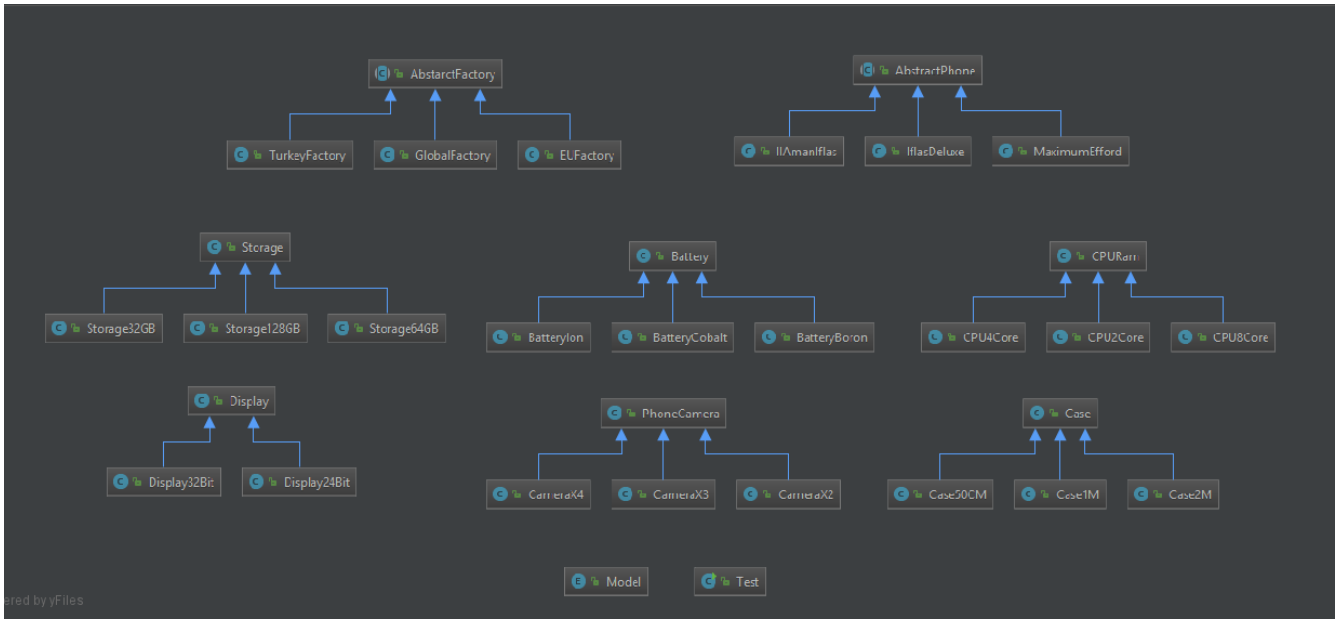
Gebze Technical Unversity Computer Engineering

