Projektowanie obiektowe oprogramowania
Wykład 3 - SOLID GRASP
Wiktor Zychla 2025

1 Taksonomia Responsibility-Driven Development

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).

Odpowiedzialność – kontrakt, zobowiązanie, związane z **działaniem** lub **wiedzą**.

Działanie:

- Wykonywanie czynności, tworzenie innego obiektu, przeprowadzanie obliczeń
- Inicjalizacja czynności wykonywanych przez inny obiekt
- Kontrola, koordynacja czynności wykonywanych przez inny obiekt

Wiedza:

- Udostępnianie danych
- Wiedza o obiektach powiązanych

RDD = Responsibility-Driven Development, przemyślane projektowanie obiektowe.

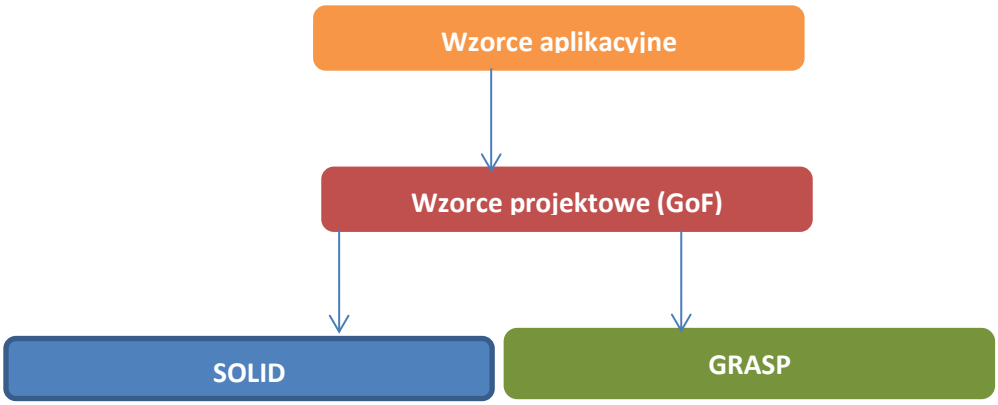
Problem – w procesie analizy obiektowej zidentyfikowano już co trzeba zrobić (przypadki użycia, mapy procesów). Na etapie projektowania architektury całych podsystemów jak i konkretnych fragmentów należy zbudować zbiory klas realizujące zidentyfikowane wymagania.

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

Wzorzec – **nazwana** para (problem, rozwiązanie), którą można powielić w różnych kontekstach, opisana ze wskazówkami stosowania i konsekwencjami użycia.

Uwaga – większość systemów informatycznych ma wiele warstw w tym:

- warstwę **dostępu do danych** – za pomocą której aplikacja komunikuje się z repozytoriami danych, np. bazami relacyjnymi
- warstwę **modelu dziedzinowego** – w której określone są klasy będące wynikiem refaktoryzacji modelu dziedzinowego do modelu klas (patrz poprzedni wykład)
- warstwę **logiki biznesowej** – gdzie zaimplementowano procesy biznesowe interfejs użytkownika
- warstwę **interfejsu użytkownika**

Wszystkie składowe warstwy aplikacji nazywa się żargonowo **stosem aplikacyjnym**.

Spośród tych warstw warstwa modelu dziedzinowego jest szczególna – jej w najmniejszym stopniu dotyczą reguły projektowania obiektowego (SOLID/wzorce), ponieważ model dziedzinowy opisuje realne byty, ich odpowiedzialności i relacje między nimi. Większość zasad projektowania nie ma więc w tej warstwie nic do gadania, wszak o tym czy „mysz" „zjada" „ser" nie decyduje „dobra praktyka", tylko po prostu „tak jest".

Od tej pory zajmujemy się więc takimi regułami projektowania obiektowego, które mają zastosowanie we wszystkich warstwach stosu aplikacyjnego z wyłączeniem warstwy modelu dziedzinowego.

Wyjątkiem są tu reguły GRASP, one są na tyle ogólne że ich świadomość często pomaga w dobrym uchwyceniu subtelności modelu dziedzinowego.

