Microsoft

Adaptywny kod

Zwinne programowanie, wzorce projektowe i SOLID-ne zasady

Wydanie II

Gary McLean Hall

Helion

Tytuł oryginału: Adaptive Code: Agile coding with design patterns and SOLID principles (2nd Edition)

Tłumaczenie: Jakub Hubisz (rozdz. 1 – 2), Andrzej Watrak (wstęp, rozdz. 3 – 13, dodatek)

ISBN: 978-83-283-3870-8

Authorized translation from the English language edition, entitled: ADAPTIVE CODE:
AGILE CODING WITH DESIGN PATTERNS AND SOLID PRINCIPLES, Second Edition;
ISBN 1509302581; by Gary Mclean Hall; published by Pearson Education, Inc., publishing as Microsoft Press.

Wydawnictwo HELION
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)

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

Printed in Poland.

Spis treści

CZĘŚĆ III SOLID-NY KOD

CZĘŚĆ IV STOSOWANIE ADAPTYWNYCH TECHNIK

CZĘŚĆ V DODATKI

ROZDZIAŁ 7.

Zasada pojedynczej odpowiedzialności

Po przeczytaniu tego rozdziału będziesz potrafił:

- opisać zasadę pojedynczej odpowiedzialności;
- identyfikować klasy zawierające zbyt wiele odpowiedzialności;
- dzielić monolityczne klasy na mniejsze, zawierające po jednej odpowiedzialności;
- stosować wzorce rozdzielania odpowiedzialności.

Zasada pojedynczej odpowiedzialności polega na tworzeniu klas, które można zmieniać *tylko z jednego* powodu. Jeżeli klasę można zmieniać z więcej niż jednego powodu, oznacza to, że klasa ta ma więcej niż jedną odpowiedzialność. Tego rodzaju klasy należy dzielić na mniejsze, o pojedynczych odpowiedzialnościach, które można zmieniać tylko z jednego powodu.

Ten rozdział opisuje powyższy proces i pokazuje, jak tworzyć użyteczne klasy o pojedynczych odpowiedzialnościach. Klasa, którą można zmieniać z kilku powodów, powinna delegować jedną lub więcej swoich odpowiedzialności do innych klas.

Nie sposób przecenić znaczenia delegowania funkcjonalności do innych klas. To jedna z podstawowych cech adaptywnego kodu. Kod pozbawiony tej cechy trudno będzie dostosowywać do zmieniających się wymagań, wykorzystując Scruma, *kanban* i inne metodyki zwinnego programowania.

Opis problemu

Aby lepiej opisać problem z klasami posiadającymi wiele odpowiedzialności, w tej części rozdziału wykorzystany jest przykład. Listing 7.1 przedstawia klasę `TradeProcessor` opisaną w rozdziale 6., „Refaktoryzacja”. Jak pamiętasz, klasa ta odczytuje dane z pliku i zapisuje je w bazie. Na tej klasie zostały również wykonane testy charakteryzujące, rejestrujące jej działanie. Na podstawie wyników tych testów powstał złoty wzorzec stanowiący zabezpieczenie przed wprowadzeniem niezamierzonych zmian w procesie refaktoryzacji, mogących negatywnie wpłynąć na działanie kodu.

LISTING 7.1. *Przykładowa klasa zawierająca zbyt wiele odpowiedzialności*

```
public class TradeProcessor
{
  public void ProcessTrades(System.IO.Stream stream)
  {
    // Odczytanie wierszy.
    var lines = new List<string>();
    using(var reader = new System.IO.StreamReader(stream))
    {
      string line;
      while((line = reader.ReadLine()) != null)
      {
        lines.Add(line);
      }
    }
    var trades = new List<TradeRecord>();
    var lineCount = 1;
    foreach(var line in lines)
    {
      var fields = line.Split(new char[] { ',' });
      if (fields.Length != 3)
      {
        Console.WriteLine("UWAGA: Błędny wiersz {0}. Zawiera tylko {1} pole/pola.",
                          lineCount, fields.Length);
        continue;
      }
      if (fields[0].Length != 6)
      {
        Console.WriteLine("UWAGA: Błędna waluta w wierszu {0}: '{1}'",
                          lineCount, fields[0]);
        continue;
      }
      int tradeAmount;
      if (!int.TryParse(fields[1], out tradeAmount))
      {
        Console.WriteLine("UWAGA: Błędna liczba akcji w wierszu {0}: '{1}'",
                          lineCount, fields[1]);
      }
      decimal tradePrice;
      if (!decimal.TryParse(fields[2], out tradePrice))
      {
        Console.WriteLine("UWAGA: Błędna wartość akcji w wierszu {0}: '{1}'",
                          lineCount, fields[2]);
      }
      var sourceCurrencyCode = fields[0].Substring(0, 3);
      var destinationCurrencyCode = fields[0].Substring(3, 3);
      // Obliczenie wartości.
      var trade = new TradeRecord
      {
        SourceCurrency = sourceCurrencyCode,
        DestinationCurrency = destinationCurrencyCode,
        Lots = tradeAmount / LotSize,
        Price = tradePrice
      };
      trades.Add(trade);
```

```
      lineCount++;
    }
    using (var connection = new System.Data.SqlClient.SqlConnection("Data Source=(local);
    ↪Initial Catalog=TradeDatabase;Integrated Security=True"))
    {
      connection.Open();
      using (var transaction = connection.BeginTransaction())
      {
        foreach(var trade in trades)
        {
          var command = connection.CreateCommand();
          command.Transaction = transaction;
          command.CommandType = System.Data.CommandType.StoredProcedure;
          command.CommandText = "dbo.insert_trade";
          command.Parameters.AddWithValue("@sourceCurrency", trade.SourceCurrency);
          command.Parameters.AddWithValue("@destinationCurrency",
                                          trade.DestinationCurrency);
          command.Parameters.AddWithValue("@lots", trade.Lots);
          command.Parameters.AddWithValue("@price", trade.Price);
          command.ExecuteNonQuery();
        }
        transaction.Commit();
      }
      connection.Close();
    }
    Console.WriteLine("INFO: Przetworzonych transakcji: {0}", trades.Count);
  }
  private static float LotSize = 100000f;
}
```

Tego rodzaju kod jest wprawdzie niewielki, jednak często w praktyce zdarza się, że trzeba w nim wprowadzać nowe funkcjonalności i dostosowywać go do zmieniających się wymagań.

Powyższy kod jest przykładem nie tylko klasy posiadającej zbyt wiele odpowiedzialności, ale również pojedynczej metody z tą samą wadą. Czytając uważnie kod, można wyróżnić operacje, jakie ta metoda wykonuje:

1. Odczytuje z obiektu typu `Stream` dane wiersz po wierszu i umieszcza je na liście łańcuchów testowych.
2. Analizuje zawartość poszczególnych pól w wierszu i umieszcza ją na strukturalnej liście obiektów typu `TradeRecord`.
3. Podczas analizy sprawdza poprawność danych i wyświetla w konsoli odpowiednie komunikaty.
4. Przegląda wszystkie obiekty `TradeRecord` i wywołuje procedurę składowaną zapisującą dane w bazie.

Zakres odpowiedzialności polega więc na odczytywaniu strumienia danych, analizowaniu ciągów znaków, weryfikowaniu poprawności danych, wyświetlaniu komunikatów i zapisywaniu danych w bazie. Zasada pojedynczej odpowiedzialności mówi, że każda klasa powinna być tak skonstruowana, aby można ją było zmieniać tylko z jednego powodu. Jak widać, klasę `TradeProcessor` można zmieniać w następujących okolicznościach:

- gdy dane wejściowe nie będą odczytywane z obiektu typu `Stream`, ale z usługi WWW;
- gdy zmieni się format danych wejściowych, np. pojawi się dodatkowe pole zawierające informacje o brokerze transakcji;
- gdy zmienią się reguły weryfikacji poprawności danych;
- gdy zmieni się sposób wyświetlania informacji, ostrzeżeń i błędów; jeżeli kod będzie wykorzystany w usłudze WWW, wtedy wyświetlanie komunikatów w konsoli nie będzie możliwe;
- gdy zmieni się baza danych, np. procedura składowana `insert_trade` (zapisz transakcję) będzie miała dodatkowy argument opisujący brokera, lub gdy dane nie będą zapisywane w relacyjnej bazie danych, ale w bazie dokumentowej, albo baza będzie obsługiwana przez usługę WWW, do której trzeba będzie się odwoływać.

W przypadku pojawienia się dowolnej z powyższych zmian trzeba będzie modyfikować klasę. Co więcej, bez utworzenia kilku wersji tej klasy nie będzie możliwe dostosowanie jej np. do odczytywania danych z innego źródła. Wyobraź sobie kłopoty, jakie będziesz miał ze zmianą klasy, gdy będziesz musiał dodać do niej funkcjonalność zapisywania transakcji w usłudze WWW, ale tylko wtedy, gdy w wierszu poleceń zostanie podany dodatkowy parametr.

Refaktoryzacja poprawiająca czytelność kodu

Pierwszym zadaniem na drodze do uzyskania takiej postaci klasy `TradeProcessor`, aby można ją było zmieniać tylko z jednego powodu, jest podzielenie metody `ProcessTrades()` na mniejsze części, z których każda będzie posiadała tylko jedną odpowiedzialność. Poniżej znajdują się listingi przedstawiające osobne metody powstałe w wyniku refaktoryzacji klasy `TradeProcessor` oraz opisy wprowadzonych zmian.

