GEBZE TECHNICAL UNIVERSITY
COMPUTER ENGINEERING

OBJECT ORIENTED ANALYSIS and DESIGN
CSE 443 - 2020 FALL

MIDTERM REPORT

SILA BENGİSU YÜKSEKLİ
1801042877

PART1 - PROGRAM TO PRODUCE SMART PHONE

➢ *Explanation of design pattern*

I used the abstract factory design pattern for this developed program. The reason I use this design is that the components used in manufacturing the phone are different in every market. Therefore, the abstract factory design pattern should be used instead of the factory design pattern. This pattern provides interfaces without indicating concrete classes of related objects with each other to create their family. So, It has made the program more useful as it reduces the maintenance cost and reduces the dependency between classes.

If a new phone model is to be produced, all that needs to be done is to derive itself from the SmartPhone class and add a new if statement for each market class. If the phones are to be sold in a new market, what needs to be done again is to implement all component (display, case ...) interfaces for each component to be produced in that market and to create a new market by implementing the SmartPhoneComponentFactory. Thus, this design offers a good loosely coupled system.

➢ *Explanation of classes and interfaces*

An abstract class called SmartPhone has been written to represent the smartphone. Phone models (MaximumEffort, IflasDeluxe...) derive from this class and keep a reference for the factory where the phone components will be produced. Thus, the model to be ordered is produced according to the market in which it is sold. This production process is implemented in each subclass's produceComponent method.

Separate interfaces are written for each component like display, camera ... Classes such as TurkeyTypeDisplay, EUTypeCamera... enable these components to be produced in their own specific type for each market. If one phone is to be sold for example in Turkey, then TurkeyTypeDisplay object is used.

An interface called SmartPhoneComponentsFactory is written and each factory (e.g. TurkeySmartPhoneComponentFactory) implements this interface so that the features of the components to be produced are separated from each other.

Also, an abstract class called SmartPhoneMarket was written and all the markets like markets in Turkey, EU etc. derived from this class and implement produceSmartPhone method so that according to the given string which indicates the model of the phone, components of the phone are produced and they are constructed in the orderSmartPhone method. As a result the phone can be delivered successfully.

➢ *Execution Process*

1. **... = new TurkeySmartPhoneMarket();** -> Creates TurkeySmartPhoneMarket object
2. **marketInTurkey.orderSmartPhone(MaximumEffortModel)->**First produceSmartPhone method is called in this method
3. TurkeySmartPhoneComponentFactory object is create, because the market is in Turkey
4. Then according to the smart phone model, it goes into if statement and it creates MaximumEffortModel phone.
5. This phone model calls produceComponents method.
6. In this method all the parts of the smartphone are produced.
7. Then the construction process begins, like attaching CPU&RAM to the board.
8. And smartphones are produced and delivered successfully with the correct model and components.