CSE 443
Midterm Assigment
Report

Barış Şahin
151044016

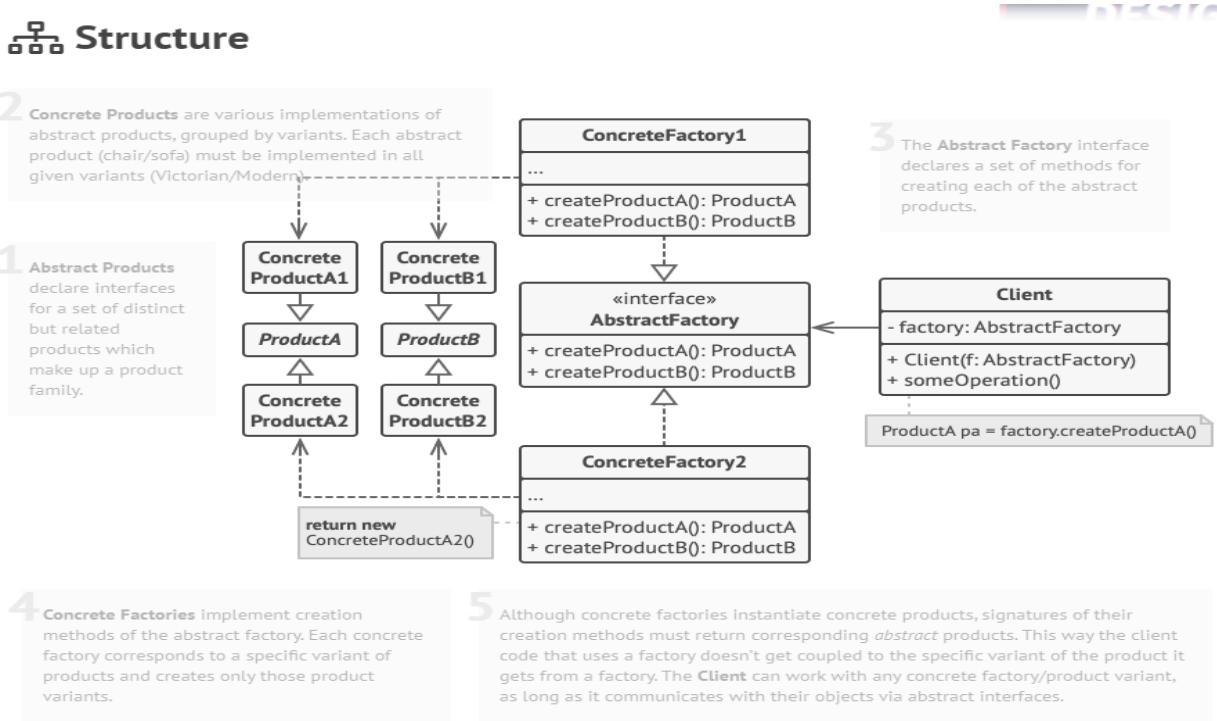
Q1

Sınıf Diyagramı



Not: Diğer Diagramların boyutu fazla büyük olduğundan SS klasörünün içinde bulunabilir.

Bu soruda bizden verilen bilgiler doğrultusunda Abstract Factoryi yazmamız istenmiştir.



Abstract Factory Pattern Structre

Bu diyagramla benim tasarımımı eşleştirirsek:

Abstract Factory →Abstract Factory

Concrete Factory →TurkeyFactory,EUFactory,GlobalFactory

Products →Display,Camera,CPURam,Storage,Battery,Case

Concrete Product → Display32,Display24,Storage128GB,Storage64GB...

Benim tasarımım tamamen Abstarct Factory'e uymamaktadır. Abstract Factory'nin sağladığı bütün kolaylıkları sağlamasının yanında kod tekrarından kaçınmak için Abstract Factory tasarımındaki Product interface olarak tanımlanır. Fakat bizim ürünlerimizde alt ürünlerin özellikleri farklı olmasına rağmen aynı niteliklere sahip oldukları için interface yerine class olarak tanımladım.

```
 * */
public class Display{

    double size;
    protected int bit;
    /**
     * For printing the properties of a display
     *
     * */
    public String getProperties(){
        return "Display: \n"
                + "\tSize: " + size +" inch\n"
                + "\tBit: " + bit +" bit\n";
    }
```

Bu classın içinde bütün alt ürünler için geçerli, ürünün niteliklerini yazdıran bir fonksiyon bulunmakta. Fakat bu durumda Concrete Product'ların oluşturulması önemini yitiriyor. Bu niteliklerden sadece bir tanesi Concrete Factory'lere göre farklılık gösteriyor. Bu değişkenlik gösteren özelliği protected olarak tanımladım ve setter veya getterını yazmadım. Sadece constructorda değer verilebilen bir hale getirmiş oldum.

```
/**
 * Concrete class for Display
 *
 * It initialize it's special protected property and has no method for changing it
 * */
public class Display32Bit extends Display{
    Display32Bit() { super.bit = 32; }
}
```

Alt ürünlerin default constructor'unda bu değerleri tanımlayarak Concrete Productlara tekrar bir anlam yükledim ve sağladıkları kolaylığı geri kazandım.