Listing 7.2 przedstawia metodę `ProcessTrades()`, która nie robi nic poza wywoływaniem innych metod.

LISTING 7.2. *Skrócona do minimum metoda ProcessTrades() wywołująca jedynie inne metody*

```
public void ProcessTrades(System.IO.Stream stream)
{
  var lines = ReadTradeData(stream);
  var trades = ParseTrades(lines);
  StoreTrades(trades);
}
```

W oryginalnym kodzie zostały wyróżnione trzy osobne procesy: odczytywanie danych ze strumienia, konwertowanie odczytanych danych na obiekty typu `TradeRecord` oraz zapisywanie danych w trwałej bazie. Zwróć uwagę, że w powyższym kodzie wynik jednej metody stanowi dane wejściowe dla następnej metody. Nie można wywołać metody `StoreTrades()` (zapisz transakcje w bazie), dopóki nie uzyska się rekordów danych za pomocą metody `ParseTrades()` (analizuj transakcje), która to metoda z kolei wymaga wierszy danych zwróconych przez metodę `ReadTradeData()` (odczytaj dane transakcji).

Przyjrzyjmy się pierwszej w kolejności metodzie, `ReadTradeData()`, przedstawionej na listingu 7.3.

LISTING 7.3. *Metoda ReadTradeData() zawierająca część oryginalnego kodu*

```
private IEnumerable<string> ReadTradeData(System.IO.Stream stream)
{
  var tradeData = new List<string>();
  using (var reader = new System.IO.StreamReader(stream))
  {
    string line;
    while ((line = reader.ReadLine()) != null)
    {
      tradeData.Add(line);
    }
  }
  return tradeData;
}
```

Powyższy kod pochodzi z oryginalnej metody `ProcessTrades()`. Został on umieszczony w osobnej metodzie zwracającej dane w postaci listy łańcuchów testowych. Zwróć uwagę, że zwracane dane można jedynie odczytywać, natomiast w oryginalnej metodzie istniała niepotrzebna możliwość dodawania do listy kolejnych wierszy.

Następna w kolejności jest metoda `ParseTrades()`, przedstawiona na listingu 7.4. Jej kod różni się nieco od oryginalnego, ponieważ część operacji również jest delegowana do innych metod.

LISTING 7.4. *Metoda ParseTrades() deleguje operacje do innych metod, dzięki czemu jest prostsza*

```
private IEnumerable<TradeRecord> ParseTrades(IEnumerable<string> tradeData)
{
  var trades = new List<TradeRecord>();
  var lineCount = 1;
  foreach (var line in tradeData)
  {
    var fields = line.Split(new char[] { ',' });
    if(!ValidateTradeData(fields, lineCount))
    {
      continue;
    }
    var trade = MapTradeDataToTradeRecord(fields);
    trades.Add(trade);
    lineCount++;
  }
  return trades;
}
```

Powyższa metoda deleguje operacje weryfikacji i mapowania danych do innych metod. Gdyby operacje nie były delegowane, kod metody byłby zbyt skomplikowany, jak również obejmowałby zbyt wiele odpowiedzialności. Metoda `ValidateTradeData()` (weryfikuj dane transakcji), przedstawiona na listingu 7.5, zwraca wartość logiczną informującą, czy poszczególne pola wiersza danych zawierają poprawne wartości.

LISTING 7.5. *Cały kod weryfikacyjny umieszczony w osobnej metodzie*

```
private bool ValidateTradeData(string[] fields, int currentLine)
{
  if (fields.Length != 3)
  {
    LogMessage("UWAGA: Błędny wiersz {0}. Zawiera tylko {1} pole/pola.", currentLine,
      fields.Length);
    return false;
  }
  if (fields[0].Length != 6)
  {
    LogMessage("UWAGA: Błędna waluta w wierszu {0}: '{1}'", currentLine,
      fields[0]);
    return false;
  }
  int tradeAmount;
  if (!int.TryParse(fields[1], out tradeAmount))
  {
    LogMessage("UWAGA: Błędna liczba akcji w wierszu {0}: '{1}'",
      currentLine, fields[1]);
    return false;
  }
  decimal tradePrice;
  if (!decimal.TryParse(fields[2], out tradePrice))
  {
    LogMessage("UWAGA: Błędna wartość akcji w wierszu {0}: '{1}'",
      currentLine, fields[2]);
    return false;
  }
  return true;
}
```

Jedyną zmianą wprowadzoną w oryginalnym kodzie jest delegowanie operacji wyświetlania komunikatów do innej metody. Zamiast kilkakrotnie wywoływanej metody `Console.WriteLine()` zastosowano metodę `LogMessage()` (wyświetl komunikat), przedstawioną na listingu 7.6.

LISTING 7.6. *Metoda LogMessage() będąca odpowiednikiem metody Console.WriteLine()*

```
private void LogMessage(string message, params object[] args)
{
  Console.WriteLine(message, args);
}
```

Listing 7.7 przedstawia inną metodę wywoływaną w opisanej wcześniej metodzie `ParseTrades()`. Metoda ta wiąże tabelę łańcuchów testowych, reprezentującą pola danych odczytanych ze strumienia, z obiektem typu `TradeRecord()`.

LISTING 7.7. *Wiązanie oddzielnych typów danych w osobnym kodzie*

```
private TradeRecord MapTradeDataToTradeRecord(string[] fields)
{
  var sourceCurrencyCode = fields[0].Substring(0, 3);
  var destinationCurrencyCode = fields[0].Substring(3, 3);
  var tradeAmount = int.Parse(fields[1]);
```

```
  var tradePrice = decimal.Parse(fields[2]);
  var tradeRecord = new TradeRecord
  {
    SourceCurrency = sourceCurrencyCode,
    DestinationCurrency = destinationCurrencyCode,
    Lots = tradeAmount / LotSize,
    Price = tradePrice
  };
  return tradeRecord;
}
```

Ostatnia metoda utworzona w wyniku refaktoryzacji to `StoreTrades()`, przedstawiona na listingu 7.8. Zawiera ona kod komunikujący się z bazą danych. Ponadto deleguje wyświetlanie komunikatów informacyjnych do opisanej wcześniej metody `LogMessage()`.

LISTING 7.8. *Po utworzeniu metody LogMessage()*
poszczególne odpowiedzialności całej klasy są wyraźnie rozgraniczone

```
private void StoreTrades(IEnumerable<TradeRecord> trades)
{
  using (var connection = new System.Data.SqlClient.SqlConnection("Data Source=(local);
  ➥Initial Catalog=TradeDatabase;Integrated Security=True"))
  {
    connection.Open();
    using (var transaction = connection.BeginTransaction())
    {
      foreach (var trade in trades)
      {
        var command = connection.CreateCommand();
        command.Transaction = transaction;
        command.CommandType = System.Data.CommandType.StoredProcedure;
        command.CommandText = "dbo.insert_trade";
        command.Parameters.AddWithValue("@sourceCurrency", trade.SourceCurrency);
        command.Parameters.AddWithValue("@destinationCurrency",
          trade.DestinationCurrency);
        command.Parameters.AddWithValue("@lots", trade.Lots);
        command.Parameters.AddWithValue("@price", trade.Price);
        command.ExecuteNonQuery();
      }
      transaction.Commit();
    }
    connection.Close();
  }
  LogMessage("INFO: {0} trades processed", trades.Count());
}
```

Podsumowując cały proces refaktoryzacji: widać wyraźnie, że nowy kod jest znacznie lepszy w porównaniu z oryginalnym. Jaki jednak konkretnie cel został osiągnięty? Choć metoda `ProcessTrades()` jest bezdyskusyjnie mniejsza od monolitycznego oryginału i zdecydowanie bardziej czytelna, w rzeczywistości pod względem adaptywności niewiele się zmieniło. Można teraz przekształcać implementację metody `LogMessage()` tak, aby np. zapisywała komunikaty w pliku, zamiast wyświetlać je w konsoli. Byłaby to jednak zmiana klasy `TradeProcessor`, czyli dokładnie to, czego chcemy uniknąć.

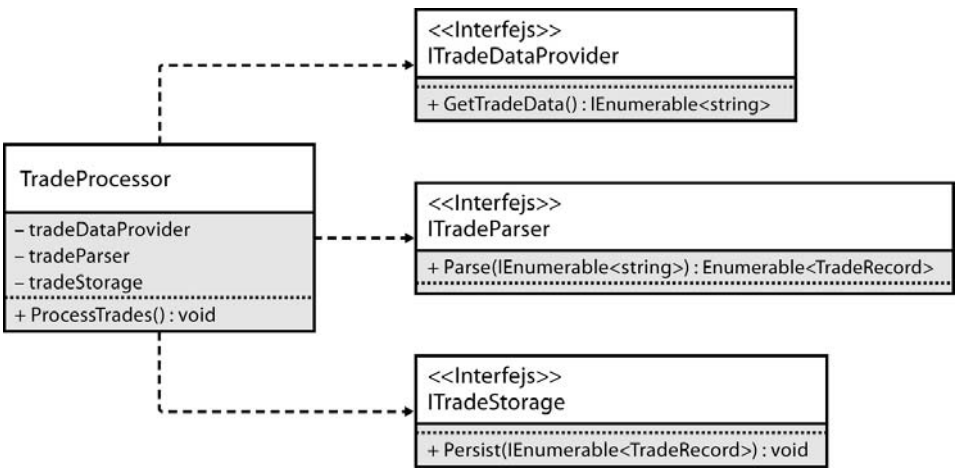
Powyższa refaktoryzacja jest ważnym etapem w procesie faktycznego rozdzielenia odpowiedzialności klasy, ale została ona wykonana w celu zwiększenia czytelności kodu, a nie poprawy jego adaptywności. Kolejnym zadaniem będzie rozdzielenie odpowiedzialności pomiędzy różne klasy i interfejsy. Celem jest osiągnięcie rzeczywistej adaptywności kodu.

