Helion

Czysty kod w C#

Techniki refaktoryzacji i najlepsze praktyki

Packt›

Jason Alls

Tytuł oryginału: Clean Code in C#: Refactor your legacy C# code base and improve application performance by applying best practices

Tłumaczenie: Radosław Meryk

ISBN: 978-83-283-7725-7

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/czykoc.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/czykoc
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

1

ROZDZIAŁ

Standardy i zasady kodowania w języku C#

Podstawowym celem stosowania przez programistów standardów i zasad kodowania w języku C# jest dążenie do stania się lepszymi w swoim rzemiośle. Dzieje się tak dzięki programowaniu kodu, który jest bardziej wydajny i łatwiejszy w utrzymaniu. W tym rozdziale zajmę się kilkoma przykładami dobrego kodu, zestawiając go ze złym kodem. Omówię powody, dla których potrzebujemy standardów kodowania, zasad i metodologii. Następnie przejdę do rozważenia konwencji nazewnictwa, komentowania i formatowania kodu źródłowego, w tym klas, metod i zmiennych.

Zrozumienie i utrzymywanie dużego programu może być dość trudne. Poznanie kodu i zrozumienie co robi, może być dla początkujących programistów nie lada wyzwaniem. Współpraca zespołów w takich projektach może stwarzać wiele kłopotów. Duże programy są również z reguły trudne do testowania. Z tego powodu przyjrzymy się sposobom dzielenia programów na mniejsze fragmenty (moduły), które wspólnie tworzą w pełni działające rozwiązania. Moduły mogą być w pełni testowalne, pozwalają na jednoczesną pracę przez wiele zespołów oraz są znacznie łatwiejsze do czytania, zrozumienia i dokumentowania.

Rozdział zakończę wyszczególnieniem kilku wytycznych projektowania oprogramowania, głównie KISS, YAGNI, DRY, SOLID oraz Brzytwy Ockhama.

W niniejszym rozdziale zostaną omówione następujące tematy:

- Potrzeba stosowania standardów kodowania, zasad i metodologii.
- Konwencje nazewnictwa i sposoby ich egzekwowania.
- Komentarze i formatowanie.
- Modułowość.
- KISS.
- YAGNI.

- DRY.
- SOLID.
- Brzytwa Ockhama.

Po przeczytaniu niniejszego rozdziału:

- Zrozumiesz, dlaczego zły kod negatywnie wpływa na projekty.
- Będziesz wiedzieć, dlaczego dobry kod pozytywnie wpływa na projekty.
- Dowiesz się, że standardy kodowania poprawiają kod i nauczysz się egzekwować te standardy.
- Zrozumiesz, że zasady kodowania poprawiają jakość oprogramowania.
- Dowiesz się dlaczego stosowanie metodologii wspomaga rozwój czystego kodu.
- Nauczysz się stosować standardy kodowania.
- Dowiesz się, jak wybierać rozwiązania z jak najmniejszą liczbą założeń.
- Nauczysz się minimalizować dublowanie kodu i pisać kod zgodnie z zasadami SOLID.

Wymagania techniczne

Do pracy nad kodem w tym rozdziale trzeba pobrać i zainstalować środowisko Visual Studio 2019 Community Edition lub jego nowszą wersję. To środowisko IDE można pobrać pod adresem *https://visualstudio.microsoft.com/*.

Kod przykładów do tej książki znajduje się na stronie *https://github.com/PacktPublishing/Clean-Code-in-C-*. Umieściłem go w jednym rozwiązaniu, a każdy rozdział to osobny folder. Kod dla poszczególnych rozdziałów znajdziesz w odpowiednich folderach. Jeśli chcesz uruchomić jakiś projekt, pamiętaj, aby ustawić go jako projekt startowy.

Dobry kod kontra zły kod

Zarówno dobry, jak i zły kod kompilują się. To pierwsza rzecz, jaką należy zapamiętać. Należy także zrozumieć, że zły kod jest zły nie bez powodu, a dobry kod jest dobry również z określonych względów. Przyjrzyjmy się niektórym z tych powodów zestawionych w tabeli na następnej stronie.

To dość obszerna lista. W kolejnych punktach przyjrzymy się, jak te cechy wpływają na wydajność kodu, zwrócimy także uwagę na różnice pomiędzy dobrym, a złym kodem.

Dobry kod	Zły kod
Odpowiednie wcięcia.	Niewłaściwe wcięcia.
Sensowne komentarze.	Komentarze zawierające oczywiste stwierdzenia.
Komentarze dokumentacji API.	Komentarze, które usprawiedliwiają zły kod. Zakomentowane wiersze kodu.
Właściwa organizacja za pomocą przestrzeni nazw.	Nieprawidłowa organizacja za pomocą przestrzeni nazw.
Dobre konwencje nazewnictwa.	Złe konwencje nazewnictwa.
Klasy, które wykonują jedno zadanie.	Klasy, które wykonują wiele zadań.
Metody, które wykonują jedno działanie.	Metody, które wykonują wiele działań.
Metody z mniejszą liczbą wierszy niż 10, a najlepiej nie więcej niż 4.	Metody zawierające więcej niż 10 wierszy kodu.
Metody zawierające nie więcej niż dwa parametry.	Metody z więcej niż dwoma parametrami.
Właściwe stosowanie wyjątków.	Korzystanie z wyjątków do sterowania przepływem programu.
Czytelny kod.	Kod trudny do czytania.
Kod, który jest luźno sprzężony.	Kod zawierający ścisłe sprzężenia.
Wysoka spójność kodu.	Niska spójność kodu.
Obiekty są niszczone w czysty sposób.	Obiekty są pozostawiane bez niszczenia.
Unikanie metody `Finalize()`.	Stosowanie metody `Finalize()`.
Właściwy poziom abstrakcji.	Nadmierna złożoność (ang. *over-engineering*).
Korzystanie z regionów w dużych klasach.	Brak stosowania regionów w dużych klasach.
Hermetyzacja i ukrywanie informacji.	Bezpośrednie ujawnianie informacji.
Kod obiektowy.	Kod spaghetti.
Wzorce projektowe.	Antywzorce projektowe.

Zły kod

W tym punkcie przyjrzymy się krótko każdej ze złych praktyk kodowania, które wymieniliśmy wcześniej. Pokażę też w jaki sposób określona praktyka wpływa na kod.

Niewłaściwe wcięcia

Stosowanie nieprawidłowych wcięć może bardzo utrudniać czytanie kodu, zwłaszcza jeśli metody są rozbudowane. Aby kod był łatwy do czytania przez człowieka, trzeba stosować odpowiednie wcięcia. Jeśli w kodzie brakuje wcięć, stwierdzenie, która część kodu należy do danego bloku, może być bardzo trudne.

Domyślnie środowisko Visual Studio 2019 poprawnie formatuje kod i stosuje w nim wcięcia bezpośrednio po wprowadzeniu w kodzie zamykającej klamry. Czasami jednak środowisko nieprawidłowo formatuje kod, aby zwrócić uwagę, że napisany kod zawiera wyjątek. Jeśli jednak do pisania programów stosujesz prosty edytor tekstu, musisz samodzielnie zadbać o sformatowanie kodu.

Poprawianie kodu z niepoprawnymi wcięciami jest czasochłonne. Jest to frustrująca strata czasu programowania, której można z łatwością uniknąć. Przyjrzyjmy się prostemu przykładowi:

```
public void DoSomething()
{
for (var i = 0; i < 1000; i++)
{
var productCode = $"PRC000{i}";
//…implementacja
}
}
```

Powyższy kod nie wygląda zbyt estetycznie, ale wciąż jest czytelny. Jednak im więcej linijek dodasz, tym program stanie się trudniejszy do odczytania.

Bardzo łatwo przegapić zamykający nawias klamrowy. Jeżeli kod nie ma prawidłowych wcięć, znalezienie brakującej klamry zamykającej może stać się znacznie trudniejsze, ponieważ nie można łatwo rozpoznać, w którym bloku kodu brakuje zamykającej klamry.

Komentarze zawierające oczywiste stwierdzenia

Wielu programistów bardzo denerwują komentarze zawierające oczywiste stwierdzenia, ponieważ sprawiają one wrażenie protekcjonalnych. Podczas rozmów z programistami słyszałem ich opinie o tym, że nie lubią komentarzy, ponieważ kod powinien sam się dokumentować.

Doskonale rozumiem ich odczucia. Jeśli potrafisz przeczytać kod bez komentarzy równie łatwo, jak czytasz książkę, i jesteś w stanie go zrozumieć, to jest to naprawdę dobry kawałek kodu. Jeśli zmienna jest zadeklarowana jako `string`, to w jakim celu dodawać komentarz `//string`? Przyjrzyjmy się przykładowi:

```
public int _value; //Używana do przechowywania wartości całkowitoliczbowych…
```

W tym fragmencie kodu wiadomo, że zmienna przechowuje wartość całkowitoliczbową, ponieważ jest typu `int`. W związku z tym nie ma potrzeby, aby w komentarzu zamieszczać oczywiste stwierdzenie. Wprowadzając tego rodzaju komentarze marnujesz tylko czas i energię, a zarazem zaśmiecasz kod.

Komentarze, które usprawiedliwiają zły kod

Czasami mamy napięte terminy, ale komentarze w stylu *//Wiem, że ten kod jest do bani, ale to przynajmniej działa!* są po prostu straszne. Nie rób tego. Wprowadzanie takich komentarzy pokazuje brak profesjonalizmu i może bardzo rozzłościć Twoich kolegów z zespołu.

