Wybrane zasady projektowania

Projektowanie obiektowe = określanie **odpowiedzialności** obiektów (klas) i ich relacji względem siebie. Wszystkie dobre praktyki, zasady, wzorce sprowadzają się do tego jak właściwie rozdzielić odpowiedzialność na zbiór obiektów (klas).

Skrajności są oczywiste:
system z **jedną olbrzymią klasą**, w której są wszystkie metody
vs
system z olbrzymią liczbą klas, z których **każda ma jedną metodę**
Projektując architekturę obiektową poruszamy się między tymi skrajnościami

GRASP – bardzo ogólne „dobre praktyki" rozdzielania odpowiedzialności
SOLID – zestaw 5 zasad, których nie należy łamać

Można także wyróżnić:

DRY - Don't Repeat Yourself

- Jedna z podstawowych zasad programowania - nie powtarzaj się. Wielokrotne użycie tego samego kodu to podstawa programowania.
- Jeśli jesteś blisko powtórzenia (np. chcesz zastosować kopiuj/wklej, seria ifów lub w kodzie występują podobne zachowania) pomyśl nad stworzeniem abstrakcji (pętla, wspólny interfejs, funkcja, klasa, jakiś wzorzec projektowy np. Strategia itp.), którą będziesz mógł wielokrotnie wykorzystać.

Plusy

- Lepsza czytelność kodu oraz łatwość w utrzymaniu.
- Zmiana implementacji tylko w jednym miejscu.

KISS - Keep it simple, stupid!

- Prostota (i unikanie złożoności) powinna być priorytetem podczas programowania. Kod powinien być łatwy do odczytania i zrozumienia wymagając do tego jak najmniej wysiłku.
- Większość systemów działa najlepiej, gdy polegają na prostocie, a nie na złożoności.
- Staraj się, aby twój kod podczas analizy nie zmuszał do zbytniego myślenia.
- Gdy po jakimś czasie wracasz do swojego kodu i nie wiesz co tam się dzieje, to znak, że musisz nad tym popracować

Zależności miedzy pakietami / klasami

Minimise Coupling

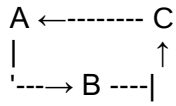
- Minimalizuj powiązania między pakietami/klasami. Mniejsza ilość zależności jest lepsza.
- Można to zrobić przez ukrywanie szczegółów implementacji czy stosowanie Law of Demeter

Plusy

- Zmniejsza to szansę na to, że zmiana kodu w zależności A nie popsuje kawałka kodu B.

ADP - Acyclic dependencies principle

- Zasada ta mówi, że w strukturze zależności nie powinno być zapętleń/cykli.
- Przykładowa struktura łamiąca tą zasadę będzie gdy: pakiet A ma zależność w pakiecie B, który ma zależność w pakiecie C, który z kolei ma zależność w pakiecie A.



- Możemy temu zapobiec stosując Dependency inversion principle, wzorce projektowe np. Observer lub stworzyć nowy pakiet i wrzucić tam wszystkie wspólne zależności.

Plusy

- łatwiej zrozumieć i utrzymać system

SDP - Stable-dependencies principle

- Pakiet, który jest zmienny nie powinien zależeć od pakietu, który jest trudny do zmiany.
- Zależności tego pakietu powinny być bardziej stabilne niż on sam.
- Taką samą zasadę możemy stosować w przypadku klas.

Plusy

- Zmniejsza efekt łancuchowej reakcji, która wymusza modyfikację kodu w kilku innych miejscach po wprowadzeniu jednej zmiany

SAP - Stable-abstractions principle

- Stabilny pakiet (trudny do zmiany) powinien być maksymalnie abstrakcyjny (czyli np. w większości operować na interfejsach), aby jego stabilność nie uniemożliwiała jego rozszerzania.
- Niestabilny pakiet (łatwy do zmiany) powinien być maksymalnie konkretny (czyli w większości operować na implementacjach) jako że jego niestabilność pozwala w łatwy sposób zmienić jego konkretną implementację.

Powyżej przedstawiono fragment z wielu z zasad tworzenia oprogramowania. Warto jednak szczególne przyjrzeć się zbiorowi zasad i wytycznych, które są ujęte we wspomnianych już SOLID i GRASP.