Refaktoryzacja zwiększająca abstrakcyjność kodu

Kolejna refaktoryzacja stworzonej wyżej klasy `TradeProcessor` polega na utworzeniu kilku abstrakcji, które będą umożliwiały wprowadzanie w tej klasie niemal dowolnych zmian. Choć kod klasy może się wydawać bardzo prosty, a nawet mało znaczący, jest praktycznym przykładem, idealnym na potrzeby tego rozdziału. Ponadto bardzo często zdarza się, że mała aplikacja, taka jak ta, rozrasta się do ogromnych rozmiarów. Gdy zacznie z niej korzystać kilka osób, coraz częściej będzie się pojawiać potrzeba wprowadzania zmian w kodzie.

Często w odniesieniu do takich pozornie małych aplikacji używa się terminów *prototyp* lub *test koncepcji* (ang. *proof of concept*). Przekształcenie prototypu w docelową wersję aplikacji jest dość proste. Refaktoryzacja zwiększająca abstrakcyjność kodu jest ważną operacją w tworzeniu adaptywnego kodu. Jeżeli się jej nie przeprowadzi, wtedy wszystkie żądania wprowadzenia zmian będą wrzucane „do jednego worka", co spowoduje utworzenie klasy, grupy klas lub zestawu o bliżej niezdefiniowanych odpowiedzialnościach i niewyraźnej abstrakcyjności. W efekcie powstanie aplikacja bez testów jednostkowych, którą trudno będzie utrzymywać i rozwijać, co może mieć krytyczny wpływ na działalność firmy.

Pierwszym krokiem w refaktoryzacji zwiększającej abstrakcyjność klasy `TradeProcessor` jest utworzenie jednego lub kilku interfejsów realizujących trzy ogólne operacje odczytywania, przetwarzania i zapisywania danych o transakcjach. Rysunek 7.1 przedstawia pierwszy zestaw takich abstrakcji.



RYSUNEK 7.1. *Klasa TradeProcessor będzie teraz uzależniona od trzech interfejsów*

Ponieważ w wyniku pierwszej refaktoryzacji cały kod metody `ProcessTrades` został przeniesiony do nowych metod, będzie można teraz łatwo zdefiniować pierwsze abstrakcje. Zgodnie z zasadą pojedynczej odpowiedzialności powyższe trzy operacje muszą być wykonywane w osobnych klasach. Jak już wiesz, należy unikać tworzenia bezpośrednich zależności pomiędzy klasami i zamiast nich stosować interfejsy. Dlatego powyższe trzy odpowiedzialności zostaną umieszczone w trzech osobnych interfejsach. Listing 7.9 przedstawia klasę `TradeProcessor` po dokonaniu powyższych zmian.

LISTING 7.9. *Klasa TradeProcessor jest teraz implementacją samego procesu i niczego więcej*

```
public class TradeProcessor
{
  public TradeProcessor(ITradeDataProvider tradeDataProvider, ITradeParser tradeParser,
    ITradeStorage tradeStorage)
  {
    this.tradeDataProvider = tradeDataProvider;
    this.tradeParser = tradeParser;
    this.tradeStorage = tradeStorage;
  }
  public void ProcessTrades()
  {
    var lines = tradeDataProvider.GetTradeData();
    var trades = tradeParser.Parse(lines);
    tradeStorage.Persist(trades);
  }
  private readonly ITradeDataProvider tradeDataProvider;
  private readonly ITradeParser tradeParser;
  private readonly ITradeStorage tradeStorage;
}
```

Nowa klasa znacznie różni się od swego poprzedniego wcielenia. Nie zawiera już szczegółów implementacyjnych całego procesu, tylko jego *zarys*. Klasa ta modeluje proces przekształcania danych z jednego formatu na inny. Jest to jedyna odpowiedzialność tej klasy i tylko z tego jednego powodu może być zmieniana. Jeżeli zmieni się sam proces, należy odpowiednio zmienić klasę. Jeżeli jednak postanowisz, że dane nie będą już odczytywane ze strumienia, komunikaty nie będą wyświetlane w konsoli albo dane transakcji nie będą zapisywane w bazie, wtedy nie będziesz musiał zmieniać powyższej klasy.

Interfejsy, od których teraz jest uzależniona klasa `TradeProcessor`, znajdują się w osobnych zestawach. Dzięki temu ani kod kliencki, ani zestawy nie odwołują się wzajemnie do siebie. Ponadto w osobnych zestawach znajdują się klasy implementujące powyższe interfejsy: `StreamTradeDataProvider`, `SimpleTradeParser` i `AdoNetTradeStorage`. Zwróć uwagę na przyjętą konwencję nazw tych klas. Przede wszystkim w miejscu prefiksu `I` stosowany jest opis kontekstu klasy. Zatem na podstawie nazwy `StreamTradeDataProvider` można się domyślić, że klasa ta stanowi implementację interfejsu `ITradeDataProvider` odczytującego dane z obiektu typu `Stream`. Klasa `AdoNetTradeStorage` wykorzystuje bibliotekę ADO.NET do zapisywania danych w bazie. Nazwa klasy implementującej interfejs `ITradeParser` zawiera prefiks `Simple` (prosty) oznaczający, że klasa ta nie zawiera żadnych zależności.

Wszystkie powyższe klasy mogą się znajdować w jednym zestawie, ponieważ są uzależnione jedynie od podstawowych klas platformy .NET. Gdybyś zamierzał wykorzystać innego rodzaju zależności, np.: od klas zewnętrznych, własnych lub innych niż podstawowe klasy platformy .NET, wtedy implementacje powinieneś umieścić w osobnych zestawach. Gdybyś zamiast ADO.NET zastosował bibliotekę Dapper, powinieneś utworzyć zestaw o nazwie `Services.Dapper`, a w nim klasę `DapperTradeStorage` implementującą interfejs `ITradeStorage`.

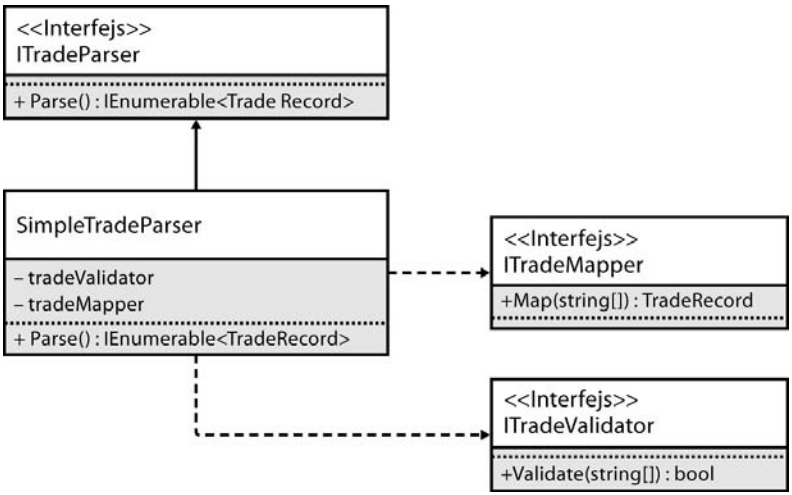
Interfejs `ITradeDataProvider` nie jest uzależniony od klasy `Stream`. Pierwsza wersja metody odczytującej dane ze strumienia miała argument typu `Stream`, ale w ten sposób metoda uzyskiwała zależność. Jeżeli tworzysz interfejsy i refaktoryzujesz kod w celu zwiększenia jego abstrakcyjności, ważne jest likwidowanie zależności, ponieważ pogarszają one adaptywność kodu. Możliwość odczytywania danych z innych źródeł niż strumień została już opisana wcześniej, zatem w wyniku nowej refaktoryzacji powyższa zależność została usunięta. Zamiast tego obiekt typu `Stream` jest podawany w argumencie konstruktora klasy `StreamTradeDataProvider`, a nie jej metody. W ten sposób można wykorzystywać w klasie dowolne zależności bez modyfikowania interfejsu. Listing 7.10 przedstawia implementację klasy `StreamTradeDataProvider`.

LISTING 7.10. *Kontekst klasy można umieścić w argumencie jej konstruktora, dzięki czemu interfejs może być prosty*

```
public class StreamTradeDataProvider : ITradeDataProvider
{
  public StreamTradeDataProvider(Stream stream)
  {
    this.stream = stream;
  }
  public IEnumerable<string> GetTradeData()
  {
    var tradeData = new List<string>();
    using (var reader = new StreamReader(stream))
    {
      string line;
      while ((line = reader.ReadLine()) != null)
      {
        tradeData.Add(line);
      }
    }
    return tradeData;
  }
  private Stream stream;
}
```

Pamiętaj, że klasa `TradeProcessor` zawiera kod kliencki, który „nie zna” (i nie powinien znać) niczego poza metodą `GetTradeData()` interfejsu `ITradeDataProvider`, w tym szczegółów implementacyjnych tej metody.

