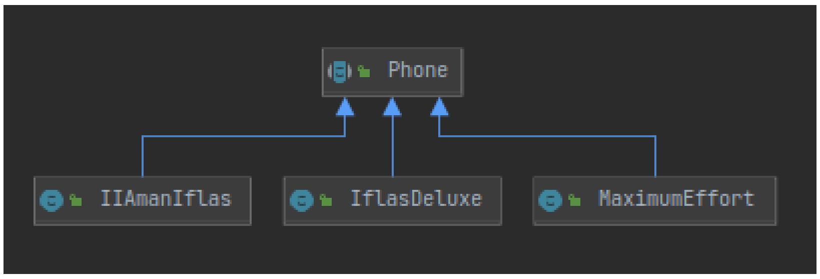
There are 3 different zones for phone production for the company. Each of them has a factory class that implements AbstractFactory. These factories can add different features for different phone models.

Each phone component has its own abstract class because its features might be with different specifications to different markets.

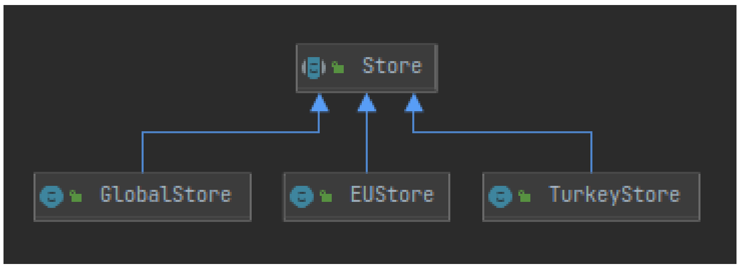


For example; Global market uses Lithium-Cobalt Battery, Turkey Market uses Lithium-Boron Battery and Europe Market uses Lithium-Ion Battery.

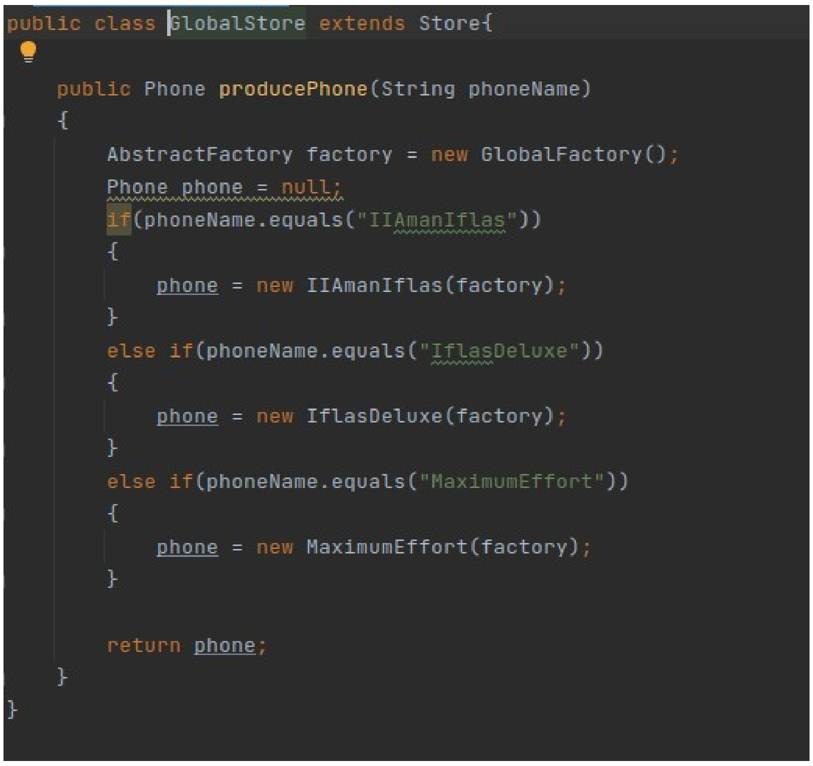
Theses classes have their own setter and getter functions for their specifications.



There is a Phone abstract class. It keeps objects of components as fields. Additionally, phone class keeps component methods to create the phone. Phone model classes extend the Phone class and creates phone using its specific features. Models keeps a factory object too. Because it is required for where the phone will be sold.

