

# Homework 1 Report

**Elif Akgün**

1801042251

## 1. Part 1

### 1.1 Design Explanation

I used Strategy Pattern in this part because the customer wants to be able to change between solving methods dynamically and might need more functionalities in the future. The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

In my design, there is an interface which is SolvingMethod. It has only solve method so all classes implements SolvingMethod, which are GaussianElimination and MatrixInversion, require reimplement this method. Also there is LinearSolver class. It has an SolvingMethod object and helps to change between solving methods dynamically. Constructor of LinearSolver takes SolvingMethod object and thanks to polymorphism, when we invoke solve method, program goes to corresponding solving method's solve method.

GaussianElimination class solves equations with gaussian elimination method. It has two field which are matrix of coefficients and matrix of RHS of equations. It also has helper

methods. First it converts the coefficients matrix like this format  $\begin{pmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \end{pmatrix}$ , then makes

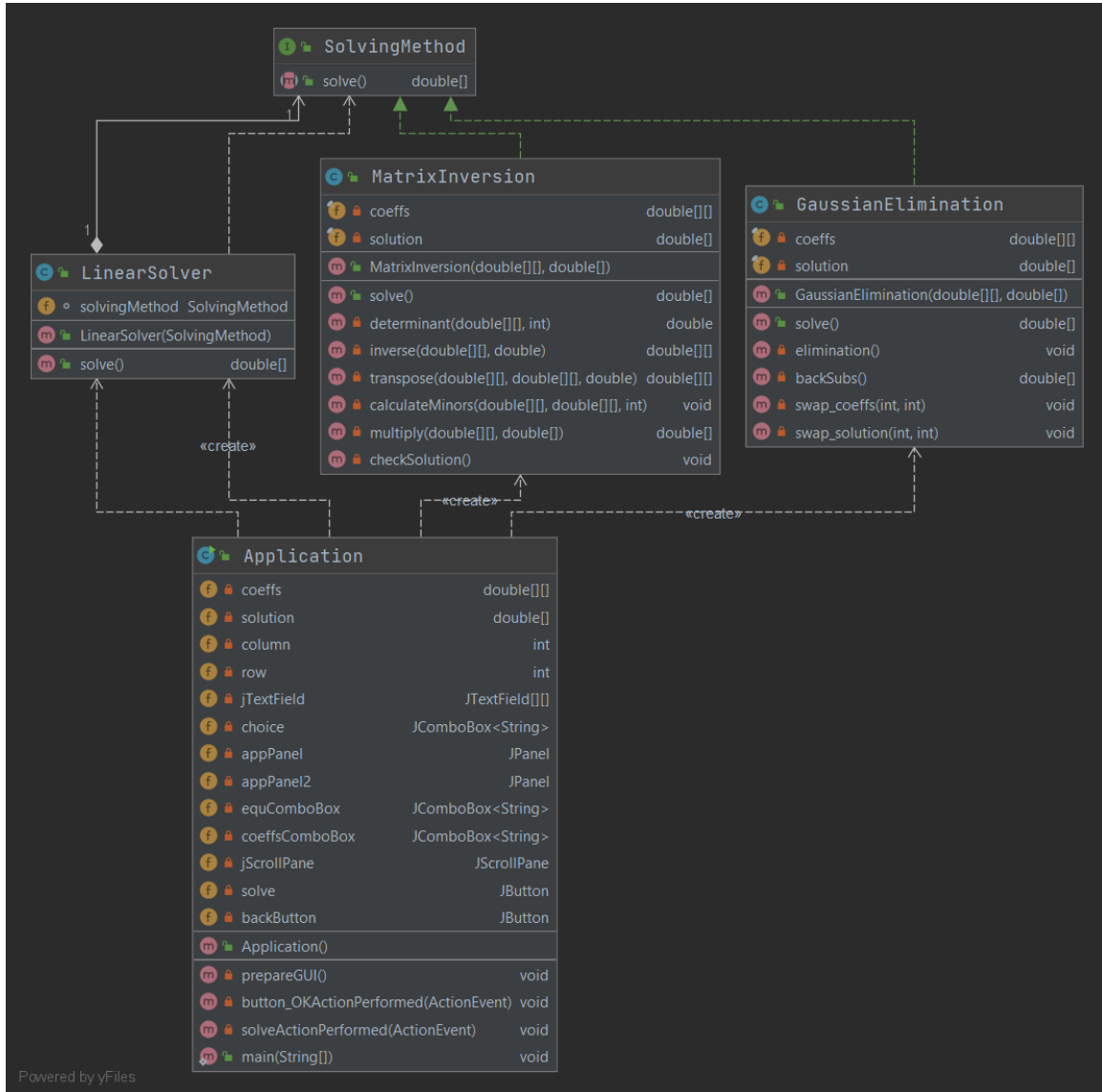
back substitution to find answer.

