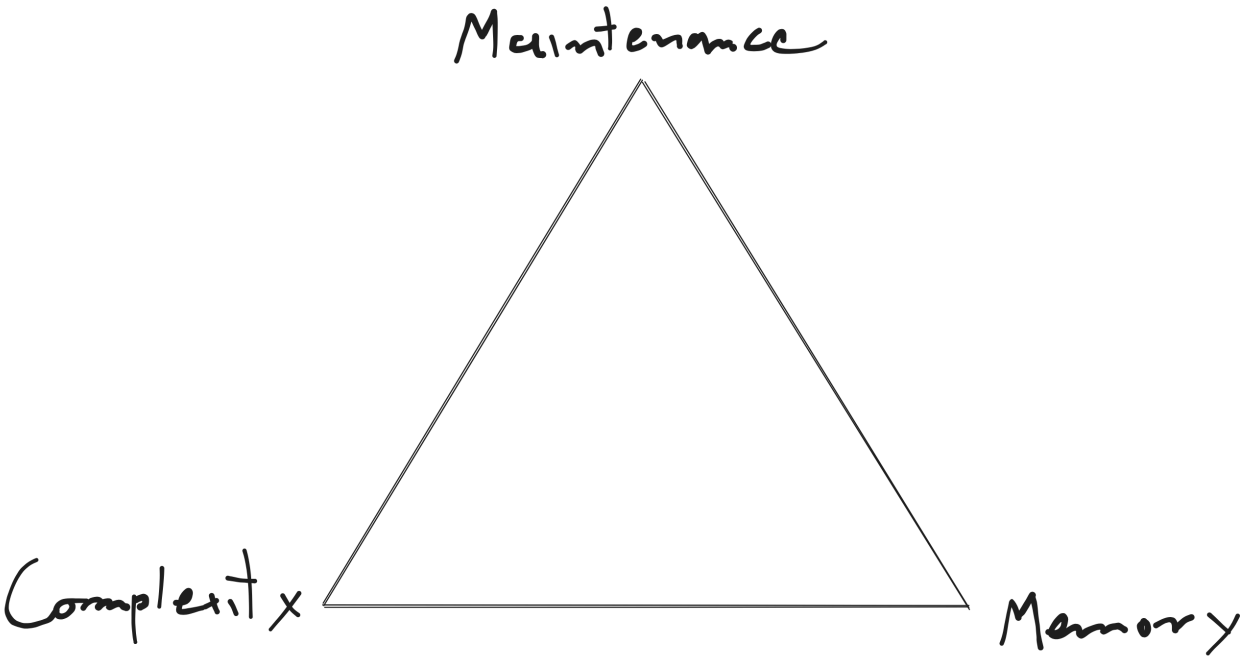
1. Jakie są dobre praktyki dotyczące jakości kodu?

- Jak możemy ocenić, czy nasza aplikacja jest utrzymywalna?
- Jakie znamy dobre praktyki programistyczne?

2. Jak możemy porównywać różne algorytmy?

Załóżmy, że mamy dwa algorytmy - oba wykonują ten sam rodzaj zadania. Jak ocenilibyśmy, które rozwiązanie jest lepsze?



Złożoność cyklomatyczna to metryka oprogramowania używana do pomiaru złożoności programu.

Określa ona liczbę liniowo niezależnych ścieżek przez kod źródłowy programu, zasadniczo licząc punkty decyzyjne. Opracowana przez Thomasa J. McCabe'a w 1976 roku, złożoność cyklomatyczna pomaga w ocenie złożoności programu na podstawie struktury jego przepływu sterowania.

Idea polega na tym, że bardziej złożony kod, który ma większą liczbę **punktów decyzyjnych** (takich jak instrukcje if, pętle i instrukcje case), jest trudniejszy do zrozumienia, testowania i utrzymania. Złożoność cyklomatyczna jest obliczana przy użyciu wzoru opartego na liczbie krawędzi i węzłów w grafie przepływu sterowania programu, wraz z liczbą połączonych komponentów:

```
Złożoność cyklomatyczna = E − N + 2P
− E to liczba krawędzi w grafie.
− N to liczba węzłów w grafie.
− P to liczba połączonych komponentów (zwykle P=1 dla większości programów).
```

Przykładowy kod:

```
public int Calculate(int a, int b, string operation)
{
    int result = 0;
    if (operation == "add")
    {
        result = a + b;
    }
    else if (operation == "subtract")
    {
        result = a - b;
    }
    else if (operation == "multiply")
    {
        result = a * b;
    }
    else
    {
        Console.WriteLine("Invalid operation");
    }
    return result;
}
```

4. Spójność i powiązanie

Spójność i powiązanie są fundamentalnymi zasadami, które wpływają na jakość architektury na każdym poziomie - od poszczególnych metod po komponenty, moduły i całe systemy.

4.1. Spójność (ang. Cohesion)

Spójność mierzy, jak bardzo element oprogramowania jest skoncentrowany na jednym celu lub odpowiedzialności. Wysoka spójność oznacza, że:

- Każdy element robi jedną rzecz dobrze
- Funkcje w module są ze sobą ściśle powiązane
- Komponenty mają jasno określone, zdefiniowane obowiązki

Wysoka spójność tworzy kod łatwy w utrzymaniu, wielokrotnego użytku i zrozumiały. Gdy komponent ma pojedynczą, jasno zdefiniowaną odpowiedzialność, łatwiej o nim rozumować, testować go i modyfikować bez nieoczekiwanych skutków ubocznych.

"Próba podzielenia spójnego modułu skutkowałaby tylko zwiększonym powiązaniem i zmniejszoną czytelnością" - Larry Constantine

4.1.1. Różne rodzaje spójności

Spójność funkcjonalna

Opis: Najwyższa forma spójności. Każda część modułu przyczynia się do realizacji pojedynczego, dobrze zdefiniowanego zadania, a moduł zawiera wszystko, co niezbędne do wykonania tej funkcji.

Przykład: Klasa walidacji hasła, która tylko waliduje hasła.

```
public class PasswordValidator
{
    public bool IsValid(string password)
    {
        if (string.IsNullOrEmpty(password))
            return false;

        bool hasMinimumLength = password.Length >= 8;
        bool hasUpperCase = password.Any(char.IsUpper);
        bool hasLowerCase = password.Any(char.IsLower);
        bool hasDigit = password.Any(char.IsDigit);
        bool hasSpecialChar = password.Any(c => !char.IsLetterOrDigit(c));

        return hasMinimumLength && hasUpperCase && hasLowerCase && hasDigit && hasSpecialChar;
    }
}
```

Spójność sekwencyjna

Opis: Moduły wchodzą w interakcje, gdzie wyjście jednego staje się wejściem dla drugiego, tworząc sekwencję powiązanych operacji.

Przykład: Potok przetwarzania danych, gdzie każdy krok opiera się na poprzednim.

```
public class OrderProcessor
{
    public OrderConfirmation ProcessOrder(OrderRequest request)
    {
        // Each step uses output from the previous step
        var validatedOrder = ValidateOrder(request);
        var pricedOrder = CalculatePricing(validatedOrder);
        var paymentResult = ProcessPayment(pricedOrder);
        var fulfillmentOrder = CreateFulfillmentOrder(pricedOrder, paymentResult);

        return GenerateConfirmation(fulfillmentOrder);
    }

    private ValidatedOrder ValidateOrder(OrderRequest request) { /* ... */ }
    private PricedOrder CalculatePricing(ValidatedOrder order) { /* ... */ }
    private PaymentResult ProcessPayment(PricedOrder order) { /* ... */ }
    private FulfillmentOrder CreateFulfillmentOrder(PricedOrder order, PaymentResult payment) { /* ... */
}
    private OrderConfirmation GenerateConfirmation(FulfillmentOrder order) { /* ... */ }
}
```

Spójność komunikacyjna

Opis: Moduły operują na tych samych danych lub przyczyniają się do wspólnego wyniku, tworząc łańcuch komunikacji.