➢ *Class Dependencies and UML diagram*

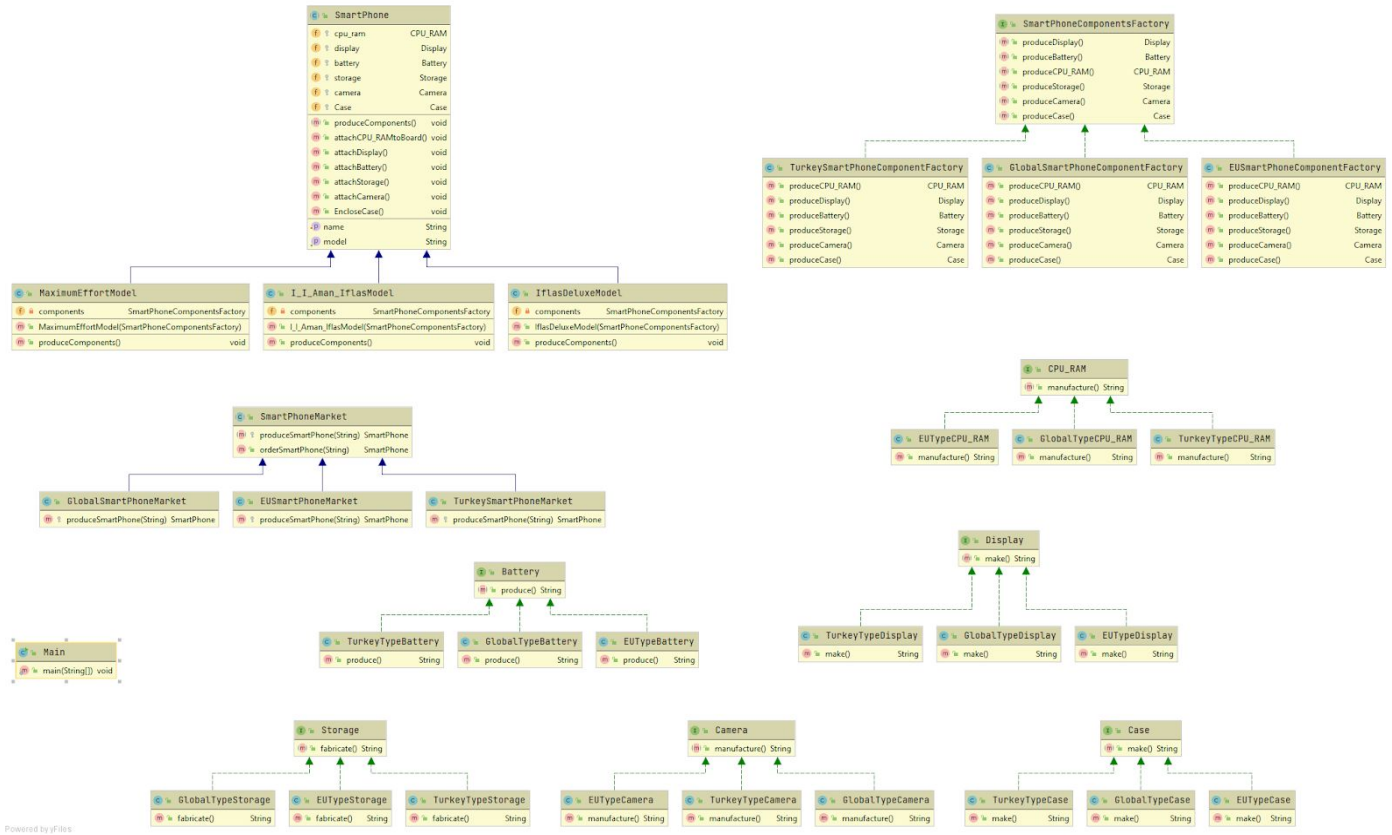

figure1.1 *UML diagram of the program*

Since the diagram showing the dependency between classes and interfaces is very complex, I also included the diagram above that does not show dependencies. Also, these images of diagrams are included in the homework folder.

