CSE443 – Object Oriented Analysis and Desing

REPORT – HW1

Furkan ÖZEV

161044036

PART 1:

In the Linear Solver Deluxe project, a linear equation system is taken from the user and it is expected that this equation can be solved with at least 2 different methods.

The customer initially requires 2 methods. These methods are Gaussian elimination and Matrix inversion. The number of methods may increase in the future.

In addition, the customer requests the solution method to change dynamically at the runtime of the program.

And project might need more functionalities in the future.

The program should satisfies the customer's requirements by maximum flexibility, loose coupling and minimizing maintenance costs.

Thus, I designed the Linear Solver Deluxe project using the **Strategy Design Pattern**.

The Strategy Design Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

This design pattern enables the object that will use this method to choose which method to apply in cases where different methods are applicable for an operation.

The classes of solution methods have implemented a common interface. Thus, these methods can be expressed with a single common type. (LinearSolverMethods)

LinearEqSolver class, which will solve the Linear Equation System, keeps this method as an object in LinearSolverMethods type.

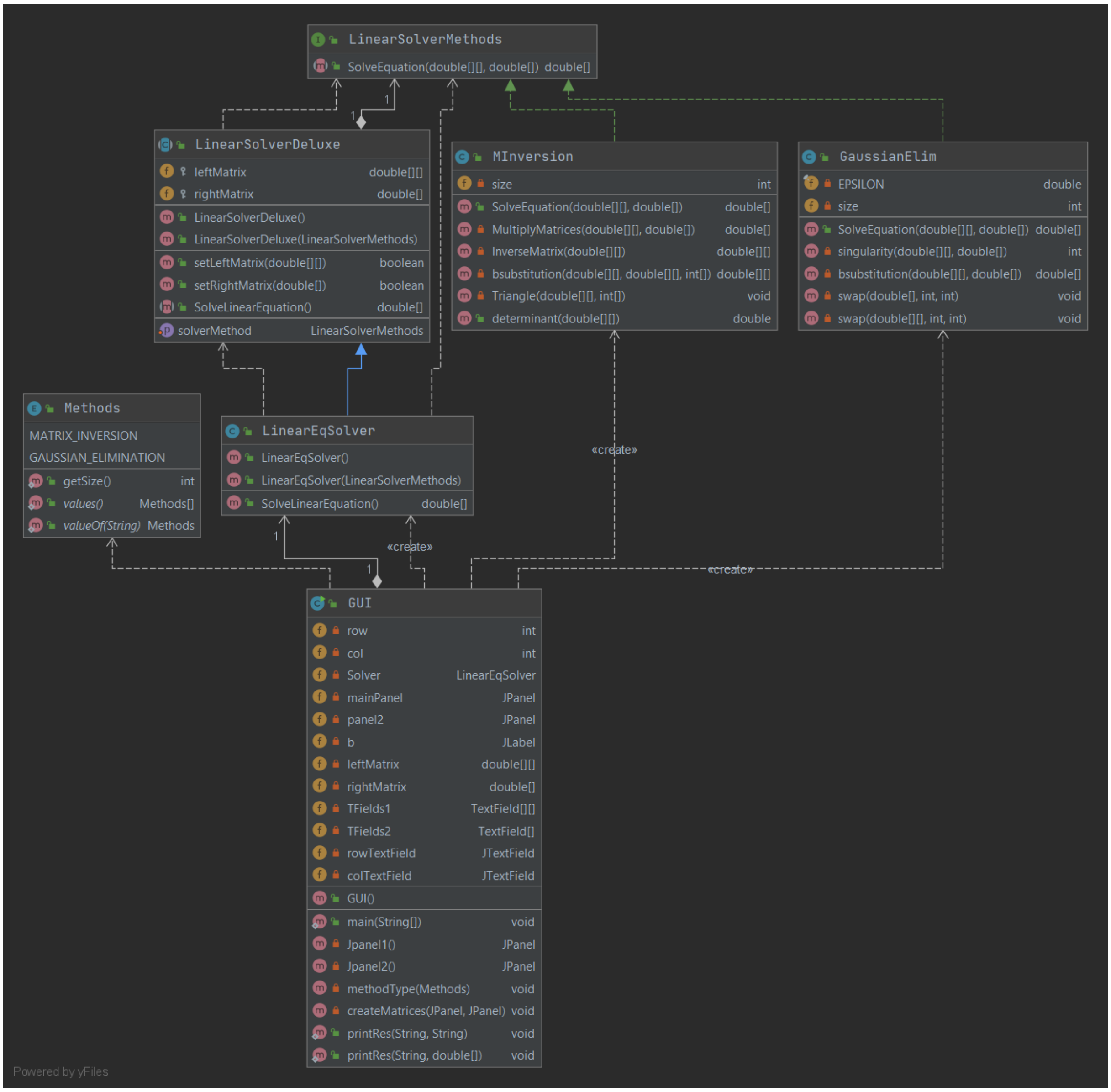
Thus, there is a HAS-A relationship between the LinearEqSolver class and the LinearSolverMethods classes.

In this way, the method can be changed dynamically. When the Solve function is called, the solve function of this method object is executed.