Przykład: System rejestracji użytkowników, który zarówno zapisuje dane użytkownika, jak i wysyła e-mail powitalny.

```
public class UserRegistrationService
{
    private readonly IUserRepository _userRepository;
    private readonly IEmailService _emailService;
```

```
    public UserRegistrationService(IUserRepository userRepository, IEmailService emailService)
    {
        _userRepository = userRepository;
        _emailService = emailService;
    }

    public RegistrationResult RegisterUser(UserData userData)
    {
        // Both operations work with the same userData
        User savedUser = _userRepository.Create(userData);
        _emailService.SendWelcomeEmail(userData.Email, userData.Name);

        return new RegistrationResult
        {
            UserId = savedUser.Id,
            Status = "Success",
            Message = "Registration complete. Welcome email sent."
        };
    }
}
```

Spójność proceduralna

Opis: Moduły muszą być wykonywane w określonej kolejności, nawet jeśli mogą wykonywać różne funkcje.

Przykład: Sekwencja uruchamiania aplikacji.

```
public class ApplicationInitializer
{
    public void Initialize()
    {
        // These steps must happen in this specific order
        ConfigureLogging();
        LoadConfiguration();
        InitializeDatabase();
        StartServices();
        RegisterEventHandlers();
    }

    private void ConfigureLogging() { /* ... */ }
    private void LoadConfiguration() { /* ... */ }
    private void InitializeDatabase() { /* ... */ }
    private void StartServices() { /* ... */ }
    private void RegisterEventHandlers() { /* ... */ }
}
```

Spójność czasowa

Opis: Moduły są pogrupowane, ponieważ są wykonywane w tym samym okresie, nawet jeśli funkcjonalnie nie są powiązane.

Przykład: Zadania uruchamiane podczas zamykania aplikacji.

```
public class ApplicationShutdown
{
    public void PerformShutdownSequence()
    {
        // These operations happen during shutdown but are functionally different
        SaveUserPreferences();
        CloseNetworkConnections();
        FlushCaches();
        BackupInMemoryData();
        ReleaseResources();
    }

    private void SaveUserPreferences() { /* ... */ }
    private void CloseNetworkConnections() { /* ... */ }
    private void FlushCaches() { /* ... */ }
```

```
    private void BackupInMemoryData() { /* ... */ }
    private void ReleaseResources() { /* ... */ }
}
```

Spójność logiczna

Opis: Moduły są pogrupowane, ponieważ wykonują podobne operacje logiczne, nawet jeśli operują na różnych danych lub dla różnych celów.

Przykład: Klasa użytkowa z metodami manipulacji ciągami znaków.

```
public static class StringUtils
{
    public static string Capitalize(string input)
    {
        if (string.IsNullOrEmpty(input))
            return input;

        return char.ToUpper(input[0]) + input.Substring(1);
    }

    public static string Reverse(string input)
    {
        if (string.IsNullOrEmpty(input))
            return input;

        char[] chars = input.ToCharArray();
        Array.Reverse(chars);
        return new string(chars);
    }

    public static int CountWords(string input)
    {
        if (string.IsNullOrWhiteSpace(input))
            return 0;

        return input.Split(new[] { ' ', '\t', '\n' }, StringSplitOptions.RemoveEmptyEntries).Length;
    }
}
```

Spójność przypadkowa

Opis: Najniższa forma spójności. Elementy w module są niepowiązane, z wyjątkiem tego, że znajdują się w tym samym pliku źródłowym - nie istnieje między nimi znaczące połączenie.

Przykład: Różnorodna klasa użytkowa z niepowiązanymi funkcjami.

```
public static class Miscellaneous
{
    public static double CalculateTax(double amount, double rate)
    {
        return amount * rate;
    }

    public static string FormatDate(DateTime date)
    {
        return date.ToString("yyyy-MM-dd");
    }

    public static void ShutdownSystem()
    {
        Environment.Exit(0);
    }

    public static List<string> ReadFileLines(string filePath)
    {
        return File.ReadAllLines(filePath).ToList();
```

```
    }

    public static Color InvertColor(Color original)
    {
        return Color.FromArgb(
            original.A,
            255 - original.R,
            255 - original.G,
            255 - original.B);
    }
}
```

4.1.2. Spójność - przykład 2

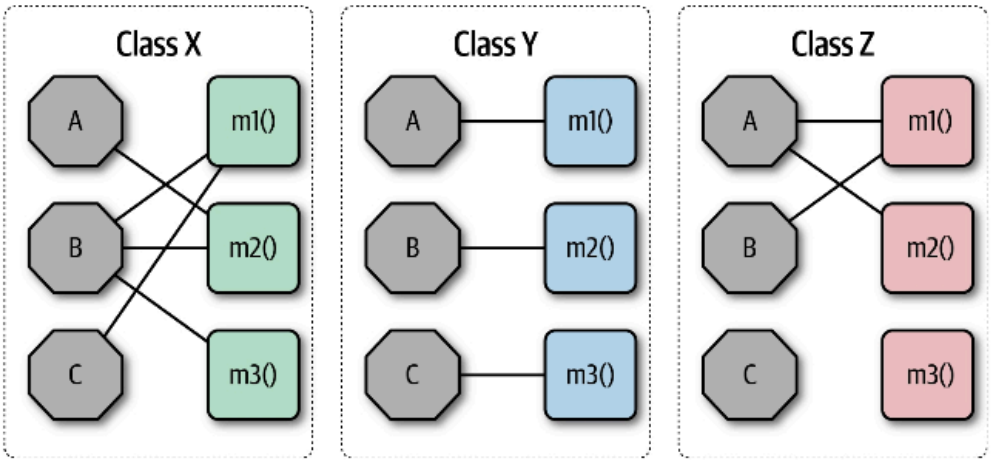


Figure 3-1. Illustration of the LCOM metric, where fields are octagons and methods are squares

Obraz ilustruje metrykę LCOM (Brak Spójności Metod) w trzech różnych projektach klas.

Na tym diagramie:

- Ośmiokąty (A, B, C) reprezentują pola/atrybuty klasy
- Kwadraty (m1(), m2(), m3()) reprezentują metody
- Linie wskazują, które metody używają których pól

Porównanie trzech projektów:

Klasa X

Pokazuje metody z wieloma połączeniami krzyżowymi do pól. Każda metoda potencjalnie używa wielu pól, a każde pole jest używane przez wiele metod. Sugeruje to wysoką współzależność między polami i metodami, wskazując na dobrą spójność.

Klasa Y

Pokazuje bezpośrednią relację jeden-do-jednego między polami a metodami. Każda metoda używa dokładnie jednego pola, a każde pole jest używane przez dokładnie jedną metodę. Sugeruje to, że metody i pola są sparowane, ale nie współdzielą danych w klasie.

Klasa Z

Pokazuje wzorzec, w którym metody mają nakładający się, ale nie pełny dostęp do pól. Występuje pewne współdzielenie pól między metodami, ale nie pełne połączenie widoczne w klasie X.

Interpretacja LCOM:

- Klasa X: Miałaby najniższą wartość LCOM (najlepsza spójność), ponieważ jej metody współdzielą wiele pól klasy, wskazując na wspólną pracę nad powiązaną funkcjonalnością.
- Klasa Y: Miałaby najwyższą wartość LCOM (najgorsza spójność), ponieważ każda metoda operuje na własnym polu, sugerując, że mogłyby to potencjalnie być oddzielne klasy.
- Klasa Z: Miałaby umiarkowaną wartość LCOM, ponieważ występuje pewne współdzielenie pól, ale nie pełne połączenie.

