

CSE443
OBJECT ORIENTED ANALYSIS AND
DESIGN
HW2
REPORT

Hakkı Erdem Duman
151044005

Question 1

1) Object's clone method does not allow to get clone by default, it throws "CloneNotSupportedException". In Oracle documents, it says "First, if the class of this object does not implement the interface Cloneable, then a CloneNotSupportedException is thrown".

[https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#clone\(\)](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#clone())

2) Cloneable interface is implemented and clone method is overridden to throw "CloneNotSupportedException" exception.

3) If Cloneable interface is implemented, singleton object can be deep copied and that's not the thing that we want. We can prevent it by throwing "CloneNotSupportedException".

```

classDiagram
    class Dec {
        name: String
        cost: Integer
        weight: Double
    }
    class Tor {
        name: String
        cost: Integer
        weight: Double
    }
    class Ora {
        name: String
        cost: Integer
        weight: Double
    }
    class Suit {
        name: String
        cost: Integer
        weight: Double
    }
    class SuitDecorator {
        decoratedSuit: Suit
        cost: Integer
        weight: Double
    }
    class Flamethrower {
        cost: Integer
        weight: Double
        name: String
        cost: Integer
        weight: Double
    }
    class RocketLauncher {
        cost: Integer
        weight: Double
        name: String
        cost: Integer
        weight: Double
    }
    class AutoRifle {
        cost: Integer
        weight: Double
        name: String
        cost: Integer
        weight: Double
    }
    class Laser {
        cost: Integer
        weight: Double
        name: String
        cost: Integer
        weight: Double
    }
    class Controller {
        base_suit: ComboBox<String>
        flame_thrower: ComboBox<String>
        auto_rifle: ComboBox<String>
        rocket_launcher: ComboBox<String>
        laser: ComboBox<String>
        cost: Text
        weight: Text
        additions: List<ComboBox<String>>
        suitObjs: List<Suit>
        chosenSuit: Suit
        flame_thrower_count: Integer
        auto_rifle_count: Integer
        rocket_launcher_count: Integer
        laser_count: Integer
        initialize(): void
        initLists(): void
        initComboBoxes(): void
        showButton(ActionEvent): void
    }

    Dec ..> SuitDecorator : <<create>>
    Tor ..> SuitDecorator : <<create>>
    Ora ..> SuitDecorator : <<create>>
    SuitDecorator ..> Suit : <<create>>
    Flamethrower ..> SuitDecorator : <<create>>
    RocketLauncher ..> SuitDecorator : <<create>>
    AutoRifle ..> SuitDecorator : <<create>>
    Laser ..> SuitDecorator : <<create>>
    Controller ..> Dec : <<create>>
    Controller ..> Tor : <<create>>
    Controller ..> Ora : <<create>>
    Controller ..> SuitDecorator : <<create>>
    Controller ..> Flamethrower : <<create>>
    Controller ..> RocketLauncher : <<create>>
    Controller ..> AutoRifle : <<create>>
    Controller ..> Laser : <<create>>

```

The diagram illustrates the following classes and their relationships:

- Dec**: Attributes: name (String), cost (Integer), weight (Double).
- Tor**: Attributes: name (String), cost (Integer), weight (Double).
- Ora**: Attributes: name (String), cost (Integer), weight (Double).
- Suit**: Attributes: name (String), cost (Integer), weight (Double).
- SuitDecorator**: Attributes: decoratedSuit (Suit), cost (Integer), weight (Double). It has a composition relationship with **Suit**.
- Flamethrower**: Attributes: cost (Integer), weight (Double), name (String), cost (Integer), weight (Double).
- RocketLauncher**: Attributes: cost (Integer), weight (Double), name (String), cost (Integer), weight (Double).
- AutoRifle**: Attributes: cost (Integer), weight (Double), name (String), cost (Integer), weight (Double).
- Laser**: Attributes: cost (Integer), weight (Double), name (String), cost (Integer), weight (Double).
- Controller**: Attributes: base_suit (ComboBox<String>), flame_thrower (ComboBox<String>), auto_rifle (ComboBox<String>), rocket_launcher (ComboBox<String>), laser (ComboBox<String>), cost (Text), weight (Text), additions (List<ComboBox<String>>), suitObjs (List<Suit>), chosenSuit (Suit), flame_thrower_count (Integer), auto_rifle_count (Integer), rocket_launcher_count (Integer), laser_count (Integer). Methods: initialize(), initLists(), initComboBoxes(), showButton(ActionEvent).