Jeśli naprawdę musisz wprowadzić coś, co zaledwie działa, zaznacz przynajmniej potrzebę refaktoryzacji za pomocą komentarza `TODO`. Na przykład *//TODO: PBI23154 Refaktoryzacja kodu w celu doprowadzenia do zgodności z firmowymi praktykami kodowania*. Następnie Ty sam — albo inni programiści, którzy otrzymają zadanie pracy nad „spłacaniem technicznego długu" — będziecie mogli pobrać zadanie PBI (ang. *Product Backlog Item*) i zrefaktoryzować kod.

Oto inny przykład:

```
...
int value = GetDataValue(); //Ten kod czasami powoduje błąd dzielenia przez zero. Nie wiem, dlaczego!
...
```

Taki komentarz jest wyjątkowo zły. Co prawda należy się chwała za poinformowanie zespołu o możliwości wystąpienia błędu dzielenia przez zero. Ale czy zgłosiłeś błąd? Czy spróbowałeś znaleźć przyczynę błędu i ją usunąć? Jeśli nie wszyscy, którzy aktywnie pracują nad projektem, dotykają tego kodu, to w jaki sposób dowiedzą się, że kod jest wadliwy?

Minimum tego, co powinieneś zrobić, to wprowadzenie odpowiedniego komentarza `//TODO:`. W takim przypadku komentarz pojawi się na liście zadań, dzięki czemu programiści będą się mogli dowiedzieć o błędzie i zacząć nad nim pracować.

Zakomentowane wiersze kodu

Zakomentowanie wierszy kodu, aby czegoś spróbować, nie jest niczym złym. Jeśli jednak masz zamiar użyć innego kodu zamiast kodu zakomentowanego, to usuń zakomentowany kod przed oddaniem go do repozytorium. Jeden lub dwa zakomentowane wiersze nie są niczym specjalnie złym. Jednak występowanie w kodzie wielu zakomentowanych wierszy jest rozpraszające i sprawia, że kod staje się trudny do utrzymania, a czasami może nawet wprowadzać w błąd:

```
/* Metoda już nie jest używana. Zastąpiono ją metodą DoSomethinElse().
public void DoSomething()
{
   //…implementacja
}
*/
```

Dlaczego usuwać niepotrzebny kod? Po prostu tak trzeba. Jeżeli metoda została zastąpiona przez inną i nie jest już potrzebna, po prostu ją usuń. Jeśli kod jest zarządzany przez system kontroli wersji i będziesz potrzebować starej metody, zawsze możesz przejrzeć historię pliku i ją odtworzyć.

Niewłaściwa organizacja przestrzeni nazw

Jeśli korzystasz z przestrzeni nazw, nie umieszczaj w nich kodu, który powinien znaleźć się w innym miejscu. Może to znacznie utrudnić lub wręcz uniemożliwić znalezienie odpowiedniego kodu, zwłaszcza w dużych bazach kodu. Przyjrzyjmy się przykładowi:

```
namespace MyProject.TextFileMonitor
{
    + public class Program { ... }
    + public class DateTime { ... }
    + public class FileMonitorService { ... }
    + public class Cryptography { ... }
}
```

Widzimy, że wszystkie klasy w powyższym kodzie należą do jednej przestrzeni nazw. Aby lepiej zorganizować ten kod, możemy jednak dodać trzy inne przestrzenie:

- `MyProject.TextFileMonitor.Core` — tutaj znajdą się podstawowe klasy definiujące powszechnie stosowane elementy, na przykład klasa `DateTime`.
- `MyProject.TextFileMonitor.Services` — w tej przestrzeni nazw znajdą się wszystkie klasy, które działają jako usługi, na przykład `FileMonitorService`.
- `MyProject.TextFileMonitor.Security` — tu z kolei będą zawarte wszystkie klasy związane z bezpieczeństwem, na przykład klasa `Cryptography`.

Złe konwencje nazewnictwa

W czasach programowania w Visual Basic 6 używano notacji węgierskiej. Pamiętam, że używałem jej po raz pierwszy, gdy przełączyłem się na Visual Basic 1.0. Obecnie stosowanie notacji węgierskiej nie jest już konieczne. Poza tym korzystanie z niej sprawia, że kod wygląda brzydko. Zamiast więc używać takich nazw jak `lblName`, `txtName` lub `btnSave`, nowoczesnym sposobem jest używanie nazw — odpowiednio — `NameLabel`, `NameTextBox` i `SaveButton`.

Korzystanie z tajemniczych nazw oraz takich, które nie oddają celu kodu, może znacznie utrudnić jego czytanie. Co oznacza nazwa `ihridx`? Oznacza *Human Resources Index* i jest liczbą całkowitą. Naprawdę! Unikaj używania takich nazw, jak `mystring`, `myint` czy `mymethod`. Ich stosowanie naprawdę nie służy żadnemu celowi.

Nie używaj znaku podkreślenia pomiędzy słowami w nazwie, na przykład `Bad_Programmer`. Może to powodować wizualny dyskomfort dla programistów i utrudniać czytanie kodu. Wystarczy usunąć znaki podkreślenia.

Nie stosuj tej samej konwencji kodu dla zmiennych na poziomie klasy i na poziomie metody. Może to spowodować trudności w ustaleniu zakresu zmiennej. Dobrą konwencją dotyczącą nazewnictwa jest używanie notacji `camelCase` dla nazw zmiennych, na przykład `alienSpawn`, i `PascalCase` dla nazw metod, klas, struktur i interfejsów, na przykład `EnemySpawnGenerator`.

Dzięki stosowaniu dobrej konwencji nazw zmiennych, bez trudu rozróżnisz zmienne lokalne (zawarte wewnątrz konstruktora lub metody) od zmiennych składowych (umieszczonych na początku klasy, na zewnątrz konstruktorów i metod). Wystarczy, że poprzedzisz zmienne składowe znakiem podkreślenia. Osobiście używam takiej konwencji kodowania. Sprawdza się naprawdę dobrze, a programiści ją lubią.

Klasy, które wykonują wiele zadań

Dobra klasa powinna wykonywać tylko jedno zadanie. Klasa, która nawiązuje połączenie z bazą danych, pobiera dane, wykonuje na nich działania, ładuje raport, przypisuje dane do raportu, wyświetla go, zapisuje na dysku, drukuje i eksportuje go, robi zbyt dużo. Taką klasę trzeba poddać refaktoryzacji na mniejsze, lepiej zorganizowane klasy. Takie wszechstronne klasy są bardzo trudne do czytania. Dla mnie czytanie ich kodu jest bardzo uciążliwe. Jeśli natkniesz się na klasy tego rodzaju, zorganizuj ich funkcjonalności za pomocą regionów. Następnie przenieś kod z tych regionów do nowych klas, które wykonują jedno zadanie.

Przyjrzyjmy się przykładowi klasy, która wykonuje wiele zadań:

```
public class DbAndFileManager
{
 #region Operacje na bazie danych
 public void OpenDatabaseConnection() { throw new
     NotImplementedException(); }
 public void CloseDatabaseConnection() { throw new
     NotImplementedException(); }
 public int ExecuteSql(string sql) { throw new
     NotImplementedException(); }
 public SqlDataReader SelectSql(string sql) { throw new
     NotImplementedException(); }
 public int UpdateSql(string sql) { throw new
     NotImplementedException(); }
 public int DeleteSql(string sql) { throw new
     NotImplementedException(); }
 public int InsertSql(string sql) { throw new
     NotImplementedException(); }

 #endregion

 #region Operacje na plikach
 public string ReadText(string filename) { throw new
     NotImplementedException(); }
 public void WriteText(string filename, string text) { throw new
     NotImplementedException(); }
 public byte[] ReadFile(string filename) { throw new
     NotImplementedException(); }
 public void WriteFile(string filename, byte[] binaryData) { throw new
     NotImplementedException(); }
 #endregion
}
```

Jak widać w powyższym kodzie, klasa wykonuje dwa podstawowe zadania: operacje na bazie danych oraz operacje na plikach. Teraz kod jest starannie zorganizowany w odpowiednio nazwanych regionach wykorzystanych do logicznego rozdzielenia kodu wewnątrz klasy. Jednak zasada

pojedynczej odpowiedzialności (ang. *Single Responsibility Principle* — SRP) jest złamana. Należałoby zacząć od refaktoryzacji tego kodu, aby wydzielić operacje bazodanowe do oddzielnej klasy o nazwie, na przykład, `DatabaseManager`.

Następnie można by usunąć operacje bazodanowe z klasy `DbAndFileManager` i pozostawić jedynie operacje na plikach, a następnie zmienić nazwę klasy z `DbAndFileManager` na `FileManager`. Warto by również zastanowić się nad przestrzeniami nazw każdego pliku i zdecydować, czy nie należy ich zmodyfikować w taki sposób, aby klasa `DatabaseManager` została umieszczona w przestrzeni nazw `Data`, natomiast `FileManager` w przestrzeni nazw `FileSystem` lub ich odpowiednikach w naszym programie.