Spójność - wzór

Brak Spójności Metod

$$LCOM96b = \frac{1}{a}\sum_{j=1}^{a}\frac{m - \mu(Aj)}{m}$$

https://en.wikipedia.org/wiki/Cohesion_(computer_science)
https://www.ndepend.com/docs/code-metrics#LCOM

Ten obraz pokazuje wzór matematyczny dla LCOM96b (określonej wersji metryki Braku Spójności Metod). Wyjaśnię, co oznacza ten wzór:

Metryka LCOM96b określa ilościowo, jak spójna jest klasa, mierząc, jak metody w klasie współdzielą atrybuty lub pola. Wzór to:

$$LCOM96b = (1/a) \times \Sigma [m - \mu(Aj)] / m$$

Gdzie:

- a to liczba atrybutów w klasie
- m to liczba metod w klasie
- $\mu(Aj)$ reprezentuje liczbę metod, które uzyskują dostęp do atrybutu j
- Sumowanie przebiega przez wszystkie atrybuty (j=1 do a)

Mówiąc prościej, ten wzór:

1. Dla każdego atrybutu oblicza, ile metod NIE używa go
2. Uśrednia te wartości dla wszystkich atrybutów
3. Normalizuje wynik

Wyższa wartość LCOM wskazuje na niższą spójność (metody nie współdzielą wielu atrybutów), sugerując, że klasa może wykonywać zbyt wiele niepowiązanych zadań i mogłaby być kandydatem do refaktoryzacji na wiele klas.

Niższa wartość LCOM wskazuje na wyższą spójność (metody współdzielą wiele atrybutów), sugerując, że klasa ma skoncentrowaną, pojedynczą odpowiedzialność z dobrze zintegrowaną funkcjonalnością.

Ta metryka pomaga programistom identyfikować klasy, które mogą naruszać Zasadę Pojedynczej Odpowiedzialności i mogłyby skorzystać z podziału na bardziej spójne jednostki.

4.2. Powiązanie (ang. Coupling)

Powiązanie odnosi się do tego, jak bardzo jeden moduł (jak klasa lub funkcja) wie lub polega na innym module. Wyobraź sobie, że ty i przyjaciel pracujecie razem nad układanką.

Jeśli pracujecie nad całkowicie różnymi częściami układanki bez potrzeby czegokolwiek od siebie nawzajem, jesteście jak dwa moduły z **niskim powiązaniem** - nie musisz wiedzieć, co robi druga osoba, aby robić postępy. Ale jeśli musisz ciągle pytać przyjaciela o elementy lub informacje, to jest to jak **wysokie powiązanie**; twoja praca jest mocno zależna od pracy twojego przyjaciela.

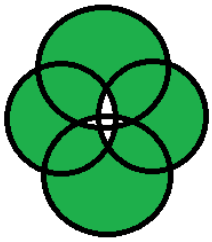
Niskie powiązanie jest generalnie lepsze, ponieważ oznacza, że zmiany w jednej części kodu nie spowodują efektu domina wymagającego zmian w wielu innych częściach. To jak możliwość pracy nad swoją częścią układanki niezależnie, bez ciągłej potrzeby rzeczy od przyjaciela.

To wyjaśnienie używa analogii układanki, aby zilustrować powiązanie w projektowaniu oprogramowania:

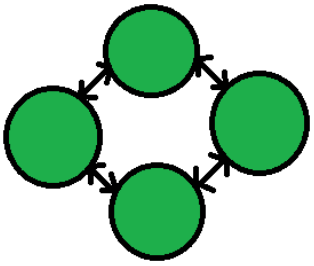
1. **Wysokie powiązanie** oznacza, że komponenty są wysoce współzależne. Gdy moduły są ściśle powiązane, zmiany w jednym module często wymagają zmian w innych, co sprawia, że system jest trudniejszy w utrzymaniu, testowaniu i zrozumieniu.
2. **Niskie powiązanie** oznacza, że komponenty są w dużej mierze niezależne. Gdy moduły są luźno powiązane, współdziałają przez dobrze zdefiniowane interfejsy bez potrzeby znajomości wewnętrznych szczegółów, co sprawia, że system jest bardziej elastyczny i łatwiejszy w utrzymaniu.

Osiągnięcie niskiego powiązania jest fundamentalnym celem w architekturze oprogramowania, ponieważ poprawia modularność, testowalność oraz umożliwia równoległy rozwój i łatwiejszą konserwację.

4.2.1. Wizualizacja powiązań



Tight coupling:
1. More Interdependency
2. More coordination
3. More information flow



Loose coupling:
1. Less Interdependency
2. Less coordination
3. Less information flow

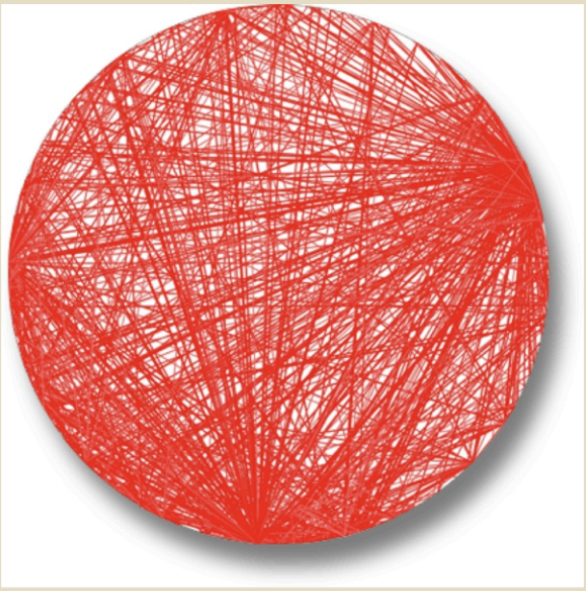


Figure 9-1. A Big Ball of Mud architecture visualized from a real code base

4.3. Powiązanie Eferentne i Aferentne

Powiązanie Eferentne (Zależności wychodzące): Dotyczy tego, ile różnych modułów zależy od twojego modułu. Jeśli twoja część układanki (moduł) wymaga elementów z wielu innych sekcji (innych modułów), to ma wysokie powiązanie eferentne. To jak powiedzenie: "Potrzebuję wielu rzeczy od innych, aby ukończyć moją pracę."

Powiązanie Aferentne (Zależności przychodzące): To przeciwieństwo; dotyczy tego, ile różnych modułów zależy od twojego modułu. Jeśli wiele innych sekcji układanki potrzebuje elementów, które tylko ty masz, twoja sekcja ma wysokie powiązanie aferentne. To jak bycie popularnym na imprezie, ponieważ masz coś, czego wszyscy potrzebują.

Równoważenie powiązania eferentnego i aferentnego jest ważne. Zbyt wysokie powiązanie eferentne oznacza, że twój moduł jest zbyt zależny od innych, co może utrudniać zmiany.

Wysokie powiązanie aferentne oznacza, że twój moduł jest kluczowy dla aplikacji, co sugeruje, że zapewnia istotną funkcjonalność, ale także wskazuje na potencjalne ryzyko, jeśli potrzebne są zmiany, ponieważ wiele części aplikacji polega na nim.

4.4. Abstrakcyjność

Abstrakcyjność to stosunek abstrakcyjnych artefaktów (klasy abstrakcyjne, interfejsy itp.) do konkretnych artefaktów (implementacji) w kodzie lub module. Reprezentuje miarę abstrakcyjności względem implementacji.