SOLID:

1. **Single responsibility**

Zasada pojedynczej odpowiedzialności mówi o tym, aby każda klasa była odpowiedzialna za jedną konkretną rzecz. W szczególności powinien istnieć jeden konkretny powód do

modyfikacji danej klasy. Stosowanie tej zasady znacząco zwiększa ilość klas w programie, a jednocześnie zmniejsza ilość klas typu scyzoryk szwajcarski. Takim mianem określa się wielkie kilkuset linijkowe klasy, skupiające za dużo funkcjonalności

```
3
4  //przykład łamiący zasadę pojedynczej odpowiedzialności
5  class Address
6  {
7      public Country CountryId { get; set; }
8      public string Voivodeship { get; set; }
9      public string County { get; set; }
10     public string Community { get; set; }
11     public string Place { get; set; }
12     public string Street { get; set; }
13     public int HouseNumber { get; set; }
14     public string ApartmentNumber { get; set; }
15     public string PostCode { get; set; }
16
17     public Address(Country _countryId, string _place, string _street, int
18 _houseNumer, string
19     _postCode)
20     {
21         CountryId = _countryId;
22         Place = _place;
23         Street = _street;
24         HouseNumber = _houseNumer;
25         PostCode = ValidatePostCode(_postCode);
26     }
27
28     private string ValidatePostCode(string postCode)
29     {
30        if (!Regex.IsMatch(postCode, @"^\d{2}-\d{3}$"))
31             throw new Exception("Pod pocztowy nie jest poprawny");
1
2          return postCode;
3      }
4  }
5
6  //przykład z uwzględnieniem zasady pojedynczej odpowiedzialności
7  class Address
8  {
9      public Country CountryId { get; set; }
10     public string Voivodeship { get; set; }
11     public string County { get; set; }
12     public string Community { get; set; }
13     public string Place { get; set; }
14     public string Street { get; set; }
15     public int HouseNumber { get; set; }
16     public string ApartmentNumber { get; set; }
17     public string PostCode { get; set; }

18     public Address(Country _countryId, string _place, string _street, int
19 _houseNumer, string
20     _postCode)
21     {
22         CountryId = _countryId;
23         Place = _place;
24         Street = _street;
25         HouseNumber = _houseNumer;
```

```
        PostCode = _postCode;
    }
}

class AddressValidator
{
    public string ValidatePostCode(string postCode)
    {
       if (!Regex.IsMatch(postCode, @"^\d{2}-\d{3}$"))
            throw new Exception("Pod pocztowy nie jest poprawny");

        return postCode;
    }
}
```

Pierwsza zasada pozwala nam uniknąć powielania kodu, ale z drugiej strony może nas zapędzić w błędne koło. Spędzimy wtedy długie godziny na refaktoryzacji i rozbijaniu kodu na coraz bardziej atomowe elementy. Uważam, że prędzej czy później i tak to będzie potrzebne, ale niestety idealny świat nie istnieje, więc i nasz kod nie będzie nigdy perfekcyjny. Dlatego, gdy następnym razem zaczniesz tworzyć nową klasę, to miej proszę z tyłu głowy informację o tym, by była ona jak najbardziej atomowa, ale nie przesadzaj z tym!

2. Open/closed

Zasada otwarty/zamknięty powinna być zawsze rozwijana do postaci “otwarty na rozbudowę, zamknięty na modyfikacje”. Dzięki temu, jest to praktycznie jej cała i kompletna definicja. Jest to bardzo ważna zasada, szczególnie w dużych projektach, nad którymi pracuje wielu programistów.

Każdą klasę powinniśmy pisać tak, aby możliwe było dodawanie nowych funkcjonalności, bez konieczności jej modyfikacji. Modyfikacja jest surowo zabroniona, ponieważ zmiana deklaracji jakiejkolwiek metody może spowodować awarię systemu w innym miejscu. Zasada ta jest szczególnie ważna dla twórców wszelkich wtyczek i bibliotek programistycznych

Istnieje pewna zależność, im bardziej trzymamy się **zasady pojedynczej odpowiedzialności**, tym bardziej musimy dbać o **zasadę otwarty na rozbudowę, zamknięty na modyfikacje**.

```
//przykład łamiący zasadę otwarte zamknięte
public class CreateDocument
{
    public void CreateDoc(string type, List<Product> products)
    {
        switch (type)
        {
            case "invoice":
                CreateInvoice(products);
                break;
            case "offer":
```

```
                CreateOffer(products);
                break;
            case "order":
                CreateOrder(products);
                break;
            default:
                break;
        }
    }

    private void CreateInvoiceDoc(List<Product> products)
    {
        // tresc metody
    }

    private void CreateOfferDoc(List<Product> products)
    {
        // tresc metody
    }

    private void CreateOrderDoc(List<Product> products)
    {
        // tresc metody
    }
}
```

```
//przykład z uwzględnieniem zasady otwarte zamknięte
public abstract class CreateDocument
{
    public abstract void CreateDoc(List<Product> products)  { }
}

public class CreateInvoiceDocument : CreateDocument
{
    public override CreateDoc(List<Product> products)
    {
       // tresc metody w klasie dotyczącej faktur
    }
}

public class CreateOfferDocument : CreateDocument
{
    public override CreateDoc(List<Product> products)
    {
       // tresc metody w klasie dotyczącej ofert
    }
}

public class CreateOrderDocument : CreateDocument
{
    public override CreateDoc(List<Product> products)
    {
       // tresc metody w klasie dotyczącej zapytań
    }
```

```
}

public class CreateInvoiceProformDocument : CreateDocument
{
    public override CreateDoc(List<Product> products)
    {
       // tresc metody w klasie dotyczącej faktur proform
    }
}
```