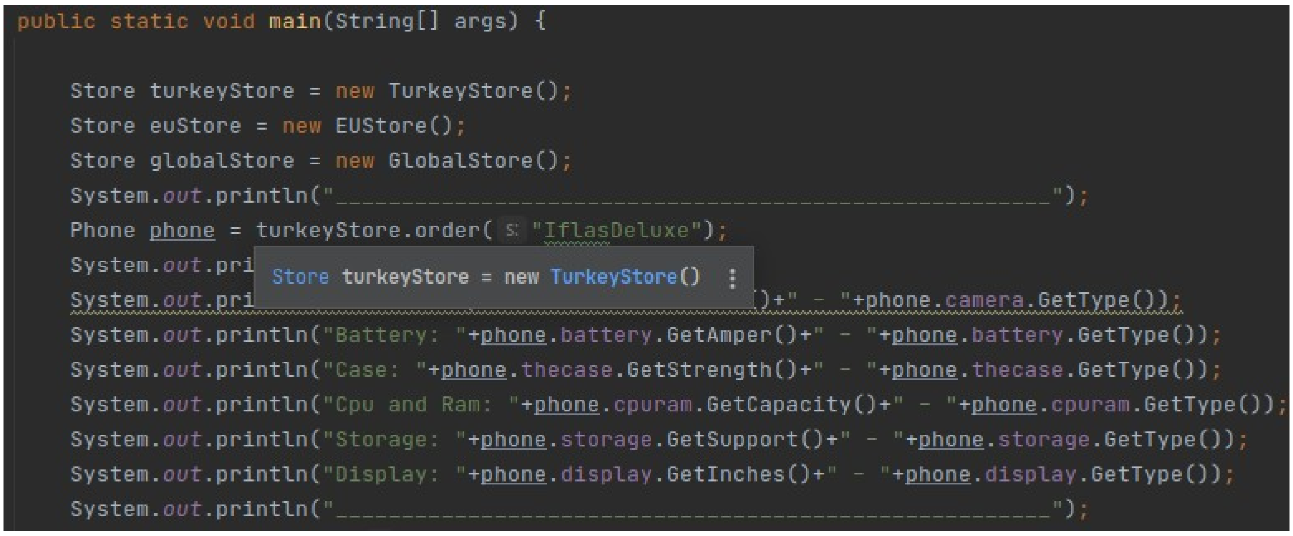


Store is an abstract class. It has 2 methods. One of them is production abstract method, the other one is order method.
Concrete store methods creates the specific model that the customer wants.



```
public class GlobalStore extends Store{

    public Phone producePhone(String phoneName)
    {
        AbstractFactory factory = new GlobalFactory();
        Phone phone = null;
        if(phoneName.equals("IIAmanIflas"))
        {
            phone = new IIAmanIflas(factory);
        }
        else if(phoneName.equals("IflasDeluxe"))
        {
            phone = new IflasDeluxe(factory);
        }
        else if(phoneName.equals("MaximumEffort"))
        {
            phone = new MaximumEffort(factory);
        }

        return phone;
    }
}
```

Main class tests all the classes and methods of the project.



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

    Store turkeyStore = new TurkeyStore();
    Store euStore = new EUStore();
    Store globalStore = new GlobalStore();
    System.out.println("______________________________________________________");
    Phone phone = turkeyStore.order( s: "IflasDeluxe");
    System.out.pri  Store turkeyStore = new TurkeyStore()  ⋮
    System.out.pri                                      )+" - "+phone.camera.GetType());
    System.out.println("Battery: "+phone.battery.GetAmper()+" - "+phone.battery.GetType());
    System.out.println("Case: "+phone.thecase.GetStrength()+" - "+phone.thecase.GetType());
    System.out.println("Cpu and Ram: "+phone.cpuram.GetCapacity()+" - "+phone.cpuram.GetType());
    System.out.println("Storage: "+phone.storage.GetSupport()+" - "+phone.storage.GetType());
    System.out.println("Display: "+phone.display.GetInches()+" - "+phone.display.GetType());
    System.out.println("______________________________________________________");
```

First, it creates a store then, creates a phone object using store's order method. This is repeated for all models to all stores.

What is the Advantage of This Design Pattern?

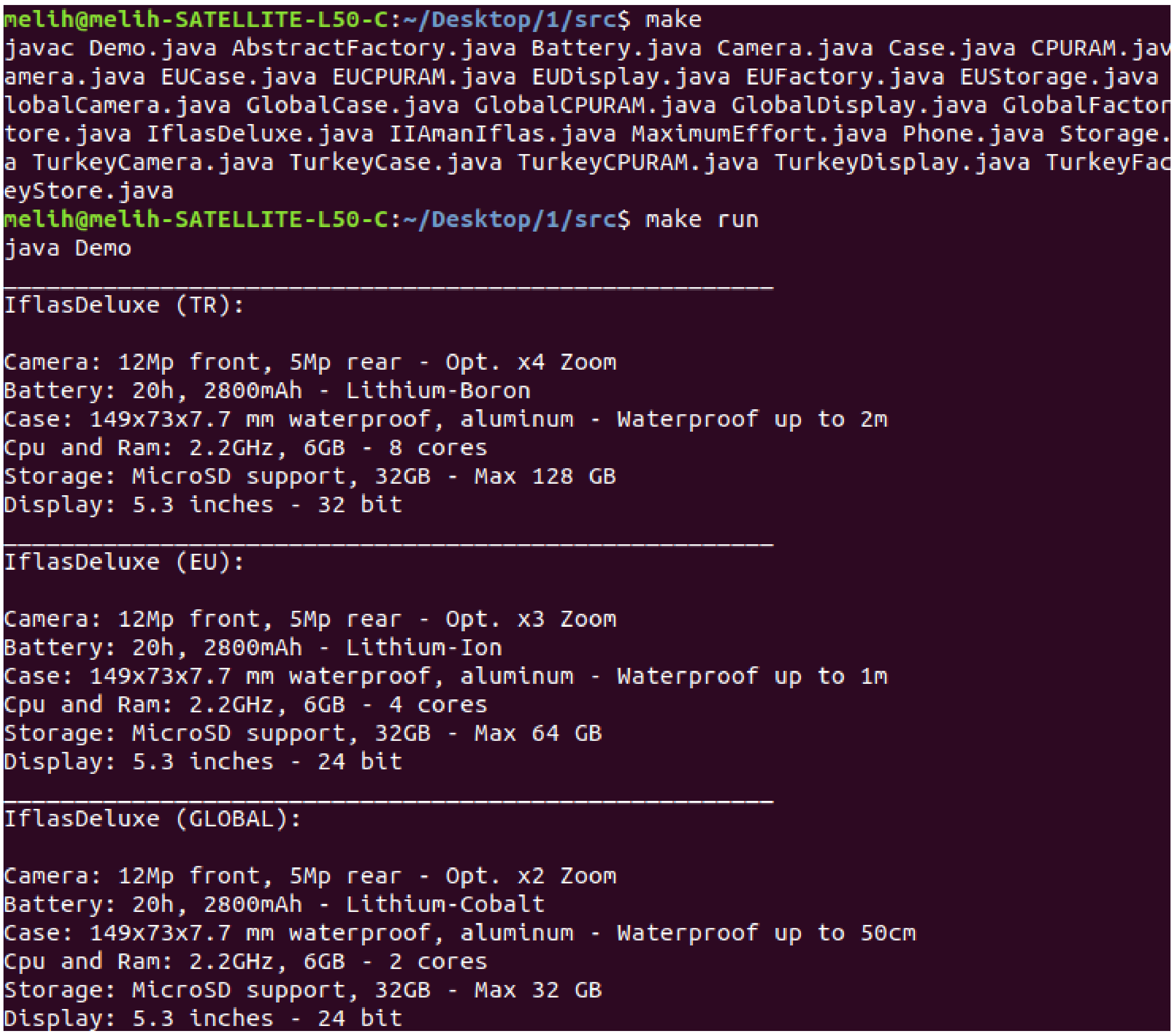
The abstract factory pattern provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes. In this project, we can easily add a new phone model, or a new factory to the system. It is very flexible and efficient to reuse.

If the company wants to change a feature of a model for a store, program allows to programmer to modify it without affect the other models or other stores.

How to Run This Program?

In src directory,

>make
>make run