Idealny poziom abstrakcji różni się w zależności od celu modułu w systemie:

- **Niska abstrakcyjność**: Głównie konkretne implementacje z niewieloma abstrakcjami
- **Wysoka abstrakcyjność**: Głównie klasy abstrakcyjne i interfejsy z mniejszą liczbą konkretnych implementacji
- **Zrównoważona abstrakcyjność**: Odpowiednia mieszanka dla roli modułu

Znalezienie właściwej równowagi jest kluczowe. Zbyt duża konkretność może prowadzić do sztywnego, trudnego do rozszerzenia kodu. Jednak zbyt duża abstrakcyjność może tworzyć niepotrzebną złożoność i pośrednictwo.

$$A = \frac{\sum m^a}{\sum m^c}$$

Przykład projektu z zbyt dużą abstrakcyjnością:
https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition

4.5 Niestabilność

Niestabilność to metryka, która określa zmienność kodu. Kod, który wykazuje wysoki stopień niestabilności, łatwiej się psuje przy zmianach ze względu na wysokie powiązanie.

$$I = \frac{C^e}{C^e + C^a}$$

Gdzie:

- Ce = Powiązanie Eferentne (zależności wychodzące)
- Ca = Powiązanie Aferentne (zależności przychodzące)

Daje to wartość między 0 a 1:

- I = 0: Całkowicie stabilny (brak zależności wychodzących)
- I = 1: Całkowicie niestabilny (tylko zależności wychodzące)

Formuła ujawnia ważne spostrzeżenia architektoniczne:

- Moduły, od których zależy wiele innych komponentów (wysokie Ca), powinny być bardziej stabilne
- Moduły, które zależą od wielu innych komponentów (wysokie Ce), są z natury bardziej niestabilne
- Zmiany w niestabilnych modułach zwykle mają zlokalizowany wpływ
- Zmiany w stabilnych modułach mogą kaskadowo wpływać na cały system

Utrzymanie odpowiednich poziomów niestabilności jest kluczowe dla utrzymywalności systemu:

- Kluczowe komponenty infrastruktury powinny mieć niską niestabilność (bliżej 0)
- Komponenty UI lub szybko zmieniające się funkcje mogą mieć wyższą niestabilność (bliżej 1)
- Unikanie komponentów zarówno z wysokim powiązaniem aferentnym, jak i eferentnym

4.6. Odległość od Głównej Sekwencji

Odległość od Głównej Sekwencji to holistyczna miara oparta na niestabilności i abstrakcyjności. Zarówno abstrakcyjność, jak i niestabilność są ułamkami, których wyniki zawsze będą mieścić się w zakresie od 0 do 1.

Ta metryka mierzy, jak daleko komponent jest od idealnej równowagi między abstrakcyjnością a niestabilnością.

"Główna Sekwencja" reprezentuje tę idealną równowagę:

$$D = | A + I - 1 |$$

Gdzie:

- D = Odległość od Głównej Sekwencji
- A = Abstrakcyjność (ułamek elementów abstrakcyjnych)
- I = Niestabilność (stosunek powiązania eferentnego do całkowitego powiązania)

Idealna relacja sugeruje, że:

- Stabilne komponenty (I → 0) powinny być abstrakcyjne (A → 1)
- Niestabilne komponenty (I → 1) powinny być konkretne (A → 0)

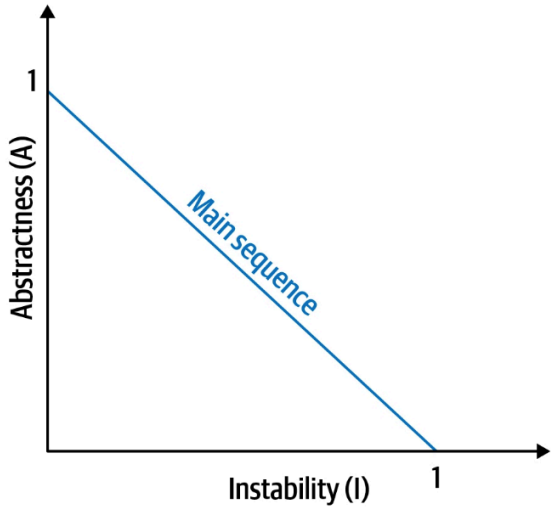


Figure 3-2. The main sequence defines the ideal relationship between abstractness and instability

Na wykresie przedstawiającym abstrakcyjność (oś y) względem niestabilności (oś x):

- Główna Sekwencja to linia od (0,1) do (1,0)
- Komponenty powinny idealnie znajdować się blisko tej linii
- Im dalej komponent jest od tej linii, tym bardziej prawdopodobne, że cierpi na problemy projektowe

Istnieją dwie problematyczne strefy:

1. Strefa Bólu: Komponenty, które są wysoce stabilne, ale nie abstrakcyjne (0,0)
    - Trudne do zmiany, ale brakuje im elastyczności abstrakcji
    - Często reprezentują konkretne klasy narzędziowe lub infrastrukturalne
2. Strefa Bezużyteczności: Komponenty, które są wysoce abstrakcyjne, ale niestabilne (1,1)
    - Abstrakcyjne bez zależnych
    - Często reprezentują nadmiernie projektowane lub nieużywane abstrakcje

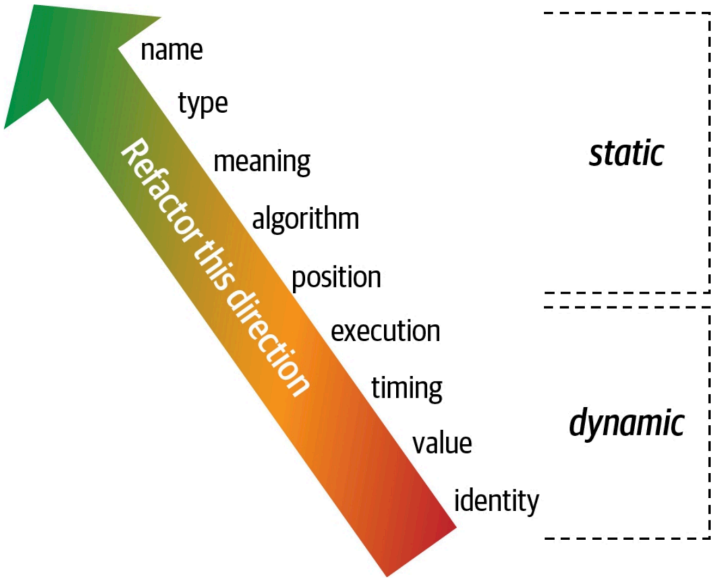
Komponenty z niską wartością odległości wykazują zdrową mieszankę abstrakcyjności i niestabilności odpowiednią dla ich roli w systemie, co czyni je zarówno utrzymywalnymi, jak i użytecznymi.

5. Connascence (Współzależność)

W 1996 roku Meilir Page-Jones opublikował "What Every Programmer Should Know About Object-Oriented Design", udoskonalając metryki powiązania aferentnego i eferentnego oraz przekształcając je dla języków zorientowanych obiektowo za pomocą koncepcji nazwanej connascence.

"Dwa komponenty są współzależne, jeśli zmiana w jednym wymagałaby modyfikacji drugiego, aby utrzymać ogólną poprawność systemu."
https://connascence.io/

Różne rodzaje współzależności:



5.1. Powiązanie vs Współzależność

Powiązanie jest szeroką miarą współzależności między modułami, zazwyczaj postrzegając połączenia jako binarne (powiązane lub niepowiązane).

Współzależność to bardziej wyrafinowana koncepcja, która:

- Zapewnia stopniowanie siły powiązania
- Rozróżnia między różnymi typami zależności
- Oferuje konkretne słownictwo do omówienia różnych relacji powiązań
- Pomaga priorytetyzować, które zależności należy adresować najpierw

5.2. Typy Współzależności z przykładami w C#

Statyczna Współzależność (czas kompilacji)

Współzależność Nazw

- **Definicja**: Gdy wiele komponentów musi zgadzać się co do nazwy encji (metody, klasy, zmiennej)
- **Wpływ**: Zmiany nazw wymagają zmian we wszystkich komponentach odwołujących się

```
// Obie klasy muszą zgadzać się co do nazwy metody "ProcessPayment"
public class OrderService {
    private PaymentGateway _paymentGateway;

    public void Checkout(Order order) {
        _paymentGateway.ProcessPayment(order.Total);
    }
}

public class PaymentGateway {
    // Jeśli nazwa tej metody się zmieni, OrderService przestanie działać
    public void ProcessPayment(decimal amount) {
        // Logika przetwarzania płatności
    }
}
```

Współzależność Typu

- **Definicja**: Gdy wiele komponentów musi zgadzać się co do typu encji
- **Wpływ**: Zmiany typu kaskadowo wpływają na wszystkie komponenty używające tego typu

```
// Obie klasy muszą zgadzać się co do typu parametru (decimal)
public class ShippingCalculator {
    public decimal Calculate(decimal weight) {
        return weight * 2.5m;
    }
}

public class OrderProcessor {
    private ShippingCalculator _calculator;

    public void Process(Order order) {
        // Przestanie działać, jeśli Calculate zostanie zmienione na przyjmowanie double zamiast decimal
        var shippingCost = _calculator.Calculate(order.Weight);
    }
}
```

