SOLID:

S - Single Responsibility Pattern (SRP)

O – Open-Closed Principle (OCP)

L – Liskov Substitution Principle (LSP)

I – Interface Segregation Principle (ISP)

D – Dependency Inversion Principle (DIP)

**SRP**

Single class should have just a single reason to change.

Single class should have one primary responsibility.

Nigdy nie powinno być więcej niż jednego powodu do modyfikacji klasy.

Przykład:

Klasa EmailSender ma więcej niż jedną odpowiedzialność. To znaczy, że istnieje więcej niż jeden powód, by w przyszłości zmienić tę klasę. Po pierwsze ów klasa odpowiada za wysyłanie maili, oprócz tego odpowiada jeszcze za dodawanie do statystyk, a także logowania informacji o błędzie. Powodów do zmiany jest stanowczo za dużo. Jeżeli klasa wysyła maila, to powinna robić tylko to, na pewno nie powinna oprócz tego jeszcze logować błędu i prowadzić statystyk. Czyli ten przykład ewidentnie łamie zasadę pojedynczej odpowiedzialności. No dobrze, ale jak zatem powinien wyglądać ten kod, stosując zasadę pojedynczej odpowiedzialności?

Przede wszystkim trzeba podzielić każdą odpowiedzialność do osobnej klasy. Czyli w tym przypadku, trzeba utworzyć 3 klasy, będą to:

-klasa do wysyłania maili,

-klasa do logowania błędów,

-klasa to prowadzenia statystyk.

//--------

@Override

public String toString(){

return String.join (System.lineSeparator(), myArrayList);

}

**OCP**

Element oprogramowania powinien być otwarty na rozbudowę, ale zamknięty na modyfikację.

Klasa powinna być zamknięta na modyfikacje tego, co już istnieje, ale otwarta na rozszerzenia, czyli mamy jakąś metodę, która jest używana w innych częściach systemu, więc nie powinniśmy jej już modyfikować, bo może coś zepsuć w innych miejscach systemu. Jeżeli mamy jakichś konsumentów naszych systemów, to taka zmiana może im trochę namieszać, ponieważ jeżeli będziemy sobie zmieniać metody to może brakować kompatybilności wstecznej z naszymi metodami, które już udostępniamy.

OPEN for extensions, CLOSED for modifications

Przykład:

Polegać na abstrakcji.

Wzorzec Specification

ZŁY PRZYKŁAD:

(bo co jeśli dojdzie dodatkowy typ? – musimy modyfikować)

public enum LogType {

File, Database

}

public class Logger {

private readonly LogType \_logType;

public Logger(LogType logType) {

\_logType = logType;

}

public void Log(string message) {

switch (\_logType) {

case LogType.File:

throw new NotImplementedException();

case LogType.Database:

throw new NotImplementedException();

default:

throw new Exception("Unexpected log type!");

}

}

}

DOBRY PRZYKŁAD:

public interface IMessageLogger {

void Log(string message);

}

public class Logger {

private readonly IMessageLogger \_messageLogger;

public Logger(IMessageLogger messageLogger) {

\_messageLogger = messageLogger;

}

public void Log(string message) {

\_messageLogger.Log(message);

}

}

+ Klasy implementujące interfejs IMessageLogger – w każdym z nich metoda ‘Log()’

public class FileLogger : IMessageLogger {

public void Log(string message) {

throw new NotImplementedException();

}

}

//------

public Stream<Product> filteProductsListByColor(List<Product> products, Color color) {

return products.stream().filter(p -> p.color)

}

//-

List<Product> products = List.of(prod1, prod2, prod3)

//--

pf.FilterByColor(products, Color.GREEN).forEach(p -> System.out.println(p.name))

**LSP**

"Funkcje, które używają wskaźników lub referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych, bez dokładnej znajomości tych obiektów".

Krócej mówiąc, w miejsce typu bazowego, możesz podstawić dowolny typ klasy pochodnej i nie powinieneś utracić poprawnego działania.