Z opisywanego kodu można wyodrębnić więcej abstrakcji. Jak pamiętasz, oryginalna metoda `ParseTrades()` deleguje do innych metod operacje weryfikowania i mapowania danych. Zrefaktoryzuj w opisany wyżej sposób klasę `SimpleTradeParser` tak, aby zawierała tylko jedną odpowiedzialność. Rysunek 7.2 przedstawia diagram UML-a takiej abstrakcji.



RYSUNEK 7.2. *Klasa SimpleTradeParser również została podzielona na mniejsze klasy o pojedynczych odpowiedzialnościach*

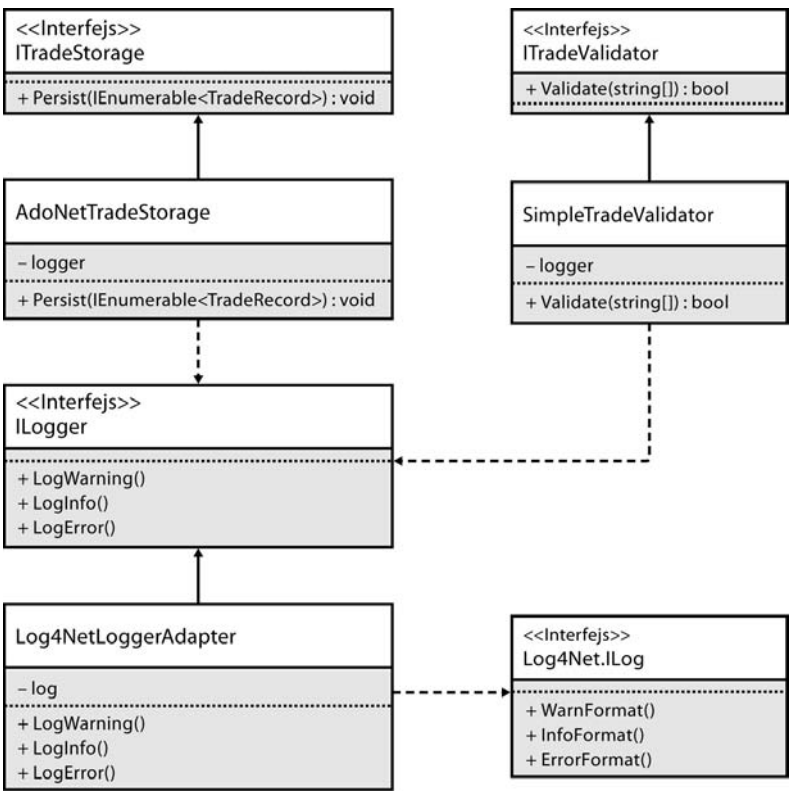
Proces dzielenia odpowiedzialności między interfejsy (i implementujące je klasy) jest procesem rekurencyjnym. Analizując każdą klasę, musisz określać jej odpowiedzialności i dzielić ją na mniejsze klasy dotąd, aż każda z nich będzie zawierała tylko jedną odpowiedzialność. Listing 7.11 przedstawia klasę `SimpleTradeParser` delegującą wszelkie operacje do odpowiednich interfejsów. Jedynym powodem, dla którego niezbędne będzie wprowadzenie zmian w tej klasie w przyszłości, jest zmieniona ogólna struktura danych transakcji, np. gdy pola nie będą oddzielone przecinkami, tylko znakami tabulacji, lub dane będą zapisywane w formacie XML-a.

LISTING 7.11. *Algorytm analizy danych transakcji jest zawarty w implementacji interfejsu ITradeParser*

```
public class SimpleTradeParser : ITradeParser
{
  public SimpleTradeParser(ITradeValidator tradeValidator, ITradeMapper tradeMapper)
  {
    this.tradeValidator = tradeValidator;
    this.tradeMapper = tradeMapper;
  }
  public IEnumerable<TradeRecord> Parse(IEnumerable<string> tradeData)
  {
    var trades = new List<TradeRecord>();
    var lineCount = 1;
    foreach (var line in tradeData)
    {
      var fields = line.Split(new char[] { ',' });
      if (!tradeValidator.Validate(fields))
      {
        continue;
      }
      var trade = tradeMapper.Map(fields);
      trades.Add(trade);
      lineCount++;
```

```
    }
    return trades;
  }
  private readonly ITradeValidator tradeValidator;
  private readonly ITradeMapper tradeMapper;
}
```

Ostatnia refaktoryzacja ma na celu utworzenie abstrakcji wykorzystywanej w dwóch innych klasach do wyświetlania komunikatów. Implementacje interfejsów `ITradeValidator` i `ITradeStorage` wyświetlają komunikaty bezpośrednio w konsoli. Tym razem zamiast własnej klasy utworzysz adapter wykorzystujący popularną bibliotekę Log4Net. Rysunek 7.3 przedstawia diagram UML-a całego kodu.



RYSUNEK 7.3. *Dzięki adapterowi nie trzeba się odwoływać do biblioteki Log4Net w każdym zestawie*

Utworzenie klasy `Log4NetLoggerAdapter` niesie taką korzyść, że zależność zewnętrzna jest zamieniana na zależność wewnętrzną. Zwróć uwagę, że klasy `AdoNetTradeStorage` i `SimpleTradeValidator` są uzależnione od interfejsu `ILogger`. Jednak w trakcie działania aplikacji klasy te odwołują się do biblioteki Log4Net. Odwołania do biblioteki Log4Net znajdują się jedynie w punkcie wejścia aplikacji (więcej informacji na ten temat znajdziesz w rozdziale 12., „Wstrzykiwanie zależności") i nowo utworzonym zestawie `Service.Log4Net`. Wszelki kod odwołujący się do biblioteki Log4Net powinien się znajdować w powyższym zestawie. W tej chwili jest w nim tylko adapter.

Wskazówka. Przekształcanie w opisany wyżej sposób zależności zewnętrznych w wewnętrzne nie zawsze jest konieczne. W przypadku często wykonywanych operacji, takich jak wyświetlanie komunikatów, w zależności od okoliczności może być korzystniejsze odwoływanie się za każdym razem do zewnętrznej biblioteki.

Listing 7.12 przedstawia zrefaktoryzowaną klasę weryfikującą dane, która nie odwołuje się w żaden sposób do konsoli. Biblioteka Log4Net dzięki swej elastyczności umożliwia rejestrowanie komunikatów niemal w dowolny sposób. Zatem pod względem rejestrowania komunikatów kod jest w pełni adaptywny.

LISTING 7.12. *Klasa SimpleTradeValidator po refaktoryzacji*

```
public class SimpleTradeValidator : ITradeValidator
{
  private readonly ILogger logger; public SimpleTradeValidator(ILogger logger)
  {
    this.logger = logger;
  }
  public bool Validate(string[] tradeData)
  {
    if (tradeData.Length != 3)
    {
      logger.LogWarning("Błędny wiersz. Zawiera tylko {1} pole/pola.",
        tradeData.Length);
      return false;
    }
    if (tradeData[0].Length != 6)
    {
      logger.LogWarning("Błędna waluta: '{1}'", tradeData[0]);
      return false;
    }
    int tradeAmount;
    if (!int.TryParse(tradeData[1], out tradeAmount))
    {
      logger.LogWarning("Błędna liczba akcji: '{1}'", tradeData[1]);
      return false;
    }
    decimal tradePrice;
    if (!decimal.TryParse(tradeData[2], out tradePrice))
    {
      logger.LogWarning("Błędna wartość akcji: '{1}'",
        tradeData[2]);
      return false;
    }
    return true;
  }
}
```

W tym miejscu należy zrobić krótkie podsumowanie. Zwróć uwagę, że nie wprowadziłeś w kodzie żadnych zmian wpływających na jego funkcjonalność. Możesz to sprawdzić, wykonując test z użyciem złotego wzorca, chroniącego przed popełnieniem pomyłek podczas refaktoryzacji. Kod pod względem

funkcjonalnym jest taki sam jak na początku. Jednak gdy pojawi się potrzeba udoskonalenia go, będziesz mógł to łatwo zrobić. Możliwość łatwego dostosowywania kodu do nowych zadań jest nagrodą za wysiłek włożony w jego refaktoryzację.

Wskazówka. Refaktoryzując kod w opisany wyżej sposób, sprawdzaj, czy robisz to zgodnie z zasadą „przede wszystkim testy". Dzięki temu będziesz mógł testować kod nie tylko za pomocą złotego wzorca.

Przypomnij sobie listę potencjalnych usprawnień kodu. W nowej wersji będziesz mógł wprowadzać wszystkie usprawnienia bez modyfikowania istniejących klas.

