

GEBZE TECHNICAL UNIVERSITY

COMPUTER ENGINEERING



CSE443 - OBJECT ORIENTED ANALYSIS AND
DESIGN

Midterm Project - Report

Student:
Nevra GÜRSES
161044071

1 QUESTION 1

1.1 PROBLEM DEFINITION

There is smartphone production in company. Every smartphone consists of 6 components (Model, Display, Battery, CPU & Ram, Storage, Camera, Case) and 3 distinct model series are producing by company that are Maximum Effort, IflasDeluxe and I-I-Aman-Iflas. The production of a phone is made definite order that is: attach cpu & ram to the board - attach display - attach battery - attach storage - attach camera and enclose the phone case.

There is important case in that company. Company sells the same models, with different specifications to different markets. For example the same model IflasDeluxe is sold in Turkey with a 5.3 inch, 32bit display while it's sold at the EU market with a 5.3 inch but 24- bit display and so on.

Our goal is developing a Java program where we implement the Abstract Factory design pattern for the production of smartphones and printing on screen step by step the production phases of every model from every market.

1.2 METHOD

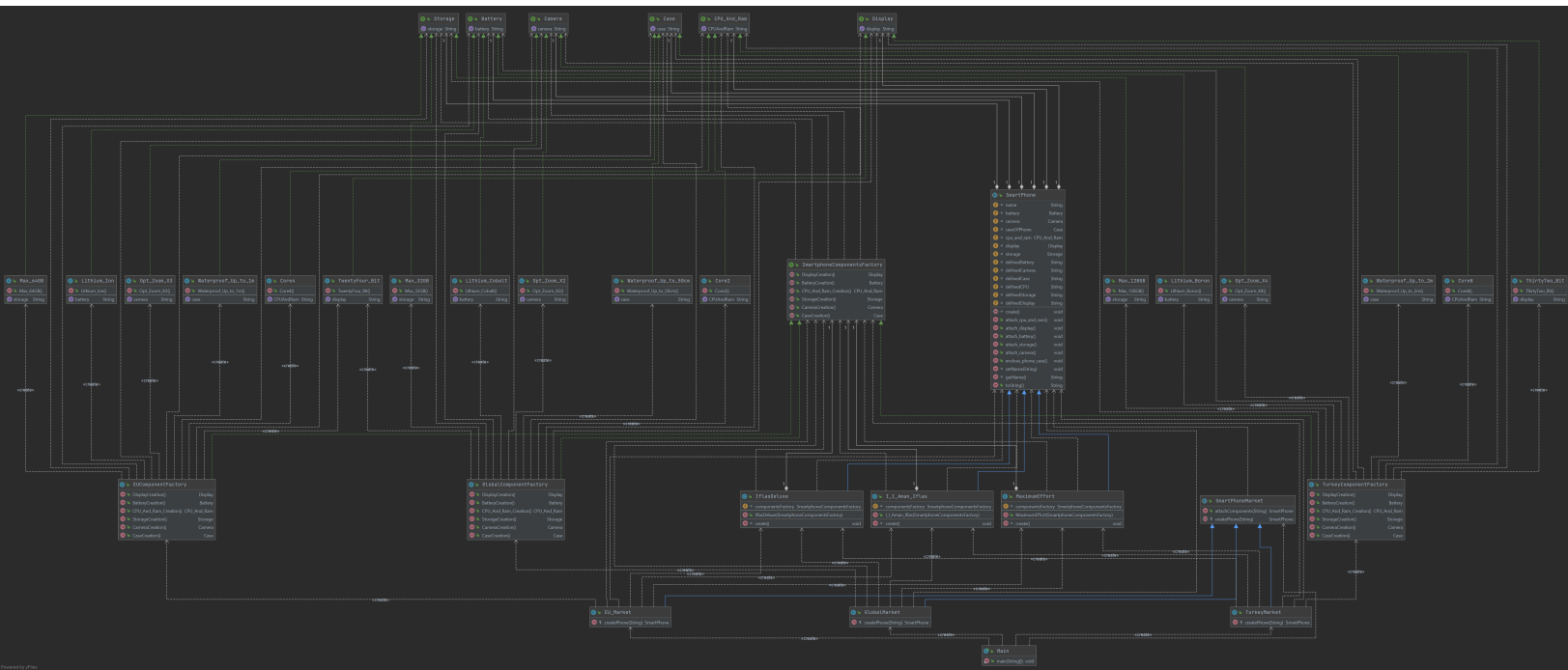
1.2.1 Why Abstract Factory Design Pattern is Using for This Goal?

