**CSE 443**

# Object Oriented Analysis and Design

**Fall 2020– 2021**

**Midterm Report**

**Caner KARAKAŞ**

**131044061**

## Part 1

Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object. In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

As a result of the design we used, we created a lot of classes. We tried to gather these classes basically around 1 interface and 2 abstract classes. Of course, we have also created sub-interfaces and classes so that some parts under these three classes can be produced specifically for some regions. The basic 3 classes we have created; MobilePhoneFactory, MobileStore, Model.

MobilePhoneFactory represents production in 3 different regions. So what elements are produced in different regions. An interface for each element and subclasses for their differences were created. For example the Bit32Display and Bit24Display classes, which are the display class and its subclasses. The stores where these elements will be sold are considered under an abstract class. With this abstract class, the desired design has been created. However, the 3 fixed model types were collected under the Model class and the cost was considered in the addition of new add-on models and they were created accordingly.

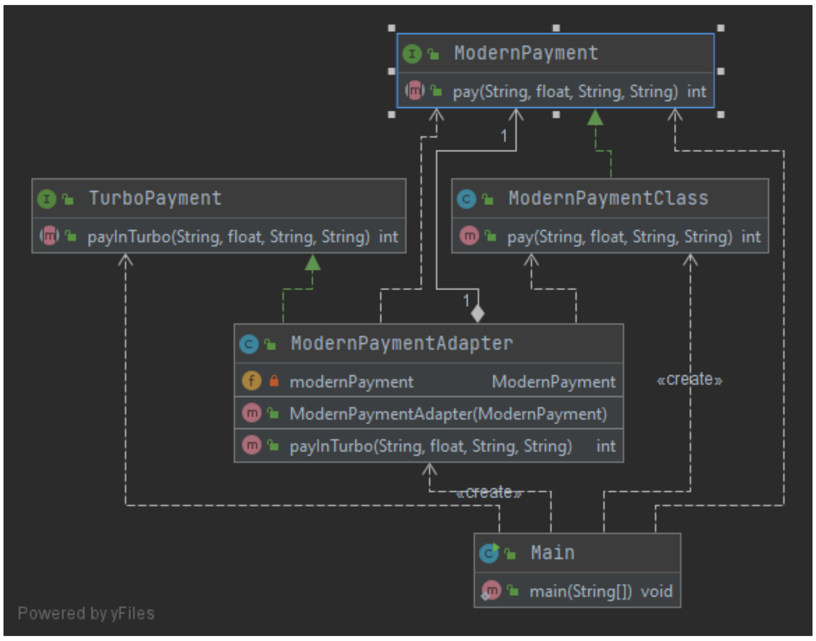
Three instances for each are made on the main function.

Since the uml diagram for this part does not appear in the report, it has been put in the diagrams folder in the assignment. Therefore, it was not included in the report.

Part 2

In some cases, the client may need to improve its software by making minor changes in the code you provide without breaking the pattern. Sometimes, the software can be developed independently of the interface you provide. For example, let's consider a rocket simulation program. This program can be developed according to the rocket information you provide, as well as the client may want to program how the rocket should be and how it should behave in which situations. You can use the Adapter model when you want to enable the client to use the services you offer directly or customize and service the method names. The purpose of the adapter is to create a structure where the services expected by the client can use the services of that class with a different interface.

