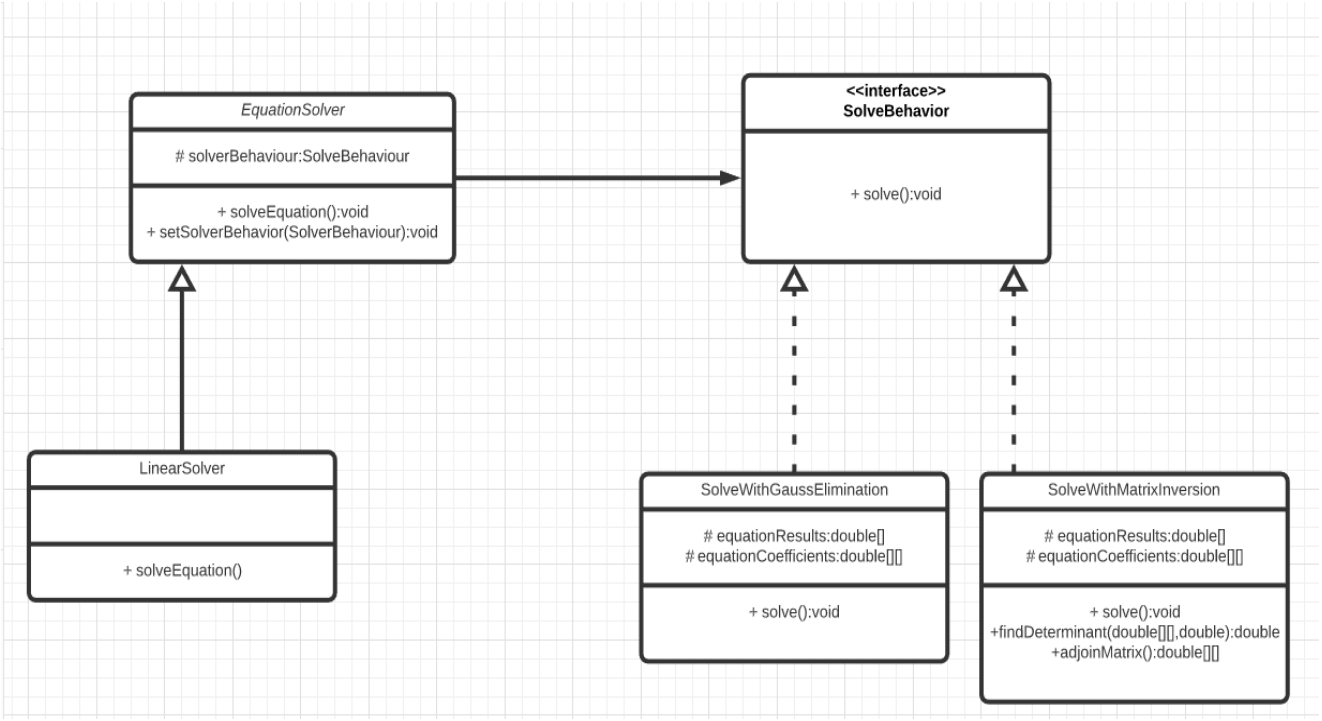
# CSE443 - Homework #1

## Part 1:

In part1, There can be more than 1 linear solving method and they can change dynamically. So my design should support that the behaviour or algorithm of a class can be change during runtime. For example I should solve a linear equation by Gauss elimination then if I want I  should be able to switch from Gauss to Matrix inverse method or another linear solving method. Also for maintanence my design should be maintanable. Because of that I use strategy design patter fort his part.

