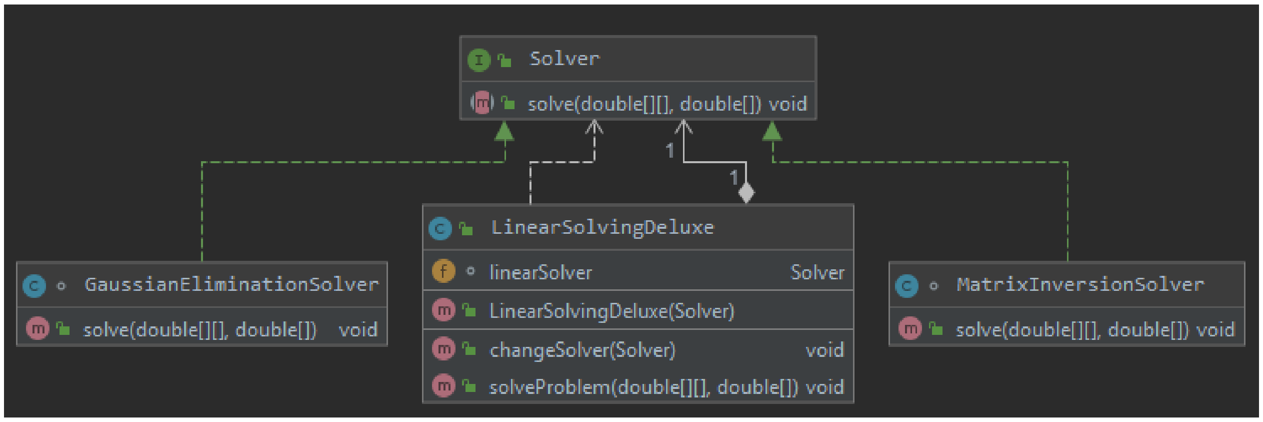
Part 1)

LinearSolverDeluxe utilizes the Strategy Design Pattern as there's only one goal, solve the linear equation and there are multiple methods for doing that such as Gaussian Elimination, Matrix Inversion and Strategy Pattern is a perfect fit here. The mentioned methods are not implemented and it prints the methods name as a placeholder.

Adding more functionality is pretty easy as all you have to do is extending **Solver** class then you can change the method type with **changeSolver(Solver anySolver)** function or give the new method to the constructor of **LinearSolvingDeluxe** class.

