Zawartość:

- zasady SOLID

- wstrzykiwanie zależności

- wzorce:

    * metoda szablonowa

    * budowniczy

    * adapter

    * singleton

    * odwiedzający

    * dekorator

    * fasada

    * kompozyt

    * obserwator

Wzorce projektowe

Nazwa brzmi groźnie, ale tak naprawdę tak nie jest. Wzorzec projektowy ma za zadanie pokazać w jaki sposób projektować klasy oraz relacje między nimi. Są też wzorce projektowe, mówiące o tym czego nie robić w programie.

Podział:

- wzorce obiektowe

- wzorce klasowe

Wzorce klasowe opisują relację między klasami zaś obiektowe relacje między obiektami.

Zanim jednak o wzorcach parę zasad programowania.

Zasady SOLID

S – Single Responsibility Principle

Zasada pojedynczej odpowiedzialności. Każda klasa, każda metoda, każda funckja zajmuje się jedną konkretną rzeczą. Metoda, której zadaniem jest wysłanie maila za pomocą odpowiedniego protokołu nie powinna zajmować się walidacją adresu mailowego.

```
class Person
{
    public string Name { get; set; }
    public string Lastname { get; set; }
    public string City { get; set; }
    public string Street { get; set; }
    public int HouseNumber { get; set; }
    public string Email { get; set; }

    public Person(string name, string lastname, string email)
    {
        Name = name;
        Lastname = lastname;
        Email = ValidateEmail(email);
    }

    private string ValidateEmail(string email)
    {
        if (!email.Contains("@") || !email.Contains("."))
        {
            throw new FormatException("Email address has a wrong format!");
        }

        return email;
    }
}
```

Powyższy przykład jest jeszcze w granicach. Autor kodu stworzył klasę, która przechowuje dane osobowe. Dodatkowo wrzucił do niej trochę logiki walidującej maila. Ma to w miarę sens. Problem pojawi się dopiero gdy z funkcji walidującej będzie chciała skorzystać inna klasa. Wtedy najlepiej zgodnie z SRP wyciągnąć metodę walidującą do klasy MailValidator lub coś podobnego.

O – Open Close Principle

Każda klasa powinna być otwarta na rozbudowę, ale ta rozbudowa powinna być możliwa bez konieczności modyfikowania obecnego w klasie kodu. Istnieje duże prawdopodobieństwo, że modyfikując kod w celu dodania nowego uszkodzimy obecną implementacje. Przykład zły i dobry

```
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;
    }
}
```

```
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();
    }
}
```

Po lewej znajduje się zły przykład. Dodanie nowej figury do kalkulatora pola wymusza modyfikację metody Area(). W przygładzeni po prawej dodanie nowej figury automatycznie dodaje możliwość liczenia pola dzięki mechanizmowi polimorfizmu. To jest raczej dosyć ekstremalny przykład.

Te zasadę najlepiej rozumieć tak: piszmy klasy i metody tak by dodanie nowych rzeczy nie zmuszało nas do modyfikowania obecnego kodu bo jest ogromna szansa, że go zepsujemy i

nawet się nie zorientujemy (zepsujemy jakiś względnie mniej popularny przypadek użycia jednak możliwy)

L – Liskov Substituton Principle

Projektując w aplikacji klasy i ich dziedziczenie należy zrobić to z głową. Mówiąc precyzyjniej. Klasy powinny dziedziczyć tylko to co jest im potrzebne. Lepiej zrobić dwie klasy bazowe o mniejszych rozmiarach, niż jedną ogromną. Przykład:

```
abstract class Animal
{
    public string Name { get; set; }
    public abstract void Run();
}

class Dog : Animal
{
    public override void Run()
    {
        Console.WriteLine("Dog runs");
    }
}

class Fish : Animal
{
    public override void Run()
    {
        throw new NotImplementedException("Fish can not run!");
    }
}
```

Powyższy przykład, pokazuje, że dziedziczenie przez rybę ze zwierzęcia nie ma sensu, ponieważ ryba nie biega. Oczywiście inne metody, mogą pasować. Jeżeli jednak trafia się jedna, która nie pasuje to powinniśmy spróbować zaprojektować to inaczej. Rozwiązaniem tej sytuacji jest zrobienie drugiej klasy bazowej dla ryby.

I – Interface Segregation

Zasada jak powyższa tylko dla interfejsów (C#, Java).

Interfejs wygląda tak:

```
interface IRaportable
{
    void PrintPdf();
    void PrintExcel();
}
```

Jest to taka w pełni abstrakcyjna klasa. Jeżeli dziedziczymy z interfejsu to musimy zaimplementować wszystkie wymienione w nim metody. W tym przypadku dwie.

```
class SalaryRaport : IRaportable
{
    public void PrintPdf()
    {
        // print pdf
    }

    public void PrintExcel()
    {
        // print excel
    }
}

class HighSchoolExam : IRaportable
{
    public void PrintPdf()
    {
        // print Pdf here
    }

    public void PrintExcel()
    {
        throw new NotImplementedException();
    }
}
```

Jak widać, autor nie planował drukować w formacie Excel wyników egzaminów, ale musiał zaimplementować metodę (dać jej jakieś ciało) bo wymuszał to interfejs. Inaczej otrzymał by błąd kompilacji. Przypadek taki podobnie jak w zasadzie trzeciej mówi, że trzeba inaczej zaplanować kto i z czego dziedziczy dodając np. więcej interfejsów, które są bardziej drobne:

```
interface IPrintablePdf
{
    void PrintPdf();
}

interface IPrintableExcel
{
    void PrintExcel();
}

class SalaryRaport : IPrintablePdf, IPrintableExcel
{
    public void PrintPdf()
    {
        // print pdf
    }

    public void PrintExcel()
    {
        // print excel
    }
}

class HighSchoolExam : IPrintablePdf
{
    public void PrintPdf()
    {
        // print Pdf here
    }
}
```

W C# można dziedziczyć z wielu interfejsów ale tylko z jednej klasy.