2 GRASP

General Responsibility Assignment Software Patterns (Principles)

1	Creator
2	Information Expert
3	Controller
4	Low Coupling
5	High Cohesion
6	Polymorphism
7	Indirection
8	Pure fabrication
9	Protected Variations

2.1 Creator

Nazwa	**Creator** (Twórca)
Problem	Kto tworzy instancje klasy A?
Rozwiązanie	Przydziel zobowiązanie tworzenia instancji klasy A klasie B, jeżeli zachodzi jeden z warunków: • B „zawiera” A lub agreguje A (kompozycja) • B zapamiętuje A • B bezpośrednio używa A • B posiada dane inicjalizacyjne dla A

Uwagi: *właściwe zarządzanie tworzeniem, niszczeniem i czasem życia obiektów to jedno z podstawowych zadań*. *Wydawałoby się, że przecież do tworzenia obiektów w językach obiektowych wystarczy operator* ***new****, ale w rzeczywistości ma on poważne ograniczenie – wiąże klienta z konkretną implementacją. Stąd tak a nie inaczej sformułowany zbiór „warunków” – bo albo B dokładnie wie że potrzebuje A albo B służy innym do tworzenia instancji A (co umożliwia wariantowość implementacji tworzenia).*

2.2 Information Expert

Nazwa	**Information Expert**
Problem	Jak przydzielać obiektom zobowiązania?
Rozwiązanie	Przydziel zobowiązanie “ekspertowi” – tej klasie, która ma *informacje* konieczne do jego realizacji.

Uwagi: *ta zasada często pozwala rozstrzygać wątpliwości projektowe typu „Gdzie powinna być metoda X”? Nazbyt często stosowane w połączeniu z Pure Fabrication może prowadzić do systemu który ma zbyt małą spójność (dużą liczbę klas, które z niczego nie korzystają tylko z nich się korzysta).*

2.3 Controller

Nazwa	**Controller**
Problem	Który z obiektów poza warstwą GUI odbiera żądania operacji systemowych i kontroluje jej wykonanie?
Rozwiązanie	Przydziel odpowiedzialność do obiektu spełniającego jeden z warunków: • Obiekt reprezentuje cały system • Obiekt reprezentuje przypadek użycia w ramach którego wykonywana jest operacja (<NazwaPrzypadku>Handler, <NazwaPrzypadku>Controller)

Uwagi: *o wzorcach MVC/MVP będziemy mówić na wykładzie*.

2.4 Low Coupling

Nazwa	**Low Coupling**
Problem	Jak zmniejszyć liczbę zależności I zasięg zmian, a zwiększyć możliwość ponownego wykorzystania kodu?
Rozwiązanie	Przydziel odpowiedzialność tak, aby ograniczyć stopień sprzężenia (liczbę powiązań obiektu). Stosuj tę zasadę na etapie projektowania.

Zależność (sprzężenie):

- Obiekt A ma atrybut typu B lub typu C związanego z B
- Obiekt A wywołuje metody obiektu typu B
- Obiekt A ma metodę związaną z typem B (typ wartości, parametru lub zmienna lokalna)
- Obiekt A dziedziczy po B

Uwagi: *istnieją wzorce pozwalające ograniczać liczbę zależności, zauważmy tylko że problem nie występuje dla układu 2 klas, ale tak od 4 wzwyż zaczyna robić się interesująco*.

2.5 High Cohesion

Nazwa	**High Cohesion**
Problem	Jak sprawić by obiekty miały jasny cel, były zrozumiałe i łatwe w utrzymaniu?
Rozwiązanie	Przydziel odpowiedzialność by spójność pozostawała wysoka.

Spójność = „sprzężenie” w ramach jednej i tej samej klasy pomiędzy jej składowymi.

Klasa o niskiej spójności wykonuje niepowiązane zadania lub ma ich zbyt dużo:

- Trudno je zrozumieć
- Trudno je ponownie wykorzystać
- Trudno je utrzymywać
- Są podatne na zmiany

Uwaga – zasady **Low Coupling** i **High Cohesion** stosuje się zarówno na poziomie pojedynczych klas jak i całych modułów:

	Low Coupling	**High Cohesion**
Klasa	Klasa ma mało zależności do innych klas	Metody wewnątrz klasy są ze sobą ściśle sprzężone (korzystają z siebie nawzajem)
Moduł (*.dll, *.jar)	Moduł ma mało zależności do innych modułów	Klasy wewnątrz modułu są ze sobą ściśle sprzężone (korzystają z siebie nawzajem)