The Abstract Factory Design Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. By writing code that uses this interface, we decouple our code from the actual factory that creates the products. That allows us to implement a variety of factories that produce products meant for different contexts. In our project this different contexts are different smartphone markets that are Turkey, EU, Global. Because our code is decoupled from the actual products, we can substitute different factories to get different behaviors (like 32 bit Display instead of 24 bit Display). This design pattern also allows for new derived types to be introduced with no change to the code that uses the base class. This is very important for our goal. Thanks to this feature new models are easily using. This feature provide loose coupling by reducing the dependency of our application on concrete classes.

The job of an Abstract Factory is to define an interface for creating a set of products. In our project, Smartphone Components interface creates families of products (components of smartphones). This interface also provides ease for future models that might use some of the same components. Each method in that interface is responsible for creating a concrete product, and we implement a subclass of the Abstract Factory to supply those implementations. And also factory methods are a natural way to implement our product methods in our abstract factories. The concrete factories implement the different product families. For example in our project these product families are Turkey, EU, Global Component markets. To create a product, the client uses one of these factories, so it never has to instantiate a product object.

Briefly, Abstract Factory Design Pattern is providing maximum flexibility, minimum cost and loose coupling for our problem solution and it is the most appropriate design pattern according to our problem definition.

1.2.2 CLASS DIAGRAM



1.3 SAMPLE RUNNING RESULTS:

Running results for step by step the production phases of every model from every market.

```

Main x
C:\Program Files\Java\jdk-14.0.2\bin\java.exe -javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2020.3\lib\idea_rt.jar
Producing Turkey Style MaximumEffort Model
Attached Cpu & Ram that is 2.8GHz, 8GB ,8 cores to the board.
Attached display that is 5.5 inches , 32 bit
Attached battery that is 27h, 3600mAh , Lithium_Boron
Attached storage that is MicroSD support, 64GB , Max 128 GB
Attached camera that is 12Mp front, 8Mp rear , Opt. Zoom x4
Enclosed the phone case that is 151x73x7.7 mm, dustproof, waterproof, aluminum , Waterproof up to 2m

Producing Turkey Style IflaxDeluxe Model
Attached Cpu & Ram that is 2.2GHz, 6GB ,8 cores to the board.
Attached display that is 5.3 inches , 32 bit
Attached battery that is 20h, 2800mAh , Lithium_Boron
Attached storage that is MicroSD support, 32GB , Max 128 GB
Attached camera that is 12Mp front, 5Mp rear , Opt. Zoom x4
Enclosed the phone case that is 149x73x7.7 mm, waterproof, aluminum , Waterproof up to 2m

Producing Turkey Style I-I-Aman-Ifilas Model
Attached Cpu & Ram that is 2.2GHz, 4GB ,8 cores to the board.
Attached display that is 4.5 inches , 32 bit
Attached battery that is 16h, 2000mAh , Lithium_Boron
Attached storage that is MicroSD support, 16GB , Max 128 GB
Attached camera that is 8Mp front, 5Mp rear , Opt. Zoom x4
Enclosed the phone case that is 143x69x7.3 mm, waterproof, plastic, Waterproof up to 2m

Producing Global Style MaximumEffort Model
Attached Cpu & Ram that is 2.8GHz, 8GB ,2 cores to the board.
Attached display that is 5.5 inches , 24 bit
Attached battery that is 27h, 3600mAh , Lithium-Cobalt
Attached storage that is MicroSD support, 64GB , Max 32 GB
Attached camera that is 12Mp front, 8Mp rear , Opt. Zoom x2
Enclosed the phone case that is 151x73x7.7 mm, dustproof, waterproof, aluminum , Waterproof up to 50cm

Producing Global Style IflaxDeluxe Model
Attached Cpu & Ram that is 2.2GHz, 6GB ,2 cores to the board.
Attached display that is 5.3 inches , 24 bit
Attached battery that is 20h, 2800mAh , Lithium-Cobalt
Attached storage that is MicroSD support, 32GB , Max 32 GB
Attached camera that is 12Mp front, 5Mp rear , Opt. Zoom x2
Enclosed the phone case that is 149x73x7.7 mm, waterproof, aluminum , Waterproof up to 50cm

Producing Global Style I-I-Aman-Ifilas Model
Attached Cpu & Ram that is 2.2GHz, 4GB ,2 cores to the board.
Attached display that is 4.5 inches , 24 bit
Attached battery that is 16h, 2000mAh , Lithium-Cobalt
Attached storage that is MicroSD support, 16GB , Max 32 GB
Attached camera that is 8Mp front, 5Mp rear , Opt. Zoom x2
Enclosed the phone case that is 143x69x7.3 mm, waterproof, plastic, Waterproof up to 50cm