MatrixInversion class solves equations with matrix inversion method. It has two field which are matrix of coefficients and matrix of RHS of equations. It also has helper methods. This class first get inverse of coefficient matrix because if  $AX = B$ , then  $X = A^{-1}B$ . That's why, it multiplies inverse matrix and solution matrix to find answer with multiply method.

I prepared a GUI and it includes some layers, buttons, comboBox etc. With this interface, you can enter the number of equations and coefficients. Also you can choose solving method dynamically with Change Method button. If you want to add new solving method, all you need is write its class then add this class to comboBox. Then you should add new if statement for new solving method to Application class which includes main method. In this application, the number of equations and the number of unknowns must be equal.

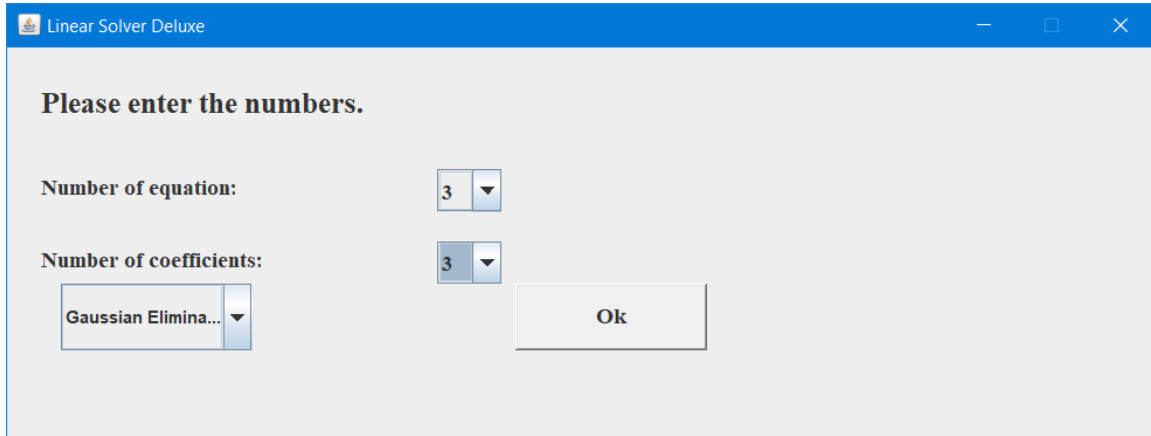
## 1.2 Class Diagram

Figure 1: Class Diagram of Part 1



### 1.3 Tests

Figure 2: User Interface 1



**Linear Solver Deluxe**

Please enter the numbers.