Powyższy kod bez dwóch zdań wymaga refaktoryzacji. Pomysł na to jest następujący. Należy utworzyć klasę abstrakcyjną CreateDocument, będzie ona zawierać metodę publiczną CreateDoc. Następnie klasy CreateInvoiceDocument, CreateOfferDocument, CreateOrderDocument, które będą dziedziczyć po klasie abstrakcyjnej i w każdej z nich dodam metodę CreateDoc, które to metody będą zawierać różne ciała metod zgodnie z tym, czego wymaga biznes w każdym z przypadków.

Jak widać, rozwiązuje nam to problemy z rozbudową projektu o nowe typy dokumentów — chodzi o fakturę proforma. Teraz wystarczy dodać nową klasę, która będzie dziedziczyć z klasy CreateDocument, zaimplementować metodę tworzenia dokumentu i cieszyć się z zachowania Open Close Principle.

Użycie polimorfizmu

Polimorfizm jest jednym z fundamentów programowania obiektowego. Jest także podstawowym mechanizmem, który powoduje, że nasza klasa będzie możliwa na rozbudowę w przyszłości.

Rozważmy przykład:

```
class Square
{
    public int A { get; set; }
}

class Rectangle
{
    public int A { get; set; }
    public int B { get; set; }
}

class Calculator
{
    public int Area(object shape)
    {
        if (shape is Square)
        {
            Square square = (Square)shape;
```

```
            return square.A * square.A;
        }
        else if (shape is Rectangle)
        {
            Rectangle rectangle = (Rectangle)shape;
            return rectangle.A * rectangle.B;
        }

        return 0;
    }
}
```

W powyższym przykładzie dodanie jakiejkolwiek nowej figury, wiąże się z koniecznością modyfikacji istniejącej klasy. Jest to ewidentne złamanie zasady otwarty/zamknięty, ponieważ klasa nie jest otwarta na rozbudowę.

Dzięki użyciu polimorfizmu można obarczyć implementacją metody liczącej pole figury każdą klasę reprezentującą figurę. Kod będzie dodatkowo o wiele prostszy.

Rozważmy przykład:

```
abstract class Shape
{
    public abstract int Area();
}

class Square : Shape
{
    public int A { get; set; }

    public override int Area()
    {
        return A * A;
    }
}

class Rectangle : Shape
{
    public int A { get; set; }
    public int B { get; set; }

    public override int Area()
    {
        return A * B;
    }
}

class Calculator
{
    public int Area(Shape shape)
    {
        return shape.Area();
    }
}
```

Czy powyższy przykład przekonał Cię do konieczności trzymania się zasady otwarty/zamknięty? Być może nie. Aby rozumieć konieczność używania tej zasady, pomyśl o klasie *Calculator* jako klasie znajdującej się w osobnym pliku DLL, który jest udostępniony tysiącom klientów. Drobna poprawka w kodzie zmusza tysiące programistów do pobrania

nowej wersji pliku DLL z nowszą wersją metody liczenia pola. Dlatego właśnie **klasa powinna być otwarta na modyfikacje bez konieczności jej edycji**.

3. Liskov substitution

Zasada podstawienia Liskov uważana jest za zadadę, którą najciężej zrozumieć, a ludzie bardzo często mylą ją z wszelkimi innymi zasadami. Jej nazwa pochodzi od nazwiska amerykańskiej programistki Barbary Liskov. W skrócie zasada polega na tym, że w miejscu klasy bazowej można zawsze użyć dowolnej klasy pochodnej. Oznacza to, że w całości musi być zachowana zgodność interfejsu i wszystkich metod.