- *Zmiana: dane wejściowe mają nie być odczytywane z obiektu typu* `Stream`*, ale z usługi WWW.*
  - Rozwiązanie: utwórz nową klasę implementującą interfejs `ITradeDataProvider` i odczytującą dane z usługi WWW.
- *Zmiana: inny format danych wejściowych, np. dodatkowe pole zawierające informację o brokerze transakcji.*
  - Rozwiązanie: zmień klasy implementujące interfejsy `ITradeDataValidator`, `ITradeDataMapper` i `ITradeStorage` tak, aby przetwarzały nowe pole.
- *Zmiana: nowe reguły weryfikacji poprawności danych.*
  - Rozwiązanie: dostosuj implementację interfejsu `ITradeDataValidator` do nowych reguł.
- *Zmiana: inny sposób wyświetlania informacji, ostrzeżeń i błędów; jeżeli kod będzie wykorzystany w usłudze WWW, wtedy wyświetlanie komunikatów w konsoli nie będzie możliwe.*
  - Rozwiązanie: jak pisałem wcześniej, adapter wykorzystujący bibliotekę Log4Net oferuje szerokie możliwości rejestrowania komunikatów.
- *Zmiana: nowa baza danych, np. procedura składowana* `insert_trade` *zawiera dodatkowy argument opisujący brokera, lub dane nie są zapisywane w relacyjnej bazie danych, ale w bazie dokumentowej, albo baza jest obsługiwana przez usługę WWW, do której trzeba się odwoływać.*
  - Rozwiązanie: jeżeli zmieni się procedura składowana, będziesz musiał w klasie `AdoNetTradeStorage` uwzględnić dane brokera. W pozostałych dwóch przypadkach możesz utworzyć klasę `MongoTradeStorage`, zapisującą dane transakcji w bazie MongoDB, oraz klasę `WebServiceTradeStorage`, odwołującą się do usługi WWW.

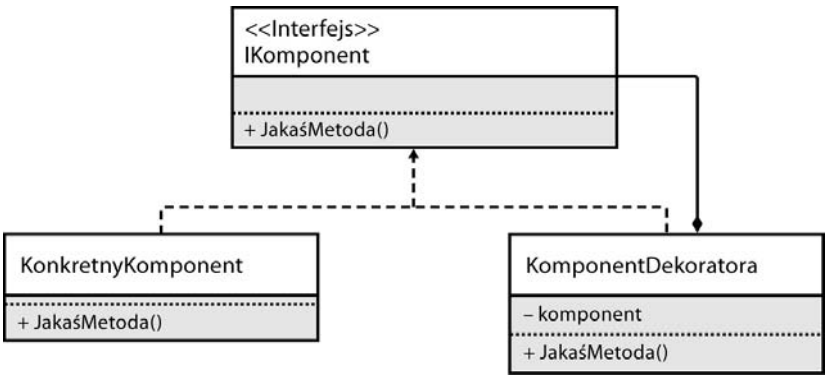
Mam nadzieję, że przekonałem Cię, że interfejsy, rozprzężone zestawy, agresywna refaktoryzacja i zasada pojedynczej odpowiedzialności stanowią podstawy tworzenia adaptywnego kodu. Gdy przekształcisz swój kod tak, że wykonywane operacje będą delegowane do abstrakcji, wtedy możliwości rozbudowy kodu będą wręcz nieograniczone.

Pozostała część rozdziału poświęcona jest innym aspektom zasady pojedynczej odpowiedzialności.

Zasada pojedynczej odpowiedzialności i wzorzec Dekorator

Wzorzec Dekorator doskonale sprawdza się podczas tworzenia klas o pojedynczych odpowiedzialnościach. Często jednak zdarza się, że klasa posiada wiele ściśle ze sobą związanych odpowiedzialności, których rozdzielenie między nowe klasy nie jest jednoznacznie zasadne.

Podstawową zasadą wzorca Dekorator jest tworzenie klasy, której konstruktor posiada argumenty tego samego typu co ta klasa. Jest to wygodne rozwiązanie, ponieważ nowe funkcjonalności kodu można dodawać, tworząc nowe implementacje określonych interfejsów, jak również sama klasa stanowi implementację wymaganego interfejsu. Rysunek 7.4 przedstawia diagram UML-a wzorca Dekorator.



RYSUNEK 7.4. *Diagram UML-a wzorca Dekorator*

Listing 7.13 przedstawia prosty kod utworzony według wzorca Dekorator. Nie jest on na tyle duży, aby było w nim konieczne zastosowanie tego wzorca, ale stanowi doskonały jego przykład.

LISTING 7.13. *Przykładowy kod utworzony według wzorca Dekorator*

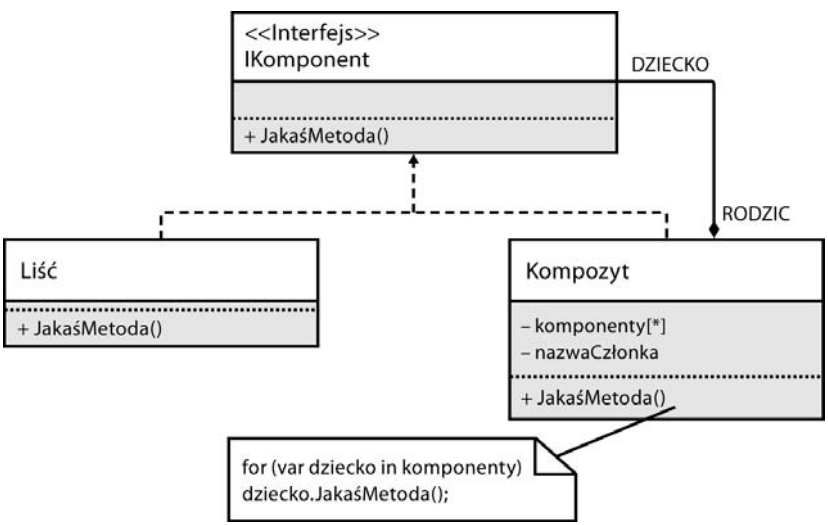
```
public interface IKomponent
{
  void JakaśMetoda();
}
//...
public class KonkretnyKomponent : IKomponent
{
  public void JakaśMetoda()
  {
  }
}
//...
public class KomponentDekoratora : IKomponent
{
  private readonly IKomponent udekorowanyKomponent;
  public KomponentDekoratora(IKomponent udekorowanyKomponent)
  {
    this.udekorowanyKomponent = udekorowanyKomponent;
  }
```

```
  public void JakaśMetoda()
  {
    InnaMetoda();
    udekorowanyKomponent.JakaśMetoda();
  }
  private void InnaMetoda()
  {
  }
}
//...
class Program
{
  static IKomponent komponent;
  static void Main(string[] args)
  {
    komponent = new KomponentDekoratora(new KonkretnyKomponent());
    komponent.JakaśMetoda();
  }
}
```

Ponieważ argumentem metody klienckiej jest klasa, można w nim podać oryginalną lub udekorowaną wersję klasy. Zwróć uwagę, że dla kodu klienckiego nie robi to żadnej różnicy. Nie trzeba go zmieniać odpowiednio do użytej wersji klasy.

Wzorzec Kompozyt

Kompozyt jest specjalną i jedną z najczęściej stosowanych odmian wzorca Dekorator. Rysunek 7.5 przedstawia jego diagram UML-a.



RYSUNEK 7.5. *Wzorzec Kompozyt jest bardzo podobny do wzorca Dekorator*

Stosując wzorzec Kompozyt, można tworzyć wiele różnych instancji interfejsu i odwoływać się do nich tak, jakby były jedną instancją. Kod kliencki może wykorzystywać jedną instancję interfejsu, ale niejawnie można stosować wiele innych instancji bez konieczności zmieniania kodu. Listing 7.14 przedstawia zastosowanie wzorca Kompozyt w praktyce.

LISTING 7.14. *Interfejs utworzony według wzorca Kompozyt*

```
public interface IKomponent
{
  void JakaśMetoda();
}
//...
public class Liść : IKomponent
{
  public void JakaśMetoda()
  {
  }
}
//...
public class KompozytowyKomponent : IKomponent
{
  public KompozytowyKomponent()
  {
    dzieci = new List<IKomponent>();
  }
  public void DodajKomponent(IKomponent komponent)
  {
    dzieci.Add(komponent);
  }
  public void UsuńKomponent(IKomponent komponent)
  {
    dzieci.Remove(komponent);
  }
  public void JakaśMetoda()
  {
    foreach (var dziecko in dzieci)
    {
      dziecko.JakaśMetoda();
    }
  }
  private ICollection<IKomponent> dzieci;
}
//...
class Program
{
  static void Main(string[] args)
  {
    var kompozyt = new KompozytowyKomponent();
    kompozyt.DodajKomponent(new Liść());
    kompozyt.DodajKomponent(new Liść());
    kompozyt.DodajKomponent(new Liść());
    komponent = kompozyt;
    komponent.JakaśMetoda();
  }
  static IKomponent komponent;
}
```

Klasa `KompozytowyKomponent` zawiera metody dodające i usuwające instancje interfejsu `IKomponent`. Metody te nie należą do interfejsu, tylko bezpośrednio do klasy. Każda metoda lub klasa fabryczna, która tworzy instancję klasy `KompozytowyKomponent`, musi również tworzyć udekorowane instancje tej klasy i umieszczać je w argumencie metody `Add()`. W przeciwnym wypadku należałoby zmienić kod kliencki wykorzystujący interfejs `IKomponent` tak, aby obsługiwał klasy kompozytowe.

Przy każdorazowym wywołaniu metody `JakaśMetoda()` przez kod kliencki wykorzystujący interfejs `IKomponent` wywoływana jest metoda `JakaśMetoda()` każdej klasy kompozytowej umieszczonej na liście. W ten sposób za pomocą jednej instancji interfejsu `IKomponent` — klasy `KompozytowyKomponent` — można odwoływać się do wielu innych instancji.