Böyle bir değişiklik yaptığım için telefon modellerinin attach metodlarında hangi tip bir alt ürün kullanacaklarını anlamaları için parametre olarak ürün tipinde bir parametre almaları gerekti.

```
/**Attaches the Display  to the mainboard it takes Display type as parameter
 * But only bit is unchangeable
 * All other paramaters in Display object can be changed by the class that implements this*/
public abstract void attachDisplay(Display gDisplay);
```

Bu parametre ürün tipinde olduğu için telefon hala ekstra özelliklerini bilmek zorunda değil sadece kendini ilgilendiren kısmı ile uğraşıyor.

```
@Override
public void attachDisplay(Display gDisplay) {
    System.out.println("Display attaced");
    gDisplay.size = 5.5;
    super.display = gDisplay;
    System.out.println(gDisplay.getProperties());
    System.out.println("-----------------");
}
```

Telefon üretilirken her Concrete Factory ise model farketmeksizin kendi kullandığı tipteki alt ürünü göndererek özelleştirmesini yapmış oluyor.

```
public class TurkeyFactory extends  AbstarctFactory{
    @Override
    public AbstractPhone takeOrder(Model model) {
        AbstractPhone phone = choosePhone(model);
        if(phone == null){
            System.out.println("Iflas-Technologies Ltd has no model like this please give a valid model:");
            System.out.println("Our models are:");
            System.out.println("MAXIMUMEFFOR\nIFLASDELUXE\nIIAMANIFLAS\n");
            return null;
        }
        System.out.println("Turkey Factory Working!");
        phone.attachCPUandRAM(new CPU8Core());
        phone.attachDisplay(new Display32Bit());
        phone.attachBattery(new BatteryBoron());
        phone.attachStorage(new Storage128GB());
        phone.attachCamera(new CameraX4());
        phone.enclosePhoneCase(new Case2M());
        return phone;
    }
```

Kullanımı için adımlar:

Kullanmak istediğimiz ülkeye göre bir fabrika oluşturuyoruz.

Bu fabrikaya hangi telefonu istediğimizi Model Enum'ını kullanarak takeOrder foksiyonuna parametre olarak veriyoruz.

Telefonumuz hazır.

Test sınıfında bütün fabrikaların bütün modelleri için kullanımı gösterilmiştir:

```java
public class Test {

    public static void main(String[] args){

        AbstarctFactory turkeyFactory = new TurkeyFactory();
        AbstarctFactory euFactory = new EUFactory();
        AbstarctFactory globalFactory = new GlobalFactory();

        System.out.println("##################################");
        System.out.println("Tests For Turkey Factory");
        System.out.println("##################################");
        System.out.println("Test Maximum Efford");
        System.out.println(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>");
        turkeyFactory.takeOrder(Model.MAXIMUMEFFORD);
        System.out.println(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>");
        turkeyFactory.takeOrder(Model.IFLASDELUXE);
        System.out.println(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>");
        turkeyFactory.takeOrder(Model.IIAMANIFLAS);
        System.out.println("##################################");
        System.out.println("Tests For Turkey Factory Finished");
        System.out.println("##################################");
        System.out.println("\nVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV\n");
        System.out.println("##################################");
        System.out.println("Tests For EU Factory");
        System.out.println("##################################");
        System.out.println("Test Maximum Efford");
        System.out.println(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>");
        euFactory.takeOrder(Model.MAXIMUMEFFORD);
        System.out.println(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>");
        euFactory.takeOrder(Model.IFLASDELUXE);
        System.out.println(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>");
        euFactory.takeOrder(Model.IIAMANIFLAS);
        System.out.println("##################################");
        System.out.println("Tests For EU Factory Finished");
        System.out.println("##################################");
        System.out.println("\nVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV\n");
        System.out.println("##################################");
        System.out.println("Tests For Global Factory");
        System.out.println("##################################");
        System.out.println("Test Maximum Efford");
        System.out.println(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>");
        globalFactory.takeOrder(Model.MAXIMUMEFFORD);
        System.out.println(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>");
        globalFactory.takeOrder(Model.IFLASDELUXE);
        System.out.println(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>");
        globalFactory.takeOrder(Model.IIAMANIFLAS);
        System.out.println("##################################");
        System.out.println("Tests For Global Factory Finished");
        System.out.println("##################################");
    }
}
```