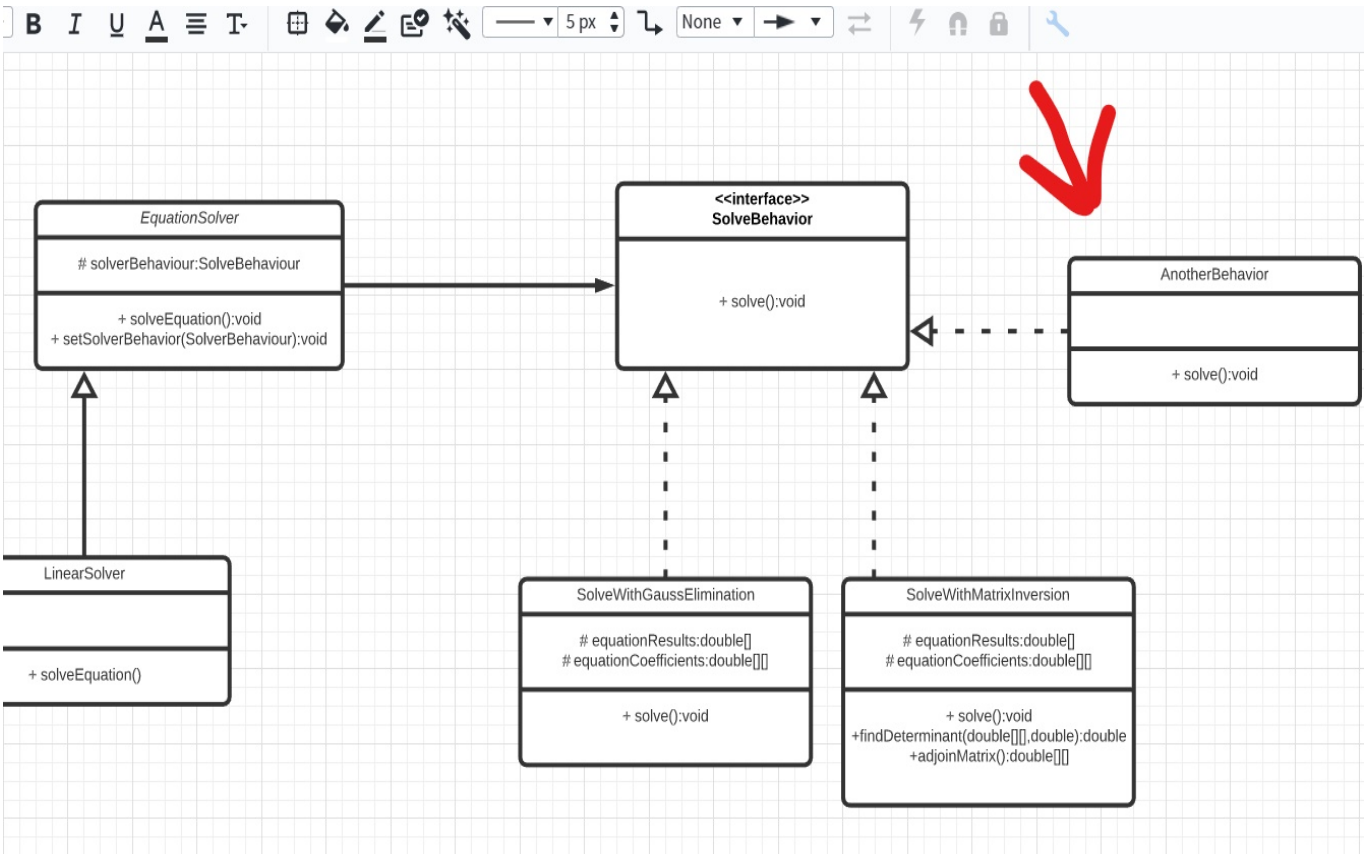



*Şekil 1(Part-1 UML Diagram)*

As you can see in the uml diagram I seperate changable behaviours and encapsulate them. Here EquationSolver class just has a data field which is solveBehavior type. LinearSolver use this behaviour because of inheritance. Now according to solveBehaviour is a SolveWithGaussElimination or SolveWithMatrixInversion it will call these classes corresponding solve methods when solveEquation method called. Also with this design I can change my behaviours dynamically with using setSolverBehaviour method. For example if my solveBehaviour is a SolveWithGaussElimination, solveEquation method will call SolveWithGaussElimination's solve method and it will solve equation by using Gauss Elimination, then I can set my beaviour to SolveWithMatrixInversion dynamically, after that when I call solveEquation method it will call SolveWithMatrixInversion's solve method and will solve equation by using Matrix Inversion technic. Additionally, My behaviours act like a closed box for LinearSolver class. My LinearSolver class just call solveEquation whether or not know its SolveWithGaussElimination or SovleWithMatrixInversion. So whenever I change behavior classes they will not effect my LinearSolver. This will increace maintanable.

