GEBZE TECHNICAL UNIVERSITY
COMPUTER ENGINEERING

OBJECT ORIENTED ANALYSIS and DESIGN
CSE 443 - 2020 FALL

HW1 HOMEWORK REPORT

SILA BENGİSU YÜKSEKLİ
1801042877

PART1 :

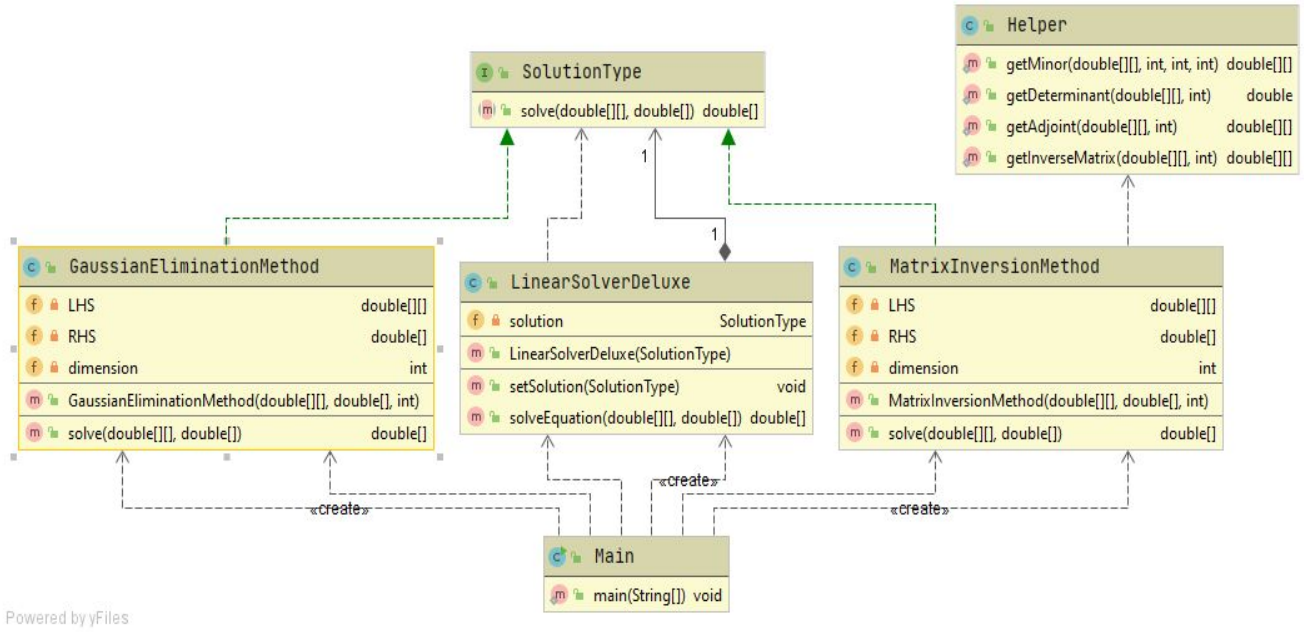

figure1.1 : class diagram of part 1

I used the strategy design pattern for this developed application. The reason I prefer this design is to create a has-A relationship by bringing together different solution behaviors under the SolutionType interface. It has made the program more useful as it reduces the maintenance cost according to the is-A relationship and reduces the dependency between classes. If the customer wants to create a new solution type later, the developer can code the new class by implementing the SolutionType interface. It doesn't need to change anything in other classes. Thus, the maintenance cost is reduced, the comprehensibility of the program for the software developer increases and it becomes easier to develop. Due to this design, solution type can be changed at run time and linear equations can be solved by using different methods.

LinearSolverDeluxe class holds a reference to a SolutionType object. It may be gaussian or inverse matrix method. In the future it can be a different method other than these too. To use one of these methods, all you have to do is change the solution method by calling the setSolutionType method in the background.

```
Please enter how many variables there are
3
Please enter the coefficients of variables
1
2
1
2
4
0
2
6
4
Please enter the values of the right hand side
10
20
40
Choose the solution type
Enter 1 to choose Gaussian Elimination Method
Enter 2 to choose Matrix Inversion Method
1
x0 : -10.0
x1 : 10.0
x2 : 0.0
Do you want to continue ? y/n
n
```

figure1.2 : example result of the program

First, you must enter number of the variables that the equation has. Then, you must enter the coefficients of the variables. After that you must enter the values on the right side of the linear equation. At the end, you should choose the solution type that you want and the program will print the solution, if any exist.