```
melih@melih-SATELLITE-L50-C:~/Desktop/1/src$ make
javac Demo.java AbstractFactory.java Battery.java Camera.java Case.java CPURAM.jav
amera.java EUCase.java EUCPURAM.java EUDisplay.java EUFactory.java EUStorage.java
lobalCamera.java GlobalCase.java GlobalCPURAM.java GlobalDisplay.java GlobalFactor
tore.java IflasDeluxe.java IIAmanIflas.java MaximumEffort.java Phone.java Storage.
a TurkeyCamera.java TurkeyCase.java TurkeyCPURAM.java TurkeyDisplay.java TurkeyFac
eyStore.java
melih@melih-SATELLITE-L50-C:~/Desktop/1/src$ make run
java Demo
______________________________________________________
IflasDeluxe (TR):

Camera: 12Mp front, 5Mp rear - Opt. x4 Zoom
Battery: 20h, 2800mAh - Lithium-Boron
Case: 149x73x7.7 mm waterproof, aluminum - Waterproof up to 2m
Cpu and Ram: 2.2GHz, 6GB - 8 cores
Storage: MicroSD support, 32GB - Max 128 GB
Display: 5.3 inches - 32 bit
______________________________________________________
IflasDeluxe (EU):

Camera: 12Mp front, 5Mp rear - Opt. x3 Zoom
Battery: 20h, 2800mAh - Lithium-Ion
Case: 149x73x7.7 mm waterproof, aluminum - Waterproof up to 1m
Cpu and Ram: 2.2GHz, 6GB - 4 cores
Storage: MicroSD support, 32GB - Max 64 GB
Display: 5.3 inches - 24 bit
______________________________________________________
IflasDeluxe (GLOBAL):

Camera: 12Mp front, 5Mp rear - Opt. x2 Zoom
Battery: 20h, 2800mAh - Lithium-Cobalt
Case: 149x73x7.7 mm waterproof, aluminum - Waterproof up to 50cm
Cpu and Ram: 2.2GHz, 6GB - 2 cores
Storage: MicroSD support, 32GB - Max 32 GB
Display: 5.3 inches - 24 bit
```

It shows a test of production phases of every model from every market.

Part 2

There is a new interface for credit card payments.

```
public interface ModernPayment{
        int pay(String cardNo, float amount, String destination, String installments);
}
```

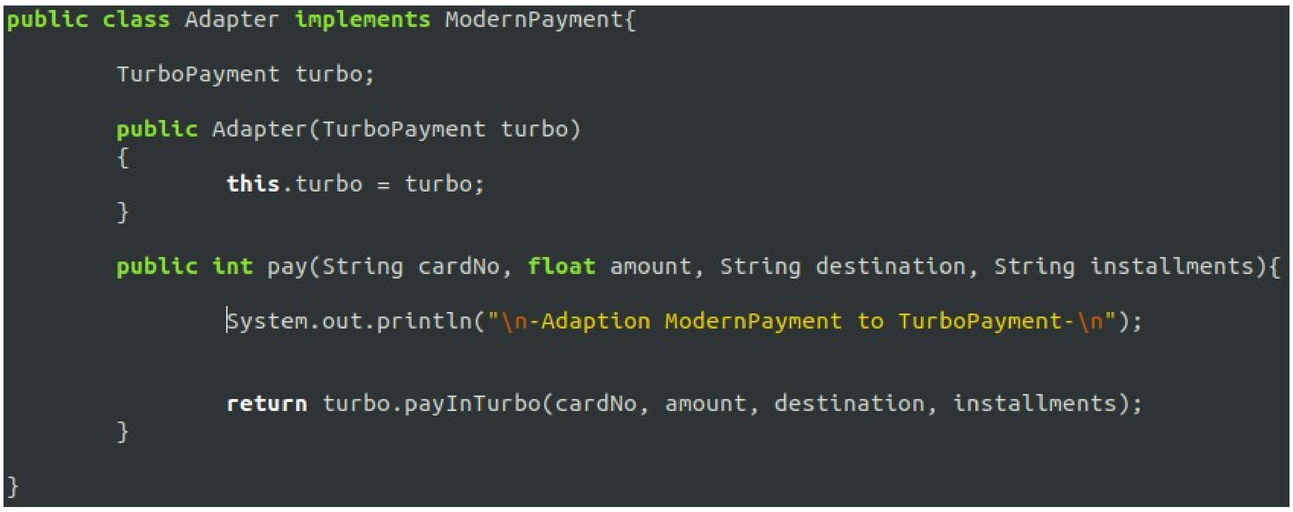
However the company uses an old payment interface.