Współzależność Znaczenia

- **Definicja**: Gdy wiele komponentów musi zgadzać się co do znaczenia konkretnych wartości
- **Wpływ**: Ukryte znaczenie tworzy ukryte zależności i niejasny kod

```
// Obie klasy muszą zgadzać się, co oznacza wartość 0 w polu statusu
public class OrderUpdater {
    public void UpdateStatus(Order order, int newStatus) {
        if (newStatus == 0) { // 0 oznacza "oczekujące"
            order.IsPending = true;
        }
    }
}

public class OrderDisplay {
    public string GetStatusLabel(Order order) {
        // Musi zgadzać się, że 0 oznacza "oczekujące"
        if (order.Status == 0) {
            return "Oczekujące";
        }
        return "Inne";
    }
}
```

Współzależność Pozycji

- Definicja: Gdy wiele komponentów musi zgadzać się co do kolejności wartości
- Wpływ: Zmiana kolejności parametrów wpływa na wszystkich wywołujących

```
// Zarówno wywołujący, jak i metoda muszą zgadzać się co do kolejności parametrów
public class UserService {
    // Parametry to (firstName, lastName, email)
    public void RegisterUser(string firstName, string lastName, string email) {
        // Logika rejestracji
    }
}

public class RegistrationController {
    private UserService _userService;

    public void RegisterUser(UserDto dto) {
        // Musi przekazać parametry w poprawnej kolejności
        _userService.RegisterUser(dto.FirstName, dto.LastName, dto.Email);
        // Jeśli UserService zmieni kolejność parametrów, to przestanie działać
    }
}
```

Współzależność Algorytmu

- Definicja: Gdy wiele komponentów musi zgadzać się co do konkretnego algorytmu
- Wpływ: Zmiany algorytmu wymagają skoordynowanych aktualizacji w komponentach

```
// Obie klasy muszą używać tego samego algorytmu haszowania
public class PasswordHasher {
    public string HashPassword(string password) {
        // Używanie SHA256
        using (var sha256 = System.Security.Cryptography.SHA256.Create()) {
            var hashedBytes = sha256.ComputeHash(System.Text.Encoding.UTF8.GetBytes(password));
            return Convert.ToBase64String(hashedBytes);
        }
    }
}

public class UserAuthenticator {
    private PasswordHasher _hasher;

    public bool VerifyPassword(string storedHash, string attemptedPassword) {
        // Musi używać tego samego algorytmu co PasswordHasher
        string attemptedHash = _hasher.HashPassword(attemptedPassword);
        return storedHash == attemptedHash;
```

```
    }
}
```

Dynamiczna Współzależność (czas wykonania)

Współzależność Wykonania

- Definicja: Gdy kolejność wykonania wielu komponentów jest ważna
- Wpływ: Nieprawidłowa kolejność wykonania powoduje błędy w czasie wykonania

```
// Komponenty muszą być wykonane w prawidłowej kolejności
public class DatabaseConnection {
    public void Open() { /* Otwórz połączenie */ }
    public void Close() { /* Zamknij połączenie */ }

    public void ExecuteQuery(string query) {
        // Wymaga wcześniejszego otwarcia połączenia
        // I powinno być później zamknięte
    }
}

public class DataRepository {
    private DatabaseConnection _connection;

    public IEnumerable<Customer> GetCustomers() {
        _connection.Open(); // Musi nastąpić przed ExecuteQuery
        var results = _connection.ExecuteQuery("SELECT * FROM Customers");
        _connection.Close(); // Musi nastąpić po ExecuteQuery
        return MapResults(results);
    }
}
```

Współzależność Czasu

- Definicja: Gdy czas wykonania wielu komponentów jest ważny
- Wpływ: Warunki wyścigu i błędy zależne od czasu

```
// Komponenty muszą koordynować czas operacji
public class CachingService {
    private Dictionary<string, object> _cache = new Dictionary<string, object>();

    public void InvalidateCache(string key) {
        _cache.Remove(key);
    }

    public object GetItem(string key) {
        return _cache.ContainsKey(key) ? _cache[key] : null;
    }
}

public class ProductService {
    private CachingService _cache;

    public void UpdateProduct(Product product) {
        SaveToDatabase(product);
        // Musi nastąpić po zapisie, ale przed próbą odczytu zaktualizowanego produktu
        _cache.InvalidateCache($"product_{product.Id}");
    }
}
```

Współzależność Wartości

- Definicja: Gdy wiele wartości odnosi się do siebie w wymagany sposób
- Wpływ: Przerwanie relacji między wartościami powoduje nieprawidłowe zachowanie

```
// Wartości są od siebie zależne
public class Rectangle {
    public int Width { get; set; }
    public int Height { get; set; }

    public int Area => Width * Height;
}

public class RectangleValidator {
    public bool IsValid(Rectangle rectangle) {
        // Powierzchnia musi być równa szerokość × wysokość
        return rectangle.Area == rectangle.Width * rectangle.Height;
    }
}
```

Współzależność Tożsamości

- Definicja: Gdy wiele komponentów musi odwoływać się do tej samej instancji encji
- Wpływ: Używanie różnych instancji, gdy wymagana jest ta sama, powoduje błędy

```
// Wiele komponentów musi odwoływać się do tej samej instancji encji
public class ShoppingCart {
    private List<CartItem> _items = new List<CartItem>();

    public void AddItem(Product product, int quantity) {
        // Musi sprawdzić, czy dokładnie ta sama instancja produktu już istnieje
        var existingItem = _items.FirstOrDefault(i => ReferenceEquals(i.Product, product));
        if (existingItem != null) {
            existingItem.Quantity += quantity;
        } else {
            _items.Add(new CartItem { Product = product, Quantity = quantity });
        }
    }
}

public class CheckoutService {
    public decimal CalculateTotal(ShoppingCart cart) {
        // Musi odwoływać się do tych samych instancji produktów co w koszyku
        return cart.Items.Sum(i => i.Quantity * i.Product.Price);
    }
}
```

5.3. Podsumowanie

Właściwości Współzależności

Wpływ współzależności określają trzy kluczowe właściwości:

1. **Siła**: Jak trudno jest ją zidentyfikować i rozwiązać (statyczna jest zazwyczaj słabsza niż dynamiczna)
2. **Lokalność**: Współzależność między blisko powiązanymi komponentami jest mniej problematyczna
3. **Stopień**: Liczba objętych komponentów