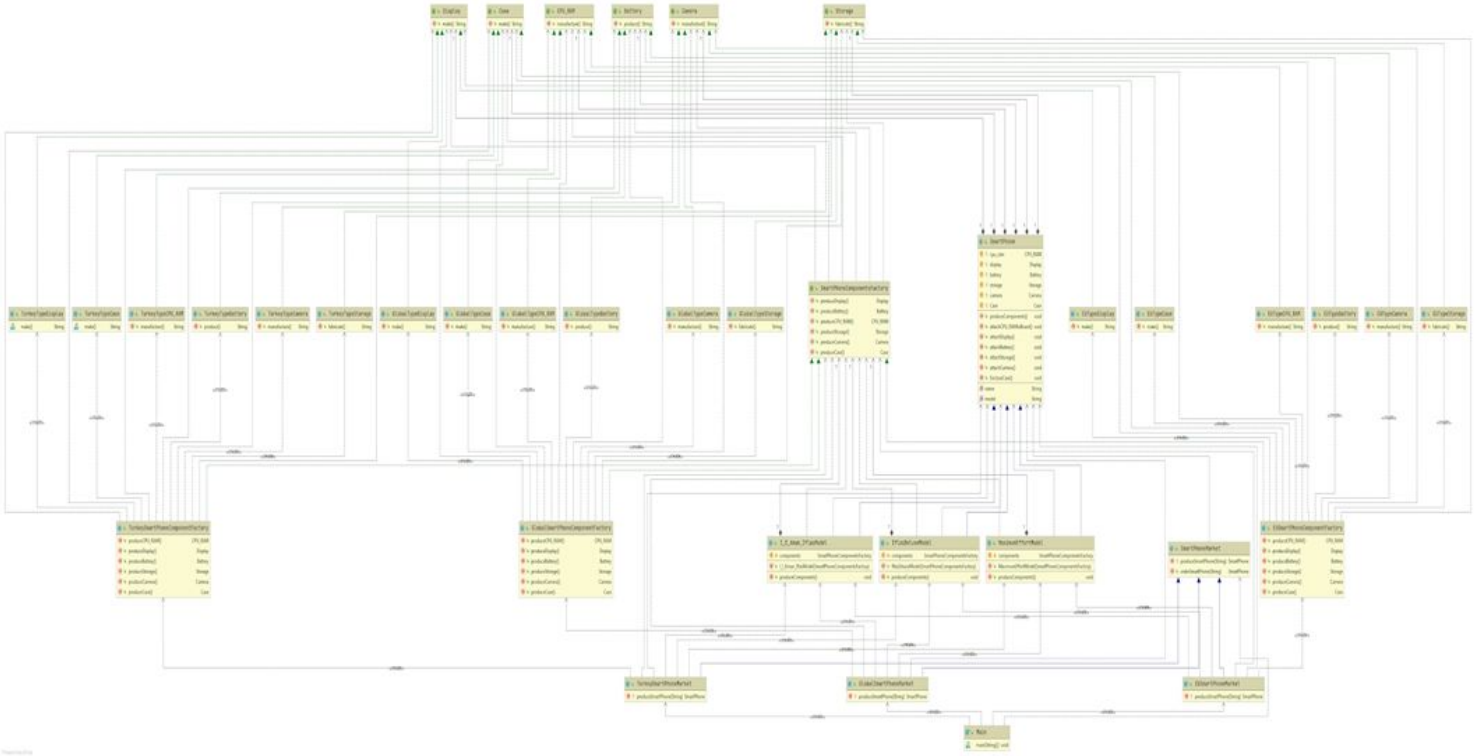

figure1.2 *UML diagram of the program that shows dependencies*

PART2 - PROGRAM TO ADAPT THE NEW INTERFACE TO THE OLD INTERFACE

➢ *explanation of design pattern*

I used the adapter design pattern for this developed program.In this scenario I have two interfaces that are incompatible with each other and I can't change the old one. So I have to make them compatible with each other and this design can provide this. Because the adapter design pattern lets classes work together that couldn't before.

➢ *explanation of classes and interfaces*

There are two interfaces called TurboPayment and ModernPayment. The first one is written before and it has a binary form, it can not be changed. They both have a method and all the parameters represent the same thing. To show the usage of these methods I derived two classes from them that are turboTypeCard and modernTypeCard classes. Just for the sake of the example, an information of the credit card payment is printed on the screen.

To make interfaces compatible with each other, an adapter class was written which is ModernPaymentAdapter. This class implements the TurboPayment interface and keeps a reference to ModernPayment object. So in the payInTurbo method which belongs to TurboPayment interface, pay method of ModernPayment object is called.As a result both the old and the new interfaces become compatible and can be used.

➢ *Execution Process*

To test methods, I created two objects of the turboTypetCard and modernTypeCard and called their payInTurbo and pay method.