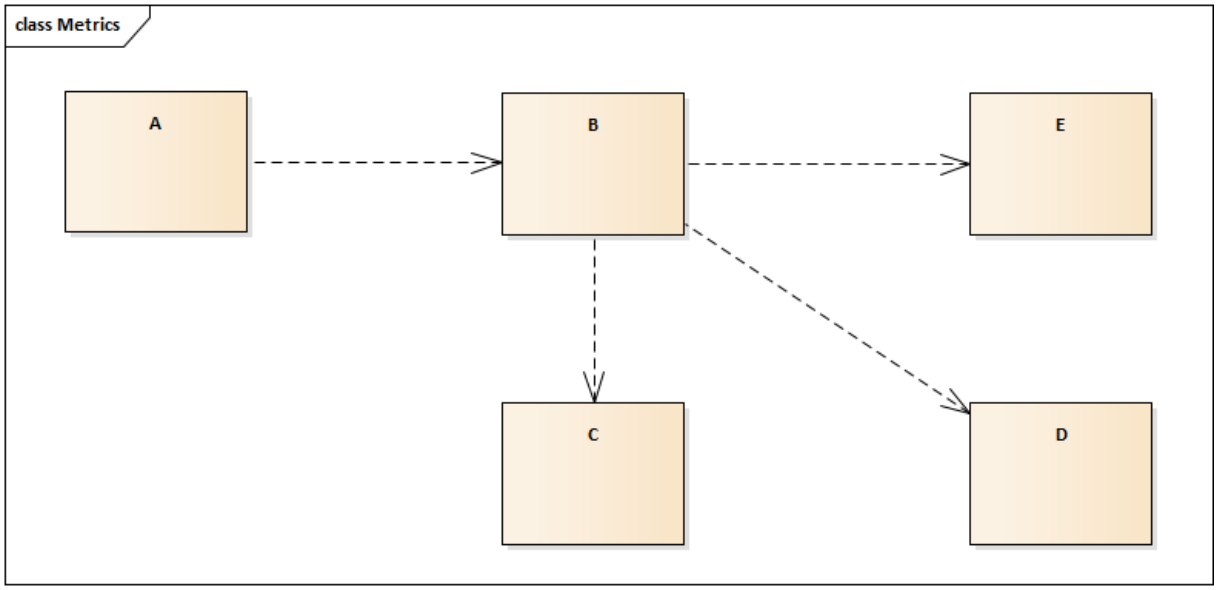
Dygresja. Zależność (coupling) i spójność (cohesion) mają swoje przełożenie na tzw. **metryki kodu**. Oryginalne sformułowanie z połowy lat 60 jest obecnie rzadko wykorzystywane, częściej korzysta się (jeśli w ogóle!) z miar współczesnych, np. Ca/Ce/I/A/D zaproponowanych przez R. Martina w książce *Agile software development*

Pomysł jest następujący.

Dla klas w pakiecie (metod w klasie) można policzyć miary:

- Ce – liczba powiązań między klasami w danym pakiecie a klasami w innych pakietach
- Ca – liczba powiązań między klasami w innych pakietach a klasami w danym pakiecie
- I – *niestabilność* liczona jako stosunek Ce / Ce + Ca
- A – *abstrakcyjność* liczona jako stosunek klas abstrakcyjnych do wszystkich klas, czyli Ta / Ta + Tc
- D – odległość od *wyidealizowanej* osi idealnego pakietu, dla którego I + A = 1, wyliczany ze wzoru | A + I – 1 |

Przykład – niech będą jednoklasowe zestawy o następujących zależnościach.



Dla zestawu B, Ce = 3, Ca = 1. W takim razie I = 0.75.