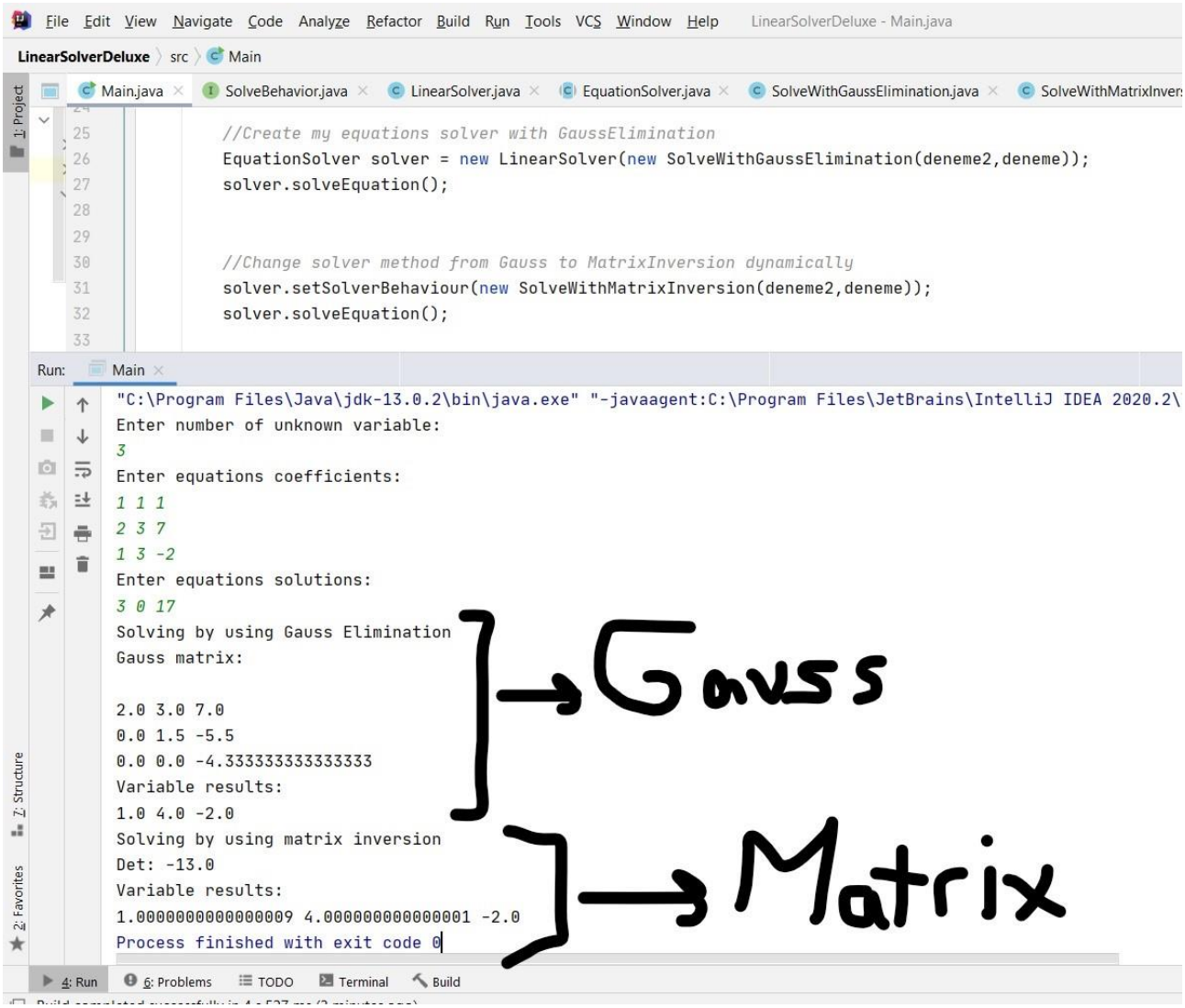
By the way if I want I can inherite another solver equation from EquationSolver like non-linear Solver and expand my program.

Also this design will not just work for 2 behaviour. You can add as much as linear equation solver method you want. This is really good for maintainable. For example, after some time customers may want to add another linear system solver method this program.




Like this, you can add another behaviour. This will not affect our program and we dont have to change anything because we seperate changable class to others and encapsulate them. Just implemet new behaviour solve method and use set method to change solver methods whenever you want and choose one of them. So with this design we can expand our program easly without modifying existing classes. Beucase of that flexibity and maintanability of this design very good.

Demo:

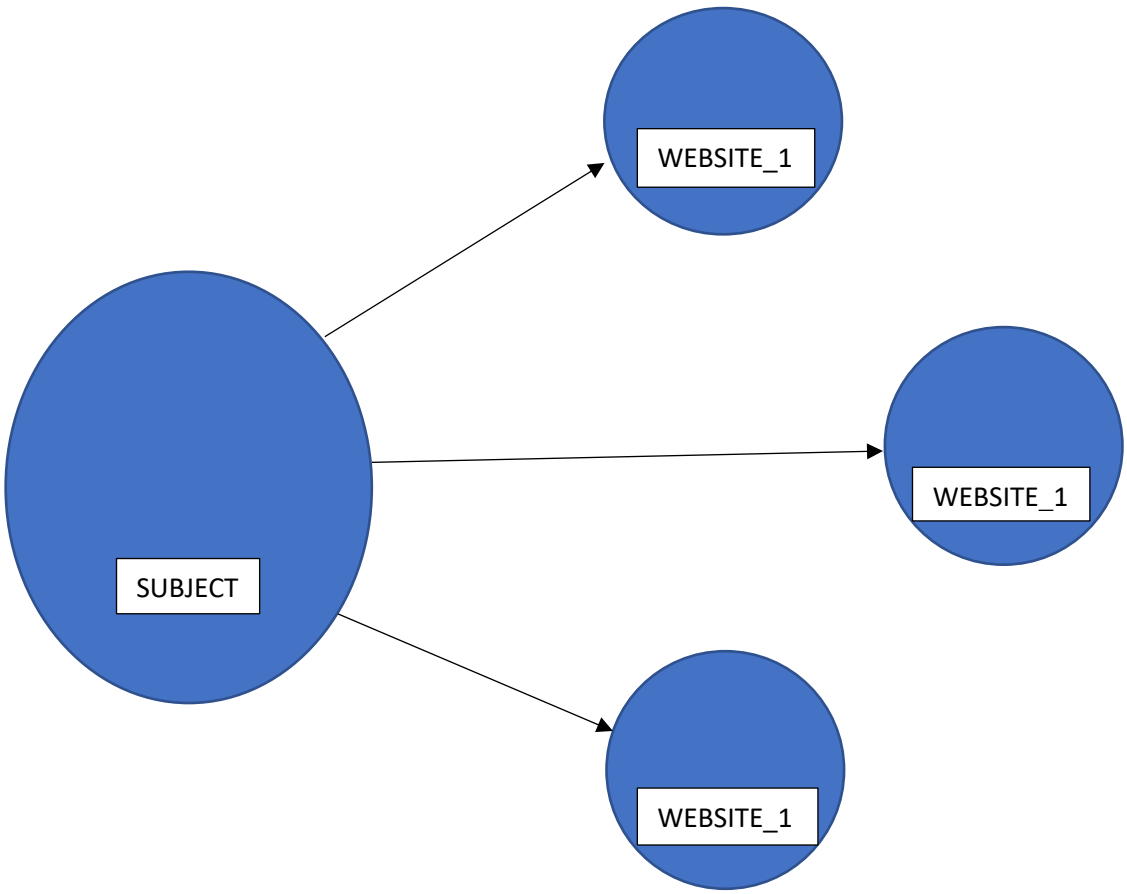



Here firsty, I enter my unknown variable count then equations coefficients and finally equations solutions. Then in code I create a solver with SolveWithGaussElimination behavior. Then call solveEquation method. This method call gauss Elimination solver as you can see in output and it will find variables by using Gauss. Then I call setSolverBehaviour method and change my behavior to SolveWithMatrixInversion in runtime(dynamically). After I call

solveEquation method again and as you can see it solve equation by using matrix Inversion method and found same variables results correcty.

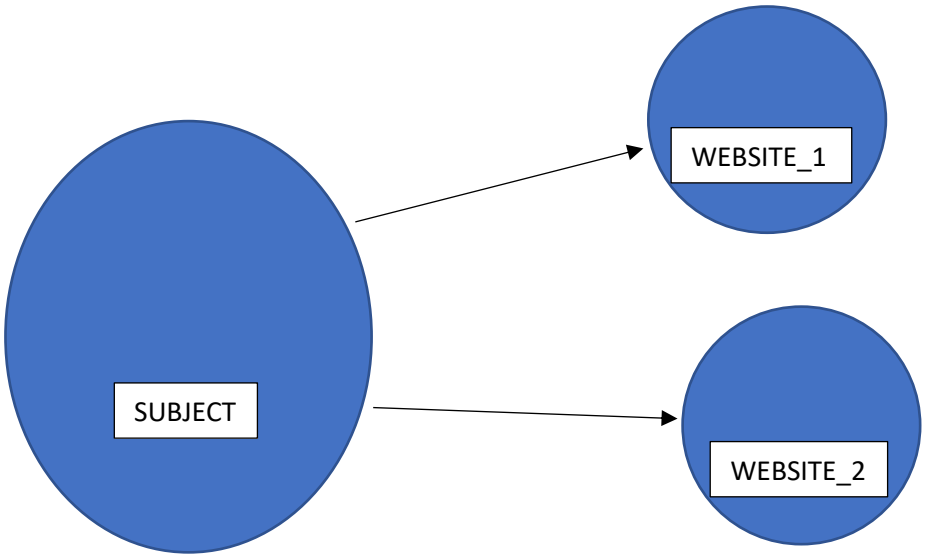
## Part 2:

In this part we have a relationship like Publisher and subscriber between our favorite websites and observer. Hence, we can use Observer pattern for this part. The observer pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically. With this design we will have a observer and observable class  which is our websites. We can bind our websited to observer or we can remove them easily in this design without change other clsses. For example:

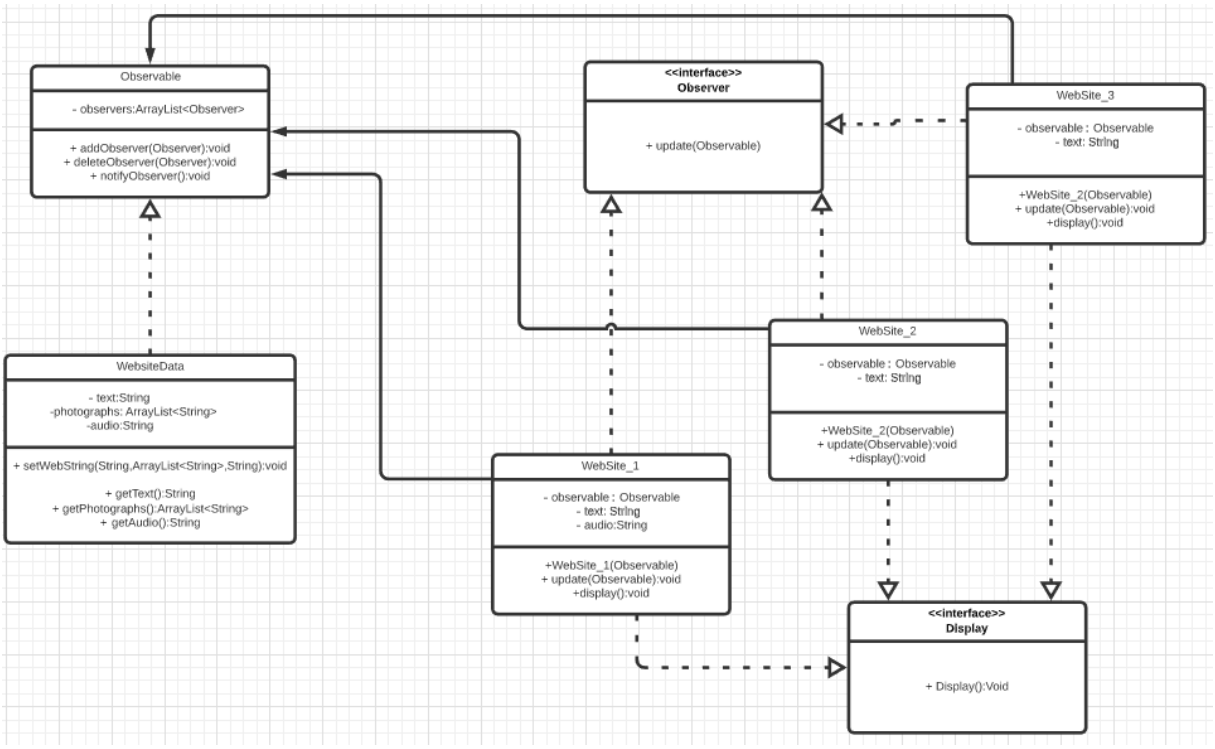


Like this, our design has a one to many realation between subject and websites. Subject can notify his subscribers(websites) whenever something change or subscribers can ask subject to pull the new changes. So there is 2 method in observer pattern for taking changes. One of them push and other method is pull. With push method subject can send all data to the subsribers whenever something changed. On the other hand subscribers can pull the up the date datas from the subject. With pull method subscribers can pull only datas that need.

Also, in this design we can add or remove websites in runtime without change any other classes.




Here we remove our favorite website_3 in design easily and website_1 and website_2 doesn't affected. They don't even know what is going on. So this increase our software flexibility and maintanability. Whenever we want we can add or remove a favorite website to our program easily.

