Wyższa Szkoła Bankowa
Wydział Finansów i Zarządzania
Informatyka Inżynierska
Studia II stopnia, semestr 2/3
Gdańsk

Zaawansowane techniki obiektowe Laboratorium

v0.6.0

Autor:
Jakub Głuszek

11 lutego 2022

Spis treści

1 Zasady SOLID

W publikacji `Design Principles and Design Patterns` Robert C. Martin opisał 5 zasad SOLID, które pomagają uzyskać lepszy, bardziej zrozumiały i łatwiejszy w utrzymaniu oraz rozbudowie kod. Są one również trzonem metodologi takich jak Agile czy programowanie adaptywne. W kolejnych podrozdziałach zostaną one po kolei opisane.

1.1 SRP - Zasada pojedynczej odpowiedzialności (ang. SOLID)

Pierwsza z pięciu zasad mówi, że każdy moduł, zestaw, klasa czy metoda powinna posiadać pojedynczy obszar odpowiedzialności (przyczynę zmian). W momencie zmian wymagań, określa się jakie klasy są odpowiedzialne za dane wymagania i to te kasy się modyfikuje. Jeśli istnieją co najmniej dwa niezależne powody mogące wymusić zmianę w danej klasie, to dana klasa rozciąga się na więcej niż jeden obszar odpowiedzialności. W przypadku naruszenia zasady SRP zmiany w dwóch różnych wymaganiach implikują modyfikację w jednej, tej samej klasie.

Zasada pojedynczej odpowiedzialności

Żadna klasa nie może być modyfikowana z więcej niż jednego powodu.

Wyobraźmy sobie sytuację, w której klasa rysująca na ekranie figurę geometryczną np. `Rectangle` jest odpowiedzialna zarówno za wspomniane rysowanie, ale również za wyznaczanie pola powierzchni danej figury. Jakie konsekwencje to za sobą niesie? Jeśli dwa zestawy np. interfejsu użytkownika `UI` oraz jakiś inny korzystający jedynie z operacji liczenia pola powierzchni będą związane z klasą `Rectangle` to zmiana wynikająca z wymagań związanych z `UI` może pociągać ryzyko nieprawidłowego działania w drugim zestawie. Aby nie naruszyć zasady SRP należałoby stworzyć dwie osobne klasy: jedną rysująca figurę i drugą liczącą pole powierzchni. Pierwsza z nich mogłaby wykorzystywać tę pierwszą.

Oczywiście należy zawsze mieć na uwadze, że rozdrabniać klasy na coraz mniejsze typy można prawie w nieskończoność. Może to prowadzić do nadmiernego skomplikowania projektu. Konieczne jest branie pod uwagę wymagań i ich potencjalnych zmian.

Jako przykład przeanalizujmy interfejs `IModem`.

```
public interface IModem
{
  void Dial(string phoneNumber);
  void Hangup();
  void Send(char c);
  char Recieve();
}
```

Listing 1: Naruszenie zasady SRP

Widać, że interfejs mógłby zostać podzielony na dwa obszary odpowiedzialności. Jeden odpowiadający za nawiązywanie i zrywanie połączeń, drugi natomiast za same funkcje komunikacyjne. Klasy klienckie mogłyby wykorzystywać klasę implementującą dwa interfejsy np. `IDataChannel` i `IConnection` i tym samym uzależnić się tylko od jednego z obszarów odpowiedzialności.

1.1.1 Zadanie 1

Poniżej przedstawiony fragment kodu[1] zawiera klasę2 z metodą `ProcessTrades`, która wyraźnie narusza zasadę SRP. Utwórz nowy projekt .NET 5.0 i dodaj do niego poniższą klasę w osobnym pliku. Nazwij ten plik `TradeProcessor.cs`.

```
public class TradeProcessor
{
  public void ProcessTrades(Stream stream)
  {
    // read rows
    var lines = new List<string>();
    using (var reader = new 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("WARN: Line {0} malformed. Only {1} field(s) found.",
        lineCount, fields.Length);
        continue;
      }
      if (fields[0].Length != 6)
      {
        Console.WriteLine("WARN: Trade currencies on line {0} malformed: '{1}'", lineCount, fields[0]);
        continue;
      }
      if (!int.TryParse(fields[1], out int tradeAmount))
      {
        Console.WriteLine("WARN: Trade amount on line {0} not a valid integer: '{1}'", lineCount, fields[1]);
      }
      if (!decimal.TryParse(fields[2], out decimal tradePrice))
      {
        Console.WriteLine("WARN: Trade price on line {0} not a valid decimal: '{1}'", lineCount, fields[2]);
      }
      var sourceCurrencyCode = fields[0].Substring(0, 3);
      var destinationCurrencyCode = fields[0].Substring(3, 3);
      // calculate values
      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: {0} trades processed", trades.Count);
    }
    private readonly float LotSize = 100000f;
}
```

Listing 2: Naruszenie zasady SRP

Dodatkowo utwórz klasę typu `TradeRecord`:

```
public class TradeRecord
{
  public string SourceCurrency { get; set; }
  public string DestinationCurrency { get; set; }
  public float Lots { get; set; }
  public decimal Price { get; set; }
}
```

Listing 3: Klasa `TradeRecord`

W powyżej klasie można na pierwszy rzut oka wydzielić trzy odpowiedzialności:

1. odczytywanie dane,
2. konwertowanie danych na obiekty typu `TradeRecord`,
3. utrwalanie danych.

Dodaj do projektu trzy interfejsy w trzech osobnych plikach:

```
public interface ITradeDataProvider
{
  IEnumerable<string> GetTradeData();
}
```

```
public interface ITradeParser
{
  IEnumerable<TradeRecord> Parse(IEnumerable<string> lines);
}
```

```
public interface ITradeStorage
{
  void Persist(IEnumerable<TradeRecord> trades);
}
```

Każdy z powyższych interfejsów będzie odpowiadał jednej z trzech odpowiedzialności klasy `TradeProcessor`. Docelowo każda zostanie umieszczona w osobnej klasie.

Następnie utwórz trzy klasy implementujące odpowiednio trzy powyższe interfejsy:

```
public class StreamTradeDataProvider : ITradeDataProvider
{
  private readonly Stream stream;
```

```
  public StreamTradeDataProvider(Stream stream) => this.stream = stream;

  public IEnumerable<string> GetTradeData()
  {
    //...
  }
}
```

```
public class SimpleTradeParser : ITradeParser
{
  public IEnumerable<TradeRecord> Parse(IEnumerable<string> lines)
  {
    //...
  }
}
```

```
public class DbTradeStorage : ITradeStorage
{
  public void Persist(IEnumerable<TradeRecord> trades)
  {
    //..
  }
}
```

Przenieś fragmenty kodu z metody `ProcessTrades` klasy `TradeProcessor` do powyższych metod. Do klasy `TradeProcessor` dodaj konstruktor, przyjmujący trzy obiekty implementujące powyższe interfejsy. Dodaj do `TradeProcessor` trzy prywatne pola i przypisz do nich przekazane przez konstruktor argumenty. W metodzie `ProcessTrades` wykorzystaj obiekty przechowywane w tych polach w celu przetworzenia sprzedaży.

```
public class TradeProcessor
{
  public TradeProcessor(ITradeDataProvider tradeDataProvider,
  ITradeParser tradeParser,
  ITradeStorage tradeStorage)
  {
    this.tradeDataProvider = tradeDataProvider;
    this.tradeParser = tradeParser;
    this.tradeStorage = tradeStorage;
  }

  public void ProcessTrades()
  {
    // read rows
    var lines = tradeDataProvider.GetTradeData();
    var trades = tradeParser.Parse(lines);
    tradeStorage.Persist(trades);
  }

  private readonly ITradeDataProvider tradeDataProvider;
  private readonly ITradeParser tradeParser;
  private readonly ITradeStorage tradeStorage;
}
```

Z metody `Parse` znajdującej się w klasie `SimpleTradeParser` można wydzielić dodatkowe dwie odpowiedzialności: walidacji oraz mapowania pól na obiekt typu `TradeRecord`. Dodaj do projektu dodatkowe dwa interfejsy:

```
public interface ITradeValidator
{
```

```
  bool Validate(string[] fields, int lineCount);
}
```

oraz

```
public interface ITradeMapper
{
  TradeRecord Map(string[] fields);
}
```

Dalej utwórz dwie nowe klasy implementujące te interfejsy:

```
public class SimpleTradeValidator : ITradeValidator
{
  public bool Validate(string[] fields, int lineCount)
  {
    //...
  }
}
```

```
public class SimpleMapper : ITradeMapper
{
  public TradeRecord Map(string[] fields)
  {
    //...
  }
}
```

Analogicznie jak wcześniej przenieś fragmenty kodu z metody `Parse` klasy `SimpleTradeParser` do odpowiednich klas. W klasie `SimpleTradeParser` dodaj konstruktor przyjmujący obiekty implementujące powyższe interfejsy. Przypisz te argumenty do prywatnych pól klasy. Na koniec wykorzystaj te obiekty w metodzie `Parse`.

```
public class SimpleTradeParser : ITradeParser
{
  private readonly ITradeValidator tradeValidator;
  private readonly ITradeMapper tradeMapper;

  public SimpleTradeParser(ITradeValidator tradeValidator, ITradeMapper
    tradeMapper)
  {
    this.tradeValidator = tradeValidator;
    this.tradeMapper = tradeMapper;
  }

  //...
}
```

Na koniec można się pokusić o dodanie interfejsu `ILogger`, tak aby uniezależnić klasy od sposobu logowania informacji o błędach.

```
public interface ILogger
{
  void LogWarning(string message, params object[] args);
  void LogInfo(string message, params object[] args);
  void LogError(string message, params object[] args);
}
```

I klasy go implementującej:

```
public class CustomLogger : ILogger
{
```

```
3   public void LogError(string message, params object[] args) => Console.
      WriteLine("ERROR: " + message, args);
4   public void LogInfo(string message, params object[] args) => Console.
      WriteLine("INFO: " + message, args);
5   public void LogWarning(string message, params object[] args) => Console.
      WriteLine("WARN: " + message, args);
6 }
```

Teraz można dodać parametr w konstruktorze klas: `SimpleTradeValidator` i `DbTradeStorage` typu `ILogger` i przypisać go prywatnych pól tych klas. Dalej należy zamienić wywołania metody `Console.WriteLine`, odpowiednimi metodami obiektu implementującego `ILogger.`

Dzięki powyższym krokom będzie istniała możliwość zmiany rejestracji komunikatów w przypadku zmiany wymagań. Wystarczy dodać nową klasę implementującą interfejs `ILogger`, albo wykorzystać wzorzec Adapter3.1 do dostosowania innych, już istniejących implementacji

Przeprowadzone zmiany nie zmieniły funkcjonalności kodu. Jednak można teraz go zdecydowanie łatwiej rozszerzać, dostosowywać do nowych wymagań. Łatwo można zmienić źródło danych wejściowych np. na z usługi WWW, wystarczy napisać nową klasę implementującą `ITradeDataProvider`. Analogicznie łatwo będzie można zmienić format danych wejściowych, reguły walidacji, sposób logowania informacji, ostrzeżeń czy błędów.

1.2 OCP - Zasada otwarte - zamknięte (ang. SOLID)

Pisząc elastyczny kod powinniśmy być w stanie wprowadzać nowe funkcjonalności dodając nowe klasy jednocześnie **nie** modyfikując tych już istniejących. Nie zawsze jest to możliwe, ale zawsze warto próbować.

Zasada otwarte - zamknięte

Składniki oprogramowania (klasy, moduły, funkcje itp.) powinny być otwarte na rozbudowę, ale zamknięte dla modyfikacji.

Aby osiągnąć ten cel oprogramowanie powinno posiadać jasno zdefiniowane, niezmienne abstrakcje. Jeśli dany moduł wykorzystuje jedynie abstrakcje, nie powinna być konieczna jego modyfikacja w przypadku zmian wymagań, rozszerzania funkcjonalności. Stosowanie zasady OCP wymaga wykorzystania polimorfizmu.

Przeanalizujmy poniższy diagram UML. W tym przypadku zmiana obiektu `Order` wymagałaby zmian w klasie klienta.

Rysunek 1: Diagram UML, gdzie `Client` narusza zasadę OCP.

Stosując wzorzec projektowy strategia4.2, który zostanie omówiony na kolejnych zajęciach mamy możliwość odwrócenia zależności. Jeśli w wyniku rozszerzania, zmian funkcjonalności okaże się, że klasa `Order` powinna zostać zamieniona, będzie można utworzyć nową wersję klasy implementującej interfejs zdefiniowany przez klienta i ją podmienić bez modyfikacji w samej klasie `Client`.

Wyobraźmy sobie moduł odpowiedzialny za rysowanie figur geometrycznych. Parametrem jednej z jego metod (`DrawerShape`) jest kolekcja obiektów. Jeśli w wyniku zmian konieczne okaże się dodanie nowej klasy np. `Diamond` wymagane będą zmiany w klasie `Drawer`.

```
1 public class Circle { }
2 public class Square { }
3
4 public static class Drawer
5 {
```

Rysunek 2: Diagram UML, gdzie `Client` **nie** narusza zasady OCP.

```
  public static void DrawShapes(IEnumerable<object> shapes)
  {
    foreach (object shape in shapes)
    {
      if (shape is Circle)
      {
        (shape as Circle).DrawCircle();
      }
      else if (shape is Square)
      {
        (shape as Square).DrawSquare();
      }
    }
  }
}
```

Listing 4: Naruszenie zasady OCP

Stosując polimorfizm i abstrakcję możemy uniezależnić klasę `Drawer` od konkretnych typów. Dzięki utworzeniu wspólnego interfejsu i przekazywanie listy obiektów implementujących ten interfejs, dodanie nowej figury nie będzie wymagało zmian w kodzie klienta. Nie musimy dostosowywać już istniejące kodu do zmian.

```
public interface IShape
{
  void Draw();
}
public class Circle : IShape { public void Draw() { }}
public class Square : IShape { public void Draw() { } }

public static class Drawer
{
  public static void DrawShapes(IEnumerable<IShape> shapes)
  {
    foreach (IShape shape in shapes)
    {
      shape.Draw();
    }
  }
}
```

Listing 5: Poprawne zastosowanie zasady OCP

Oczywiście zmiany będą również konieczne w module, który tworzy instancje obiektów typu `Shape`. Ale ten proces jest zazwyczaj hermetyzowany w metodzie `Main` albo w fabrykach2.1.

1.2.1 Zadanie 2

Utwórz nowy projekt i dodaj do niego klasę o nazwie `UI` z metodą `Drawer` analogicznie jak zostało to pokazane powyżej5. Utwórz w osobnych plikach klasy: `Circle`, `Square`, `Rectangle` oraz `Triangle`. Wszystkie powinny implementować interfejs `IShape` posiadający metodę `Draw()`. Zaimplementuj w klasach kształtów

ten interfejs przez proste wypisywanie na ekranie konsoli informacji, że funkcja `Draw` została wywołana na przykład w następujący sposób:

```
public class Square : IShape
{
  public void Draw() => Console.WriteLine("Square has been drawn.");
}
```

W metodzie `Main` utwórz kilka instancji klas kształtów i dodaj je do listy albo innego kontenera mogącego przechowywać obiekty implementujące interfejs `IShape`. Następnie przekaż do metody `DrawShapes` tę listę (wcześniej utwórz obiekt klasy `Draw`, aby móc z niej skorzystać).

1.3 LSP - Zasada podstawienia Liskov (ang. SOLID)

Zasada podstawienia Liskov pozawala odpowiedzieć na pytanie jakie są dobre praktyki tworzenia hierarchii klas i co zrobić, aby były one zgodne z zasadą otwarte-zamknięte. Klient powinien móc bez zastanowienia się używać wymiennie klasy bazowej oraz klas pochodnych. W sytuacji braku możliwości podstawienia obiektów pochodnych w miejsce obiektów klasy bazowej występuje naruszenie zasady LSP. Często w konsekwencji zostaje również naruszona zasada OCP.

Zasada podstawienia Liskov

Musi istnieć możliwość zastępowania typów bazowych ich podtypami.

Lepsze zrozumienie zasady LSP można uzyskać analizując popularny przykład klasy prostokąta oraz klasy kwadratu, która jest pochodną klasy prostokąta. Zasadność stosowania dziedziczenia często jest analizowania przez zadanie sobie pytania czy klasy są w relacji **IS-A**. Niewątpliwie kwadrat jest prostokątem. Jednak pierwszy problem pojawia się w sytuacji, gdy chcemy zaimplementować/skorzystać z właściwości `Height` i `Width` oraz dalej metody `Area`, zwracającej pole powierzchni danej figury. Właściwości `Height` i `Width` są całkowicie uzasadnione w przypadku prostokąta, jednak wątpliwe w klasie kwadratu.