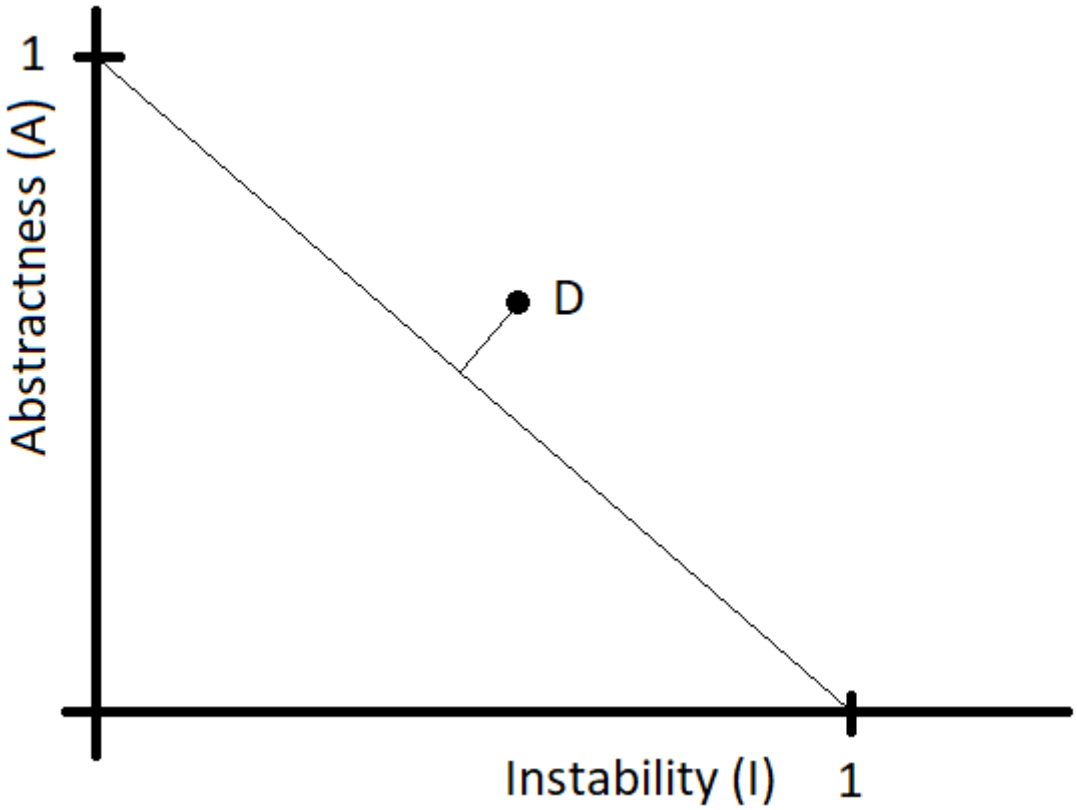
Zauważmy że I pozwala odróżnić następujące skrajności:

- I bliskie 1 – czyli nieduże lub wręcz zerowe Ca, taki pakiet zależy od wielu innych pakietów i jest **niestabilny** z powodu narażenia na zmiany przy zmianach w innych pakietach
- I bliskie 0 – czyli nieduże lub wręcz zerowe Ce, taki od takiego pakietu zależą inne pakiety ale on sam nie zależy aż tak bardzo od niczego, jest więc całkowicie **stabilny**

Tu pakiety całkowicie stabilne to C, D i E a całkowicie niestabilny to A.

Preferowane wg. Martina wartości I to albo przedział 0 do 0.3 (pakiety tworzące fundament architektury) lub 0.7 do 1 (pakiety tworzące fasadę architektury).

Idealna linia D oznacza, że pakiet który jest całkowicie stabilny powinien być też abstrakcyjny (wtedy I = 0, A = 1) a pakiet niestabilny powinien być konkretny (I = 1, A = 0)



Maksymalnie duże D możliwe jest w dwóch przypadkach:

- A = 0 i I = 0 – to pakiet pozbawiony abstrakcji, ulokowany gdzieś w fundamencie architektury. Dostarczanie wyłącznie konkretnych implementacji może w przyszłości powodować trudność w rozszerzaniu funkcjonalności
- A = 1 i I = 1 – to pakiet abstrakcyjny, ulokowany gdzieś na fasadzie architektury, sytuacja trudna do wyobrażenia, taki pakiet byłby bezwartościowym elementem architektury

2.6 Polymorphism

Nazwa	**Polymorphism**
Problem	Jak obsługiwać warunki zależne od typu?
Rozwiązanie	Przydziel zobowiązania - przy użyciu operacji polimorficznych – typom dla których to zachowanie jest różne

Uwagi: *wydaje się oczywiste, a jednak wymaga pewnej rutyny.*

2.7 Indirection

Nazwa	**Indirection**
Problem	Komu przydzielić zobowiązanie jeśli zależy nam na uniknięciu bezpośredniego powiązania między obiektami?
Rozwiązanie	Przydzielić zobowiazanie obiektowi, który pośredniczy między innymi komponentami

Uwagi: *jest wiele sposobów unikania sprzężenia. Pewnymi uszczegółowieniami tej zasady są dwie kolejne, Pure Fabrication i Law of Demeter. Kilka wzorców projektowych (Mediator, Observer, Event Aggregator) pokazuje jak tę zasadę realizować w praktyce.*

2.8 Pure fabrication

Nazwa	**Pure fabrication**
Problem	Jak przydzielić odpowiedzialność by nie naruszyć zasad High Cohesion I Low Coupling a nie odpowiada nam rozwiązanie innych zasad (np. Information Expert)?
Rozwiązanie	Przypisz zakres odpowiedzialności sztucznej lub pomocniczej klasie, która nie reprezentuje konceptu z dziedziny problemu.

