

Eprog Bonus Survival Guide

Emil with help from Dario

November 20, 2024

Contents

1	Disclaimer	3
2	Factory Pattern	3
2.1	Introduction	3
2.2	Referencing the Factory	5
2.3	Application to the Bonus Exercise	6
3	Scheduling Actions	7
4	The Party assembles	9
5	We come to an End	12

1 Disclaimer

1. This should be an explanation on how to pass the test cases easily, design pattern might not be good
2. To make the examples short (and readable because I don't manage to install minted package smh) I will leave out "public", "private", ... keywords
3. There might be mistakes (in the code) or inconsistencies but I am just trying to explain some tips here with low effort so plz have mercy
4. Some explanations might be whacky and stuff, but you can look up game design patterns / tutorials and should find some interesting stuff involving lots of classes and inheritance and stuff for better explanations
5. Excuse the typos

2 Factory Pattern

We will very quickly explain what goes on in a factory class.

2.1 Introduction

Often in Java you will see a **factory class** which "produces" objects. Let's look at an example:

```
class CarFactory {  
    Car produceCar(int speed, String name) {  
        return new Car(speed, name);  
    }  
}
```

With the car class being:

```
class Car {  
    int speed;  
    String name;  
  
    Car(int speed, String name) {  
        this.speed = speed;  
        this.name = name;  
    }  
}
```

Okay this looks good, but kind of redundant? Why do we need this factory class when we can just create an object using the constructor? Well, let's say we want to give each car a serial number. Using our factory we can do that really easily:

```
class CarFactory {
    int amountProduced = 0;

    Car produceCar(int speed, String name) {
        int serialNumber = amountProduced++;
        return new Car(speed, name, serialNumber);
    }
}
```

Now each Car has a unique serial number (if we change the constructor accordingly).

Or let's say we know that our costumers are messy and will sometimes lose the reference to their car. Then it might be useful to keep references to all of the cars so we can access them using their serial number.

```
class CarFactory {
    int amountProduced = 0;
    Car[] allCars = new Car[100];

    Car produceCar(int speed, String name) {
        int serialNumber = amountProduced++;
        Car producedCar = new Car(speed, name, serialNumber);

        allCars[serialNumber] = producedCar;
        return producedCar;
    }

    Car findCar(serialNumber) {
        return allCars[serialNumber];
    }
}

//we assume that we only produce 100 cars here
//we could use an ArrayList for e.g. for more
```

Now we can use the `findCar()` method to access any car we produced using its serial number.

2.2 Referencing the Factory

We already saw that we can keep references to the produced objects in the factory class, which can be really useful. But what if we need to access the factory class from a produced object somehow? One might try to do something like this:

```
Factory myFactory = new Factory();
```

And then reference this factory in every produced Object:

```
class ProducedObject() {
    Factory origin = myFactory;
}
```

But solutions like this quickly break down, for example when we want to reuse our code for the factory. A better solution is to pass each created object a reference to the factory. We illustrate this using our CarFactory example:

```
class CarFactory {
    int amountProduced = 0;
    Car[] allCars = new Car[100];
    String factoryName = "CoolFactory"

    Car produceCar(int speed, String name) {
        int serialNumber = amountProduced++;
        Car producedCar = new Car(speed, name, serialNumber, this);

        allCars[serialNumber] = producedCar;
        return producedCar;
    }
}
```

```
//see how we pass a reference to the factory using the "this"-keyword
```

And our Car class looks like this:

```
class Car {
    int speed;
    int serialNumber;
    String name;

    // this is a reference to the factory that produced the car
    CarFactory origin;
```

```

    Car(int speed, String name, int serialNumber, CarFactory origin) {
        this.speed = speed;
        this.name = name;
        this.serialNumber = serialNumber;
        this.origin = origin;
    }

    void sayOrigin() {
        System.out.println("I am car number " + this.serialNumber);
        System.out.println("From the factory: " + this.origin.factoryName);
        System.out.println("Out of " + this.origin.amountProduced);
    }
}

```

Now we can access all the important metrics from each of the cars since they have access to their Factory through the reference. If now for e.g. the name of the factory changes, we don't have to make any changes to the cars since they will access the updated name directly from the factory.

2.3 Application to the Bonus Exercise

In the bonus exercise we had a **Game** class (the factory) and a **Human** class. The problem was that we wanted to store all the humans / actions in one central place (for e.g. in an array inside the **Game** class). However, we also needed access to this storage from our **Human** class for the `scheduleAction` method. An easy solution to this would be to pass a reference to the **Game** class to each human we create. This way every human has access to everything we store inside the **Game** class. This could look something like this:

```

class Game {
    Human[] allPlayers = new Human[100];
    List<Action> scheduledActions;
    int playerCount = 0;

    Human createHuman(int health, int position) {
        Human myHuman = new Human(health, position, this);
        this.allPlayers[playerCount++] = myHuman;
        return myHuman;
    }
}

/** You could implement this in many ways
// This is just for illustration

```

And now the human can access all the other humans to execute an action:

```
class Human {
    int health, position;
    Game origin;

    Human(int health, int position, Game origin) {
        this.health = health;
        this.position = position;
        this.origin = origin;
    }

    void scheduleAction()* {
        this.origin.scheduledActions.add(Action);
    }

    void hitPlayer(x)* {
        this.origin.allPlayers[x].doDamage();
    }

    /* Again the methods are just illustrations
    // An actual solution would need more logic
}
```

This eliminates the problem of how to pass information between the **Game** and the **Human** class. If a player now wants to do something to the other players, they can simply access the array with all the players through their reference to the **Game** class.

