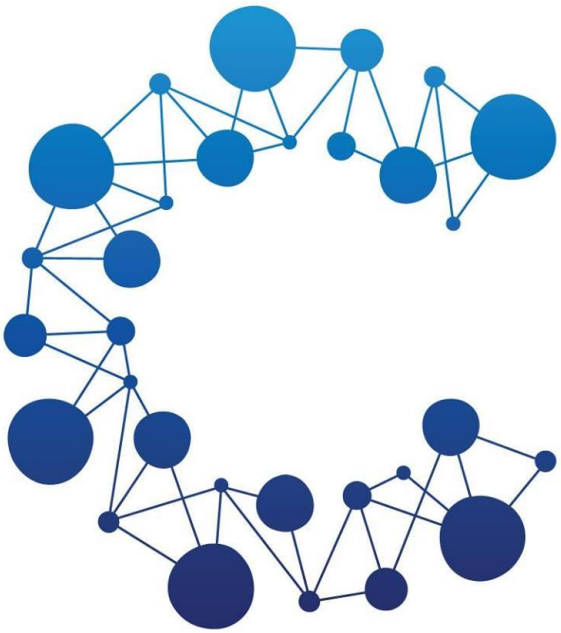
DUMNY Z POSTĘPU

Kod człowieka rozwijającego się



JAN KOCOŃ
KATOWICE – KURS CORELATE
20.02.2019

1. Słowem wstępu

Niniejszy dokument przedstawia moje umiejętności programistyczne wraz z wiedzą uzyskaną podczas kursu. Dokument składa się z podpunktów omawiających zagadnienia którymi chciałbym się podzielić. Przedstawione w nim zostały metodyki dobrego programowania takie jak KISS, DRY, YAGNI oraz SOLID, które w znacznym stopniu usprawniły moją prace. Przedstawione zostały również bliskie mi wzorce projektowe takie jak wzorzec strategii oraz wzorzec fabryki. Pokazałem również możliwe zastosowanie (anty) wzorca Singletona. Na ostatniej stronie zostało zamieszczone podsumowanie całego dokumentu oraz wypunktowane moje osiągnięcia.

2. Metodyki dobrego programowania

Na kursie poznałem wiele metod i praktyk dobrego programowania. Jedną z nich jest zasada **KISS** (Keep It Simple), która mówi o tym aby kod który piszemy był prosty – bez udziwnień. Prosty kod jest czytelny, przejrzysty, zrozumiały dla nas oraz dla ludzi z którymi pracujemy.

W projekcie będącym symulacją z życia krasnoludów mieliśmy instytucje banku, która przechowywała konta bankowe krasnali, sklepu oraz gildii krasnali. Założeniem do kont bankowych było to aby każde konto miało swój unikalny klucz ID. Pomysłów generowania unikalnych numerów było wiele – wszystkie niepotrzebnie skomplikowane i niezrozumiałe. W myśl zasady **KISS** utworzyłem statyczną klasę IDCreator (rys. 1), która zawiera metodę GetUniqeID() zwracającą unikalne ID. Metoda wywoływana jest przy tworzeniu kont bankowych i przy każdym wywołaniu zwraca wartość ID o jeden większą. Klasa została maksymalnie uproszczona co wpłynęło na jej zrozumienie oraz łatwość użycia przez członków zespołu.

```
public static class IDCreator
{
    private static int UniqueID = 0;
    public static int GetUniqueID()
    {
        UniqueID++;
        return UniqueID;
    }
}
```

Rysunek 1. Klasa IDCreator

Zasada **DRY** (Don't Repeat Yourself) mówi o tym aby nie powielać utworzonego kodu. Potrafię pisać „leniwie” – czyli tak aby wykorzystywać już istniejący kod. Poznałem wiele technik refaktoryzacji kodu w tym wydzielania powtarzającego się kodu do metod. Wpłynęło to na jakość, czystość oraz czytelność pisanego kodu. Jestem w stanie dużo łatwiej i szybciej wytwarzać kod jak również sprawniej dokonuje refaktoryzacji kodu już istniejącego.

Do niedawna pisałem kod na zapas nie będąc świadomym tego, że jest to tak na prawdę martwy kod. O pisaniu niepotrzebnego kodu dokładnie mówi zasada **YAGNI** (You Aren’t Gonna Need It). Będąc świadomym tego czynu **dokładnie zadaje pytania** klientom (trenerom) i robię **dokładnie to**

czego ode mnie oczekują bez dodawania kodu „na przyszłość” który być może się kiedyś przyda a być może nie. Kod na zapas zaśmieca projekt i pogarsza jego czytelność.

3. Wzorce projektowe

Wzorzec który najszybciej zrozumiałem i znalazłem dla niego setki zastosowań to wzorzec **strategia**. Umożliwia on rozwiązywanie tego samego problemu przy użyciu różnych strategii działania. Odbywa się to poprzez wydzielenie interfejsu (wspólnej abstrakcji) dla wszystkich strategii działania (rys.3). Obrazuje to przykład przedstawiony poniżej.