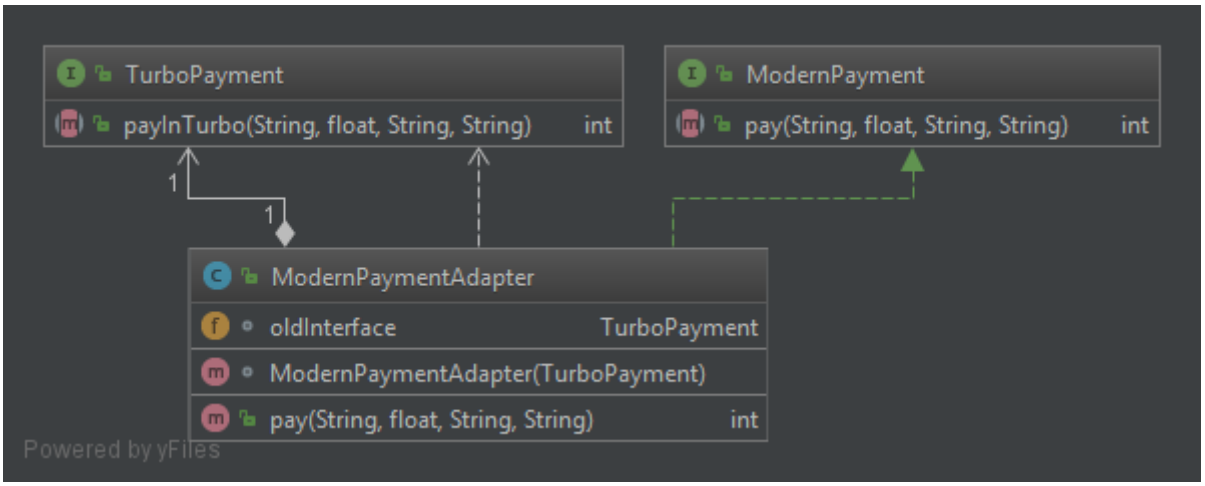
Test sınıfının çıktısını Q1 dosyasındaki testResult.txt dosyasında bulabilirsiniz.

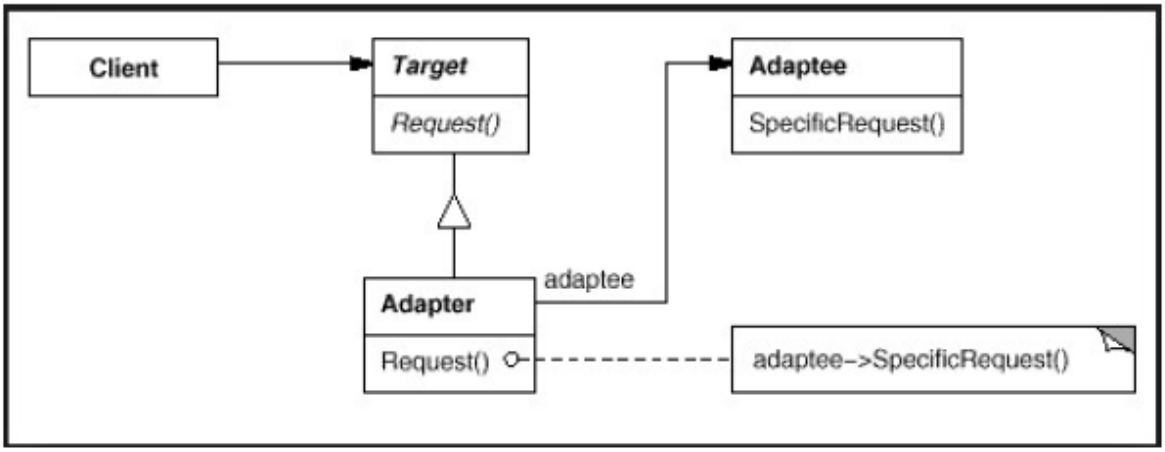
Q2

Soruda söylenmemiş olmasına karşın bariz bir şekilde Adapter Pattern ile çözülmesi gerektiği görülmektedir.

İçine aldığı TurboPayment arayüzünü gerçekleyen bir sınıfı, ModernPayment'a adapte eden ve ModernPayment arayüzünü gerçekleyen bir sınıf yazdım.

UML Diyagramı





Adaptor Design Pattern Structure

Bu diyagram ile benim tasarımımı eşleştirirsek:

Adaptee-> Turbo Payment

Adapter ->Payment Adapter

Target -> Modern Payment

Q3

Bu soruda bizden SQL komut demetlerini işleyip bu demetlerin herhangi birinde problem çıkarsa bütün demetin etkisinin geri alınması istenmiştir. Ben bunun için 2 Tasarım Örüntüsünü beraber kullandım. SQL görevlerini işlemek için Command Pattern, geri alma işlemleri için ise Memento Pattern kullandım.