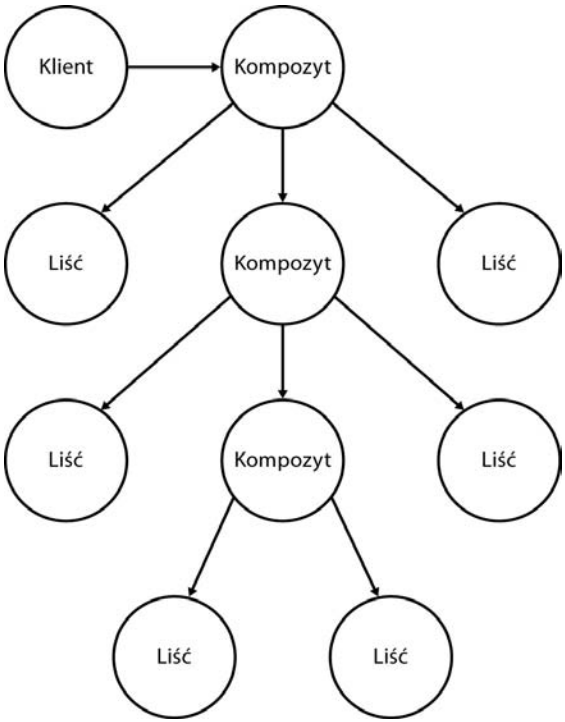
Każda instancja klasy umieszczona na liście zawartej w klasie `KompozytowyKomponent` musi być implementacją interfejsu `IKomponent`, zgodnie ze ścisłym typowaniem kompilatora języka C#. Jednak nie muszą to być instancje tej samej konkretnej klasy. Dzięki polimorfizmowi wszystkie implementacje interfejsu można traktować jak jego instancje. W przykładzie przedstawionym na listingu 7.15 do listy w klasie `KompozytowyKomponent` dodawane są instancje różnych klas, co jeszcze lepiej dowodzi przydatności tego wzorca.

LISTING 7.15. *Do klasy kompozytowej można dodawać instancje różnych klas*

```
public class DrugiTypLiścia : IKomponent
{
  public void JakaśMetoda()
  {
  }
}
//...
public class TrzeciTypLiścia : IKomponent
{
  public void JakaśMetoda()
  {
  }
}
//...
public void AlternatywnyKompozyt()
{
  var kompozyt = new KompozytowyKomponent();
  kompozyt.DodajKomponent(new Liść());
  kompozyt.DodajKomponent(new DrugiTypLiścia());
  kompozyt.DodajKomponent(new TrzeciTypLiścia());
  komponent = kompozyt;
  kompozyt.JakaśMetoda();
}
```

Podsumowując: dzięki wzorcowi Kompozyt można za pomocą metody `Add()` tworzyć łańcuch instancji klasy `KompozytowyKomponent`.

W rozdziale 3., „Zależności i warstwy”, zależności między klasami były modelowane za pomocą grafów. Ten model wykorzystany jest również w tym rozdziale do zaprezentowania wzorca Kompozyt. Węzły na rysunku 7.6 reprezentują instancje klasy, a krawędzie — wywoływane metody.



RYSUNEK 7.6. *Graf ilustrujący strukturę działającej aplikacji*

Dekoratory predykatu

Dekorator predykatu jest przydatną konstrukcją pozwalającą ukryć przed kodem klienckim warunkowe wykonywanie kodu. Listing 7.16 przedstawia odpowiedni przykład.

LISTING 7.16. *Kod kliencki wywołuje metodę JakaśMetoda() tylko w parzyste dni miesiąca*

```
public class TesterDaty
{
  public bool DziśJestParzystyDzieńMiesiąca
  {
    get
    {
      return DateTime.Now.Day % 2 == 0;
    }
  }
}
//...
class PrzykładDekoratoraPredykatu
{
  public PrzykładDekoratoraPredykatu(IKomponent komponent)
  {
    this.komponent = komponent;
  }
```

```
  public void Start()
  {
    TesterDaty testerDaty = new TesterDaty();
    if (testerDaty.DziśJestParzystyDzieńMiesiąca)
    {
      komponent.JakaśMetoda();
    }
  }
  private readonly IKomponent komponent;
}
```

Klasa `TesterDaty` jest nową zależnością, której należy się pozbyć. W pierwszym odruchu kod można przekształcić do postaci pokazanej na listingu 7.17. Jest to jednak częściowe rozwiązanie problemu.

LISTING 7.17. *Ulepszenie kodu polegające na umieszczeniu zależnej klasy w argumencie metody*

```
class PrzykładDekoratoraPredykatu
{
  public PrzykładDekoratoraPredykatu(IKomponent komponent)
  {
    this.komponent = komponent;
  }
  public void Start(TesterDaty testerDaty)
  {
    if (testerDaty.DziśJestParzystyDzieńMiesiąca)
    {
      komponent.JakaśMetoda();
    }
  }
  private readonly IKomponent komponent;
}
```

Teraz metoda `Start()` ma argument, przez co kod kliencki musi tworzyć instancję klasy `TesterDaty`. Stosując wzorzec Dekorator, możesz pozostawić kod kliencki bez zmian i zachować warunkowe wykonywanie kodu. Listing 7.18 przedstawia lepsze rozwiązanie problemu.

LISTING 7.18. *Dekorator predykatu zawiera zależność, a kod kliencki jest znacznie prostszy*

```
public class KomponentPredykatu : IKomponent
{
  public KomponentPredykatu(IKomponent udekorowanyKomponent, TesterDaty testerDaty)
  {
    this.udekorowanyKomponent = udekorowanyKomponent;
    this.testerDaty = testerDaty;
  }
  public void JakaśMetoda()
  {
    if (testerDaty.DziśJestParzystyDzieńMiesiąca)
    {
      udekorowanyKomponent.JakaśMetoda();
    }
  }
  private readonly IKomponent udekorowanyKomponent;
  private readonly TesterDaty testerDaty;
}
```

```
//...
class PrzykładDekoratoraPredykatu
{
  public PrzykładDekoratoraPredykatu(IKomponent komponent)
  {
    this.komponent = komponent;
  }
  public void Start()
  {
    komponent.JakaśMetoda();
  }
  private readonly IKomponent komponent;
}
```

Zwróć uwagę, że w powyższym kodzie została wprowadzona instrukcja warunkowa, ale kod kliencki ani oryginalna klasa nie zostały zmodyfikowane. W tym przykładzie klasa `TesterDaty` jest zależnością, jednak możesz pójść krok dalej i zdefiniować własny interfejs, ogólnie rozwiązujący problem. Po wprowadzeniu kilku zmian kod wygląda jak na listingu 7.19.

LISTING 7.19. *Dzięki zdefiniowaniu interfejsu IPredykat powstaje bardziej uniwersalne rozwiązanie*

```
public interface IPredykat
{
  bool Test();
}
//...
public class KomponentPredykatu : IKomponent
{
  public KomponentPredykatu(IKomponent udekorowanyKomponent, IPredykat predykat)
  {
    this.udekorowanyKomponent = udekorowanyKomponent;
    this.predykat = predykat;
  }
  public void JakaśMetoda()
  {
    if (predykat.Test())
    {
      udekorowanyKomponent.JakaśMetoda();
    }
  }
  private readonly IKomponent udekorowanyKomponent;
  private readonly IPredykat predykat;
}
//...
public class PredykatDziśJestParzystyDzieńMiesiąca : IPredykat
{
  public PredykatDziśJestParzystyDzieńMiesiąca(TesterDaty testerDaty)
  {
    this.testerDaty = testerDaty;
  }
  public bool Test()
  {
    return testerDaty.DziśJestParzystyDzieńMiesiąca;
  }
  private readonly TesterDaty testerDaty;
}
```

Klasa `PredykatDziśJestParzystyDzieńMiesiąca` przekształca oryginalną zależność od klasy `TesterDaty` na zależność od interfejsu `IPredykat`. Jest to przykład zastosowania wzorca Adapter opisanego w sekcji „Refaktoryzacja zwiększająca abstrakcyjność kodu" tego rozdziału oraz w rozdziale 4., „Interfejsy i wzorce projektowe".

Uwaga. Platforma .NET od wersji 2.0 zawiera delegat `Predicate<T>` modelujący predykat z jednym generycznym argumentem opisującym kontekst. W tym przykładzie nie użyłem tego delegata z dwóch powodów. Po pierwsze, nie trzeba tu określać żadnego kontekstu, ponieważ oryginalna metoda testowa nie ma argumentów. Mogłem jednak użyć delegata `Func<bool>` do zamodelowania predykatu bezkontekstowego, ale pojawił się drugi powód: *delegaty nie są tak uniwersalne jak interfejsy*. Modelując interfejs `IPredykat`, mogłem go udekorować w sposób, jaki będę stosował w nowych interfejsach w przyszłości. Innymi słowy: zdefiniowałem nowy punkt rozszerzający, który można w nieskończoność dekorować.

Dekoratory warunkowe

Możesz rozbudować dekorator predykatu o argument zawierający interfejs, którego kod będzie wykonywany przez instrukcję warunkową, gdy testowane wyrażenie będzie miało wartość `false`, jak na listingu 7.20.

