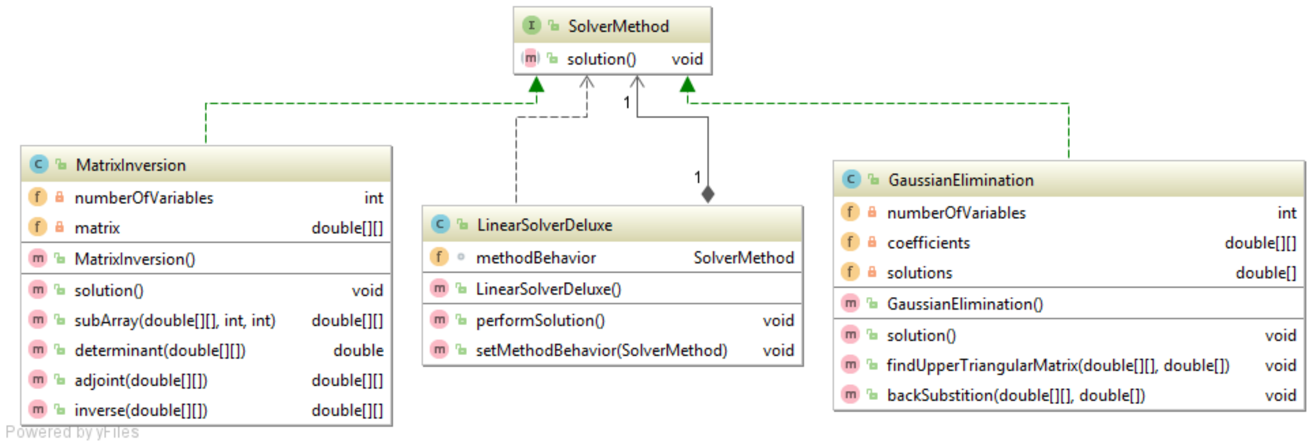
Gebze Technical University Computer Engineering Department

CSE443 - Object Oriented Analysis and Design

HOMEWORK1 REPORT

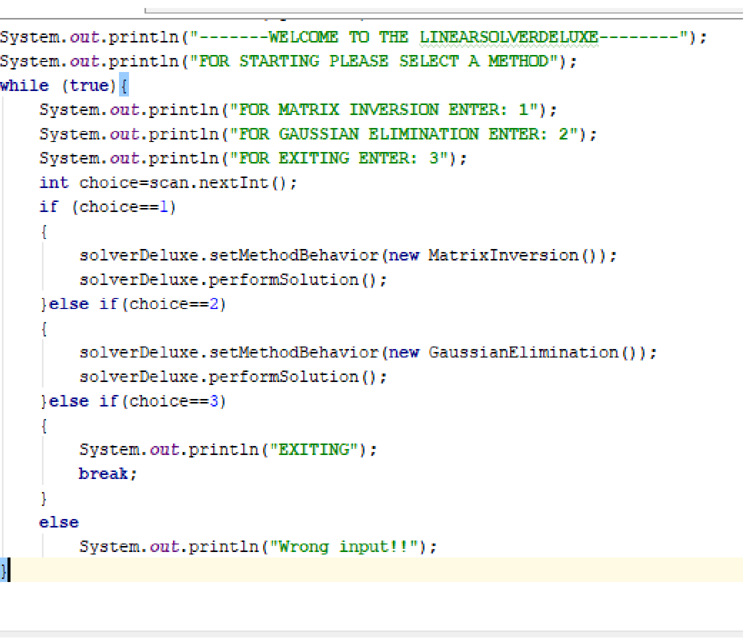
Sevgi Bayansalduz
151044076

PART 1

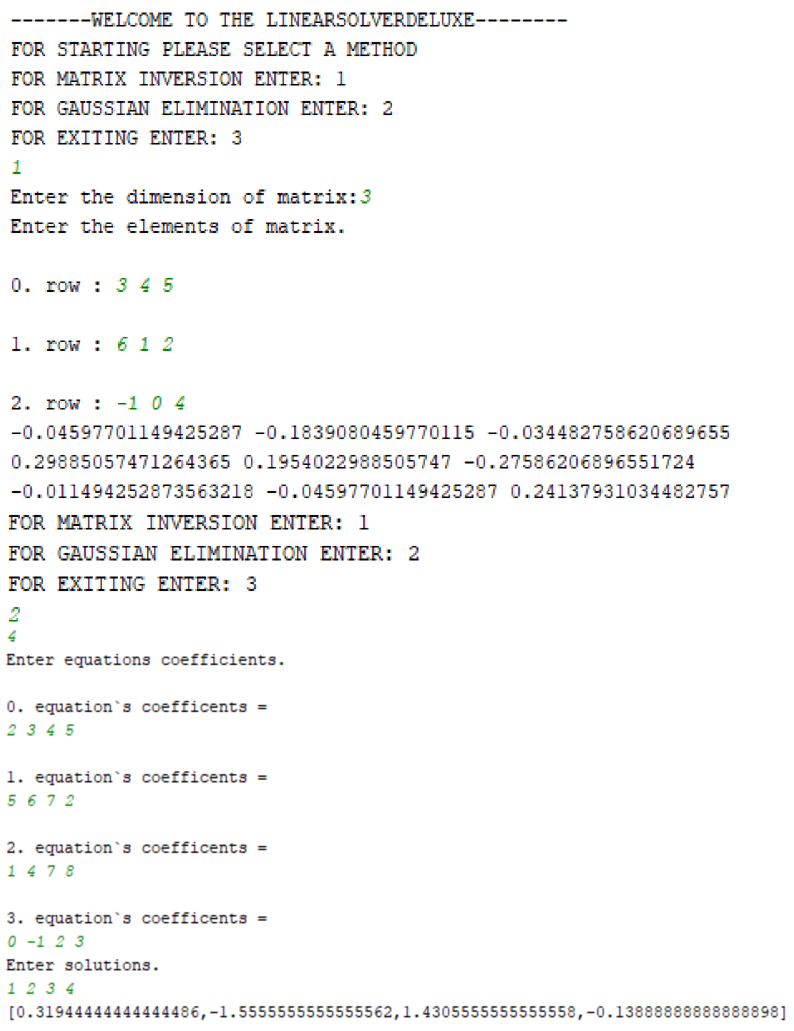


I used the strategy pattern for the problem in part one. Because thanks to the strategy pattern we can change solving method dynamically at run time.LinearSolverDeluxe method has an instance of the SolverMethod. Also has setMethodBehovir method to create this instance dynamically.