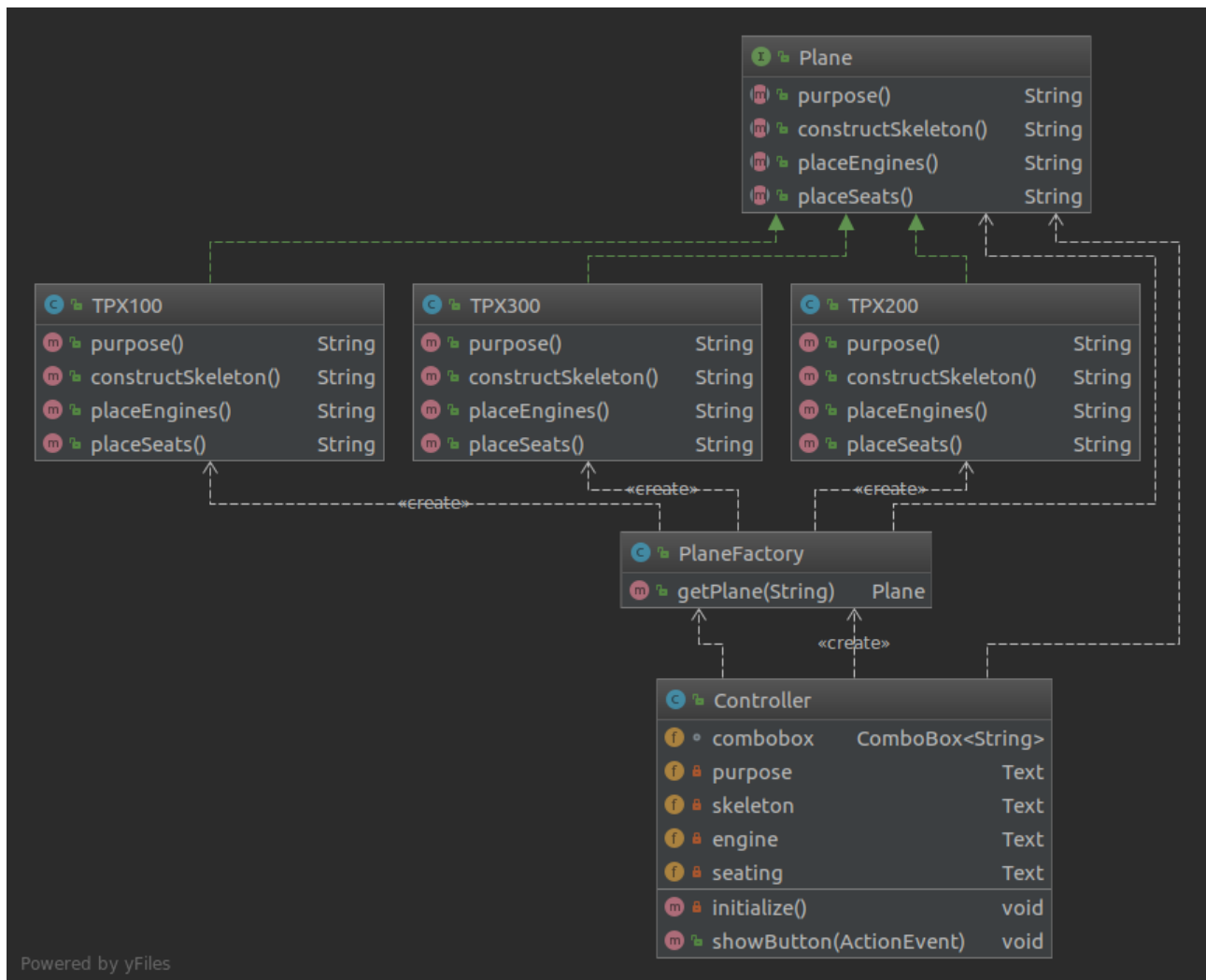
Relationships (Associations):

- Dec**, **Tor**, **Ora**, **SuitDecorator**, **Flamethrower**, **RocketLauncher**, **AutoRifle**, and **Laser** all have associations with **SuitDecorator** labeled with the role `<<create>>`.
- SuitDecorator** has a composition relationship with **Suit**.
- Controller** has associations with **Dec**, **Tor**, **Ora**, **SuitDecorator**, **Flamethrower**, **RocketLauncher**, **AutoRifle**, and **Laser**, all labeled with the role `<<create>>`.

In user interface, user can change suit type and number of decoration items. “Show” button is needed to be clicked to see the result.

