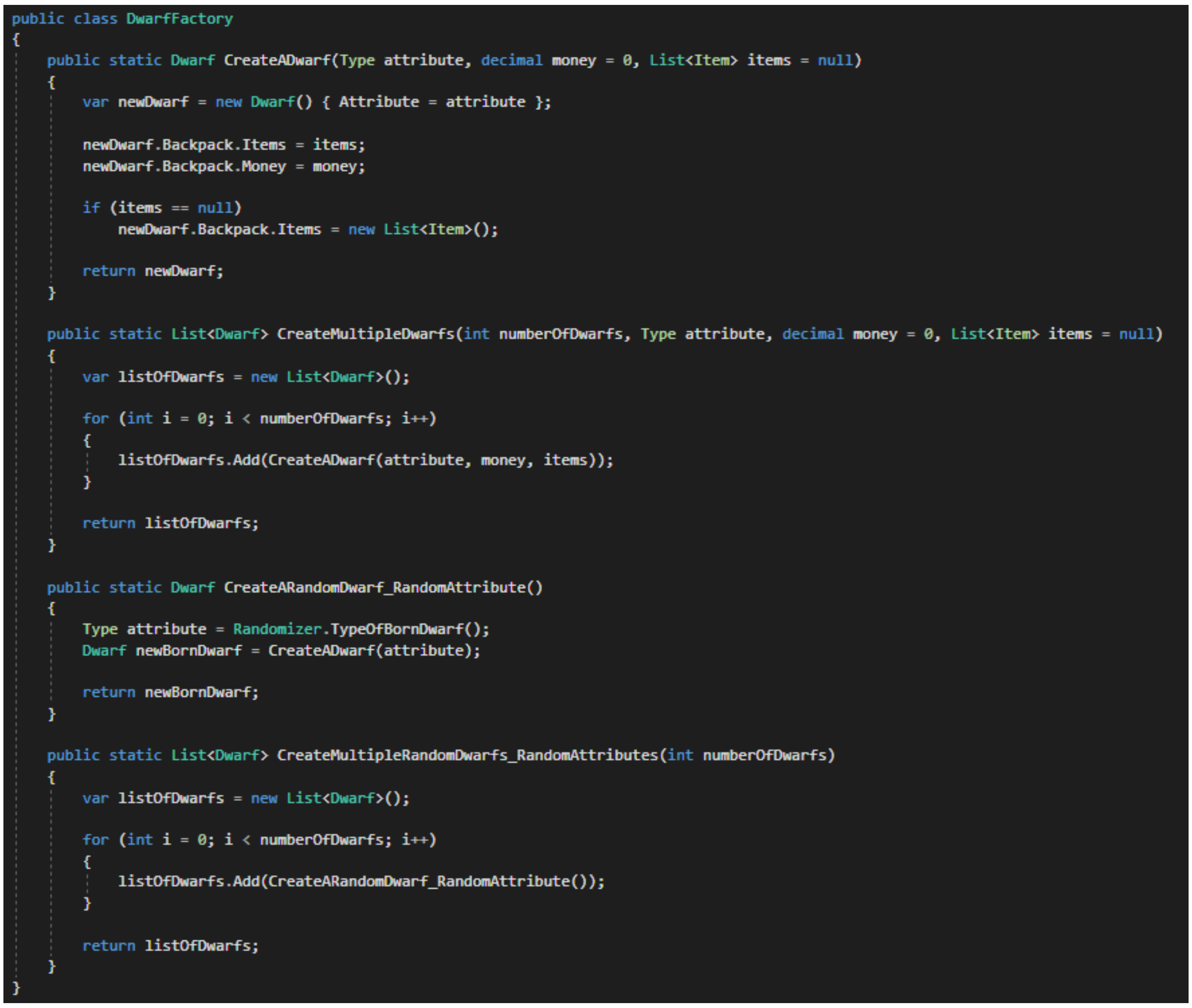
Mateusz Mazur

pieces of code

This document contains few figures and short descriptions presenting parts of my code that I have written during Corealate course.