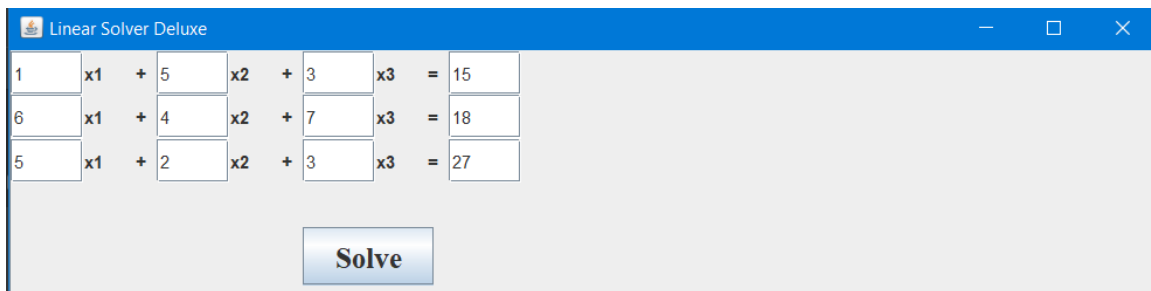
Number of equation:  ▼

Number of coefficients:  ▼

Gaussian Elimina... ▼

Ok

Figure 3: User Interface 2

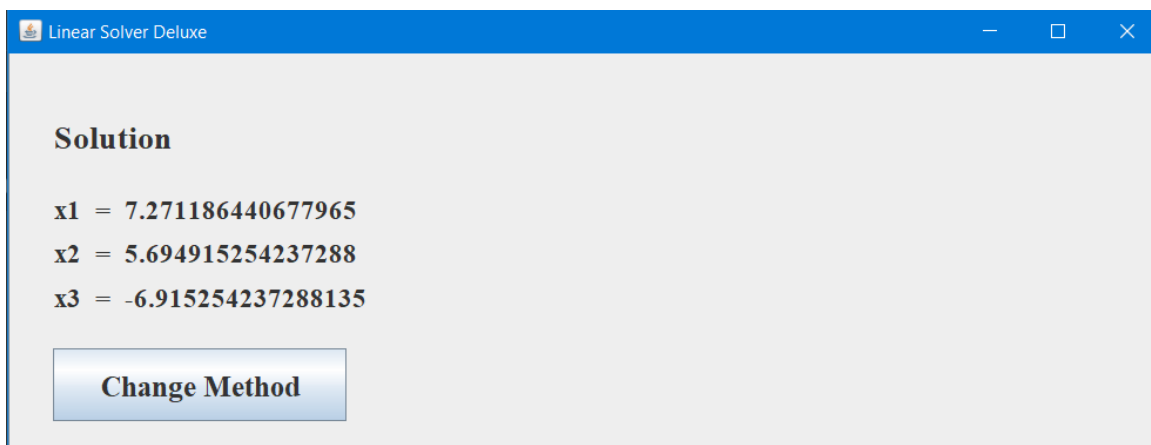


**Linear Solver Deluxe**

1	x1	+	5	x2	+	3	x3	=	15
6	x1	+	4	x2	+	7	x3	=	18
5	x1	+	2	x2	+	3	x3	=	27

Solve

Figure 4: Solution with Gaussian Elimination Method



**Linear Solver Deluxe**

**Solution**

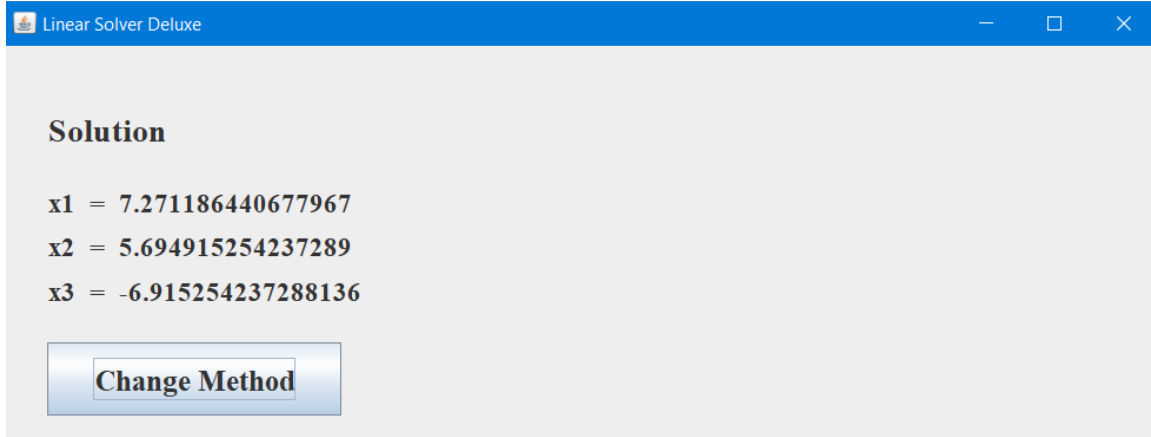
x1 = 7.271186440677965

x2 = 5.694915254237288

x3 = -6.915254237288135

Change Method

Figure 5: Solution with Matrix Inversion Method