LISTING 7.20. *Argumentami dekoratora instrukcji warunkowej są dwa komponenty i predykat*

```
public class KomponentWarunkowy : IKomponent
{
  public KomponentWarunkowy(IKomponent prawdziwyKomponent,
    IKomponent fałszywyKomponent, IPredykat predykat)
  {
    this.prawdziwyKomponent = prawdziwyKomponent;
    this.fałszywyKomponent = fałszywyKomponent;
    this.predykat = predykat;
  }
  public void JakaśMetoda()
  {
    if (predykat.Test())
    {
      prawdziwyKomponent.JakaśMetoda();
    }
    else
    {
      fałszywyKomponent.JakaśMetoda();
    }
  }
  private readonly IKomponent prawdziwyKomponent;
  private readonly IKomponent fałszywyKomponent;
  private readonly IPredykat predykat;
}
```

Za każdym razem, gdy predykat będzie miał wartość `true`, będzie wywoływana metoda instancji interfejsu `prawdziwyKomponent`, a gdy będzie miał wartość `false` — metoda instancji interfejsu `fałszywyKomponent`.

Leniwe dekoratory

Za pomocą leniwego dekoratora kod kliencki może wykorzystywać interfejs, którego instancja zostanie utworzona dopiero przy pierwszym odwołaniu do niej. Zazwyczaj w kodzie klienckim błędnie stosuje się ten dekorator, umieszczając w argumencie metody instancję klasy `Lazy<T>`, jak na listingu 7.21.

LISTING 7.21. *W tym kodzie klienckim wykorzystywana jest klasa Lazy<T>*

```
public class KomponentKliencki
{
  public KomponentKliencki(Lazy<IKomponent> komponent)
  {
    this.komponent = komponent;
  }
  public void Start()
  {
    komponent.Value.JakaśMetoda();
  }
  private readonly Lazy<IKomponent> komponent;
}
```

W tego typu kodzie klienckim nie można użyć innych niż leniwe instancje interfejsu `IKomponent`. Jeżeli jednak wrócisz do bardziej standardowego przypadku użycia interfejsu, możesz utworzyć taki dekorator, że kod kliencki nie będzie „wiedział", że wykorzystuje klasę `Lazy<T>`. Dzięki temu w niektórych obiektach typu `KomponentKliencki` będziesz mógł stosować inne niż leniwe instancje interfejsu `IKomponent`. Listing 7.22 przedstawia tego typu dekorator.

LISTING 7.22. *Klasa LeniwyKomponent jest leniwą instancją interfejsu IKomponent, lecz klasa KomponentKliencki o tym „nie wie"*

```
public class LeniwyKomponent : IKomponent
{
  public LeniwyKomponent(Lazy<IKomponent> leniwyKomponent)
  {
    this.leniwyKomponent = leniwyKomponent;
  }
  public void JakaśMetoda()
  {
    leniwyKomponent.Value.JakaśMetoda();
  }
  private readonly Lazy<IKomponent> leniwyKomponent;
}
//...
public class KomponentKliencki
{
  public KomponentKliencki(IKomponent komponent)
  {
    this.komponent = komponent;
  }
  public void Start()
  {
  komponent.JakaśMetoda();
  }
  private readonly IKomponent komponent;
}
```

Dekoratory logujące

Listing 7.23 przedstawia typowy kod zawierający wiele instrukcji wyświetlających komunikaty. Instrukcje te są wszechobecne, przez co kod aplikacji jest mało czytelny.

LISTING 7.23. *Instrukcje logujące zaciemniają przeznaczenie metod*

```
public class KonkretnyKalkulator : IKalkulator
{
  public int Dodaj(int x, int y)
  {
    Console.WriteLine("Dodaj(x={0}, y={1})", x, y);
    var suma = x + y;
    Console.WriteLine("wynik={0}", suma);
    return suma;
  }
}
```

Zamiast mnożyć w kodzie instrukcje logujące, można wykorzystać dekorator do utworzenia jednego zestawu zawierającego jedną taką instrukcję, jak na listingu 7.24.

LISTING 7.24. *Za pomocą dekoratora logującego można zdefiniować tylko jedną instrukcję logującą i uprościć główny kod*

```
public class LogującyKalkulator : IKalkulator
{
  public LogującyKalkulator(IKalkulator kalkulator)
  {
    this.kalkulator = kalkulator;
  }
  public int Dodaj(int x, int y)
  {
    Console.WriteLine("Dodaj(x={0}, y={1})", x, y);
    var suma = kalkulator.Dodaj(x, y);
    Console.WriteLine("wynik={0}", suma);
    return suma;
  }
  private readonly IKalkulator kalkulator;
}
//...
public class KonkretnyKalkulator : IKalkulator
{
  public int Dodaj(int x, int y)
  {
    return x + y;
  }
}
```

Kod kliencki może wykorzystywać interfejs `IKalkulator` z różnymi parametrami, którego metody mogą zwracać lub nie zwracać wartości. Ponieważ klasa `LogującyKalkulator` może przechwytywać metody obu rodzajów, robi to bezpośrednio. Jest to pewne ograniczenie dekoratora logującego, o którym należy pamiętać. Przede wszystkim każda prywatna właściwość udekorowanej klasy jest niedostępna dla dekoratora logującego, więc nie można wyświetlać jej wartości. Ponadto trzeba tworzyć

dekoratory logujące dla wszystkich interfejsów, co może być pracochłonnym zadaniem. Dlatego często wykonywaną operację lepiej jest kodować za pomocą aspektu logującego. Programowanie aspektowe zostało opisane w rozdziale 3.

Dekoratory profilujące

Jednym z głównych powodów stosowania platformy .NET jest jej zgodność z metodyką RAD (ang. *Rapid Application Development* — szybkie tworzenie aplikacji). Dzięki tej platformie gotową aplikację można utworzyć znacznie szybciej niż za pomocą niskopoziomowego języka, np. C++. Jest to możliwe m.in. dzięki automatycznemu zarządzaniu pamięcią i bogatej liście bibliotek. Panuje przekonanie, że za pomocą języka C# kod aplikacji tworzy się szybko, jednak kod ten działa wolno. Natomiast kodowanie w języku C++ jest wolniejsze, ale aplikacje działają szybciej.

Choć aplikacje oparte na platformie .NET również działają szybko, mogą zawierać słabe punkty. W jaki sposób stwierdzić, która część kodu działa wolno? Dzięki *profilowaniu* metod można uzyskać informacje o częściach kodu działających wolniej niż inne części. Rozważmy listing 7.25.

LISTING 7.25. *Kod z założenia działający wolno*

```
public class WolnyKomponent : IKomponent
{
  public WolnyKomponent()
  {
    losowaLiczba = new Random((int)DateTime.Now.Ticks);
  }
  public void JakaśMetoda()
  {
    for (var i = 0; i < 100; ++i)
    {
      Thread.Sleep(losowaLiczba.Next(i) * 10);
    }
  }
  private readonly Random losowaLiczba;
}
```

Metoda `JakaśMetoda()` w powyższym przykładzie działa wolno. Określenia *szybko* i *wolno* można różnie interpretować w różnych okolicznościach. W tym przypadku wolna metoda to taka, która wykonuje się dłużej niż jedną sekundę. W jaki sposób sprawdzić, czy dana metoda jest wolna? Można zmierzyć czas, jaki upływa od rozpoczęcia jej wykonywania do zakończenia, w sposób pokazany na listingu 7.26.

LISTING 7.26. *Za pomocą klasy System.Diagnostics.Stopwatch można mierzyć czas wykonywania metody*

```
public class WolnyKomponent : IKomponent
{
  public WolnyKomponent()
  {
    losowaLiczba = new Random((int)DateTime.Now.Ticks);
    stoper = new Stopwatch();
  }
  public void JakaśMetoda()
  {
```

```
    stoper.Start();
    for (var i = 0; i < 100; ++i)
    {
      System.Threading.Thread.Sleep(losowaLiczba.Next(i) * 10);
    }
    stoper.Stop();
    Console.WriteLine("Metoda wykonywała się przez {0} sekund",
      stoper.ElapsedMilliseconds / 1000);
  }
  private readonly Random losowaLiczba;
  private readonly Stopwatch stoper;
}
```

W powyższym kodzie za pomocą klasy `Stopwatch` z zestawu `System.Diagnostics` mierzony jest czas wykonywania metody. Zwróć uwagę, że stoper jest uruchamiany na początku metody `JakaśMetoda()` i zatrzymywany na jej końcu.

Oczywiście powyższe operacje można wykonać za pomocą dekoratora profilującego. Na listingu 7.27 interfejs jest udekorowany w całości, a stoper jest uruchamiany przed wywołaniem metody udekorowanej instancji. Gdy metoda kończy działanie, stoper jest zatrzymywany i następuje powrót do kodu klienckiego.

LISTING 7.27. *Dekorator profilujący*

```
public class ProfilowanyKomponent : IKomponent
{
  public ProfilowanyKomponent(IKomponent udekorowanyKomponent)
  {
    this.udekorowanyKomponent = udekorowanyKomponent;
    stoper = new Stopwatch();
  }
  public void JakaśMetoda()
  {
    stoper.Start();
    udekorowanyKomponent.JakaśMetoda();
    stoper.Stop();
    Console.WriteLine("Metoda wykonywała się przez {0} sekund",
      stoper.ElapsedMilliseconds / 1000);
  }
  private readonly IKomponent udekorowanyKomponent;
  private readonly Stopwatch stoper;
}
```

Klasę `ProfilowanyKomponent` można dodatkowo zmienić tak, aby w transparentny sposób wyświetlała komunikaty profilujące. W tym celu najpierw należy kod mierzący czas umieścić w osobnym interfejsie, dzięki czemu będzie można stosować jego różne implementacje, włącznie z dekoratorami. Jest to pierwszy krok, który często wykonuje się podczas rozdzielania odpowiedzialności. Listing 7.28 przedstawia kod po wykonaniu tego kroku.