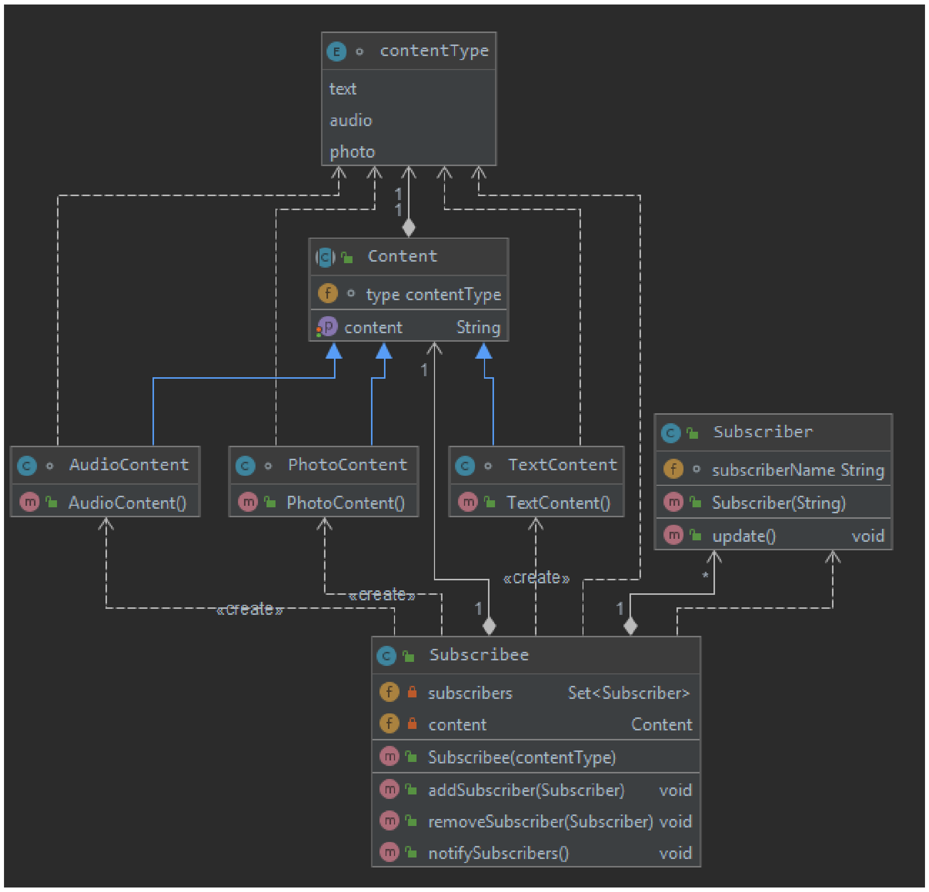

Solver
solve(double[][], double[]) void
1
1
LinearSolvingDeluxe
linearSolver Solver
LinearSolvingDeluxe(Solver)
changeSolver(Solver) void
solveProblem(double[][], double[]) void
GaussianEliminationSolver
solve(double[][], double[]) void
MatrixInversionSolver
solve(double[][], double[]) void

Part 2)

Observer design pattern is used here as subscribers want to be notified when there's a change. Observer pattern is a perfect fit because it doesn't require observants to constantly check the observer, observer notifies them on change. All subscribers have to do here is subscribe to subscribee with **AddSubscriber(Subscriber sub)** method and they can also leave subscription with removeSubscriber(Subscriber sub) method.

Subscribee's content type is defined on constructor with given ContentType and subscribers can call multiple subscribee's **AddSubscriber** method to subscribe to multiple subscribee.

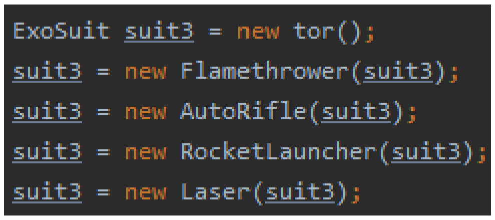
Subscribee stores their subscribers in a **Set** as time complexity of adding and removing is O(1) and since Set can only have unique elements, it prevents subscribers from subscribing twice.



Part 3)