Poniższy kod jest wynikiem wyodrębnienia kodu obsługi bazy danych z klasy `DbAndFileManager` do oddzielnej klasy i skorygowania przestrzeni nazw:

```
using System;
using System.Data.SqlClient;

namespace CH01_CodingStandardsAndPrinciples.GoodCode.Data
{
    public class DatabaseManager
    {
        #region Operacje na bazie danych

        public void OpenDatabaseConnection() { throw new
            NotImplementedException(); }
        public void CloseDatabaseConnection() { throw new
            NotImplementedException(); }
        public int ExecuteSql(string sql) { throw new
            NotImplementedException(); }
        public SqlDataReader SelectSql(string sql) { throw new
            NotImplementedException(); }
        public int UpdateSql(string sql) { throw new
            NotImplementedException(); }
        public int DeleteSql(string sql) { throw new
            NotImplementedException(); }
        public int InsertSql(string sql) { throw new
            NotImplementedException(); }

        #endregion
    }
}
```

W wyniku refaktoryzacji kodu obsługi systemu plików otrzymujemy klasę `FileManager` w przestrzeni nazw `FileSystem`, jak pokazałem w poniższym kodzie:

```
using System;

namespace CH01_CodingStandardsAndPrinciples.GoodCode.FileSystem
{
    public class FileManager
    {
        #region Operacje na plikach
```

```
        public string ReadText(string filename) { throw new
            NotImplementedException(); }
        public void WriteText(string filename, string text) { throw new
            NotImplementedException(); }
        public byte[] ReadFile(string filename) { throw new
            NotImplementedException(); }
        public void WriteFile(string filename, byte[] binaryData) { throw
            new NotImplementedException(); }

        #endregion
    }
}
```

Powyżej pokazałem jak rozpoznać klasy, które wykonują zbyt wiele zadań i jak można je zrefaktoryzować, aby wykonywały tylko jedno zadanie. Teraz powtórzę ten proces dla metod — przyjrzyjmy się metodom, które wykonują wiele działań.

Metody, które wykonują wiele działań

Osobiście gubię się czytając metody zawierające wiele poziomów wcięć, w których wykonywanych jest bardzo wiele różnych działań. Spotykałem zadziwiające permutacje wcięć i działań wykonywanych w pojedynczej metodzie. Niejednokrotnie chciałem zrefaktoryzować kod, aby ułatwić jego utrzymanie, ale mój szef mi tego zabraniał. Widziałem wyraźnie, w jaki sposób można skrócić metodę dzięki przeniesieniu kodu do innych metod.

Czas na przykład. W tym fragmencie metoda pobiera ciąg znaków. Ten ciąg znaków jest następnie szyfrowany i odszyfrowywany. Metoda jest zbyt długa. Na jej przykładzie pokażę, dlaczego metody powinny być krótkie:

```
public string security(string plainText)
{
    try
    {
        byte[] encrypted;
        using (AesManaged aes = new AesManaged())
        {
            ICryptoTransform encryptor = aes.CreateEncryptor(Key, IV);
            using (MemoryStream ms = new MemoryStream())
            using (CryptoStream cs = new CryptoStream(ms, encryptor,
            CryptoStreamMode.Write))
            {
                using (StreamWriter sw = new StreamWriter(cs))
                    sw.Write(plainText);
                encrypted = ms.ToArray();
            }
        }
        Console.WriteLine($"Zaszyfrowane dane:
          { System.Text.Encoding.UTF8.GetString(encrypted)}
        ");
        using (AesManaged aesm = new AesManaged())
        {
            ICryptoTransform decryptor = aesm.CreateDecryptor(Key, IV);
```

```
                using (MemoryStream ms = new MemoryStream(encrypted))
                {
                    using (CryptoStream cs = new CryptoStream(ms, decryptor,
                    CryptoStreamMode.Read))
                    {
                        using (StreamReader reader = new StreamReader(cs))
                            plainText = reader.ReadToEnd();
                    }
                }
            }
            Console.WriteLine($"Odszyfrowane dane: {plainText}");
        }
        catch (Exception exp)
        {
            Console.WriteLine(exp.Message);
        }
        Console.ReadKey();
        return plainText;
    }
```

Jak widać, powyższa metoda ma powyżej 10 wierszy kodu i jest trudna do czytania. Poza tym wykonuje więcej niż jedno działanie. Jej kod można podzielić na dwie metody, z których każda będzie wykonywała tylko jedno działanie. Jedna z metod szyfrowałaby ciąg znaków, a druga odszyfrowywałaby go. Powyższy przykład pokazuje też, dlaczego metody nie powinny mieć więcej niż 10 wierszy kodu.

Metody zawierające więcej niż 10 wierszy kodu

Rozbudowane metody nie są łatwe do czytania i trudno je zrozumieć. Występujące w nich błędy mogą być również bardzo trudne do odszukania. Kolejny problem z rozwlekłymi metodami polega na tym, że bardzo łatwo zapomnieć o ich pierwotnym celu. Jest jeszcze gorzej, gdy mamy rozbudowaną metodę, która zawiera sekcje kodu rozdzielone komentarzami oraz kod podzielony na regiony.

Jeśli musisz przewijać ekran, by czytać metodę, to jest ona zbyt długa. Czytanie takiej metody może być kłopotliwe i prowadzić do błędnej interpretacji jej działania. To z kolei może prowadzić do wprowadzania modyfikacji, które powodują błędne działanie kodu, wypaczają jego zamiar, albo jedno i drugie. Metody powinny być jak najkrótsze. Trzeba jednak zachować zdrowy rozsądek. Można bowiem doprowadzić do nadmiernego skracania metod. Kluczem do uzyskania właściwej równowagi jest zadbanie o to, by cel metody był jasny i zaimplementowane w zwięzły sposób.

Powyższy kod jest dobrym kandydatem do uzasadnienia powodów, dla których warto dbać o to, by metody były krótkie. Krótkie metody są czytelne i łatwe do zrozumienia. Zazwyczaj jeśli kod metody przekracza 10 wierszy, to istnieje prawdopodobieństwo, że robi więcej niż powinien. Zadbaj o to, aby nazwy metod odpowiadały wykonywanym przez nie działaniom, na przykład `OpenDatabaseConnection()` i `CloseDatabaseConnection()`. Pamiętaj też, że po wprowadzeniu modyfikacji powinny utrzymywać ich pierwotny cel i nie powinny od niego odbiegać.

W kolejnym punkcie przyjrzymy się parametrom metod.

Metody z więcej niż dwoma parametrami

Metody z wieloma parametrami mogą być nieporęczne. Poza tym, że są trudne do czytania, to bardzo łatwo może dojść do przekazania wartości do niewłaściwego parametru i złamania bezpieczeństwa typów.

Wraz ze wzrostem liczby parametrów rośnie trudność testowania metod. Głównym powodem jest konieczność zastosowania większej liczby permutacji w przypadkach testowych. Istnieje zatem zagrożenie, że pominiesz przypadek użycia, który później spowoduje problemy w produkcji.

Korzystanie z wyjątków do sterowania przepływem programu

Wyjątki wykorzystywane do sterowania przepływem programu mogą ukryć wyrażone w kodzie intencje programisty. Mogą również prowadzić do nieoczekiwanych i niezamierzonych rezultatów. Sam fakt, że kod został zaprogramowany tak, że spodziewa się wystąpienia jednego lub większej liczby wyjątków, świadczy o błędach w projekcie. Typowy scenariusz opisałem bardziej szczegółowo w rozdziale 5., „Obsługa wyjątków".

Zazwyczaj w oprogramowaniu wykorzystywane są wyjątki reguł biznesowych (*Business Rule Exceptions* — BRE). Metoda wykonując działanie przewiduje możliwość zgłoszenia wyjątku. Przepływ programu jest określony przez to, czy wyjątek wystąpił, czy nie. Dużo lepszym sposobem postępowania jest użycie dostępnych konstrukcji języka w celu dokonania weryfikacji, w wyniku której otrzymujemy wartość logiczną.

Zastosowanie wyjątków BRE do sterowania przepływem programu pokazałem w poniższym przykładzie kodu:

```
public void BreFlowControlExample(BusinessRuleException bre)
{
    switch (bre.Message)
    {
        case "OutOfAcceptableRange":
            DoOutOfAcceptableRangeWork();
            break;
        default:
            DoInAcceptableRangeWork();
            break;
    }
}
```

Powyższa metoda przyjmuje parametr typu `BusinessRuleException`. W zależności od komunikatu w wyjątku `BreFlowControlExample()` kod wywołuje metodę `DoOutOfAcceptableRangeWork()` lub metodę `DoInAcceptableRangeWork()`.

O wiele lepszym sposobem sterowania przepływem jest korzystanie z logiki Boole'a. Przyjrzyjmy się poniższej metodzie `BetterFlowControlExample()`:

```
public void BetterFlowControlExample(bool isInAcceptableRange)
{
    if (isInAcceptableRange)
```

```
            DoInAcceptableRangeWork();
        else
            DoOutOfAcceptableRangeWork();
    }
```

Do metody `BetterFlowControlExample()` przekazywana jest wartość logiczna. Ta wartość jest używana do ustalenia ścieżki wykonania programu. Jeśli warunek mieści się w dopuszczalnym zakresie, wykonywana jest metoda `DoInAcceptableRangeWork()`.W przeciwnym razie jest wywoływana metoda `DoOutOfAcceptableRangeWork()`.

W kolejnym podpunkcie rozważymy kod, który jest trudny do czytania.

Kod trudny do czytania

Kod określany mianem *lasagne code* lub *spaghetti code* jest bardzo trudny do czytania lub śledzenia. Kłopot mogą również sprawiać źle nazwane metody, ponieważ mogą one zaciemniać cel kodu. Metody stają się jeszcze bardziej niejasne, jeśli są rozbudowane oraz jeśli powiązane metody są rozdzielone kilkoma metodami, które nie mają z nimi związku.

Kod lasagne, zwany również powszechnie kodem typu *indirection*, odwołuje się do warstw abstrakcji, w których obowiązują odwołania przez nazwę, a nie przez działania. W programowaniu obiektowym warstwy mają szerokie zastosowanie i są bardzo skuteczne. Jednak im więcej poziomów abstrakcji w kodzie, tym staje się on bardziej złożony. To może sprawiać, że szybkie zrozumienie kodu przez nowych programistów pracujących nad projektem staje się bardzo trudne. Trzeba więc zachować równowagę pomiędzy poziomem abstrakcji, a łatwością zrozumienia kodu.