Próbując rozwiązać ten problem można próbować napisać klasę `Square` tak, że w momencie ustawiania właściwości wysokości albo szerokość w klasie kwadratu, będzie ustawiana zarówno wysokości jak i szerokość na taką samą wartość. Właściwości te mogą w klasie bazowej być wirtualne. Spójrzmy na poniższy kod, w sytuacji gdy do funkcji zostałby przekazany właśnie taki obiekt kwadratu.

```
void Foo(Rectangle r)
{
  r.Width = 5;
  r.Height = 4;
  if(r.Area() != 20) {throw new ...}
}
```

Listing 6: Naruszenie zasady LSP

Pomimo tego zabiegu klient dalej nie jest w stanie użyć instancji klasy pochodnej zamiennie w klasą bazową. Najlepszym rozwiązaniem byłoby pominięcie dziedziczenia i napisania klasy `Sqaure` tak, aby nie dziedziczyła ona po `Rectangle`.

Przewidzieć takie założenia, można za pomocą programowanie przez kontrakt (tzw. DBC ang. Design By Contract). Polega ona na dodaniu do metod pewnych warunków wejściowych oraz wyjściowych, które muszą być spełnione, aby metoda mogła się wykonać.

Ponowna deklaracja procedury (w klasie potomnej) może zastępować warunki wstępne tylko warunkami równymi lub słabszymi, natomiast warunki wyjściowe może zastępować tylko warunkami równymi lub mocniejszymi.

Warunki wyjściowe właściwości `Rectangle.Width` mogłyby zostać zdefiniowane jako:

```
  assert((width == w ) && (height == old.height));
```

Innym problemem, może być sytuacja, gdy dana metoda klasy bazowej przyjmuje dowolny obiekt, natomiast w klasie pochodnej istnieje nieznany wymóg, że obiekt ten musi być konkretnego typu np. z powodu wykonywania operacji rzutowania. Klasa pochodna będzie zgłaszać niezrozumiały dla klienta wyjątek.

Reguły którymi należy się kierować, aby zachować zgodność z zasadą LSP mogą zostać podzielone na dwie kategorie: warunki kontraktu oraz warunki wariancji. Podczas laboratoriów skupimy się na tych pierwszych. Warunki wariancji natomiast są związane z zmiennością argumentów i typów zwracanych.

1.3.1 Warunki kontraktu

Warunki wstępne klasy bazowej nie mogą zostać zaostrzone w klasie pochodnej. Warunki końcowe nie mogą zostać złagodzone w klasie pochodnej. Inwarianty muszą pozostać takie same w klasie pochodnej jakie były w klasie bazowej.

Sygnatura określona w interfejsie danej metody informuje o tym jakiego typu parametry metoda przyjmuje. Jednak w przypadku, gdy wymagane są dodatkowe ograniczenia np. waga przedmiotu nie może być liczbą ujemną, konieczne jest użycie kontraktu. Warunki początkowe powinny być definiowane tak, aby metoda mogła się wykonać poprawnie. Jednym ze sposobów, aby wymusić stosowanie kontraktu jest rzucenie wyjątkiem, w przypadku złej wartości argumentu wejściowego. Jeśli kontrakt nie zostanie spełniony, klient będzie zmuszony przechwycić i obsłużyć wyjątek, w przeciwnym razie wykonywanie programu się zakończy.

Warto w tym miejscu powiedzieć, że w przypadku kontraktów należy używać wyjątków, a nie asercji. Asercje służą, do znajdowania naszych błędów, natomiast wyjątki, aby poradzić sobie z błędami popełnionymi przez użytkowników czy innych programistów wykorzystujących nasz kod. Inaczej mówiąc, w przypadku sprawdzania warunków w publicznym API należy stosować wyjątki. Jeśli sprawdzamy własne, wewnętrzne warunki można używać asercji.

Asercji można używać bardzo liberalnie. Informują one o tym czego kod oczekuje w danym momencie. Wyjątki dotyczą bardziej tego czego żądamy. Dobrze napisane asercje mogą nam powiedzieć nie tylko co się stało oraz gdzie, ale również dlaczego. Służą za dodatkową dokumentację. Jeśli podczas działania programu wystąpi błąd związany z asercją możemy dołączyć debugger do procesu i sprawdzić stan stosu.

1.3.2 Zadanie 3

Stwórz nowy projekt .NET 5.0.

Utwórz klasę o nazwie `ShippingStrategy` i dodaj do niej metodę `CalculateShippingCost`, która będzie przyjmowała argumenty: `float packageWeigthInKilograms`, `float packageDimensionXInInches`, `float packageDimensionYInInches`, `RegionInfo destination`.

Na górze pliku `ShippingStrategy` dodaj przestrzeń nazw: `System.Globalization`, aby móc skorzystać z `RegionInfo`:

```
using System.Globalization;
//...
```

Wewnątrz metody utwórz zmienną pomocniczą typu `decimal` i przypisz do niej dowolną wartość (nie ma ona w tym momencie znaczenia). Będzie ona przechowywała zwracaną wartość, nadaj jej stosowną opisową nazwę.

Dodaj do metody kontrakt w postaci rzucanego wyjątku typu `ArgumentOutOfRangeException` w przypadku, gdy przekazywane parametry są niewłaściwe np. są liczbami ujemnymi:

```
public decimal CalculateShippingCost(float packageWeightInKilograms, float
    packageDimensionXInInches, float packageDimensionYInInches, RegionInfo
    destination)
{
  //Preconditions
  if (packageWeightInKilograms <= 0)
  {
    throw new ArgumentOutOfRangeException(nameof(packageWeightInKilograms)
    , "Package weight can not be negative");
  }
  //...
}
```

Na końcu metody zwróć utworzoną wcześniej pomocniczą zmienną.

Analogicznie warunki końcowe powinny być sprawdzane na końcu metody, aby zagwarantować że metoda nie zmieniła stanu obiektu albo, że zwracana wartość jest poprawna. Dodaj warunek końcowy, który będzie sprawdzał, czy obliczone koszty wysyłki są dodatnie. W przeciwnym razie zgłoś wyjątek.

```
public decimal CalculateShippingCost(float packageWeightInKilograms, float
    packageDimensionXInInches, float packageDimensionYInInches, RegionInfo
    destination)
{
  //...
  //Postconditions
     if (shippingCost <= decimal.Zero) throw new
   ArgumentOutOfRangeException(nameof(shippingCost), "Shipping cost is out
    of range");

  return shippingCost;
}
```

W metodzie `Main` utwórz obiekt klasy `ShippingStrategy` i wywołaj funkcję z poprawnymi i błędnymi argumentami, sprawdź czy został wygenerowany wyjątek.

Dodatkowo jeśli klient ma możliwość zmiany pewnej właściwości, która musi przyjmować ściśle określone wartości, powinna ona również zostać zabezpieczona w analogiczny sposób. Jeśli właściwość ta byłaby chroniona, albo prywatna, a jaj wartość byłaby ustawiana w konstruktorze, sprawdzenie należałoby dodać w konstruktorze klasy.

W utworzonej przed chwilą klasie dodaj prywatną **zmienna** typu `decimal` o nazwie `flatRate`.

Dodaj właściwość `FlatRate` z metodą dostępu `get` oraz akcesorem `set`.

Wewnątrz akcesora `set` dodaj sprawdzenie ustawianej wartości tak jak dla warunków początkowych i końcowych.

```
public class ShippingStrategy
{
  private decimal flatRate;
  public decimal FlatRate
  {
    get { return flatRate;}
    set
    {
      //Data invariants
      if (value <= decimal.Zero) throw new ArgumentOutOfRangeException(
   nameof(value), "Flat rate must be positive and non zero");
      flatRate = value;
    }
  }
  //...
}
```

Sprawdź działanie powyższego kontraktu z poziomu klienta - metody `Main`.

Zamiast umieszczać kod kontraktów wewnątrz metody, lepiej byłoby utworzyć osobne klasy typu `Weight` czy `Size` i to w nich dodać powyższe kontrakty. Dodaj do projekt klasy pomocnicze `Size` oraz `Weight` w dwóch osobnych plikach:

```
public class Size
{
  public double Height { get; init; }
  public double Width { get; init; }

  public Size(double height, double width)
  {
    this.Height = height;
    this.Width = width;
  }
}
```

```
public class Weight
```

```
{
  public double Value { get; init; }
  public Weight(double value) => this.Value = value;
}
```

W ten sposób w całym kodzie kontrakty dla danych typów będą spójne oraz zmniejszy się liczba duplikowanego kodu. Utwórz nowe klasy `Weight` oraz `Size` i umieść w nich właściwości wraz z zabezpieczeniem przed możliwością ustawienia niepoprawnej wartości. Klasa `Weight` może posiadać właściwość `Value`, natomiast `Size` właściwości `Height`, `Width`. Zamień deklarację i ciało metody `CalculateShippingCost` tak, aby korzystała z tych klas.

1.3.3 Zadanie 4

Zbadaj kontrakty w kontekście zasady LSP:

Oznacz metodę `CalculateShippingCost` jako wirtualną. Utwórz klasę pochodną klasy `ShippingStrategy` i nazwij ją np. `WorldWideShippingStrategy`. Za pomocą słowa kluczowego `override` przesłoń wirtualną metodę klasy bazowej i dodaj w niej sprawdzenie czy wartość argumentu `destination` jest `null`. Warunek ten narusza zasadę LSP mówiącą o tym, że nie wolno zaostrzać warunków początkowych.

```
public class WorldWideShippingStrategy : ShippingStrategy
{
  public override decimal CalculateShippingCost(Weight
   packageWeightInKilograms, Size packageDimensionInInches, RegionInfo
   destination)
  {
    //LSP violation
    if (destination == null) throw new ArgumentNullException(nameof(
   destination), "Destination can not be null or empty");
    //...
  }
}
```

Utwórz teraz w metodzie `Main()` dwa obiekty w sposób pokazany poniżej7.

Zwróć uwagę, że w drugim przypadku został zgłoszony wyjątek. Klient musi w takiej sytuacji rozróżniać różne typu wykorzystywanych obiektów co łamie zasadę LSP oraz wprowadza dodatkowe zależności.

```
class Program
{
  static void Main(string[] args)
  {
    ShippingStrategy shippingStrategyCustom = new ShippingStrategy();
    ShippingStrategy shippingStrategyWorldWide = new
   WorldWideShippingStrategy();

    var retA = shippingStrategyCustom.CalculateShippingCost(new Weight(10)
   ,  new Size(10, 10), null);
    var retB = shippingStrategyWorldWide.CalculateShippingCost(new Weight
   (10), new Size(10, 10), null);
  }
}
```

Listing 7: Wywołanie metod klas ShippingStrategy oraz WolrdWideShippingStrategy

Analogicznie złamany może być warunek końcowy. Załóżmy, że w klasie bazowej `ShippingStrategy` tak jak wcześniej koszt wysyłki zawsze jest niezerowy. Natomiast w klasie pochodnej może on być zerowy jeśli, wysyłka jest do tego samego kraju:

```
public class WorldWideShippingStrategy : ShippingStrategy
{
  public override decimal CalculateShippingCost(Weight
   packageWeightInKilograms, Size packageDimensionInInches, RegionInfo
   destination)
```

```
  {
    //...
    //Postconditions LSP violation
    if (destination == RegionInfo.CurrentRegion)
    {
      shippingCost = decimal.Zero;
    }
  }
}
```

Powoduje to ponowne złamanie zasady, że warunki końcowe nie powinny być rozluźniane w klasie pochodnej. Dokładnie ta sama zasada dotyczy wartości niezmiennych (ang. invariant data). W naszym przykładzie właściwości `FlatRate`. Klasa pochodna nie powinna w żaden sposób zmieniać nałożonych wcześniej ograniczeń. Aby to sprawdzić dodaj do klasy `WorldWideShippingStrategy` ponownie właściwość wraz ze słowem kluczowym `new`:

```
public class WorldWideShippingStrategy : ShippingStrategy
{
  //LSP violation
  public new decimal FlatRate { get; set; }
  //...
}
```

Sprawdź powyższe złamania zasady LSP w metodzie `Main`:

```
class Program
{
  static void Main(string[] args)
  {
    var retC = shippingStrategyWorldWide.CalculateShippingCost(new Weight
   (10), new Size(10, 10), RegionInfo.CurrentRegion);
    (shippingStrategyWorldWide as WorldWideShippingStrategy).FlatRate =
   -20;
  }
}
```

Aby uprościć tworzenie kontraktów można napisać prostą metodę statyczną, która ułatwi ten proces:

```
public class CustomContract
{
  public static void Requires<TException>( bool Predicate, string Message
   )
  where TException : Exception, new()
  {
    if ( !Predicate )
    {
      Debug.WriteLine( Message );
      throw new TException();
    }
  }
}
```

`Requires` jest metodą generyczną, gdzie `T` musi być typu referencyjnego oraz posiadać bezparametrowy konstruktor.

1.4 ISP - Zasada segregacji interfejsów (ang. SOLID)

Interfejsy klas nie powinny być rozbudowane/złożone. Jeżeli jedna klasa korzysta jedynie z pewnych funkcjonalności jakie zakłada interfejs, a druga z innych to taki interfejs powinien zostać podzielony na dwa niezależne od siebie. Do celów zapewnienia zgodności z zasadą separacji interfejsów można skorzystać z techniki dziedziczenia wielokrotnego. Należy pamiętać, że w C# klasa nie może dziedziczyć po więcej

niż jeden klasie, natomiast może implementować wiele interfejsów. Obiekty klienckie mogą korzystać z tego samego obiektu za pośrednictwem różnych interfejsów. Powinny one zależeć wyłącznie od wywoływanych przez siebie metod.

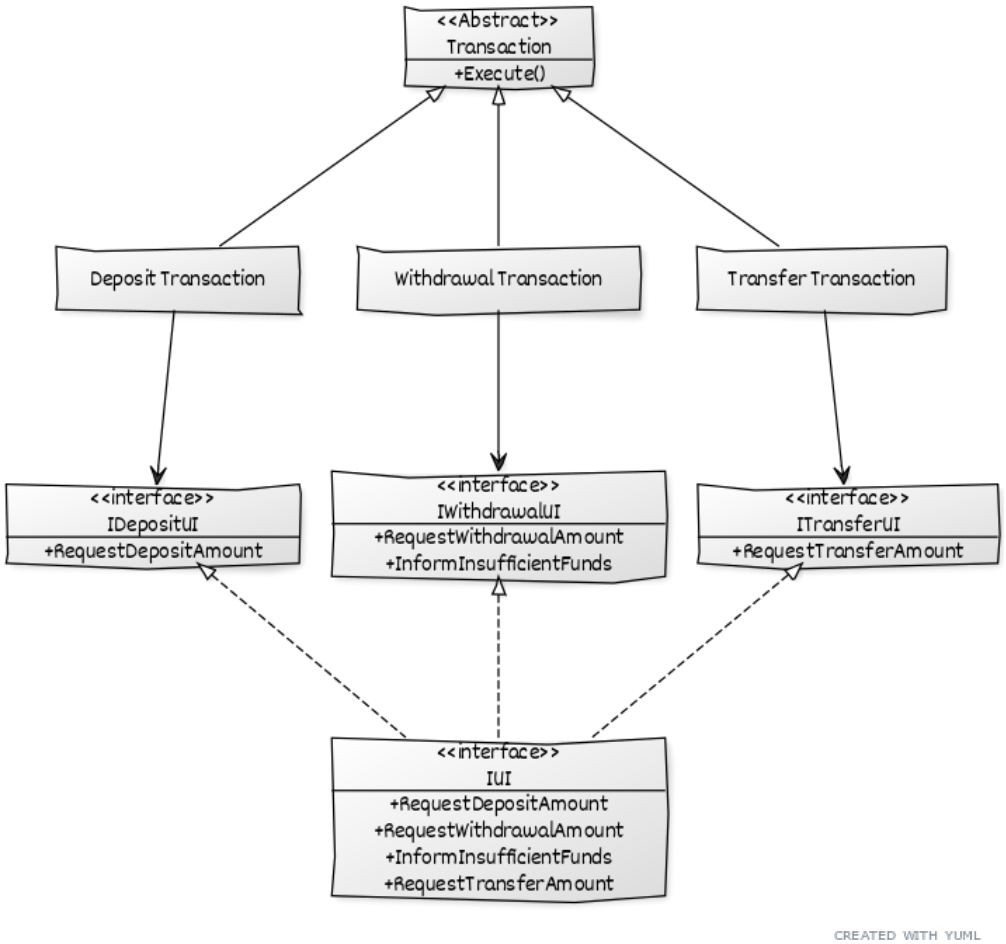
Zasada segregacji interfejsów

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

Wyobraźmy sobie, że musimy przygotować program obsługujący interfejs użytkownika bankomatu. Konieczne jest zapewnienie możliwości obsługi urządzenia z wykorzystaniem interfejsu graficznego, syntezatora mowy oraz za pomocą języka Braille'a (klasy interfejsu użytkownika).