3 Scheduling Actions

Another problem was the scheduling of the actions the players can perform. We need to have something like a countdown for each scheduled action to know when we can execute it. But at the same time we need to keep track of which actions were scheduled first, so we can execute them in the correct order.

So our system for advancing one turn should look something like this:

1. What are the actions scheduled?
2. Which of these actions need to be executed this turn (reached countdown 0)
3. In what order were these actions scheduled?

The simplest structure we can use here is a queue. There are many ways to implement this, so I will keep the example general. We could either create a separate Action class or simply make it a field in the Human class. The latter could look like this:

```
class Human {
    Game origin;
    Action action;
    int countdown;

    void scheduleAction(Action action) {
        //set the countdown accordingly*
        this.action = action;
        origin.scheduledActions.add(this);
    }

    void doTurn() {
        if(this.countdown > 0) {
            this.countdown--;
        } else {
            //execute the action by using the reference to origin*
            doAction(origin.allPlayers);
        }
    }

    /* Of course I omitted a lot of checks and bookkeeping here
}
```

Now our Game class could look like this:

```
class Game {
    Human[] allPlayers = new Human[100];
    Queue scheduledActions;
    int playerCount = 0;

    Human createHuman(int health, int position) {
        Human myHuman = new Human(health, position, this);
        this.allPlayers[playerCount++] = myHuman;
        return myHuman;
    }

    void advanceTurn() {
        Queue uncompletedActions;
```



```

        for(Player p : scheduledActions) {
            p.doTurn();
            if(p.hasUncompletedAction) { //if they just reduced the countdown
                uncompletedActions.add(p);
            }
        }
        this.scheduledActions = uncompletedActions;
    }
}

```

The core idea here is that we have a Queue to keep track of which players scheduled their actions first. Everytime a turn advances we create a new Queue and then loop over all the players who have a scheduled action. If the action is executed we can remove them (don't add them to the new Queue), otherwise we need to add them to the new Queue since they are yet to be executed. At the end we simply make the new Queue our `scheduledActions`.

Of course you could also do this in **a lot** of other ways, but again the idea here is that we use the Queue stored centrally in the `Game` class to keep track of the scheduled actions and players. But to make things simpler for us we move the other logic into the `Human` class.

Not having a Queue could be the problem why the "Cleric"-testcase fails. If we take a look at the testcase we can see that the ordering matters here. If the position change from the cleric and the attack from the warrior are executed in the wrong order, the warrior will not hit the other one, resulting in a wrong test case.

4 The Party assembles

Okay okay cool cool now we can do stuff with our Humans and Games and we can access all the things we want from everywhere we need. But not every human is built the same. Our game should include: Jester (literally does nothing), Warrior (why are you hitting yourself?) and Clerics (wait they can have the same positions?). Of course we can think a lot about how to design this thing properly... Who should inherit from whom? Who should have access to what? Which level of game logic should be responsible for which actions?

But there are three simple ways we can solve the issue in the context of an eprog exercise:

1. **Throw everything into a giant if-statement:** Pro? fast Con? whacky
2. **Give every Human a "type" field.** Pro? fast Con? we will need some if-statements
3. **Make children-classes that inherit from Human.** Pro? easy Con? takes more time to write

I did the third version and will copy some of the code here. The big upside here is that you can focus on all the stuff separately for the different types of humans. As long as you get the inheritance right and have the three classes set up, you can go through the tasks and implement the actions one by one with full access to the player-array and without having to worry about anything else at the moment. Of course you will write some redundant code in each class and might copypaste some stuff, but it makes things (at least for an exercise like this) much simpler.

Oki, so what does each Human need? We know we will have (besides the constructor, which is the same for all classes and the given methods) a `scheduleAction` and a `doAction` method. So let's add those to our class and it will look something like this:

```
public class Human {

    public int health, position;
    public Action action;
    public int delay;
    public Game creator;

    public boolean scheduleAction(Action action) {
        return false;
    }

    public Human(int h, int p, Game c) {
        health = h;
        position = p;
        creator = c;
    }

    public void doAction(Human[] p) {
        return;
    }
}
```

Now we can simply make a new class that inherits from human for each of the three human-types and override the methods there. For my Warrior this looked something like that:

```
public class Warrior extends Human {
    Warrior(int h, int pos, Game c) {
        super(h, pos, c);
    }
}
```

```

public boolean scheduleAction(Action a) {
    if(!this.isAlive()) {
        return false;
    }
    if(this.action != null) {
        return false;
    }
    this.action = a;
    if(a == Action.SUMMON) {
        this.delay = 1;
    } else {
        this.delay = 0;
    }
    this.creator.queue.add(this);
    return true;
}

public void doAction(Human[] p) {
    if(this.action == null) return;
    if(!this.isAlive()) return;
    if(this.delay > 0) {
        this.delay--;
    } else {
        if(this.action == Action.ATTACK) {
            for(Human h : p) {
                if(h!=null && Math.abs(h.position - this.position) == 1) {
                    h.health -= 10;

                }
            }
        } else {
            this.health -= 5;
        }
        this.action = null;
    }
}

//mistakes might be made here but it passed the 5 test cases...
}

```

Note that in my version I passed the player-array into the `doAction` method. The `doAction`

method here is basically `doTurn` from the previous example.

Now you can do the same thing for the other classes.

5 We come to an End

Sorry for the hasty explanations, I hope it was kind of understandable and helped someone (:

Basically if you have problems like this it is helpful to first figure out who needs access to what and take care of that so you can then think about the actual algorithms without having to deal with visibility and stuff. And just breaking it down into separate subproblems and solving those separately might be inefficient time-wise, but makes the process much easier in my opinion.