Kod spaghetti oznacza splątany bałagan szczelnie sprzężonego kodu o niskiej spójności. Taki kod jest bardzo trudny do utrzymania, refaktoryzacji, rozszerzania i wprowadzania zmian w projekcie. Ma jednak jeden plus — ze względu na to, że jest w większym stopniu proceduralny, może być łatwy do czytania i śledzenia. Pamiętam moją pracę na stanowisku młodszego programisty nad programem VB6 GIS, który został sprzedany do różnych firm i był wykorzystywany do celów marketingowych. Mój dyrektor techniczny i starsi programiści wcześniej starali się zmienić projekt oprogramowania, ale ich próby się nie powiodły. Dlatego to mnie powierzyli to zadanie. Jednak w tamtym czasie nie byłem specjalistą w analizie i projektowaniu oprogramowania, dlatego mnie również się nie udało.

Kod był po prostu zbyt skomplikowany, aby można go było śledzić oraz pogrupować na powiązane ze sobą części. Był po porostu zbyt rozbudowany. Z perspektywy czasu uważam, że osiągnąłbym lepsze efekty, gdybym zrobił zestawienie wszystkiego, co program robił, rozdzielił tę listę na grupy według funkcji, a następnie stworzył listę wymagań nawet nie patrząc na kod.

Zasada, której nauczyłem się podczas próby zmiany projektu oprogramowania brzmi, że podczas takiej próby należy unikać zaglądania do kodu. Zapisz wszystko, co program robi, a potem wymień wszystkie nowe funkcjonalności, które powinien zawierać. Przekształć listę na zbiór wymagań programowych z powiązanymi zadaniami, testami oraz kryteriami akceptacji, a następnie zacznij programować zgodnie ze specyfikacją.

Kod zawierający ścisłe sprzężenia

Kod, który ma ścisłe sprzężenia, jest trudny do testowania, rozszerzania lub modyfikowania. Kod, który zależy od innego kodu w systemie, trudno jest również wykorzystywać wielokrotnie.

Przykładem ścisłego sprzężenia jest odwoływanie się na liście parametrów do konkretnego typu klasy zamiast do interfejsu. Gdy odwołujesz się do konkretnej klasy, wszelkie zmiany w konkretnej klasie bezpośrednio wpływają na klasę, która się do niej odwołuje. W związku z tym, jeśli masz klasę realizującą połączenie z bazą danych dla klienta, który łączy się z systemem SQL Server, a następnie będziesz chciał sprzedać rozwiązanie innemu klientowi, który wymaga bazy danych Oracle, to będziesz zmuszony zmodyfikować konkretną klasę dla tego konkretnego klienta i stosowanej przez niego bazy danych Oracle. Takie postępowanie prowadzi do powstania dwóch wersji kodu.

Im więcej masz klientów, tym więcej potrzebnych wersji kodu. Utrzymywanie takiego projektu staje się prawdziwym koszmarem. Wyobraź sobie, że Twoja klasa do połączeń z bazą danych ma 100 000 różnych klientów używających 1 z 30 odmian tej klasy, a wszystkie one mają ten sam błąd, który dotyczy wszystkich tych odmian. To oznacza, że trzeba wprowadzić tę samą zmianę w 30 klasach, a następnie je przetestować, zbudować i wdrożyć. To oznacza olbrzymią pracochłonność utrzymania, a także wysokie koszty.

Rozwiązaniem problemów w wymienionym powyżej scenariuszu może być odwoływanie się do interfejsu, a następnie zastosowanie fabryki bazy danych w celu zbudowania wymaganego obiektu połączenia. Następnie klient może ustawić ciąg połączenia w pliku konfiguracyjnym i przekazać go do fabryki. Fabryka utworzy konkretną klasę połączenia, która implementuje interfejs połączenia dla konkretnego typu bazy danych określonego w ciągu połączenia.

Oto przykład źle — zbyt ściśle — sprzężonego kodu:

```
public class Database
{
    private SqlServerConnection _databaseConnection;
    public Database(SqlServerConnection databaseConnection)
    {
        _databaseConnection = databaseConnection;
    }
}
```

Jak widać w tym przykładzie, klasa `Database` jest powiązana z użyciem systemu SQL Server, a zmiana jej w celu przystosowania do wykorzystania dowolnego innego typu bazy danych wymaga wprowadzenia poprawek w kodzie. Sposób refaktoryzacji kodu, wraz z odpowiednimi przykładami, omówię w dalszych rozdziałach.

Niska spójność kodu

Kod o niskiej spójności to grupa niepowiązanych ze sobą fragmentów, które realizują wiele różnych zadań. Przykładem może być klasa narzędziowa zawierająca zbiór różnych metod narzędziowych do obsługi dat, tekstu, liczb, plikowych operacji wejścia-wyjścia, walidacji danych oraz szyfrowania i deszyfrowania.

Pozostawione obiekty

Obiekty niezniszczone i pozostawione w pamięci mogą prowadzić do jej wycieków.

Do wycieków pamięci (z kilku powodów) może również prowadzić korzystanie ze zmiennych statycznych. Jeśli nie używasz klasy `DependencyObject` lub `INotifyPropertyChanged`, to w gruncie rzeczy korzystasz z subskrypcji zdarzeń. Środowisko CLR (*Common Language Runtime*) tworzy silne odwołania przy użyciu zdarzenia `ValueChanged` za pośrednictwem zdarzenia `PropertyDescriptor.AddValueChanged`, które powoduje przechowywanie obiektu `Property` ↪`Descriptor` odwołującego się do powiązanego z nim obiektu.

Jeśli nie anulujesz subskrypcji tych powiązań, dojdzie do wycieku pamięci. Może do niego dojść również wtedy, gdy używasz zmiennych statycznych odwołujących się do obiektów, które nie zostały zwolnione. Każdy obiekt, do którego odwołujesz się za pomocą zmiennej statycznej, jest oznaczony po to, by mechanizm odśmiecania (ang. *garbage collector*) go nie niszczył. To dlatego, że statyczne zmienne odwołujące się do obiektów są tzw. korzeniami GC (*Garbage Collection roots*) oznaczonymi w celu wyłączenia ich z procesu odśmiecania.

Podczas korzystania z metod anonimowych, które odwołują się do składowych klasy, odwołania dotyczą egzemplarza tej klasy. Z tego powodu odwołanie do egzemplarza klasy pozostaje aktywne tak długo, jak długo jest wykorzystywana metoda anonimowa.

Do wycieków pamięci dojdzie także wtedy, gdy używasz kodu niezarządzanego (COM) i wyraźnie nie zwolnisz dowolnych obiektów zarządzanych i niezarządzanych.

Kod, który w nieskończoność buforuje obiekty, bez użycia słabych referencji, usuwania nieużywanej pamięci podręcznej lub ograniczenia jej rozmiaru w końcu doprowadzi do tego, że pamięci zabraknie.

Do wycieków pamięci może również dojść w przypadku, gdy stworzysz referencje do obiektów w wątku, który nigdy się nie kończy.

Subskrypcje zdarzeń, które nie są anonimowe, odwołują się do klas. Jeśli subskrypcje tych zdarzeń nie będą anulowane, obiekty, których one dotyczą, pozostaną w pamięci. Trzeba zatem pamiętać, że jeśli nie anulujesz subskrypcji zdarzeń, gdy przestaną być potrzebne, również może dojść do wycieku pamięci.

Korzystanie z metody Finalize()

Choć finalizatory — poprzez to, że mogą pomóc w zwalnianiu zasobów obiektów, które nie zostały prawidłowo zniszczone — pomagają zapobiec wyciekom pamięci, mają szereg wad.

Nie wiemy, kiedy finalizator zostanie wywołany. Finalizatory są promowane przez mechanizm odśmiecania wraz ze wszystkimi zależnościami do następnej generacji i nie są sprzątane do czasu, aż mechanizm odśmiecania zdecyduje się to zrobić. To może oznaczać, że obiekty pozostaną w pamięci przez długi czas. Używanie finalizatorów może prowadzić do powstawania wyjątków wyczerpywania się pamięci, ponieważ często nowe obiekty są tworzone szybciej niż stare są niszczone przez mechanizm odśmiecania.

Nadmierna złożoność

Nadmierna złożoność kodu może stać się prawdziwym koszmarem. Trzeba pamiętać, że jesteśmy tylko ludźmi, a analiza zagmatwanego systemu, dążenie do zrozumienia jego mechanizmów oraz sposobów ich wykorzystywania to bardzo czasochłonny proces. Zrozumienie systemu jest szczególnie trudne, gdy nie ma on dokumentacji, jest dla nas nowością, a nawet osoby, które używają go znacznie dłużej niż my, nie potrafią odpowiedzieć na stawiane pytania.

Nadmierna złożoność może być główną przyczyną stresu. Dzieje się tak zwłaszcza wtedy, gdy menedżerowie wymagają od nas pracy pod presją czasu.

Naucz się stosować zasadę KISS

Dobrym przykładem słuszności stosowania tej zasady jest sytuacja z jednego z miejsc, w którym pracowałem. Miałem napisać test dla aplikacji webowej, który miał przyjmować ciąg JSON z usługi, pozwalał procesowi potomnemu wykonać test, a następnie przekazywał uzyskaną punktację do innej usługi. Nie zastosowałem zasad OOP, SOLID ani DRY, tak jak powinienem zgodnie z zasadami obowiązującymi w firmie. Wykonałem jednak zadanie stosując zasadę KISS (ang. *Keep It Simple, Stupid* — dosł. to ma być proste, głupcze). Zrobiłem to bardzo szybko wykorzystując programowanie proceduralne i zdarzenia. Zostałem za to zganiony i zmuszony do przepisania zadania z użyciem stworzonego w firmie systemu testowego.