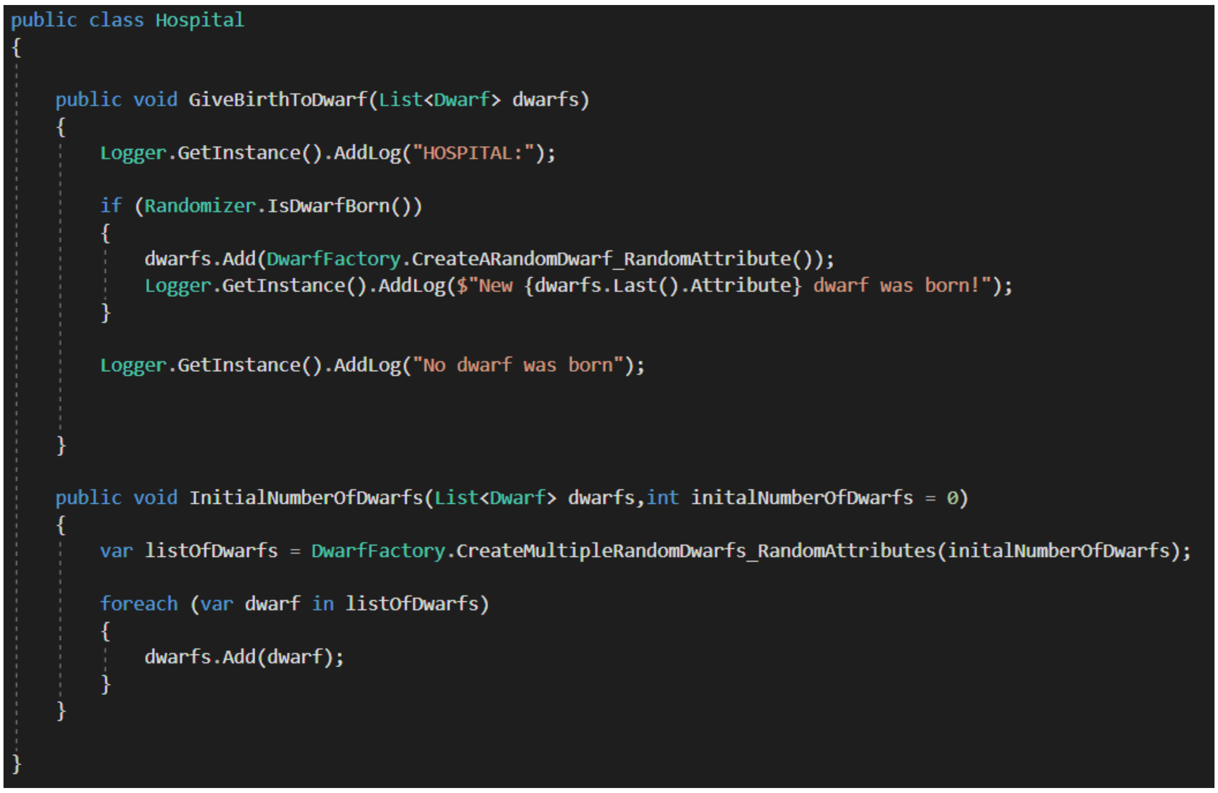
The code I would like to present concerns project of Dwarves' Mine Simulation. The goal we were supposed to achieve was daily and final report printed on the console. Reports had to show daily routine of dwarves. Report was created depending on the changes in few main classes such as Hospital, Mine, Shop etc.



```csharp
public class DwarfFactory
{
    public static Dwarf CreateADwarf(Type attribute, decimal money = 0, List<Item> items = null)
    {
        var newDwarf = new Dwarf() { Attribute = attribute };

        newDwarf.Backpack.Items = items;
        newDwarf.Backpack.Money = money;

        if (items == null)
            newDwarf.Backpack.Items = new List<Item>();

        return newDwarf;
    }

    public static List<Dwarf> CreateMultipleDwarfs(int numberOfDwarfs, Type attribute, decimal money = 0, List<Item> items = null)
    {
        var listOfDwarfs = new List<Dwarf>();

        for (int i = 0; i < numberOfDwarfs; i++)
        {
            listOfDwarfs.Add(CreateADwarf(attribute, money, items));
        }

        return listOfDwarfs;
    }

    public static Dwarf CreateARandomDwarf_RandomAttribute()
    {
        Type attribute = Randomizer.TypeOfBornDwarf();
        Dwarf newBornDwarf = CreateADwarf(attribute);

        return newBornDwarf;
    }

    public static List<Dwarf> CreateMultipleRandomDwarfs_RandomAttributes(int numberOfDwarfs)
    {
        var listOfDwarfs = new List<Dwarf>();

        for (int i = 0; i < numberOfDwarfs; i++)
        {
            listOfDwarfs.Add(CreateARandomDwarf_RandomAttribute());
        }

        return listOfDwarfs;
    }
}
```