D – Depency Inversion

Zasada mówiąca, że powinno się korzystać jak najbardziej z polimorfizmu:

```
1 reference
class DzieckoA: Bazowa
{
    2 references
    public override void PrzedstawSie() => Console.WriteLine("DzieckoA");
}

1 reference
class DzieckoB :Bazowa
{
    2 references
    public override void PrzedstawSie() => Console.WriteLine("DzieckoB");
}

0 references
internal class Program
{
    private static Bazowa zaleznosc; //dependency inversion, mogę tu przypisać obiekt 3 klas

    3 references
    static void PrzedstawSie(Bazowa target) //dependency inversion, dzięki temu mogę tu podać ob
    {
        target.PrzedstawSie();
    }

    0 references
    static void Main(string[] args)
    {
        PrzedstawSie(new Bazowa());
        PrzedstawSie(new DzieckoA());
        PrzedstawSie(new DzieckoB());

        Console.WriteLine("Hello World!");
    }
}
```

Mam klasę bazową i dwoje jej dzieci. W klasie Program, stworzyłem pole statyczne na klasę Bazową, dzięki czemu mogę tam przypisać zarówno obiekt klasy bazowej jak i obojga dzieci.

Drugi przykład jest jaśniejszy. Funkcja PrzedstawSie() przyjmuje obiekt klasy Bazowej, ale mogę tam podać zarówno obiekt klasy bazowej jak i dzieci co widać w mainie. Jest to możliwe dzięki, dziedziczeniu i polimorfizmowi.

Wzorzec: Dependency Injection

Wzorzec, mówiący o tym, że zależności klasy powinno się wstrzykiwać z zewnątrz, np. poprzez konstruktor.

Zły przykład:

```
class MailSender
{
    1 reference
    public void SendMail() { }
}

0 references
class MailMessager
{
    private MailSender sender = new MailSender();

    0 references
    public void MakeMail() { }
    0 references
    public void SendMail() { sender.SendMail(); }
}
```

Klasa MailMessager korzysta z MailSender za pomocą na sztywno umieszczonej w niej referencji. Rozwiązanie takie zadziała, ale jest mało mobilne.

Dobry przykład:

```
class MailSender
{
    1 reference
    public void SendMail() { }
}

1 reference
class MailMessager
{
    private MailSender sender;

    0 references
    public MailMessager(MailSender sender)
    {
        this.sender = sender;
    }

    0 references
    public void MakeMail() { }
    0 references
    public void SendMail() { sender.SendMail(); }
}
```

Jak widać MailSender nie jest ustawiony na sztywno. Jest dostarczany z zewnątrz przez konstruktor. To podejście w połączeniu z polimorfizmem, dziedziczeniem i dependenci inversion daje mega możliwości. Bo tak jak było to z funkcją (zasadę wyżej) możliwe będzie wrzucenie przez konstruktor do pola obiektów rożnych klas, które mogą zawierać różne implementacje tej samej funkcjonalności.

Wzorzec: Kompozyt

Bardzo prosty wzorzec projektowy. Opisuje on relacje między obiektami. Zobaczmy oba przykłady z poprzedniej strony. W obu przypadkach mamy do czynienia z kompozytem. Klasa MailMessager potrzebuje skorzystać z funkcjonalności zawartej w klasie MailSender. Tworzy ona/lub otrzymuje przez konstruktor instancje klasy MailSender i zapisuje w polu. Wykorzystanie pola widać w funkcji SendMail() Messengera.

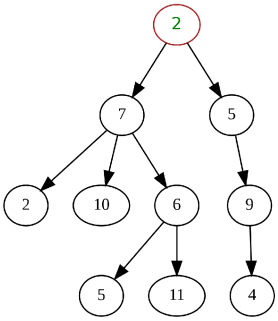
Inny przykład kompozytu to struktura znana pod nazwą drzewa:

```
12 references
class ThreeNode
{
    public int value;

    public ThreeNode leftChild;
    public ThreeNode rightNode;

    2 references
    public ThreeNode(int value)
    {
        this.value = value;
    }

    1 reference
    public ThreeNode(int value, ThreeNode leftChild, ThreeNode rightNode)
    {
        this.value = value;
        this.leftChild = leftChild;
        this.rightNode = rightNode;
    }
}
```

Klasa oprócz swoich danych przechowuje dwie referencje na obiekty samej siebie czyli na swoje dzieci. Oczywiście można dodać możliwość ustawienia tylko jednego dziecka. Albo dodać możliwość posiadania wielu dzieci (tablica/lista typu ThreeNode jako pole).

```
0 references
static void Main(string[] args)
{
    ThreeNode lChild = new ThreeNode(2);
    ThreeNode rChild = new ThreeNode(6);
    ThreeNode parent = new ThreeNode(9, lChild, rChild);

}
```

Oboje dzieci również mogą przyjąć swoje dzieci itd. W ten sposób drzewo się rozrasta w dół:



Wzorzec: Singleton

Bardzo prosty wzorzec pokazujący jak zrobić klasę, której instancje można stworzyć tylko raz. Każda następna próba kończy się zwróceniem tej samej instancji. Do czego można to wykorzystać? Do przechowywania globalnych zmiennych, które muszą być dostępne między wieloma plikami projektu.

```
4 references
class UserData
{
    //normalna część klasy
    public string name;
    public string number;
    //...

    //część od wzorca
    private static UserData self;

    1 reference
    private UserData() { }

    0 references
    public static UserData GetUserData()
    {
        if (self is null)
        {
            self = new UserData();
        }

        return self;
    }
}
```

Klasa posiada swoje pola i metody jak zawsze. Poza tym posiada statyczny wskaźnik/referencje na siebie, prywatny konstruktor oraz metodę zwracającą jedną i zawsze te samą instancje.

```
0 references
static void Main(string[] args)
{
    UserData u = new UserData(); //error bo konstuktor jest prywatny

    UserData u1 = UserData.GetUserData(); //ta metoda zwróci ZAWSZE ten samo obiekt
    //zapisany w polu self
}
```

Dzięki zastosowaniu prywatnego konstruktora wyłączamy możliwość tworzenia obiektów w normalny sposób, ale nie blokujemy jej wewnątrz naszego singletona. Utworzenie obiektu UserData możliwe jest tylko wewnątrz UserData. Metoda tworząca obiekt jest statyczna i dzięki instrukcji if tworzy obiekt tylko za pierwszym razem. Czas życia statycznego jest praktycznie równy czasowi życia programu, więc raz ustawiony self jest dostępny cały czas.

Niektórzy pewnie powiedzą, że singleton to antywzorzec bo powinno się unikać statyczności, ale to kwestia sporna.

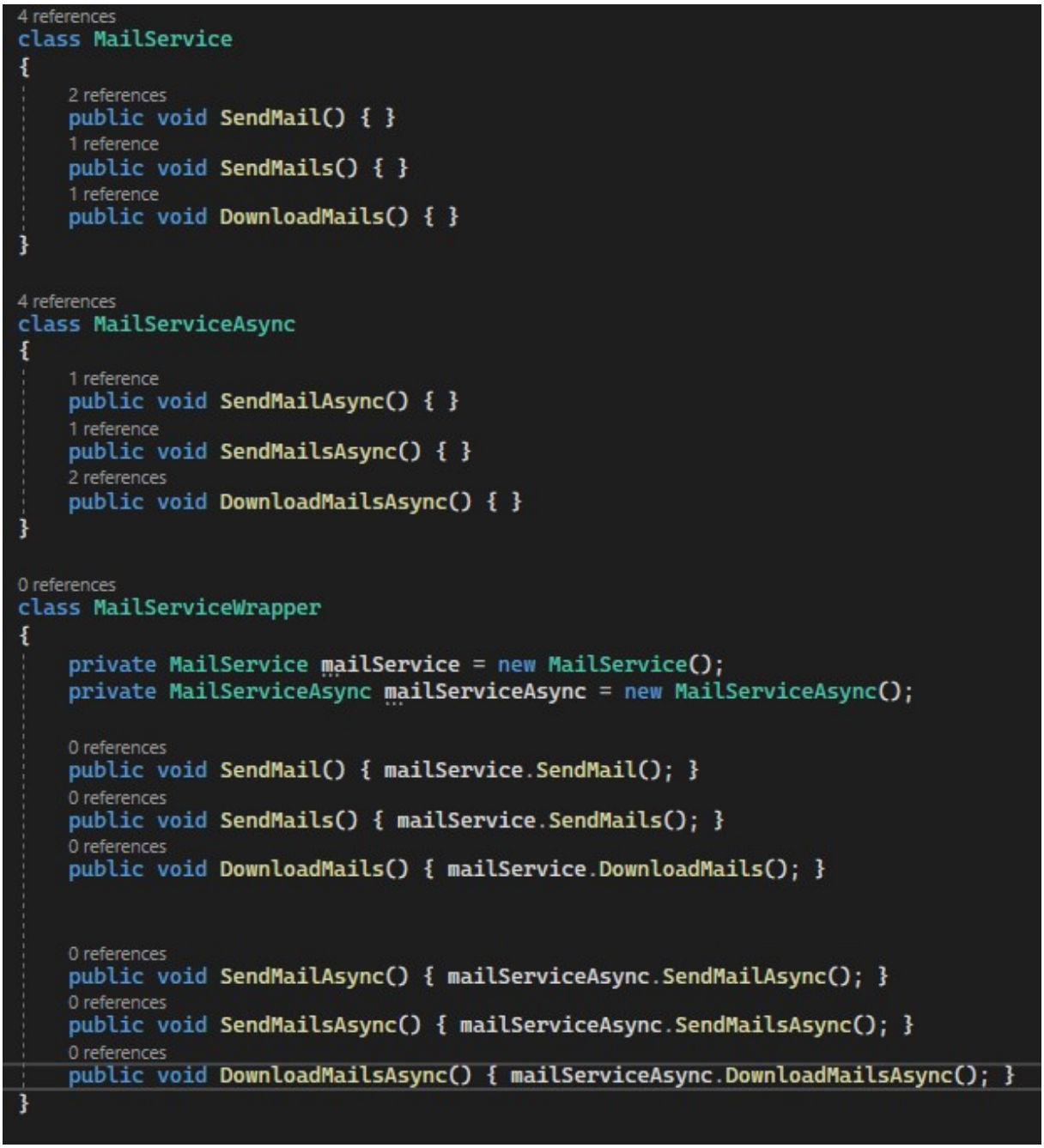
Wzorzec: Fasada

Wzorzec pozwalający za pomocą jednej klasy zagregować funkcjonalności kliku klas tak by wystarczyło zrobienie jednego obiektu z wszystkimi funkcjami a nie np. dziesięciu:

Przykład bez fasady:

```
2 references
class MailService
{
    1 reference
    public void SendMail() { }
    0 references
    public void SendMails() { }
    0 references
    public void DownloadMails() { }
}

2 references
class MailServiceAsync
{
    0 references
    public void SendMailAsync() { }
    0 references
    public void SendMailsAsync() { }
    1 reference
    public void DownloadMailsAsync() { }
}

0 references
internal class Program
{
    0 references
    static void Main(string[] args)
    {
        MailService mailService = new MailService();
        MailServiceAsync mailServiceAsync = new MailServiceAsync();

        mailService.SendMail();
        mailServiceAsync.DownloadMailsAsync();

    }
}
```

Mamy n klas z pewną funkcjonalnością. Chcąc korzystać z ich możliwości musimy tworzyć n obiektów i wywoływać na nich metody. Na dłuższą metę może to być uciążliwe. Dlatego lepiej stworzyć klasę agregującą:



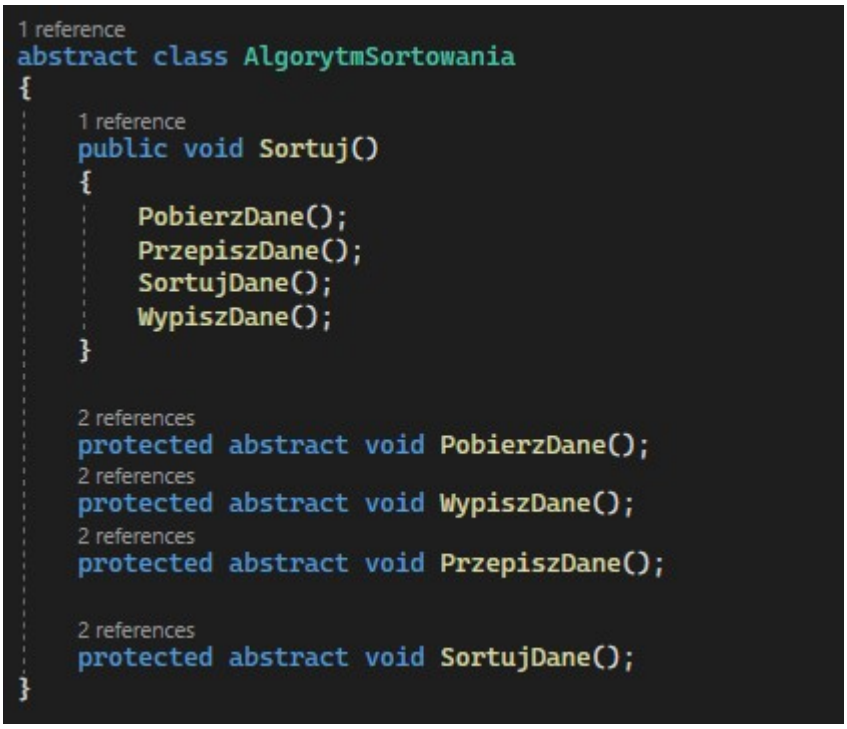
```
4 references
class MailService
{
    2 references
    public void SendMail() { }
    1 reference
    public void SendMails() { }
    1 reference
    public void DownloadMails() { }
}

4 references
class MailServiceAsync
{
    1 reference
    public void SendMailAsync() { }
    1 reference
    public void SendMailsAsync() { }
    2 references
    public void DownloadMailsAsync() { }
}

0 references
class MailServiceWrapper
{
    private MailService mailService = new MailService();
    private MailServiceAsync mailServiceAsync = new MailServiceAsync();

    0 references
    public void SendMail() { mailService.SendMail(); }
    0 references
    public void SendMails() { mailService.SendMails(); }
    0 references
    public void DownloadMails() { mailService.DownloadMails(); }


    0 references
    public void SendMailAsync() { mailServiceAsync.SendMailAsync(); }
    0 references
    public void SendMailsAsync() { mailServiceAsync.SendMailsAsync(); }
    0 references
    public void DownloadMailsAsync() { mailServiceAsync.DownloadMailsAsync(); }
}
```

Teraz za każdym razem gdy będziemy potrzebowali funkcji MailSenderX wystarczy utworzyć jeden obiekt wrappera a nie n obiektów.

Polecam agregować zgodnie z zasadą pojedynczej odpowiedzialności czyli tematycznie w grupy a nie byle jak.