Zakładamy, że każda oęperacja dowolnej transakcji jest realizowana przez osobną klas na przykład mogą to być typy: `DepositTransaction`, `WithdrowalTransaction` oraz `TransferTransaction`, wszystkie mogą dziedziczyć po abstrakcyjnej klasie bazowej `Transaction` z pojedynczą metodą `Execute`. Jednocześnie wszystkie muszą korzystać z klas, które realizują funkcję UI (klasy te powinny implementować interfejs `IUI`). Klasa realizująca obsługę z wykorzystaniem syntezatora mowy, musi implementować wszystkie możliwe operacje przewidziane przez klasy pochodne względem `Transaction`, co jest sensowne. Problemem jest jednak fakt, że np. klasa `DepositTransaction` musi niepotrzebnie znać/posiadać referencje do całego obiektu implementującego UI. Wystarczyłaby by funkcja `RequestDepositAmount`, obiekt nie jest zainteresowany np. `RequestTransferAmount`.

Rozwiązanie tego problemu zostało pokazano na diagramie UML 3. Dzieląc gruby interfejs `IUI` można odciążyć klientów od posiadania referencji do obiektów, których funkcjonalności nie potrzebują. Co więcej, jeśli zostanie dodana nowa transakcja np. `PayGasBillTransaction` nie będzie konieczności ponownej przebudowy pozostałych klas transakcji w wyniku zmiany grubego interfejsu UI.



Rysunek 3: Diagram UML projektu w którym zrealizowano zasadę segregacji interfejsów.

W sytuacji, gdy pewna klasa kliencka albo metoda, potrzebowałaby wykorzystać obiekt implementujący zarówno `IDepositUI` oraz `IWithdrowalUI`, można jako argumenty konstruktora albo metody przekazać dwa

razy ten sam obiekt w następujący sposób: `Foo(ui,ui)`. Deklaracja konstruktora miałaby postać: `void Foo(IDepositUI depositUI, IWithdrowalUI withdrowalUi)`.

Ciekawym przykładem zastosowania zasady ISP jest wykorzystanie interfejsów do umożliwienia wykonywania operacji jedynie po wcześniejszym uwierzytelnieniu użytkownika. Przed operacją logowania klient korzysta z interfejsu `IUnauthorized`:

```
public interface IUnauthorized
{
  IAuthorized Login(string username, string password);
  void RequestPasswordReminder(string emailAddress);
}
```

natomiast po zalogowaniu, poprawnym uwierzytelnieniu, zwracany jest obiekt implementujący `IAuthorized`:

```
public interface IAuthorized
{
  void ChangePassword(string oldPassword, string newPassword);
  void AddToBasket(Guid itemID);
  void Checkout();
  void Logout();
}
```

Drugi z wymienionych interfejsów umożliwia wykonanie większej liczby operacji. Rozwiązanie to zabezpiecza programistę przez udostępnieniem uprzywilejowanych operacji przez niezalogowanego użytkownika. Jest to lepsze rozwiązanie niż umieszczać w interfejsie wszystkie operacje jakie może użytkownik wykonać.

1.4.1 Zadanie 5

Poniższy interfejs8 zawiera zbiór zapytań i poleceń do pamięci trwałej. Dalej natomiast została pokazana przykładowa implementacja9 tego interfejsu korzystająca z bazy danych MongoDB[1] do zapytań oraz NHibernate[2] do poleceń.

```
public interface IPersistence
{
  IEnumerable<Entity> GetAll();
  Entity GetByID(Guid identity);
  IEnumerable<Entity> FindByCriteria(string criteria);
  void Save(Entity entity);
  void Delete(Entity entity);
}
```

Listing 8: Interfejs zawierający zbiór operacji CRUD

```
public class Persistence : IPersistence
{
  private readonly ISession session;
  private readonly MongoDatabase mongo;
  public Persistence(ISession session, MongoDatabase mongo)
  {
    this.session = session;
    this.mongo = mongo;
  }
  public IEnumerable<Entity> GetAll()
  {
    return mongo.GetCollection<Entity>("entities").FindAll();
  }
  public Entity GetByID(Guid identity)
  {
    return mongo.GetCollection<Entity>
```

[1] Baza danych typu NoSQL w której dane przechowywane są jako pliki z formacie JSON.
[2] Biblioteka ORM przeznaczone na platformę .NET do mapowania obiektów modelu domeny na relacyjną bazę danych.

```
    ("entities").FindOneById(identity.ToBson());
  }
  public IEnumerable<Entity> FindByCriteria(string criteria)
  {
    var query = BsonSerializer.Deserialize<QueryDocument>
    (criteria);
    return mongo.GetCollection<Entity>("entities").Find(query);
  }
  public void Save(Entity entity)
  {
    using(var transaction = session.BeginTransaction())
    {
      session.Save(entity);
      transaction.Commit();
    }
  }
  public void Delete(Entity entity)
  {
    using(var transaction = session.BeginTransaction())
    {
      session.Delete(entity);
      transaction.Commit();
    }
  }
}
```

Listing 9: Implementacja interfejsu `IPersistence`

Często można natomiast się spotkać z rozdzieleniem operacji odczytu i aktualizacji bazy danych. Przykładem tej koncepcji jest m.in wzorzec CQRS, o którym szerzej będziemy mówić przy okazji czynnościowych wzorców projektowych 4.3.1. Na ten moment najprościej ujmując polega on na rozdzieleniu operacji odczytu od operacji aktualizacji, dodawania i usuwania z/do bazy danych.

Zastanów się jak można podzielić interfejs8 na dwa mniejsze: jeden dla poleceń, drugi dla zapytań. Dwie klasy np. `CommandsNHibernate` oraz `QueriesMongo` mogłyby implementować odpowiednio `ICommands` i `IQueries`. Klasa kontrolera w takiej sytuacji mogłaby posiadać referencję do dwóch obiektów implementujących wspomniane interfejsy. Takie podejście pozwala na niezależne korzystanie z dwóch rodzajów baz danych, ich swobodną wymianę i skalowanie. Implementacje mogłyby znajdować się różnych w pakietach co jest dodatkową korzyścią

Utwórz nowy projekt .NET 5.0 i zaproponuj rozdzielenie powyższego interfejsu8 w sposób wyżej opisany. Aby wyeliminować błędy kompilacji, konieczne będzie dodanie dwóch pakietów NuGet. W tym celu PPM kliknij na nazwę projektu i wybierz opcję Manage NuGet Packages. Następnie znajdź i zainstaluj pakiet `mongocsharpdriver` oraz `NHibernate`. Dodatkowo utwórz pustą klasę `Enitity`:

```
public class Entity {}
```

1.5 DIP - Zasada odwracania zależności (ang. SOLID)

Głównie za sposób działania aplikacji odpowiadają moduły wysokopoziomowe. To one zawierają logikę biznesową, która często jest wielokrotnie wykorzystywana. Przemyślana architektura obiektowa powinna składać się z wyraźnie zdefiniowanych i odznaczających się warstw. Usługi powinny być opisane odpowiednimi, niezmiennymi interfejsami[3].

Zasada odwracania zależności

A. Moduły wysokopoziomowe nie powinny zależeć od modułów niskopoziomowych. Obie grupy modułów powinny zależeć od abstrakcji.
B. Abstrakcje nie powinny zależeć od szczegółowych rozwiązań. To szczegółowe rozwiązania powinny zależeć od abstrakcji.

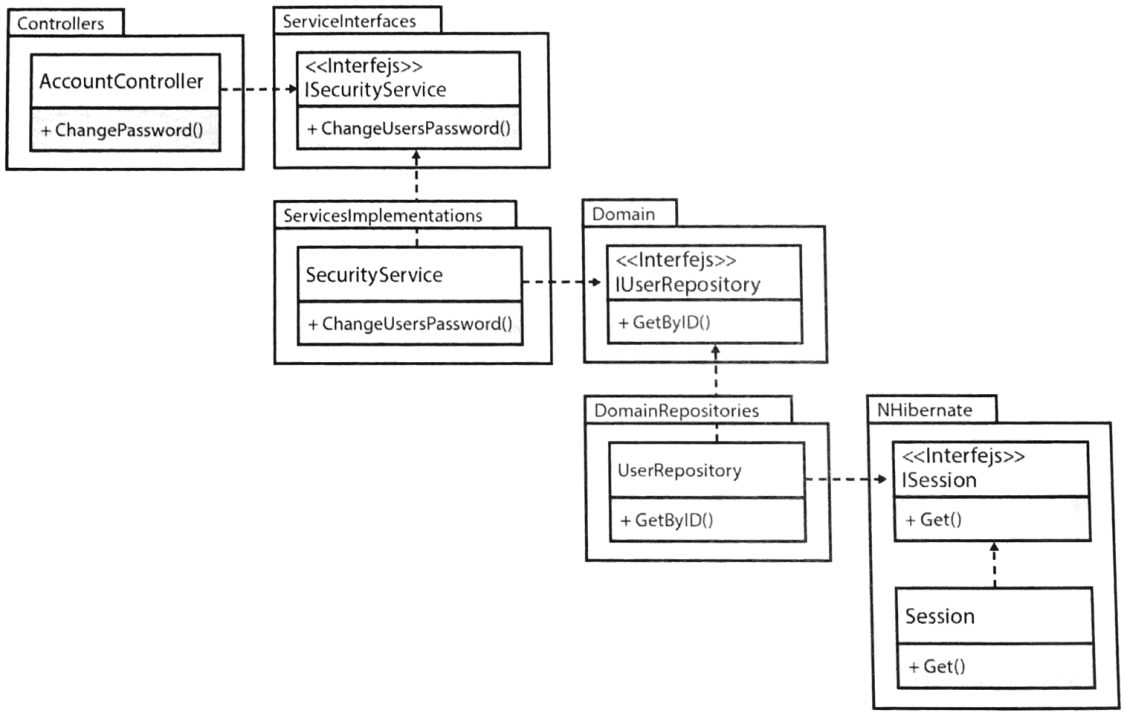
[3]Rzadko udaje się osiągnąć sytuację, w której interfejsy są niezmienne w szczególności na początkowym etapie projektu. Jednak powinny one być zdecydowanie bardziej stabilne niż klasy.

Nie należy postrzegać bibliotek jako właścicieli interfejsów. Interfejsy powinny być powiązane ze swoimi właścicielami i to klasy z innej warstwy powinny implementować te interfejsy. Pozwala to tworzyć oprogramowanie, które jest elastyczne. Jeśli dany zestaw jest wykorzystywany przez kilku klientów np. pobrany pakiet NuGet to warto stworzyć dodatkową warstwę abstrakcji, która umożliwi w przyszłości zmianę tego pakietu, bez konieczności wprowadzania zmian po stronie klienta. Można to osiągnąć np. stosując wzorzec projektowy Adapter 3.1, który zostanie omówiony na przyszłych zajęciach. Oczywiście jeśli mamy do czynienia z klasami, które są zmieniane bardzo rzadko, albo mamy wysoki poziom zaufania to autorów to uzależnienie się od nich nie jest czymś złym. Ciężko znaleźć sens, tworzenia dodatkowej warstwy pośredniej do komunikowania się np. z klasą `string`.

Jako przykład zastosowania zasady DIP wyobraźmy sobie dwie klasy `Button` oraz `Lamp`. Klasa przycisku ma możliwość sterowania lampą. W najprostszej, naiwnej implementacji można by dodać pole w klasie `Button` przechowujące referencje do obiektu lampy. `Lamp` mógłby być przekazywany np. jako argument konstruktora. Dlaczego te podejście jest złe i łamie zasadę DIP? `Button` będący wysokopoziomową abstrakcją, jest uzależniony od niskopoziomowego obiektu lampy.

Powyższy problem można rozwiązać przez dodanie interfejsu np. `ISwitchableDevice` i wstrzykiwanie obiektu go implementującego do obiektu `Button`. Teraz wystarczy, aby `Lamp` implementował ten interfejs. W przypadku, kiedy przycisk będzie miał zostać wykorzystany do sterowania np. silnikiem albo piecem wystarczy utworzyć nowy obiekt `Motor` albo `Heater`, który będzie implementował `ISwitchableDevice` i przekazać go do obiektu `Button`.

Na poniższym diagramie UML4 pokazano hierarchię klas zgodną z zasadą odwracania zależności. W tym przypadku interfejsy i implementacje zostały umieszczone w osobnych zestawach. Klienci posiadają referencje jedynie do zestawów z interfejsami. W idealnym scenariuszu interfejsy nie powinny posiadać żadnych zależności, kod kliencki również. To implementacje powinny zależeć od interfejsów. Klasy zestawu `Controllers` nie są związane z klasami zestawu `ServicesImplementations`, a jedynie z abstrakcją w postaci interfejsów zestawu `ServiceInterfaces`. Analogicznie sytuacja wygląda w przypadku zależności zestawu `ServicesImplementations` od `Domain`. Warto zauważyć, że od szczegółu jakim jest wykorzystywany ORM (ang. Object-Relational Mapping)[4] nie zależy żaden moduł wyższego poziomu. Powoduje to, że zmiana tego komponentu nie pociągnie za sobą zmian w kodzie warstw wyższych i nie będzie wymagała ich ponownej kompilacji.



Rysunek 4: Diagram UML hierarchii klas zgodnej z regułą DIP[1]

[4]Technika, która pozwala na zapytania i polecenia do bazy danych z wykorzystaniem paradygmatów programowania obiektowego. Zazwyczaj pisząc ORM mamy na myśli bibliotekę, która implementuje technikę ORM. Pozwala nam na uniezależnienie się od konkretnej bazy danych, komunikacją z nią odbywa się za pomocą konkretnej biblioteki ORM np. NHiberante czy EntityFramework.

1.5.1 Zadanie 6

Przeanalizuj poniższy kod i zmień go tak, aby był zgodny z omawianą zasadą DIP.

```
public class Regulator
{
  const byte THERMOMETER = 0X86;
  const byte FURNACE = 0X87;
  const byte ENGAGE = 1;
  const byte DISENGAGE = 0;

  void Regulate(double minTemp, double maxTemp)
  {
    for (; ; )
    {
      while (this.Read(THERMOMETER) > minTemp)
      {
        Thread.Sleep(1000);
      }
      this.Write(FURNACE, ENGAGE);
      while (this.Read(THERMOMETER) < maxTemp)
      {
        Thread.Sleep(1000);
      }
      this.Write(FURNACE, DISENGAGE);
    }
  }

  byte Read(byte register)
  {
    return 0x00; //returned value not relevant.
  }
  void Write(byte register, byte value)
  {
    //...
  }
}
```

Możesz utworzyć dwa dodatkowe projekty jeden w którym dodasz interfejsy, jeden dla obiektów mierzących temperaturę `ISensor` i jeden dla obiektów odpowiedzialnych za zmianę temperatury np. pieców `IDevice`. Natomiast drugi projekt niech zawiera implementacje utworzonych przed chwilą interfejsów np. `Thermometer` oraz `Heater`. Klasa `Regulator` powinna następnie zostać uzależniona od projektu z interfejsami. Obiekty implementujące `ISensor` oraz `IDevice` powinny zostać wstrzyknięte do klasy przez konstruktor. W metodzie `main` utwórz instancję klasy `Regulator` i sprawdź jej działanie wywołując metodę `Regulate()`.

1.6 Kontenery IoC (ang. Inversion of Control)

Kontenery IoC takie jak Castle Windsor, Autofac czy Ninject ułatwiają wstrzykiwanie zależności. Jeśli pracujemy nad dużym projektem, w którym klasy zależą od wielu innych i wykorzystują mechanizm wstrzykiwania zależności można wykorzystać kontener IoC, aby ułatwić sobie proces tworzenia obiektów.