```

```

Producing EU Style MaximumEffort Model
Attached Cpu & Ram that is 2.8GHz, 8GB ,4 cores to the board.
Attached display that is 5.5 inches , 24 bit
Attached battery that is 27h, 3600mAh , Lithium-Ion
Attached storage that is MicroSD support, 64GB , Max 64 GB
Attached camera that is 12Mp front, 8Mp rear , Opt. Zoom x3
Enclosed the phone case that is 151x73x7.7 mm, dustproof, waterproof, aluminum , Waterproof up to 1m

Producing EU Style IFlaxDeluxe Model
Attached Cpu & Ram that is 2.2GHz, 6GB ,4 cores to the board.
Attached display that is 5.3 inches , 24 bit
Attached battery that is 20h, 2800mAh , Lithium-Ion
Attached storage that is MicroSD support, 32GB , Max 64 GB
Attached camera that is 12Mp front, 5Mp rear , Opt. Zoom x3
Enclosed the phone case that is 149x73x7.7 mm, waterproof, aluminum , Waterproof up to 1m

Producing EU Style I-I-Aman-IfLas Model
Attached Cpu & Ram that is 2.2GHz, 4GB ,4 cores to the board.
Attached display that is 4.5 inches , 24 bit
Attached battery that is 16h, 2000mAh , Lithium-Ion
Attached storage that is MicroSD support, 16GB , Max 64 GB
Attached camera that is 8Mp front, 5Mp rear , Opt. Zoom x3
Enclosed the phone case that is 143x69x7.3 mm, waterproof, plastic, Waterproof up to 1m

```

2 QUESTION 2

2.1 PROBLEM DEFINITION

There are 2 payment type that are ModernPayment and TurboPayment. We want to use ModernPayment for payments but we also have to use TurboPayment for payments. Our goal is implementing a java a design pattern so that we can continue using all the classes implementing the TurboPayment interface with our new ModernPayment interface. And also we cannot modify the interface TurboPayment or the classes that implement it.