PART2 :

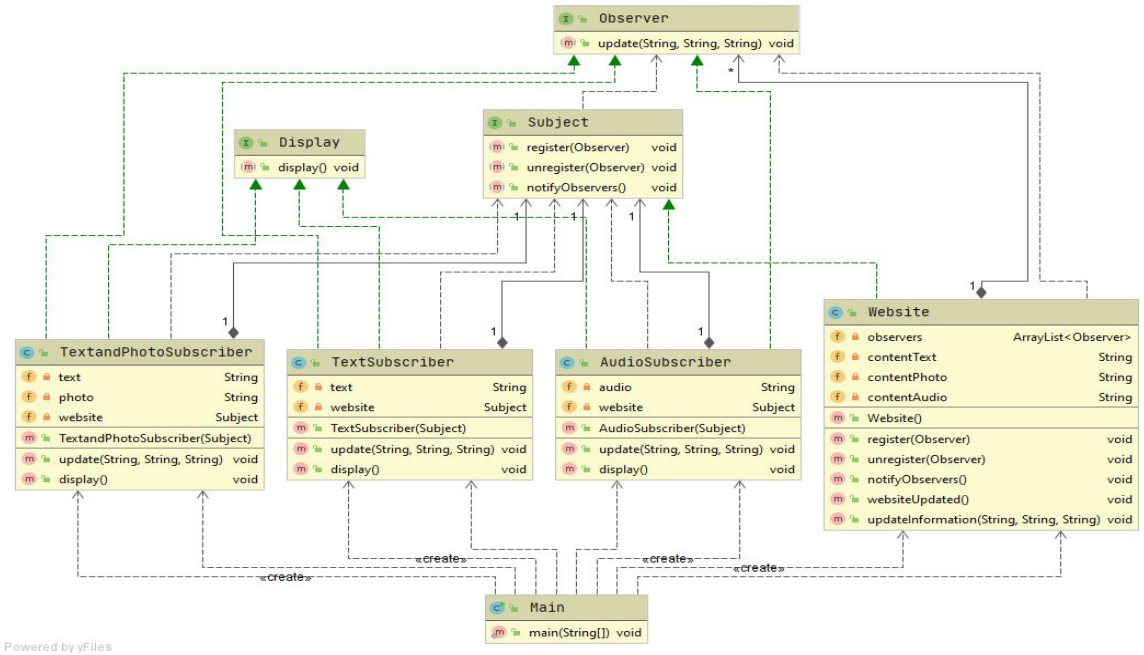

figure2.1 : class diagram of part 2

I used the observer design pattern for this developed application. The reason I chose this design is that there is a dependency from one to many in this scenario. The website class that implements the Subject interface is a site that shares the text, photo and music of the day. There may be multiple members of the site, some following only music, some text, or some both, and these members will want to be notified whenever a new update comes to the site regarding their area of interest. Therefore, the most suitable design for this situation is the observer pattern. Later on, the subscriber may unsubscribe or someone else can subscribe. This event is easily handled by using register and unregister methods. All the observers hold a subject reference and when an observer want to register itself, subject reference handles it. Notify method is used to inform all the observers when there is an update. Also, all the observers implement display interface to print the updated information to the screen. When an update occurs, subject notifies all the observers, but only the observers who receive updates in their area of interest call the display method and update it by using update method. If there is a change in the field that the subscriber is not interested in, subscriber does nothing.

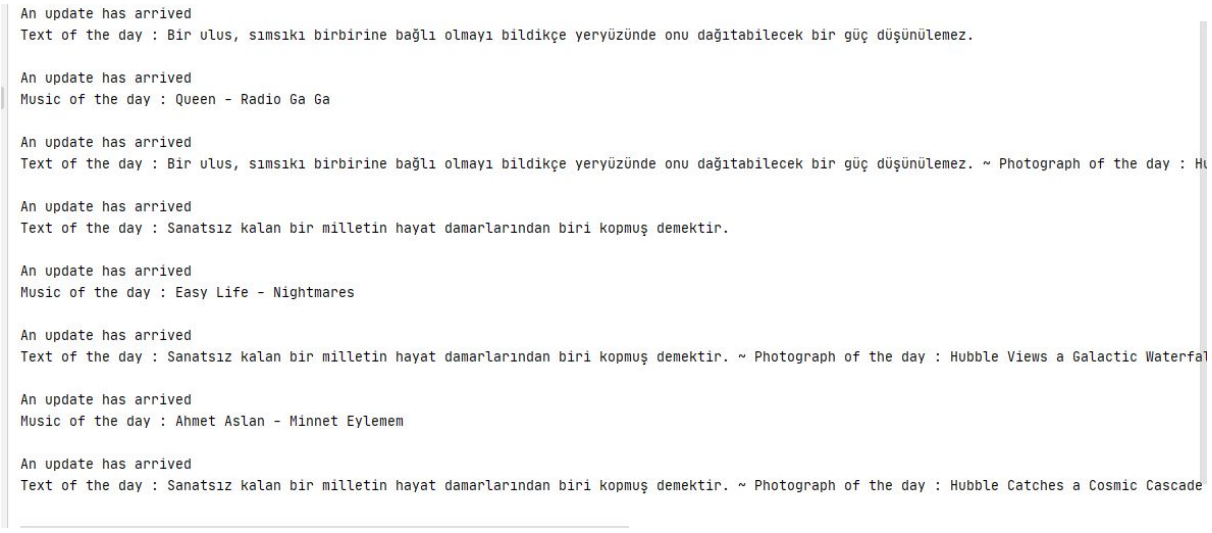

figure2.2 : example result of the program

I wrote a driver class to test the program. There are three observer, first one is interested in text, second one is interested in audio and the third one is interested in text and the photograph on the website. In the first case, since all the content is updated, all the observers display the information. In the second case, the situation is the same. However in the third scenario, only photograph and audio was updated. That's way only second and third observers display them, first observer do nothing.