```
public interface TurboPayment{
        int payInTurbo(String turboCardNo, float turboAmount,
                        String destinationTurboOfCourse, String installmentsButInTurbo);
}
```

It is not allowed to modify these interfaces. So we should find a way to use ModerrnPayment interface although the company uses an old payment method.

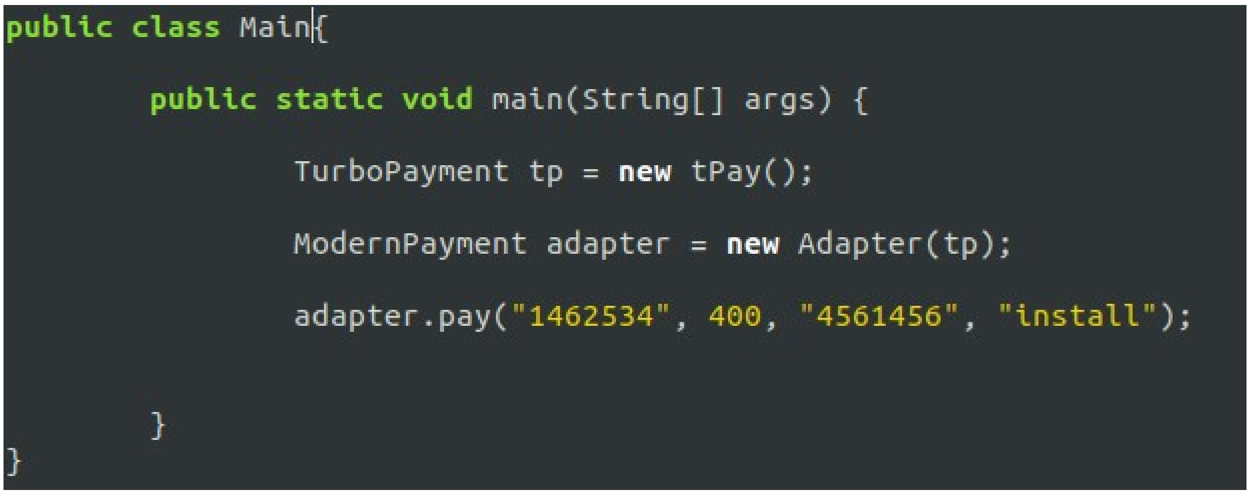
Which Design Pattern Should Be Used?

We should use **Adapter Design Pattern** because it provides a transformation without modifying existing interfaces.
There is no need to change these interfaces. We need an adapter class to ensure necessary changes.
The adapter pattern is a software design pattern that allows the interface of an existing class to be used as another interface.



```
public class Adapter implements ModernPayment{

        TurboPayment turbo;

        public Adapter(TurboPayment turbo)
        {
                this.turbo = turbo;
        }

        public int pay(String cardNo, float amount, String destination, String installments){

                System.out.println("\n-Adaption ModernPayment to TurboPayment-\n");


                return turbo.payInTurbo(cardNo, amount, destination, installments);
        }

}
```

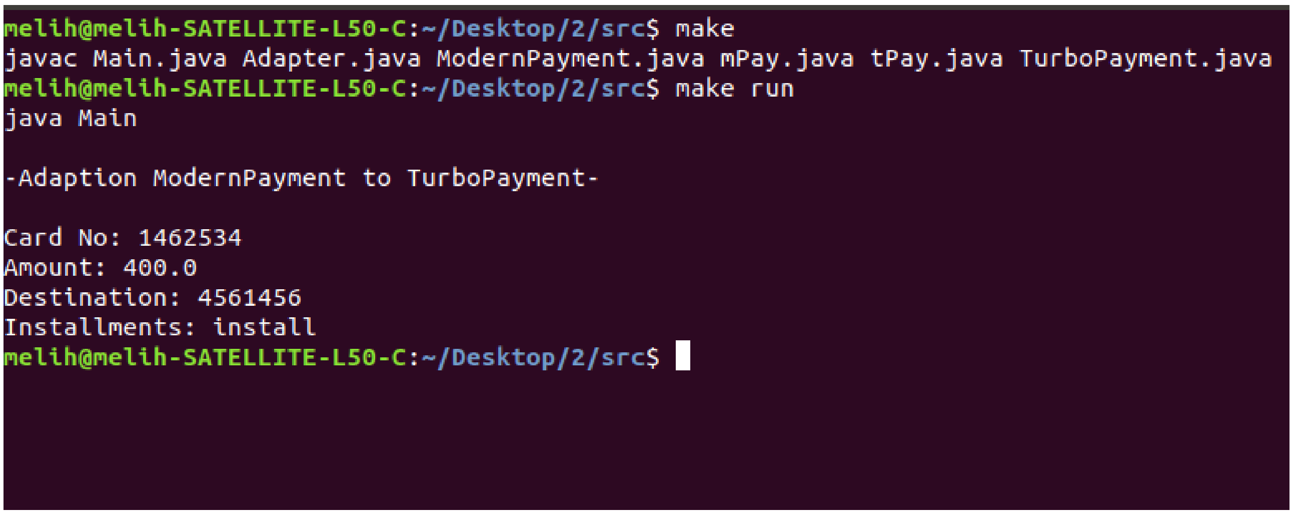
Adapter Class implements ModernPayment and keeps a TurboPayment object as a field. Its constructor gets a TurboPayment object as parameter and its pay method calls turbo object's payInTurbo method.



```
public class Main{

        public static void main(String[] args) {

                TurboPayment tp = new tPay();

                ModernPayment adapter = new Adapter(tp);

                adapter.pay("1462534", 400, "4561456", "install");


        }
}
```

Usage of this system is very easy. First create a TurboPayment object then create an adapter using TurboPayment object as a parameter. Then call adapter's pay method using information.

How to Run This Program?



```
melih@melih-SATELLITE-L50-C:~/Desktop/2/src$ make
javac Main.java Adapter.java ModernPayment.java mPay.java tPay.java TurboPayment.java
melih@melih-SATELLITE-L50-C:~/Desktop/2/src$ make run
java Main