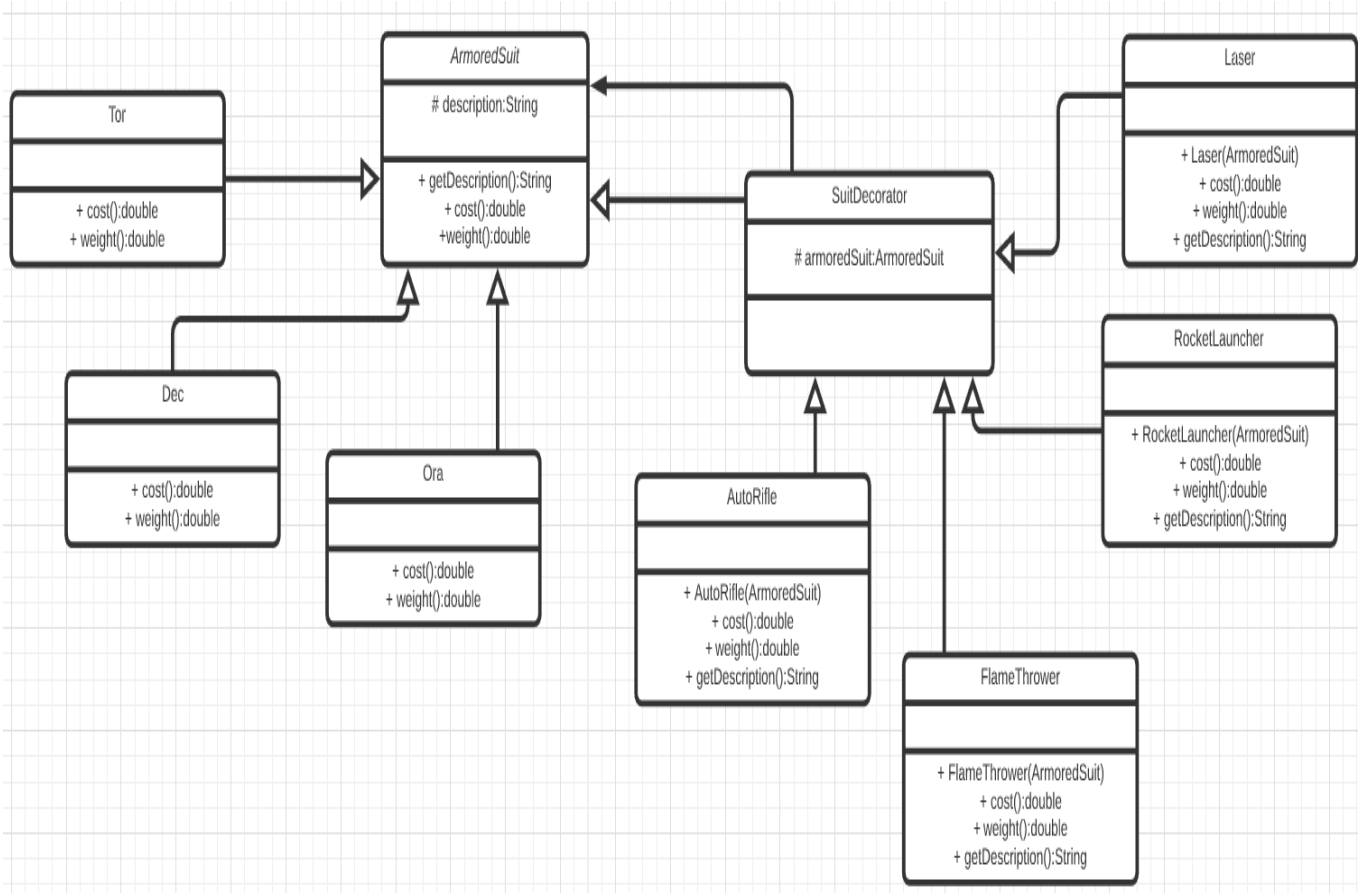



This is uml diagram for my desing in part2. As you can see there is an observable-observer pattern. In Observable class there is a ArrayList which holds observers. So I can add or remove my favorite websites to this array. For example, right now there are 3 website in my design and in runtime I can add or remove these websites to my ArrayList and with notifyObserver methods whenever something has changed I can notify and update them. Also as you can see some websites has only text,audio or photograpgs or combination thereof. Because of I use pull method in this design they only pull data fields which interested. Such as website_1 only pull text and audio not photograpgs in update method because it doesn't has photograpgs in interest. This is very flexible and maintanable design you can create more websites or remove them without changing anything.

Also you can add new type of content this program. Just add it to WenSiteData and declare a getter method for him and you can use this content for your desired websites. It is just that easy to modify.