## 2. Part 2

### 2.1 Design Explanation

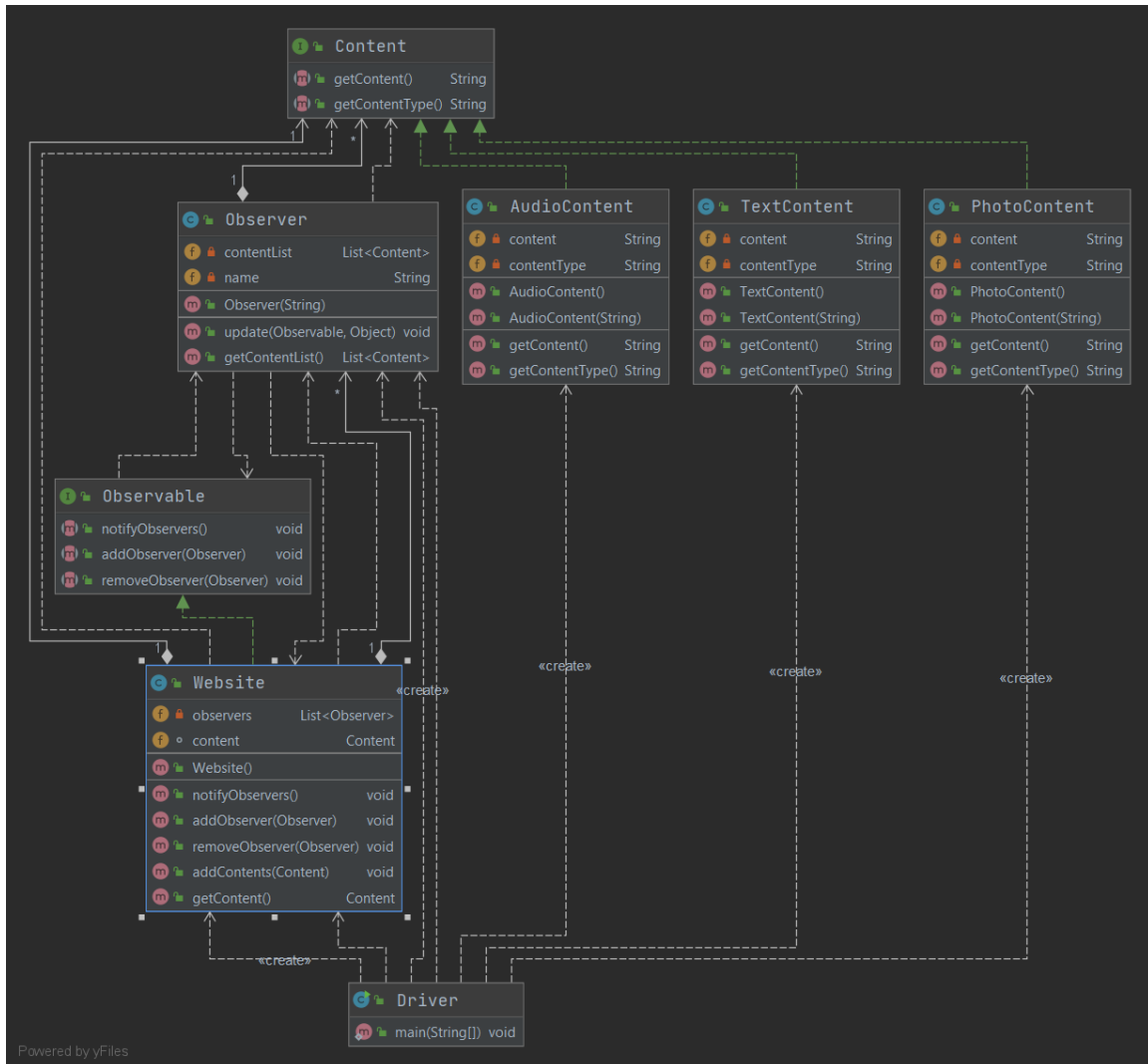
I used observer pattern in this part because there is one subject which is website and there are some observers which are subscribers. With the observer pattern, the subject is the object that contains the state and controls it. So, there is one subject with state. The observers, use the state, even if they don't own it. There are many observers and they rely on the subject to tell them when its state changes. So there is a relationship between the one subject to the many observers.