Musiałem więc nauczyć się korzystać z tego systemu. Nie było żadnej dokumentacji, system nie był stworzony zgodnie z zasadami DRY i bardzo niewiele osób rozumiało zasady jego funkcjonowania. Zadanie zamiast zająć mi kilka dni (tyle zajęło mi tworzenie mojego systemu w prostej postaci), w nowej wersji zajęło mi kilka tygodni, ponieważ system testowy nie robił tego, co było potrzebne do wykonania zadania, a nie wolno mi było go zmodyfikować, aby robił to, czego potrzebowałem. Moje prace zostały zatem spowolnione, ponieważ byłem zmuszony czekać, aż inny programista wprowadzi dla mnie potrzebne zmiany.

Moje pierwsze rozwiązanie spełniało wymagania biznesowe i było niezależnym fragmentem kodu, który nie wpływał na pracę nikogo innego. Drugie rozwiązanie spełniało wymagania techniczne zespołu programistów. Realizacja projektu trwała dłużej niż planowano. Każdy projekt, który przekracza wyznaczony termin, kosztuje firmę więcej pieniędzy niż pierwotnie zakładano.

Poza tym, mój system, który został odrzucony, był o wiele prostszy i łatwiejszy do zrozumienia niż nowszy system, który przepisałem tak, by korzystał z firmowego systemu uruchamiania testów.

Nie zawsze trzeba stosować zasady OOP, SOLID i DRY. Czasami się to nie opłaca. Ostatecznie, nawet system stosujący ściśle zasady OOP „pod maską" jest konwertowany do kodu proceduralnego, który jest bliższy temu, w jaki sposób komputer go interpretuje!

Brak stosowania regionów w rozbudowanych klasach

Rozbudowane klasy z wieloma regionami są bardzo trudne do czytania i śledzenia, zwłaszcza gdy powiązane ze sobą metody nie są pogrupowane. Regiony są bardzo dobre do grupowania powiązanych ze sobą składowych w dużej klasie. Nie jest dobrze, jeśli z nich nie korzystasz!

Utrata intencji wyrażonych w kodzie

Jeśli przeglądasz klasę, która robi kilka rzeczy, nie masz pojęcia, jaki były pierwotne intencje programisty. Jeśli, na przykład, szukasz metody przetwarzania daty i znajdujesz ją w klasie należącej do przestrzeni nazw wejścia-wyjścia, to czy ta metoda przetwarzania daty znajduje się we właściwym miejscu? Nie. Czy innym programistom, którzy nie znają Twojego kodu, znalezienie tej metody przysporzy trudności? Oczywiście, że tak. Spójrzmy na poniższy kod:

```
public class MyClass
{
    public void MyMethod()
    {
        //…implementacja
    }
    public DateTime AddDates(DateTime date1, DateTime date2)
    {
        //…implementacja
    }
    public Product GetData(int id)
    {
        //…implementacja
    }
}
```

Jaki jest cel działania tej klasy? Nazwa nie daje żadnych wskazówek, nie wiemy też, co robi metoda `myMethod`. Wydaje się, że klasa wykonuje operacje na datach oraz pobiera dane o produkcie. Metoda `AddDates` powinna znaleźć się w klasie zajmującej się wyłącznie zarządzaniem datami. Z kolei metoda `GetData` powinna należeć do modelu widoku produktu.

Bezpośrednie ujawnianie informacji

Klasy, które bezpośrednio ujawniają informacje, są złe. Oprócz tego, że generują ścisłe sprzężenia, które mogą prowadzić do błędów, to jeśli chcesz zmienić typ informacji, musisz zmienić typ wszędzie, gdzie jest on wykorzystywany. Ponadto powstaje kłopot w sytuacji, gdy chcesz sprawdzić poprawność danych przed ich przypisaniem. Oto przykład:

```
public class Product
{
    public int Id;
    public int Name;
    public int Description;
    public string ProductCode;
    public decimal Price;
    public long UnitsInStock
}
```

Przy takiej postaci klasy `Product`, gdybyś chciał zmienić pole `UnitsInStock` z typu `long` na `int`, musiałbyś to zrobić we wszystkich miejscach w kodzie, w których odwołujesz się do klasy `Product`. To samo dotyczy pola `ProductCode`. Jeśli kody nowych produktów muszą mieć ścisły format, nie będziesz w stanie sprawdzić ich poprawności w sytuacji, gdy kody są przypisane bezpośrednio przez klasę wywołującą.

Dobry kod

Teraz, gdy już wiesz, czego nie robić, nadszedł czas, aby krótko przyjrzeć się kilku dobrym praktykom kodowania, abyś mógł pisać dobry i wydajny kod.

Odpowiednie wcięcia

Czytanie kodu jest znacznie łatwiejsze, jeśli są w nim odpowiednie wcięcia. Na podstawie wcięć można powiedzieć, gdzie zaczynają się bloki kodu i gdzie się kończą, a także jaki kod należy do tych bloków kodu:

```
public void DoSomething()
{
    for (var i = 0; i < 1000; i++)
    {
        var productCode = $"PRC000{i}";
        //…implementacja
    }
}
```

W powyższym prostym przykładzie kod wygląda estetycznie i jest czytelny. Można wyraźnie zobaczyć, gdzie rozpoczyna się każdy blok kodu i gdzie się kończy.

Sensowne komentarze

Sensowne komentarze to takie, które wyrażają intencje programisty. Kod może wydawać się zrozumiały, gdy jest poprawny, ale może być niezrozumiały dla osób, które go nie znają lub nawet dla programisty będącego jego autorem po upływie kilku tygodni. W takim przypadku sensowne komentarze bardzo się przydają.

Komentarze dokumentacji API

Dobry interfejs API to taki, który ma dobrą i łatwą do śledzenia dokumentację. Komentarze API mają format XML i mogą być wykorzystane do generowania dokumentacji HTML. Dokumentacja HTML jest ważna dla programistów chcących korzystać z Twojego API. Im lepsza dokumentacja, tym więcej programistów będzie korzystać z API. Oto przykład:

```
/// <summary>
/// Tworzy nowy <see cref="KustoCode"/> egzemplarz na podstawie tekstu i wartości globalnych. Nie wykonuje
/// analizy semantycznej.
/// </summary>
/// <param name="text">Tekst kodu</param>
/// <param name="globals">
/// Ustawienia globalne do wykorzystania podczas parsowania i analizy semantycznej.
Domyślnie <see cref = GlobalState.Default />
/// </param>.
public static KustoCode Parse(string text, GlobalState globals = null)
{
    ...
}
```

Ten fragment projektu Kusto Query Language jest dobrym przykładem komentarza zawierającego dokumentację API.

Właściwa organizacja kodu za pomocą przestrzeni nazw

Kod, który jest właściwie zorganizowany i umieszczony w odpowiednich przestrzeniach nazw, może zaoszczędzić programistom sporą ilość czasu podczas poszukiwania określonego fragmentu. Na przykład, jeśli szukasz klas i metod przetwarzających daty i godziny, dobrym pomysłem jest utworzenie przestrzeni nazw o nazwie `DateTime`, klasy o nazwie `Time` zawierającej metody przetwarzania godzin oraz klasy o nazwie `Date` zawierającej metody dotyczące dat.

Poniżej znajduje się przykład właściwej organizacji przestrzeni nazw:

Nazwa	Opis
`CompanyName.IO.FileSystem`	Przestrzeń nazw zawierająca klasy, które definiują operacje na plikach i katalogach.
`CompanyName.Converters`	Przestrzeń nazw zawierająca klasy do wykonywania różnych operacji konwersji.
`CompanyName.IO.Streams`	Przestrzeń nazw zawierająca typy zarządzania strumieniami wejścia i wyjścia.

Dobre konwencje nazewnictwa

Zalecam stosowanie konwencji nazewnictwa Microsoft C#. Do nazw klas, interfejsów, typów wyliczeniowych i metod używaj konwencji PascalCase. Do nazw zmiennych i argumentów stosuj konwencję camelCase. Zadbaj o to, aby zmienne składowe miały prefiks w postaci znaku podkreślenia.

Przyjrzyjmy się poniższemu fragmentowi kodu:

```
using System;
using System.Text.RegularExpressions;
namespace CompanyName.ProductName.RegEx
{
    /// <summary>
    /// Klasa rozszerzeń zawierająca operacje z wykorzystaniem wyrażeń regularnych
    /// </summary>
    public static class RegularExpressions
    {
        private static string _preprocessed;
        public static string RegularExpression { get; set; }
        public static bool IsValidEmail(this string email)
        {
            // Adres e-mail: format RFC 2822.
            // Pasuje do standardowego adresu e-mail. Nie sprawdza
            // domeny najwyższego poziomu.
            // Wymaga ustawienia opcji "case insensitive" na ON.
            var exp = @"\A(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.
[a-z0-9!#$%&'*+/=?^_`{|}~-]+)@(?:[a-z0-9](?:[a-z0-9-]
```

```
    [a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?)\Z";
                bool isEmail = Regex.IsMatch(email, exp, RegexOptions.IgnoreCase);
                return isEmail;
            }
            // … pozostała część implementacji…
        }
    }
```

Powyższy fragment kodu prezentuje odpowiednie przykłady konwencji nazewnictwa dla przestrzeni nazw, klas, zmiennych składowych, parametrów i zmiennych lokalnych.

Klasy, które wykonują jedno zadanie

Dobra klasa to taka, która wykonuje tylko jedno zadanie. Podczas czytania kodu takiej klasy, bez trudu można odczytać jej cel. W klasie powinien znaleźć się tylko ten kod, który dotyczy danej klasy i nic poza tym.