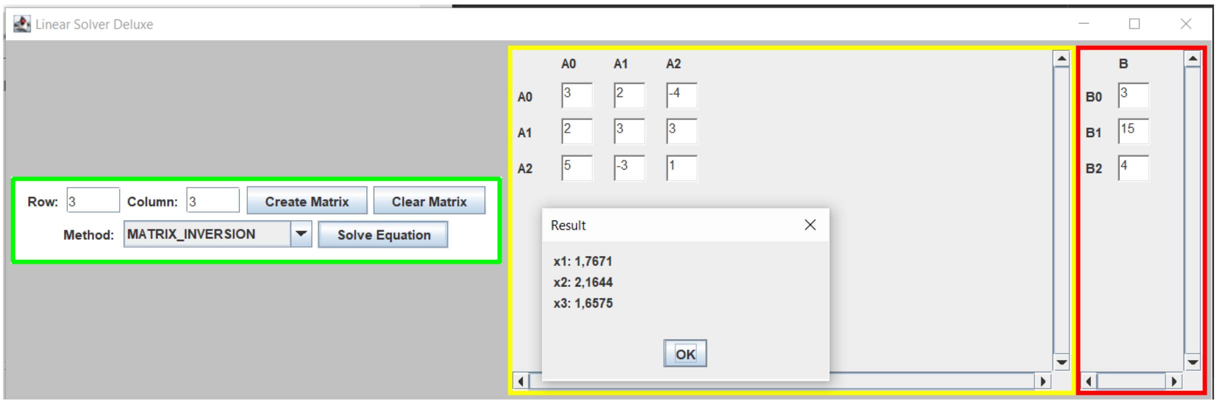
When a new method wants to be added, the things to do are simply as follows:

1- The class of the new method must implement the "LinearSolverMethods" interface. Then, It should override the SolveEquation () function of this interface.
2- The name of the new method must be added to the "Methods" enum class.
3- The enum name of the new method must be added for the JBox section where we select the method in the GUI.

Class Diagram:


LinearSolverMethods
SolveEquation(double[][], double[]) double[]
LinearSolverDeluxe
leftMatrix double[][]
rightMatrix double[]
LinearSolverDeluxe()
LinearSolverDeluxe(LinearSolverMethods)
setLeftMatrix(double[][]) boolean
setRightMatrix(double[]) boolean
SolveLinearEquation() double[]
solverMethod LinearSolverMethods
MInversion
size int
SolveEquation(double[][], double[]) double[]
MultiplyMatrices(double[][], double[]) double[]
InverseMatrix(double[][]) double[][]
bsubstitution(double[][], double[][], int[]) double[][]
Triangle(double[][], int[]) void
determinant(double[][]) double
GaussianElim
EPSILON double
size int
SolveEquation(double[][], double[]) double[]
singularity(double[][], double[]) int
bsubstitution(double[][], double[]) double[]
swap(double[], int, int) void
swap(double[][], int, int) void
Methods
MATRIX_INVERSION
GAUSSIAN_ELIMINATION
getSize() int
values() Methods[]
valueOf(String) Methods
LinearEqSolver
LinearEqSolver()
LinearEqSolver(LinearSolverMethods)
SolveLinearEquation() double[]
«create»
«create»
«create»
1
1
1
1
GUI
row int
col int
Solver LinearEqSolver
mainPanel JPanel
panel2 JPanel
b JLabel
leftMatrix double[][]
rightMatrix double[]
TFields1 TextField[][]
TFields2 TextField[]
rowTextField JTextField
colTextField JTextField
GUI()
main(String[]) void
Jpanel1() JPanel
Jpanel2() JPanel
methodType(Methods) void
createMatrices(JPanel, JPanel) void
printRes(String, String) void
printRes(String, double[]) void
Powered by yFiles

Output:

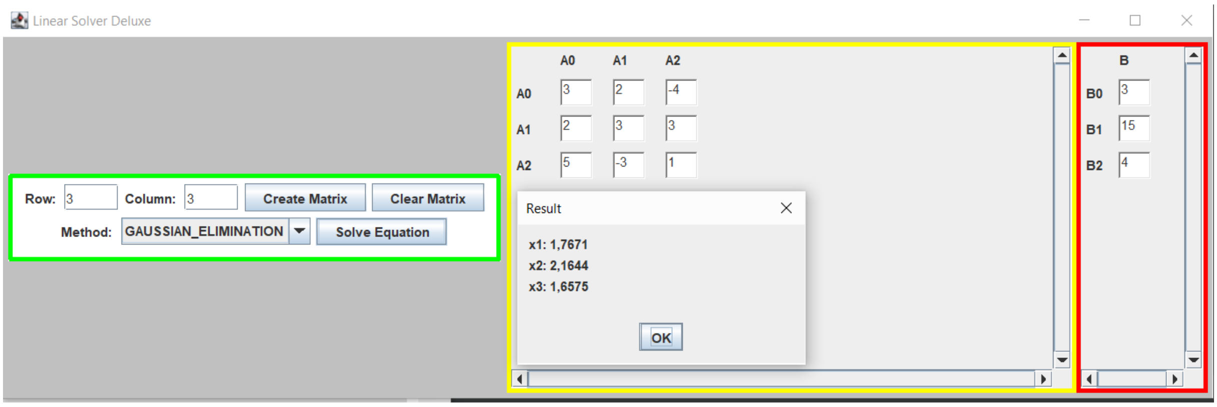


Row: 3 Column: 3

Method: MATRIX_INVERSION

	A0	A1	A2	B
A0 / B0	3	2	-4	3
A1 / B1	2	3	3	15
A2 / B2	5	-3	1	4

```
x1: 1,7671
x2: 2,1644
x3: 1,6575
```