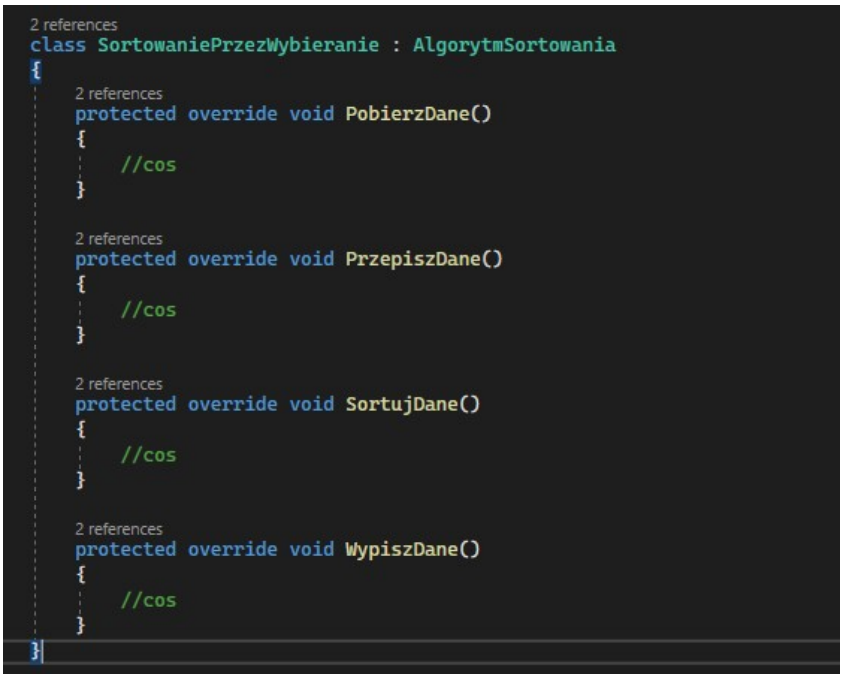
Wzorzec: Metoda Szablonowa

Wzorzec ma za zadanie zbudować szkielet pewnego procesu/funkcjonalności a konkretną implementacje poszczególnych etapów pozostawić klasom pochodnym.



```
1 reference
abstract class AlgorytmSortowania
{
    1 reference
    public void Sortuj()
    {
        PobierzDane();
        PrzepiszDane();
        SortujDane();
        WypiszDane();
    }

    2 references
    protected abstract void PobierzDane();
    2 references
    protected abstract void WypiszDane();
    2 references
    protected abstract void PrzepiszDane();

    2 references
    protected abstract void SortujDane();
}
```

Zdeklarowaliśmy klasę szablonu. Definiuje ona, że sortowanie przebiega w czterech krokach (oczywiście jakaś logika w tej metodzie jest dozwolona), ale konkretną implementacje etapów zostawia dzieciom.



```
2 references
class SortowaniePrzezWybieranie : AlgorytmSortowania
{
    2 references
    protected override void PobierzDane()
    {
        //cos
    }

    2 references
    protected override void PrzepiszDane()
    {
        //cos
    }

    2 references
    protected override void SortujDane()
    {
        //cos
    }

    2 references
    protected override void WypiszDane()
    {
        //cos
    }
}
```

Każde dziecko, ile byśmy ich nie stworzyli dostarcza swoją implementacje etapów. Jednak każde z nich ma identyczną metodę Sortuj().

```
0 references
static void Main(string[] args)
{
    SortowaniePrzezWybieranie s = new SortowaniePrzezWybieranie();
    s.Sortuj();
}
```

Inna implementacja:

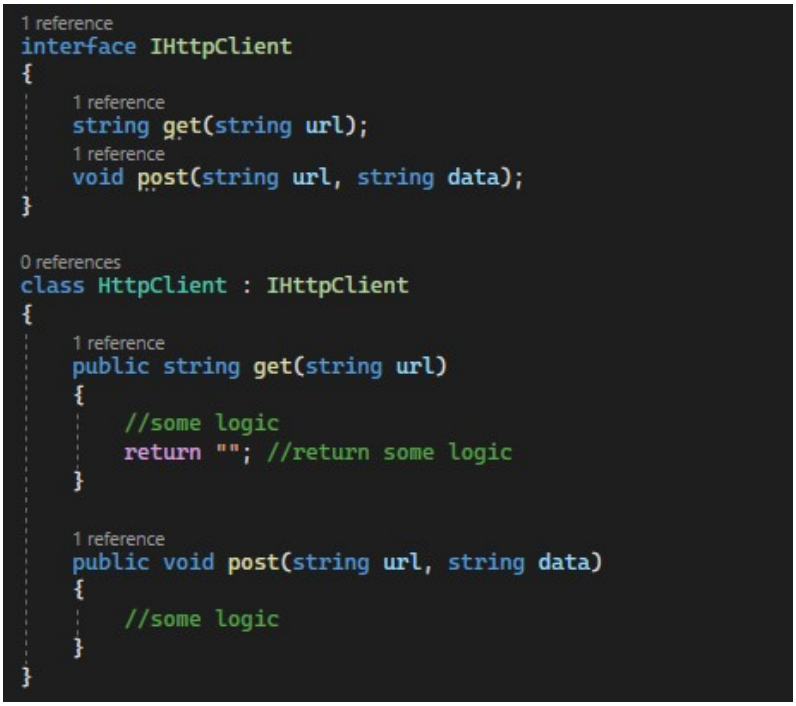
```
1 reference
class SortowaniePrzezWstawianie : AlgorytmSortowania
{
    2 references
    protected override void PobierzDane()
    {
        //cos
    }

    2 references
    protected override void PrzepiszDane()
    {
        //cos
    }

    2 references
    protected override void SortujDane()
    {
        //inna implementacja
    }

    2 references
    protected override void WypiszDane()
    {
        //cos
    }
}
```

```
0 references
static void Main(string[] args)
{
    AlgorytmSortowania s = new SortowaniePrzezWybieranie();
    s.Sortuj();

    AlgorytmSortowania s2 = new SortowaniePrzezWstawianie();
    s2.Sortuj();
}
```