Klasa Shop umożliwia krasnalom dokonywanie zakupów. Posiada ona metodę PerformShoppingForAllDwarves, która przyjmuje na wejściu listę krasnali. Dla każdego krasnala wykonuje ona metodę PerformAction, która uruchamia odpowiednią strategię zakupu krasnala (rys. 3). Krasnal Father posiada strategię zakupu alkoholu za połowę jego pieniędzy (rys 4). Krasnal Single posiada strategię zakupu jedzenia za połowę jego wszystkich pieniędzy (rys 5). Krasnal Lazy nie kupuje nic w sklepie. W myśl zasad **SOLID**, a dokładniej zasady **Open - Close principle** mówiącej o tym by kod był zamknięty na modyfikacje a otwarty na rozszerzenia, zastosowałem wzorzec strategii. Strategia umożliwia wykonywanie jednej czynności na wiele sposobów (polimorfizm). Dzięki temu dodanie nowej strategii zakupów dla nowego typu krasnali nie spowoduje modyfikacji w już istniejącym kodzie a rozszerzy jego funkcjonalność.

```
public class Shop
{
    private BankAccount _bankAccount;
    public Shop()
    {
        _bankAccount = AccountCreator.CreateNewAccount(IDCreator.GetUniqueID());
    }
    public void PerformShopping(Dwarf dwarf)
    {
        dwarf.Buy(_bankAccount);
    }

    public void PerformShoppingForAllDwarves(List<Dwarf> dwarves)
    {
        foreach(Dwarf dwarf in dwarves)
        {
            PerformShopping(dwarf);
        }
    }
    public BankAccount GetBankAccount() => _bankAccount;

}
```

Rysunek 2. Klasa Shop

```
public interface IShoppingStrategy
{
    void Buy(BankAccount dwarfBankAccount,BankAccount bankAccountToTopUp);
}
```

Rysunek 3. Interfejs strategii zakupu

```
public class FatherShoppingStrategy : IShoppingStrategy
{
    public void Buy(BankAccount dwarfBankAccount, BankAccount bankAccountToTopUp)
    {
        decimal moneyForTransaction = dwarfBankAccount.CheckMoneyOnAccount() / 2;
        dwarfBankAccount.MakeTransaction(bankAccountToTopUp, moneyForTransaction);
        Logger.GetInstance().AddLog("Dwarf father bought alcohol");
    }
}
```

Rysunek 4. Strategia zakupu dla krasnala typu Father

```
public class LazyShoppingStrategy : IShoppingStrategy
{
    public void Buy(BankAccount dwarfBankAccount, BankAccount bankAccountToTopUp)
    {
        // Do nothing , Too lazy to buy
        Logger.GetInstance().AddLog("Dwarf lazy did not buy nothing");
    }
}
```

Rysunek 5. Strategia zakupu dla krasnala typu Lazy

```
public class SingleShoppingStrategy : IShoppingStrategy
{
    public void Buy(BankAccount dwarfBankAccount, BankAccount bankAccountToTopUp)
    {
        decimal moneyForTransaction = dwarfBankAccount.CheckMoneyOnAccount() / 2;
        dwarfBankAccount.MakeTransaction(bankAccountToTopUp, moneyForTransaction);
        Logger.GetInstance().AddLog("Dwarf single bought food");
    }
}
```

Rysunek 6. Strategia zakupu dla krasnala typu Single

Klasa Shop została przetestowana za pomocą **testów jednostkowych** przy użyciu narzędzia **NUnit**. Na rysunkach 7 oraz 8 zostały przedstawione wybrane testy. Sprawdzają one prawidłowe zmiany stanu konta bankowego sklepu dla klientów typu Father oraz Lazy. Testy umożliwiły mi sprawdzenie działania odpowiednich funkcji niezależnie od innych klas oraz niezależnie od poziomu ukończenia projektu jako całości. Zastosowanie wzorca projektowego strategia dla zakupów umożliwia testowanie poszczególnych strategii osobno. Zmniejsza się zależność w kodzie co ułatwia nie tylko rozszerzanie kodu ale jak i również jego testowanie.

```
[Test]
public void WhenDwarfFatherVisitsShopThenShopAccountIncreasesForHalfDwarfMoney()
{
    // Given
    Shop shop = new Shop();
    shop.GetBankAccount().TopUp(100);
    Dwarf dwarf = DwarfFactory.CreateDwarf(DwarfType.Father);
    dwarf.GetBankAccount().TopUp(200);
    decimal expectedDwarfMoney = 100;
    decimal expectedShopMoney = 200;

    // when
    shop.PerformShopping(dwarf);

    //then
    decimal actualDwarfMoney = dwarf.GetBankAccount().CheckMoneyOnAccount();
    decimal actualShopMoney = shop.GetBankAccount().CheckMoneyOnAccount();
    Assert.AreEqual(expectedDwarfMoney, actualDwarfMoney);
    Assert.AreEqual(expectedShopMoney, actualShopMoney);
}
```