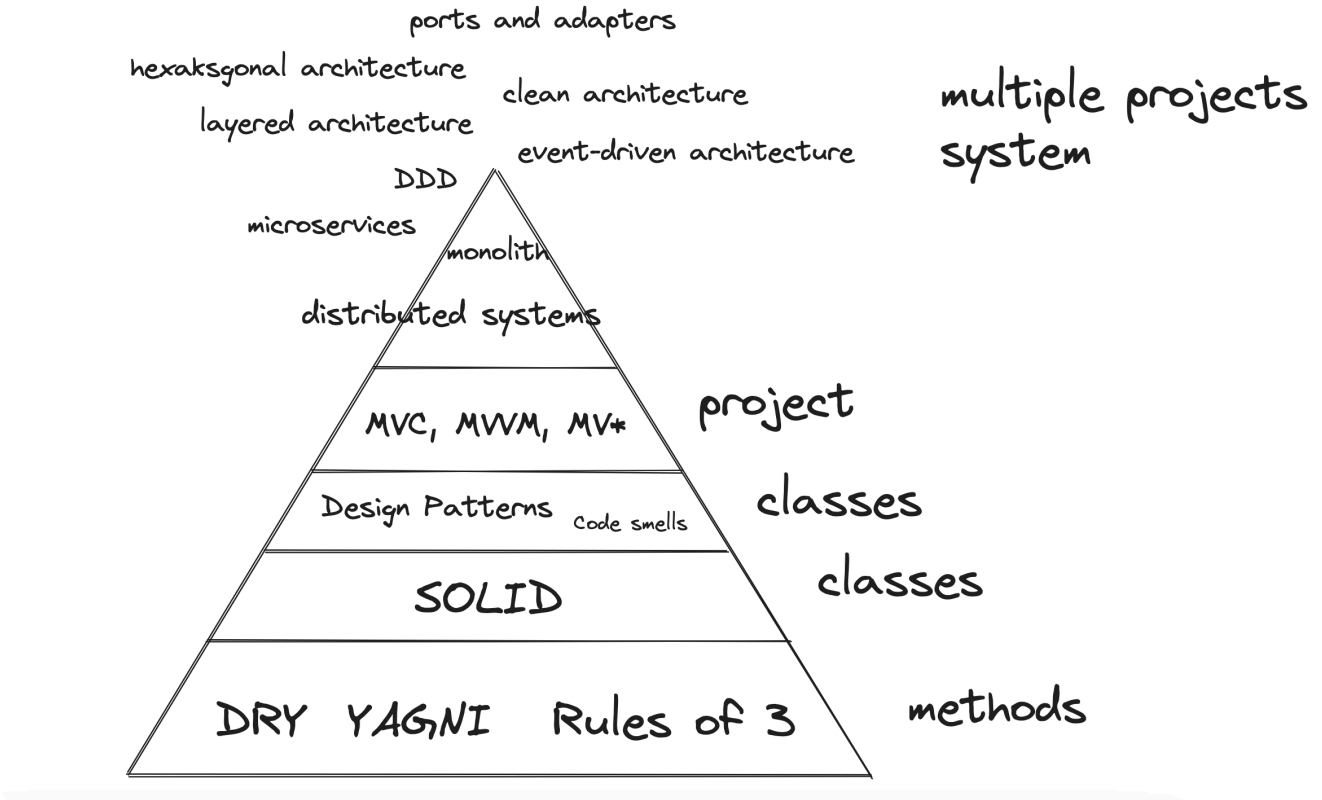
Wytyczne projektowe

Framework współzależności sugeruje następujące usprawnienia projektowe:

- **Minimalizuj ogólną współzależność** poprzez zmniejszanie powiązań
- **Przekształcaj silniejsze współzależności w słabsze formy** (np. przekształcaj zależności czasu wykonania na zależności czasu kompilacji)
- **Lokalizuj współzależność w granicach enkapsulacji**, gdzie to możliwe

6. Bottom up - od metod do klas i architektury

Jakie są typowe dobre praktyki, o których słyszałeś podczas programowania?



Kod vs architektura Architektura oprogramowania odnosi się do fundamentalnych struktur systemu oprogramowania oraz dyscypliny tworzenia takich struktur i systemów. Służy jako plan zarówno dla systemu, jak i projektu go rozwijającego, definiując:

- Jak komponenty są zorganizowane i połączone
- Jak komunikują się ze sobą
- Ograniczenia i zasady kierujące ich projektowaniem i ewolucją
- Decyzje techniczne odpowiadające celom biznesowym i atrybutom jakości
- Wytyczne do rozwijania, wdrażania i utrzymania systemu

Architektura ustanawia fundament, na którym budowane są wszystkie decyzje dotyczące rozwoju, i znacząco wpływa na **skalowalność, wydajność, utrzymywalność i bezpieczeństwo.**

Projektujemy nie pod kątem obecnych wymagań, ale także aby przygotować się na przyszłe zmiany.

Wzorce projektowe mogą znacząco usprawnić proces rozwoju, zapewniając ustandaryzowany język wśród programistów i framework do efektywnego rozwiązywania złożonych problemów projektowych. Służą jako narzędzie komunikacji, pozwalając zespołom dyskutować o wyrafinowanych projektach za pomocą zwięzłej, dobrze rozumianej terminologii.

Koncepcja wzorców projektowych w inżynierii oprogramowania została spopularyzowana przez "Gang of Four" (GoF), składający się z Ericha Gamma, Richarda Helma, Ralpha Johnsona i Johna Vlissidesa, w ich przełomowej książce "Design Patterns: Elements of Reusable Object-Oriented Software", opublikowanej w 1994 roku.

Wzorce projektowe reprezentują najlepsze praktyki, które ewoluowały z czasem, gdy programiści wielokrotnie napotykali i rozwiązywali podobne problemy. Korzystając z ustalonych wzorców, programiści mogą:

- Przyspieszyć proces rozwoju
- Stosować sprawdzone rozwiązania zamiast wynajdywać koło na nowo
- Tworzyć bardziej utrzymywalne i elastyczne struktury kodu
- Zmniejszyć ryzyko poważnych problemów w systemie
- Poprawić czytelność kodu dla osób znających wzorce

Choć wzorce projektowe dostarczają wartościowych rozwiązań, ważne jest, by stosować je rozsądnie i tylko wtedy, gdy odpowiednio adresują dany problem. Nadużywanie wzorców lub zmuszanie ich do scenariuszy, do których nie pasują, może prowadzić do niepotrzebnej złożoności.

https://www.amazon.pl/Design-patterns-elements-reusable-object-oriented/dp/0201633612/

Wzorce kreacyjne: Te wzorce zajmują się mechanizmami tworzenia obiektów, próbując tworzyć obiekty w sposób odpowiedni do sytuacji. Podstawowa forma tworzenia obiektów mogłaby prowadzić do problemów projektowych lub dodatkowej złożoności projektu. Kreacyjne wzorce projektowe rozwiązują ten problem poprzez kontrolowanie tego tworzenia obiektów. Przykładami są wzorce Singleton, Metoda Fabrykująca, Fabryka Abstrakcyjna, Builder i Prototyp.

Wzorce strukturalne: Te wzorce zajmują się kompozycją obiektów lub strukturą klas. Pomagają zapewnić, że jeśli jedna część systemu się zmienia, cały system nie musi robić tego samego. Pomagają również upewnić się, że zmiana w jednej części systemu nie wymaga zmian w innych częściach. Przykładami są wzorce Adapter, Most, Kompozyt, Dekorator, Fasada, Flyweight i Proxy.

Wzorce behawioralne: Te wzorce dotyczą komunikacji obiektów klasy. Pomagają definiować, jak obiekty wchodzą w interakcje w sposób zwiększający elastyczność w prowadzeniu komunikacji. Przykładami są wzorce Obserwator, Mediator, Polecenie, Stan, Strategia, Iterator i Visitor.

Ogólnie rzecz biorąc, wszystkie różne dobre praktyki starają się utrzymać spójność i powiązanie pod kontrolą.

Chcemy, aby nasz kod wykazywał: **"Wysoką spójność i niskie powiązanie"**

Te cechy kodu zwiększą również testowalność naszego kodu. Gdy komponenty mają jasne obowiązki (wysoka spójność) i minimalne zależności od innych komponentów (niskie powiązanie), naturalnie stają się łatwiejsze do izolacji w celach testowych. Prowadzi to do bardziej niezawodnych testów, lepszego pokrycia testami i prostszej konfiguracji testów.

Dobrze zaprojektowany kod z wysoką spójnością i niskim powiązaniem zwykle ma kilka korzystnych cech:

- Poprawiona utrzymywalność
- Zwiększona możliwość ponownego użycia
- Lepsza skalowalność
- Zmniejszona złożoność
- Większa elastyczność na zmiany
- Łatwiejsze debugowanie i rozwiązywanie problemów

Stosowanie zasad takich jak SOLID, wzorce projektowe i ustalone heurystyki, wszystko to przyczynia się do osiągnięcia tego fundamentalnego celu architektury oprogramowania.

SOLID to akronim reprezentujący pięć zasad projektowania i programowania obiektowego. Te zasady mają na celu sprawienie, by projekty oprogramowania były bardziej zrozumiałe, elastyczne i utrzymywalne.

Robert C. Martin jest uważany za twórcę tego akronimu.