SolverMethos is an interface, and this interface implemented by two class, whic are called MatrixInversion and GaussionElimination. If user wants new solving methods, it can simply added as new class which implements SolverMethod.

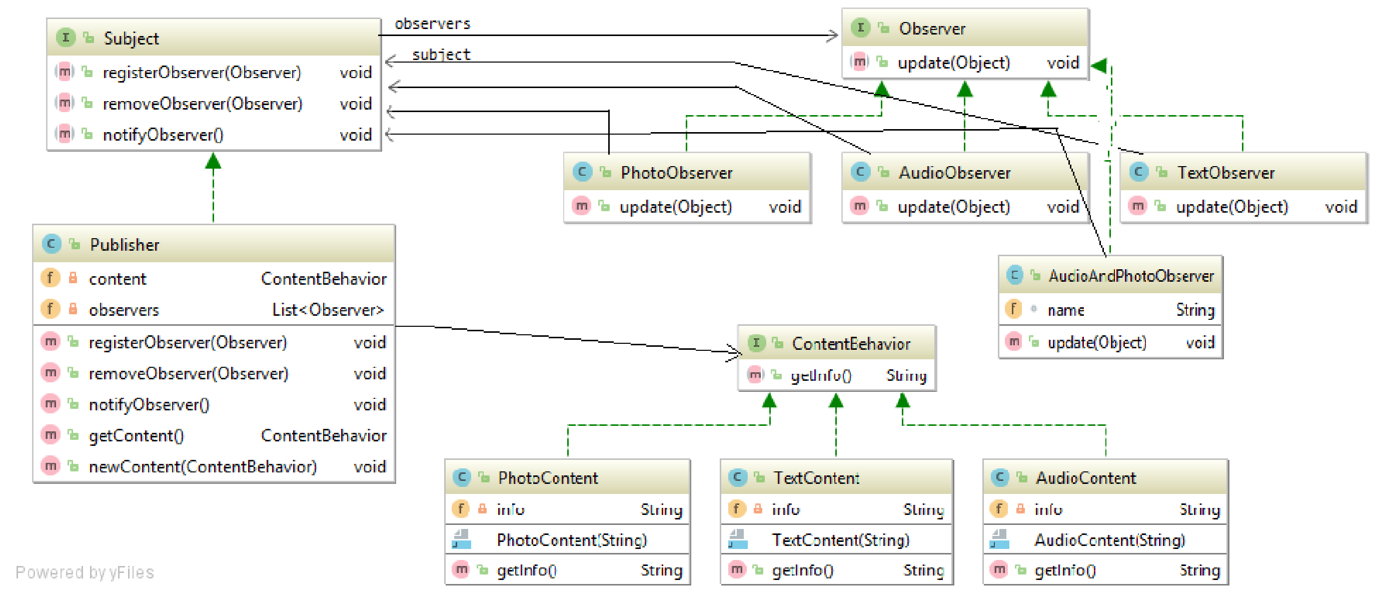


```
System.out.println("-------WELCOME TO THE LINEARSOLVERDELUXE--------");
System.out.println("FOR STARTING PLEASE SELECT A METHOD");
while (true){
    System.out.println("FOR MATRIX INVERSION ENTER: 1");
    System.out.println("FOR GAUSSIAN ELIMINATION ENTER: 2");
    System.out.println("FOR EXITING ENTER: 3");
    int choice=scan.nextInt();
    if (choice==1)
    {
        solverDeluxe.setMethodBehavior(new MatrixInversion());
        solverDeluxe.performSolution();
    }else if(choice==2)
    {
        solverDeluxe.setMethodBehavior(new GaussianElimination());
        solverDeluxe.performSolution();
    }else if(choice==3)
    {
        System.out.println("EXITING");
        break;
    }
    else
        System.out.println("Wrong input!!");
}
```



```
-------WELCOME TO THE LINEARSOLVERDELUXE--------
FOR STARTING PLEASE SELECT A METHOD
FOR MATRIX INVERSION ENTER: 1
FOR GAUSSIAN ELIMINATION ENTER: 2
FOR EXITING ENTER: 3
1
Enter the dimension of matrix:3
Enter the elements of matrix.

0. row : 3 4 5

1. row : 6 1 2

2. row : -1 0 4
-0.04597701149425287 -0.1839080459770115 -0.034482758620689655
0.2988505747126436 5 0.1954022988505747 -0.27586206896551724
-0.011494252873563218 -0.04597701149425287 0.24137931034482757
FOR MATRIX INVERSION ENTER: 1
FOR GAUSSIAN ELIMINATION ENTER: 2
FOR EXITING ENTER: 3
2
4
Enter equations coefficients.

0. equation`s coefficents =
2 3 4 5

1. equation`s coefficents =
5 6 7 2

2. equation`s coefficents =
1 4 7 8

3. equation`s coefficents =
0 -1 2 3
Enter solutions.
1 2 3 4
[0.31944444444444486,-1.5555555555555562,1.4305555555555558,-0.13888888888888898]
```