Uwaga: *innymi słowy – wprowadź dodatkową klasę tam gdzie naprawdę nie bardzo wiadomo która istniejąca powinna robić to co trzeba zrobić.*

Uwaga: *Indirection vs Pure fabrication:*

- *Nie każde Indirection będzie polegało na utworzeniu nowej klasy*
- *Nie każda nowa klasa (Pure fabrication) będzie pośredniczyć w komunikacji między innymi klasami*
- *W wielu wypadkach jednak dodatkowa klasa (Pure Fabrication) jest dodawana do modelu właśnie po to żeby pośredniczyć między innymi klasami (Indirection)*
- *Klasy typu Indirection są silnie sprzężone z tymi klasami których używają i całkowicie wolne od sprzężeń z klasami które będą z nich korzystać*
- *Klasy typu Pure-fabrication powinny charakteryzować się dużą wewnętrzną spójnością*
- *Pojawienie się klasy typu Pure-fabrication wymaga pewnego wyczucia, ponieważ zgodnie z zasadą Information Expert często naturalnym kandydatem do realizacji jakiejś funkcjonalności jest jakaś istniejąca klasa.*

  ***Przykład**: jest w modelu dziedzinowym klasa **Order**, która, oprócz informacji o zamówieniu, potrzebuje również funkcji składowania danych w trwałym repozytorium (np. w bazie danych).*

  *Zgodnie z zasadą Information Expert, za funkcjonalność składowania do repozytorium powinna odpowiadać sama klasa **Order**. Taki projekt systemu oznacza jednak niższą spójność (naruszenie zasady High Cohesion) – w samej klasie pojawiają się bowiem dodatkowe odpowiedzialności, nie związane bezpośrednio z informacjami o zamówieniach. Kolejny minus to związanie klasy **Order** z szczegółami technicznymi operacji składowania, które zgodnie z zasadą Indirection można umieścić w obiekcie pośredniczącym (w tym przypadku: między **Order** a szczegółami jego zapisu do repozytorium).*

  *Wyniesienie takiej odpowiedzialności do dodatkowej klasy **OrderRepository** ma więc wiele zalet – obie klasy, **Order** i **OrderRepository** mają już wysoką spójność, a klasa **OrderRepository** jako Pure Fabrication spełnia również postulat Indirection.*

2.9 Protected Variations (Law of Demeter)

Nazwa	**Protected Variations**
Problem	Jak projektować obiekty, by ich zmiennić nie wywierała szkodliwego wpływu na inne elementy?
Rozwiązanie	Rozpoznaj punkty zmienności o otocz je stabilnym interfejsem.

Uwaga: *ta zasada zwana jest także* **Law of Demeter** (*prawo Demeter*). *Formułuje się je jako „nie rozmawiaj z nieznajomymi”. Co ciekawe, nie ma ono żadnego związku z mitologiczną Demeter, a tylko z projektem naukowym prowadzonym na Northeastern University pod koniec lat 90-tych ubiegłego wieku, nazwanym Project Demeter, którego celem było wypracowanie od podstaw dobrych zasad projektowania aspektowego. Więcej tu: http://www.bradapp.com/docs/demeter-intro.html*

3 SOLID

S	SRP	Single Responsibility Principle *Klasa ma tylko jedną odpowiedzialność*
O	OCP	Open-Closed Principle *otwarty na rozszerzenia, zamknięty na modyfikacje*
L	LSP	Liskov Substitution Principle *Każda obiekt klasy w kontekście swojego użycia powinien być zastępowalny przez obiekt klasy potomnej*
I	ISP	Interface Segregation Principle *Klient nie powinien być zmuszany do zależności od metod, których nie używa*
D	DIP	Dependency Inversion Principle *Moduły wyższego poziomu zależą od abstrakcji, nie od implementacji*