Zasada podstawienia Liskov jest powiązana z zasadą otwarte-zamknięte, ponieważ dzięki możliwości zastępowania podtypów, mamy możliwość rozbudowy klas bez konieczności ich modyfikowania. Aby dziedziczenie było dobre, klasy pochodne nie powinny nadpisywać metod klas bazowych. Natomiast można je rozszerzyć, poprzez wywołanie metody z klasy bazowej, czyli klasa pochodna powinna rozszerzać klasę bazową bez wpływania na jej działanie.

Jak mówi definicja funkcje, które używają referencji do wskaźników klas bazowych, powinny również mieć możliwość użycia funkcji klas pochodnych bez znajomości tego obiektu. Czyli można powiedzieć, że zasada podstawień Liskov między innymi sprowadza się do zakazu zadawania pytania o typ obiektu.

Implementując w poprawny sposób zasadę podstawienia Liskov, nie powinniśmy się posługiwać żadnym konstrukcjami warunkowymi, aby wymusić poprawne działanie, Obiekt pochodny musi z logicznego punktu widzenia być szczególnym przypadkiem obiektu bazowego. Musisz zawsze pamiętać, że możesz podstawić dowolny obiekt pochodny w miejsce obiektu bazowego i nie możesz zadawać pytania o to, jakiej klasy jest obiekt.

Przykład:

public interface IDuck {

void Swim();

bool IsSwimming { get; }

}

public class OrganicDuck : IDuck {

private bool \_isSwimming = false;

public void Swim() {

Console.WriteLine("OrganicDuck swims");

\_isSwimming = true;

}

public bool IsSwimming { get { return \_isSwimming; } }

}

public class ElectricDuck : IDuck {

private bool \_isSwimming;

public void Swim() {

if (!IsTurnedOn) return;

Console.WriteLine("ElectricDuck swims");

\_isSwimming = true;

}

public bool IsTurnedOn { get; set; }

public bool IsSwimming { get { return \_isSwimming; } }

}

public class Program {

static void Main() {

var ducks = new List();

IDuck organicDuck = new OrganicDuck();

IDuck electricDuck = new ElectricDuck();

ducks.Add(organicDuck);

ducks.Add(electricDuck);

MakeDuckSwim(ducks); //OrganicDuck swims

}

private static void MakeDuckSwim(IEnumerable ducks) {

foreach (var duck in ducks)

duck.Swim();

}

}

Obie kaczki implementują interfejs IDuck. Zawierają metody Swim, lecz po wywołaniu tej metody kaczka elektryczna nie pływa, ponieważ nie została wcześniej włączona. Jeżeli chciałbyś "naprawić" ten kod, tak aby obie kaczki pływały, musiałbyś zmodyfikować metodę MakeDuckSwim i zrobić osobną logikę w tej metodzie, gdy kaczka jest typu ElectricDuck. Taka zmiana oczywiście też jest zła i narusza zasadę LSP oraz OCP (klasa nie jest zamknięta na modyfikacje). Podsumowując, zdecydowanie mamy w tym wypadku zaprojektowaną złą abstrakcję.

**ISP**

Żaden klient nie powinien być zmuszany do polegania na metodach, z których nie korzysta. Nie powinniśmy stosować tłustych interfejsów (fat interfaces), mające deklarację metod, które niekoniecznie każdy z konsumentów tego interfejsu powinien implementować. Kilka dedykowanych interfejsów jest lepsze niż jeden, który jest zbyt ogólny.

Zły przykład:

public interface IReportable {

void GeneratePdf();

void GenerateExcel();

}

public class SalaryReport : IReportable {

public void GeneratePdf() {

//Code to generate pdf report

}

public void GenerateExcel() {

//Code to generate excel report

}

}

public class Invoice : IReportable {

public void GeneratePdf() {

//Code to generate pdf report

}

public void GenerateExcel() {

throw new NotImplementedException();

}

}

Dobry przykład:

public interface IReportablePdf {

void GeneratePdf();

}

public interface IReportableExcel {

void GenerateExcel();

}

**DIP**

Dependency Inversion Principle <==> Zasada Odwrócenia Zależności.