Memento Patterni daha önceden çalışmış olduğum bir şirkette geri alma işlemleri için fazlaca kullandığımızdan ötürü daha önceden biliyordum.

SQL komutları için ortak olan Command arayüzünü yazdım.

```
/**
 * Interface for commands
 *
 */
public interface Command {
    /**
     * @return execute function returns 1 on failed command return 0 on successful one
     */
    public int execute();
}
```

Normalde Command Pattern e göre bunun bir sınıf olması tercih edilirdi fakat içinde bulunan kaydetme ve geri yükleme işini Memento Pattern içinde Originator sınıfı olarak belirlediğim DB içinde hallettim.

```
    public Backup save() { return new Backup(tables); }
    public void restore(Backup backupFile) { tables = backupFile.getState(); }
     /**
```

Memento Pattern'i gerçeklemek için DB içinde bir Backup adında dahili sınıf oluşturdum.

```
    class Backup {
        private ArrayList<Table> tables;
        Backup(ArrayList<Table> gTables){
            tables = new ArrayList<>();
            tables.addAll(gTables);

        }
        private ArrayList<Table> getState() { return tables; }
    }
```

Test edilebilmesi için Update sınıfını sürekli hata verecek şekilde,Alter sınıfını da en eski tabloyu silecek şekilde yazdım.

```java
public class Alter implements Command {
    DB reciver;
    String[] params;

    Alter(DB database,String[] args){
        reciver = database;
        params = args;
    }
    /**
     * For showing backup system and rolling back work this command delete oldest table in DB and return always success
     * */
    @Override
    public int execute() {
        System.out.println("Alter Executed");
        reciver.delTable();
        return 0;
    }
}
```

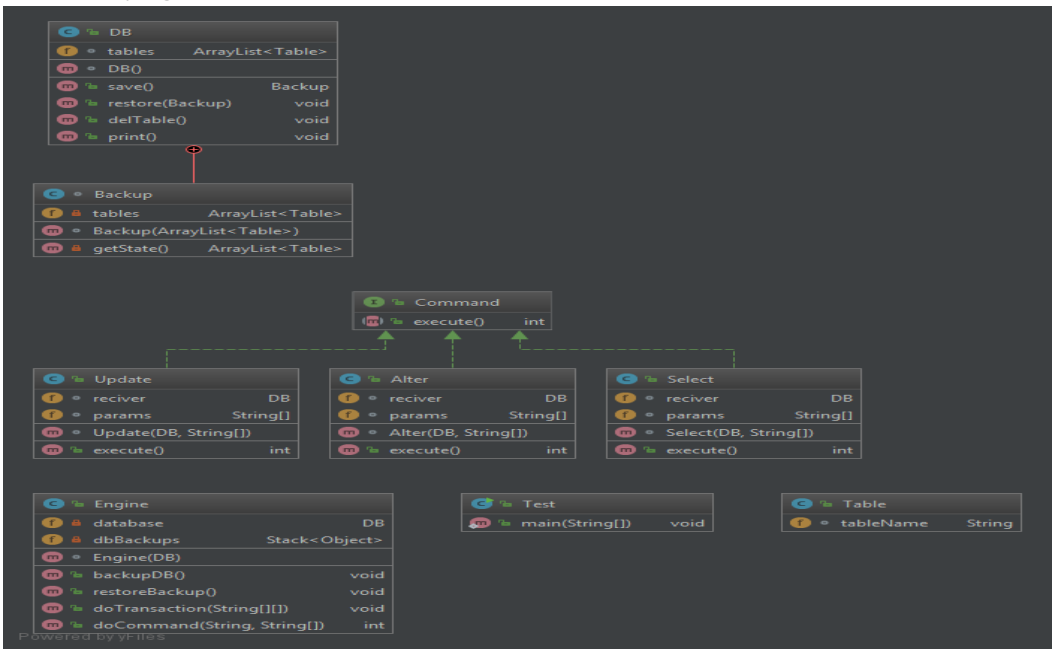
```java
public class Update implements Command {
    DB reciver;
    String[] params;

    Update(DB database,String[] args){
        reciver = database;
        params = args;
    }
    /**
     * For showing backup system and rolling back work this command does nothing and return always fail
     * */
    @Override
    public int execute() {
        System.out.println("Update Executed");
        return 1;
    }
}
```

Test sınıfında geri alma işlemini yapabildiği gösterilmiştir.