Metody, które wykonują tylko jedno zadanie

Metoda powinna realizować tylko jedno zadanie. Nie należy tworzyć metod, które wykonują więcej niż jedną operację, na przykład odszyfrowywanie ciągu znaków i jego zastępowanie. Intencja utworzenia metody powinna być czytelna. Metody, które wykonują tylko jedną rzecz zazwyczaj są krótkie, czytelne i lepiej komunikują swój cel.

Metody z mniejszą liczbą wierszy niż 10, a najlepiej nie więcej niż 4

W idealnej sytuacji metody nie powinny mieć więcej niż 4 wiersze kodu. Nie zawsze jest to jednak możliwe, dlatego należy dążyć do tworzenia metod, które nie mają więcej niż 10 wierszy. Dzięki temu będą one łatwe do czytania i utrzymania.

Metody z nie więcej niż dwoma parametrami

Najlepiej, gdy metody nie mają żadnych parametrów, ale nie ma problemu, jeśli metoda przyjmuje jeden lub dwa parametry. Jeśli jednak ma więcej niż dwa parametry, warto się zastanowić nad odpowiedzialnością klasy, do której należy, oraz jej metod: czy parametrów nie jest zbyt dużo? Jeśli potrzebujesz więcej niż dwóch parametrów, to będzie lepiej, jeśli przekażesz obiekt.

Każda metoda z więcej niż dwoma parametrami może stać się trudna do czytania i śledzenia. Jeśli metody mają nie więcej niż dwa parametry, ich kod jest czytelny, a pojedynczy parametr, który jest obiektem, jest znacznie bardziej czytelny niż metoda z kilkoma parametrami.

Właściwe stosowanie wyjątków

Nigdy nie korzystaj z wyjątków do sterowania przepływem programu. Staraj się obsługiwać znane warunki, które mogą powodować wyjątki, w taki sposób, aby wyjątki nie były zgłaszane. Dobrze zaprojektowana klasa powinna unikać wyjątków.

Działania naprawcze po wystąpieniu wyjątków i (lub) zwalnianie zasobów wykonuj za pomocą konstrukcji `try /catch/finally`. Podczas przechwytywania wyjątków używaj konkretnych typów wyjątków, które mogą być rzucane w kodzie. Dzięki temu będziesz mieć dostęp do szczegółowych informacji, które można umieścić w logu bądź wykorzystać do obsługi wyjątku.

Czasami używanie predefiniowanych typów wyjątków środowiska .NET nie jest możliwe. W takich przypadkach konieczne jest stworzenie własnych wyjątków. Klasy definiujące niestandardowe wyjątki powinny mieć nazwy kończące się słowem `Exception`. Zadbaj o zdefiniowanie w nich następujących trzech konstruktorów:

- `Exception()` — używa wartości domyślnych.
- `Exception(string)` — przyjmuje ciąg komunikatu.
- `Exception(string, exception)` — przyjmuje ciąg komunikatu i wyjątek wewnętrzny.

W przypadku konieczności rzucania wyjątku, nie zwracaj kodów błędów, ale wyjątki zawierające istotne informacje.

Czytelny kod

Im bardziej czytelny kod, tym więcej programistów będzie miało przyjemność pracy z nim. Takiego kodu łatwiej się nauczyć i wygodniej się z nim pracuje. Programiści przychodzą do projektu i z niego odchodzą, dlatego warto zadbać o zapewnienie czytelności kodu tak, by nowym osobom w projekcie łatwiej było czytać, rozszerzać i utrzymywać kod. Czytelny kod jest również mniej podatny na błędy i problemy z zabezpieczeniami.

Luźno sprzężony kod

Luźno sprzężony kod jest łatwiejszy do testowania i refaktoryzacji. Także zastępowanie i modyfikowanie takiego kodu jest łatwiejsze. Kolejną jego zaletą jest możliwość jego wielokrotnego wykorzystywania.

Odwołajmy się do naszego przykładu złego przekazywania połączenia do bazy danych SQL Server. Tę samą klasę moglibyśmy zaprojektować tak, by była luźno sprzężona, dzięki odwołaniu do interfejsu zamiast do konkretnego typu. Spójrzmy na przykład wcześniejszego kodu po refaktoryzacji:

```
public class Database
{
    private IDatabaseConnection _databaseConnection;
    public Database(IDatabaseConnection databaseConnection)
    {
        _databaseConnection = databaseConnection;
    }
}
```

Jak widać w tym dość prostym przykładzie, do klasy możemy przekazać dowolną klasę implementującą połączenie z dowolną bazą danych, pod warunkiem, że klasa ta implementuje interfejs `IDatabaseConnection`. Jeśli zatem błąd występuje w klasie implementującej połączenie z SQL Server, to ten błąd wpływa wyłącznie na klienty SQL Server. Oznacza to, że aplikacje klienckie

korzystające z innych baz danych będą nadal działać, a żeby naprawić błąd, wystarczy zmodyfikować klasę implementującą połączenie z bazą SQL Server. Takie pisanie kodu ułatwia jego utrzymywanie, a tym samym zmniejsza ogólne koszty związane z jego konserwacją.

Wysoka spójność

O współdzielonych funkcjonalnościach, które są poprawnie pogrupowane, mówimy, że są spójne. Jeśli kod jest spójny, możemy łatwo odnaleźć w nim te fragmenty, które nas interesują. Na przykład, jeśli przyjrzymy się przestrzeni nazw `Microsoft System.Diagnostics` odkryjemy, że zawiera ona kod dotyczący wyłącznie diagnostyki. Umieszczenie w przestrzeni nazw `Diagnostics` kolekcji oraz kodu obsługi systemu plików nie miałoby sensu.

Czyste niszczenie obiektów

Jeśli korzystamy z klas implementujących interfejs `IDisposable`, zawsze powinniśmy pamiętać o wywołaniu metody `Dispose()`, aby w czysty sposób zniszczyć wszystkie wykorzystywane zasoby. Pomaga to zminimalizować ryzyko wystąpienia wycieków pamięci.

Czasami trzeba ustawić obiekt na `null` po to, aby usunąć go z zasięgu. Przykładem może być zmienna statyczna zawierająca referencję do obiektu, którego już nie potrzebujemy.

Dobrym sposobem na czyste używanie jednorazowych obiektów jest również korzystanie z instrukcji `using`. Dzięki niej, gdy obiekt znajdzie się poza zasięgiem, jest automatycznie usuwany. Programista nie musi więc jawnie wywoływać metody `Dispose()`. Przyjrzyjmy się poniższemu fragmentowi kodu:

```
using (var unitOfWork = new UnitOfWork())
{
    // Wykonanie jednostki pracy.
}
    // W tym miejscu obiekt UnitOfWork został zniszczony.
```

Kod definiuje jednorazowy obiekt w instrukcji `using`. Obiekt robi to, do czego został stworzony, pomiędzy otwierającym a zamykającym nawiasem klamrowym, i zostaje automatycznie usunięty po tym, jak sterowanie wyjdzie poza blok oznaczony nawiasami klamrowymi. Nie ma więc potrzeby, aby ręcznie wywoływać metodę `Dispose()`, ponieważ jest ona wywoływana automatycznie.

Unikanie korzystania z metody Finalize()

Podczas korzystania z zasobów niezarządzanych, najlepiej zaimplementować interfejs `IDisposable`, by uniknąć stosowania metody `Finalize()`. Nie ma pewności, kiedy zostaną uruchomione finalizatory. Nie zawsze są one uruchamiane w kolejności, w jakiej tego oczekujemy, czy w ogóle wtedy, gdy się tego spodziewamy. Zamiast metody `Finalize()` lepszym i bardziej niezawodnym sposobem niszczenia niezarządzanych zasobów jest skorzystanie z metody `Dispose()`.

Właściwy poziom abstrakcji

Kod ma właściwy poziom abstrakcji, jeśli do wyższego poziomu udostępniamy tylko to, co powinno być udostępnione, i nie gubimy się w implementacji.

Jeśli gubisz się w szczegółach implementacji, to znaczy, że poziom abstrakcji jest zbyt wysoki. Jeśli okazuje się, że wiele osób musi pracować nad tą samą klasą jednocześnie, to poziom abstrakcji jest zbyt niski. W obu przypadkach potrzebna jest refaktoryzacja, której celem jest doprowadzenie abstrakcji do właściwego poziomu.

Korzystanie z regionów w dużych klasach

Regiony są bardzo przydatne do grupowania elementów w dużych klasach, ponieważ pozwalają na rozwijanie i zwijanie fragmentów kodu. Czytanie kodu rozbudowanej klasy, związane z koniecznością przełączania się pomiędzy metodami, może być dość trudne, dlatego grupowanie w klasie metod, które wywołują się nawzajem, jest dobrym sposobem na ich grupowanie. Podczas pracy nad fragmentem kodu metody te mogą być w razie potrzeby rozwijane i zwijane.

Jak można zobaczyć na podstawie przykładów, z którymi spotkaliśmy się do tej pory, stosowanie dobrych praktyk kodowania sprawia, że powstaje kod, który jest znacznie bardziej czytelny i łatwiejszy do utrzymania. W poniższym punkcie omówimy potrzebę stosowania standardów i reguł kodowania, które poprzemy kilkoma metodami wytwarzania oprogramowania, takimi jak SOLID i DRY.

Potrzeba stosowania standardów kodowania, zasad i metodologii

Większość współczesnego oprogramowania jest efektem pracy wielu zespołów programistów. Jak wiadomo, każdy z nas ma własne, unikatowe sposoby kodowania, a wszyscy mamy jakąś formę ideologii programowania. Toczy się wiele dyskusji dotyczących różnych paradygmatów rozwijania oprogramowania. Wszyscy jednak jesteśmy zgodni co do tego, że życie programistów staje się łatwiejsze, jeśli stosujemy się do ustalonego zbioru standardów kodowania, zasad i metodologii.