- http://principles-wiki.net/collections:robert_c._martin_s_principle_collection
- https://blog.cleancoder.com/uncle-bob/2020/10/18/Solid-Relevance.html

9.1. Zasada Pojedynczej Odpowiedzialności (SRP)

Klasa powinna mieć tylko jeden powód do zmiany, co oznacza, że powinna mieć tylko jedno zadanie lub odpowiedzialność. Ta zasada pomaga uczynić system łatwiejszym do zrozumienia i utrzymania, ponieważ każda klasa koncentruje się na pojedynczej funkcjonalności.

```
// Bad: Class has multiple responsibilities
public class UserService
{
    public void RegisterUser(string email, string password)
    {
        // Validate input
        if (string.IsNullOrEmpty(email) || string.IsNullOrEmpty(password))
            throw new ArgumentException("Email and password are required");

        // Save to database
        SaveToDatabase(email, password);

        // Send email
        SendWelcomeEmail(email);
    }

    private void SaveToDatabase(string email, string password)
    {
        // Database code
    }

    private void SendWelcomeEmail(string email)
    {
        // Email sending code
    }
}

// Good: Responsibilities are separated
public class UserRegistrationService
{
    private readonly IUserRepository _userRepository;
    private readonly IEmailService _emailService;
    private readonly IValidator _validator;

    public UserRegistrationService(
        IUserRepository userRepository,
        IEmailService emailService,
        IValidator validator)
    {
        _userRepository = userRepository;
        _emailService = emailService;
        _validator = validator;
    }

    public void RegisterUser(string email, string password)
    {
        _validator.Validate(email, password);
        _userRepository.Save(email, password);
        _emailService.SendWelcomeEmail(email);
    }
}
```

9.2. Zasada Otwarte/Zamknięte (OCP)

Encje oprogramowania (klasy, moduły, funkcje itp.) powinny być otwarte na rozszerzenia, ale zamknięte na modyfikacje. Oznacza to, że powinno być możliwe dodanie nowej funkcjonalności do encji bez zmiany jej istniejącego kodu, co może pomóc w zmniejszeniu ryzyka wprowadzenia błędów do istniejącego kodu podczas dodawania nowych funkcji.

```
// Bad: Need to modify class to add new payment method
public class PaymentProcessor
{
    public void ProcessPayment(string paymentType, decimal amount)
    {
        if (paymentType == "CreditCard")
        {
            // Process credit card payment
        }
        else if (paymentType == "PayPal")
        {
            // Process PayPal payment
        }
        // Need to modify this class when adding a new payment method
    }
}

// Good: Open for extension, closed for modification
public abstract class PaymentMethod
{
    public abstract void ProcessPayment(decimal amount);
}

public class CreditCardPayment : PaymentMethod
{
    public override void ProcessPayment(decimal amount)
    {
        // Process credit card payment
    }
}

public class PayPalPayment : PaymentMethod
{
    public override void ProcessPayment(decimal amount)
    {
        // Process PayPal payment
    }
}

// Can add new payment methods without changing existing code
public class BitcoinPayment : PaymentMethod
{
    public override void ProcessPayment(decimal amount)
    {
        // Process Bitcoin payment
    }
}

public class PaymentProcessor
{
    public void ProcessPayment(PaymentMethod paymentMethod, decimal amount)
    {
        paymentMethod.ProcessPayment(amount);
    }
}
```

9.3. Liskov Substitution Principle (LSP)

Obiekty klasy nadrzędnej powinny być zastępowalne przez obiekty klasy podrzędnej bez wpływu na poprawność programu. Ta zasada zapewnia, że podklasa może zastąpić swoją nadklasę, prowadząc do bardziej solidnego i elastycznego kodu, szczególnie przy pracy z hierarchiami typów.

```
// Bad: Violates LSP because Square changes behavior of Rectangle
public class Rectangle
{
    public virtual int Width { get; set; }
    public virtual int Height { get; set; }

    public int CalculateArea()
    {
        return Width * Height;
    }
}

public class Square : Rectangle
{
    private int _size;

    public override int Width
    {
        get => _size;
        set
        {
            _size = value;
            Height = value; // Changing behavior of base class!
        }
    }

    public override int Height
    {
        get => _size;
        set
        {
            _size = value;
            Width = value; // Changing behavior of base class!
        }
    }
}

// Good: Does not violate LSP
public interface IShape
{
    int CalculateArea();
}

public class Rectangle : IShape
{
    public int Width { get; set; }
    public int Height { get; set; }

    public int CalculateArea()
    {
        return Width * Height;
    }
}

public class Square : IShape
{
    public int Size { get; set; }

    public int CalculateArea()
    {
        return Size * Size;
```

```
    }
}
```

9.4. Zasada Segregacji Interfejsów (ISP)

Żaden klient nie powinien być zmuszony do zależności od metod, których nie używa. Ta zasada zaleca tworzenie szczegółowych interfejsów, które są specyficzne dla klienta, zamiast jednego dużego, ogólnego interfejsu, poprawiając modularność systemu i zmniejszając wpływ zmian.

```
// Bad: Interface with methods that not all implementers use
public interface IMultiFunction
{
    void Print();
    void Scan();
    void Fax();
    void Copy();
}

public class AllInOnePrinter : IMultiFunction
{
    public void Copy() { /* ... */ }
    public void Fax() { /* ... */ }
    public void Print() { /* ... */ }
    public void Scan() { /* ... */ }
}

public class BasicPrinter : IMultiFunction
{
    public void Print() { /* ... */ }

    // Forced to implement methods it doesn't support
    public void Scan() { throw new NotSupportedException(); }
    public void Fax() { throw new NotSupportedException(); }
    public void Copy() { throw new NotSupportedException(); }
}

// Good: Segregated interfaces
public interface IPrinter
{
    void Print();
}

public interface IScanner
{
    void Scan();
}

public interface IFax
{
    void Fax();
}

public interface ICopier
{
    void Copy();
}

public class AllInOnePrinter : IPrinter, IScanner, IFax, ICopier
{
    public void Copy() { /* ... */ }
    public void Fax() { /* ... */ }
    public void Print() { /* ... */ }
    public void Scan() { /* ... */ }
}

public class BasicPrinter : IPrinter
{
```

```
    public void Print() { /* ... */ }
    // No need to implement unused methods
}
```

9.5. Zasada Odwrócenia Zależności (DIP)

Moduły wysokiego poziomu nie powinny zależeć od modułów niskiego poziomu. Oba powinny zależeć od abstrakcji. Ponadto, abstrakcje nie powinny zależeć od szczegółów; szczegóły powinny zależeć od abstrakcji. Ta zasada kieruje strukturą zależności w systemie, aby zmniejszyć powiązanie między modułami wysokiego i niskiego poziomu, czyniąc system łatwiejszym do refaktoryzacji, zmiany i ponownego wdrożenia.

```
// Bad: High-level module depends on low-level module
public class NotificationService
{
    private readonly EmailSender _emailSender;

    public NotificationService()
    {
        _emailSender = new EmailSender();
    }

    public void SendNotification(string email, string message)
    {
        _emailSender.SendEmail(email, message);
    }
}

public class EmailSender
{
    public void SendEmail(string email, string message)
    {
        // Send email implementation
    }
}

// Good: Both depend on abstraction
public interface IMessageSender
{
    void SendMessage(string to, string message);
}

public class EmailSender : IMessageSender
{
    public void SendMessage(string to, string message)
    {
        // Send email implementation
    }
}

public class SMSSender : IMessageSender
{
    public void SendMessage(string to, string message)
    {
        // Send SMS implementation
    }
}

public class NotificationService
{
    private readonly IMessageSender _messageSender;

    // Dependency is injected
    public NotificationService(IMessageSender messageSender)
    {
        _messageSender = messageSender;
    }

    public void SendNotification(string to, string message)
    {
        _messageSender.SendMessage(to, message);
```

```
    }
}
```

10. Heurystyki dla tworzenia klas

Korzystając z poniższych linków, znajdziesz dodatkowe heurystyki przydatne do tworzenia dobrze zaprojektowanych klas:

- https://www.amazon.com/Object-Oriented-Design-Heuristics-Arthur-Riel/dp/020163385X
- https://manclswx.com/talks/top_heuristics.html

Te zasoby rozszerzają zasady SOLID, dostarczając praktycznych wskazówek i reguł praktycznych dla projektowania obiektowego. Książka Arthura Riela "Object-Oriented Design Heuristics" zawiera ponad 60 wytycznych projektowych, które mogą pomóc programistom uniknąć typowych pułapek i tworzyć bardziej utrzymywalne systemy. Drugi link zawiera zwięzłe podsumowanie niektórych najważniejszych heurystyk z tego wpływowego dzieła.