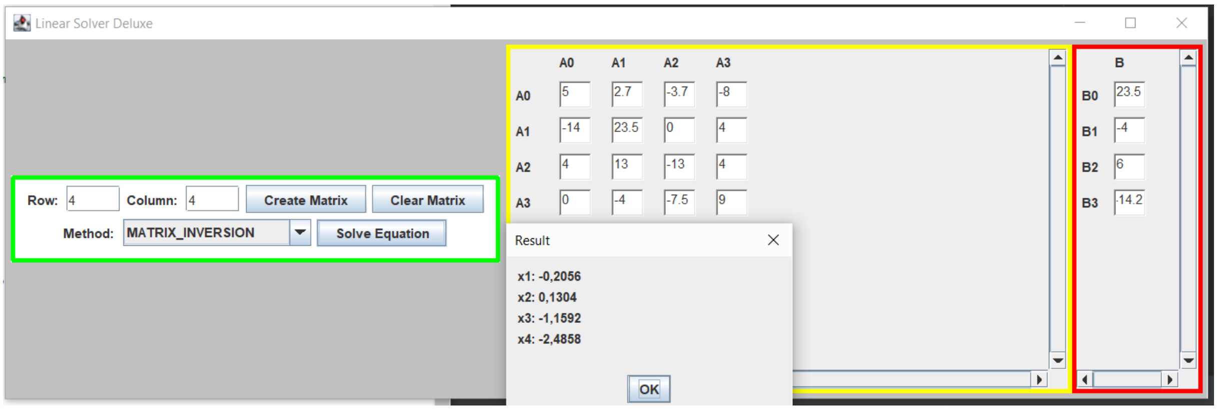


Row: 3 Column: 3

Method: GAUSSIAN_ELIMINATION

	A0	A1	A2	B
A0 / B0	3	2	-4	3
A1 / B1	2	3	3	15
A2 / B2	5	-3	1	4

```
x1: 1,7671
x2: 2,1644
x3: 1,6575
```

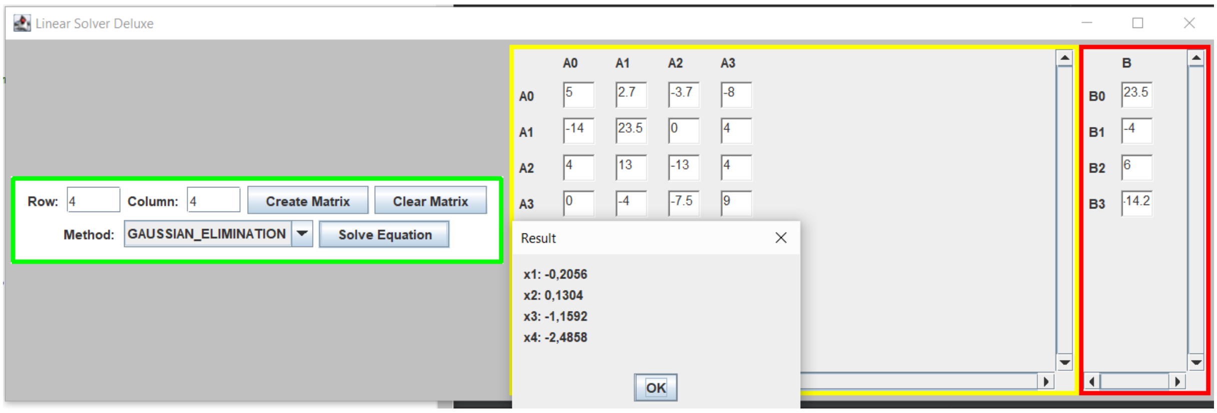


Row: 4 Column: 4

Method: MATRIX_INVERSION

	A0	A1	A2	A3	B
A0 / B0	5	2.7	-3.7	-8	23.5
A1 / B1	-14	23.5	0	4	-4
A2 / B2	4	13	-13	4	6
A3 / B3	0	-4	-7.5	9	14.2

```
x1: -0,2056
x2: 0,1304
x3: -1,1592
x4: -2,4858
```

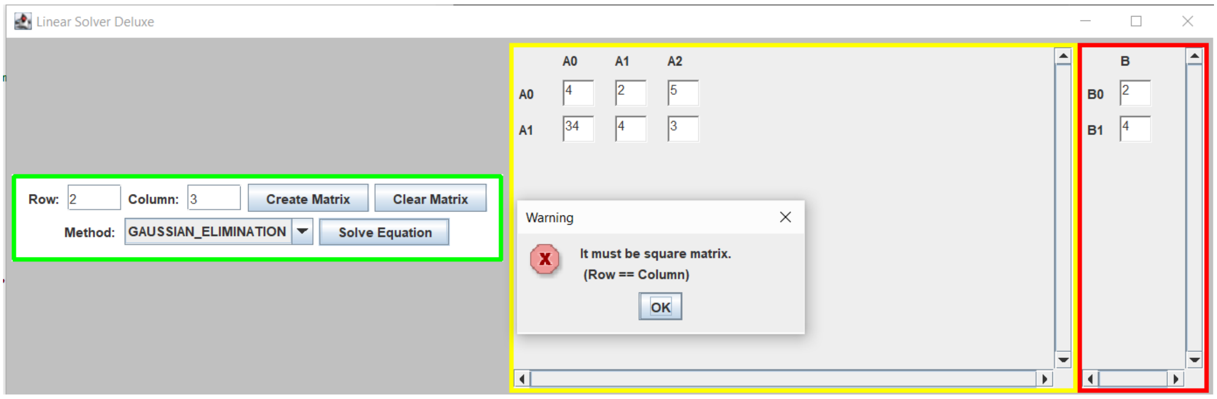


Row: 4 Column: 4

Method: GAUSSIAN_ELIMINATION

	A0	A1	A2	A3	B
A0 / B0	5	2.7	-3.7	-8	23.5
A1 / B1	-14	23.5	0	4	-4
A2 / B2	4	13	-13	4	6
A3 / B3	0	-4	-7.5	9	14.2

```
x1: -0,2056
x2: 0,1304
x3: -1,1592
x4: -2,4858
```