# Part 3:

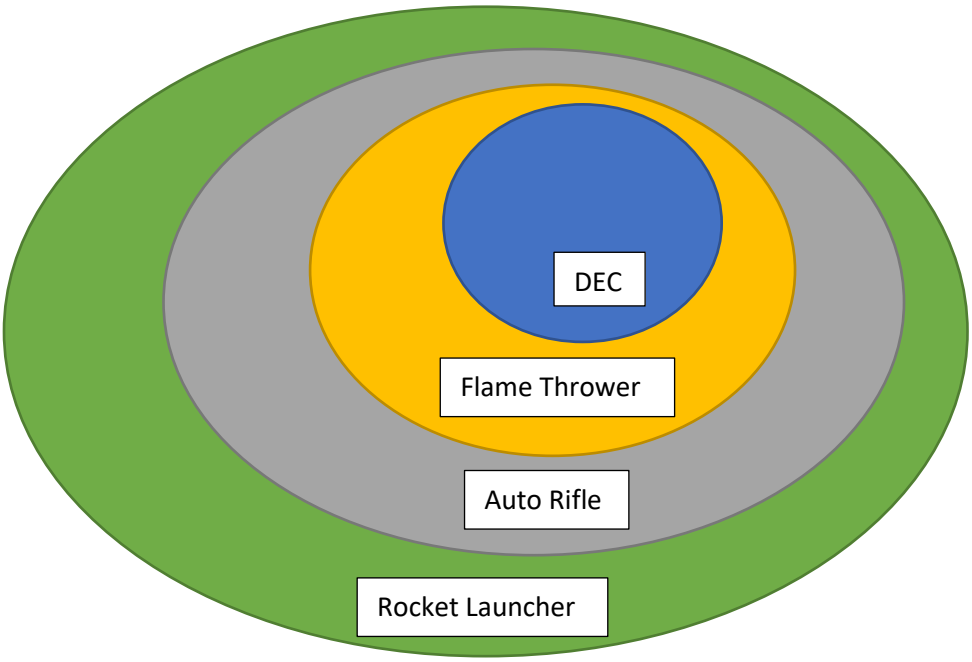
Fort his part, we choose a suit(Dec,Ora or Tor) then we will add this suit Flamethrower, Autorifle, Rocket launcher and laser as much as we want. This is like coffie example as we see in class. Like coffie example we can handle this program by using Decorater design pattern. With decorater design patter we can create our suit then we can Wrap it our decoraters as much as we want. After that our cost and weight method can calculate total cost and weigt of this wrapped object.




*Şekil 2Part3-Uml*

This is my Uml diagram for part3. Here as you can see I have ArmoredSuit abstract class and my 3 main suit (Tor,Dec,Ora) is inhereting it. Also there is SuitDecorator class which is inheret

ArmoredSuid class. There are 4 decorator (AutoRifle,FlameThrower,RocketLauncher and Laser). The imported part here is There is has-a and is-a relation between my SuitDecorator and ArmoredSuit classes. Because of that I can wraped decoraters and create new objects. For example, a dec with 1 flamethrower, 1 automatic rifles and 1 rocket launcher object will look like this:

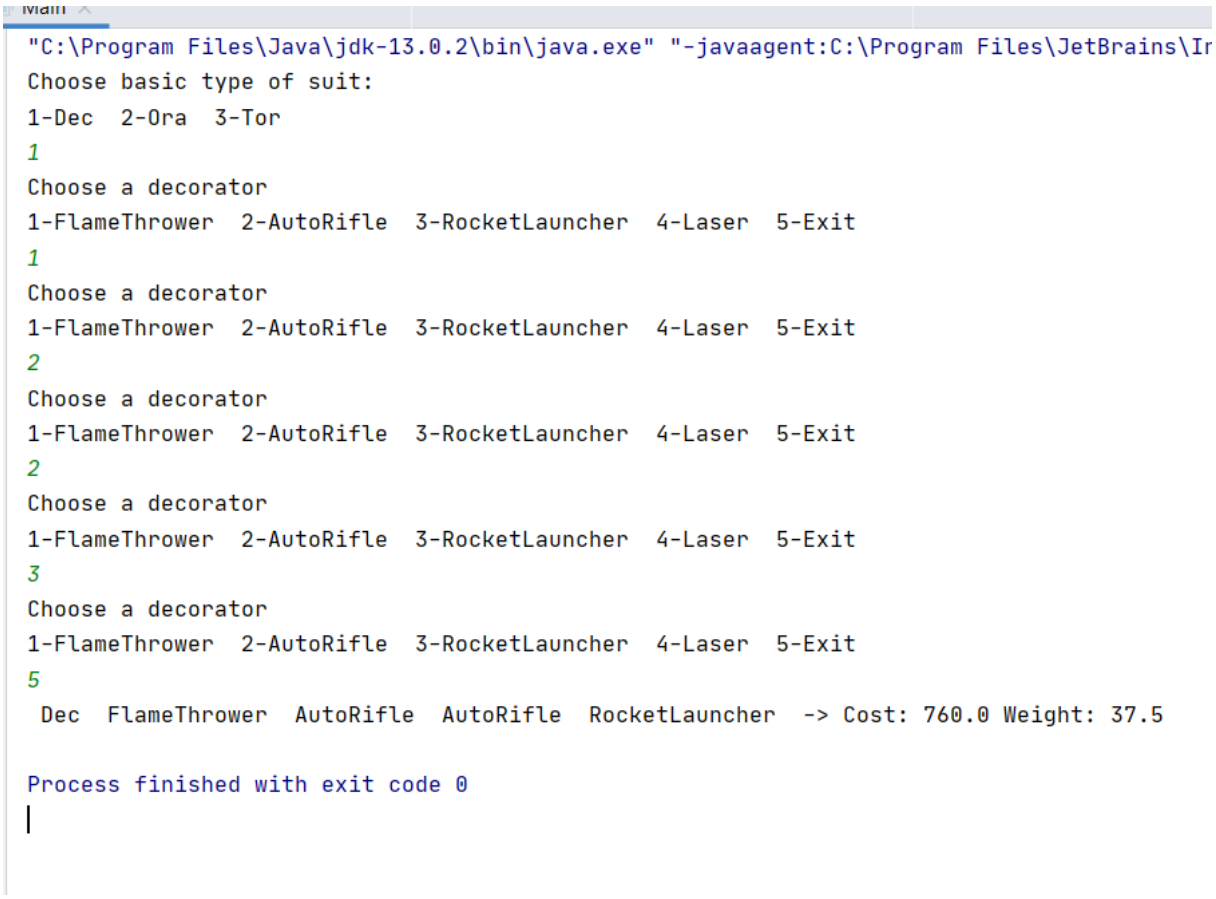



And then when we try to calculate cost or weight, Just like recursive function we traverse cost methods and calculate DEC.cost + flameThrower.cost+ AutoRifle.cost+ RocketLauncher.cost and find our total suit cost and weight.

Also with Decorater pattern our code will be dynamic which mean we can add as much as decorator in runtime. The user of the software be

able to designate any combination of accessories dynamically at runtime. Lets see an example of that in my code.

Demo:



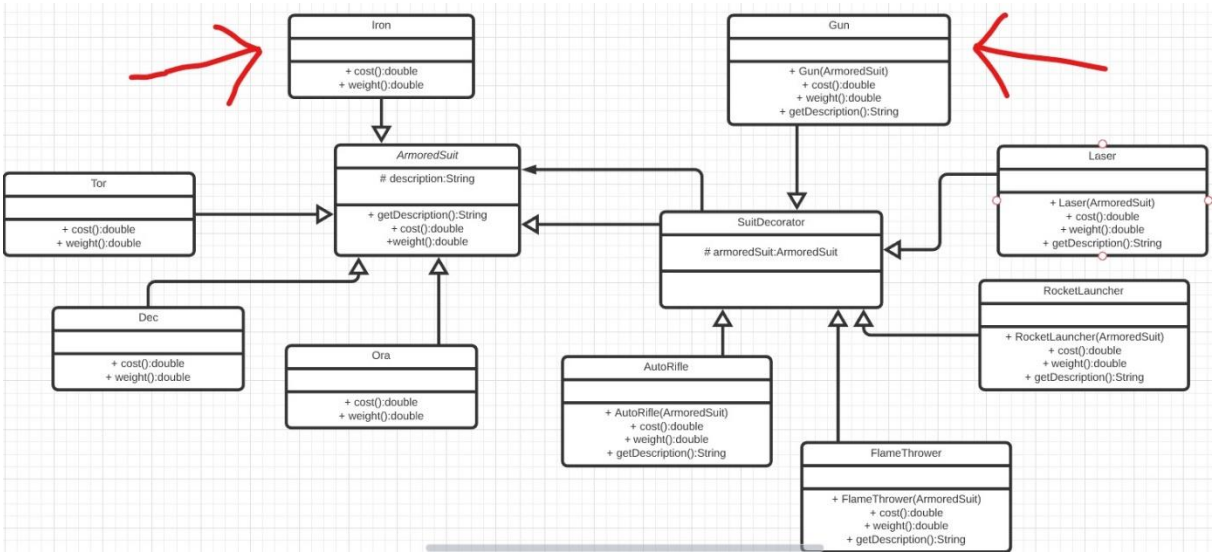

I started my program and firstly choose my basic type of suit. Then I can add as much as decorater in my code dynamically. Such as I choose Dec for basic suit then add 1 FlameThrower, 2 AutoRiffle and 1 Laser then I enter 5 and exit. After that, my code print my suit description, total cost and weight,

Here:

Dec.cost + FlameThrower.cost + AutoRifle.cost + AutoRifle.cost+ RocketLauncher.cost = 760.0

Dec.weight + FlameThrower. weight + AutoRifle. weight + AutoRifle. weight + RocketLauncher. weight = 37.5

This deisgn also very flexible and easy to maintanable.




We can add another decorator for suit or we can add a basic type of suit without change an existing class. For instance I add a new decorater for suits called Gun and a new basic type of suit called Iron without changing other classes. Even other classes dont aware that new decorater and a new suit added to our design. They will continue to work the way they worked before. So we can expand our software without modifying existing classes and this is very good for flexibility and maintability.