11. Testowanie

Testowanie jest kluczową częścią projektowania oprogramowania. Istnieje wiele różnych technik testowania wykonywanych przez programistów lub testerów.

Możemy podzielić testy w oparciu o znajomość wewnętrznego działania:

- testy czarnoskrzynkowe
- testy białoskrzynkowe
- testy szaroskrzynkowe

Możemy podzielić testy na:

- testy ręczne
- testy automatyczne
  - testy jednostkowe
  - testy integracyjne
  - testy end-to-end

Możemy podzielić testy według typu testu:

- testy obciążeniowe
- testy wydajnościowe
- testy użyteczności
- testy kompatybilności
- testy regresyjne
- i wiele innych...

Testowanie jednostkowe to rodzaj automatycznego testowania, który koncentruje się na weryfikacji najmniejszych części systemu oprogramowania, znanych jako jednostki, w izolacji.

Jednostką może być pojedyncza funkcja, metoda, procedura, moduł lub obiekt. Głównym celem testowania jednostkowego jest zapewnienie, że każda jednostka oprogramowania działa zgodnie z projektem.

Testy jednostkowe są zazwyczaj pisane i utrzymywane przez programistów. Testy te są wykonywane często w trakcie procesu rozwoju, aby wcześnie wykrywać błędy, ułatwiać refaktoryzację kodu i zapewnić, że ostatnie zmiany nie wpłynęły negatywnie na istniejącą funkcjonalność.

Framework do testów jednostkowych, specyficzny dla używanego języka programowania (np. JUnit dla Javy, NUnit dla .NET lub PyTest dla Pythona), zapewnia ustrukturyzowany sposób definiowania przypadków testowych, wykonywania testów i raportowania wyników.

Charakterystyka testów jednostkowych

- **Izolacja**: Testy jednostkowe są zaprojektowane do testowania jednostek kodu w izolacji od reszty systemu. Jest to często osiągane za pomocą obiektów mock, zaślepek lub fałszywek do symulacji zachowania złożonych zależności, takich jak bazy danych, systemy plików lub usługi zewnętrzne.
- **Prostota**: Każdy test powinien koncentrować się na pojedynczym aspekcie zachowania jednostki, jasno wskazując, co jest testowane i jaki jest oczekiwany wynik.
- **Automatyzacja**: Testy jednostkowe są zautomatyzowane, co oznacza, że mogą być uruchamiane szybko i często bez ręcznej interwencji. Ta automatyzacja jest kluczowa dla potoku ciągłej integracji i ciągłego dostarczania (CI/CD).
- **Powtarzalność**: Test jednostkowy może być uruchamiany dowolną liczbę razy w dowolnym środowisku z oczekiwaniem tego samego wyniku za każdym razem, zakładając, że nie wprowadzono zmian w testowanej jednostce.

11.2. Dlaczego piszemy testy?

1. **Wczesne wykrywanie błędów i testowanie regresji**: Testy jednostkowe pomagają zidentyfikować błędy na wczesnym etapie procesu rozwoju. Testując poszczególne jednostki kodu niezależnie, programiści mogą wskazać błędy, zanim zostaną zintegrowane z większym systemem, gdzie są często trudniejsze i droższe do zdiagnozowania i naprawienia.
2. **Ułatwianie zmian**: Z kompleksowym zestawem testów jednostkowych, programiści mogą pewnie wprowadzać zmiany w kodzie, wiedząc, że testy ujawnią, czy ich zmiany nieumyślnie zepsuły istniejącą funkcjonalność. Jest to szczególnie wartościowe w większych projektach lub tych z wieloma współpracownikami, ponieważ pomaga utrzymać stabilność oprogramowania w czasie.
3. **Poprawa jakości kodu**: Proces pisania testów jednostkowych zachęca programistów do pisania czystszego, bardziej modularnego kodu. Jednostki kodu muszą być dobrze izolowane, aby były testowalne, co naturalnie prowadzi do bardziej modularnej architektury. To nie tylko ułatwia zrozumienie kodu, ale także ułatwia ponowne użycie kodu.
4. **Dokumentacja**: Testy jednostkowe służą jako forma żywej dokumentacji. Demonstrują, jak kod ma być używany i jakie jest jego oczekiwane zachowanie w różnych warunkach. Nowi programiści lub współpracownicy mogą spojrzeć na testy jednostkowe, aby szybko zapoznać się z kodem.
5. **Ułatwianie projektowania**: Pisanie testów często pomaga w procesie projektowania oprogramowania. Potrzeba utrzymania kodu testowalnego może prowadzić programistów do lepszych wyborów projektowych, zachęcając ich do myślenia o tym, jak komponenty współdziałają ze sobą i jak efektywnie enkapsulować zachowanie.
6. **Pewność refaktoryzacji**: Testy jednostkowe zapewniają siatkę bezpieczeństwa, która pozwala programistom refaktoryzować kod z pewnością. Refaktoryzacja to proces zmiany wewnętrznej struktury kodu bez zmiany jego zewnętrznego zachowania. Testy jednostkowe pomagają zapewnić, że wysiłki refaktoryzacyjne nie wprowadzą nowych błędów.
7. **Ciągła integracja (CI)**: Testy jednostkowe są integralne dla praktyk CI, gdzie zmiany kodu są automatycznie budowane, testowane i scalane do wspólnego repozytorium wiele razy dziennie. Solidny zestaw testów jednostkowych umożliwia szybką informację zwrotną na temat zmian kodu, zmniejszając problemy integracyjne i pozwalając na szybsze cykle rozwoju.

11.3. Nazewnictwo testów

Strategie nazewnictwa testów jednostkowych są kluczowe dla zapewnienia, że zestawy testów są łatwo zrozumiałe i utrzymywalne.

Dobre nazwy testów mogą szybko przekazać cel testu, warunki, w których jest uruchamiany, i jaki wynik jest oczekiwany. Oto kilka powszechnych strategii nazewnictwa testów jednostkowych, wraz z wyjaśnieniem, dlaczego nazwy testów często są dość długie:

1. NazwaMetody_StanTestowany_OczekiwaneZachowanie Ta strategia zaczyna się od nazwy testowanej metody, następnie stanu lub warunku, w którym jest testowana, i na końcu oczekiwanego zachowania lub wyniku. To podejście bardzo jasno pokazuje, jaki aspekt metody jest testowany i w jakich okolicznościach.
2. NazwaMetody_KiedyStanTestowany_PowinienOczekiwaneZachowanie Włączając słowo "powinien" do konwencji nazewnictwa testu jednostkowego, artykułujesz oczekiwane zachowanie lub wynik w sposób, który brzmi naturalnie i jasno określa, co testowany kod powinien robić w określonych warunkach.

Na przykład:

- calculateTotal_WhenDiscountApplied_ShouldReduceTotalAmount (obliczSumę_GdyZastosowanoRabat_PowinienZmniejszyćCałkowitąKwotę)

12. Refaktoryzacja

Refaktoryzacja to proces restrukturyzacji istniejącego kodu komputerowego - zmiany faktorowania lub "kształtu" kodu - **bez zmiany jego zewnętrznego zachowania**.

Celem refaktoryzacji jest poprawa projektu, struktury i/lub implementacji oprogramowania (jego niefunkcjonalnych atrybutów), przy zachowaniu jego funkcjonalności. Ta technika pozwala programistom oczyścić bazę kodu, czyniąc ją łatwiejszą do zrozumienia, utrzymania i rozszerzenia.

13. Dodatkowe linki

Wzorce projektowe
https://refactoring.guru/design-patterns

Refaktoryzacja
https://refactoring.guru/