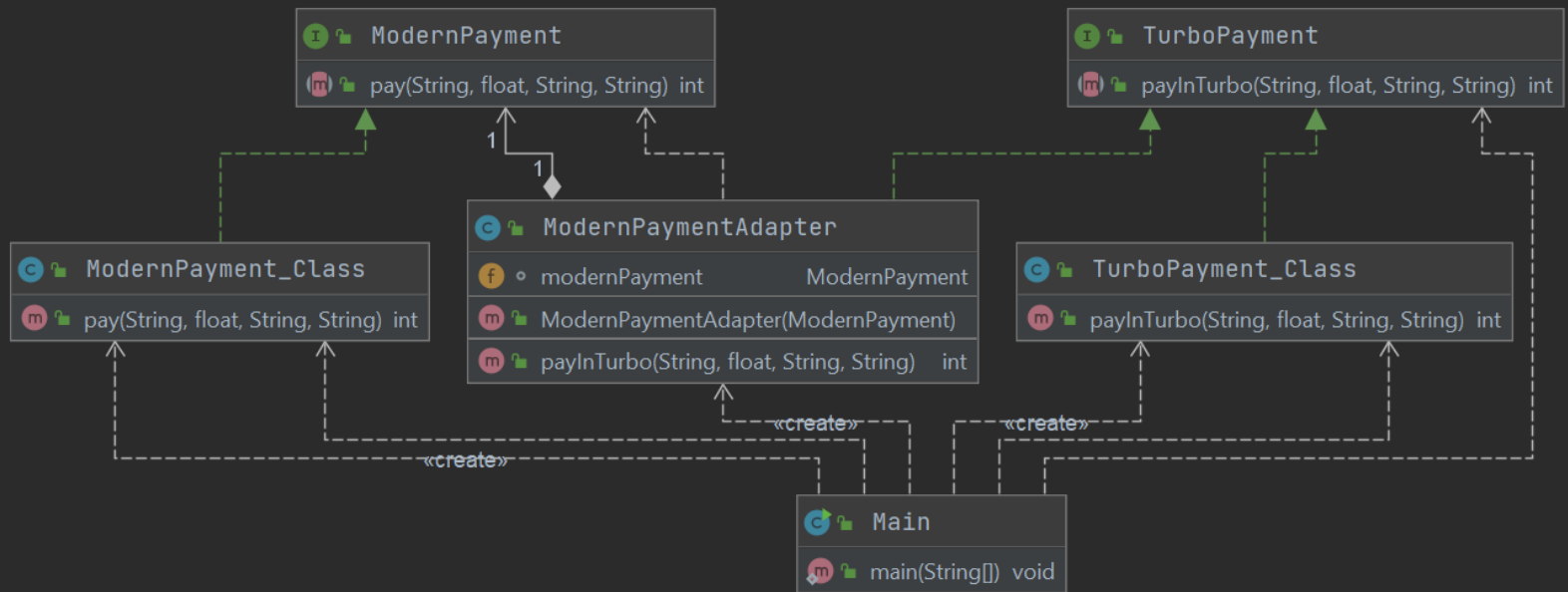
2.2 METHOD

2.2.1 Selected Design Pattern and Why It was Selected?

I select Adapter Design Pattern for this goal.Because;
The Adapter Design Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.In our problem we want to use 2 payment styles work together and adapter design pattern provides this

goal. By using ModernPayment Adapter Class that implements the target TurboPayment interface and holds instance of Adaptee that is ModernPayment interface. And this adapter class translates request so ModernPayment interface can be used with TurboPayment interface.

2.2.2 CLASS DIAGRAM



2.3 SAMPLE RUNNING RESULTS:

```
public static void main(String[] args) {  
    ModernPayment_Class modernPayment = new ModernPayment_Class();  
    TurboPayment_Class turboPayment = new TurboPayment_Class();  
    TurboPayment modernPaymentAdapter = new ModernPaymentAdapter(modernPayment);  
  
    System.out.println("The Actual Modern Payment Class payments: ");  
    modernPayment.pay( cardNo: "1999" , amount: 200 , destination: "Bahar" , installments: "1224");  
    System.out.println("The Actual Turbo Payment Class payments: ");  
    turboPayment.payInTurbo( turboCardNo: "1998" , turboAmount: 100 , destinationTurboOfCourse: "Mehmet" , installmentsButInTurbo: "1293");  
  
    System.out.println("\nThe Modern Payment Adapter Class payments:");  
    modernPaymentAdapter.payInTurbo( turboCardNo: "1997" , turboAmount: 300 , destinationTurboOfCourse: "Aysun" , installmentsButInTurbo: "1223");  
}
```

```
C:\Program Files\Java\jdk-14.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2020.3\lib\idea_rt.jar=55426:C:\Progra  
The Actual Modern Payment Class payments:  
~~~~~In concrete Modern Payment Class~~~~~  
Modern Payment Operation is doing with card no: 1999 amount: 200.0 destination: Bahar installments: 1224  
  