-Adaption ModernPayment to TurboPayment-

Card No: 1462534
Amount: 400.0
Destination: 4561456
Installments: install
melih@melih-SATELLITE-L50-C:~/Desktop/2/src$
```

In src directory,

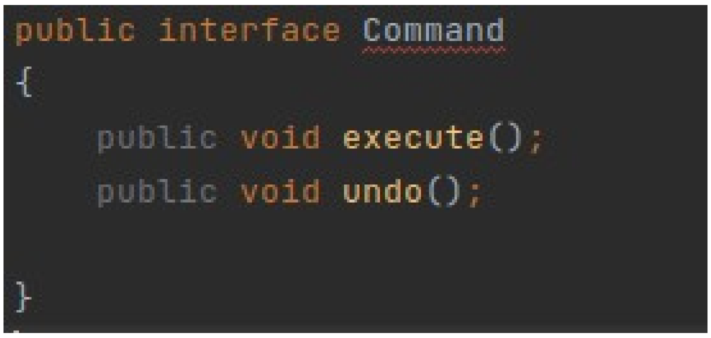
>make
>make run

I shows a test using all java files of the program.

Part 3

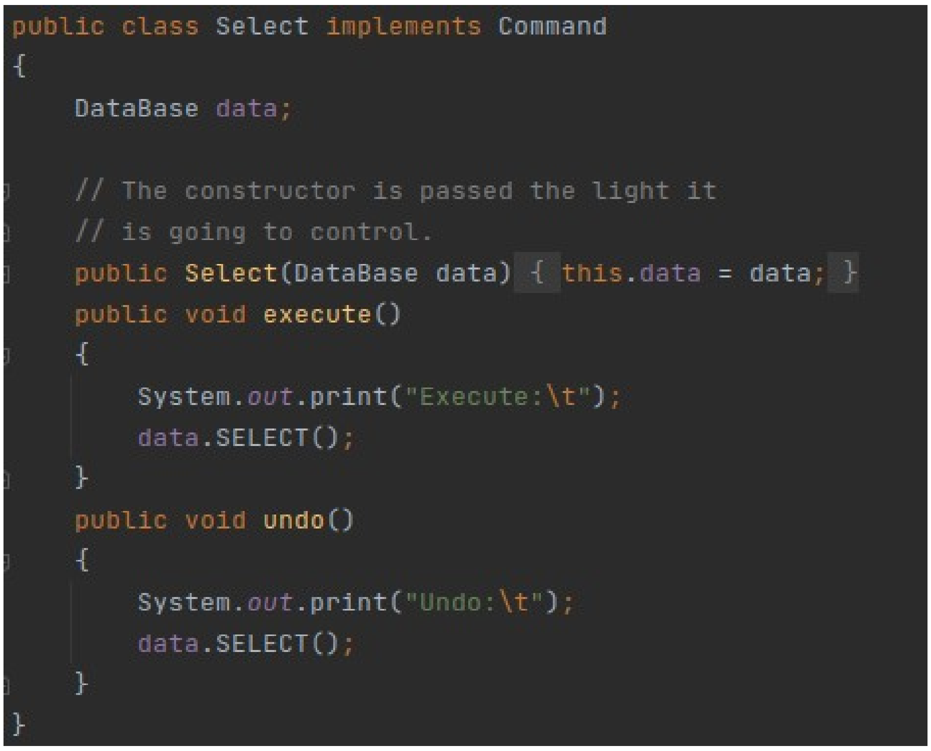
Which Design Pattern Should Be Used?

In the database engine, there are 3 types of commands. These are SELECT, UPDATE, and ALTER. Transaction is a series of these operations. So we have different kinds of commands, a database and an invoker for these commands.
We should use **Command Design Pattern** because in object-oriented programming, the command pattern is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time.
If one of the operations fail, all others must be reversed or discarded. The Command Pattern encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.



```
public interface Command
{
    public void execute();
    public void undo();

}
```

There is a Command interface to create database commands. Execute and Undo methods should exist in all commands.

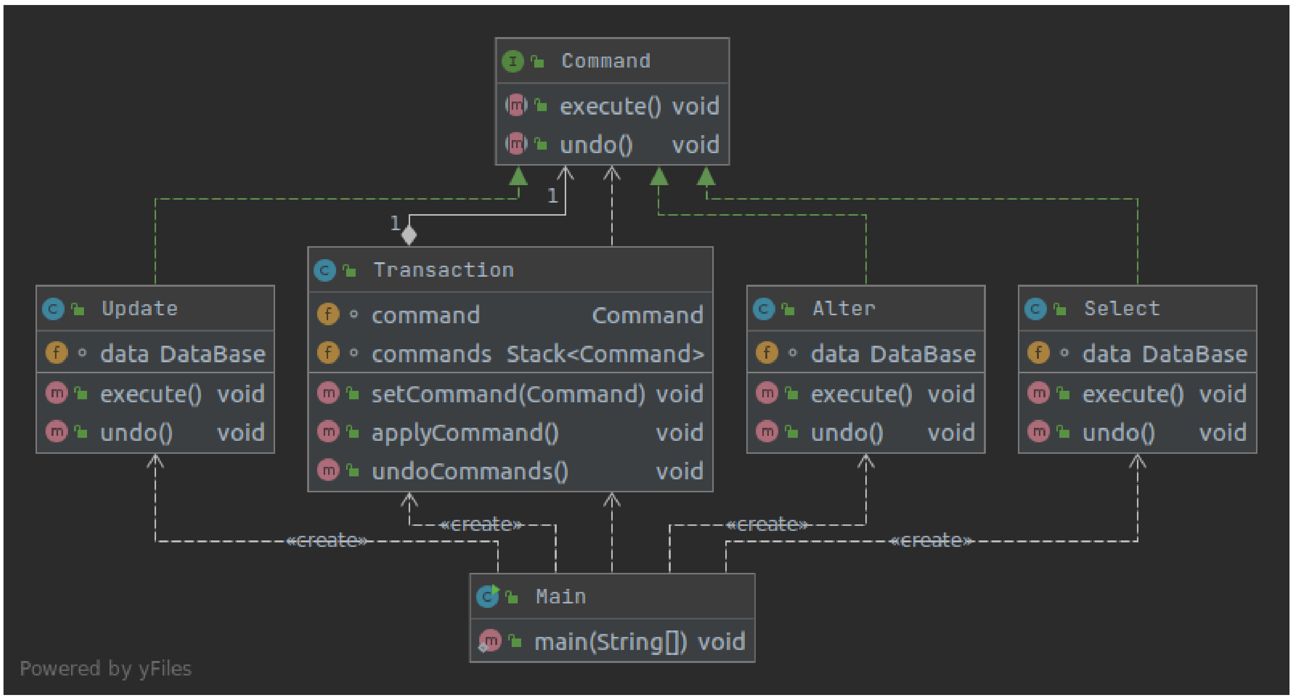


```java
public class Select implements Command
{
    DataBase data;