Zasada podstawienia Liskov najczęściej łamana jest w przypadkach:

- kiedy programista źle rozplanował mechanizm dziedziczenia, interfejs polimorficzny jest zbyt ogólny
- zastosowane dziedziczenie bez mechanizmu polimorfizmu (mało efektywne i często prowadzi do złamania Liskov)
- klasy pochodne nadpisują metody klasy bazowej zastępując jej niepasującą logikę

4. Interface segregation

Zasada segregacji interfejsów jest kolejną z prostych zasad SOLID. Jak sama nazwa mówi, robi pewien porządek w interfejsach oraz klasach abstrakcyjnych. Trochę przypomina zasadę numer trzy. W zasadzie podstawienia Liskov głównie chodzi o to, by oczywiście klasy bazowe były odpowiednio małe, ale również o to, by metod w środku klas dziedziczących używać intuicyjnie. W aktualnie omawianej zasadzie natomiast chodzi o maksymalną atomowość, czyli uproszczenie interfejsu lub klasy bazowej.

Ten przypadek też możemy sobie wyobrazić. Można sobie wyobrazić interface **Account**, który jest przerośnięty. Zawierał definicję kilkudziesięciu metod, od utworzenia, edycji, usuwania po walidacje i nadawanie ról czy uprawnień. Gigant taki był następnie w paru miejscach wykorzystywany jako element, po którym jedna czy druga klasa dziedziczyła, tworząc problemy z metodami, które w danym fragmencie kodu były zbędne.

Przeanalizuj proszę przypadek poniżej.

```
1  //przykład łamiący zasadę segregacji interfejsów
2  public interface IAccount
3  {
4      public void CreateUser() { }
5      public void EditUser() { }
6      public void DeleteUser() { }
7      public void ValidateLoginUser() { }
8  }
9
10 public class Register : IAccount
11 {
12     public void CreateUser()
13     {
14         // tresc metody
15     }
16
```

```
15      public void EditUser()
16      {
17            throw new NotImplementedException();
18      }
19
20      public void DeleteUser()
21      {
22            throw new NotImplementedException();
23      }
24
25      public bool ValidateLoginUser()
26      {
27            // tresc metody
28      }
29}
30
31public class UserProfile : IAccount
32{
33    public void CreateUser()
34    {
35         throw new NotImplementedException();
36    }
37
38     public void EditUser()
39     {
40           // tresc metody
41     }
42
43     public void DeleteUser()
44     {
45           throw new NotImplementedException();
46     }
47
48     public bool ValidateLoginUser()
49     {
50           throw new NotImplementedException();
51     }
52}
53
54
1 //przykład z uwzględnieniem zasady segregacji interfejsów
2 public interface ICreateU
3 {
4     public void CreateUser() { }
5 }
6
7 public interface IEditU
8 {
9         public void EditUser() { }
10 }
11
12public interface IDeleteU
```

```
{
    public void DeleteUser() { }
}

public interface IValidateU
{
    public void ValidateLoginUser() { }
}

public class Register : ICreateU, IValidateU
{
    public void CreateUser()
    {
        // tresc metody
    }

    public bool ValidateLoginUser()
    {
        // tresc metody
    }
}

public class UserProfile : IEditU
{
    public void EditUser()
    {
        // tresc metody
    }
}
```

Powyżej bardzo dobrze widać, jak dużo zbędnego kodu w pierwszym przykładzie nam się wytworzyło. Mamy jeden interfejs **IAccount**, który zawiera cztery metody — **CreateUser**, **EditUser**, **DeleteUser** oraz **ValidateLoginUser**. Klasy **Register** i **UserProfile** dziedziczą z interfejsu wszystkie metody, mimo że nie używamy niektórych z nich. Wtedy musimy wrzucić **Exception** w ciało metody.

Inaczej ma się sytuacja w drugim przypadku. Zrobiłem niewielką refaktoryzację, duży interfejs **IAccount** zamieniłem na cztery mniejsze **ICreateU**, **IEditU**, **IDeleteU** oraz **IValidateU**. Teraz w klasach **Register** i **UserProfile** dodałem dziedziczenie odpowiednio do potrzeby. Jak wiadomo, klasy mogą dziedziczyć po wielu interfejsach, więc skorzystałem z tej możliwości i w przypadku rejestracji dziedziczę po **ICreateU** oraz **IValidateU** a w przypadku **UserProfile** tylko po **IEditU**.

5. Dependency inversion

Moduły aplikacji takie jak klasy nie powinny być zależne. Zależność w tym przypadku rozumiemy jako na przykład dziedziczenie. Klasa więc powinna dziedziczyć tylko po obiektach abstrakcyjnych takich jak klasy abstrakcyjne czy interfejsy. Analizując głębiej piątą zasadę, musimy dopowiedzieć, że moduły wysokopoziomowe nie powinny zależeć od tych niższego

poziomu. Chodzi o to, by klasy, które zawierają logikę biznesową, nie były zależne od klas, które nie odgrywają kluczowej roli.