Jar file is “HW2_Part2/out/artifacts/JavaFXApp/hw2part2.jar”

Question 3.1



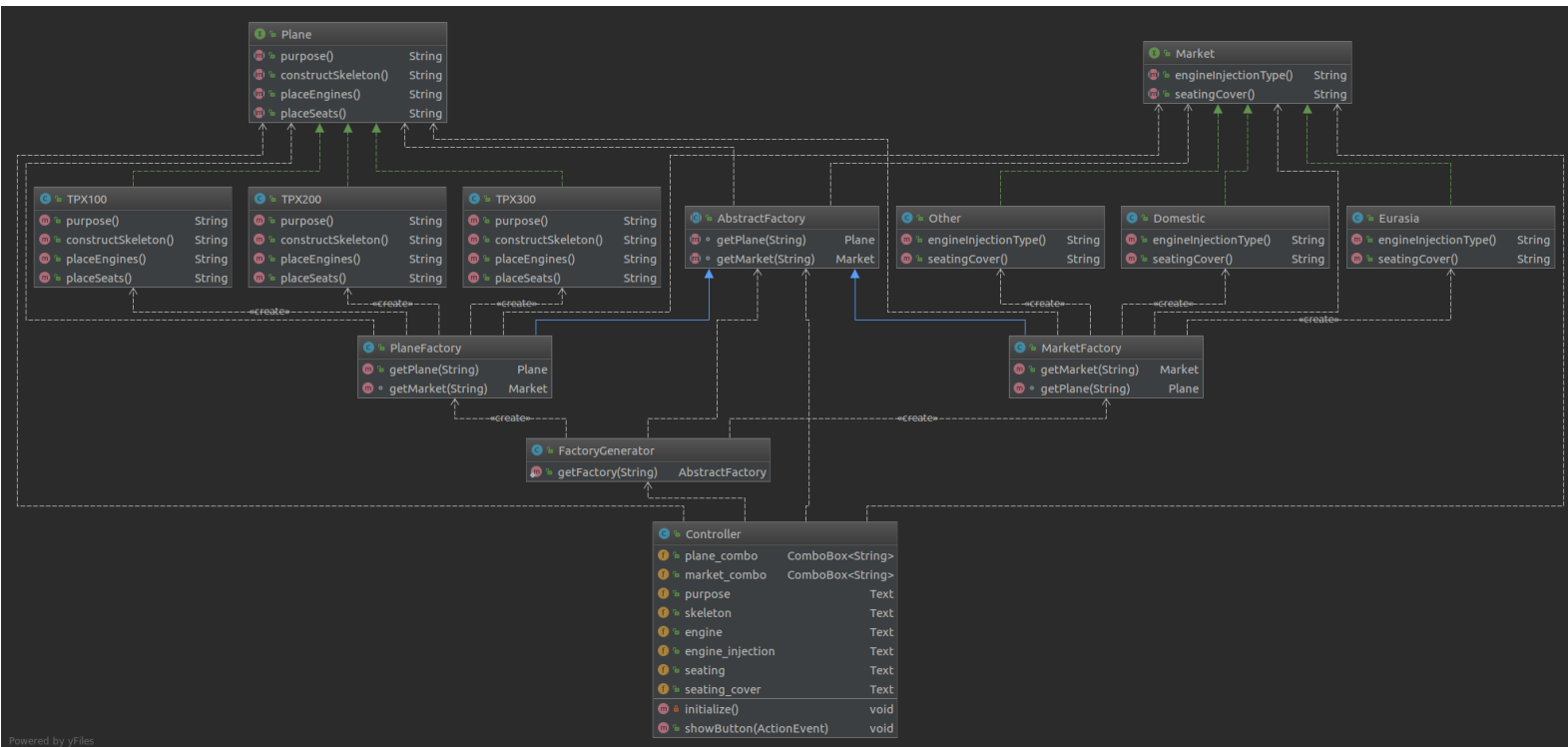
In this part, factory pattern is used. Logic of factory pattern is generating object according to given parameter. There is an interface called “Plane” and this interface is implemented by three classes that symbolize three type of planes. In “PlaneFactory” class, there is a method called “getPlane” and this method generates an object of one of three types of planes, according to its parameter.

In user interface, type of planes is chosen and “produce” button is clicked. It generates the construction phase of the chosen plane.

Jar file is

“HW2_Part3_1/out/artifacts/JavaFXApp/hw2part3_1.jar”

Question 3.2



In this part, abstract factory pattern is used. Difference between this part and the previous part is getting two different input from user. Since these inputs' types are different, there should be a class that generates a factory before getting an object. This is what "FactoryGenerator" class does. Its method called "getFactory" gets a parameter and generates either plane or market factory. After that, PlaneFactory and MarketFactory can generate their objects according to their string parameters. "AbstractFactory" class should be used to gather PlaneFactory and MarketFactory under a single roof.

In user interface, user choose plane type and market. After the button is clicked, the plane's production steps and market details are shown.

Jar file is HW2_Part3_2/out/artifacts/JavaFXApp/hw2part3_2.jar