Then I created an adapter object to test if it works correctly and called its method as shown in below:

```
turboTypeCard turbo_payment = new turboTypeCard();
TurboPayment modern_adapter = new ModernPaymentAdapter(modern_payment);
```

Also I wrote a static method that takes a parameter in the type of TurboPayment. In this method the payInTurbo method is called.

In the both results of the execution, the pay method is worked although the adapter class implements the TurboPayment interface. So the program executed correctly.

➢ *Class Dependencies and UML diagram*

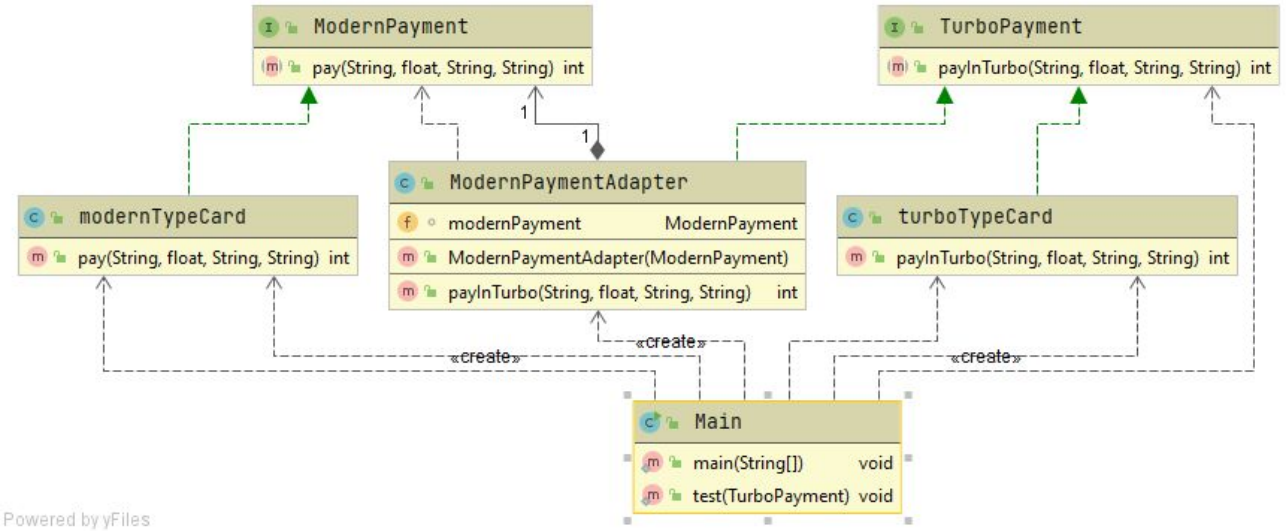

figure 2.1 *UML diagram of the program*

PART3 - PROGRAM TO MAKE THE TRANSACTIONS AND DATABASE OPERATIONS TO BE MADE IN THE BANK

➢ *Explanation of design pattern*

The aim of this program is to design transaction and database operations. A database operation can be select, update or alter; on the other hand transactions can be a series of these database operations. Actually all of them represent a command. The Command Pattern encapsulates a request as an object, thereby letting you parameterize other objects with different requests. The proper pattern for this program is the command design pattern. So in this program command objects (such as select, update) encapsulate a request by binding together a set of actions on the customer list. In the execute() method of every command, some operation is done on the customer table. Other objects don't know what is going on in this method, they only know if some object calls the execute function, its request will be done. Thus, this design offers a good loosely coupled system. If additional features would be needed in the future, the only thing to do is to write new command classes by implementing from the Command interface.

➢ *Explanation of classes and interfaces*

Database is represented as a two dimensional String array and all the commands perform operation on this array. Initially there are four customers of the bank and the columns of the table represent the name of the customer, address of the customer and the total assets that the customer has.

There is a command interface and it has two methods which are execute and undo. The execute method does what the command should do and the undo method undid the operation.