SOLID w JavaScript:

- https://dev.to/denisveleaev/5-solid-principles-with-javascript-how-to-make-your-code-solid-1kl5
- https://zniszcz.github.io/solid-js/

GRASP

Podczas gdy zasady SOLID dotyczą szerszego aspektu programowania, GRASP (General Responsibility Assignment Software Principles) opisuje reguły dotyczące przede wszystkim programowania obiektowego OOP (Object-Oriented Programming). Jest to zbiór 9. zasad mówiących o zasadach przydziału odpowiedzialności do klas.

- **Information Expert (Ekspert)** – główna zasada programowania obiektowego, która mówi że odpowiedzialność powinna być przydzielona klasie, która posiada informacje niezbędne do wykonania zadania, tzn. takiej która jest najlepiej do tego zadania przygotowana.
- **Creator (Twórca)** – mówi o tym, kto powinien tworzyć obiekt. Klasa B powinna być odpowiedzialna za tworzenie instancji klasy A wtedy gdy: zawiera lub agreguje klasę A, posiada dane inicjujące klasę A, zapamiętuje klasę A lub intensywnie z niej korzysta.
- Controller (Kontroler) – reguła mówiąca o tym, który obiekt jako pierwszy przyjmuje i koordynuje żądaniami. Odpowiedzialność tą powinien otrzymać obiekt, który reprezentuje system, podsystem lub scenariusz przypadku użycia.
- **Low Coupling (Niskie Sprzężenie)** – zasada mówiąca o przydziale odpowiedzialności obiektom, w taki sposób aby zależności (sprzężenia) pomiędzy nimi były jak najmniejsze. Stosowanie tej zasady jest niezwykle ważne ponieważ pozwala zmniejszyć zasięg oddziaływania zmian w aplikacji, a co za tym idzie zmniejszenie ryzyka powstawania błędów. Pozwala również na lepsze zrozumienie kodu.
- **High Cohesion (Wysoka Spójność)** – podkreśla to, aby klasy były skoncentrowane na jednym, ściśle określonym zadaniu. Eliminuje to tworzenie nadmiernie rozbudowanych klas, pozwala na lepsze zrozumienie i utrzymanie kodu oraz zmniejsza zależności pomiędzy klasami. Operacje oferowane przez obiekty powinny tworzyć spójną całość.
- **Polymorphism (Polimorfizm)** – jedna z najważniejszych cech OOP, mówiąca o różnicowaniu zachowań operacji w klasach. Różnicowanie to powinno następować dzięki polimorficznym wywołaniom operacji w klasach pochodnych.
- **Pure Fabrication (Czysty Wymysł) –** opisuje sytuacje, gdy nie jesteśmy w stanie przypisać odpowiedzialności do żadnej z klas opisujących domenę systemu. W takiej sytuacji należy przypisać ten zakres odpowiedzialności sztucznej, pomocniczej klasie, która nie reprezentuje konceptu z dziedziny problemu. Taka klasa nazywana jest serwisem lub usługą.
- **Indirection (Pośrednictwo)** – zasada mówiąca o unikaniu bezpośredniego powiązania (zależności). W takiej sytuacji należy przydzielić zobowiązanie obiektowi pośredniczącemu, który będzie przekazywał informacje pomiędzy poszczególnymi komponentami lub usługami. Pozwala to uniknąć nadmiernych, bezpośrednich sprzężeń.
- **Protected Variations (Ochrona Zmienności)** – reguła podkreślająca konieczność ochrony przed zmiennymi lub niestabilnymi obiektami, podsystemami lub systemami. Należy zidentyfikować miejsca przewidywanej zmienności lub niestabilności oraz przydzielić zobowiązania w taki sposób, aby wokół tych miejsc powstały stabilne interfejsy.

GRASP z przykładami w języku Java - https://devcezz.pl/2021/12/09/grasp-garsc-zasad-wytwarzania-oprogramowania/

GRASP z przykładami w języku PHP - https://patrykwozinski.medium.com/grasp-dd7ac5384371

Źródła:
https://devcave.pl/notatnik-juniora/zasady-projektowania-kodu
https://mateuszrus.pl/solid-zasady-programowania/
https://nofluffjobs.com/blog/projektujac-nasz-kod-powszedni/
https://www.p-programowanie.pl/paradygmaty-programowania/zasady-solid
http://www.blog.molitorys.pl/wzorce-projektowe-czyli-zasady-dobrego-programowania