I would like to present this little piece of code where I found maybe not quite the right solution, but this still is working. In Bank class I create a static class that realize in application all financial operations during a simulation of dwarves' life like buying food, exchange minerals to gold etc. The problem was how to test this class, because it is static and so more it is a singleton pattern.

```
namespace DwarfLifeSimulationsTests.BankTests
{
    public class BankMock : Bank
    {
        public BankMock(Dictionary<int, BankAccount> Accounts, BankAccount GeneralAccount)
        {
            this.Accounts = Accounts;
            this.GeneralAccount = GeneralAccount;
        }
    }
```

So, I found that a can create I child class that inherits from Bank class and overload its constructor to have possibility to forward variables and then I can simulate each behavior of that class. It is simple, maybe not full correct solution but on the beginning of this project, when our team makes decision of how this class would be working, we did not predict that this may cause a problem in testing. When we finalized the project there was no time to such a serious change to remodel how this class can be implemented so, I have found this solution will be most correct.

Thanks to that I can test each critical method of Bank class and be sure it is working properly like for example test cases of correctly charged transaction by tax and whether the sum of collected tax is correct.

```
        [TestCase(150, 115.5)]
        [TestCase(200, 154)]
        [TestCase(250, 192.5)]
        [TestCase(300, 231)]
        [TestCase(350, 269.5)]
        [TestCase(400, 308)]
        [TestCase(450, 346.50)]
        [TestCase(500, 385)]
        [TestCase(550, 423.5)]
        [TestCase(600, 462)]
        public void BankCollectTaxes(decimal actual, decimal expected)
        {
            // given
            accounts = new Dictionary<int, BankAccount>();
            accounts.Add(1, new BankAccount());
            accounts[1].DailyIncome = actual;
            bankAccount = new BankAccount();
            bankMock = new BankMock(accounts, bankAccount);

            // when
            bankMock.PayTax(1);

            // then
            Assert.AreEqual(bankAccount.DailyIncome, actual * 0.23m);
            Assert.AreEqual(accounts[1].DailyIncome, expected);
        }
```

I am proud of this because, that confirms how I work when such as problem appears, I always looking for proper solutions and never give up.

It is also a lesson in the future, how much that simple decision on the beginning of the project can have implications and I will have it in my mind.