Na przykład tworzenie obiektu `CustomerService` pokazanego poniżej10 wymaga utworzeniu wielu innych obiektów i przekazania ich jako argumenty konstruktora.

```
public CustomerData GetCustomerData(string customerNumber)
{
  var customerApiEndpoint = ConfigurationManager.AppSettings["customerApi:
   customerApiEndpoint"];
  var logFilePath = ConfigurationManager.AppSettings["logwriter:
   logFilePath"];
  var authConnectionString = ConfigurationManager.ConnectionStrings["
   authorization"].ConnectionString;
```

```
  using(var logWriter = new LogWriter(logFilePath ))
  {
    using(var customerApiClient = new CustomerApiClient(
   customerApiEndpoint))
    {
      var customerService = new CustomerService(
      new SqlAuthorizationRepository(authorizationConnectionString,
   logWriter),
      new CustomerDataRepository(customerApiClient, logWriter),
      logWriter
      );

      return customerService.GetCustomerData(string customerNumber);
    }
  }
}
```

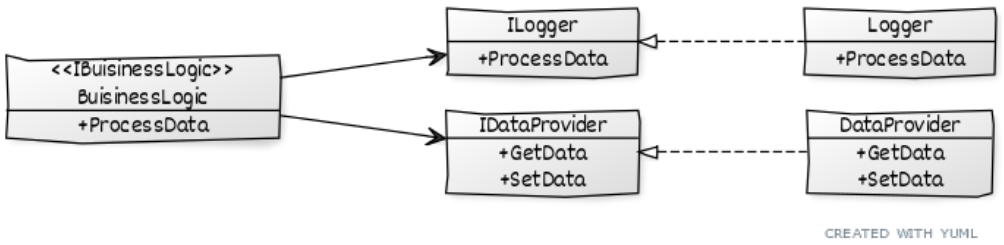
Listing 10: Tworzenie obiektu posiadającego zależności wstrzykiwane przez konstruktor

Wykorzystując kontener IoC można zarejestrować zależności, które będą tworzone i wstrzykiwane automatycznie. Jeśli dana klasa potrzebuje np. obiektu implementującego `ILogWriter`, kontener utworzy odpowiednią instancję pod warunkiem, że została ona wcześniej zarejestrowana. Jeżeli np. tym obiektem jest `LogWriter` i wymaga on ścieżki do pliku, można również taką ścieżkę zarejestrować jako wartość klucza w `AppSettings`. Kontener „wie" jakie obiekty, jakich typów muszą być utworzone i wstrzyknięte, aby spełnić wymagania tworzonej klasy. Jeśli odpowiedni obiekt nie został wcześniej zarejestrowany to zostanie zgłoszony wyjątek o tym informujący. Dodatkowo kontenery umożliwiają sprecyzowanie czy jedna i ta sama instancja ma zostać przekazana do wszystkich obiektów, które jej potrzebuje czy np. za każdym razem powinien być tworzony nowy obiekt.

Kontenery IoC powinny być wykorzystywane jedynie do wiązania poszczególnych elementów programu. Jeśli chcemy tworzyć obiekty podczas działania programu należy wykorzystywać fabryki2.1. Trzeba unikać sytuacji w których w całym programie znajdują się referencje do kontenera. Zazwyczaj wystarcza jedno pobranie obiektu np. typu `Application`, natomiast reszta powiązań powinna odbywać się po stronie kontenera IoC.

1.6.1 Zadanie 7

Utwórz dwa projekty w solucji. Jeden niech będzie biblioteką, drugi natomiast aplikacją konsolową. Dodaj referencję do biblioteki w aplikacji konsolowej. Aby to zrobić kliknij PPM na projekt aplikacji konsolowej, dalej `Add` i `Project Reference...`, następnie wskaż odpowiedni zestaw. W projekcie biblioteki utwórz następującą hierarchię klas pokazaną na rysunku5.



Rysunek 5: Diagram UML hierarchii klas biblioteki z zadania 1.6.1

Zaimplementuj metody tak, aby wypisywały informację na konsoli o tym jaka metoda jest aktualnie wykonywana np.:

```
public class DataProvider : IDataProvider
{
  public void GetData() => Console.WriteLine("Data downloading...");
  public void SetData() => Console.WriteLine("Data saving...");
}
```

Dodaj za pomocą menadżera pakietów NuGet pakiet `Autofac` do projektu aplikacji konsolowej. Następnie utwórz w osobnym pliku klasę w której będzie następowała konfiguracja kontenera:

```
public static class ContainerConfig
{
  public static IContainer Configure()
  {
    var builder = new ContainerBuilder();

    //...

    return builder.Build();
  }
}
```

Listing 11: Konfiguracja kontenera IoC

Aby móc „pobrać" instancję danej klasy z kontenera w pierwszej kolejności należy skojarzyć dane klasy z odpowiadającymi im interfejsami. Inaczej mówiąc konieczne jest przekazanie informacji jaką klasę chcemy wykorzystać w miejsce danej abstrakcji. Należy to zrobić w następujący sposób:

```
builder.RegisterType<BusinessLogic>().As<IBusinessLogic>();
//...
```

Powyższy fragment kodu oraz pozostałe rejestracje umieść w metodzie `Configure` klasy `ContainerConfig` 11.

Do projektu aplikacji konsolowej dodaj również interfejs `IApplication`, z jedną metodą `Run()` nie zwracającą żadnego obiektu. W pliku `Application` umieść implementację interfejsu `IApplication`:

```
public class Application : IApplication
{
  private readonly IBusinessLogic businessLogic;
  public Application(IBusinessLogic businessLogic) => this.businessLogic =
    businessLogic;
  public void Run() => businessLogic.ProcessData();
}
```

W metodzie `Main` umieść poniższy kod:

```
static void Main(string[] args)
{
  //First thing we want to in program do is wireup the container and
   dependencies
  var container = ContainerConfig.Configure();

  using(var scope = container.BeginLifetimeScope()) //new scope
  {
    var app = scope.Resolve<IApplication>(); //we need IApplication object
   , hence container gives us one
    app.Run();
  }

  Console.ReadLine();
}
```

Uruchom program i sprawdź, czy kontener poprawnie powiązał i stworzył wszystkie obiekty.

W momencie wywołania `Resolve` kontener sprawdza jakiego typu obiekty należy przekazać do konstruktora obiektu i wstrzykuje je za nas. W naszym przypadku po wywołaniu metody `Resolve<IApplication>` kontener „widzi", że wymagany jest obiekt implementujący `IBusinessLogic`, aby utworzyć `Application`. Dalej `BusinessLogic` wymaga przekazania obiektu implementującego `ILogger` oraz `IDataProvider`. I w tym przypadku zostaną za nas utworzone odpowiednie obiekty i przekazane do `BusinessLogic` tak, aby mógł on zostać stworzony na potrzeby `Application`.

Aby ułatwić proces wiązania danych obiektów, można zamiast rejestrować poszczególne typy, zarejestrować cały zestaw wraz z określeniem pewnych warunków. Można na przykład zarejestrować wszystkie obiekty z zestawu, których nazwy zaczynają się danym prefiksem, albo znajdują się w danym folderze. Można również powiązać wszystkie klasy z zestawu z odpowiadającymi im interfejsami.

```
builder.RegisterAssemblyTypes(Assembly.Load(nameof(
    DependencyInversionPrincipleLib)))
//.Where(t => t.Namespace.Contains("Loggers")) //only from specific folder
.AsImplementedInterfaces(); //register the type as providing all of its
    public interfaces as services (excluding IDisposable).
```

2 Wzorce projektowe (1) - konstrukcyjne

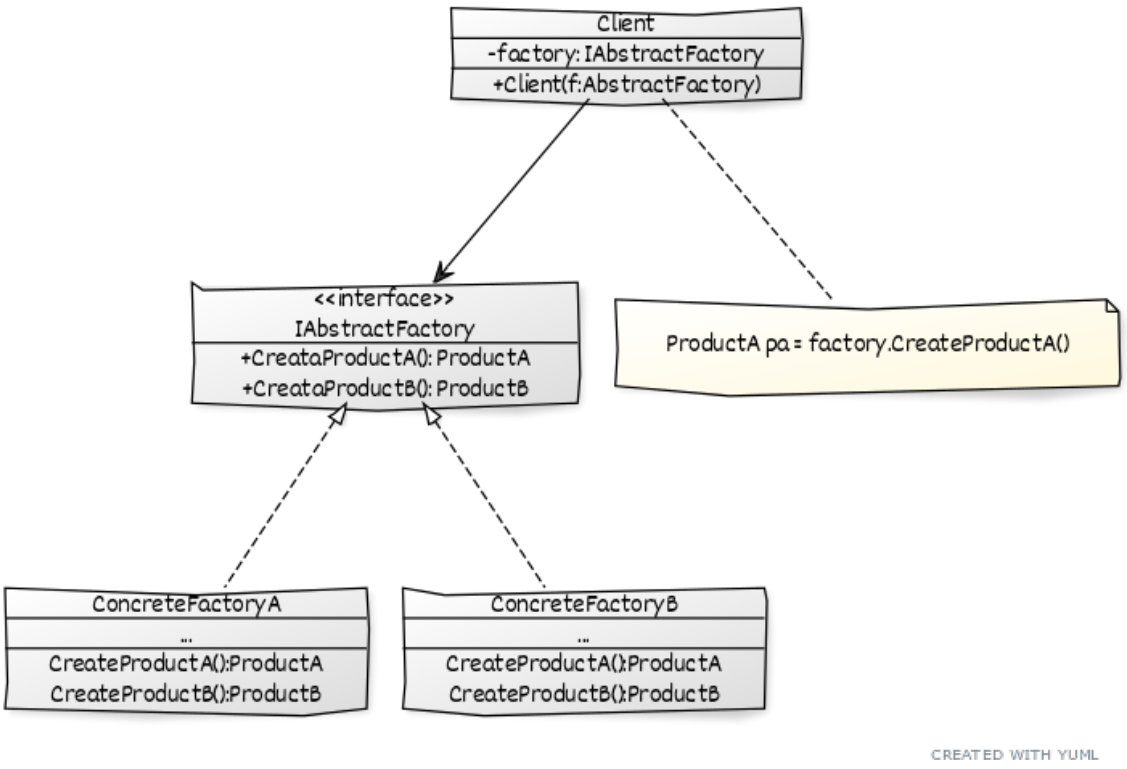
Wzorce kreacyjne pozwalają na tworzenie obiektów w sposób zwiększający elastyczność. Zamiast tworzyć obiekty w wielu miejscach w programie, zadaniem konstrukcji nowych instancji można obciążyć osobne klasy. Użycie słowa kluczowego `new` narusza omawianą wcześniej zasadę odwracania zależności, powodując trudności z testowaniem kodu i zmniejszając jego elastyczność.

2.1 Fabryka abstrakcyjna (ang. abstract factory)

Fabryka abstrakcyjna jest wzorcem projektowym, który przenosi proces tworzenia obiektów (często ze sobą w jakiś sposób powiązanych) do osobnej klasy implementującej interfejs zawierający metody zwracające te obiekty. Tworzone przez fabryki instancje powinny posiadać zgodne interfejsy.

Klient fabryki może posiadać konstruktor, który jako parametr przyjmie obiekt konkretnej fabryki. Wewnątrz konstruktora klient może przypisać przekazaną instancję do prywatnego pola. Zakładamy, że zarówno prywatne pole wewnątrz klasy klienta jak i typ przekazywanego przez konstruktor argumentu jest typem interfejsu, który implementuje każda konkretna fabryka. Wywołując odpowiednie metody obiektu fabryki, klient może tworzyć konkretne instancje zwracane przez fabrykę bez znajomości ich szczegółowej implementacji. Przekazując inny obiekt fabryki abstrakcyjnej możliwa jest łatwa zmiana pośrednio tworzonych przez klienta obiektów.

Jako przykład użycia omawianego wzorca można wyobrazić sobie aplikację wykorzystującą interfejs graficzny. Klasa tworząca kontrolki interfejsu użytkownika może przyjmować obiekt fabryki implementującej interfejs `IGuiFactory`. Interfejs ten mógłby definiować metody tworzenia poszczególnych elementów graficznych np. `CreateButton()`, `CreateListBox()`, `CreateProgressBar()` itp. Następnie w zależności od systemu operacyjnego można by stworzyć obiekty fabryk `WindowsFactory` oraz `UnixFactory`, oba implementujące określony wcześniej interfejs. Obiekt zawierający implementację logiki rysującej okno użytkownika nie musiałby wiedzieć z jakim systemem operacyjnym ma do czynienia, nie posiadałby zależności do konkretnej implementacji. Mógłby wywoływać metody tworzące kontrolki UI za pomocą abstrakcyjnego interfejsu `IGuiFactory`. Wybór konkretnej fabryki mógłby być określany w momencie uruchamiania aplikacji.



Rysunek 6: Diagram UML wzorca Fabryka Abstrakcyjna.

2.1.1 Zadanie 1

Utwórz nowy projekt aplikacji konsolowej .NET 5.0. W osobnych plikach dodaj do projektu trzy interfejsy: `IButton`, `ICheckbox` oraz `IGuiFactory`. Pierwsze dwa niech posiadają jedną metodę `void Draw()`, natomiast interfejs fabryki `IGuiFactory` dwie metody:`IButton CreateButton()` oraz `ICheckbox CreateCheckBox()` Typem zwracanym przez te dwie metody są obiekty implementujące odpowiednio `IButton` oraz `ICheckbox`.

Umieść w projekcie dwa foldery `Macintosh` oraz `Windows`. Wewnątrz tych folderów dodaj implementacje stworzonych przed chwilą interfejsów. Wszystkie metody `Draw` zaimplementuj tak, aby wypisywały na ekranie konsoli komunikat symulujący, że kontrolki zostały narysowane np.:

```
internal class MacButton : IButton
{
  public void Draw() => Console.WriteLine("Macintosh button has been drawn
    ");
}
```

Po napisaniu analogicznych metod dla pozostałych kontrolek, utwórz po jednej klasie fabryki w każdym z folderów. Każda klasa fabryki powinna zwracać obiekty konkretnych kontrolek UI np.:

```
public class MacFactory : IGuiFactory
{
  public IButton CreateButton() => new MacButton();
  public ICheckbox CreateCheckBox() => new MacCheckbox();
}
```

Teraz w katalogu głównym projektu utwórz klasę `UserInterface`, która będzie posiadała prywatne pole typu `IGuiFactory`. Dodaj do tej klasy konstruktor z jednym parametrem typu `IGuiFactory`. Przypisz przekazywany do wnętrza klasy obiekt do utworzonego przed chwilą prywatnego pola. Napisz metodę `DrawWindow`, w której pobierzesz z fabryki instancje klas implementujących `IButton` oraz `ICheckbox` i wywołasz metody `Draw` każdego z nich.

W metodzie `Main` sprawdź działanie wzorca fabryki abstrakcyjnej:

```
class Program
{
  static void Main(string[] args)
  {
    // Based on configuration file one can choose the specific factory
   over another one
    var factoryA = new MacFactory();
    var factoryB = new WinFactory();

    var uiA = new UserInterface(factoryA);
    uiA.DrawWindow();

    var uiB = new UserInterface(factoryB);
    uiB.DrawWindow();
  }
}
```

2.1.2 Inne wzorce fabryk

Fabryka jest ogólnym terminem, który jest używany w stosunku do metod i klas, które tworzą obiekty. Podczas laboratoriów został omówiony wzorzec fabryki abstrakcyjnej, który służy do tworzenia całych rodzin podobnych albo powiązanych ze sobą obiektów bez precyzowania konkretnych klas. Jeśli jednak program nie wymaga tworzenia takich rodzin najprawdopodobniej fabryka abstrakcyjna nie jest konieczna.

Inną prostą fabryką może być klasa, która posiada metodę wytwórczą, z wyrażeniem `switch-case` które analizuje przekazywany do niej parametr i tworzy odpowiednią instancję konkretnej klasy. Często jest to pojedyncza metoda w pojedynczej klasie. Z czasem kiedy liczba tworzonych obiektów rośnie może ona się przekształcić z metodę fabrykującą.

Metoda fabrykująca udostępnia interfejs do tworzenia obiektów, ale pozwala klasom pochodnym zmienić typ tworzonego obiektu. W klasie bazowej umieszczona jest pewna funkcjonalność, do klas pochodnych jest przeniesiony jedynie proces tworzenia obiektów. Jest to szczególny przypadek metody szablonowej.

2.2 Budowniczy (ang. builder)

Budowniczy jest wzorcem, który może być wykorzystany do tworzenia złożonego obiektu **krok po kroku**. W przeciwieństwie do wzorca fabryki abstrakcyjnej, która tworzy obiekty **w jednym kroku**.

Czasami istnieje pokusa stworzenia konstruktora przyjmującego dużą liczbę argumentów. Może tak się zdarzyć jeśli klasa korzysta z wielu zależności albo gdy niektóre jej cechy mogą być parametryzowane. Część z tych argumentów może dodatkowo być opcjonalna. Budowniczy przenosi proces tworzenia takiej instancji do osobnego obiektu.

Czynności wywoływania poszczególnych metod konfigurujących tworzony obiekt można przenieść do obiektu kierownika (ang. director), który będzie przyjmował jako argument konstruktora obiekt budowniczego i wywoływał metody interfejsu `IBuilder`. Obiekt kierownika jest opcjonalny, klient może samemu tworzyć instancję budowanego typu.

Konkretni budowniczowie powinni dostarczyć swoje własne metody zwracania wyników procesu budowania obiektu. Różni budowniczowie mogą zwracać zupełnie inne typy dlatego tego procesu często nie da się umieścić w interfejsie `IBuilder` (przynajmniej w przypadku języków programowania, które są silnie typowane).