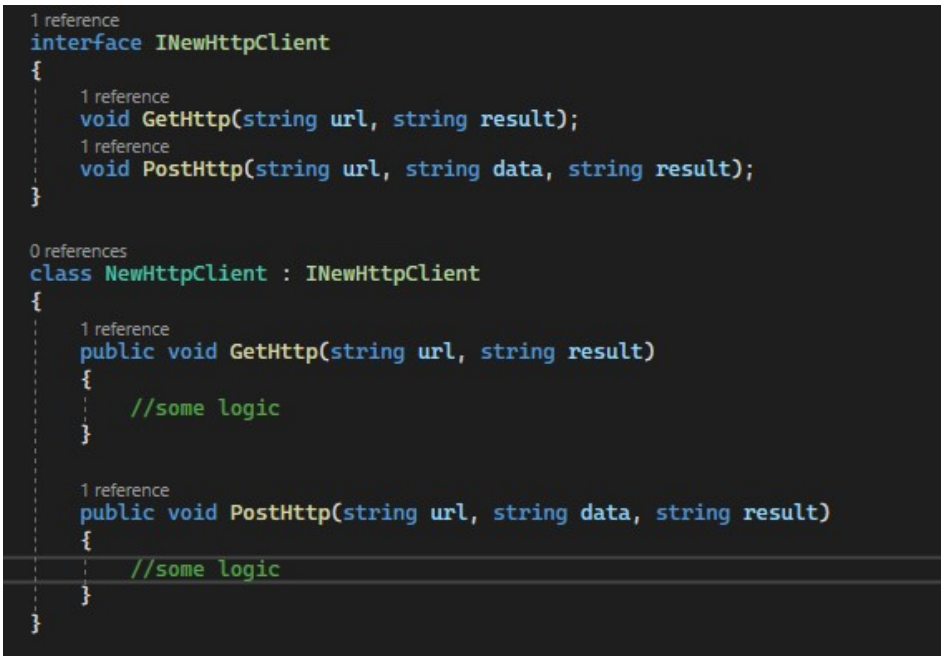
Wzorzec: Adapter

Adapter to wzorzec, którego zadaniem jest zaadaptowanie nowego api tak by było kompatybilne ze starym. Po co? Żeby uniknąć zmian w obecnym kodzie wykorzystującym stare API. Łatwiej będzie pokazać to na przykładzie:

Mamy takie API klasy HttpClient:



Nasze nowe api dla klienta http:



Problem polega na tym, że jeżeli chcemy podmienić w naszej aplikacji starą implementację na nową musimy dokonać wielu zmian. Np. nie pokrywają się parametry funkcji oraz sposób zwracania wyniku.

Rozwiązaniem problemu jest stworzenie adaptera czyli klasy, który „ubierze” nowe API w stare szaty. Można tego dokonać na dwa sposoby.

Adapter klasowy – bazuje na dziedziczeniu:

```
0 references
class HttpClientAdapter : NewHttpClient
{
    0 references
    public string get(string url)
    {
        string result = String.Empty;
        GetHttp(url, result);
        return result;
    }

    0 references
    public void post(string url, string data)
    {
        string result = String.Empty;
        PostHttp(url, data, result);
    }
}
```

Jak widać klasa adaptera wykonała swego rodzaju nakładkę na nowe API, która wygląda jak stare API. Dzięki czemu możemy nowego API używać jak starego.

Drugi sposób do adapter obiektowy:

```
0 references
class HttpClientAdapter
{
    private NewHttpClient newClient = new NewHttpClient();
    0 references
    public string get(string url)
    {
        string result = String.Empty;
        newClient.GetHttp(url, result);
        return result;
    }

    0 references
    public void post(string url, string data)
    {
        string result = String.Empty;
        newClient.PostHttp(url, data, result);
    }
}
```

Tutaj wykorzystaliśmy kompozycję zamiast dziedziczenia. Efekt jest ten sam.

Adapter to trywialny wzorzec, a niezmiernie przydatny.

Wzorzec: Dekorator

Dekorator to bardzo prosty wzorzec projektowy. Jego zadaniem jest dodanie funkcjonalności do istniejącej już klasy. Wyobraźmy sobie, że zaimportowaliśmy klasę z biblioteki. Jest to czyjaś praca, którą dodaliśmy do swojego projektu. Chcielibyśmy ją zamienić w pewien sposób, ale nie możemy bo nie mamy do niej dostępu. Możemy jednak ją udekorować za pomocą swojej klasy.

```
//klasa z biblioteki
0 references
class Calculator
{
    0 references
    public int add(int a, int b)
    {
        return a + b;
    }

    0 references
    public int sub(int a, int b)
    {
        return a - b;
    }

    0 references
    public int mul(int a, int b)
    {
        return a * b;
    }

    0 references
    public int div(int a, int b)
    {
        return a / b;
    }
}
```

```
//nasza klasa dekorująca
0 references
class MyCalculator : Calculator
{
    0 references
    public int myAdd(int a, int b)
    {
        return add(a, b);
    }

    0 references
    public int mySub(int a, int b)
    {
        return sub(a, b);
    }

    0 references
    public int myMul(int a, int b)
    {
        return mul(a, b);
    }

    0 references
    public int myDiv(int a, int b)
    {
        if (b == 0)
            throw new DivideByZeroException();
        return div(a, b);
    }
}
```