PART 2

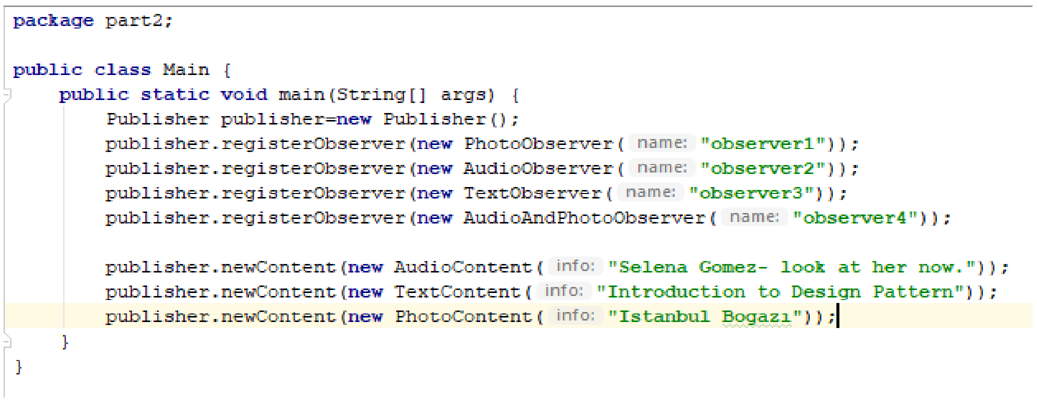


I used Observer Pattern with the Strategy Pattern for the problem in part2. Because Observer - defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Observer Patern Part: The Subject interface is created to control observers such as removing ,adding notifying observers. After creating interface, Publisher class is created to implements Subject interface. After publisher part is done, Observer part is created. For Observer Part, firstly Observer interface is created, secondly concrete observers are created to implements the Observer interface.

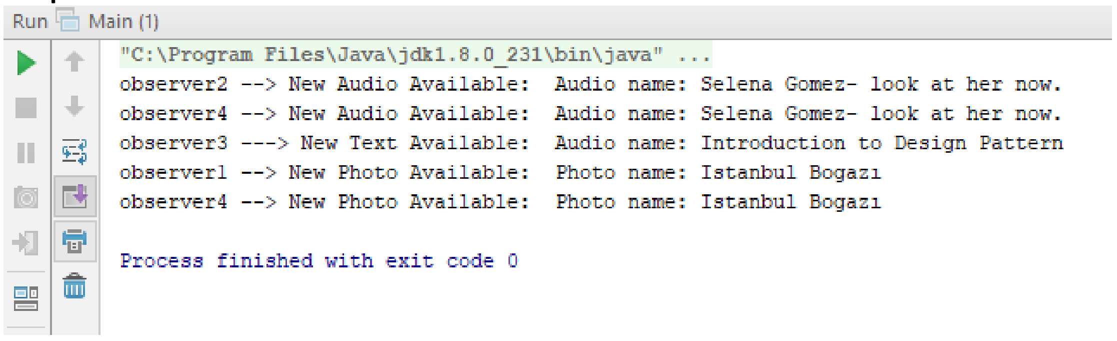
Strategy Pattern Part: Publisher class has an instance of ContentBehavior. ContentBehavior interface hold the information about the contents. PhotoContent, TextContent and AudioContent classes implements the ContentBehavior interface. ContentBehavior instance can change dynamically. Also new content can be added easily; only 2 class should be add: a class which implements ContentBehavior, and its observer class.

Main:



```java
package part2;

public class Main {
    public static void main(String[] args) {
        Publisher publisher=new Publisher();
        publisher.registerObserver(new PhotoObserver( name: "observer1"));
        publisher.registerObserver(new AudioObserver( name: "observer2"));
        publisher.registerObserver(new TextObserver( name: "observer3"));
        publisher.registerObserver(new AudioAndPhotoObserver( name: "observer4"));

        publisher.newContent(new AudioContent( info: "Selena Gomez- look at her now."));
        publisher.newContent(new TextContent( info: "Introduction to Design Pattern"));
        publisher.newContent(new PhotoContent( info: "Istanbul Bogazı"));
    }
}
```

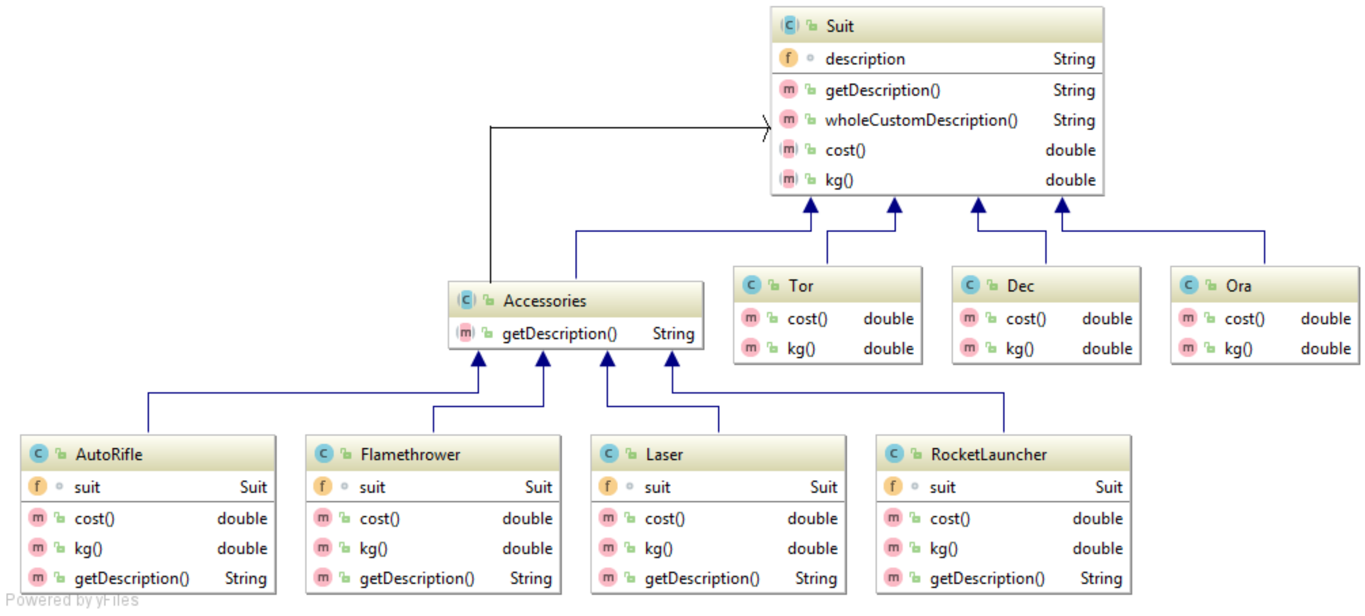
Output:



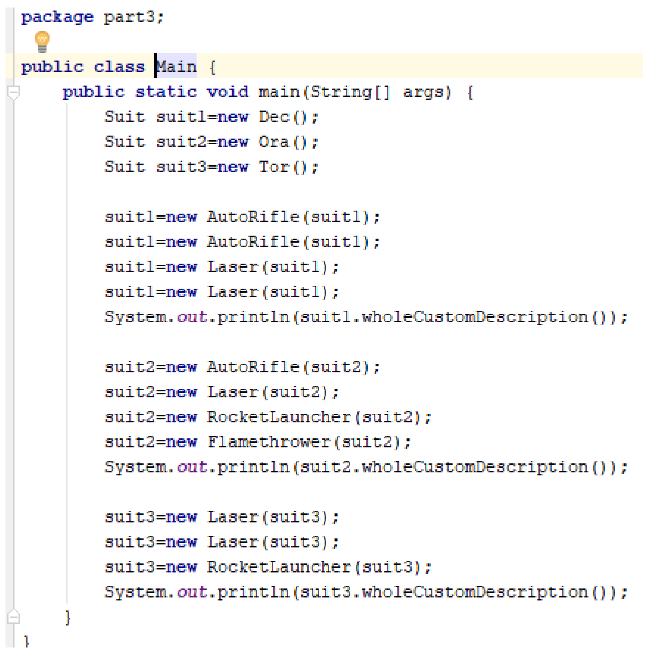
Run Main (1)

```
"C:\Program Files\Java\jdk1.8.0_231\bin\java" ...
observer2 --> New Audio Available:  Audio name: Selena Gomez- look at her now.
observer4 --> New Audio Available:  Audio name: Selena Gomez- look at her now.
observer3 ---> New Text Available:  Audio name: Introduction to Design Pattern
observer1 --> New Photo Available:  Photo name: Istanbul Bogazı
observer4 --> New Photo Available:  Photo name: Istanbul Bogazı

Process finished with exit code 0
```

Part 3

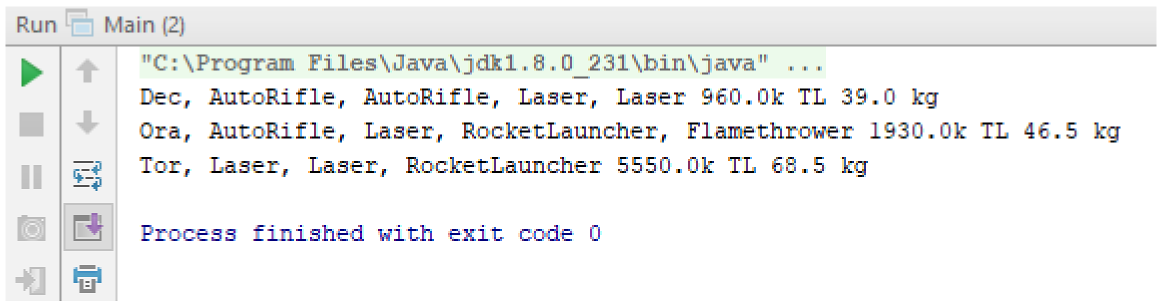


I used Decorator pattern for the problem in part3. Because additional responsibilities should attach to an object dynamically. Decorators provides this. Accessories is an abstract class for condiment decorator; AutoRifle, FlameThrower, Laser and RocketLauncher are condiment decorators and extends from the abstract class Accessories. Tor, Dec and Ora classes are concrete components, one per suit type.



```
package part3;

public class Main {
    public static void main(String[] args) {
        Suit suit1=new Dec();
        Suit suit2=new Ora();
        Suit suit3=new Tor();

        suit1=new AutoRifle(suit1);
        suit1=new AutoRifle(suit1);
        suit1=new Laser(suit1);
        suit1=new Laser(suit1);
        System.out.println(suit1.wholeCustomDescription());

        suit2=new AutoRifle(suit2);
        suit2=new Laser(suit2);
        suit2=new RocketLauncher(suit2);
        suit2=new Flamethrower(suit2);
        System.out.println(suit2.wholeCustomDescription());

        suit3=new Laser(suit3);
        suit3=new Laser(suit3);
        suit3=new RocketLauncher(suit3);
        System.out.println(suit3.wholeCustomDescription());
    }
}
```