LISTING 7.28. *Przed zastosowaniem dekoratora trzeba klasy zastąpić interfejsami*

```
public class ProfilowanyKomponent : IKomponent
{
  public ProfilowanyKomponent(IKomponent udekorowanyKomponent, IStoper stoper)
  {
```

```
    this.udekorowanyKomponent = udekorowanyKomponent;
    this.stoper = stoper;
  }
  public void JakaśMetoda()
  {
    stoper.Start();
    udekorowanyKomponent.JakaśMetoda();
    var milisekundy = stoper.Stop();
    Console.WriteLine("Metoda wykonywała się przez {0} sekund", milisekundy / 1000);
  }
  private readonly IKomponent udekorowanyKomponent;
  private readonly IStoper stoper;
}
```

Teraz, gdy klasa `ProfilowanyKomponent` nie jest bezpośrednio uzależniona od klasy `System.Diagnostics.Stopwatch`, można stosować dowolne implementacje interfejsu `IStoper`. Na listingu 7.29 widoczny jest dekorator `StoperLogujący`, wzbogacający implementacje powyższego interfejsu o funkcjonalności logowania.

LISTING 7.29. *Dekorator StoperLogujący jest wyświetlającą komunikaty implementacją interfejsu IStoper*

```
public class StoperLogujący : IStoper
{
  public StoperLogujący(IStoper udekorowanyStoper)
  {
    this.udekorowanyStoper = udekorowanyStoper;
  }
  public void Start()
  {
    udekorowanyStoper.Start();
    Console.WriteLine("Uruchomienie stopera...");
  }
  public long Stop()
  {
    var milisekundy = udekorowanyStoper.Stop();
    Console.WriteLine("Zatrzymanie stopera po {0} sekundach",
    TimeSpan.FromMilliseconds(milisekundy).TotalSeconds);
    return milisekundy;
  }
  private readonly IStoper udekorowanyStoper;
}
```

Oczywiście potrzebna jest nieudekorowana implementacja interfejsu `IStoper`, która działa jak prawdziwy stoper. Można to osiągnąć, odwołując się do klasy `System.Diagnostics.Stopwatch` platformy .NET, jak na listingu 7.30.

LISTING 7.30. *Podstawowa implementacja interfejsu IStoper wykorzystuje klasę Stopwatch*

```
public class AdapterStoper : IStoper
{
  public AdapterStoper(Stopwatch stoper)
  {
    this.stoper = stoper;
  }
```

```
  public void Start()
  {
    stoper.Start();
  }
  public long Stop()
  {
    stoper.Stop();
    var milisekundy = stoper.ElapsedMilliseconds;
    stoper.Reset();
    return milisekundy;
  }
  private readonly Stopwatch stoper;
}
```

Zwróć uwagę, że implementacja interfejsu `IStoper` wykorzystuje klasę `System.Diagnostics.Stopwatch` i jej metody `Start()` i `Stop()`. Jednak ponieważ metoda `Start()` wznawia działanie stopera po jego zatrzymaniu, to aby stoper odliczał czas zawsze od zera, należy po zatrzymaniu go metodą `Stop()` odczytać wartość właściwości `ElapsedMilliseconds`, a następnie zresetować stoper, wywołując metodę `Reset()`. Jest to inny przykład zastosowania wzorca Adapter.

Dekorowanie właściwości i zdarzeń

W tym rozdziale zobaczyłeś, jak dekorować metody interfejsu, ale co z jego właściwościami i zdarzeniami? Te elementy, oprócz autowłaściwości i autozdarzeń, również można dekorować. W tym celu należy je jawnie zdefiniować.

Listing 7.31 przedstawia klasę ze zdefiniowaną właściwością, która jednak nie odwołuje się do odpowiadającego jej pola, ale wywołuje metody `get` i `set` udekorowanej instancji interfejsu.

LISTING 7.31. *Właściwości, podobnie jak metody, również można dekorować*

```
public class KomponentDekoratora : IKomponent
{
  public KomponentDekoratora(IKomponent udekorowanyKomponent)
  {
    this.udekorowanyKomponent = udekorowanyKomponent;
  }
  public string Właściwość
  {
    get
    {
      // Po odczytaniu właściwości można ją modyfikować za pomocą dodatkowych instrukcji.
      return udekorowanyKomponent.Właściwość;
    }
    set
    {
      // Podobnie tutaj przed zapisaniem wartości.
      udekorowanyKomponent.Właściwość = value;
    }
  }
  private readonly IKomponent udekorowanyKomponent;
}
```

Listing 7.32 przedstawia klasę ze zdefiniowanym zdarzeniem, która jednak nie odwołuje się do odpowiadającego mu pola, ale wywołuje metody `add` i `remove` udekorowanej instancji interfejsu.

LISTING 7.32. *Wzorzec Dekorator można stosować nie tylko w odniesieniu do metod, ale również zdarzeń*

```
public class KomponentDekoratora : IKomponent
{
  public KomponentDekoratora(IKomponent udekorowanyKomponent)
  {
    this.udekorowanyKomponent = udekorowanyKomponent;
  }
  public event EventHandler Zdarzenie
  {
    add
    {
      // Po instrukcji rejestrującej zdarzenie można wpisać dodatkowy kod.
      udekorowanyKomponent.Zdarzenie += value;
    }
    remove
    {
      // Podobnie tutaj po wyrejestrowaniu zdarzenia.
      udekorowanyKomponent.Zdarzenie -= value;
    }
  }
  private readonly IKomponent udekorowanyKomponent;
}
```

Podsumowanie

Zasada pojedynczej odpowiedzialności ma ogromny pozytywny wpływ na adaptywność kodu. Kod utworzony zgodnie z tą zasadą, w porównaniu z odpowiadającym mu zwykłym kodem, składa się z większej liczby mniejszych i bardziej wyspecjalizowanych klas. Zamiast jednej lub kilku powiązanych wzajemnymi zależnościami dużych klas o skomplikowanych odpowiedzialnościach uzyskuje się uporządkowany, czytelny kod.

Stosowanie zasady pojedynczej odpowiedzialności polega przede wszystkim na dzieleniu kodu na interfejsy i delegowaniu do nich poszczególnych odpowiedzialności. Niektóre wzorce, zwłaszcza Adapter i Dekorator, doskonale wpisują się w tę zasadę. Pierwszy z nich pozwala tworzyć kod zawierający przede wszystkim wewnętrzne zależności od interfejsów, nad którymi programista ma kontrolę, choć w rzeczywistości interfejsy te zawierają zewnętrzne zależności. Natomiast wzorzec Dekorator stosuje się wtedy, gdy z klasy trzeba usunąć pewne funkcjonalności, które są jednak zbyt silnie związane z daną klasą, aby umieszczać je w osobnych klasach.

W tym rozdziale nie zostało opisane wykorzystanie tworzonych w powyższy sposób abstrakcji podczas działania aplikacji. Przedstawione zostały oczywiste przykłady umieszczania interfejsów w argumentach konstruktorów klas, jednak w rozdziale 12. opisanych jest więcej sposobów wykorzystania abstrakcji.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

X

Z

Kod adaptywny: SOLID-ny, elastyczny i łatwy w utrzymaniu!

Tworzenie oprogramowania nie może być procesem powolnym. Dziś zespoły projektowe muszą cechować się elastycznością i dynamiką działania, aby odnieść sukces. Wymagania stawiane kodowi mogą wielokrotnie się zmieniać podczas projektu. Oznacza to, że wprowadzanie zmian do kodu powinno być proste i możliwie mało pracochłonne. Deweloperzy, którzy wdrożą zasady programowania zwinnego i będą przestrzegać związanych z nimi dobrych praktyk, będą w stanie sprostać tym wymaganiom.

Niniejsza książka jest przeznaczona dla średnio zaawansowanych programistów, którzy dobrze opanowali tworzenie kodu zorientowanego obiektowo i chcą przyswoić sobie najlepsze praktyki programistyczne. Książka stanowi pomost między teorią a praktyką, ułatwiający wdrożenie w codziennej pracy wzorców projektowych, zasad SOLID, testów jednostkowych czy refaktoringu. Wyjaśniono tu stosowanie zawiłych reguł, takich jak „otwarte – zamknięte", a także zasad podstawienia Liskov, metod wstrzykiwania zależności czy zwiększania adaptywności kodu za pomocą interfejsów. Przedstawiono również pewne antywzorce projektowe wraz ze wskazówkami, w jaki sposób można ich uniknąć i zapewnić potrzebną funkcjonalność bez utraty elastyczności kodu.

Najważniejsze zagadnienia:

- metodologie Scrum i *kanban*
- zależności i warstwy architektury kodu
- testy i refaktoring
- odwracanie zależności
- wiązanie i spójność kodu

Gary McLean Hall
jest programistą i architektem oprogramowania. Jest cenionym konsultantem, który specjalizuje się w dobrych wzorcach i praktykach programistycznych. Pracował w wielu zespołach ukierunkowanych na tworzenie adaptywnego kodu w takich firmach jak Eidos, Xerox, Nephila Capital czy The LateRooms Group. W swojej pracy zawsze szukał złotego środka pomiędzy tworzeniem funkcjonalnego produktu i wysokiej jakości jego kodu źródłowego.

ISBN 978-83-283-3870-8
9 788328 338708
Cena: 69,00 zł

Microsoft Press