Jak widać dodaliśmy walidację parametru do funkcji dzielącej bez modyfikowania oryginalnej klasy. Również nie implementowaliśmy już danych nam operacji od zera. Cały trick bazuje na dziedziczeniu. Można go jednak wykonać za pomocą kompozycji:

```
//nasza klasa dekorująca
0 references
class MyCalculator
{
    private Calculator baseCalculator = new Calculator();

    0 references
    public int myAdd(int a, int b)
    {
        return baseCalculator.add(a, b);
    }

    0 references
    public int mySub(int a, int b)
    {
        return baseCalculator.sub(a, b);
    }

    0 references
    public int myMul(int a, int b)
    {
        return baseCalculator.mul(a, b);
    }

    0 references
    public int myDiv(int a, int b)
    {
        if (b == 0)
            throw new DivideByZeroException();
        return baseCalculator.div(a, b);
    }
}
```

Wzorzec: Obserwator

Wzorzec obserwator ma zastosowanie w sytuacji gdy zmiana stanu jednego obiektu (obiektu obserwowanego) ma zostać zasygnalizowana innym obiektom (obiektom obserwatorom).

```
0 references
class Target
{
    //this class base logic
    //...

    private Observer observer = new Observer();

    0 references
    public void NotifyObserver()
    {
        observer.DoSomething();
    }
}
```

Nasza obserwowana klasa wygląda tak. Posiada ona jednego obserwatora. W momencie wykonania jakiejś akcji z logiki klasy, może ona wywołać metodę NotifyObserver, która powiadomi obserwującego lub obserwujących o zdarzeniu.

Możemy trochę udoskonalić ten przykład:

```
2 references
interface IObserver
{
    3 references
    void DoSomething(); //called to notify
}

2 references
class Observer : IObserver
{
    //this class logic

    2 references
    public void DoSomething()
    {
        //do somthing
    }
}

0 references
class AnotherObserver : IObserver
{
    1 reference
    public void DoSomething()
    {
        //do something
    }
}
```

Stworzyliśmy kilku obserwatorów. Wykorzystamy polimorfizm na interfejsie do łatwego dodania ich do obiektu obserwowanego.

```
0 references
class Target
{
    //this class base logic
    //...

    private List<IObserver> observers = new List<IObserver>();

    0 references
    public void RegisterObserver(IObserver o)
    {
        observers.Add(o);
    }

    0 references
    public void NotifyObserver()
    {
        foreach(IObserver o in observers) //notify all
        {
            o.DoSomething();
        }
    }
}
```

Dzięki zastosowaniu polimorfizmu i interfejsu w jednej liście jesteśmy w stanie przechowywać wielu obserwatorów (każdy zaimplementowany inaczej, ale mający takie samo api zdefiniowane w interfejsie).

Wzorzec MVVM bardzo mocno bazuje na obserwowaniu.

Wiele operacji asynchronicznych bazuje na obserwowaniu. W momencie powiadomienia zostaje uruchomiony callback podany jako onFinished (lub coś w ten deseń).

Wzorzec: Odwiedzający

Celem wzorca jest zbudowanie klasy, która będzie w jakimś celu odwiedzać czyli używać obiektów innej klasy.

Przykład z życia to klasa ListView (lista przewijana w programowaniu mobilnym) oraz klasa ListViewItem (każdy z elementów na liście). Klasa ListView w celu wyrenderowania siebie, musi odwiedzić każdy z elementów i najpierw go wyrenderować.

```
2 references
class ListViewItem
{
    //some logic

    1 reference
    public void RenderMe() { /**/ }
}

0 references
class ListView
{
    //some logic

    private List<ListViewItem> items = new List<ListViewItem>();

    0 references
    public void RenderMe()
    {
        items.ForEach(item => item.RenderMe()); //render each child
        //actuall self render
    }
}
```

To ten rodzaj wzorca, który brzmi groźnie a w praktyce jest trywialny. Jak widać główna klasa przed wykonaniem swoich operacji wykonuje operacje przypisane do każdego odwiedzanego obiektu-dziecka.

Do czego można zastosować ten wzorzec?

Wyobraźmy sobie, że klasa odwiedzająca przechowuje logikę jakiegoś algorytmu. Każda obiekt odwiedzany przechowuje tylko dane do obróbki przez algorytm, Klasa odwiedzająca odwiedza każdy obiekt z danymi, wykonuje na nim algorytm i zostawia obrobionym. Proste.

Wzorzec: Budowniczy oraz Fabryka

Wyobraźmy sobie sytuacje, w której mamy do czynienia z bardzo rozbudowaną klasą. Klasa ta posiada ogromne ilości logiki, ogromne ilości parametrów do ustawienia (które są wymagane do jej prawidłowego działania) oraz skomplikowane konstuktory/settery.

Zadaniem wzorca budowniczego/fabryki jest stworzenie klasy, która buduje za nas obiekt tej skomplikowanej klasy ułatwiając nam jej użycie np. poprzez dostarczenie bardziej przyjaznego API do konfiguracji klasy docelowej.

```
// kierownik - logika inicjalizacji obiektu
class CarDirector
{
    public void ConstructCar(ICarBuilder builder)
    {
        builder.BuildWheels();
        builder.BuildAddons();
        builder.BuildEngine();
    }
}

// budowniczy - interfejs implementacji
interface ICarBuilder
{
    void BuildWheels();
    void BuildAddons();
    void BuildEngine();
    Car GetResult();
}

// konkretny budowniczy - implementacja
class SmallCarBuilder : ICarBuilder
{
    private Car car = new Car();
    private EngineFactory engineFactory = new EngineFactory();

    public void BuildWheels()
    {
        car.Wheels = "Steel rims 15 inches";
    }

    public void BuildAddons()
    {
        car.Addons = new List();
        car.Addons.Add("CD radio with MP3");
        car.Addons.Add("CD radio");
    }

    public void BuildEngine()
    {
        car.Engine = engineFactory.CreateEngine("120 HP engine");
    }

    public Car GetResult()
    {
        return car;
    }
}
```

```
static void Main(string[] args)
{
    CarDirector carDirector = new CarDirector();
    ICarBuilder smallCarBuilder = new SmallCarBuilder();

    // buduj według logiki CarDirector używając implementacji z SmallCarBuild
    carDirector.ConstructCar(smallCarBuilder);

    Car smallCar = smallCarBuilder.GetResult();
}
```