3.1 SRP, Single Responsibility Principle

Żadna klasa nie może być modyfikowana z więcej niż jednego powodu

3.1.1 Test odpowiedzialności SRP

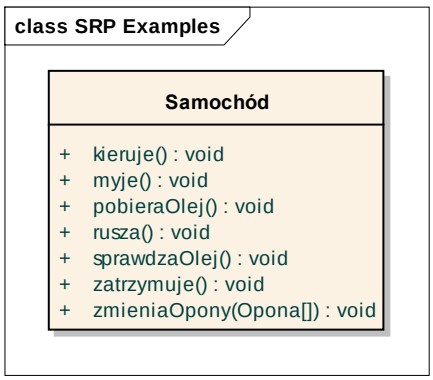
Naruszenie czasem możliwe do wykrycia za pomocą tzw. *testu odpowiedzialności SRP.*

Analiza SRP klasy **XXXX**

XXXX _____ sam.
XXXX _____ sam.
XXXX _____ sam.

3.1.2 Przykład.

Klasa samochód.



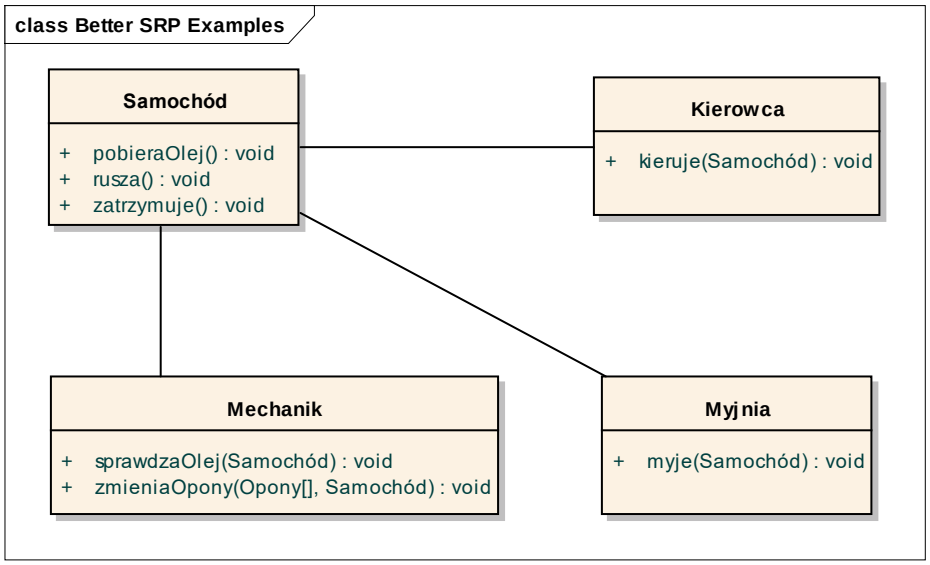
Test SRP:

Analiza SRP klasy **Samochód**

+ Samochód rusza sam.
+ Samochód zatrzymuje się sam.
? Samochód zmienia opony sam.
? Samochód kieruje sam.

? Samochód myje sam.
? Samochód sprawdza olej sam.
+ Samochód pobiera olej sam.

Poprawiony model:



3.2 Open-closed Principle

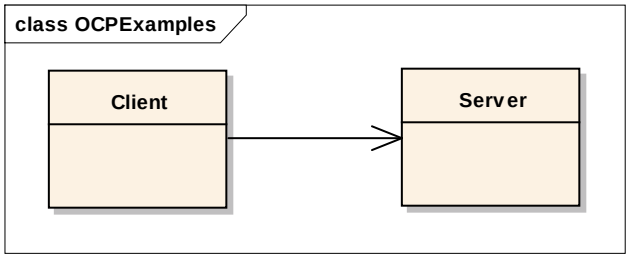
(=Protected Variations)

Składniki oprogramowania (klasy, moduły) powinny być otwarte (na rozszerzenia, adaptowalne) i zamknięte (na modyfikacje wpływające na klientów)

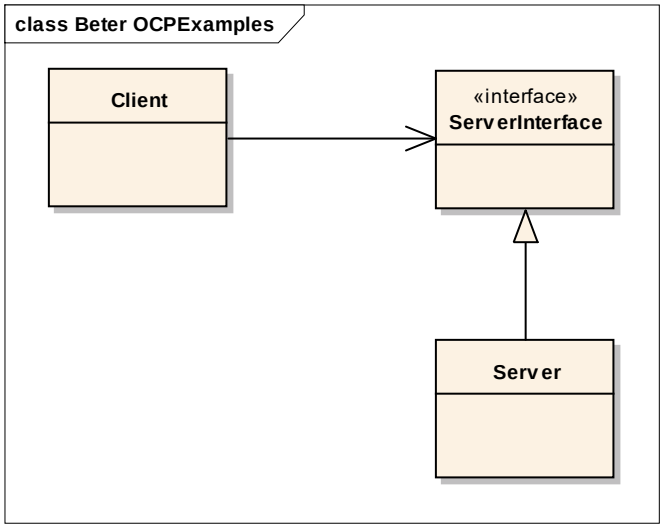
Typowe sposoby radzenia sobie:

- uzależnienie od **abstrakcji** zamiast od konkretnej implementacji (Dependency Inversion)
- wykorzystanie polimorfizmu (Visitor, Interpreter, State itd.)
- wymienność implementacji (Delegate Factory, Strategy, Template Method itd.)

3.2.1 Przykład



Poprawiony diagram



3.3 Liskov Substitution Principle

= zasada dobrego dziedziczenia

Musi istnieć możliwość zastępowania typów bazowych ich podtypami (również w kontekście semantycznym: poprawności działania programu, a nie tylko syntaktycznym: program się skompilował)

3.3.1 Przykład opisowy

Załóżmy że mamy funkcję *f* przyjmującą parametr typu *A* (to bez znaczenia w jakiej klasie znajduje się *f*)

```
class A { ... }

function f( a: A ) { ...

var a = new A();
f( a ); // prawidłowe
```

W każdym języku obiektowym poprawne syntaktycznie jest zastąpienie typu jego podtypem, w miejscu użycia:

```
class A { ... }
class B : A { ... }

function f( a: A ) { ...

var b = new B();
f( b ); // prawidłowe (syntaktycznie)
```

W takiej sytuacji jak powyżej, jeżeli przekazanie do funkcji *f* parametru typu *B* dziedziczącego z *A* powoduje błędne działanie *f*, to powiemy że ***B* narusza zasadę LSP, *B* jest wrażliwa na LSP w kontekście *f***.

Projektant funkcji *f* mógłby w jej implementacji testować argument na *bycie B* i uzależnić od tego implementację (naprawiając problem naruszenia LSP), ale naruszyłby wtedy OCP (!).

Remedium: taka sytuacja zwykle oznacza, że mamy *pozorne* dziedziczenie, a tak naprawdę klasy nie są w oczywisty sposób zależne relacją dziedziczenia. Być może na przykład są w hierarchii dziedziczenia swoim rodzeństwem – są potomkami jednego, tego samego typu bazowego. Inna możliwość: być może oczekiwania *f* w stosunku do obiektów A są zbyt duże, należy rozważyć ich modyfikację.

Aby nie łamać LSP należy trzymać się następującej zasady

Zasada chroniąca przed naruszeniem LSP: wolno **osłabić warunek wejścia** (precondition) lub **wzmocnić warunek wyjścia** (postcondition) w przeciążanych metodach.

Warunek wejścia i **warunek wyjścia** to formalne predykaty logiczne, opisujące więzy spełniane przez parametry metody (warunek wejścia) lub wartość zwracaną z metody (warunek wyjścia):

- Warunek wejścia – musi być spełniony przy wywołaniu metody (jeśli nie jest, metoda nie powinna się wykonywać), jest to wymaganie dla klienta metody odnoszące się do prawidłowości przekazania parametrów
- Warunek wyjścia – jest spełniony na końcu wykonywania metody, jest gwarancją dostawcy metody odnoszącym się do wartości zwracanej

Przykład:

```
function Abs( x: number ) {
    if ( x <= 0 )
       return -x;
    else {
       return x;
    }
}
```

Warunek wejścia: *true*
Warunek wyjścia: *RETVAL >= 0*

Przykład:

```
class ValueHolder {
   value: number;
}
```

```
function Abs( vh: ValueHolder ) {
    if ( vh.value <= 0 )
       return -vh.value;
    else {
       return vh.value;
    }
}
```

Warunek wejścia: *vh != null*
Warunek wyjścia: *RETVAL >= 0*

Zasada chroniąca przed naruszeniem LSP ta ma następujące uzasadnienie: zastępując wystąpienie obiektu wystąpieniem innego obiektu typu potomnego, klient chce mieć gwarancję że:

- Każda metoda którą do tej pory mógł wywołać z klasy bazowej, nadal da się wywołać z klasy potomnej (nie spowoduje błędu). Mówiąc nieformalnie – jeśli były wymagania na argumenty wywołania metody, to wymagania mogą się co najwyżej „poluzować".
- Każda metoda z klasy bazowej zwracająca wyniki zwraca z klasy potomnej nadal co najmniej „tak dobre" wyniki (a możliwe że „lepsze")

Przypomnienie: interpretacja pojęcia „osłabić" i „wzmonić" dotyczy zależności logicznej między predykatami. Dla dwóch predykatów, *p* i *q*, jeśli *p* => *q*, to mówimy że *p* jest silniejsze od *q*.