Dodatkowo należy wspomnieć, że wzorzec budowniczy często wykorzystuje tzw. Płynny Interfejs (ang. Fluent Interface). Polega on na łańcuchowym połączeniu metod. Pozwala na zwiększenie czytelności kodu. Kaskadowe połączenie metod realizuje się zwracając w metodzie instancję własnej klasy za pomocą słowa kluczowego `this`. Z przykładem użycia tego mechanizmu można się spotkać w zapytaniach LINQ12, gdzie zamiast wywoływać kolejne metody progresywnie (w kolejnych wierszach), można je wywoływać kaskadowo po symbolu kropki.

```
var translations = new Dictionary<string, string>
{
  {"cat", "chat"},
  {"dog", "chien"},
  {"fish", "poisson"},
  {"bird", "oiseau"}
};

// Find translations for English words containing the letter "a",
// sorted by length and displayed in uppercase
IEnumerable<string> query = translations
.Where(t => t.Key.Contains("a"))
.OrderBy(t => t.Value.Length)
.Select(t => t.Value.ToUpper());

// The same query constructed progressively:
var filtered = translations.Where(t => t.Key.Contains("a"));
var sorted  = filtered.OrderBy(t => t.Value.Length);
var finalQuery = sorted.Select(t => t.Value.ToUpper());
```

Listing 12: Wykorzystanie płynnych interfejsów w zapytania LINQ

2.2.1 Zadanie 2

Utwórz projekt .NET 5.0. Dodaj do niego klasę `Car` i umieść do niej właściwości ze słowem kluczowym `init`[5]13.

```
public string Manual { get; init;}
public string Engine { get; init; }
public string Seats { get; init; }
public string TripComputer { get; init; }
public string Gps { get; init; }
```

Listing 13: Inicjalizacja obiektu klasy `Car` ze niemutowalnymi właściwościami

Zdefiniuj w klasie `Car` prywatny konstruktor (wykorzystaj modyfikator dostępu private):

[5]Słowo kluczowe `init` pojawiło się w C# 9.0 i stanowi ułatwienie w tworzeniu obiektów niemutowalnych (niezmiennych). Wcześniej konieczne było wykorzystywanie słowa kluczowego `readonly` i tworzenie obszernych konstruktorów.

```
private Car() { }
```

Zwróć uwagę, że nie będzie możliwe utworzenie obiektu klasy `Car` za pomocą słowa kluczowego `new` w powodu braku dostępnego konstruktora.

W osobnym pliku utwórz interfejs `ICarBuilder`, który będzie posiadał następujące metody zwracające typ `void`:

- Reset(),
- SetEngine(),
- SetSeats(),
- SetTripComputer(),
- SetGps(),
- SetManual().

Wewnątrz klasy `Car`, zdefiniuj klasę `CarBuilder` implementującą interfejs `ICarBuilder`. Zwróć uwagę, że klasa ta ma dostęp do prywatnych składowych klas (w tym konstruktora). Utworzenie instancji tej klasy będzie możliwe jedynie przez klasę budowniczego. W klasie budowniczego utwórz prywatne pola typu `string`: `seats`, `engine`, `tripComputer` oraz `gps`.

Zaimplementuj interfejs `ICarBuilder`. W każdej metodzie ustawiającej dany fragment samochodu, przypisz jakiś łańcuch znaków pozwalający na późniejszą identyfikację np.:

```
public class CarBuilder : ICarBuilder
{
  private string seats;
  //...

  public void SetSeats() => this.seats = "Four fancy seats";
  //...
}
```

Do klasy `CarBuilder` dodaj metodę `Build` zwracającą typ `Car`. Utwórz w niej instancję klasy `Car`, przypisz do jej właściwości ustawione wcześniej wartości prywatnych pól np.:

```
public Car Build()
{
  var car = new Car()
  {
    Engine = this.engine,
    //...
  };

  //...
}
```

W metodzie `Main` korzystając z budowniczego utwórz instancję klasy `Car`. Aby to zrobić stwórz obiekt `CarBuilder`, następnie wywołaj metody ustawiające poszczególne właściwości (nie jest konieczne wywoływanie ich wszystkich) i utwórz obiekt za pomocą metody `Build`. Sprawdź czy utworzona instancja typu `Car` zawiera prawidłowe wartości odpowiednich właściwości tej klasy. Możesz to zrobić wypisując je na ekranie konsoli. Ewentualnie można przeciążyć metodę `ToString` tak, aby zwracała łańcuch znaków zawierający te informacje.

Utwórz klasę kierownika `Director`. Niech posiada ona prywatne pole typu `ICarBuilder` oraz właściwość `set`:

```
internal class Director()
{
  private ICarBuilder _builder;
  public ICarBuilder Builder { set { _builder = value; } }
  //...
}
```

Dodatkowo umieść w niej dwie metody: `BuildMinimalViableProduct` oraz `BuildFullFeaturedProduct`. Obie metody niech zwracają typ `void`. W pierwszej wywołaj dwie z dostępnych metod tworzenia obiektu przez budowniczego, w drugim wszystkie.

Z poziomu metody `Main` utwórz obiekt `CarBuilder` oraz `Director`. Przypisz kierownikowi obiekt budowniczego. Następnie wywołaj jedną z dostępnych metod kierownika. Na koniec pobierz „zbudowany" obiekt za pomocą metody `Build` klasy `CarBuilder`. Sprawdź utworzone elementy obiektu `Car`.

2.2.2 Zadanie 3

Wewnątrz klasy `Car` utwórz drugiego budowniczego. Tym razem do implementacji wykorzystaj tzw. Płynne Interfejsy. Jedyną różnicą względem poprzedniej implementacji będzie fakt, że tym razem metody ustawiające wartości prywatnym polom (te metody, które „budują" obiekt), będą zwracały instancję klasy budowniczego. Konieczne jest skorzystanie ze słowa kluczowego `this` zwracającego instancję kasy np.:

```
public CarBuilderFluent SetSeats()
{
  this.seats = "Four fancy seats";
  return this;
}
```

Sprawdź działanie drugiej wersji wzorca projektowego Budowniczy. Tym razem kolejne metody można wywoływać z wykorzystaniem symbolu kropki.

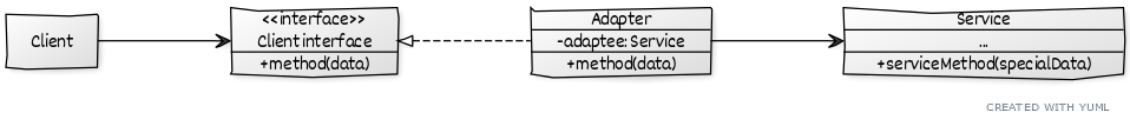
```
// Builder with a fluent interface
var builderFluent = new Car.CarBuilderFluent();
var carFluent = builderFluent.SetEngine()
  .SetGPS()
  .SetSeats()
  .SetTripComputer()
  .Build();
```

3 Wzroce projektowe (2) - strukturalne

Strukturalne wzorce projektowe wyjaśniają jak łączyć obiekty i klasy w większe struktury zachowując jednocześnie prostotę i elastyczność.

3.1 Adapter (ang. adapter)

Wzorzec który pozwala na działanie dwóch komponentów o niezgodnych interfejsach nazywany jest Adapterem. Jeśli klasa potrzebuje do działania obiektu klasy o określonym interfejsie, natomiast klasa która mogłaby zostać wykorzystana posiada inny, niezgodny interfejs, można utworzyć klasę adaptera, która dostosuje niezgodne interfejsy. Stosuje się go do dopasowywania kodu, który już istnieje. Jest często wykorzystywany w celu wykorzystania zewnętrznego zestawu firm trzecich np. pobranego z menadżera pakietów NuGet, którego interfejs jest niezgodny z tym używanym w naszej aplikacji.



Rysunek 7: Diagram UML wzorca Adapter.

Wyobraźmy sobie sytuację, w której mamy działający komponent i do logowania informacji o swoim działaniu wykorzystuje napisany przez nas zestaw np. bibliotekę. Klasa logera implementuje określony interfejs np. `ILogger`. Wykorzystując ten interfejs, klasa kliencka używa obiektu klasy logera do logowania. Jeśli za jakiś czas okaże się konieczne wykorzystanie bardziej rozbudowanego narzędzia do logowania np. pakietu NLog, możemy albo zmieniać odwołania w kodzie klienckim, albo wykorzystać wzorzec Adapter. W drugim przypadku adapterem będzie klasa implementująca nasz interfejs `ILogger` i posiadająca obiekt typu `NLog.Logger` przekazany/wstrzyknięty np. jako argument konstruktora. Wszystkie metody interfejsu `ILogger` są przekazywane do odpowiednich metod pakietu `NLog`. Po napisaniu adaptera wystarczy wstrzyknąć nowo utworzony obiekt do kodu klienckiego bez dodatkowych modyfikacji. Nie będzie również problemem napisanie dodatkowego adaptera dla innego pakietu np. Log4N. Ze względu na fakt, że oba adaptery implementują ten sam interfejs można napisać wiele różnych adapterów dla adaptowanych obiektów, które będą wykorzystywane przez kod kliencki.

Innym przykładem zastosowania wzorca Adapter może być sytuacja w której posiadamy aplikację, która pobiera dane z plików tekstowych w formacie `XML` i następnie je przetwarza oraz wizualizuje te dane w formie wykresów. Z czasem może pojawić się potrzeba, aby dane pobierać z zewnętrznego serwera w innym formacie np. `JSON`, korzystając z gotowego zestawu (np. pakietu NuGet). Jeśli korzystamy z rozwiązań innych firm nie ma możliwość zmiany interfejsu zestawu pobierającego dane z serwera. Implementując wykorzystywany wcześniej interfejs możemy przekierowywać żądania od klienta do zewnętrznego zestawu „w locie", zmieniając format danych z `XML` na `JSON`.

Opisana powyżej wersja adaptera jest wersją obiektową tego wzorca. Można się również spotkać z podejściem klasowym. W drugim podejściu obiekt adaptera zamiast posiadać adaptowany obiekt, może po nim dziedziczyć. Wersja obiektowa jest jednak częściej stosowana zgodnie z zasadą, aby przekładać kompozycję nad dziedziczenie.

3.1.1 Zadanie 1

Celem tego zadania, będzie napisanie adaptera dla pakietu służącego do logowania informacji diagnostycznych w naszej aplikacji. Wykorzystany zostanie popularny pakiet NLog, który dostarcza bogaty zbiór różnych możliwości logowania m.in. komunikaty mogą być wyświetlane na ekranie konsoli, wysyłane mailem, zapisywane do pliku, bazy danych, czy chmury. Zastosowanie wzorca Adapter pozwoli nam na wykorzystanie tego pakietu, bez konieczności uzależnienia się od niego. Jeśli w przyszłości konieczne okaże się użycie np. Log4N zamiana będzie bardzo prosta.

W pierwszej kolejności utwórz projekt aplikacji konsolowej .NET 5.0. Za pomocą menedżera pakietów dodaj pakiet NLog. Konfiguracja pakietu odbywa się z użyciem pliku NLog.config. Utwórz plik o takiej nazwie i rozszerzaniu w folderze projektu. Skopiuj do niego poniższe reguły logowania:

```
<?xml version="1.0" encoding="utf-8" ?>
<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

```
4
5   <targets>
6     <target
7       name="_File"
8       xsi:type="File"
9       fileName="${basedir}/Logs/${shortdate}.log"
10      layout="${longdate}|${uppercase:${level}}: ${message}"/>
11    <target
12      name="_Console"
13      xsi:type="Console"
14      layout="${longdate}|${uppercase:${level}}: ${message}"/>
15  </targets>
16
17  <rules>
18    <logger name="*" minlevel="Info" writeTo="_Console" />
19    <logger name="*" minlevel="Debug" writeTo="_File" />
20  </rules>
21 </nlog>
```

Listing 14: Konfiguracja pakietu NLog

Plik14 zawiera targety oraz reguły. Te pierwsze wskazują, w jaki sposób chcemy zapisać informacje[6], w przypadku naszej konfiguracji, będzie to wypisanie wiadomości na ekranie konsoli oraz zapisanie w pliku tekstowym. Reguły natomiast definiują jaki poziom „ważności" informacji ma zostać zapisany w jakim targecie. Aby plik NLog.config mógł być wykorzystany konieczne jest jego przekopiowanie do folderu z plikiem wykonywalnym. Klikając PPM na nazwę pliku należy wybrać Properties i dalej w opcji Copy to Output Directory wskazać opcję Copy always. Dzięki temu w momencie kompilacji plik konfiguracyjny znajdujący się katalogu głównym projektu zostanie skopiowany do folderu z plikiem wykonywalnym.

Sprawdzić działanie pakietu NLog można pobierając instancję logera metodą `GetCurrentClassLogger()` w pokazany na listingu 15 sposób. Po uruchomieniu programu w folderze bin/Debug/Logs powinien pojawić się plik tekstowy *.log. Dodatkowo komunikaty powinny zostać wyświetlone na ekranie konsoli.

```
1  using NLog;
2
3  namespace AdapterDesignPattern
4  {
5    class Program
6    {
7      private static readonly Logger Logger = LogManager.
       GetCurrentClassLogger();
8
9      static void Main()
10     {
11       Logger.Info("Hello world");
12       Logger.Error("Goodbye cruel world!");
13       Console.ReadLine();
14     }
15   }
16 }
```

Listing 15: Wywołanie metod logujących pakietu NLog

Załóżmy, że nasza aplikacja wymaga wykorzystania narzędzia logującego. W tym celu będziemy wykorzystywać własny, wcześniej zdefiniowany interfejs `ILogger`. Użycie własnej abstrakcji uniezależnia nas od konkretnego pakietu/zestawu. Będzie on deklarował następującego metody zwracające typ `void` i przyjmował argument typu `string`:

- LogTrace
- LogDebug

[6] Targety pakietu NLog moją bardzo bogate możliwości parametryzacji. Szczegóły można znaleźć na wiki repozytorium pakietu: `https://github.com/nlog/NLog/wiki`

- LogInformation
- LogWarning
- LogError
- LogCritical

Stwórz plik z powyższym interfejsem w osobnym pliku w projekcie.

Dodaj do projektu prostą klasę, która w konstruktorze będzie oczekiwała obiektu implementującego utworzony przed chwila interfejs `ILogger`. Klasa ta będzie klientem obiektu logera.

```
public class CustomController
{
  private readonly ILogger logger;
  public CustomController(ILogger logger) => this.logger = logger;
  public void Get() => logger.LogInformation("API request");
}
```

Listing 16: Przykład klasy wykorzystującej obiekt klasy implementującej `ILogger`

Nie możemy przekazać logera NLog do obiektu typu `CustomController` ze względu na niezgodne interfejsy. Aby je dopasować wykorzystamy wzorzec projektowy Adapter. Dodaj do projektu klasę `NLogAdapter`. Klasa musi posiadać referencję do obiektu, który adaptuje. Zostało to pokazane na diagramie UML7. W tym przypadku będzie to klasa `Logger` pakietu NLog. W klasie adaptera dodaj konstruktor z parametrem typu `NLog.Logger`. Dodatkowo klasa powinna implementować interfejs `ILogger`. Zaimplementuj metody deklarowane przez ten interfejs[7], w taki sposób aby przekierowywały wywoływania metod to metod obiektu `NLog.Logger` np. w następujący sposób:

```
public class NLogAdapter : ILogger
{
  private readonly NLog.Logger logger;
  public NLogAdapter(NLog.Logger logger) => this.logger = logger;

  public void LogDebug(string message) => this.logger.Debug(message);
  //...
}
```

Listing 17: Fragment klasy adaptera dla pakietu NLog

Teraz usuń z metody `Main` napisane wcześniej sprawdzenie poprawności działania pakietu NLog i utwórz instancję klasy `CustomController`. Jako argument przekaż obiekt `NLogAdapter`. Wywołaj metodę `Get()` klasy `CustomController` i sprawdź czy informacja pojawiła się na ekranie konsoli oraz w pliku tekstowym *.log w folderze Logs. Zmiana pakietu do logowania wymagałaby teraz jedynie ściągnięcia innego pakietu np. Log4N, napisania jednej klasy adaptera i wstrzyknięcie jej do `CustomController`.

3.2 Kompozyt (ang. composite)

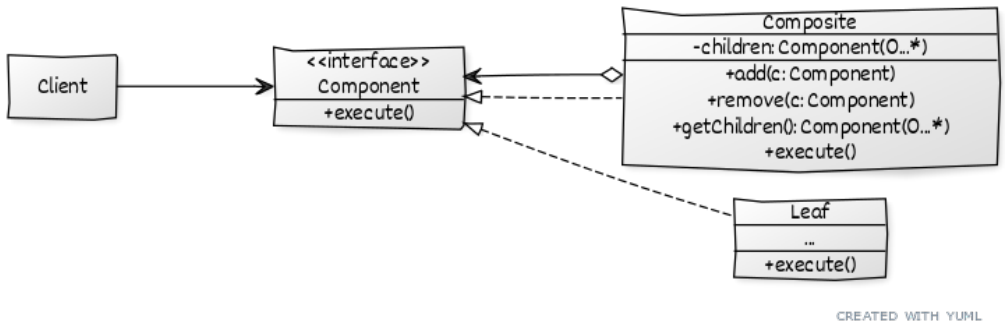
Kompozyt łączy obiekty w struktury drzewiaste i pozwala klientom traktować je jak osobne, pojedyncze obiekty. Drzewo składa się z dwóch rodzajów elementów liści oraz kontenerów. Kontener może składać się z liści oraz z innych kontenerów. Wzorzec ten powinien być stosowany jeżeli z punktu widzenia klienta nie ma znaczenia czy odwołuje się on do pojedynczego elementu czy całej drzewiastej struktury. Zazwyczaj klient nie wie z jakiego typu elementu korzysta. Klient będzie ignorował różnice pomiędzy pojedynczymi, a złożonymi obiektami. Diagram UML omawianego wzorca został pokazany na rysunku 8.

Dodatkowo istnieje możliwość dodania nowych typów, które mogą być liśćmi drzewa bez zmiany kodu klienta. Jest to cecha zgodna z zasadą otwarte/zamknięte1.2. Co więcej do tworzenia drzew kompozytowych można korzystać z wzorca Budowniczy2.2.

Wzorzec Kompozyt wykorzystuje interfejs albo klasę abstrakcyjną do reprezentacji zarówno typów prostych jak i złożonych. Wszystkie muszą implementować wspólny interfejs albo dziedziczyć po klasie abstrakcyjnej. Pewną trudnością może okazać się znalezienie wspólnego interfejsu dla wszystkim elementów

[7] Wszystkie wymagane składowe można łatwo umieścić w klasie, poprzez najechanie myszką na nazwę interfejsu (powinna być podkreślona czerwonym kolorem) i kliknięcie „Show potential fixes" (albo skrótem Alt+Enter) i wybranie opcji „Implement interface"

Kompozytu. Często prowadzi to do tworzenia skomplikowanych, rozbudowanych interfejsów co jest wadą omawianego wzorca.



Rysunek 8: Diagram UML wzorca kompozyt.

Jako przykład wykorzystania wzorca wyobraźmy sobie drzewo składające się z produktów oraz pudełek (kontenerów) które te produkty przechowują. Zarówno liście jak i kontenery implementują wspólny interfejs. Klient wywołuje metodę np. `GetPrice()` głównego kontenera (korzenia). Jeśli dzieckiem danego wierzchołka jest liść (produkt) zwracana jest jego cena, żądanie zostaje obsłużone bezpośrednio. Jeśli natomiast dzieckiem jest inny kontener (obiekt `Composite`), to przeglądana jest jego zawartość i ponownie w zależności od zawartości podejmowana jest akcja dotycząca dalszego przeglądania albo zwrócenia ceny.

Interfejsy graficzne, systemy SCADA zwykle pozwalają na rysowanie złożonych elementów z wielu innych, prostszych elementów. Klient nie chce traktować prostych i złożonych elementów w odmienny sposób. Powodowałoby to dodatkowe skomplikowanie modułu programu. Klasy `Line`, `Rectangle`, `Text` definiują proste elementy graficzne. Klasa `Picture` natomiast mogłaby stanowić kontener, w którym były przechowywane proste elementy podrzędne. Jest to jeden z przykładów, gdzie mógłby zostać wykorzystany wzorzec projektowy Kompozyt.

3.2.1 Zadanie 2

Jak w zadaniu pierwszym utwórz nowy projekt (aplikację konsolową .NET 5.0). Dodaj w nim folder o nazwie `Equipments`. Wewnątrz tego folderu dodaj klasę abstrakcyjną komponentu `Equipment`18. Definiuje ona wspólny dla wszystkich klas interfejs. To po niej będą dziedziczyły klasy liści oraz kompozytów (kontenerów). Struktura drzewiasta będzie zawierała elementy komputera PC. Płyta główna i obudowa będą pełnić funkcję kontenerów natomiast procesor, pamięć RAM oraz dysk twardy będą pełnić funkcję liści.

```
abstract class Equipment
{
  public abstract decimal NetPrice();
  public abstract decimal DiscountPrice();
  public abstract int Power();
  public virtual void Add(Equipment component) => throw new
    NotImplementedException();
  public virtual void Remove(Equipment component) => throw new
    NotImplementedException();
  public virtual bool IsComposite() => true;
}
```

Listing 18: Przykład abstrakcyjnej klasy komponentu

Następnie w tym samym projekcie dodaj klasy zarówno kontenerów np. `Chasis` oraz `Motherboard` oraz liści np. `Cpu`, `FloppyDisk`. Wszystkie powinny dziedziczyć po klasie `Equipment`. Przykładowa implementacja klasy `FloppyDisk` została pokazana na listingu 19.

```
class FloppyDisk : Equipment
{
  private readonly decimal price;
  private readonly decimal discount;
  private readonly int power;

```

```
  public FloppyDisk(decimal price, decimal discount, int power)
  {
    this.price = price;
    this.discount = discount;
    this.power = power;
  }

  public override decimal NetPrice() => price;
  public override bool IsComposite() => false;
  public override decimal DiscountPrice() => price * (1 - discount);
  public override int Power() => power;
}
```

Listing 19: Przykład klasy będącej liściem kompozytu

Klasa kontenera dodatkowo powinna posiadać pole listy na inne kontenery i liście oraz implementować metody wirtualne `Add` oraz `Remove`. Wspomniana lista może przechowywać zarówno obiekty kontenerów jak i liści ponieważ oba typy dziedziczą po tej samej klasie abstrakcyjnej. Metody `NetPrice`, `DiscountPrice` oraz `Power` klasy kontenera powinny przeglądać dodane wcześniej komponenty (liście lub inne kontenery) i zwracać sumę cen wszystkich elementów podrzędnych. Fragment klasy `MotherBoardEquipment` wraz z metodą zwracającą wartość będącą sumą cen wszystkich elementów pokazano na listingu 20.

```
class MotherBoardEquipment : Equipment
{
  protected List<Equipment> equipments = new List<Equipment>();

  //...

  public override void Add(Equipment component) => equipments.Add(
    component);
  public override void Remove(Equipment component) => equipments.Remove(
    component);
  public override decimal NetPrice()
  {
    decimal result = price;
    equipments.ForEach(x => result += x.NetPrice());
    return result;
  }

  //...
}
```

Listing 20: Fragment klasy `MotherBoardEquipment`

W klasie klienta, albo w metodzie `Main` sprawdź poprawność stworzonej struktury. Utwórz obiekt obudowy (`ChasisEquipment`) oraz dodaj do niego (za pomocą metody `Add`) kilka dysków twardych `FloppyDisk` oraz płytę główną `MotherBoardEquipment` do której wcześniej dodaj procesor `Cpu`.

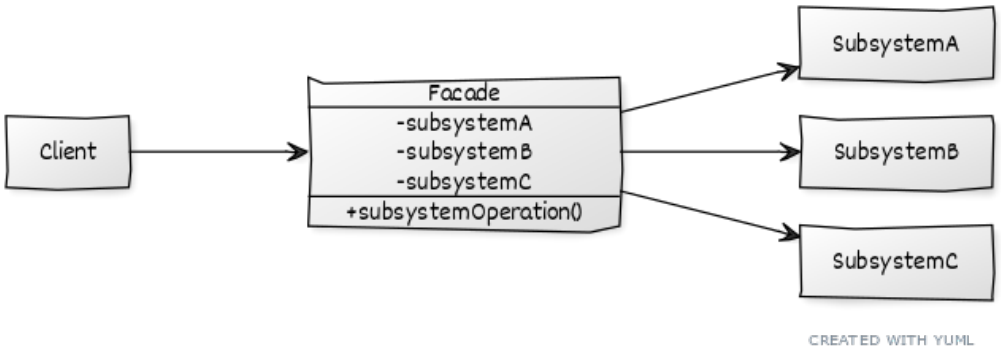
Tak utworzoną strukturę przekaż do innej klasy klienckiej. Zwróć uwagę, że z punktu widzenia klienta nie ma znaczenia czy korzysta on z pojedynczego elementu liścia (np. klasy reprezentującej procesor) czy z kontenera, który zawiera wiele dodatkowych elementów podrzędnych. Złożoność struktury drzewiastej jest przed nim ukryta przy pomocy wzorca Kompozyt.

```
class Client
{
  public void PrintBill(Equipment equipment)
  {
    Console.WriteLine($"Bill:\n\t{equipment.NetPrice()}\n\t{equipment.
   DiscountPrice()}\n\t{equipment.Power()}");
  }
}
```

3.3 Fasada (ang. facade)

Jeśli istnieje potrzeba, aby ukryć złożoność całej biblioteki albo pewnego zbioru wielu klas można użyć wzorca projektowego Fasada (diagram UML przedstawiono na rysunku 9). Jest to wzorzec upraszczający sposób w jaki klienci korzystają z współpracujących, zależnych od siebie klas. Fasada udostępnia jednolity interfejs dla zbioru interfejsów z podsystemu. Wykorzystując Fasadę klient nie musi tworzyć, ani zarządzać zależnościami pomiędzy obiektami, są one przed nim ukryte za interfejsem Fasady. Wzorzec ten może również służyć za punkt wejścia dla podsystemu w warstwowo podzielonym zestawie.

Fasada pozwala zmniejszyć liczbę obiektów z której korzystają klienci. Dodatkowo wprowadza luźne powiązanie pomiędzy klientami, a podsystemami. Ułatwione jest dzięki temu wprowadzanie zmian w podsystemach jak również zmiana zależności między nimi.



Rysunek 9: Diagram UML wzorca Fasada.

Interfejs udostępniany przez omawiany wzorzec jest uproszczony. Klasa Fasady nie musi korzystać z wszystkich funkcjonalności klas, które wykorzystuje. Dostarczane są jedynie te niezbędne.

Przykładem zastosowania Fasady może być ukrycie złożoności procesu konwersji materiału wideo. Klient **nie** korzystający z omawianego wzorca byłby zmuszony do wykorzystywania różnych klas. Inna klasa zostałaby użyta do kompresji materiału wideo inna do miksowania audio, a jeszcze inna do zmiany formatu pliku. Klient może potrzebować jedynie uproszczonego interfejsu do konwertowania pliku wideo. Nie interesują go złożone mechanizmy tego procesu. Dlatego zamiast zmuszać klienta do posiadania i zarządzania wieloma klasami, można utworzyć klasę fasady, która te złożoności przed nim ukryje. Jeśli w przyszłości część wykorzystywanych klas się zmieni, aktualizacja kodu będzie wymagana tylko w klasie Fasady.

Innym przykładem, gdzie omawiany wzorzec ma zastosowanie jest podsystem kompilujący. Również w tym przypadku klienta nie interesują złożone zależności pomiędzy parserem, preprocesorem, analizatorem semantyki czy optymalizatorem. Autorzy klientów chcą tylko skompilować kod. Jeśli proces kompilacji zostanie ukryty w pojedynczej klasie `Compiler`, to będzie ona odgrywała rolę fasady.

3.3.1 Zadanie 3

Utwórz kolejny projekt w analogiczny jak w poprzednich zadaniach sposób. Tym razem zostanie wykorzystany wzorzec projektowy Fasady do ukrycia złożoności serwisów odpowiedzialnych za pobieranie informacji o „danych meteorologicznych".

Dodaj do projektu folder `WeatherServices`. Następnie umieść w nim trzy interfejsy w osobnych plikach: `ITemperatureService`, `IHumidityService` oraz `IForecastService`. Każdy z nich niech deklaruje jedną metodę: `double GetTemperature()`, `double GetHumidity()` i `double GetForecast()`.

Dodaj do tego samego folderu trzy klasy implementujące te interfejsy. W celu wygenerowania pomiarów możesz skorzystać z generatora liczb pseudolosowych np. klasy `System.Random` i jej metody `NextDouble`. Generacja liczb z pewnego przedziału może być zrealizowana w następujący sposób:

```
1 random.NextDouble() * (maxValue - minValue) + minValue;
```

Można również wykorzystać Metodę Rozszerzającą (ang. Extension Method)[8]. Dodaj do projektu folder `Extensions`, a w nim statyczną klasę `RandomExtension`. W klasie natomiast dodaj statyczną metodę `NextDouble` o następującej postaci:

[8]Metody rozszerzające pozwalają na „dodanie" metody do już istniejącego typu, bez konieczności tworzenia nowej klasy pochodnej czy modyfikacji klasy już istniejącej. Metody rozszerzające są statyczne, ale ich wywołanie wygląda jakby należało do instancji danej klasy. Kompilator tłumaczy wywołanie tej metody na metodę statyczną.

```
public static class RandomExtensions
{
  public static double NextDouble(this Random random, double minValue,
    double maxValue)
  {
    return random.NextDouble() * (maxValue - minValue) + minValue;
  }
}
```

Zwróć uwagę na pierwszy parametr, określa on do jakiego typu (klasy) chcemy „dodać" nową metodę. Konieczne jest, aby poprzedzić go słowem kluczowym `this`. Kolejne parametry nie różnią się niczym od tych ze standardowej deklaracji metody.

Po dodaniu klasy `RandomExtension` klasa `Random` „otrzymała" nową metodę, która generuje liczbę pseudolosową typu `double` z określonego przez klienta przedziału. Metod rozszerzających nie należy stosować tam gdzie możemy modyfikować kod klasy, dodać składową. Jest to jednak dobre narzędzie jeśli chcemy dodać funkcjonalność do kodu, który nie jest pod naszą kontrolą, a dziedziczenie jest nieodpowiednie albo niemożliwe.

Klasy implementujące utworzone wcześniej interfejsy mogą teraz wykorzystywać metody rozszerzające. Listing 21 przedstawia przykładową implementację klasy `TemperatureService`.

```
public class TemperatureService : ITemperatureService
{
  public double GetTemperature() => new Random().NextDouble(-20, 40);
}
```

Listing 21: Fragment klasy `TemperatureService` wykorzystującej metodę rozszerzającą

Dalej dodaj do głównego katalogu projektu plik z rekordem[9] `Weather`:

```
public record Weather(double Temperature, double Humidity, double Forecast
  );
```

Umieść z katalogu głównym nową klasę Fasady dla wcześniej stworzonych serwisów. Klasa Fasady niech zawiera trzy pola w których przechowasz referencję do obiektów implementujących utworzone wcześniej interfejsy. W konstruktorze przypisz im przekazane przez parametry konstruktora obiekty. Dodaj również metodę `GetWeatherInformation` zwracającą obiekt typu `Weather`. Metoda ta niech pobiera informacje o temperaturze, wilgotności oraz estymacji przekazanych serwisów i zwraca jeden obiekt zawierający te dane:

```
public class WeatherApiFasade
{
  //...

  public WeatherApiFasade(ITemperatureService temperatureService,
    IHumidityService humidityService, IForecastService forecastService)
  {
    //...
  }

  public Weather GetWeatherInformation()
  {
    return  new(
      temperatureService.GetTemperature(),
      humidityService.GetHumidity(),
      forecastService.GetForecast());
  }
```

[9]Słowo kluczowe `record` wraz z `init` pojawiło się C# 9.0 i pozwala zdefiniować typ referencyjny, który służy do przechowywania danych. Tworzone w ten sposób typ jest **niezmienny** tzn. stan takiego obiektu nie może się zmienić po jego utworzeniu. Zmiana jego stanu jest możliwa jedynie poprzez utworzenie nowej instancji. Rekordy pozwalają na utworzenie nowego obiektu bez konieczności przepisywania wszystkich właściwości, jedna po drugiej. Można posłużyć się słowem kluczowym `with` i zmienić jedną właściwość, pozostałe zostaną natomiast przepisane automatycznie. Przed pojawieniem się rekordów konieczne było tworzenie właściwości jedynie z akcesorem `get` oraz ustawianie ich w konstruktorze za pomocą przekazanych przez klienta argumentów.