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

        DB database = new DB();
        Engine dbEngine = new Engine(database);

        Table table = new Table();
        table.tableName = "0. table";

        Table table1 = new Table();
        table1.tableName = "1. table";


        Table table2 = new Table();
        table2.tableName = "2. table";

        database.tables.add(table);
        database.tables.add(table1);
        database.tables.add(table2);

        String[][] transaction = {{"SELECT","Param1","Param2","Paran3"},
                {"SELECT","Param1","Param2","Paran3"},
                {"ALTER","Param1","Param2","Paran3"},
                {"ALTER","Param1","Param2","Paran3"},
                {"ALTER","Param1","Param2","Paran3"},
                {"UPDATE","Param1","Param2","Paran3"}};


        String[][] transaction1 = {{"SELECT","Param1","Param2","Paran3"},
                {"SELECT","Param1","Param2","Paran3"},
                {"ALTER","Param1","Param2","Paran3"},
                {"ALTER","Param1","Param2","Paran3"},
                {"ALTER","Param1","Param2","Paran3"}};

        System.out.println("Start Of DB:");
        database.print();
        System.out.println("//////////////////////");
        dbEngine.doTransaction(transaction);
        System.out.println("###################################\nAfter \"SELECT\",\"SELECT\",\"ALTER\",\"ALTER\",\"ALTER\",\"UPDATE\"(fail) transaction:");
        database.print();
        System.out.println("###################################");
        dbEngine.doTransaction(transaction1);
        System.out.println("###################################\nAfter \"SELECT\",\"SELECT\",\"ALTER\",\"ALTER\",\"ALTER\" transaction:");
        database.print();
        System.out.println("###################################");