Main

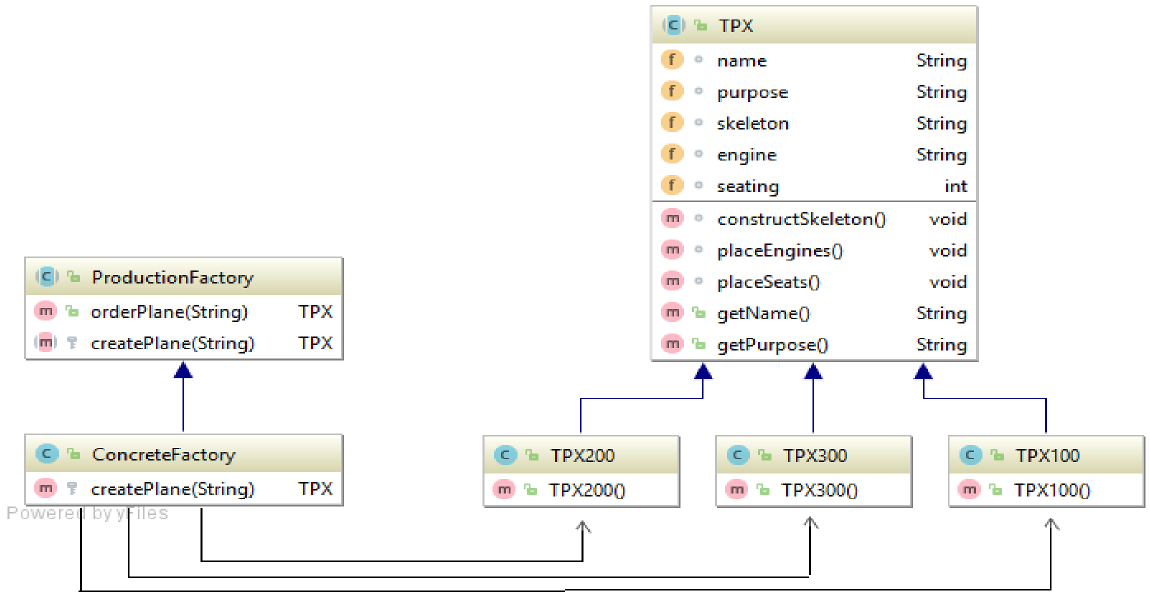


```
"C:\Program Files\Java\jdk1.8.0_231\bin\java" ...
Dec, AutoRifle, AutoRifle, Laser, Laser 960.0k TL 39.0 kg
Ora, AutoRifle, Laser, RocketLauncher, Flamethrower 1930.0k TL 46.5 kg
Tor, Laser, Laser, RocketLauncher 5550.0k TL 68.5 kg

Process finished with exit code 0
```

Output

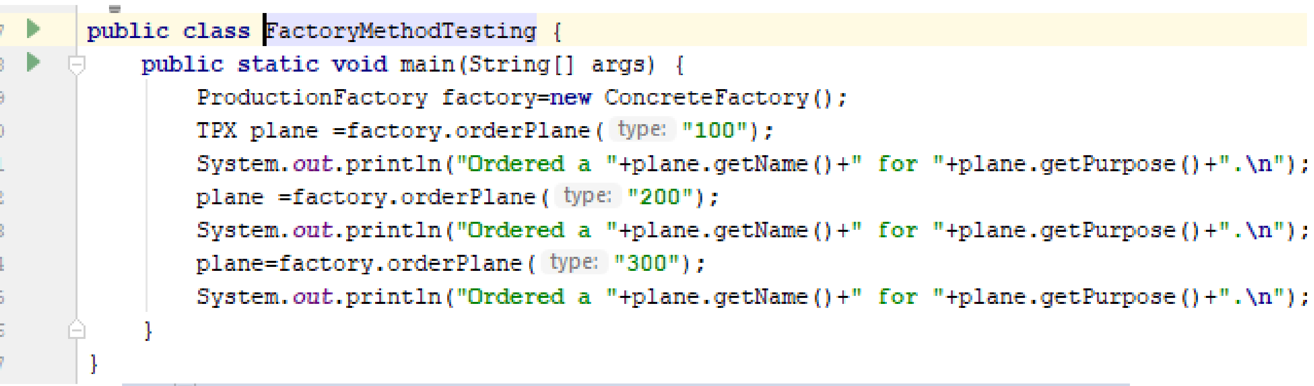
PART 4

Factory Method



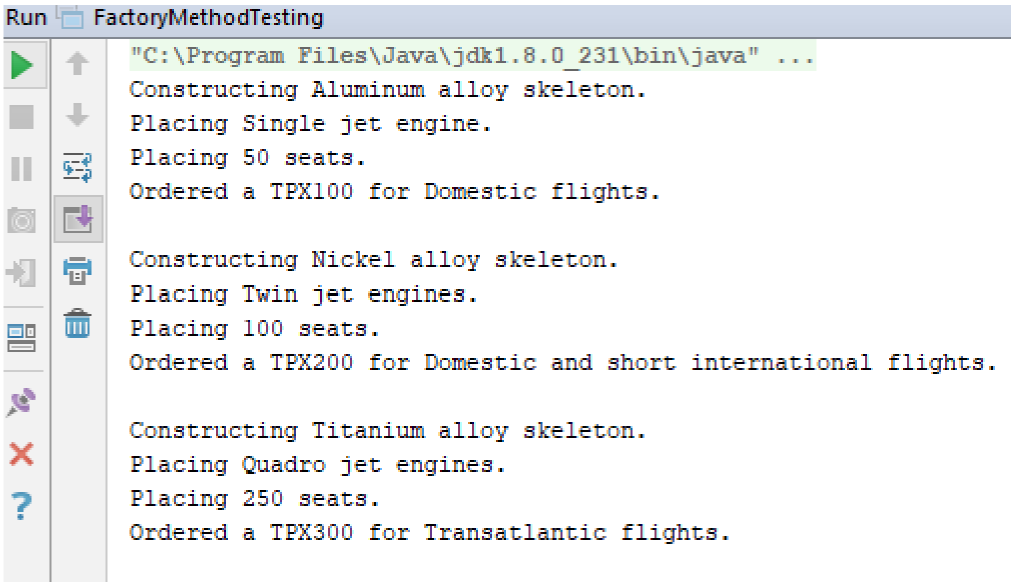
I created these classes based on the pizzastore sample which is in the course`s textbook. TPX is an abstract class ; its includes methods for aircraft construction process. TPX100, TPX200 and TPX300 extends from the TPX class; these classes set sttributes of the TPX in their default constructor.