The Actual Turbo Payment Class payments:  
~~~~~In concrete Turbo Payment class~~~~~  
Turbo Payment Operation doing with card no: 1998 amount: 100.0 destination: Mehmet installments: 1293  
  
The Modern Payment Adapter Class payments:  
~~~~~In concrete Modern Payment Class~~~~~  
Modern Payment Operation is doing with card no: 1997 amount: 300.0 destination: Aysun installments: 1223  
  
Process finished with exit code 0
```

By using Adapter Class in Adapter Design Pattern ModernPayment pay type can be used. And also by Adapter Design Pattern all the classes implementing the TurboPayment interface can also be used. So 2 interface can work together.

3 Question 3

3.1 PROBLEM DEFINITION

We are responsible database operations of bank. We must design a solution for modelling transactions and database operations. A database operation can be a SELECT, an UPDATE or an ALTER. A transaction is a series of operations (a SELECT followed by an ALTER, or an UPDATE followed by ALTER followed by SELECT, and so on). If one of the operations fail, all others must be reversed or discarded (known as rollback in database lingo).

3.2 METHOD

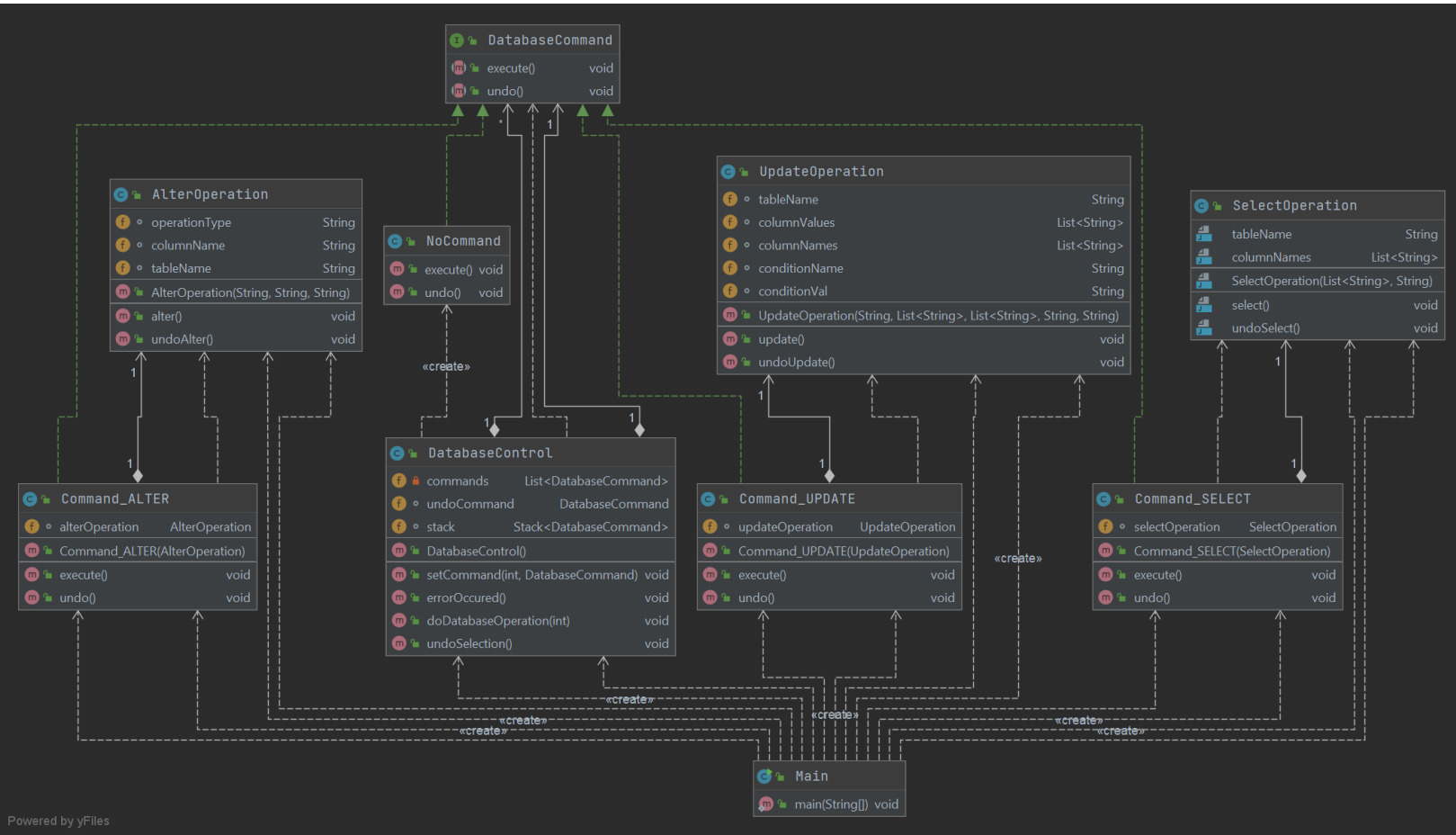
3.2.1 Selected Design Pattern and Why It was Selected?

I select Command Design Pattern for this goal. Because;
The Command Design Pattern encapsulates a request as an object, thereby letting us parameterize other objects with different requests, queue or log requests, and support undoable operations. In our project, we must do bank operations with transaction and if fail, all of operations must be reversed. For undoable feature of command design pattern, we provide this goal perfectly. And also in command design pattern; From the outside, no other objects really know what actions get performed on what receiver; they just know that if they call the execute() method, their request will be serviced. This feature important for flexibility and loose coupling. Because of this features I select Command Design Pattern for this goal.