    }
}
```

Test sonuçlarını Q3 klasöründeki testResults.txt de görebilirsiniz.

Sınıf Diyagramı



UML Diagram

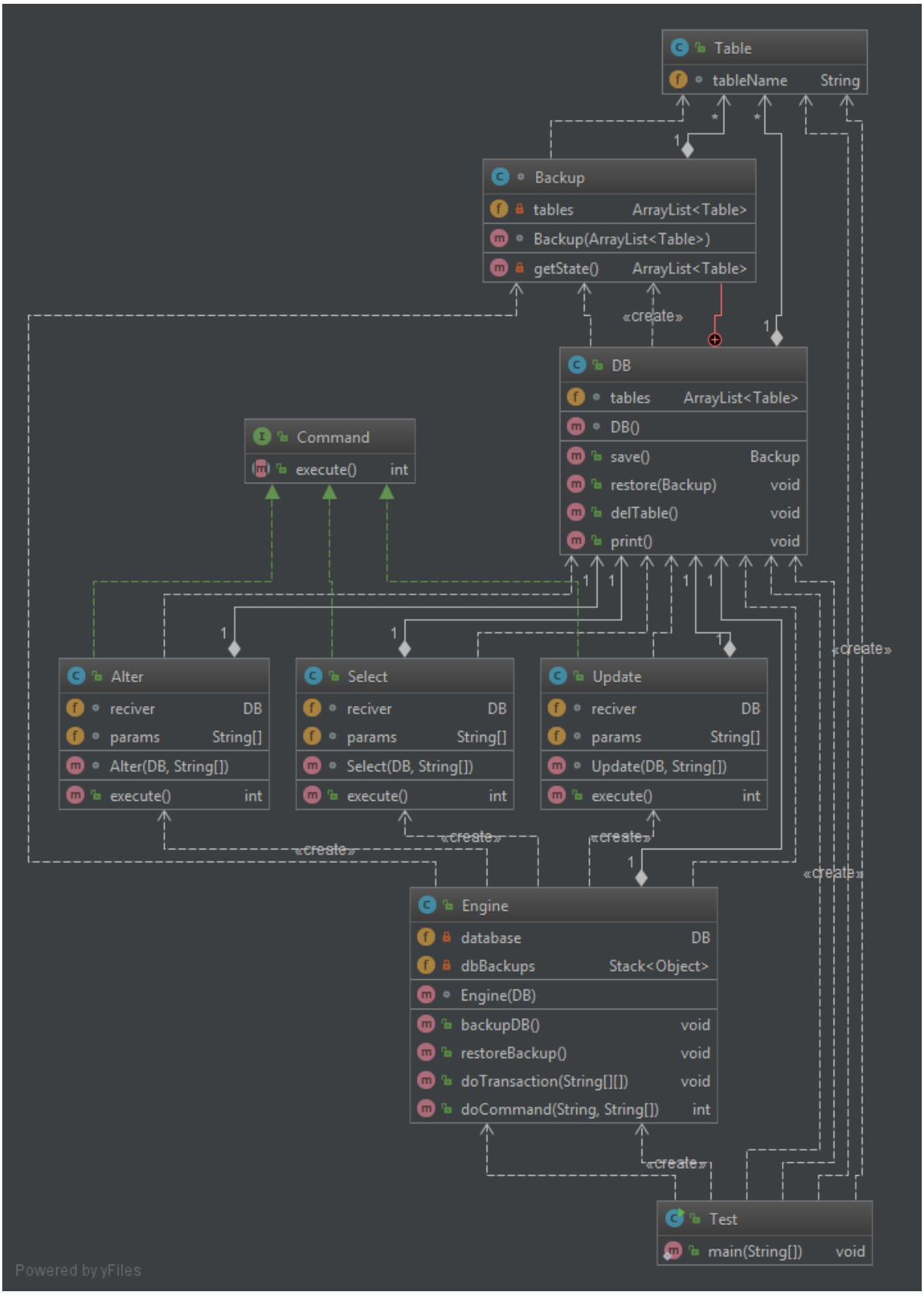


Table
tableName String
Backup
tables ArrayList<Table>
Backup(ArrayList<Table>)
getState() ArrayList<Table>
«create»
DB
tables ArrayList<Table>
DB()
save() Backup
restore(Backup) void
delTable() void
print() void
Command
execute() int
Alter
reciver DB
params String[]
Alter(DB, String[])
execute() int
Select
reciver DB
params String[]
Select(DB, String[])
execute() int
Update
reciver DB
params String[]
Update(DB, String[])
execute() int
Engine
database DB
dbBackups Stack<Object>
Engine(DB)
backupDB() void
restoreBackup() void
doTransaction(String[][]) void
doCommand(String, String[]) int
Test
main(String[]) void
Powered by yFiles

Q4

İki farklı Transform yapısının da benzer adımlarla bir arada kullanılabilieceği bir tasarım örneği implement edilmesi istenmektedir. Template Method Design Pattern kullanılması istenmiştir.

Adımların sıralması aynı kaldığından final method ,template method ‘a yer verilmiştir.

```
public final void calculateSolution(){

    try {

        readFromFile();
        ExecTimeWant();
        solve();
        writeToFile();

    }
    catch (Exception e){
        System.out.println(e.getMessage());
    }

}
```

Adımlar sırasındaki tek fark müşteri tercihine göre DFT çözümünde zaman tutulmasını isteyebilir. Buna yine de final method’da yer verilmiştir. Akışı bozmaması için bu fonksiyon DCT sınıfında boş olarak yazılmıştır.

```
@Override
public boolean ExecTimeWant() {
    //In DCT there is no need for this function
    return true;
}
```

DCT › ExecTimeWant()

İşlemler için input ve outpur dosyalarının konumu constructor aracılığı ile alınmaktadır. Bunlar main fonksiyonun argümanları aracılığı ile verilmiştir.

2 çözüm de test sınıfı aracılığı ile aşşağıdaki gibi test edilebilir:

```java
public class Test{
        public static void main(String[] args) {
            System.out.println("Liste:" + args.length);
            FunctionTransform sol1 = new DFT(args[0],args[1]);
            FunctionTransform sol2 = new DCT(args[0],args[1]);


            System.out.println("Select solution:\n1 for DFT\n2 for DCT");
            Scanner reader = new Scanner(System.in);
            String opt = reader.next();
            while(true){
                if (opt == "1"){
                    sol1.calculateSolution();
                    return;
                }
                else if (opt == "2"){
                    sol2.calculateSolution();
                    return;}
                else{
                    System.out.println("Please only enter 1 or 2:\n1 for DFT\n2 for DCT");
                }
            }



        }
```

UML Diyagramı