There are five subclasses that implement this interface.

1. **Select** : This class represents the select command. This command was defined as selecting a customer from the list. In the execute method of this class, an information message is printed to show that the select command works. In the undo method, It just prints the information message to show that the select command is undid.

2. **Update :** This class represents the update command.This command was defined as updating an information of a customer from the list. For example when the customer deposits money in the bank, an update command is called and it changes the amount of money in the database. In the execute method, it prints the information of the customer after the table was updated.In the undo method update process is canceled and prints the customer's information

3. **Alter :** This class represents the alter command. This command was defined as adding a new information column about customers to the list. In the execute method of this class a new column is added to the list; in the undo method of this class the last column that is added, is deleted. For example e-mail addresses can be added by using this command.

4. **Transaction :** This class performs the situation where multiple commands function at the same time. For instance to achieve an operation, select and then update commands need to be done or it can be alter, select and update. It is a macro command, thus it holds a command array. In the execute method of this class, all the execute methods of the command object that are held in array are called. In the undo method the same thing is done by calling their undo method. I assume depositing money in the bank occurs in two steps : Select then Update. It is an example of a transaction.

5. **NoCommand :** This class behaves like a command that does nothing. It is used for getting rid of checking a command is null every time.

I wrote a class which is depositMoneyInBank. It behaves like an invoker in the command pattern. I assume depositing money in the bank has two steps : select and update. So according to this assumption, the commands array which is held as a data member has the size of two. In the set method, elements of the array are assigned. In the DepositMoneyInBank method, execute methods of the commands are called and operation is completed successfully. Also this class has a method which is cancelOperation. It is just for test the error case. A detailed explanations are in below :

■ *Explanation of rollback in case of an error*

I hold a Stack data structure as a data member. This structure holds the every command that is executed. I mean when a command is executed, after the execution is done, this command is pushed into the stack. The reason for holding stack and pushing commands in it is to provide rollback in case of an

error. I wrote a method which is cancelOperation to test the error case. Let's say an error is occured during the operation. Then this method will work and until the stack is empty, it removes the commands from stack and calls undo methods of every command that it pops. Thus, the interrupted transaction is taken back without causing any trouble.

➢ *Execution Process*

- To test the program, I created different commands which are select, update, transaction and alter.
- Alter command added a new column to indicate email addresses of the customers.
- The transaction object consists of two commands which are select and update.
- Also, I created two depositMoneyInBank objects. First one uses the transaction object and second one uses select and update objects separately. DepositeMoneyInBank method is called and information is updated of the selected customer.
- To test the error case, I called cancelOperation method dynamically. Let's say customer1 wants to deposit money and a problem occurred before the process was completed. Then cancelMethod will work and it rollbacks the process by calling the undo method of the commands that are popped from stack. After this method worked, the information that shows the amount of the money will be the same with the old amount.