In another page, I added Class Diagram and I explained Command Design Pattern implementation for this problem via Class Diagram.

3.2.2 Detail to how I Implement the Rollback of Transactions

For providing rollback of Transaction, I use stack. When an operation is made (select, alter, update) I add this command in stack. When an error occurred and there must be rollback of all operations, my invoker pops all items off the stack and calls its undo(). In this way, all operations performed are undone.

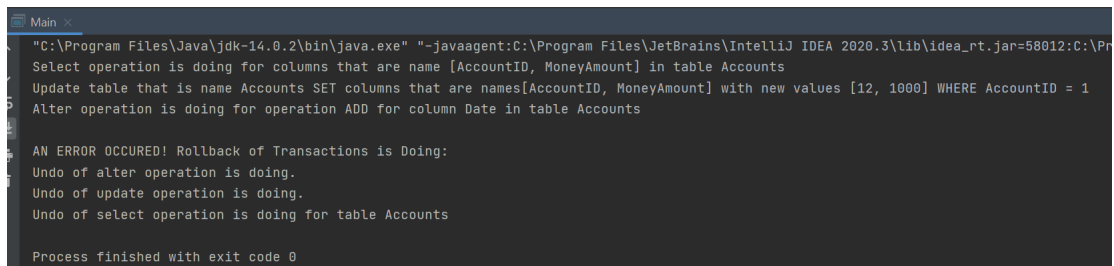
3.2.3 Class Diagram



- **DatabaseControl Class:** This class is Invoker class in Command Design Pattern.
- **DatabaseCommand Interface:** Command interface of Command Design Pattern. All command objects implement this interface. `execute()` method performs action. `undo()` method reverse/undo an action.
- **Command_ALTER - Command_SELECT - Command_UPDATE Classes:** These classes are Concrete Command Classes. They define a binding between an action and receiver in Command Design Pattern.

- **SelectOperation- UpdateOperation- AlterOperation Classes:** These classes are Receiver classes in Command Design Pattern. These classes know how to perform the work needed to carry out the request. SelectOperation Class make select and undo select operation. AlterOperation Class make alter and undo alter operation. UpdateOperation Class make update and undo update operation.
- **NoCommand Class:** This Concrete NoCommand Class does nothing. This class is using when initializing commands in DatabaseControl class.
- **Main Class:** This class testing and working class.

3.3 SAMPLE RUNNING RESULT:



```

Main
"C:\Program Files\Java\jdk-14.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2020.3\lib\idea_rt.jar=58012:C:\Pr
Select operation is doing for columns that are name [AccountID, MoneyAmount] in table Accounts
Update table that is name Accounts SET columns that are names[AccountID, MoneyAmount] with new values [12, 1000] WHERE AccountID = 1
Alter operation is doing for operation ADD for column Date in table Accounts

AN ERROR OCCURED! Rollback of Transactions is Doing:
Undo of alter operation is doing.
Undo of update operation is doing.
Undo of select operation is doing for table Accounts

Process finished with exit code 0

```

Explanation of Output: Select, Update ,Alter operations doing in sequence. When error occurred Rollback of Transaction is doing. From stack all commands pop.

4 QUESTION 4

4.1 PROBLEM DEFINITION

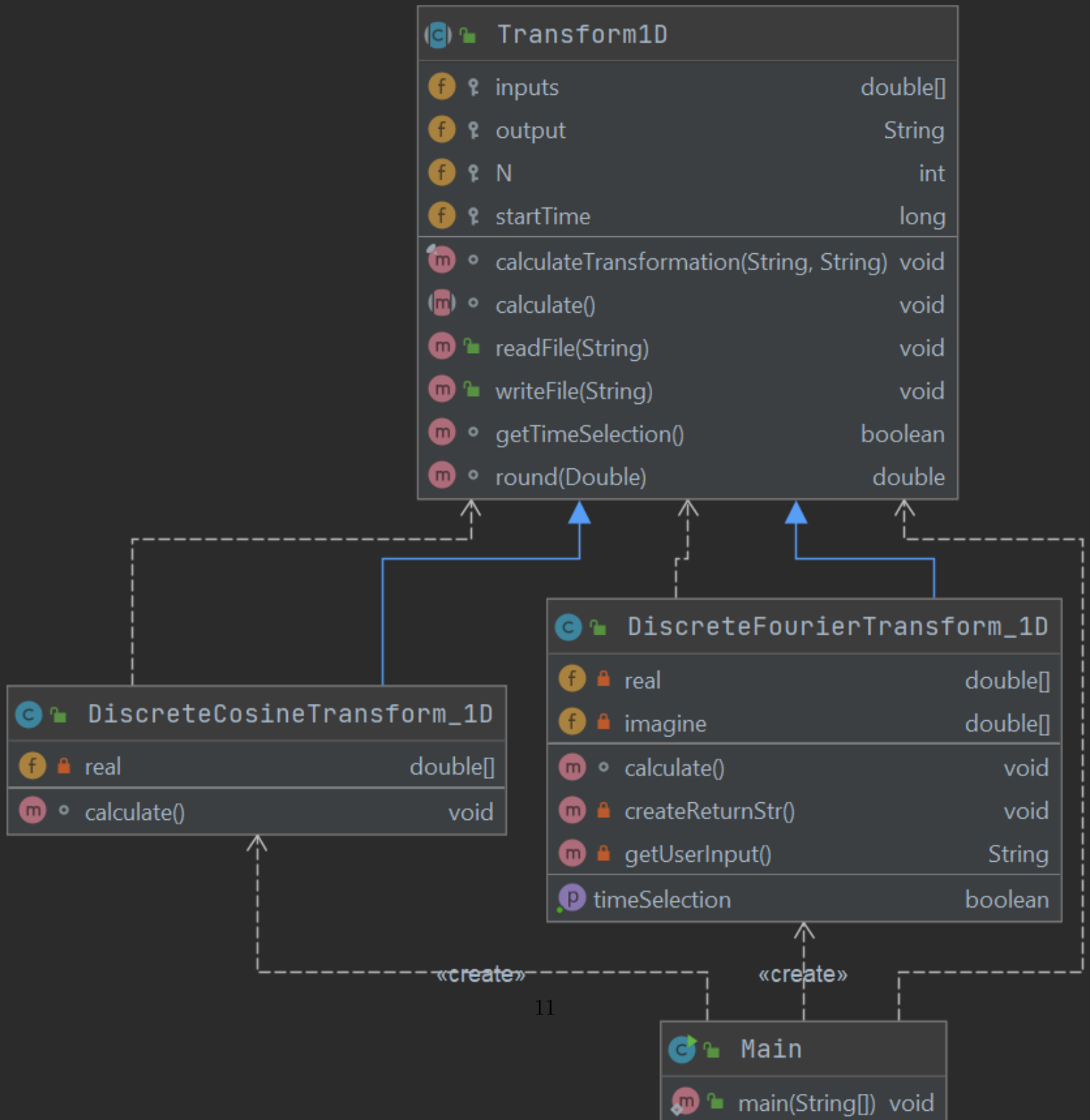
There are 2 algorithms that are Discrete Fourier Transform and Discrete Cosine Transform and two of them make same thing. They take finite sequence of numbers and after they express them as a sum of basis functions. Their difference is Discrete Fourier Transform uses complex exponentials for transformation and Discrete Cosine Transform uses real-valued cosine functions for transformations. Discrete Fourier Transform produces complex numbers and Discrete Cosine Transform produces real numbers. Both of them trace same way: Read N tab separated numbers from a file provided as a command line argument, transform the numbers into N outputs (DFT or DCT outputs.) , write the N outputs to a new file, only in the case of DFT sometimes the user wants additionally the time of execution printed on screen; by default NO. Our goal is implementing these things by using Template Method Design Pattern.

4.2 METHOD

4.2.1 Why Template Method Design Pattern was Selected?