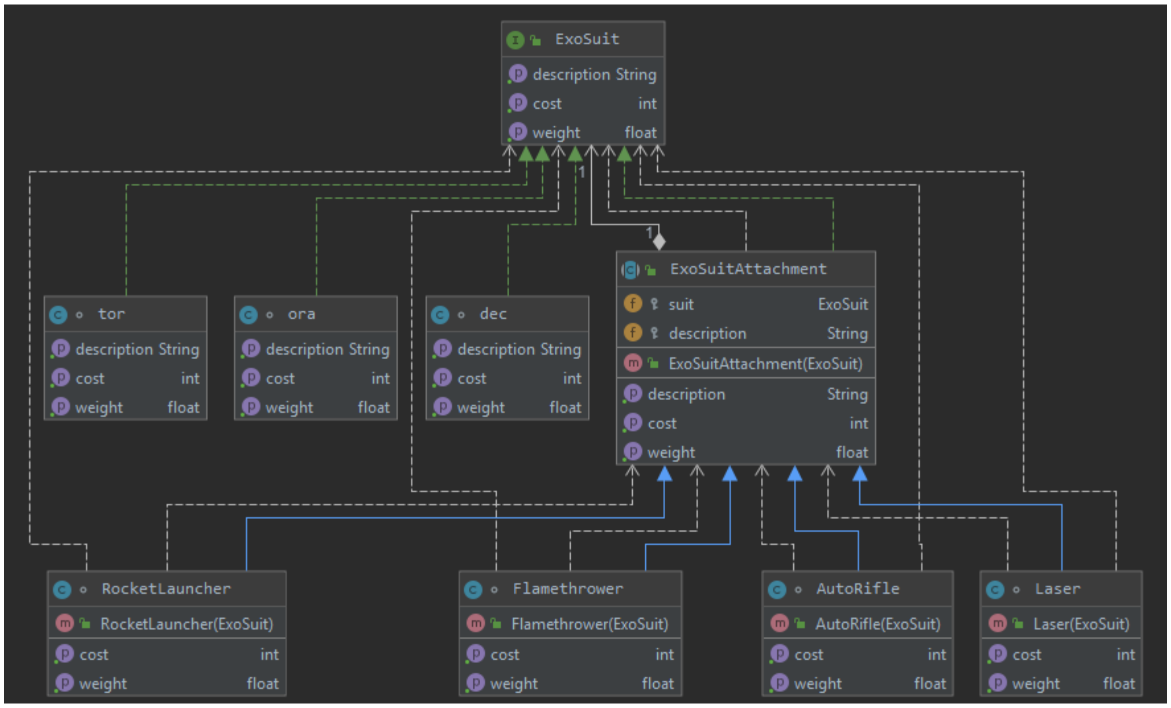
The design pattern used here is Decorator and it is a perfect fit as we have different base suits and different modifications or decorators for them and there can be a lot of different combinations and it would be really hard to maintain if you wanted to create a new class for each combination and even harder to add new modifications or suits.

With decorator pattern, you create an object of base suits, and wrap it with decorators by constructing a decorator from itself and assign it to itself.



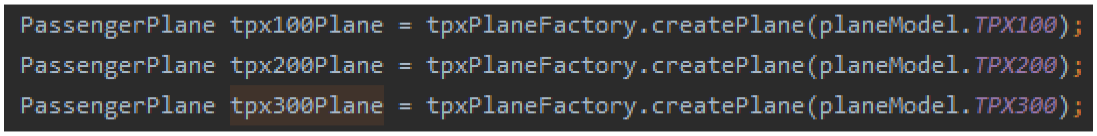
```
ExoSuit suit3 = new tor();
suit3 = new Flamethrower(suit3);
suit3 = new AutoRifle(suit3);
suit3 = new RocketLauncher(suit3);
suit3 = new Laser(suit3);
```

like this to create a tor suit with one of each modifications.