➢ *Class Dependencies and UML diagram*

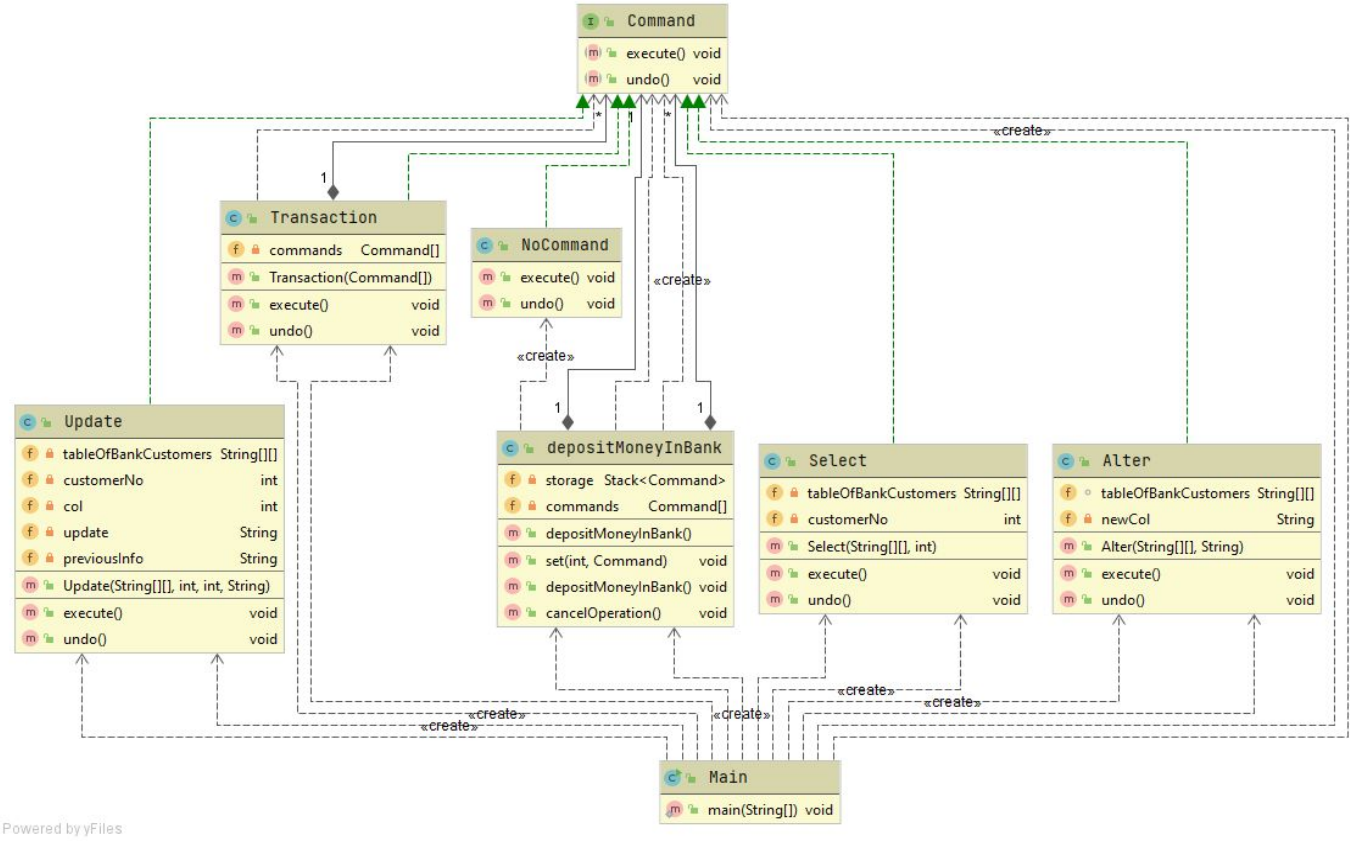

figure 3.1 *UML diagram of the program*

PART4 - PROGRAM TO MAKE 1D TRANSFORMATION USING DIFFERENT ALGORITHMS

➢ *Explanation of design pattern*

The task is to develop 1D transformation algorithms that are Discrete Fourier Transform and Discrete Cosine Transform.Both of the algorithms have common steps which are reading samples from file and writing the result to a file. So I used the template design pattern for this developed program. This pattern provides us to encapsulate algorithms. If in the future an additional 1D transformation wants to be used, it can be implemented easily without changing anything in other classes. Thus, anything new for this program will be easy to integrate and the maintenance cost will be low.

➢ *Explanation of classes and interfaces*

I wrote an abstract class which is called Transformations. It implements the performsProcess, readFromFile and writeToFile methods. All of them are common for transformation operations. performsProcess method collects all the necessary functions to perform the transformation. calculateResultOfTransformation is an abstract method. It carries out the mathematical steps and since the algorithms are different, each subclass implements this method in its own way. Also, the Transformations class has a hook method which is called userWantExecutionTime. This method returns false by default. Subclasses can override it to differentiate its behavior.