Production Factory is an abstract class; Concrete Factory extends from the ProductionFactory and implements the createPlane method. It takes a string for a plane type then create an instance according to given type.



```java
public class FactoryMethodTesting {
    public static void main(String[] args) {
        ProductionFactory factory=new ConcreteFactory();
        TPX plane =factory.orderPlane( type: "100");
        System.out.println("Ordered a "+plane.getName()+" for "+plane.getPurpose()+".\n");
        plane =factory.orderPlane( type: "200");
        System.out.println("Ordered a "+plane.getName()+" for "+plane.getPurpose()+".\n");
        plane=factory.orderPlane( type: "300");
        System.out.println("Ordered a "+plane.getName()+" for "+plane.getPurpose()+".\n");
    }
}
```

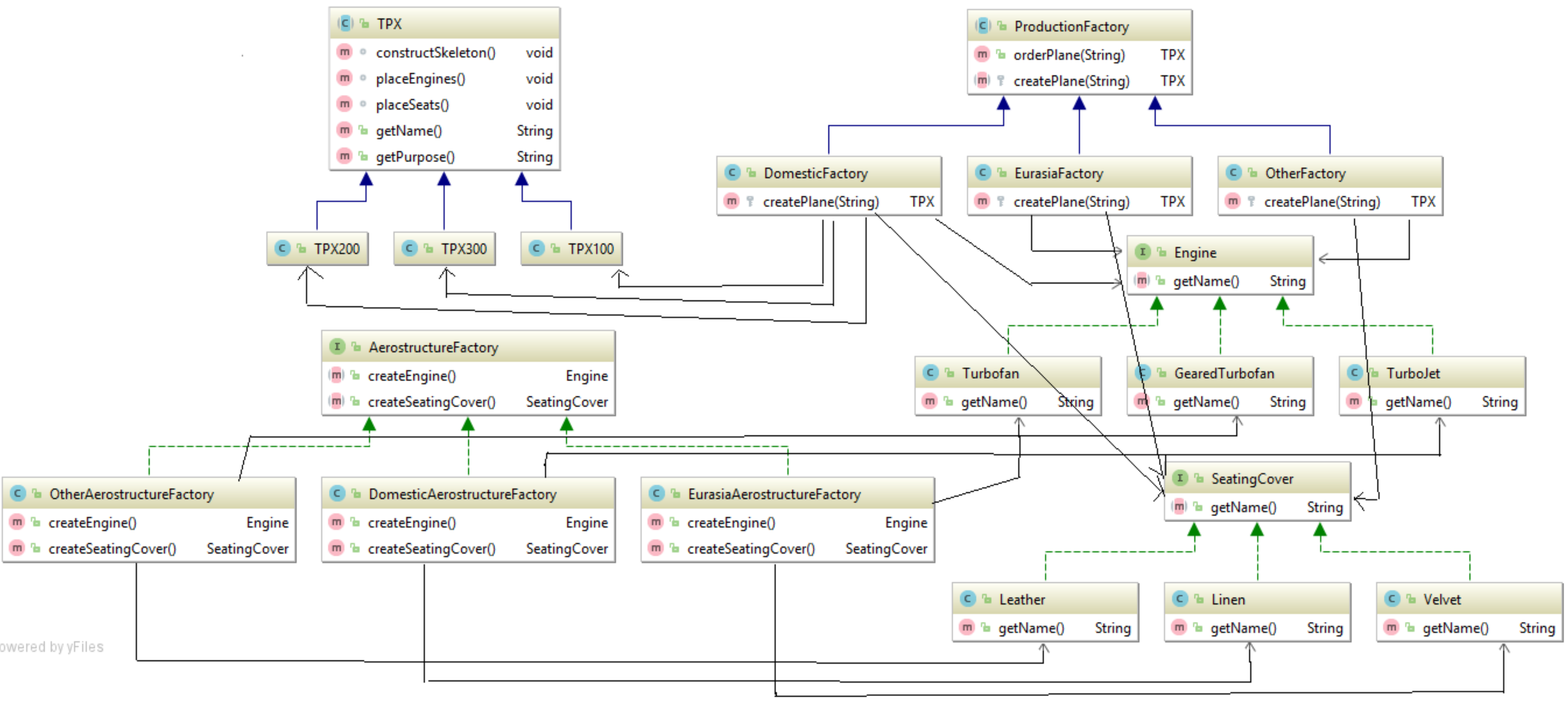
Main



```
"C:\Program Files\Java\jdk1.8.0_231\bin\java" ...
Constructing Aluminum alloy skeleton.
Placing Single jet engine.
Placing 50 seats.
Ordered a TPX100 for Domestic flights.

Constructing Nickel alloy skeleton.
Placing Twin jet engines.
Placing 100 seats.
Ordered a TPX200 for Domestic and short international flights.

Constructing Titanium alloy skeleton.
Placing Quadro jet engines.
Placing 250 seats.
Ordered a TPX300 for Transatlantic flights.
```