In my design, there are two interfaces which are *Observable* and *Content*. *Observable* has *notifyObservers*, *addObserver*, and *removeObserver* methods. *Website* implements *Observable* and also implements these methods for subscription. *Website* has an *Observer* list to keep observers. With *addObserver* method, new observers can be added. This class also has *addContents* method and with this method new contents can be added. When new content is added, *notifyObservers* method checks every observer and if the observer is interested in this content, it updates this observer. If the observer is not interested in it, it does not update. There are three classes that implement *Content* interface which are *TextContent*, *AudioContent*, and *PhotoContent*. Each class keeps its own type and content. *Observer* class has a *contentList* which keeps the observer's contents.

You can add and remove observers easily with *addObserver* and *removeObserver* methods. If you want to add a new observer, all you need is create an *Observer* object and add its contents. You can add new content easily, too. You should only write a class which implements *Content* interface.

## 2.2 Class Diagram

Figure 6: Class Diagram of Part 2



## 2.3 Tests

Figure 7: Main Method

```
public static void main(String[] args) {
    Website website = new Website();

    Observer o1 = new Observer( name: "Elif");
    o1.getContentList().add(new TextContent());

    Observer o2 = new Observer( name: "Esra");
    o2.getContentList().add(new PhotoContent());
    o2.getContentList().add(new AudioContent());

    Observer o3 = new Observer( name: "Seda");
    o3.getContentList().add(new PhotoContent());
    o3.getContentList().add(new AudioContent());
    o3.getContentList().add(new TextContent());

    website.addObserver(o1);
    website.addObserver(o2);
    website.addObserver(o3);

    website.addContents(new PhotoContent("Moon"));
    website.addContents(new AudioContent("Evgeny Grinko - Field"));
    website.addContents(new TextContent("Sound of Space"));
}
```

Figure 8: Output

```
Esra - New photo content: Moon
Seda - New photo content: Moon
Esra - New audio content: Evgeny Grinko - Field
Seda - New audio content: Evgeny Grinko - Field
Elif - New text content: Sound of Space
Seda - New text content: Sound of Space

Process finished with exit code 0
```

### 3. Part 3

#### 3.1 Design Explanation

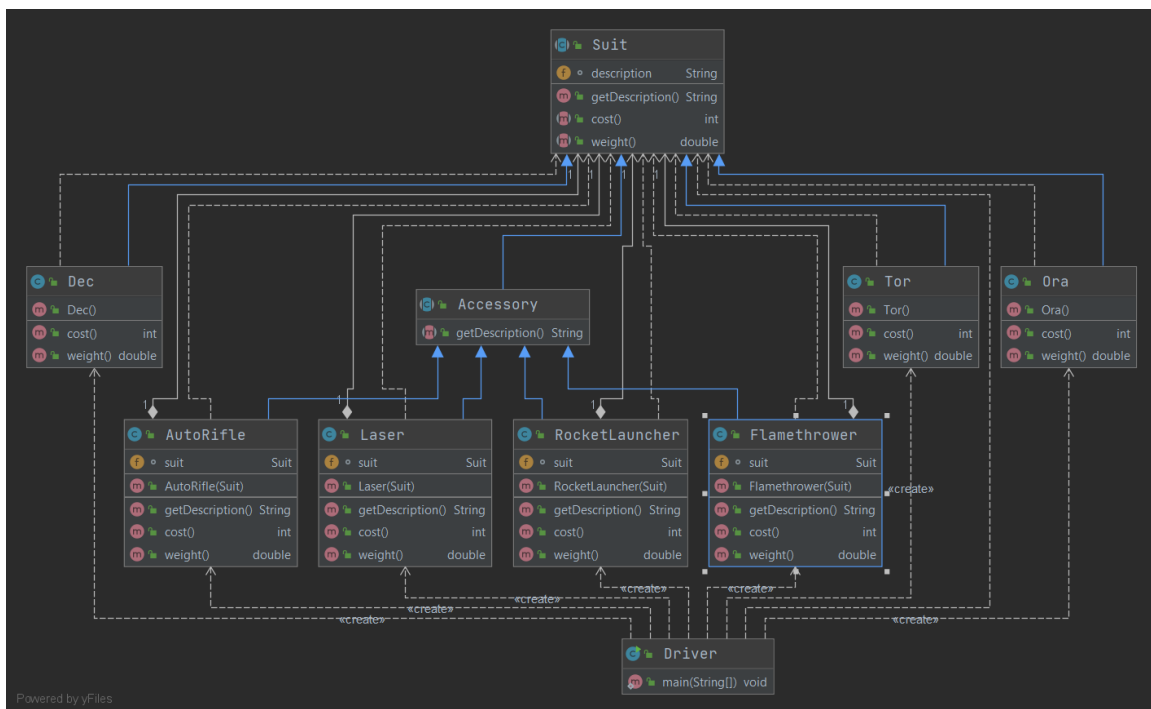
I used decorator pattern in this part because the user of the software should be able to designate any combination of accessories dynamically at run time. The decorator pattern attaches additional responsibilities to an object dynamically.