Rysunek 7. Test jednostkowy klasy Shop sprawdzający prawidłowe działanie dla krasnali typu Father.

```
[Test]
public void WhenDwarfLazyVisitsShopThenShopAccountDoesntChange()
{
    // Given
    Shop shop = new Shop();
    shop.GetBankAccount().TopUp(100);
    Dwarf dwarf = DwarfFactory.CreateDwarf(DwarfType.Lazy);
    dwarf.GetBankAccount().TopUp(200);
    decimal expectedDwarfMoney = 200;
    decimal expectedShopMoney = 100;

    // when
    shop.PerformShopping(dwarf);

    //then
    decimal actualDwarfMoney = dwarf.GetBankAccount().CheckMoneyOnAccount();
    decimal actualShopMoney = shop.GetBankAccount().CheckMoneyOnAccount();
    Assert.AreEqual(expectedDwarfMoney, actualDwarfMoney);
    Assert.AreEqual(expectedShopMoney, actualShopMoney);
}
```

Rysunek 8. Test jednostkowy klasy Shop sprawdzający prawidłowe działanie dla krasnali typu Lazy.

Zaimplementowałem podstawową wersję obiektu logującego dane w programie w klasie Logger (rys.9). Użyłem do tego celu (anty) wzorca projektowego **Singleton**. Wzorzec ten daje sposobność posiadania dokładnie jednej instancji danej klasy w programie. Loger ma możliwość logowania danych z każdego miejsca w kodzie co bardzo usprawniło nam pracę przy tworzeniu raportu końcowego symulacji krasnali. Singleton często postrzegany jest jako antywzorzec - jest nie rozszerzalny oraz jest trudno testowalny. Przeczy również zasadom SOLIDa. Zdecydowałem sie na jego użycie świadomie oraz dobrowolnie po wcześniejszej konsultacji z osobami o większej wiedzy i doświadczeniu. Uważam, że w tej sytuacji spełnia

swoje zadanie. Aby poradzić sobie z przetestowaniem loggera musiałem utworzyć metodę ClearData, którą uruchamiam przed każdym testem związanym z logowaniem.

```
public sealed class Logger
{
    static private Logger logger = new Logger();
    private List<string> reports = new List<string>();

    private Logger() { }
    static public Logger GetInstance() => logger;
    public void AddLog(string log)
    {
        reports.Add(log);
    }
    public List<string> GetLogs() => reports;

    public void ClearData()
    {
        reports.Clear();
    }
}
```

Rysunek 9. Klasa Logger

Zaimplementowałem prostą fabrykę krasnali aby ujednolicić tworzenie krasnali oraz ułatwić testowanie klas zależnych od nich (rys 10).

```
public class DwarfFactory
{
    9 references | Jan, 3 days ago | 1 author, 2 changes
    public static Dwarf CreateDwarf(DwarfType dwarfType)
    {
        switch(dwarfType)
        {
            default:
            case DwarfType.Father:
                return new Dwarf(dwarfType, new FatherShoppingStrategy(), new StandardWorkingStrategy());
            case DwarfType.Lazy:
                return new Dwarf(dwarfType, new LazyShoppingStrategy(), new StandardWorkingStrategy());
            case DwarfType.Bomber:
                return new Dwarf(dwarfType, new BomberShoppingStrategy(), new BomberWorkingStrategy());
            case DwarfType.Single:
                return new Dwarf(dwarfType, new SingleShoppingStrategy(), new StandardWorkingStrategy());
        }
    }
}
```

Rysunek 10. Klasa DwarfFactory

Podsumowanie o mnie

- **Zadaje dużo pytań, nie boje się pytać**

- Znam metodyki dobrego programowania (DRY, KISS, YAGNI) oraz potrafię z nich korzystać

- Znam niektóre wzorce projektowe i „widzę je w kodzie”. Znam ich zalety i potrafię ich użyć.

- Poznaje chmury – zintegrowałem swoją aplikację mobilną tworzącą filmy na podstawie codziennie wykonywanych zdjęć z Dyskiem Google – dzięki czemu zdjęcia są w chmurze i zmiana urządzenia na nowszy model nie ma wpływu na ciągłość korzystania z aplikacji. Posiadam też aplikację działającą w chmurze heroku – projekt księgarni - https://fathomless-beach-66400.herokuapp.com/

- Znam algorytmy genetyczne – potrafię zaimplementować algorytm genetyczny czego dowodem są moje programy rozwiązujące problem komiwojażera oraz problem 8 hetmanów

- Znam podstawy SQL – miałem styczność z ETL oraz Hurtowniami Danych, potrafię napisać własnego CRUDa

- Znam podstawy GITa – radzę sobie z pullowaniem , commitowaniem, pushowaniem, mergowaniem oraz rozwiązywaniem konfliktów.

- Znam podstawy RESTa – umiem połączyć się z zewnętrznym API czego dowodem jest moja aplikacja do pogody korzystająca z danych z serwisu openweather.

- Szybko uczę się nowych języków programowania – na studiach poznałem C#, jave oraz pythona. Poznałem też F# oraz język deklaratywny – Prolog. Zmiana języka nie była dla mnie wyzwaniem.

- Posiadam certyfikat Cisco IT Essentials potwierdzający moje umiejętności techniczne oraz wiedzę na temat sprzętu komputerowego

- Posiadam certyfikat Kodołamacz programowanie w języku java