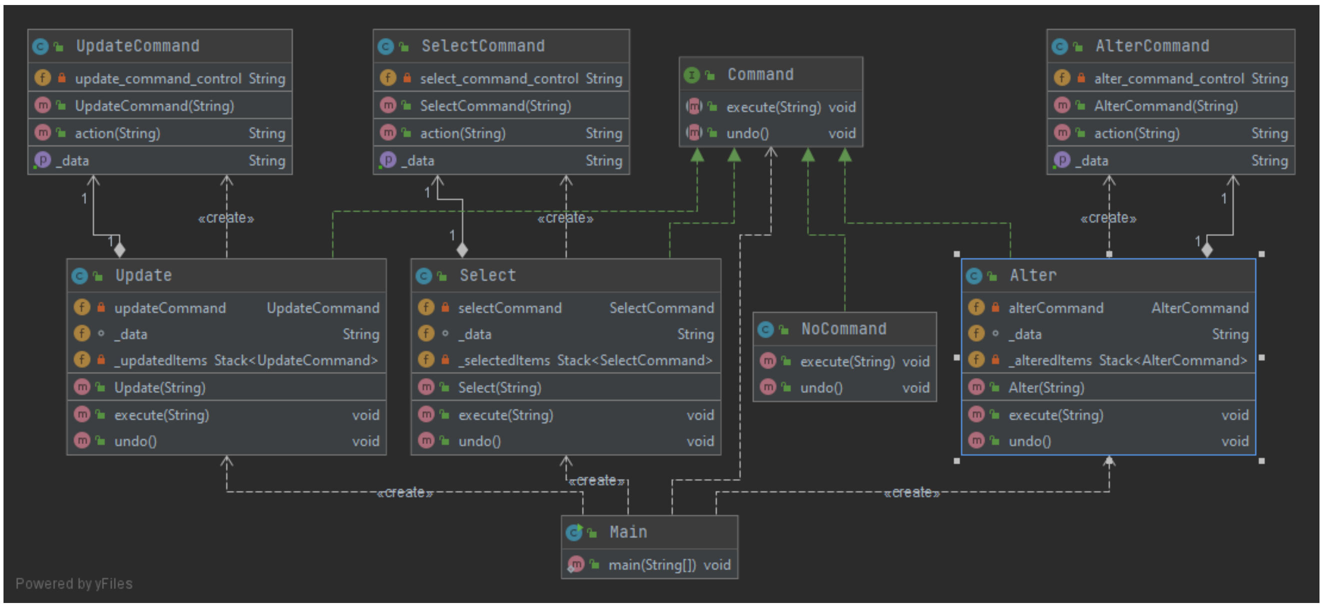
This method is used in this part. The TurboPayment interface shows the old method, while the ModernPayment interface shows the new method. ModernPaymentClass has implemented the modern method. ModernPaymentAdapter, on the other hand, acts as a transformer between the two methods. More precisely, it provides the opportunity to use the new method over the old method.


ModernPayment
pay(String, float, String, String) int
TurboPayment
payInTurbo(String, float, String, String) int
ModernPaymentClass
pay(String, float, String, String) int
ModernPaymentAdapter
modernPayment ModernPayment
ModernPaymentAdapter(ModernPayment)
payInTurbo(String, float, String, String) int
1
1
«create»
«create»
Main
main(String[]) void
Powered by yFiles

## Part 3

Command pattern is a data driven design pattern and falls under behavioral pattern category. A request is wrapped under an object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

As can be seen in this design, a Command interface has been created for your design. Select Alter and Update classes implemented from this interface actually act as a waiter. These classes used objects from the AlterCommand SelectCommand and UpdateCommand classes. He implemented the execute method by calling the action methods of these objects. These objects were then kept in a stack for the undu method. Besides, the undo method returns the last operation by deleting these objects from the stack.