    // The constructor is passed the light it
    // is going to control.
    public Select(DataBase data) { this.data = data; }
    public void execute()
    {
        System.out.print("Execute:\t");
        data.SELECT();
    }
    public void undo()
    {
        System.out.print("Undo:\t");
        data.SELECT();
    }
}
```

So, each command has its own class that implements Command interface.
And each of them keeps a DataBase object because execute and undo methods manage data.

Transaction class keeps a Command object and a stack for undo operations.
When one of the operations fail, all the others reversed.

DataBase class keeps only its command methods.



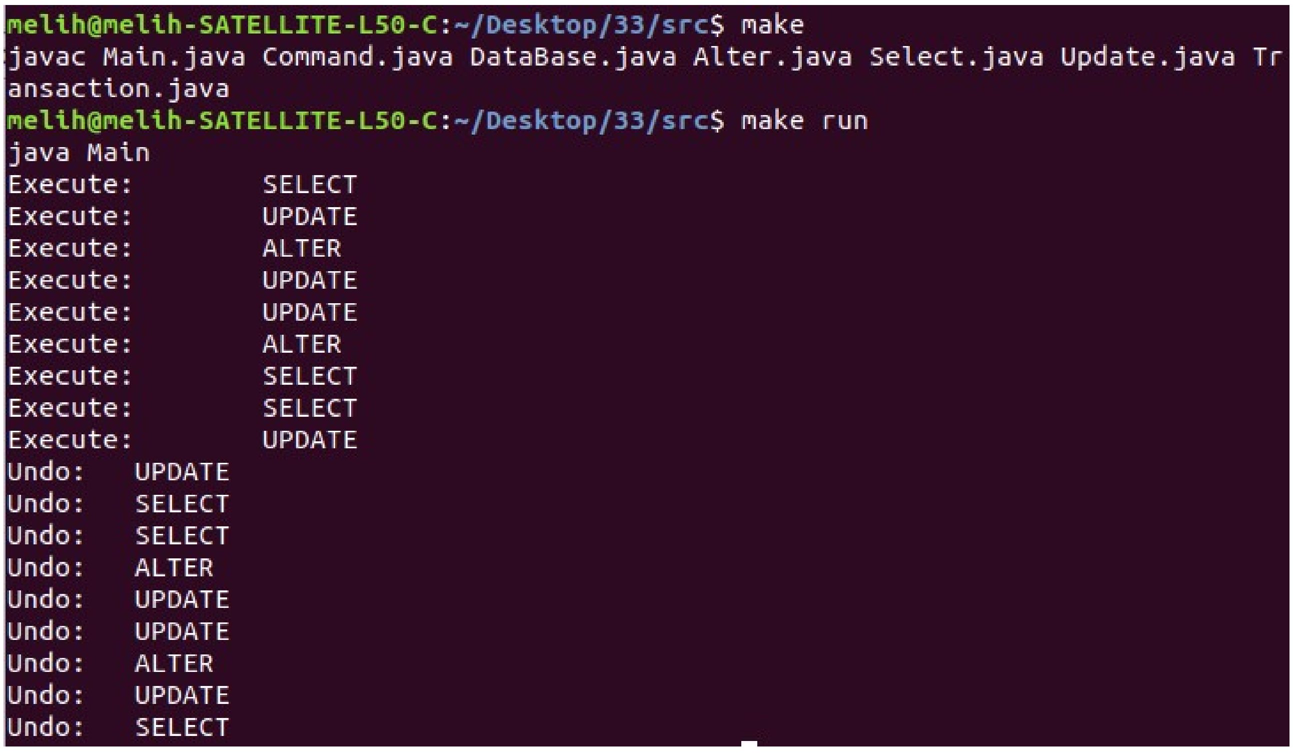
Main class has a main method that ensures an executable demo.
It creates a Transaction object and all commands’ objects.

It sets a command to Transaction object and calls its applyCommand method to execute the command.

What is the Advantage of This Design Pattern?

The Command Pattern allows us to create a sequence of commands. Creating a new device like database engine and adding them new commands is so easy. When a programmer needs to add a new command, he/she does not have to change existing code. For a new command, create a class and implement the Command interface. That is it.
Also, this pattern suitable to define a rollback system. This means that all commands can be reversed using undo method.

How to Run This Program?