```
}
```

Na koniec dodaj do projektu klasę `Application`:

```
class Application
{
  public static void Run(WeatherApiFasade weatherApiFasade)
  {
    Console.Write(weatherApiFasade.GetWeatherInformation());
  }
}
```

W metodzie `Main` utwórz obiekt typu `WeatherApiFasade` przekazując do niego wcześniej utworzone obiekty wymagane przez konstruktor. Wywołaj metodę `Run()` i sprawdź czy na ekranie konsoli zostały wypisane oczekiwane wyniki:

```
static void Main(string[] args)
{
  WeatherApiFasade weatherApiFasade = new WeatherApiFasade(
    new TemperatureService(),
    new HumidityService(),
    new ForecastService());

  Application.Run(weatherApiFasade);
}
```

4 Wzroce projektowe (3) - czynnościowe

Wzorce czynnościowe dotyczą interakcji między obiektami. Umożliwiają one komunikację obiektów przy jednoczesnych zachowaniu luźnego powiązania i braku sztywnych zależności.

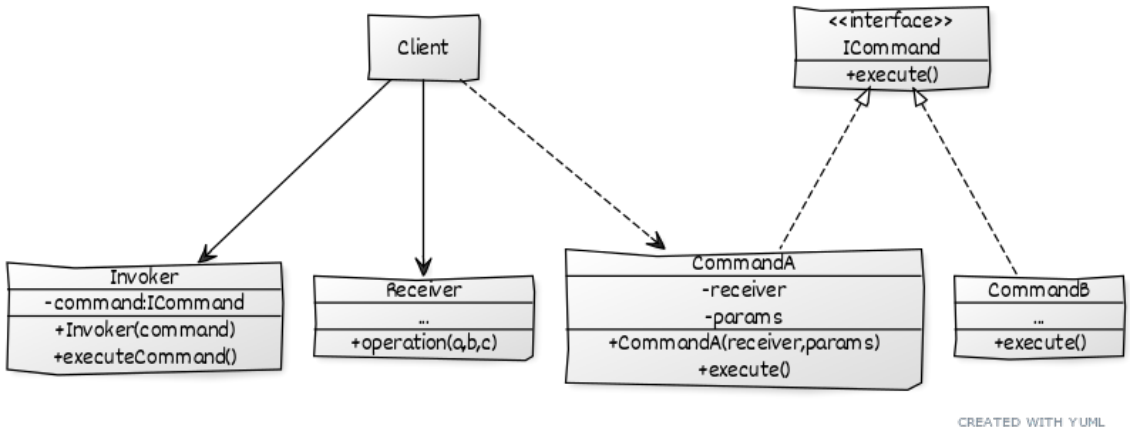
4.1 Polecenie (ang. command)

Polecenie umożliwia zamianę żądania w samodzielny, osobny obiekt. Z tym wzorcem można się spotkać w kontekście interfejsów użytkownika. Do przycisku może być przypisana akcja (obiekt implementujący interfejs `ICommand`). Obiekt przycisku powinien posiadać pole/właściwość typu `ICommand`, do którego przypisywany jest obiekt konkretnego polecenia. Po kliknięciu w element, instancja przycisku nie jest świadoma akcji, która powinna się wykonać. Pozwala to na łatwą zmianę działania przycisku i umożliwia zachowanie zasady pojedynczej odpowiedzialności 1.1 (komponent odpowiedzialny za interfejs graficzny nie jest świadomy działania pozostałych warstw aplikacji). Chcąc wprowadzić nową funkcjonalność wystarczy stworzyć nowy obiekt implementujący `ICommand` i przekazać go do nadawcy (utrzymana jest tym samym zasada otwarte/zamknięte 1.2).

Co więcej, wykorzystanie wzorca Polecenie pozwala na przypisanie kilku przyciskom tej samej czynności. Na przykład operacja zapisywania może być dostępna zarówno jako osobny przycisk orz jako kombinacja klawiszy `Ctrl+S`.

Dodatkowo omawiany wzorzec pozwala na implementacje cofania, ponawiania oraz kolejkowania operacji. Możliwe jest również łączenie wielu prostych poleceń w jedno bardziej skomplikowane.

Jeśli obiekt polecenia do wykonania żądania wymaga dodatkowych parametrów (np. z pól formularza), należy je przekazać jako argument konstruktora (jeśli obiekt ma być niezmienny/niemutowalny) albo przez jego właściwości.



Rysunek 10: Diagram UML wzorca Polecenie.

4.1.1 Zadanie 1

Utwórz projekt aplikacji konsolowej .NET 5.0. W nowym pliku dodaj interfejs `ICommand` z dwiema deklaracjami metod: `void Execute()` oraz `void Undo()`.

Dodaj klasę `Character` z jedną publiczną właściwością typu `Vector3` o nazwie `Position` (konieczne będzie dodanie przestrzeni nazw `using System.Numerics`):

```
public class Character
{
  public Vector3 Position { get; set; }
}
```

Następnie do projektu dodaj klasę o nazwie `MoveCommand` implementującą interfejs `ICommand`. Niech posiada ona konstruktor z trzema parametrami, określającymi obiekt do przesunięcia oraz wektor o jaki ma zostać przesunięty obiekt:

```
public class MoveCommand : ICommand
```

```
{
  private readonly Character objectToMove;
  private readonly Vector3 displacement;

  public Move(Character objectToMove, Vector3 displacement)
  {
    this.objectToMove = objectToMove;
    this.displacement = displacement;
  }

  //...
}
```

Aby skorzystać ze struktury `Vector3` również i w tej klasie konieczne jest dodanie przestrzeni nazw `System.Numerics` za pomocą dyrektywy `using`. Zaimplementuj w `MoveCommand` interfejs, tak aby po wywołaniu metody `Execute()` pozycja przekazanego w konstruktorze obiektu została zaktualizowana o wektor `displacement` (również przekazany jako argument konstruktora) np. w następujący sposób:

```
public class MoveCommand : ICommand
{
  //inject Character object as well as Vector3 via constructor
  public MoveCommand(Character objectToMove, Vector3 displacement)
  {
    //...
  }

  public void Execute() => objectToMove.Position += displacement;
  //add Undo method as well
}
```

Analogicznie zaimplementuj operację `Undo()`.

Wzorzec Polecenie często jest wykorzystywany do przechowywania, cofania i ponawiania jakieś operacji. Dodaj do projektu klasę `CommandHandler`, która będzie odpowiedzialna za przechowywanie poleceń i umożliwiała ich cofanie oraz ponowne wywoływanie:

```
public class MoveCommandHandler
{
  private readonly Stack<ICommand> commands = new Stack<ICommand>();
  private readonly Stack<ICommand> redos = new Stack<ICommand>();

  public void Handle(ICommand command)
  {
    //clear the redos container
    //push the command to the commands container
    //execute the command
  }
  public void Undo()
  {
    if (commands.Any()) //checks if the container has any elements
    {
      redos.Push(commands.Peek()); //takes the first element from the
   stack and adds it to the redoes
      commands.Pop().Undo(); //takes the first command from the stack and
   executes it
    }
  }
  public void Redo()
  {
    //as in the preceding method
  }
```

```
}
```

Zaimplementuj metody: `Handle`, `Undo` oraz `Redo`.

Następnie w klasie `Character` dodaj prywatne pole typu `MoveCommandHandler`. Dodatkowo umieść w niej metody `Move`, `ComeBack` oraz `MoveAgain`. Będą one obsługiwać operacje „przesuwania" obiektu typu `Character` wykorzystując `MoveCommandHandler`.

```
public class Character
{
  public Vector3 Position { get; set; }
  private readonly MoveCommandHandler commands = new();

  public void Move(ICommand command)
  {
    commands.Handle(command);
  }

  public void ComeBack()
  {
    //...
  }

  public void MoveAgain()
  {
    //...
  }

  public override string ToString() => $"Current position: {this.Position
    }";
}
```

Dodaj implementację metod `ComeBack` oraz `MoveAgain`. Powinny one jedynie wywoływać metody `Undo` oraz `Redo` obiektu typu `MoveCommandHandler`.

W metodzie `Main` utwórz obiekt typu `Character` i cztery obiekty typu `MoveCommand`. Następnie wywołując metody `Move`, `ComeBack` i `MoveAgain` zbadaj poprawność napisanych klas (sprawdź pod debuggerem albo wypisz na ekranie konsoli właściwość `Position` obiektu typu `Character`):

```
class Program
{
  static void Main(string[] args)
  {
    var character = new Character();
    MoveCommand up = new MoveCommand(character, new Vector3(0f, 1f, 0f));
    //...

    character.Move(up);
    Console.WriteLine(character.ToString());

    //...

    Console.ReadLine();
  }
}
```

4.2 Strategia (ang. strategy)

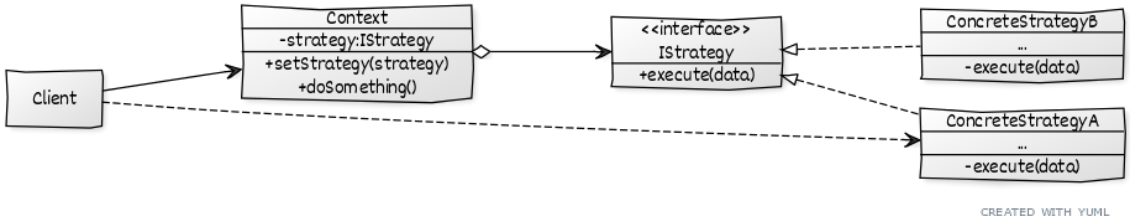
Klient, aby zrealizować pewne zadania może wykorzystywać konkretny algorytm. Często może się on zmieniać w czasie oraz być zależny od dodatkowych czynników. Może on się zmienić w zależności od opcji, którą wybrał użytkownik programu. Wzorzec Strategia pozwala utworzyć rodzinę algorytmów realizujących te samo zadanie w różny sposób. Dalej za pomocą konstruktora, właściwości albo metody klient może

podmienić obiekt strategii i tym samym zmienić sposób realizacji danej czynności przez daną klasę. Gorszą alternatywą mogłoby być wbudowanie na stałe wielu algorytmów w daną klasę i naruszenie tym samym pierwszych dwóch zasad **SOLID** (1.1 i 1.2).

Strategia bazuje na mechanizmie odwracania zależności1.5. Koncepcyjnie DI jest raczej zarezerwowane dla przypadków, gdzie podczas wykonywania się programu dane zachowanie (wstrzykiwany obiekt) jest stałe. Jeśli natomiast zakładamy, że sposób realizacji danego algorytmu będzie się zmieniał to wtedy odnosimy się do wzorca Strategia. Różnice między nimi są niewielkie i subtelne. Pod względem struktury Strategia jest również podobna do Mostu i Stanu, które także bazują na kompozycji. Mimo, że pod względem struktury są ona do siebie bardzo podobne, nazewnictwo stosowane w tych wzorcach powinno programistę informować o specyficznym problemie, który został za ich pomocą rozwiązany.

Warto również wspomnieć, że w programowaniu obiektowym istnieje zasada, aby przekładać kompozycję nad dziedziczenie. Podobnym do strategii wzorcem projektowym jest wzorzec Metoda Szablonowa[10], który wykorzystuje dziedziczenie. Konkretny algorytm w tym przypadku jest definiowany w klasie pochodnej. Te wiązanie odbywa się podczas kompilacji i sprawia, że algorytmu nie można zmieniać w trakcie działania aplikacji.

Diagram UML omawianego wzorca został przedstawiony na 11. Klient tworzy obiekt konkretnej strategi (musi ona implementować wspólny dla wszystkich strategii interfejs). Dalej taka instancja może zostać przekazana np. przy wykorzystaniu metody/właściwości (`SetStrategy(strategy)`) do obiektu realizującego daną czynność. Klient ma możliwość w trakcie działania programu płynnie zmieniać sposób realizowania danej czynności.



Rysunek 11: Diagram UML wzorca Strategia.

Jeżeli obiekt strategii potrzebuje dodatkowych argumentów mogą one zostać udostępnione przez kontekst/klienta w momencie wywoływania algorytmu. Ewentualnie `Context` może przekazać do obiektu strategii samego siebie.

Przykładem zastosowania wzorca Strategia może być nawigacja i różne algorytmy obliczania optymalnej drogi. Na przykład w zależności od środka transportu może być stosowany inny obiekt `ConcreteStrategy`. W bibliotece `ObjectWindows` strategia jest stosowana w oknach dialogowych do sprawdzania poprawności wprowadzanych przez użytkownika danych. Zestaw oferuje część predefiniowanych algorytmów, a użytkownik jeśli nie znajdzie dla niego odpowiedniego, może łatwo rozszerzyć możliwości przez stworzenie własnego obiektu implementującego konkretny interfejs. Strategia może również być wykorzystywana w algorytmach parsujących (różne algorytmy w zależności od formatu danych wejściowych) albo optymalizacji kompilowanego kodu (inne obiekty dla poszczególnych rodzajów i poziomów kompilacji).

4.2.1 Zadanie 2

W solucji dodaj kolejny projekt aplikacji konsolowej .NET 5.0. Dodaj w nim interfejs `IRouteStrategy` z jedną metodą `BuildRoute`, nie przyjmującą żadnych argumentów i zwracającą typ `void`.

Następnie dodaj trzy klasy: `PublicTransportStrategy`, `RoadStrategy` i `WalkingStrategy` implementujące interfejs `IRouteStrategy` np. w następujący sposób[11]:

```
1 public class PublicTransportStrategy : IRouteStrategy
2 {
3   public void BuildRoute() => Console.WriteLine("Public transport strategy
      has been used for travel time estimation.");
4 }
```

[10] Metoda Szablonowa jest wzorcem, który definiuje szkielet algorytmu w klasie bazowej i pozwala go parametryzować, zmieniając niektóre jego kroki w klasie pochodnej. Dokładny opis tego wzorca można znaleźć na:`https://refactoring.guru/design-patterns/template-method`

[11] W rzeczywistym przypadku metody `BuildRoute` powinny przyjmować współrzędne lokalizacyjne i zwracać faktyczną drogę umożliwiającą przemieszczenie się z punktu A do punktu B.

Dalej dodaj do projektu klasę `Navigator`, w której umieść konstruktor z argumentem typu `IRouteStrategy`. Wewnątrz tego konstruktora przypisz przekazany argument do publicznej właściwości (wcześniej ją dodaj do klasy).

Dodatkowo w klasie `Navigator` dodaj metodę `Navigate`:

```
public class Navigator
{
  //...

  public void Navigate() => this.routeStrategy.BuildRoute();
}
```

W metodzie `Main` sprawdź działanie przykładowego programu:

```
static void Main(string[] args)
{
  var context = new Navigator(new RoadStrategy());
  context.Navigate();

  context.SetStrategy(new PublicTransportStrategy());
  context.Navigate();