Output

Abstract Factory Pattern



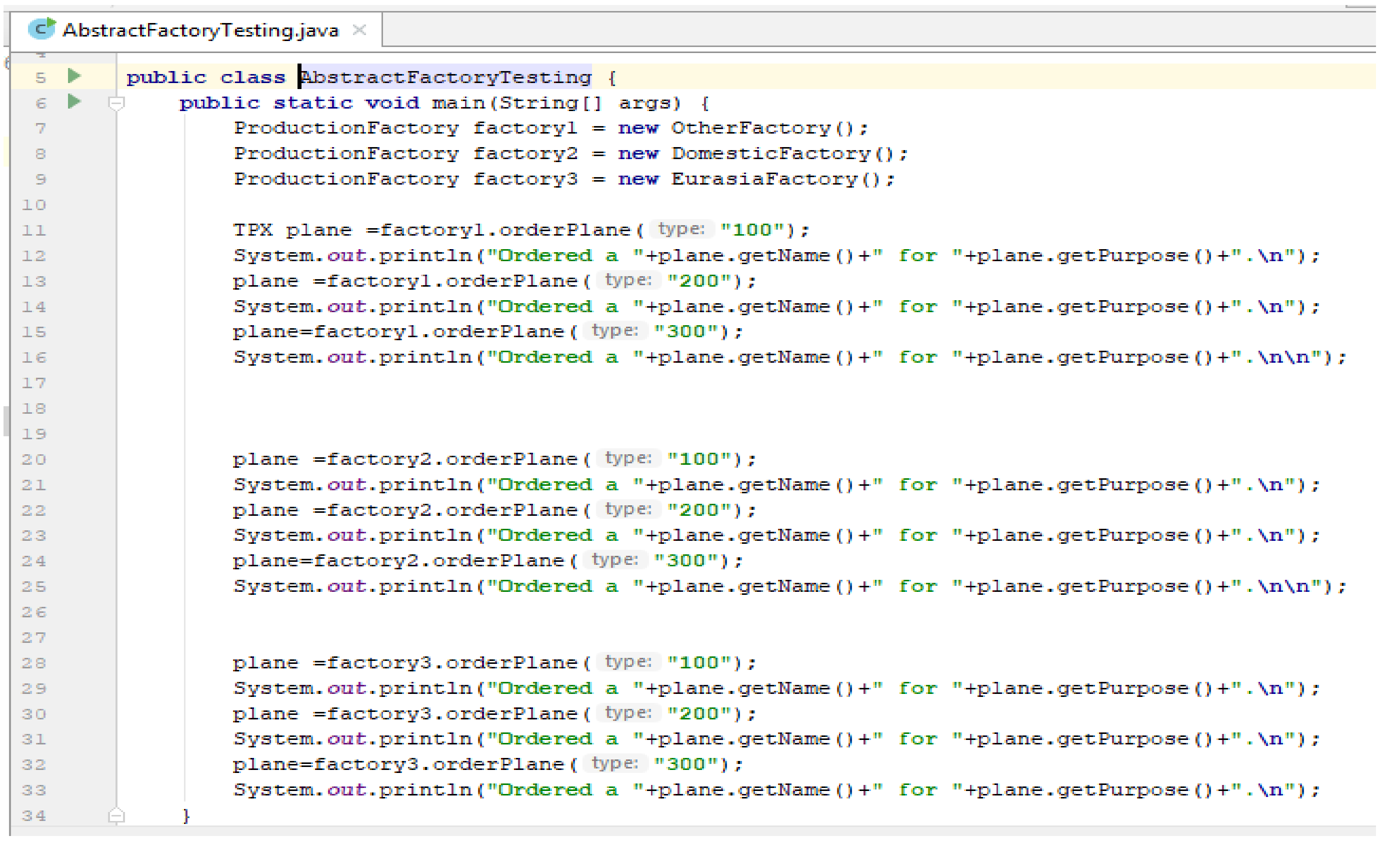
(EurasiaFactory and OtherFactory also has aggregation with the concrete TPX classes. But I didnt draw these dependencies ,because i didnt want to the diagram be more complicated.)

I create these classes, their dependencies and implementations according to PizzaIngredientFactory sample which is in the course`s textbook.

AerosStructureFactory is an abstract classes and includes create methods for the aerostructure of the plane such as engine. And its concrete classes creates aerostructure according to their local needs.

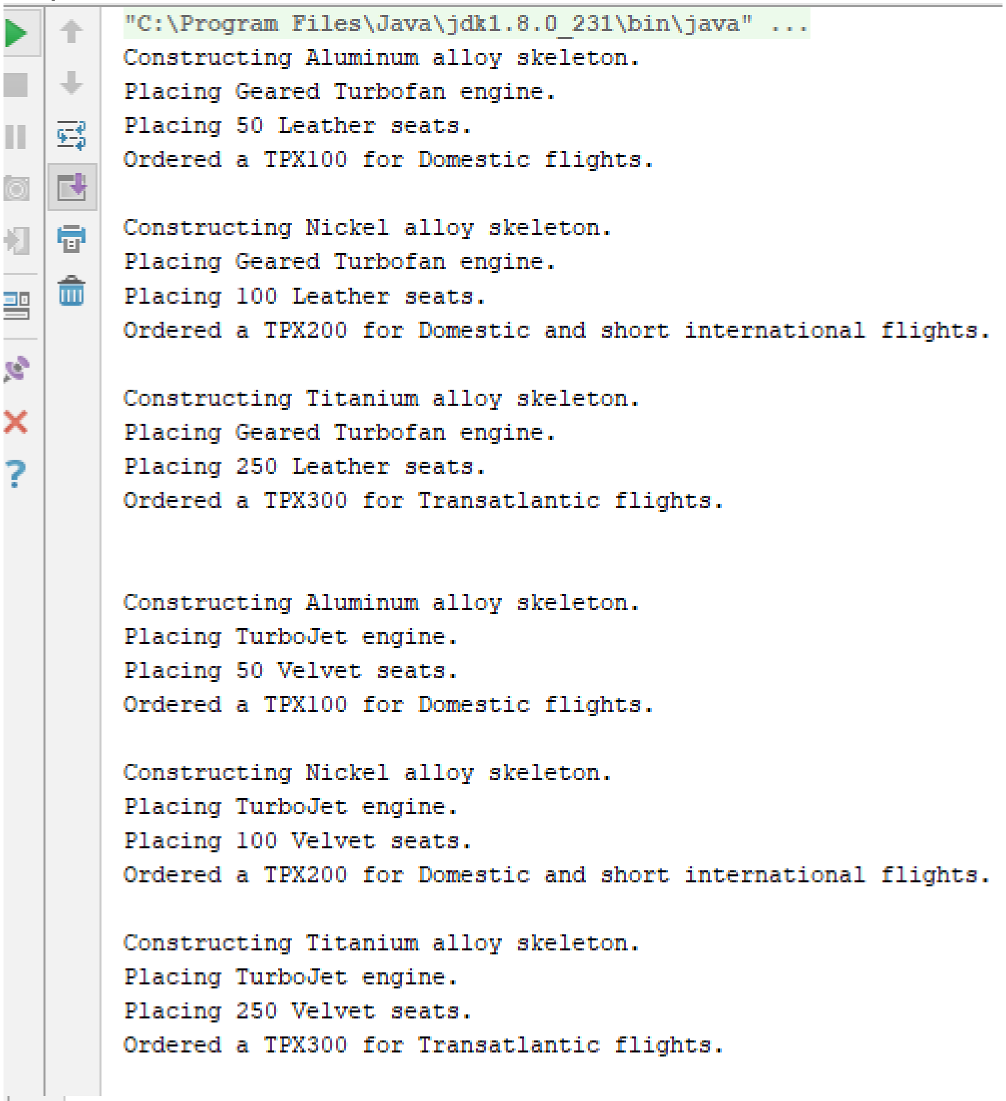
Engine and SeatingCover is an interface for aerostructure products.