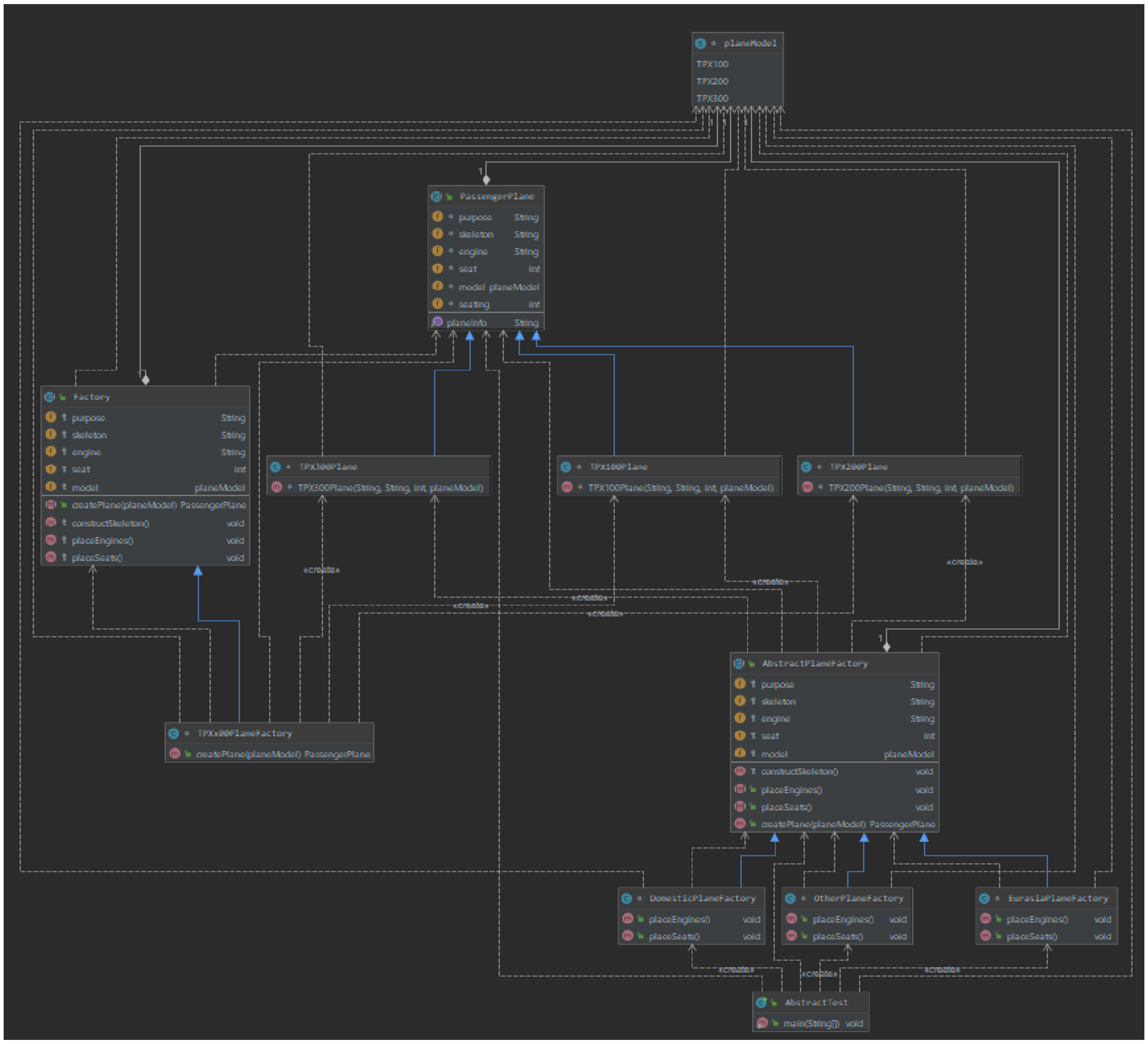
Part 4)

The factory pattern has one concrete class, and creates a plane of desired model. Different models can be created from a single factory object.



```
PassengerPlane tpx100Plane = tpxPlaneFactory.createPlane(planeModel.TPX100);
PassengerPlane tpx200Plane = tpxPlaneFactory.createPlane(planeModel.TPX200);
PassengerPlane tpx300Plane = tpxPlaneFactory.createPlane(planeModel.TPX300);
```

The abstract factory however has three concrete classes for 3 different markets. They all have different type of seats and engines. All three of them create 3 different model planes but with different engines and seats.


planeModel
TPX100
TPX200
TPX300
PassengerPlane
purpose String
skeleton String
engine String
seat int
model planeModel
seating int
planeInfo String
Factory
purpose String
skeleton String
engine String
seat int
model planeModel
createPlane(planeModel) PassengerPlane
constructSkeleton() void
placeEngines() void
placeSeats() void
TPX300Plane
TPX300Plane(String, String, int, planeModel)
TPX100Plane
TPX100Plane(String, String, int, planeModel)
TPX200Plane
TPX200Plane(String, String, int, planeModel)
«create»
TPXx00PlaneFactory
createPlane(planeModel) PassengerPlane
AbstractPlaneFactory
purpose String
skeleton String
engine String
seat int
model planeModel
constructSkeleton() void
placeEngines() void
placeSeats() void
createPlane(planeModel) PassengerPlane
DomesticPlaneFactory
placeEngines() void
placeSeats() void
OtherPlaneFactory
placeEngines() void
placeSeats() void
EurasiaPlaneFactory
placeEngines() void
placeSeats() void
AbstractTest
main(String[]) void
1