```
melih@melih-SATELLITE-L50-C:~/Desktop/33/src$ make
javac Main.java Command.java DataBase.java Alter.java Select.java Update.java Tr
ansaction.java
melih@melih-SATELLITE-L50-C:~/Desktop/33/src$ make run
java Main
Execute:        SELECT
Execute:        UPDATE
Execute:        ALTER
Execute:        UPDATE
Execute:        UPDATE
Execute:        ALTER
Execute:        SELECT
Execute:        SELECT
Execute:        UPDATE
Undo:   UPDATE
Undo:   SELECT
Undo:   SELECT
Undo:   ALTER
Undo:   UPDATE
Undo:   UPDATE
Undo:   ALTER
Undo:   UPDATE
Undo:   SELECT
```

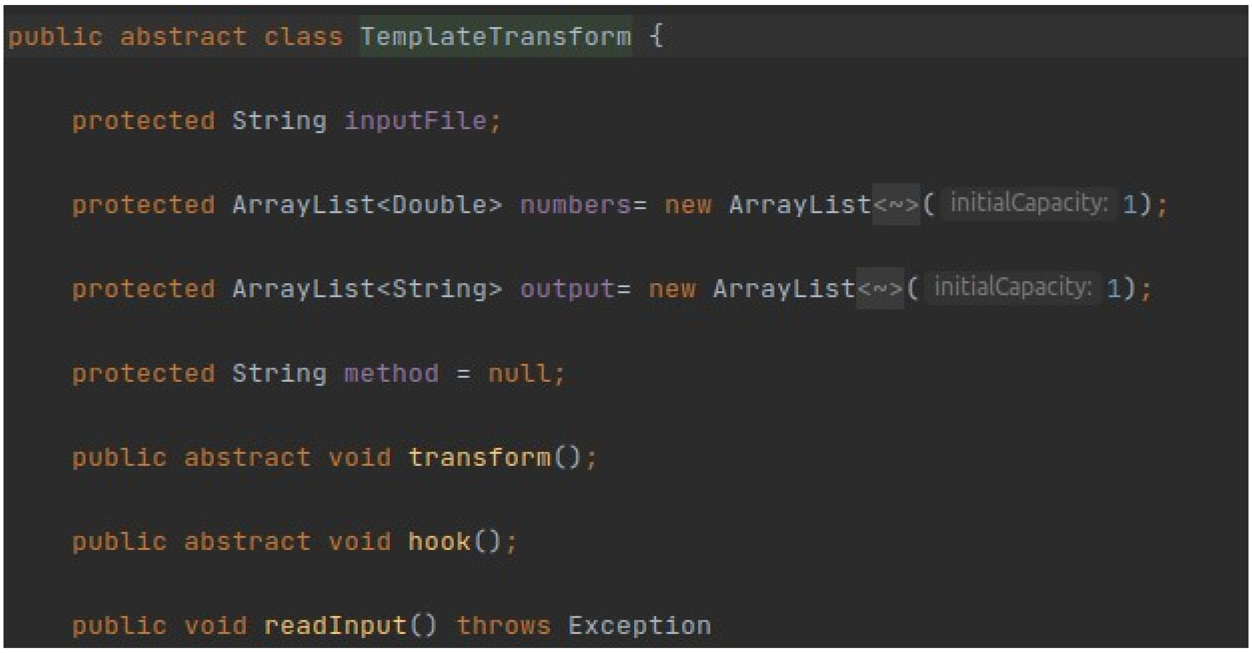
In src Directory;

>make
>make run

It shows an example of database engine test.

Part 4

There are 2 kinds of method to do the same thing. These are Discrete Fourier Transform and Discrete Cosine Transform. Difference is, DFT uses complex exponentials and DCT uses real-valued cosine functions.
So there are some common methods to implement.
In this project, we can use **Template Method Design Pattern** because Template method design pattern is to define an algorithm as a skeleton of operations and leave the details to be implemented by the child classes. It seems very efficient for this project.

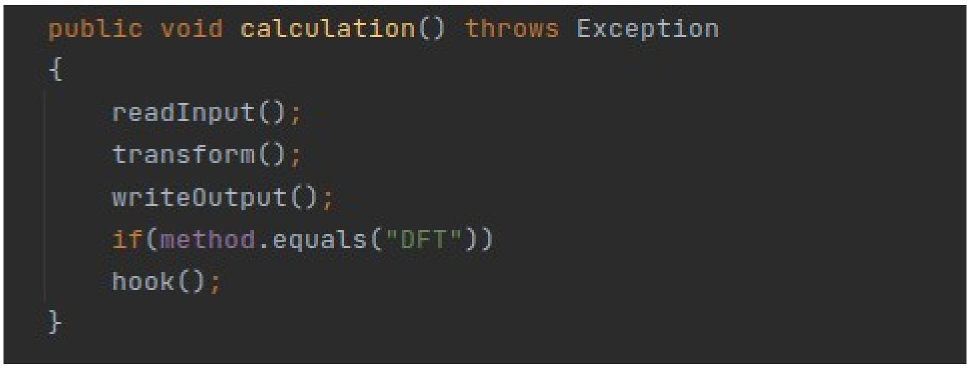


```
public abstract class TemplateTransform {

    protected String inputFile;

    protected ArrayList<Double> numbers= new ArrayList<~>( initialCapacity: 1);

    protected ArrayList<String> output= new ArrayList<~>( initialCapacity: 1);

    protected String method = null;

    public abstract void transform();

    public abstract void hook();

    public void readInput() throws Exception
```

First, an abstract template class created. What are common?
Reading an input file, writing output into a new file and so on.
Both methods use the same input file from commandline argument.
TemplateTransform class keeps input numbers into a double type array list and outputs keeping in a string type array list.

ReadInput reads the content of input file and fills numbers arraylist with them.

WriteOutput method writes content of output arraylist into the output file.

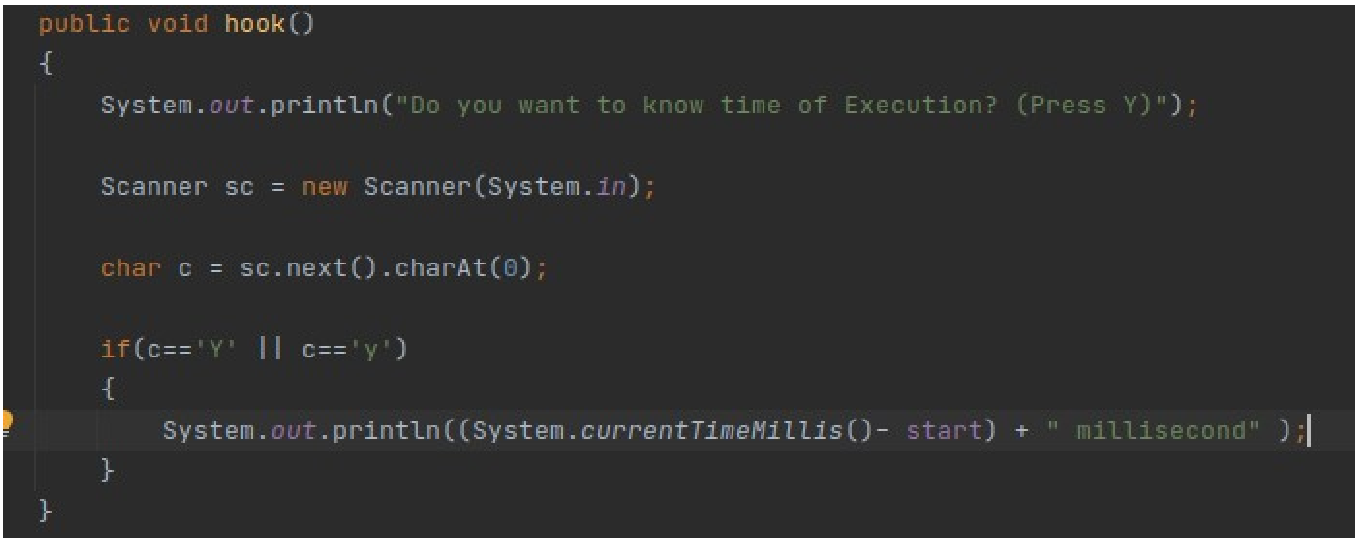