PART3 :

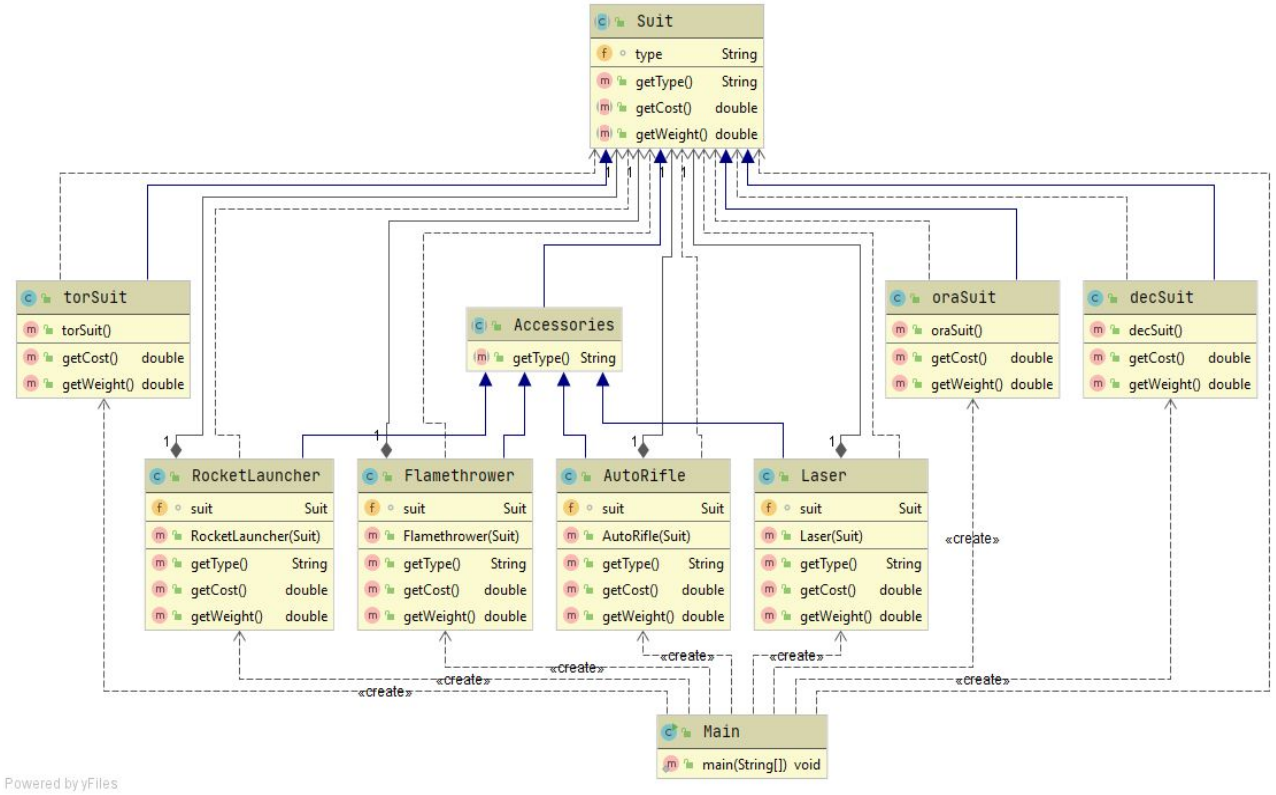

figure3.1 : class diagram of part 3

I used the decorator design pattern for this developed application. In this program many accessories can be wrapped by a specific suit or you can only buy suit without accessories. So each suit can be used on its own or may be wrapped by accessories. That's way the proper design is the decorator. Every concrete suit class extends from suit abstract class and every concrete accessories class extends from the accessories abstract class and all of them hold a reference to suit so that when user wants to decorate his/her suit with other things, the cost and the weight can be calculated by delegating instead of inheritance.

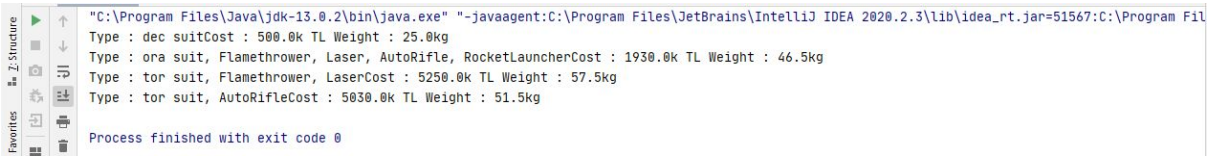

figure3.2 : example result of the program

I wrote driver class to test program. First one only buys one suit without any decoration. Second one buys all the decorators with his/her suit. Third one buys only two of them and the last one buys only one decorator. Thus, price and the weight of the all cases are calculated correctly using decorator design pattern.