The Template Method Design Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. In our project both of two algorithms trace same way (actually algorithm); Read N tab separated numbers from a file provided as a command line argument, transform the numbers into N outputs (DFT or DCT outputs.) , write the N outputs to a new file. Only difference is transformation method. By using template design pattern, DCT and DFT subclasses redefine transformation method of an algorithm without changing the algorithm's structure. And also by using hook method in template design pattern, only in the case of DFT, sometimes the user wants additionally the time of execution printed on screen; by default NO goal is provided. For this reasons, template method design pattern was selected.

4.2.2 CLASS DIAGRAM



4.3 SAMPLE RUNNING RESULTS:

NOTE: I use same read function for DCT and DFT algorithms so input file is always contain real numbers. And also I assume that imagine part is 0 for DFT calculations and I apply DFT formula by assumption imagine part is zero beause of reading real numbers from file.

Input File:

inputFile.txt						
1	6	9	2.5	6.7	-4	8
2						
3						

Discrete Cosine Transform Result:

output.txt						
1	28.2	6.173	4.157	-7.637	6.6	-13.767
2						

Discrete Fourier Transform Result:

output2.txt								
1	28.2	8.55	-6.495i	4.95+4.763i	-19.2	4.95	-4.763i	8.55+6.495i

If user wants getting execution time for DFT:

```
"C:\Program Files\Java\jdk-14.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA\bin\idea-agent.jar" -jar DFT.jar
Successfully wrote to the file.
Successfully wrote to the file.
Would you get the time of execution for Discrete Fourier Transform? (write yes or no)
yes
The execution time is: 3570633800 ns
Process finished with exit code 0
```

If user does not want getting execution time for DFT:

```
"C:\Program Files\Java\jdk-14.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA\bin\idea-agent.jar" -jar DFT.jar
Successfully wrote to the file.
Successfully wrote to the file.
Would you get the time of execution for Discrete Fourier Transform? (write yes or no)
no
Process finished with exit code 0
```

4.4 References:

Eric Freeman & Elisabeth Robson with Kathy Sierra and Bert Bates ; Head
First Design Patterns.

https://en.wikipedia.org/wiki/Abstract_factory_pattern

https://en.wikipedia.org/wiki/Command_pattern

https://en.wikipedia.org/wiki/Adapter_pattern

https://en.wikipedia.org/wiki/Template_method_pattern