- Moduły wysokopoziomowe nie powinny zależeć od modułów niskopoziomowych.

I jedne, i drugie powinny zależeć od abstrakcji.

- Abstrakcje nie powinny zależeć od szczegółów. To szczegóły powinny zależeć od abstrakcji.

Systemy, które piszemy, powinny odnosić się do abstrakcji (poprzez zastosowanie interfejsów lub klas abstrakcyjnych), a nie konkretnych elementów. Dzięki stosowaniu się do zasady odwrócenia zależności minimalizujemy zależności (loose coupling) w aplikacji. Dzięki temu nasze aplikacje będą bardziej elastyczne, a w przyszłości zmiany będą łatwiejsze do wprowadzania.

Powinniśmy zmniejszać zależności do konkretnych implementacji. Najlepiej uzyskać to polegając na interfejsach, wtedy mamy w kodzie małe zależności. Interfejsy są stabilne. To znaczy, jeżeli zrobimy jakąś zmianę w interfejsie, to ta zmiana jest powiązana również, ze zmianą w implementacjach tego interfejsu. Jeżeli jednak, zrobimy zmianę w konkretnej implementacji, to zazwyczaj nie potrzebujemy zmieniać naszego interfejsu. Co za tym idzie, interfejsy są stabilniejsze od implementacji. Dodatkowo pisząc kod oparty na interfejsach, staje się on bardziej testowalny. Możesz z łatwością pisać do takiego kodu testy jednostkowe, ponieważ za interfejs możesz podstawić jakąś fake'ową implementację (mock'a).

Kod niestosujący się do zasady odwrócenia zależności:

public class Employee {

public string Name { get; set; }

}

public class EmployeeRepository {

public void Add(Employee employee) {

//Add employee to database

}

}

public class EmployeeService {

private EmployeeRepository \_employeeRepository = new EmployeeRepository();

public void Add(Employee employee) {

\_employeeRepository.Add(employee);

}

}

W powyższym kodzie modułem wysokopoziomowym jest klasa EmployeeService, a modułem niskopoziomowym klasa EmployeeRepository. Moduł wysokopoziomowy w tym przykładzie zależy od modułu niskopoziomowego, ponieważ używa konkretnej implementacji, czyli klasy EmployeeRepository, a to właśnie jest łamaniem zasady DIP. Aby ten kod był zgodny z DIP, musimy właśnie odwrócić te zależności (jak mówi sama nazwa). Jak to najlepiej zrobić? Implementacja gdzie zależności będą odwrócone może wyglądać na przykład tak jak na przykładzie poniżej.

Kod stosujący się do zasady odwrócenia zależności:

public class Employee {

public string Name { get; set; }

}

public interface IEmployeeRepository {

void Add(Employee employee);

}

public class EmployeeRepository : IEmployeeRepository {

public void Add(Employee employee) {

//Add employee to database

}

}

public class EmployeeService {

private IEmployeeRepository \_employeeRepository;

public EmployeeService(IEmployeeRepository employeeRepository) {

\_employeeRepository = employeeRepository;

}

public void Add(Employee employee) {

\_employeeRepository.Add(employee);

}

}

W powyższym przykładzie nasz serwis EmployeeService nie zależy już od konkretnej implementacji EmployeeRepository, a jedynie zależy od abstrakcji. W tym przypadku od interfejsu IEmployeeRepository. Zmiany w module niskopoziomowym nie mają wpływu na moduł wysokopoziomowy. Udało się w ten sposób odwrócić zależności. Teraz zgodnie z zasadą Dependency Inversion Principle, moduł wysokopoziomowy nie zależy już od modułu niskopoziomowego, a zależy od abstrakcji. Wysokopoziomowy moduł nie interesują zmiany, które zostaną wprowadzone w module niskopoziomowym. Ważne jest tylko to, aby implementowała powyższy interfejs.