```java
public void calculation() throws Exception
{
    readInput();
    transform();
    writeOutput();
    if(method.equals("DFT"))
    hook();
}
```

Calculation method calls all the methods in an order.
These were common fields and methods in the methods.
What is different? Of course transformation techniques.

That is why transform method is an abstract method. Hook method is for showing the time of execution. However it is just needed only for DFT. So it is an abstract method too.

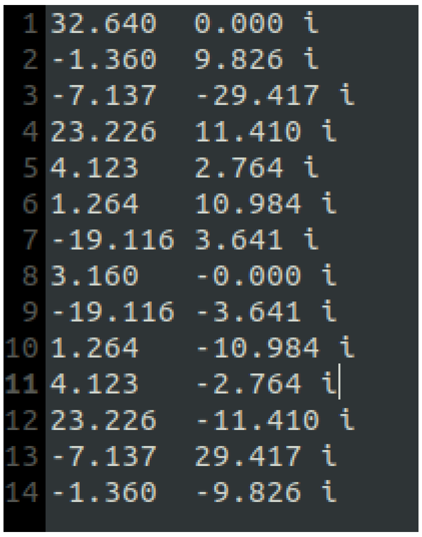
DFT produces a series of complex numbers. DCT produces a series of real numbers. Both of them writes their results into output arraylist.



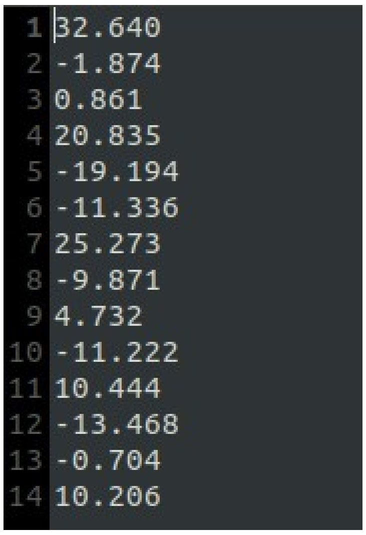
```java
public void hook()
{
    System.out.println("Do you want to know time of Execution? (Press Y)");

    Scanner sc = new Scanner(System.in);

    char c = sc.next().charAt(0);

    if(c=='Y' || c=='y')
    {
        System.out.println((System.currentTimeMillis()- start) + " millisecond" );
    }
}
```

DFT implements hook method for users who wants to know time of execution.

Example Outputs

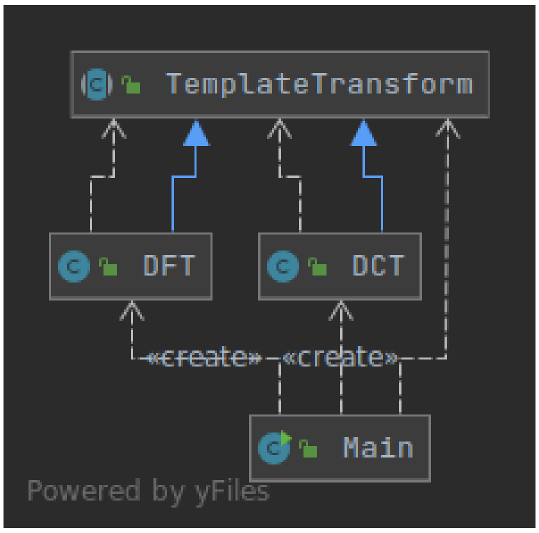


```
32.640   0.000 i
-1.360   9.826 i
-7.137   -29.417 i
23.226   11.410 i
4.123    2.764 i
1.264    10.984 i
-19.116 3.641 i
3.160    -0.000 i
-19.116 -3.641 i
1.264    -10.984 i
4.123    -2.764 i
23.226   -11.410 i
-7.137   29.417 i
-1.360   -9.826 i
```

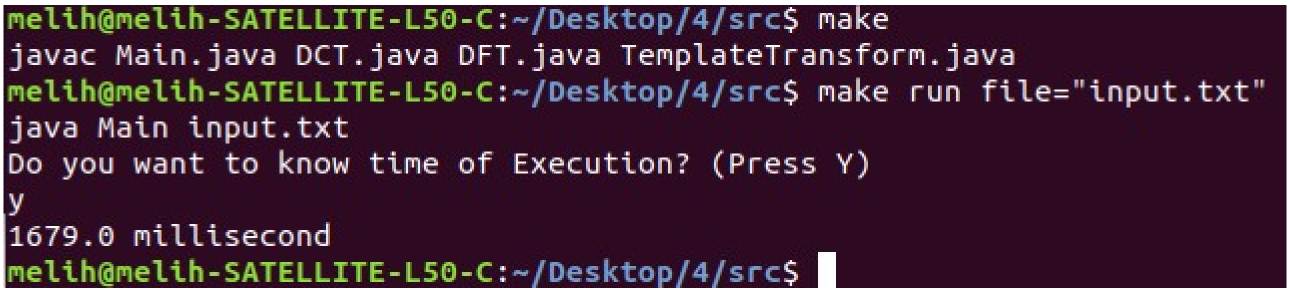


```
32.640
-1.874
0.861
20.835
-19.194
-11.336
25.273
-9.871
4.732
-11.222
10.444
-13.468
-0.704
10.206
```

UML Class Diagram



How to Run This Program?



```
melih@melih-SATELLITE-L50-C:~/Desktop/4/src$ make
javac Main.java DCT.java DFT.java TemplateTransform.java
melih@melih-SATELLITE-L50-C:~/Desktop/4/src$ make run file="input.txt"
java Main input.txt
Do you want to know time of Execution? (Press Y)
y
1679.0 millisecond
melih@melih-SATELLITE-L50-C:~/Desktop/4/src$
```

---In the src directory:

```
>make
>make run file="input.txt"
```

OR

```
>make
>java Main input.txt
```