Row: 2 Column: 3

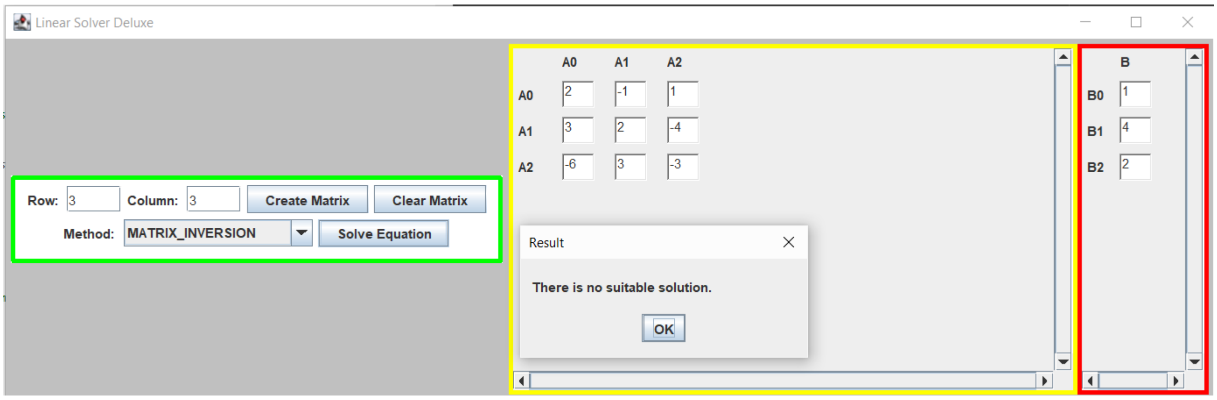
Method: GAUSSIAN_ELIMINATION

	A0	A1	A2
A0	4	2	5
A1	34	4	3

	B
B0	2
B1	4

Warning

It must be square matrix.
(Row == Column)



Row: 3 Column: 3

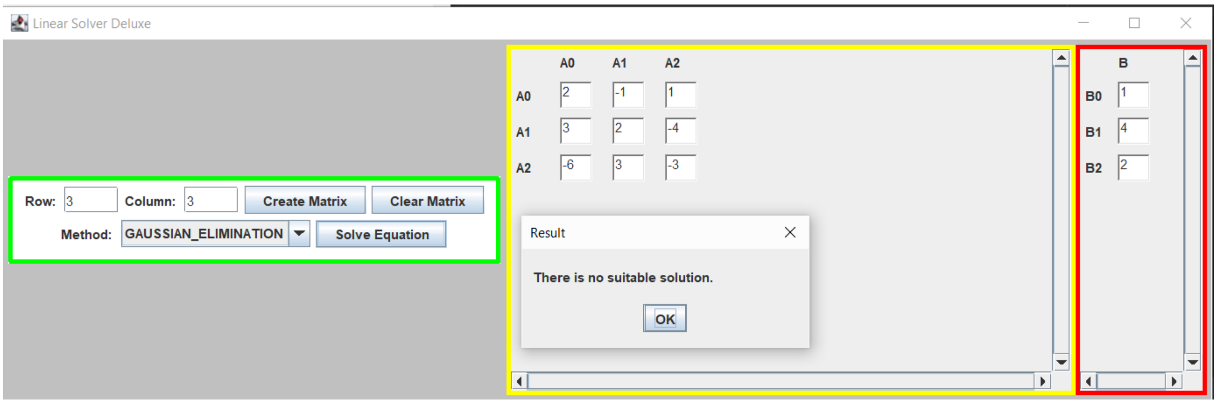
Method: MATRIX_INVERSION

	A0	A1	A2
A0	2	-1	1
A1	3	2	-4
A2	-6	3	-3

	B
B0	1
B1	4
B2	2

Result

There is no suitable solution.



Row: 3 Column: 3

Method: GAUSSIAN_ELIMINATION

	A0	A1	A2
A0	2	-1	1
A1	3	2	-4
A2	-6	3	-3

	B
B0	1
B1	4
B2	2

Result

There is no suitable solution.

PART 2:

The WebAPI project has been developed in order to enable Users to subscribe to a website and to follow content updates on this website.

The contents that the user can subscribe are as follows; photo, text, audio and any combination of these.

There is no need to disturb them if the update is not of the desired type.In addition, the customer requests the solution method to change dynamically at the runtime of the program.

New content may come in or the subscriber can follow a new content. So, project might need more contents in the future.

The program should satisfies the customer's requirements by maximum flexibility, loose coupling and minimizing maintenance costs.

Thus, I designed the L WebAPI project using the **Observer Design Pattern**.

The Observer Design 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.

This design pattern allows the user to subscribe to the content they want on a website they want, terminate their subscription, and notify the subscriber when the content changes.

WebSubjectInterface -> This interface is the Subject interface in the observer design pattern. It includes functions such as add(), delete(), notify().