Przykład:

false (najsilniejszy) => x > 5 && y > 1 => y > 1 => y > 1 || y < 0 => *true* (najsłabszy)

3.3.2 Przykład na żywo – Coffee/DecafCoffee

Naruszenie zasady wzmocnienia warunku wyjścia.

```
/// <summary>
/// Zawartość kofeiny w kawie
/// </summary>
public class CaffeineContent
{
    public int Content { get; set; }
    public bool Strong { get { return Content > 50; } }
}

/// <summary>
/// Kawa (ma kofeinę)
/// </summary>
public interface ICoffee
{
    CaffeineContent CaffeineContent { get; }
}


/// <summary>
```

```
/// Kawa irlandzka
/// </summary>
public class IrishCoffee : ICoffee
{

    public CaffeineContent CaffeineContent
    {
        get { return new CaffeineContent() { Content = 100 }; }
    }

}

/// <summary>
/// Ups, kawa bezkofeinowa
/// </summary>
public class DecafCoffee : ICoffee
{

    public CaffeineContent CaffeineContent
    {
        get { return null; }
    }

}

/// <summary>
/// Klient, w kontekście którego pojawia się naruszenie LSP
/// </summary>
public class CoffeeClient
{
    public void BuyCoffee(ICoffee coffee)
    {
        // tu jest problem !!!
        if (coffee.CaffeineContent.Strong)
            Console.WriteLine("good");
        else
            Console.WriteLine("I need stronger");
    }
}
```

Warunek wyjścia metody w klasie potomnej został **osłabiony** (ponieważ zwraca się obiekt pusty którego klient się nie spodziewa). Kod kliencki przestaje działać.

Formalne uzasadnienie złamania LSP:

- warunek wyjścia dla klasy bazowej B jest: Ret(B) = *result* != null
- warunek wyjścia dla klasy potomnej C jest: Ret(C) = *result* != null || *result* == null

Między tymi dwoma warukami zachodzi zależność Ret(B) => Ret(C). Warunek wyjścia został osłabiony w klasie potomnej – jest to niezgodne z wymaganiem (warunek wyjścia można tylko wzmocnić).

Dobrym przykładem warunku wyjścia w klasie potomnej byłby warunek **silniejszy**, na przykład taki w którym nie tylko zwraca się niepusty obiekt, ale mówi się coś więcej o tym obiekcie, na przykład Ret(C) = *result* != null && *result*.*Content* > 50 (kawa mocna).

Wtedy istotnie Ret(C) => Ret(B).

3.3.3 Przykład na żywo – List/Set

Naruszenie zasady osłabienia warunku wejścia.

Wzmocniliśmy warunek wejścia ponieważ wymagamy aby nowo dodawany element był różny od wszystkich poprzednio dodanych. Kod kliencki przestaje działać.

3.4 Interface Segregation Principle

Klient nie powinien być zmuszany do zależności od metod których nie używa

Problem dużych interfejsów:

- Klasa potrzebuje tylko części funkcjonalności, a musi implementować cały interfejs
- Zmiana innej części interfejsu wymusza zmianę klasy i wszystkich jej klientów

3.5 Dependency Inversion Principle

Moduły wysokiego poziomu nie powinny zależeć od modułów niskiego poziomu tylko od abstrakcji.
Abstrakcje nie powinny zależeć od szczegółowych rozwiązań.

Zależność od szczegółów powoduje małą elastyczność.

4 Inne

4.1 Don’t Repeat Yourself (DRY)

Zalecenie unikania powtórzeń – kodu, odpowiedzialności.

4.2 Law of Demeter (LoD) (Principle of least knowledge)

= Protected Variations

4.3 (Don’t Make Me Think) DMMT

4.4 (Don’t Optimize Prematurely) DOP

4.5 Inne

http://msdn.microsoft.com/en-us/library/ee658124.aspx

http://www.artima.com/weblogs/viewpost.jsp?thread=331531