There are two subclasses in our situation that are DFT and DCT_II. Both of them implement calculateResultOfTransformation. In the case of DFT, hook method is overridden. There is another method called getUserInput. It returns the answer of the user read from the keyboard. User should enter “yes” or “no” to the question that asks if the user wants to know the execution time of the process. According to the answer userWantExecutionTime returns true or false.

➢ *Execution Process*

To test algorithms, I created objects of both DFT and DCT_II. Then the performProcess method is called. This method achieves these steps :

1. It calls the readFromFile method. This method reads values of the samples that existed in the file. This filename is given as a command line argument.

All the values are tab separated and all of them are real numbers. Because the DCT_II algorithm works with only real numbers. So in this program one file is used for both algorithms.

2. It calls calculateResultOfTransformation method. This method carries out the mathematical steps. It calculates the result for every sample and saves them in imagOutput and realOutput. In the case of DCT_II imaginary outputs will be zero directly.
3. It writes the results to the file. In the driver class the second test is a comment. Because the writeToFile method creates the same file for algorithms. If I test both of them at the same time, results of the DCT_II are overwritten to new results of the DFT. So you should try to test one by one.
4. Finally, it calls the hook method according to the algorithm. Since this method returns false by default, it does not ask the user if he/she wants to know execution time in the case of DCT algorithm

---

Driver class is shown below :

```
DCT_II dct_transformation = new DCT_II(args[0]);      dct_transformation.performProcess();
DFT dft_transformation = new DFT(args[0]);            //dft_transformation.performProcess();
```

➢ *Class Dependencies and UML diagram*

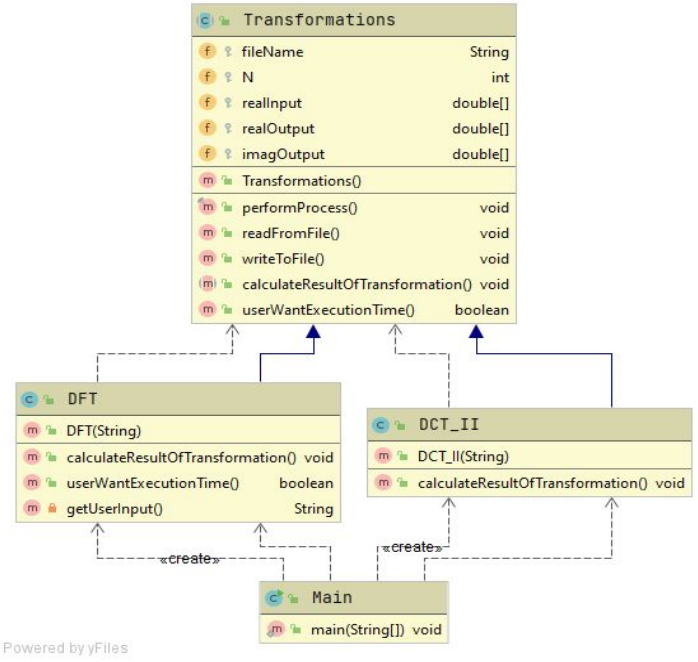

figure 4.1 *UML diagram of the program*

NOTES :

I used the formulas for DFT and DCT given in this link :

- *https://en.wikipedia.org/wiki/Discrete_Fourier_transform*
- *https://en.wikipedia.org/wiki/Discrete_cosine_transform#:~:text=A%20discrete%20cosine%20transform%20(DCT,signal%20processing%20and%20data%20compression*

The file should contain only real numbers because DCT works with only real numbers. Since I used one file for both of them, the file should contain only real numbers.

Because the file contains only real numbers, I removed some part of the formula in the DFT algorithm. Imaginary part is zero automatically. So I didn't write the part which multiplies with the imaginary part of the input.