WebSubject -> It is an abstract class. It implements WebSubjectInterface. It keeps the subscriber list as an array list. Different types of content should extend this class. Subscribers will use this declared type when subscribing to Contents.

WebPhotoSubject, WebTextSubject, WebAudioSubject -> They are content classes. They extend WebSubject class. These are our real type (Object can be created) subjects. They contain their own customized fields (like photo, text audio) and functions. Subscribers use the objects of these content classes and their functions extended from the upper class while performing subscription operations.

Subscriber -> This interface is the Observer interface in the observer desing pattern. It includes uptade() function.

WebSubscriber -> This class is the observer class. Unlike normal, it keeps a subject list. In this way, more than one content can be subscribed. Desired subscriptions can be made in the form of photo, text, audio and combinations. It stores the content it will follow as a field. It includes 2 different methods for the subscriber to unsubscribe a content (unSubscribeContent()) or subscribe to a different content (subscribeNewContent) at any time.

So each content is a subject, and an observer can subscribe to more than one subject.

When a new content wants to be added, the things to do are simply as follows:

1- The class of the new content must extend the "WebSubject" abstract class. Then, It should implement own functions.

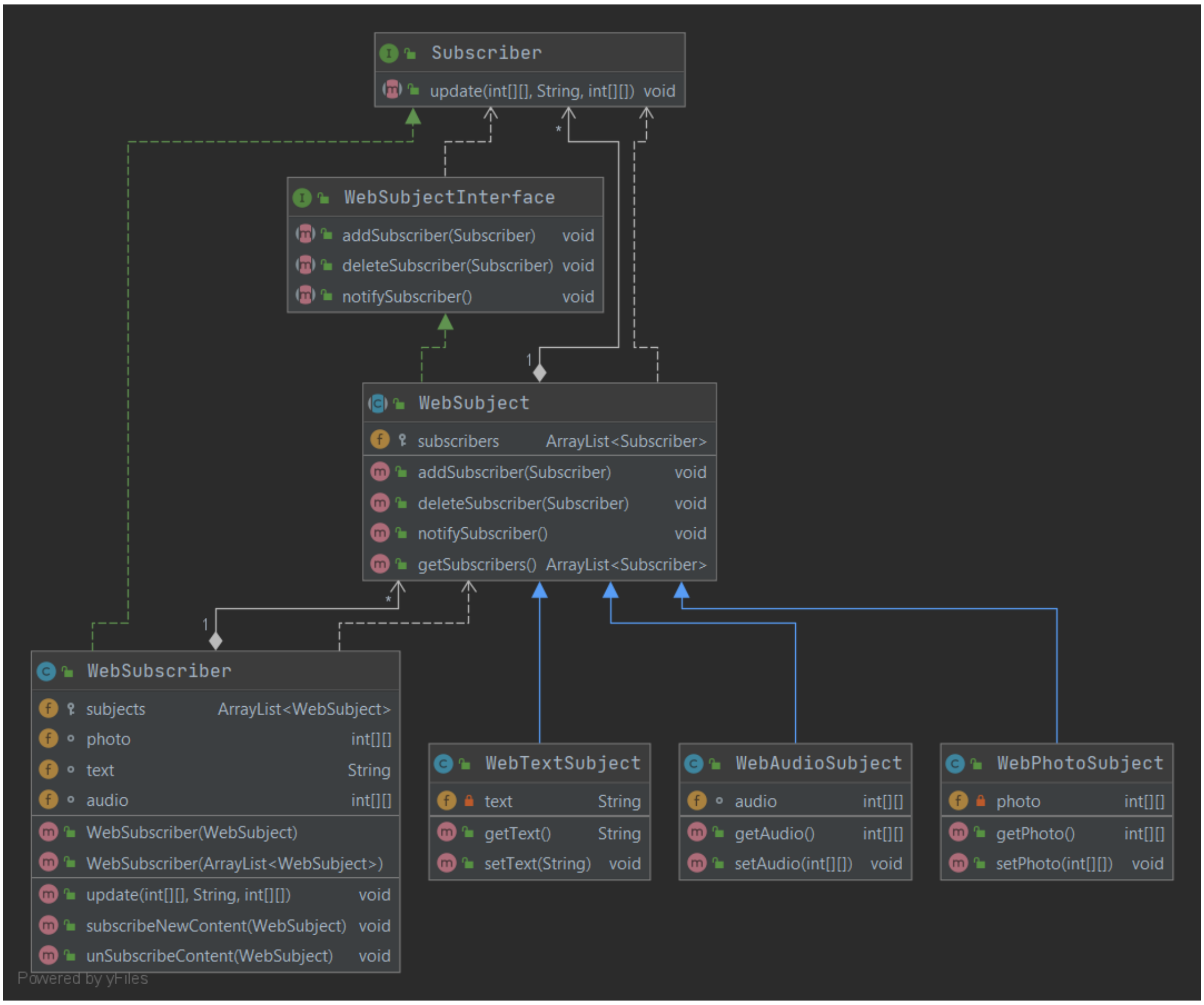
When a subscriber wants to subscribe new content, the things to do are simply as follows:

1- No change in code needed. Just call subscribeNewContent(WebSubject)

When a subscriber wants to unsubscribe new content, the things to do are simply as follows:

1- No change in code needed. Just call unSubscribeContent(WebSubject)

Class Diagram:


Subscriber
update(int[][], String, int[][]) void
WebSubjectInterface
addSubscriber(Subscriber) void
deleteSubscriber(Subscriber) void
notifySubscriber() void
WebSubject
subscribers ArrayList<Subscriber>
addSubscriber(Subscriber) void
deleteSubscriber(Subscriber) void
notifySubscriber() void
getSubscribers() ArrayList<Subscriber>
WebSubscriber
subjects ArrayList<WebSubject>
photo int[][]
text String
audio int[][]
WebSubscriber(WebSubject)
WebSubscriber(ArrayList<WebSubject>)
update(int[][], String, int[][]) void
subscribeNewContent(WebSubject) void
unSubscribeContent(WebSubject) void
WebTextSubject
text String
getText() String
setText(String) void
WebAudioSubject
audio int[][]
getAudio() int[][]
setAudio(int[][]) void
WebPhotoSubject
photo int[][]
getPhoto() int[][]
setPhoto(int[][]) void
*
1
1
*
Powered by yFiles

PART 3:

The Suit project has been developed calculate cost and weight of exoskeleton armored suits for military personnel, equipped with various custom weapons.

There are 3 basic types of suits. And each of them has different weight and cost.

Each of these suits can be equipped with differrent accessories. These are Flamethrower, AutoRifle, RocketLauncher, Laser. And each of them has different weight and cost.

Basic suite can accommodate more than one accessory. It should be supplied any custom combination of accessories.

The suit project calculates the total cost and weight of an equipped suit.

New basic suit or accessory may come. So, project might need more suits or accessories in the future.

The program should satisfies the customer's requirements by maximum flexibility, loose coupling and minimizing maintenance costs. And easy to accommodate new accessories and suit types.

Thus, I designed the L WebAPI project using the **Decorator Design Pattern**.

The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

With Decorator Design Pattern, accessories are added dynamically.

Suits -> This abstract class is the Component class in the decorator design pattern. It includes functions such as cost(), weight(). All components extend this class.

dec, ora, tor -> These classes extend Suits class. They are ConcreteComponent class in the decorator design pattern. These refer to basic suites. Each has a different cost and weight.

Accessories -> This class extend Suits class. This abstract class is the Decorator class in the decorator design pattern. All accessories are derived from this class.

Laser, AutoRifle, RocketLauncher, Flamethrower -> These claesses extend Accessories class. They are ConcreteDecorators in decorator design pattern. These refer to accessories. Each has a different cost and weight. These class have a suit field. Thus, combine 1 basic suit and many accessories.

ReadyDemo -> It is a ready demo with ready values.

ManualDemo -> It is a manual demo for user. There is an menu. User can enter own selections.

So any combination of accessories can be determined dynamically at runtime.

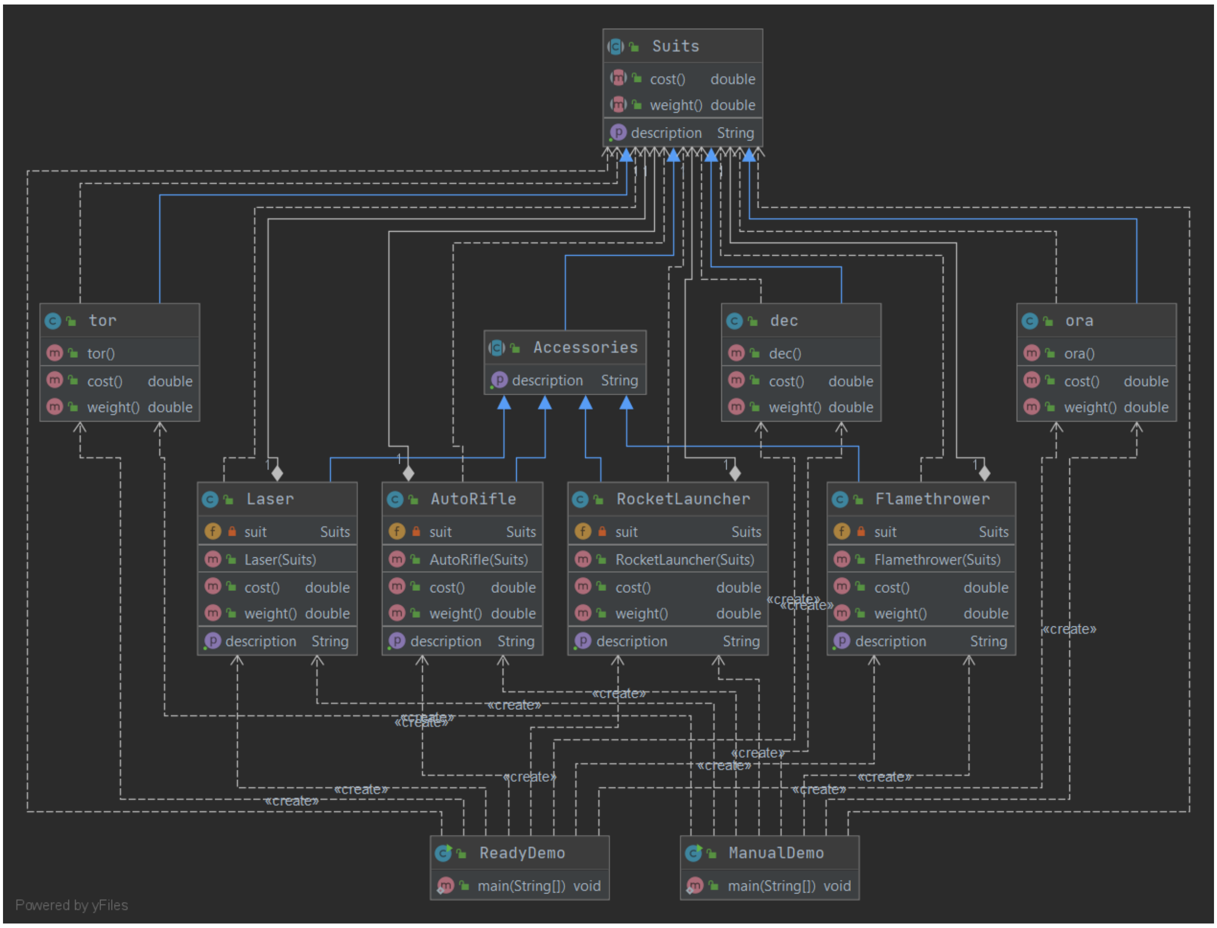
When a new basic suit wants to be added, the things to do are simply as follows:

1- The class of the new basic suit must implements the "Suits" interface. Then, It should implements cost() and weight() functions.

When a subscriber wants to be added new accesory, the things to do are simply as follows:

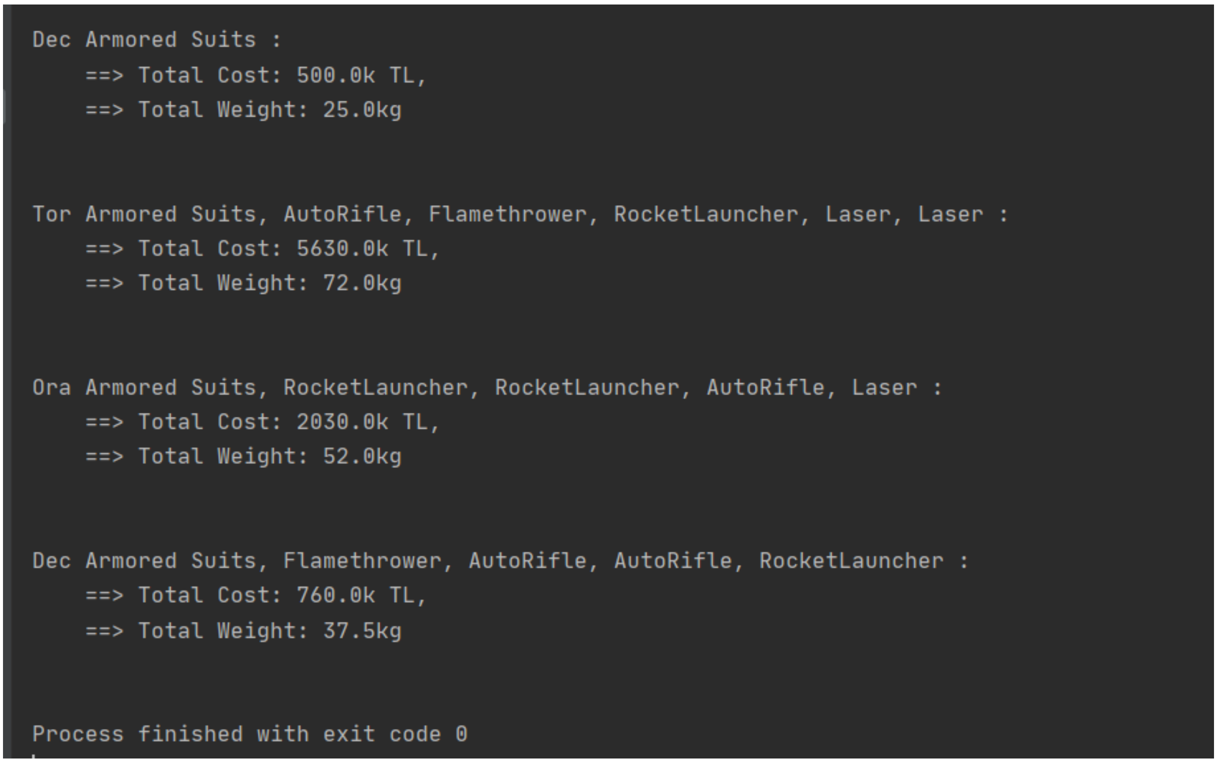
1- The class of the new accessory suit must extends the “Acessories” abstract class. Then, It should implements cost() and weight() functions. And It must have “Suits” field.

Class Diagram:



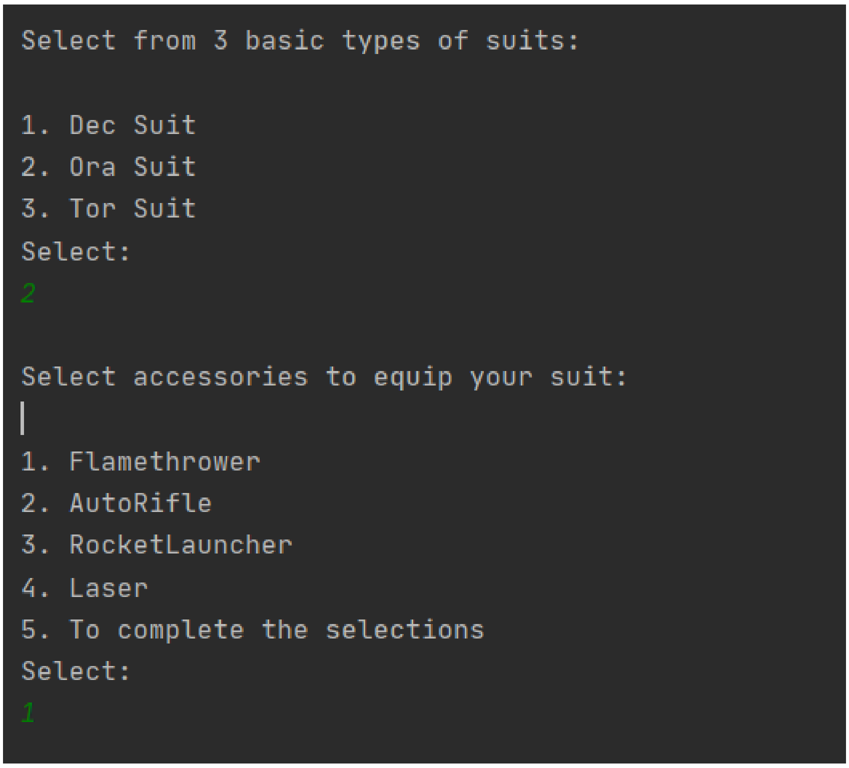
Output:

READY DEMO

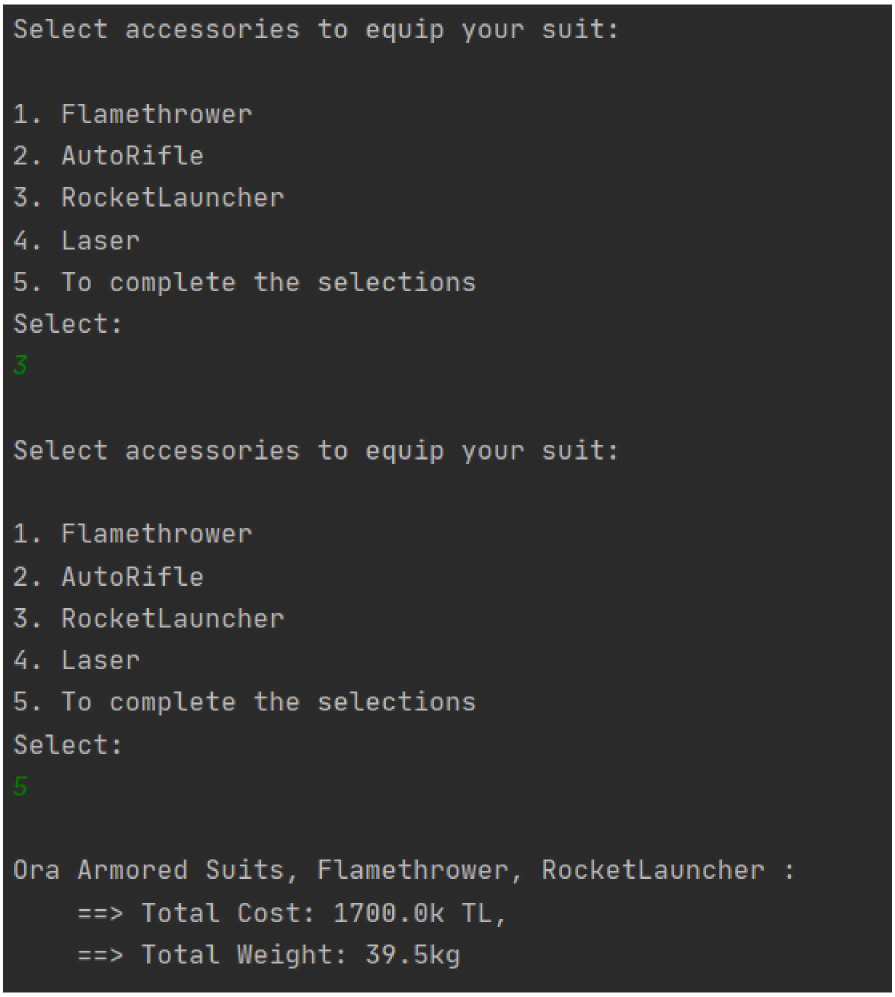


```
Dec Armored Suits :
    ==> Total Cost: 500.0k TL,
    ==> Total Weight: 25.0kg


Tor Armored Suits, AutoRifle, Flamethrower, RocketLauncher, Laser, Laser :
    ==> Total Cost: 5630.0k TL,
    ==> Total Weight: 72.0kg


Ora Armored Suits, RocketLauncher, RocketLauncher, AutoRifle, Laser :
    ==> Total Cost: 2030.0k TL,
    ==> Total Weight: 52.0kg


Dec Armored Suits, Flamethrower, AutoRifle, AutoRifle, RocketLauncher :
    ==> Total Cost: 760.0k TL,
    ==> Total Weight: 37.5kg


Process finished with exit code 0
```

MANUAL DEMO



```
Select from 3 basic types of suits:

1. Dec Suit
2. Ora Suit
3. Tor Suit
Select:
2

Select accessories to equip your suit:

1. Flamethrower
2. AutoRifle
3. RocketLauncher
4. Laser
5. To complete the selections
Select:
1
```



```
Select accessories to equip your suit:

1. Flamethrower
2. AutoRifle
3. RocketLauncher
4. Laser
5. To complete the selections
Select:
3

Select accessories to equip your suit:

1. Flamethrower
2. AutoRifle
3. RocketLauncher
4. Laser
5. To complete the selections
Select:
5

Ora Armored Suits, Flamethrower, RocketLauncher :
    ==> Total Cost: 1700.0k TL,
    ==> Total Weight: 39.5kg
```