Przyjrzyjmy się nieco bardziej szczegółowo temu, co przez nie rozumiemy.

Standardy kodowania

Standardy kodowania określają szereg zakazów i nakazów, których powinniśmy przestrzegać. Takie standardy mogą być egzekwowane za pomocą odpowiednich narzędzi, na przykład FxCop, a także ręcznie, poprzez stosowanie wzajemnych przeglądów kodu. Wszystkie firmy mają własne standardy kodowania, które muszą być przestrzegane przez pracujących w nich programistów. Jednak w realnym świecie obserwujemy sytuacje, kiedy w obliczu zbliżającego

się terminu te standardy kodowania znikają z pola widzenia, a dotrzymanie terminu staje się ważniejsze od rzeczywistej jakości kodu. Zazwyczaj w takich sytuacjach do listy błędów do obsłużenia dodaje się wymagane refaktoryzacje jako dług techniczny, który należy spłacić po opublikowaniu oprogramowania.

Firma Microsoft ma swoje własne standardy kodowania. W większości przypadków są to przyjęte standardy, modyfikowane w zależności od konkretnych potrzeb biznesowych. Oto kilka przykładów standardów kodowania opisanych w internecie:

- *https://www.c-sharpcorner.com/UploadFile/ankurmalik123/C-Sharp-codingstandards/*
- *https://www.dofactory.com/reference/csharp-coding-standards*
- *https://blog.submain.com/coding-standards-c-developers-need/*

Stosowanie standardów kodowania przez programistów z różnych zespołów lub z tego samego zespołu prowadzi do ujednolicenia bazy kodu. Ujednolicona baza kodu staje się znacznie łatwiejsza do czytania, rozszerzania i utrzymania. Zwykle jest również mniej podatna na błędy. A jeśli błędy wystąpią, znalezienie ich jest łatwiejsze, ponieważ kod jest zgodny ze standardowym zestawem wytycznych stosowanych przez wszystkich programistów.

Zasady kodowania

Zasady kodowania to zbiór wytycznych do pisania kodu wysokiej jakości, testowania i debugowania tego kodu oraz jego utrzymania. Zasady mogą się różnić w zależności od konkretnych programistów i zespołów programistów.

Nawet jeśli pracujesz sam, zrobisz sobie wielką przysługę, jeśli zdefiniujesz własne zasady kodowania i będziesz je konsekwentnie stosować. W przypadku pracy w zespole bardzo korzystne jest uzgodnienie zbioru standardów kodowania. Dzięki temu praca nad wspólnym kodem jest łatwiejsza.

W tej książce pokażę i szczegółowo wyjaśnię przykłady zasad kodowania takich jak SOLID, YAGNI, KISS i DRY. Zacznijmy od wyjaśnienia skrótów. SOLID to skrót pochodzący od pierwszych liter pięciu zasad: ***S****ingle Responsibility Principle* (zasada pojedynczej odpowiedzialności), ***O****pen-Closed Principle* (zasada otwarty-zamknięty), ***L****iskov Substitution* (zasada podstawiania Liskov), ***I****nterface Segregation Principle* (zasada segregacji interfejsów) oraz ***D****ependency Inversion Principle* (zasada odwracania zależności). YAGNI to skrót od *You Ain't Gonna Need It* — dosł. nie potrzebujesz tego. KISS to skrót od *Keep It Simple, Stupid* (to ma być proste, głupcze), a DRY oznacza *Don't Repeat Yourself* (dosł. nie powtarzaj się).

Metodologie kodowania

Metodologie kodowania pozwalają rozbić proces tworzenia oprogramowania na kilka predefiniowanych faz. Każda z nich składa się z szeregu działań do wykonania. Różni programiści i zespoły programistów stosują różne metodologie kodowania. Głównym celem stosowania metodologii kodowania jest usprawnienie procesu wytwarzania oprogramowania — od pomysłu, poprzez fazę kodowania, aż do wdrożenia i utrzymania.

W tej książce zaprezentuję metodologie TDD (*Test-Driven Development* — dosł. wytwarzanie oprogramowania sterowane testami), BDD (*Behavioral Driven Development* — dosł. wytwarzanie oprogramowania sterowane zachowaniami) z użyciem frameworka SpecFlow oraz AOP (*Aspect-Oriented Programming* — dosł. programowanie aspektowe) z wykorzystaniem frameworka PostSharp.

Konwencje kodowania

Najlepiej stosować konwencje kodowania Microsoft C#. Można się z nimi zapoznać pod adresem *https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-aprogram/coding-conventions*.

Stosowanie standardów kodowania Microsoftu daje gwarancję pisania kodu w formalnie akceptowanym i uzgodnionym formacie. Stosowanie wspomnianych konwencji kodowania w języku C# pomaga skoncentrować się na czytaniu kodu i pozwala zaoszczędzić czas pracy nad układem kodu. Standardy kodowania Microsoftu promują stosowanie najlepszych praktyk.

Modułowość

Dzielenie dużych programów na mniejsze moduły ma wiele sensu. Małe moduły są łatwe do testowania, łatwiej je wielokrotnie wykorzystać oraz można nad nimi pracować niezależnie od innych modułów. Niewielkie moduły są także łatwiejsze do rozszerzania i utrzymania.

Modułowy program można podzielić na różne komponenty oraz różne przestrzenie nazw w obrębie tych komponentów. Modułowe programy są również znacznie łatwiejsze do pracy w środowiskach złożonych z zespołów, ponieważ różne zespoły mogą pracować nad różnymi modułami.

W ramach tego samego projektu kod dzieli się na moduły poprzez dodanie folderów odpowiadających przestrzeniom nazw. Przestrzeń nazw powinna zawierać wyłącznie kod, który jest związany z jej nazwą. Jeśli zatem mamy — na przykład — przestrzeń nazw `FileSystem`, to w folderze przypisanym do tej przestrzeni nazw powinny się znaleźć wyłącznie typy związane z plikami i katalogami. Podobnie, jeśli mamy przestrzeń nazw `Data`, to powinny się w niej znaleźć wyłącznie typy związane z danymi i źródłami danych.

Kolejną cechą wynikającą z prawidłowej modułowości jest czytelność kodu — jeśli moduły są niewielkie i proste, to czytanie ich jest łatwe. Oprócz pisania kodu większość czasu programisty zajmuje czytanie go i dążenie do jego zrozumienia. Zatem im kod jest mniejszy i lepiej podzielony na moduły, tym czytanie go i zrozumienie staje się łatwiejsze. Prowadzi to do lepszego zrozumienia kodu i ułatwia programistom jego użytkowanie i modyfikowanie.

KISS

Być może jesteś geniuszem programowania komputerów. Być może potrafisz tworzyć kod, który jest tak „seksowny", że inni programiści patrzą na niego z podziwem i ślinią się na swoje klawiatury. Ale czy ci inni programiści potrafią powiedzieć, co robi Twój kod, jeśli tylko rzucą na niego okiem? Jeśli popatrzysz na ten kod po 10 tygodniach pracy nad innym kodem w celu dotrzymania napiętych terminów, to czy będziesz w stanie wyjaśnić z absolutną pewnością, co robi Twój kod i dlaczego zastosowałeś właśnie taką metodę kodowania? Czy przyszło Ci do głowy, że będziesz pracować nad tym kodem później?

Czy kiedykolwiek zdarzyło Ci się zaprogramować jakiś kod, oddać go, a następnie powrócić do niego po więcej niż kilku dniach i pomyśleć: przecież to nie może być mój kod? Co to ma być? Wiem, że to robota moja oraz moich byłych współpracowników.

Podczas programowania kodu należy pamiętać, by zachować jego prosty format tak, by był czytelny i zrozumiały nawet dla początkujących programistów. Młodsi programiści często muszą przeczytać kod, zrozumieć, a następnie go utrzymywać. Im kod jest bardziej skomplikowany, tym czas wdrażania nowych osób do projektu trwa dłużej. Nawet doświadczonym programistom zrozumienie złożonych systemów może stwarzać tak duże trudności, że są skłonni zmienić pracę na mniej wymagającą.

Na przykład, jeśli pracujesz nad prostą stroną internetową, zadaj sobie kilka pytań. Czy naprawdę potrzebujesz mikrousług? Czy projekt typu brownfield, nad którym pracujesz, jest bardzo skomplikowany? Czy istnieje możliwość jego uproszczenia tak, by stał się łatwiejszy do utrzymania? Gdy pracujesz nad nowym systemem zastanów się, jaka jest minimalna liczba jego komponentów wymagana do napisania solidnego, łatwego w utrzymaniu i skalowalnego rozwiązania, które będzie wydajnie działać.

YAGNI

YAGNI to zasada wytwarzania zwinnego oprogramowania, która stanowi, że programista nie powinien dodawać żadnego kodu, jeśli nie jest to absolutnie konieczne. Uczciwy programista pisze testy dla projektu, które nie przechodzą, a następnie pisze tylko tyle kodu produkcyjnego, aby te testy zaczęły przechodzić, po czym refaktoryzuje kod, aby usunąć występujące duplikaty. Stosowanie metodologii tworzenia oprogramowania YAGNI polega na utrzymywaniu liczby klas, metod i ogólnej liczby wierszy kodu na absolutnym minimum.

Podstawowym celem stosowania YAGNI jest zapobieganie nadmiernej złożoności systemów oprogramowania. Nie wprowadzaj złożoności, jeśli nie jest to konieczne. Musisz pamiętać, aby pisać tylko taki kod, który jest niezbędny. Nie pisz kodu, którego nie potrzebujesz, ani takiego, którego jedynym celem jest eksperymentowanie i uczenie się. Kod eksperymentalny i służący do uczenia się utrzymuj w projektach typu „piaskownica" przeznaczonych specjalnie do tych celów.