Part 4

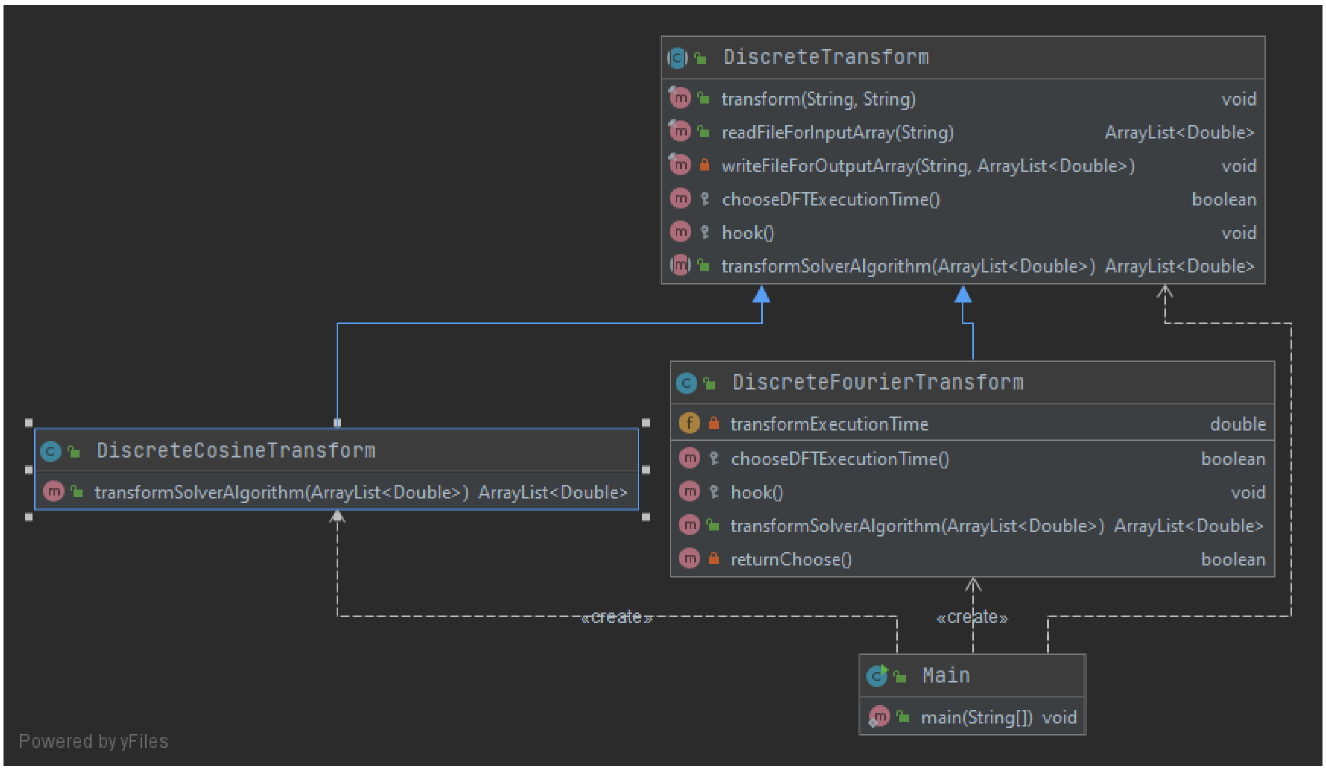
Template Method design pattern has two main parts:

The "template method" is implemented as a method in a base class (usually an abstract class). This method contains code for the parts of the overall algorithm that are invariant. The template ensures that the overarching algorithm is always followed.[1] In the template method, portions of the algorithm that may vary are implemented by sending self messages that request the execution of additional helper methods. In the base class, these helper methods are given a default implementation, or none at all (that is, they may be abstract methods).

Subclasses of the base class "fill in" the empty or "variant" parts of the "template" with specific algorithms that vary from one subclass to another.[3] It is important that subclasses do not override the template method itself.

The DiscreteTransform abstract class is the class in which we wrap the two methods. This class includes transform method, readFileForInputArray method, writeFileForOutputArray method, chooseDFTExecutionTime method, hook method, transformSolverAlgorithm abstract method. The transform method performs all the works sequentially. One of the two strings entered is the input file and the other is the output file. First, the numbers are read from the input file. The readFileForInputArray method implements this reading. these numbers are then kept in an ArrayList. These numbers are transformed with the transformSolverAlgorithm abstract method.

The purpose of the transformSolverAlgorithm abstract method to be abstract is that subclasses, that is, method classes, must fill this method. With this method, methods implement their own algorithms. DFT method, which is one of these methods, approaches the DCT method differently. Since the DFT method expects the numbers to be complex, it thinks that N numbers come in pairs and actually N / 2 numbers. This is not the case with the DCT method. In the DiscreteFourierTransform class, a question is asked to the user for the execution time that the desired user is likely to request. Here chooseDFTExecutionTime method is used. According to this method, base value is false. The hook method performs the hook process here. that is, the subclass using the optional state only uses the method that other classes do not ovirride.



DiscreteTransform
transform(String, String) void
readFileForInputArray(String) ArrayList<Double>
writeFileForOutputArray(String, ArrayList<Double>) void
chooseDFTExecutionTime() boolean
hook() void
transformSolverAlgorithm(ArrayList<Double>) ArrayList<Double>
DiscreteFourierTransform
transformExecutionTime double
chooseDFTExecutionTime() boolean
hook() void
transformSolverAlgorithm(ArrayList<Double>) ArrayList<Double>
returnChoose() boolean
DiscreteCosineTransform
transformSolverAlgorithm(ArrayList<Double>) ArrayList<Double>
«create»
«create»
Main
main(String[]) void
Powered by yFiles