Fig. 1. DwarfFactory class as a factory design pattern

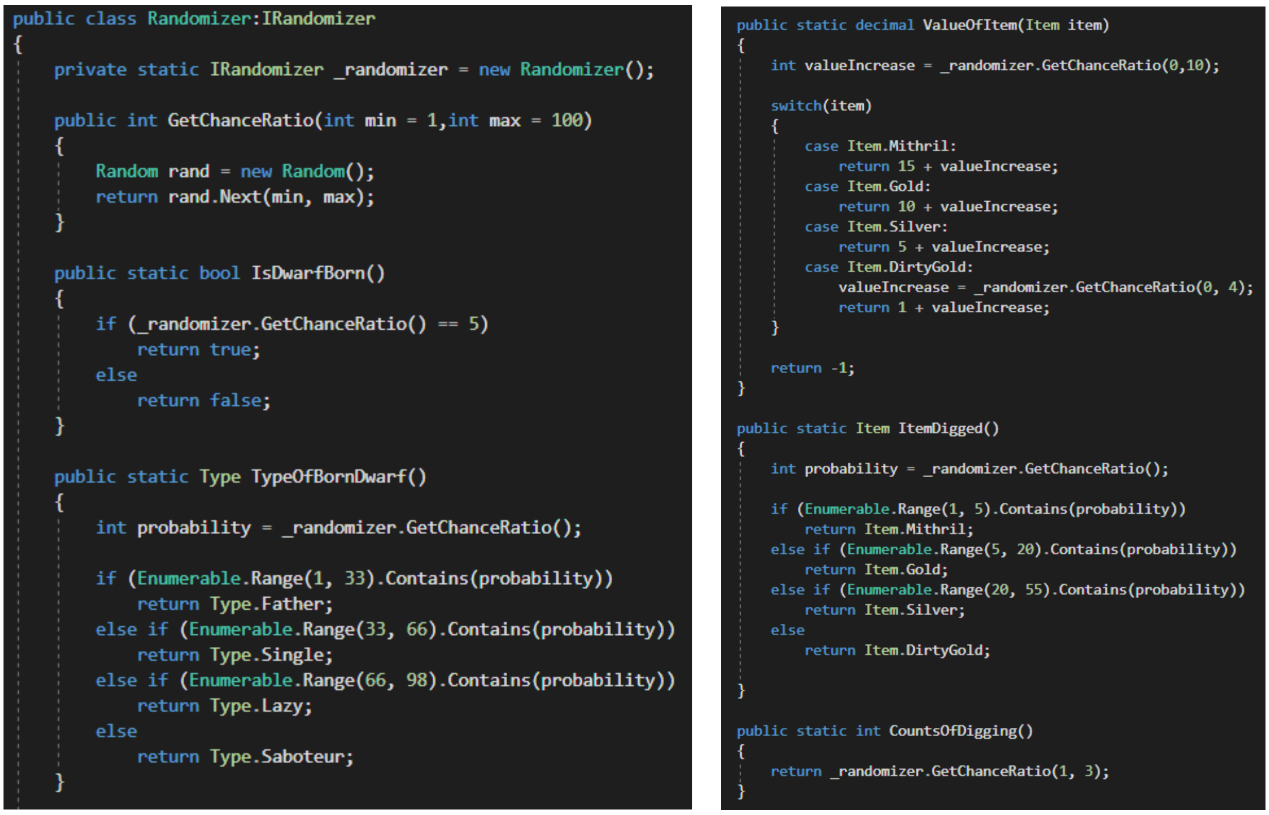
Code presented in figure no. 1 shows factory pattern. Its main purpose is to create new instances of Dwarf class. Methods implemented in the class are static not by an accident. It is due to the need to eliminate passing factory in methods. It was written to help rest of the team to write TDD. It was also used in Hospital class that is presented in fig. 2.



```
public class Hospital
{

    public void GiveBirthToDwarf(List<Dwarf> dwarfs)
    {
        Logger.GetInstance().AddLog("HOSPITAL:");

        if (Randomizer.IsDwarfBorn())
        {
            dwarfs.Add(DwarfFactory.CreateARandomDwarf_RandomAttribute());
            Logger.GetInstance().AddLog($"New {dwarfs.Last().Attribute} dwarf was born!");
        }

        Logger.GetInstance().AddLog("No dwarf was born");


    }

    public void InitialNumberOfDwarfs(List<Dwarf> dwarfs,int initalNumberOfDwarfs = 0)
    {
        var listOfDwarfs = DwarfFactory.CreateMultipleRandomDwarfs_RandomAttributes(initalNumberOfDwarfs);

        foreach (var dwarf in listOfDwarfs)
        {
            dwarfs.Add(dwarf);
        }
    }

}
```