DRY

Zasada DRY oznacza *Don't Repeat Yourself* — nie powtarzaj się! Jeśli odkryjesz, że piszesz ten sam kodu w wielu miejscach, to z pewnością jest to kandydat do refaktoryzacji. Należy przyjrzeć się kodowi, aby sprawdzić, czy można go uogólnić i umieścić w klasie pomocniczej, która może być użyta w wielu miejscach systemu, albo umieścić go w bibliotece, aby użyć jej w innych projektach.

Jeśli ten sam fragment kodu występuje w wielu miejscach i okaże się, że jest w nim błąd, będziesz zmuszony zmodyfikować kod we wszystkich tych miejscach. W takich sytuacjach bardzo łatwo przeoczyć kod, który wymaga modyfikacji. W rezultacie może dojść do opublikowania kodu, w którym problem został poprawiony w niektórych miejscach, ale wciąż istnieje w innych.

Dlatego dobrym pomysłem jest usunięcie duplikatów w kodzie natychmiast po ich odnalezieniu. Jeśli tego nie zrobisz, możesz natknąć się później na więcej problemów.

SOLID

SOLID to zestaw pięciu zasad projektowych, których celem jest tworzenie oprogramowania łatwiejszego do zrozumienia i utrzymania. Kod oprogramowania powinien być łatwy do czytania i rozszerzania bez konieczności modyfikowania części istniejącego kodu. Oto pięć zasad projektowania SOLID:

- **Zasada pojedynczej odpowiedzialności** (*Single Responsibility Principle*) — klasy i metody powinny odpowiadać za tylko jedno zadanie. Wszystkie elementy, które tworzą pojedynczą odpowiedzialność, powinny być pogrupowane i shermetyzowane.
- **Zasada otwarty-zamknięty** (*Open/Closed Principle*) — klasy i metody powinny być otwarte na rozbudowę i zamknięte dla modyfikacji. Gdy wymagana jest zmiana oprogramowania, powinieneś być w stanie rozszerzyć je bez modyfikowania istniejącego kodu.
- **Zasada podstawiania Liskov** (*Liskov Substitution*) — Twoja funkcja ma wskaźnik do klasy bazowej? Powinna być w stanie wykorzystać dowolną klasę pochodną nie wiedząc o tym, jaka jest jej klasa konkretna.
- **Zasada segregacji interfejsów** (*Interface Segregation Principle* — ISP) — gdy interfejsy są rozbudowane, klienty, które z nich korzystają, nie zawsze wymagają wszystkich metod. W związku z tym, stosując zasadę ISP, można wyodrębnić metody do różnych interfejsów. Dzięki temu zamiast posługiwać się jednym, dużym interfejsem, korzystamy z wielu małych. Klasy mogą następnie implementować interfejsy zawierające tylko te metody, które są niezbędne.

- **Zasada odwracania zależności** (*Dependency Inversion Principle*) — wysokopoziomowy moduł nie powinien zależeć od jakichkolwiek modułów niskopoziomowych. Powinieneś być w stanie swobodnie przełączać się między niskopoziomowymi modułami bez wpływu na używane przez nie moduły wysokopoziomowe. Zarówno moduły wysokopoziomowe, jak i niskopoziomowe, powinny zależeć od abstrakcji.

Abstrakcja nie powinna zależeć od szczegółów, ale szczegóły powinny zależeć od abstrakcji.

Kiedy deklarujesz zmienne, zawsze powinieneś używać typów statycznych takich jak interfejs lub klasa abstrakcyjna. Następnie do zmiennej można przypisywać klasy konkretne, które implementują interfejs lub dziedziczą po klasie abstrakcyjnej.

Brzytwa Ockhama

Zasada brzytwy Ockhama brzmi następująco: nie należy mnożyć bytów bez konieczności. W istocie oznacza to, że najprostsze rozwiązanie zazwyczaj jest również najbardziej prawidłowe. Tak więc w dziedzinie rozwoju oprogramowania łamanie zasady brzytwy Ockhama polega na przyjmowaniu niepotrzebnych założeń oraz wykorzystywaniu zbyt złożonych rozwiązań problemu programistycznego.

Projekty oprogramowania zazwyczaj bazują na zbiorze faktów i założeń. Fakty są łatwe do obsłużenia, inaczej jest z założeniami. Kiedy przystępujesz do pracy nad projektem oprogramowania, zazwyczaj omawiasz problem i jego możliwe rozwiązania w zespole. Przy wyborze rozwiązania zawsze należy wybierać projekt z najmniejszą liczbą założeń, ponieważ jest to najdokładniejszy wybór do implementacji. Jeżeli w projekcie istnieje kilka założeń, to im więcej ich jest, tym większe prawdopodobieństwo, że projekt okaże się wadliwy.

W projekcie z mniejszą liczbą komponentów istnieje mniejsze ryzyko wystąpienia problemów. Zatem by spełnić zasadę brzytwy Ockhama należy dbać o to, by projekty były jak najmniejsze, by nie przyjmować założeń, jeśli nie są konieczne, oraz by korzystać wyłącznie z faktów.

Podsumowanie

W tym rozdziale zaprezentowałem wprowadzenie do dobrego i złego kodu. Mam nadzieję, że po jego lekturze rozumiesz, co oznacza dobry kod. Zamieściłem link do konwencji kodowania Microsoft C#. Zachęcam do ich stosowania i przestrzegania najlepszych praktyk kodowania proponowanych przez Microsoft.

Krótko zaprezentowałem również różne metodologie i zasady tworzenia oprogramowania, w tym DRY, KISS, SOLID, YAGNI oraz brzytwę Ockhama.

Pokazałem ponadto zalety modułowości kodu oraz stosowania przestrzeni nazw i komponentów. Wśród tych korzyści należy wymienić możliwość pracy nad niezależnymi modułami przez oddzielne zespoły, a także możliwość wielokrotnego używania kodu oraz łatwość utrzymania.

W następnym rozdziale przyjrzymy się zagadnieniu przeglądów kodu przez partnerów z zespołu. Czasami mogą one być nieprzyjemne, ale ich stosowanie pomaga przestrzegać firmowych procedur kodowania.

Pytania

1. Jakie są skutki pisania złego kodu?
2. Jakie są niektóre z efektów pisania dobrego kodu?
3. Jakie korzyści wynikają z pisania kodu modułowego?
4. Jaki kod spełnia zasadę DRY?
5. Dlaczego podczas pisania kodu warto stosować zasadę KISS?
6. Co oznacza skrót SOLID?
7. Wyjaśnij zasadę YAGNI.
8. Co to jest brzytwa Ockhama?

Dalsza lektura

- Gary McLean Hall, *Adaptive Code: Agile coding with design patterns and SOLID principles*, wydanie drugie.
- Jeffrey Chilberto i Gaurav Aroraa, *Hands-On Design Patterns with C# and .NET Core*.
- Rob can der Leek, Pascal can Eck, Gijs Wijnholds, Sylvan Rigal i Joost Visser, *Building Maintainable Software, C# Edition*.
- Interesujące informacje na temat antywzorców tworzenia oprogramowania można znaleźć na stronie *https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Architecture/Anti-Patterns*.
- Ciekawe informacje na temat wzorców projektowych, wraz z listą wzorców projektowych oraz linkami do schematów i kodu źródłowego, można znaleźć na stronie *https://en.wikipedia.org/wiki/Software_design_patterns*.

C#. Sekret sukcesu tkwi w czystym i najbardziej przejrzystym kodzie

Język C# cechują dojrzałość, prostota i nowoczesność. Służy on do wielu celów: do tworzenia aplikacji sieciowych, aplikacji działających w chmurze, oprogramowania dla urządzeń mobilnych i internetu rzeczy. Choć pozwala na pisanie kodu bezpiecznego, przejrzystego, wydajnego i prostego w konserwacji, zdarzają się przypadki, gdy jest on tak fatalnej jakości, że uzyskanie właściwej skalowalności i wydajności oprogramowania staje się niemożliwe. W takim wypadku trzeba zidentyfikować i rozwiązać problemy występujące w kodzie. Nie jest to łatwe zadanie.

Dzięki tej książce zrozumiesz znaczenie standardów kodowania, zasad i metodologii. Dowiesz się, czemu służą przeglądy kodu oraz jak przyczyniają się do jego poprawiania i zapewnienia zgodności z uznanymi standardami. Opisano tu także testy jednostkowe, zagadnienia związane z techniką TDD oraz rozwiązywaniem zadań przekrojowych. Zaprezentowano dobre praktyki w zakresie programowania obiektów, struktur danych, obsługiwania wyjątków oraz innych aspektów pisania programów w języku C#. Poszczególne zagadnienia zilustrowano licznymi przykładami działającego kodu C# oraz wyczerpującymi wyjaśnieniami w postaci procedur krok po kroku.

W książce:

- dobre praktyki pisania kodu w C#
- implementacja metodologii *fail-pass-refactor* dla kodu w C#
- wzorce projektowe i ich stosowanie
- rozpoznawanie kodu złej jakości
- zabezpieczanie interfejsów API i usługa Azure Key Vault
- wykorzystywanie narzędzi do profilowania i refaktoryzacji

Jason Alls — od ponad dwóch dekad jest programistą. Specjalizuje się w wykorzystywaniu technologii Microsoftu. Tworzył aplikacje marketingowe GIS, zajmował się bazami danych w sektorze bankowym, a także różnymi aplikacjami desktopowymi, internetowymi i mobilnymi. W 2005 roku zdobył certyfikat MCAD. Obecnie rozwija i utrzymuje oprogramowanie do badania i oceny dysleksji, napisane w ASP.NET, Angularze i C#.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-7725-7
9 788328 377257
INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 89,00 zł
Packt