Main:

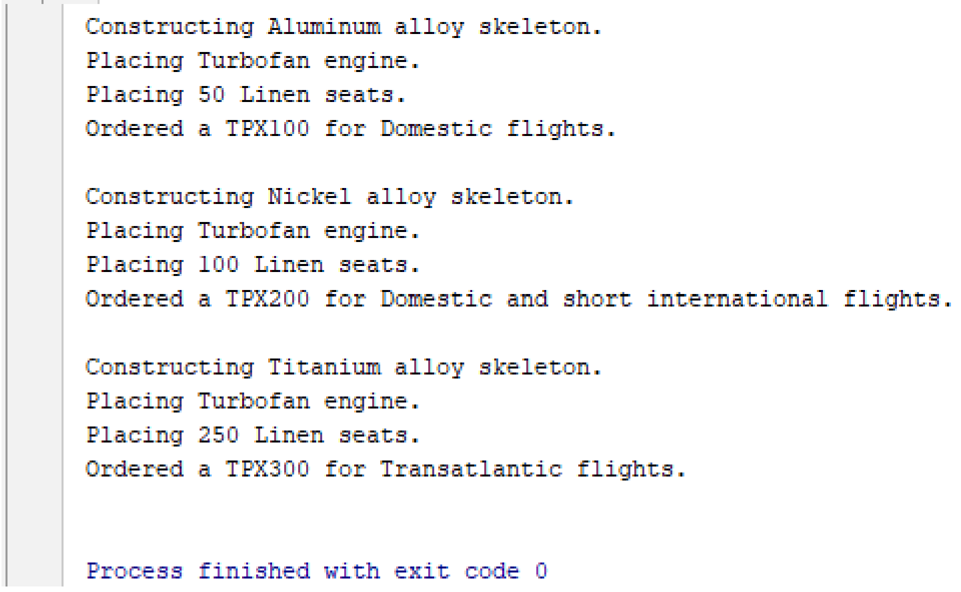

AbstractFactoryTesting.java

```java
public class AbstractFactoryTesting {
    public static void main(String[] args) {
        ProductionFactory factory1 = new OtherFactory();
        ProductionFactory factory2 = new DomesticFactory();
        ProductionFactory factory3 = new EurasiaFactory();

        TPX plane =factory1.orderPlane( type: "100");
        System.out.println("Ordered a "+plane.getName()+" for "+plane.getPurpose()+".\n");
        plane =factory1.orderPlane( type: "200");
        System.out.println("Ordered a "+plane.getName()+" for "+plane.getPurpose()+".\n");
        plane=factory1.orderPlane( type: "300");
        System.out.println("Ordered a "+plane.getName()+" for "+plane.getPurpose()+".\n\n");



        plane =factory2.orderPlane( type: "100");
        System.out.println("Ordered a "+plane.getName()+" for "+plane.getPurpose()+".\n");
        plane =factory2.orderPlane( type: "200");
        System.out.println("Ordered a "+plane.getName()+" for "+plane.getPurpose()+".\n");
        plane=factory2.orderPlane( type: "300");
        System.out.println("Ordered a "+plane.getName()+" for "+plane.getPurpose()+".\n\n");


        plane =factory3.orderPlane( type: "100");
        System.out.println("Ordered a "+plane.getName()+" for "+plane.getPurpose()+".\n");
        plane =factory3.orderPlane( type: "200");
        System.out.println("Ordered a "+plane.getName()+" for "+plane.getPurpose()+".\n");
        plane=factory3.orderPlane( type: "300");
        System.out.println("Ordered a "+plane.getName()+" for "+plane.getPurpose()+".\n");
    }
```

Output:



```
"C:\Program Files\Java\jdk1.8.0_231\bin\java" ...
Constructing Aluminum alloy skeleton.
Placing Geared Turbofan engine.
Placing 50 Leather seats.
Ordered a TPX100 for Domestic flights.

Constructing Nickel alloy skeleton.
Placing Geared Turbofan engine.
Placing 100 Leather seats.
Ordered a TPX200 for Domestic and short international flights.

Constructing Titanium alloy skeleton.
Placing Geared Turbofan engine.
Placing 250 Leather seats.
Ordered a TPX300 for Transatlantic flights.


Constructing Aluminum alloy skeleton.
Placing TurboJet engine.
Placing 50 Velvet seats.
Ordered a TPX100 for Domestic flights.

Constructing Nickel alloy skeleton.
Placing TurboJet engine.
Placing 100 Velvet seats.
Ordered a TPX200 for Domestic and short international flights.

Constructing Titanium alloy skeleton.
Placing TurboJet engine.
Placing 250 Velvet seats.
Ordered a TPX300 for Transatlantic flights.
```



```
Constructing Aluminum alloy skeleton.
Placing Turbofan engine.
Placing 50 Linen seats.
Ordered a TPX100 for Domestic flights.

Constructing Nickel alloy skeleton.
Placing Turbofan engine.
Placing 100 Linen seats.
Ordered a TPX200 for Domestic and short international flights.

Constructing Titanium alloy skeleton.
Placing Turbofan engine.
Placing 250 Linen seats.
Ordered a TPX300 for Transatlantic flights.


Process finished with exit code 0
```