In my design, Suit is an abstract class with two abstract methods: cost and weight. So all classes extend the Suit which are Dec, Ora and Tor, have to implement this methods. Also there is an abstract class Accessory extends Suit and it has getDescription method. So it is required that the accessories ,which are Flamethrower, AutoRifle, RocketLauncher and Laser, implement the getDescription, cost and weight methods.

Each accessory has-a (wraps) a suit, which means the accessory has an instance variable that holds a reference to a suit. In this way, you can add accessories that you want at run time.

#### 3.2 Class Diagram

Figure 9: Class Diagram of Part 3



### 3.3 Tests

Figure 10: Main Method

```

public static void main(String[] args) {
    Scanner scan = new Scanner(System.in);
    int choice = -1;
    Suit suit;

    while(!(choice == 1 || choice == 2 || choice == 3)){
        System.out.println("Please enter the number of suit type you want.");
        System.out.println("1) Dec (500k TL, 25kg)");
        System.out.println("2) Ora (1500k TL, 30kg)");
        System.out.println("3) Tor (5000k TL, 50kg)");
        choice=scan.nextInt();
    }

    if(choice == 1)
        suit = new Dec();
    else if(choice == 2)
        suit = new Ora();
    else
        suit = new Tor();

    choice = -1;

    while(!(choice == 1 || choice == 2)){
        System.out.println("Do you want any accessory?");
        System.out.println("1) Yes");
        System.out.println("2) No");
        choice=scan.nextInt();
    }

    if(choice == 1){
        choice = -1;

        while(choice != 0){
            System.out.println("Please enter the number of accessory you want.");
            System.out.println("0) Exit");
            System.out.println("1) Flamethrower (50k TL per item, 2k)");
            System.out.println("2) AutoRifle (30k TL per item, 1.5kg)");
            System.out.println("3) RocketLauncher (150k TL per item, 7.5kg)");
            System.out.println("4) Laser (200k TL per item, 5.5kg)");

            choice=scan.nextInt();

            if(choice == 1)
                suit = new Flamethrower(suit);
            else if(choice == 2)
                suit = new AutoRifle(suit);
            else if(choice == 3)
                suit = new RocketLauncher(suit);
            else if(choice == 4)
                suit = new Laser(suit);
        }
    }

    System.out.println("Your suit:      " + suit.getDescription());
    System.out.println("Total cost:    " + suit.cost());
    System.out.println("Total weight:  " + suit.weight());
}

```



Figure 11: Output

```
Please enter the number of suit type you want.
1) Dec (500k TL, 25kg)
2) Ora (1500k TL, 30kg)
3) Tor (5000k TL, 50kg)
2
Do you want any accessory?
1) Yes
2) No
1
Please enter the number of accessory you want.
0) Exit
1) Flamethrower (50k TL per item, 2k)
2) AutoRifle (30k TL per item, 1.5kg)
3) RocketLauncher (150k TL per item, 7.5kg)
4) Laser (200k TL per item, 5.5kg)
3
Please enter the number of accessory you want.
0) Exit
1) Flamethrower (50k TL per item, 2k)
2) AutoRifle (30k TL per item, 1.5kg)
3) RocketLauncher (150k TL per item, 7.5kg)
4) Laser (200k TL per item, 5.5kg)
1
Please enter the number of accessory you want.
0) Exit
1) Flamethrower (50k TL per item, 2k)
2) AutoRifle (30k TL per item, 1.5kg)
3) RocketLauncher (150k TL per item, 7.5kg)
4) Laser (200k TL per item, 5.5kg)
0
Your suit:      Ora, RocketLauncher, Flamethrower
Total cost:     1700
Total weight:   39.5

Process finished with exit code 0
```