Fig. 2. Hospital class

Hospital class is presented in figure no. 2 and as the name suggests the class's main purpose is to create dwarves with a given probability. This fragment of code shows that I was struggling to conserve first rule of S.O.L.I.D heuristic - single responsibility principle. In the beginning there was no Randomizer class and the probability was generated in Hospital. That is why I also created Randomizer class to help myself and in the course of time also rest of the team.



```
public class Randomizer:IRandomizer
{
    private static IRandomizer _randomizer = new Randomizer();

    public int GetChanceRatio(int min = 1,int max = 100)
    {
        Random rand = new Random();
        return rand.Next(min, max);
    }

    public static bool IsDwarfBorn()
    {
        if (_randomizer.GetChanceRatio() == 5)
            return true;
        else
            return false;
    }

    public static Type TypeOfBornDwarf()
    {
        int probability = _randomizer.GetChanceRatio();

        if (Enumerable.Range(1, 33).Contains(probability))
            return Type.Father;
        else if (Enumerable.Range(33, 66).Contains(probability))
            return Type.Single;
        else if (Enumerable.Range(66, 98).Contains(probability))
            return Type.Lazy;
        else
            return Type.Saboteur;
    }
```

```
public static decimal ValueOfItem(Item item)
{
    int valueIncrease = _randomizer.GetChanceRatio(0,10);

    switch(item)
    {
        case Item.Mithril:
            return 15 + valueIncrease;
        case Item.Gold:
            return 10 + valueIncrease;
        case Item.Silver:
            return 5 + valueIncrease;
        case Item.DirtyGold:
            valueIncrease = _randomizer.GetChanceRatio(0, 4);
            return 1 + valueIncrease;
    }

    return -1;
}

public static Item ItemDigged()
{
    int probability = _randomizer.GetChanceRatio();

    if (Enumerable.Range(1, 5).Contains(probability))
        return Item.Mithril;
    else if (Enumerable.Range(5, 20).Contains(probability))
        return Item.Gold;
    else if (Enumerable.Range(20, 55).Contains(probability))
        return Item.Silver;
    else
        return Item.DirtyGold;

}

public static int CountsOfDigging()
{
    return _randomizer.GetChanceRatio(1, 3);
}
```

Fig. 3. Randomizer class

Randomizer showed in figure no. 3 firstly had to return whether a dwarf was born or not. Then it started to grow as it turned out that it could be very useful for the rest of the team. It was responsible for generating probability dwarf's birth, type of born dwarf, digged item and its value and finally for returning how many times the dwarf can dig. It is the part of code that definitely needs improvement but there was not enough time for it. Now the class has too many responsibilities. In the future I would like to upgrade it and use Moq for testing. It is also the reason why I created the interface.