Powyżej znajduje się mega przesadzony przykład z Internetu. Dlaczego przesadzony? Ponieważ oprócz klasy budującej posiada klasę zarządzającą budową. Budujemy obiekt klasy Car. Budowaniem zajmuje się SmallCarBuilder, ale decyzje co i w jaki sposób zostanie przez niego ustawione zależy od CarDirectora.

Wzorzec ten może wydawać się bez sensu, ale istnieje wiele przypadków gdzie jest stosowany i nie koniecznie wynika to z faktu iż API klasy budowanej jest skompikowane.

Przykład z życia:

```
string napis = String.Empty;
for(int i = 0; i < 10; i++)
{
    napis += i.ToString();
}
//napis = "0123456789"
```

Niby banalny kod, ale bardzo nie przyjemny dla komputera. Można to zrobić lepiej:

```
string napis = String.Empty;
StringBuilder napisBuilder = new StringBuilder();
for(int i = 0; i < 10; i++)
{
    napisBuilder.Append(i.ToString());
}
napis = napisBuilder.ToString();
//napis = "0123456789"
```

Dlaczego doklejając do zwykłego stringa inny string robimy coś nie tak? Przypisywanie takie powoduje tworzenie za każdym razem nowego stringa w pamięci (alokację nowej przestrzeni i przepisanie starej wartości do niej w połączeniu z nową). Takie coś kosztuje. Może nie przy 10 iteracjach, ale przy tysiącach powtarzanych wiele razy już tak. StringBuilder nie alokuje nowej przestrzeni a dynamicznie rozszerza aktualną dzięki czemu koszt jest mniejszy.

Jak widać budowniczy może przykryć nie tylko skomplikowane API, ale też złagodzić kosztowną podstawową implementację.

Podsumowanie:

Jeżeli ktoś jest w stanie budować aplikację z góry stosując wzorce projektowe oznacza to, że osiągnął całkiem wysoki poziom. Należy pamiętać jednak, że stosowanie wzorców na siłę, może nieść odwrotne skutki do tych zamierzonych. Z drugiej strony często nie zdajemy sobie sprawy, że użyliśmy wzorca projektowego.

Pytanie co jest lepsze? Dziedziczenie czy kompozycja (jak w wzorcu adapter)?

Oczywiście kompozycja jest o niebo lepsza. Jeżeli klasa coś dziedziczy to nie da się tego zmienić w trakcie działania aplikacji. Nie da się „oddziedziczyć" i kazać dziedziczyć z czegoś innego. W przypadku kompozycji możliwe jest podmienienie zawartości pola z obiektem. Jeżeli do tego zastosujemy polimorfizm i dependency injection to podmiana implementacji może być płynna w czasie trwania programu.

```
2 references
interface IMailSender
{
    2 references
    void SendMail();
}

0 references
class MailSender : IMailSender
{
    1 reference
    public void SendMail()
    {
        //send mail
    }
}

0 references
class MailSender2 : IMailSender
{
    1 reference
    public void SendMail()
    {
        //also send mail but do some stuff diffrent way
    }
}
```

```
0 references
class App : MailSender
{
    //some logic

    0 references
    public void SendMessage()
    {
        //some stuff
        SendMail();
    }
}
```

W tym przypadku nie jesteśmy w stanie użyć już MailSender2, ponieważ dziedziczymy z MailSender. W C# klasa może dziedziczyć tylko z jednej klasy. Nie da się zmienić dziedziczenia.

```
0 references
class App
{
    //some logic

    //kompozycja
    public IMailSender sender = new MailSender();

    0 references
    public void SendMessage()
    {
        //some stuff
        sender.SendMail();
    }
}
```

W tym przypadku nawet jak użyjemy metody SendMessage() w domyślnej konfiguracji wciąż możliwe jest podmienienie sendera (jest to pole publiczne) na implementacje 2.

Lepszy przykład wygląda tak:

```
1 reference
class App
{
    //some logic

    //kompozycja
    private IMailSender sender;
    2 references
    internal IMailSender Sender { get => sender; set => sender = value; }

    0 references
    public App(IMailSender sender)
    {
        this.Sender = sender;
    }

    0 references
    public void SendMessage()
    {
        //some stuff
        Sender.SendMail();
    }
}
```

Tutaj korzystamy z dependency injection. Wstrzykujemy jeden z dwóch senderów z zewnątrz poprzez konstruktor. Jeżeli zajdzie potrzeba podmiany implementacji można tego dokonać za pomocą publicznego settera.

Tak więc. **Dependency Injection** to bardzo potężne narzędzie. Polecam mocno. Każdy wzorzec staje się jeszcze lepszy z wykorzystaniem tej zasady.

Lista wszystkich wzorców została wymieniona na polskiej Wiki:

https://pl.wikipedia.org/wiki/Wzorzec_projektowy_(informatyka)