  context.SetStrategy(new WalkingStrategy());
  context.Navigate();
}
```

4.3 Mediator (ang. mediator)

Wzorzec projektowy mediator umożliwia kapsułkowanie interakcji pomiędzy obiektami poprzez dodatkowy obiekt mediatora. Obiekty mogą się komunikować jedynie wykorzystując ten pośredniczący obiekt.

Wyobraźmy sobie sytuację, w której okno dialogowe składa się z wielu widgetów. Pewne pole może być aktywne albo nieaktywne w zależności od wybranej wcześniej opcji widgetu typu `Checkbox`. W innym przypadku przycisk do wysłania formularza aktywowany jest dopiero po wypełnieniu wszystkich niezbędnych pól. Bez użycia obiektu mediatora poszczególne instancje widgetów musiałby posiadać referencje do sobie nawzajem, aby np. po zaznaczeniu pola wyboru, aktywować pole tekstowe. Ponowne użycie takich elementów staje się utrudnione, a czytelność kodu się pogarsza.

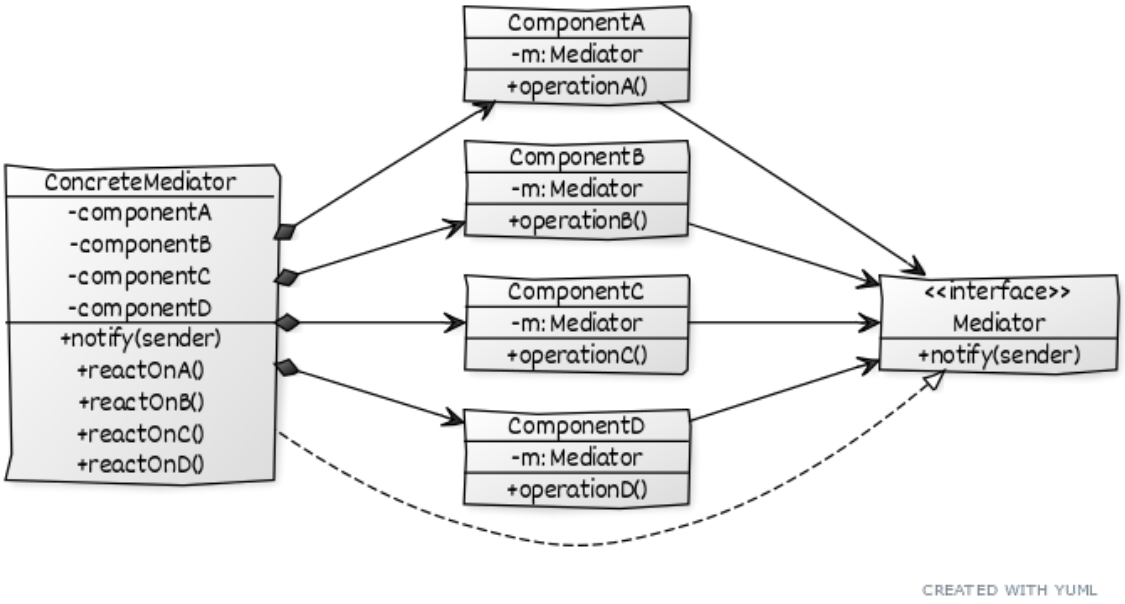
Zamiast zmuszać obiekty do posiadania informacji o sobie nawzajem można wprowadzić dodatkowy obiekt mediatora (diagram UML przedstawiono poniżej12), który będzie pośredniczył w komunikacji między obiektami. Komponenty w tej sytuacji są zależne jedynie od obiektu moderatora i to do niego kierują informację np. o zmianie stanu pola wyboru. Na przykład w momencie użycia przycisku przez użytkownika, obiekt `Button` mógłby wysyłać informację o operacji do mediatora, który sprawdzałby pozostałe pola i podejmował decyzję o wysłaniu formularza.

Obiekty mogą się komunikować np. z wykorzystaniem zdarzeń albo wzorca Obserwator. We wzorcu MVC (ang. Model View Controller - Model Widok Kontroler) mediatorem może być nazwany kontroler.

Wzorzec Mediator warto zastosować jeśli istnieje wiele zależności pomiędzy klasami, a ich ponowne użycie w innym programie jest z tego powodu mocno utrudnione. Podobnym wzorcem do Mediatora jest Fasada. Główną różnicą między nimi jest fakt, że ta druga posiada zależności jednokierunkowe. Obiekty reprezentowane przez fasadę mogą kierować żądania do podsystemów, ale podsystemy nie mogą komunikować się z Fasadą.

Wadą omawianego wzorca jest fakt, że mediator z czasem może stać się bardzo złożony tzw. Boski Obiekt[12]. Należy dodatkowo zaznaczyć, że Mediator powinien być raczej używany w większych projektach. Wprowadza on dodatkową złożoność. Tak samo jak z innymi wzorcami projektowymi, jeśli tworzymy proste aplikacje, albo krótkie przykłady tak jak na laboratoriach to użycie niektórych wzorców jest zbędne albo wręcz niepożądane. Nie ma sensu komplikować prostego projektu niewielkich rozmiarów. Jeśli jednak aplikacja zaczyna się rozrastać to dobre praktyki i wzorce zaczynają mieć znaczenie. Tak samo Madiator jak i opisany w kolejnych akapitach wzorzec CQRS **nie powinno** się sotoswać zawsze, gdy nadarzy się okazja.

[12]Boskim Obiektem jest nazywana klasa, która posiada bardzo dużo odpowiedzialności i łamie jednocześnie zasadę pojedynczej odpowiedzialności 1.1. Taki obiekt jest przykładem antywzorca projektowego.



Rysunek 12: Diagram UML wzorca Mediator.

4.3.1 CQRS oraz framework MediatR

W prostych aplikacjach webowych często korzysta się z tzw. podejścia CRUD[13]. W sytuacji wielu zapytań o różne obiekty danych (DTO) aplikacja musi wykonywać wiele operacji mapowania. Dodatkowo proces zapisu może wymagać złożonej walidacji i dodatkowej logiki biznesowej. Powoduje to, że model staje się skomplikowany i trudny w utrzymaniu. Mimo tych wad CRUD jest bardzo często (i słusznie) stosowany w prostych aplikacjach. Jeśli domena jest prosta albo CRUD jest wystarczający nie ma większego sensu stosować omówionego poniżej wzorca CQRS.

CQRS (ang. Command Query Responsibility Segregation) jest wzorcem, który oddziela zapytania (ang. Query) i polecenia (ang. Command) do bazy danych, ułatwiając skalowanie i zapewniając większą kontrolę nad wydajnością. Warto w tym miejscu przypomnieć, że zapytania `Queries` nie powinny modyfikować danych, mogą one jedynie je pobierać. Polecenia natomiast służą do dodawania nowych danych, usuwania ich albo aktualizacji. W CQRS używany jest inny model do zapisu i odczytu. Pisząc o różnych modelach mamy na myśli różne modele obiektów, które mogą działać w różnych procesach, albo na oddzielnych maszynach. Np. w przypadku stron www renderowanie może wykorzystywać model zapytań. Jeśli natomiast użytkownik wykona jakąś akcję np. „danie" kciuka w górę na portalu społecznościowym to takie polecenie jest wysyłane do osobnego modelu do przetworzenia. Innymi korzyściami jest zwiększenie bezpieczeństwa ponieważ ta sama encja nie jest wykorzystywana zarówno do operacji czytania i zapisywania. W przeciwnym przypadku informacje mogłyby zostać użyte w złym kontekście.

Wykorzystywane modele są elastyczne i łatwe w utrzymaniu. Skomplikowana logika biznesowa jest zazwyczaj skupiona w modelu zapisu, natomiast odczyt może pozostać względnie prosty. Komponent ten wysyła zazwyczaj jedynie zapytanie do bazy danych oraz serializuje wynik do obiektu DTO.

CQRS pozwala również na **fizyczne** oddzielenie danych do czytania i danych do zapisu. Model poleceń może być utrwalany w jednej bazie danych, natomiast model zapytań w drugiej. W takich sytuacjach często model zapisu zgłasza zdarzenie za każdym razem, gdy baza danych jest aktualizowana. Baza przechowująca dane do czytania może być taką samą bazą jak ta do zapisu, albo mogą one posiadać zupełnie różne struktury. Takie rozwiązanie pozwala również na sterowanie obciążeniem systemu (zazwyczaj operacje czytania występują częściej niż zapisu). Można również wykorzystać różne struktury, techniki dostępu do danych zoptymalizowane dla zapisu i odczytu. Plusem jest również, że oddzielna część programistów może skupić się na osobnych modelach.

Minusem tego wzorca jest wprowadzanie dodatkowej złożoności do programu. Co więcej, jeśli mamy rozdzielone bazy danych dla zapisu i odczytu musimy zapewnić ich spójność.

CQRS jest warto stosować jeśli wiele użytkowników oczekuje równoległego dostępu do danych. Oma-

[13]Skrót CRUD (ang. Create Read Update Delete) określa cztery podstawowe operacje służące do utrwalania danych np. w bazie danych. Za pomocą tych operacji kontroler albo inny komponent aplikacji może dodawać, odczytywać, aktualizować i usuwać rekordy w bazie danych.

wiany wzorzec pozwala zminimalizować liczbę konfliktów na poziomie domeny. Dodatkowo jeśli polecenia są skomplikowane, albo trzeba je wykonać wielokrotnie aby zrealizować daną operację np. połączyć dane w obiekt transferu danych (DTO), to model zapisu musi być wstanie obsłużyć wiele poleceń, wykorzystać logikę biznesową oraz walidować wejścia. Operacje te muszą być wykonywane, aby zapewnić poprawny stan danych. Im dana dziedzina jest bardziej skomplikowana tym stosowanie CQRS staje się bardziej uzasadnione.

4.3.2 Zadanie 3

Utwórz dwa nowe projekty w solucji:

- ASP.NET Core Web API o nazwe MediatorPatternApi (zwróć uwagę, aby zaznaczona była opcja „Enable OpenAPI support")
- Class library o nazwie MediatorPatternLib

Oba niech korzystają z .NET 5.0 i domyślnych ustawień projektu.

Zainstaluj pakiet NuGet `MediatR.Extensions.Microsoft.DependencyInjection`. Umożliwia on zarejestrowanie wszystkich handlerów MediatR do kontenera DI. Wraz z nim zainstaluje się również sam pakiet `MediatR`[14].

W projekcie MediatorPatternLib w folderze Models umieść klasę o nazwie `PersonModel` o następujących właściwościach: `Id` typu `int`, oraz `FirstName` i `LastName` typu `string`.

Dalej w projekcie MediatorPatternLib utwórz pięć folderów: Commands, DataAccess, Handlers, Models oraz Queries. W folderze Commands dodaj rekord:

```
public record InsertPersonCommand(string FirstName, string LastName) :
    IRequest<PersonModel>;
```

, a do folderu Queries dodaj dwa rekordy w osobnych dwóch plikach:

```
public record GetPersonByIdQuery(int Id) : IRequest<PersonModel>;
```

oraz

```
public record GetPersonListQuery() : IRequest<List<PersonModel>>;
```

Generyczne interfejsy `IRequiest<>` pochodzą z dodanego przed chwilą pakietu `MediatR`. Aby móc skorzystać w tych interfejsów, konieczne będzie dodatnie odpowiedniej przestrzeni nazw za pomocą dyrektywy `using MediatR`.

Dodaj do folderu DataAccess klasę `DataAccessMocker` implementującą `IDataAccess`, która będzie służyła jako źródło danych:

```
public class DataAccessMocker : IDataAccess
{
  private readonly List<PersonModel> people = new List<PersonModel>();

  public DataAccessMocker()
  {
    people.Add(new PersonModel
    {
      Id = 1,
      FirstName = "Joey",
      LastName = "Tribbiani"
    });
    people.Add(new PersonModel
    {
      Id = 2,
      FirstName = "Monica",
      LastName = "Geller"
    });
  }

```

[14] MediatR jest prostą implementacją wzorca Mediator dla .NETu.

```
  public List<PersonModel> GetPeople() => people;

  public PersonModel InsertPerson(string firstName, string lastName)
  {
    PersonModel p = new PersonModel()
    {
      FirstName = firstName,
      LastName = lastName
    };

    p.Id = people.Max(x => x.Id) + 1;
    people.Add(p);

    return p;
  }
}
```

W osobnym pliku umieść interfejs `IDataAccess`:

```
public interface IDataAccess
{
  List<PersonModel> GetPeople();
  PersonModel InsertPerson(string firstName, string lastName);
}
```

Teraz pozostało dodać handlery, zarówno dla poleceń jak i zapytań. W folderze Handlers umieść następujące klasy:

```
public class GetPersonListHandler : IRequestHandler<GetPersonListQuery,
    List<PersonModel>>
{
  private readonly IDataAccess data;

  public GetPersonListHandler(IDataAccess data) => this.data = data;

  public Task<List<PersonModel>> Handle(GetPersonListQuery request,
    CancellationToken cancellationToken)
  {
    // FromResult take synchronous call and turns it into an asynchronous
    return Task.FromResult(data.GetPeople());
  }
}
```

```
public class GetPersonByIdHandler : IRequestHandler<GetPersonByIdQuery,
    PersonModel>
{
  private readonly IDataAccess data;

  public GetPersonByIdHandler(IDataAccess data) => this.data = data;

  public Task<PersonModel> Handle(GetPersonByIdQuery request,
    CancellationToken cancellationToken)
  {
    return Task.FromResult(data.GetPeople().FirstOrDefault(x => x.Id ==
    request.Id));
  }
}
```

```
public class InsertPersonHandler : IRequestHandler<InsertPersonCommand,
    PersonModel>
```

```
{
  private readonly IDataAccess data;

  public InsertPersonHandler(IDataAccess data) => this.data = data;

  public Task<PersonModel> Handle(InsertPersonCommand request,
   CancellationToken cancellationToken)
  {
    return Task.FromResult(data.InsertPerson(request.FirstName, request.
   LastName));
  }
}
```

Pierwsza będzie odpowiedzialna za pobieranie listy wszystkich modeli osób (`PersonModel`), druga pojedynczego modelu po identyfikatorze `Id`, trzecia do dodania nowego modelu do „bazy danych".

W projekcie MediatorPatternApi dodaj klasę kontrolera do folderu `Controllers`:

```
[Route("api/[controller]")]
[ApiController]
public class PersonController : ControllerBase
{
  private readonly IMediator mediator;

  public PersonController(IMediator _mediator) => mediator = _mediator;

  // GET: api/<PersonController>
  [HttpGet]
  public async Task<List<PersonModel>> Get() //a list
  {
    return await mediator.Send(new GetPersonListQuery());
  }

  // GET api/<PersonController>/5
  [HttpGet("{id}")]
  public async Task<PersonModel> Get(int id) //get one item by id
  {
    return await mediator.Send(new GetPersonByIdQuery(id));
  }

  // POST api/<PersonController>
  [HttpPost] // insert
  //one liner, very simple controller
  public async Task<PersonModel> Post([FromBody] PersonModel value) =>
   await mediator.Send(new InsertPersonCommand(value.FirstName, value.
   LastName));
}
```

Na koniec w pliku `Startup.cs` w metodzie `ConfigureServices` dodaj do kontenera rejestracje klasy `IDataAccess` oraz rejestrację wszystkich zależności handlerów i typów mediatora:

```
public void ConfigureServices(IServiceCollection services)
{

  //...
  services.AddSingleton<IDataAccess, DataAccessMocker>();
  services.AddMediatR(System.Reflection.Assembly.Load(nameof(
   MediatorPatternLib)));
}
```

Po wykonaniu powyższych kroków i uruchomieniu projektu MediatorPatternApi powinniśmy móc skorzystać z utworzonego web serwisu wykorzystującego wzorzec mediator oraz CQRS. Skompiluj i uruchom

program. Ponieważ podczas tworzenia projektu MediatorPatternApi domyślnie została wybrana opcja Enable OpenAPI support, po uruchomieniu projektu otworzy się w oknie przeglądarki internetowej Swagger UI. Pozwoli on na przetestowanie REST API naszej aplikacji.

5 TDD ang. Test-Driven-Developlment

Wytwarzanie oprogramowania sterowane testami (TDD ang. Test-Driven Development) polega na wytwarzaniu oprogramowania w krótkich powtarzających się cyklach:

1. napisanie testu, który będzie się kończył niepowodzeniem (nie ma jeszcze kodu, który można byłoby przetestować),
2. napisanie kodu powodującego wykonanie się z wynikiem pozytywnym wcześniej napisanego testu,
3. refaktoryzacja napisanego kodu.

Testy powinny być automatyczne i same sprawdzać poprawność wyników.

Korzyścią ze stosowania tego podejścia jest posiadanie testu dla każdego fragmentu kodu. Dodawanie nowych funkcjonalności może odbywać się bez obaw, że nasze działania przyczynią się do awarii systemu. Jest to bardzo istotna zaleta pisania testów. Innym powodem przemawiającym za stosowaniem TDD jest inne spojrzenie na wytwarzane oprogramowania. Kod projektowany z myślą o testowaniu wymusza luźne powiązania komponentów. TDD pozwala podjąć dobre decyzje dotyczące struktury programu. Podejście TDD zmusza programistę do skupienia się na interfejsie, a nie implementacji. Dodatkowo testy jednostkowe, które powstają naturalnie podczas TDD pełnią bardzo dobrą formę dokumentacji wytwarzanego oprogramowania. Stają się one przykładami użycia pisanych przez nas klas i tym samym ułatwiają pracę z naszym projektem innym programistom.

Istotne jest, aby testować oprogramowanie pod kątem tego co może pójść źle (tzw. "brudny test"). Częstym błędem jest poświęcanie większej części czasu na sprawdzania standardowego, poprawnego działania kodu.

Dane wejściowe powinny być tak dobrane, aby mogły wyjawić błędy niewidoczne w innych przypadkach. Nie ma sensu testować modułu dla danych, które są zbliżone do przypadku z danymi już sprawdzonymi. Jeśli dwa testy prowadzą do wykrycia tego samego błędu najprawdopodobniej potrzeby jest tylko jeden test. Dodatkowo bazując na doświadczeniu warto przewidywać miejsca, gdzie mogą znajdować się błędy.

Obszarem wartym przetestowania są wartości graniczne. Pozwalają na wykrycie np. błędów związanych z użyciem znaku `>` zamiast `>=`. W przypadku wartości granicznych można napisać trzy testy dla wartości równej badanej, trochę większej oraz nieco mniejszej. Warto również sprawdzić przypadki mnożenia, dzielenia dużych liczb przez siebie albo w sytuacji, gdy jedna z nich ma wartość zero. Można sprawdzić klasy złych danych np. gdy posiadamy zbyt mało danych, zbyt wiele danych, dane mają zły rodzaj, zły rozmiar albo nie zostały zainicjalizowane. Nie należy zapominać o sprawdzeniu przypadków nominalnych (standardowych) wraz z wartościami minimalnymi oraz maksymalnymi. Dodatkowo warto ułatwić sobie prace i sprawdzać wartości łatwe do ręcznego obliczenia, tak aby zmniejszyć ryzyko pomyłki.

W przypadku testów jak i w wielu innych obszarach obowiązuje zasada pareto czyli 80% błędów znajduje się w 20% procentach klas lub procedur projektu. Należy pamiętać, że zwiększenie jakości skraca czas pracy i zmniejsza koszty. W latach '70 oraz '80 przeprowadzono badania[2] i ustalono, że około 95% błędów powodują programiści, 2% oprogramowanie systemowe (kompilator i system operacyjny), 2% inne oprogramowanie i około 1% warstwa sprzętowa. Aktualnie można przypuszczać, że udział błędów programisty jest większy. Na błędy programistów składają się zarówno błędy popełnione przy pisaniu kodu jak i defekty związane z wymaganiami oraz błędy w projekcie. Błędy zdarzają się równie często w testowanym kodzie jak w samych testach. Dlatego należy testy pisać z taką samą starannością jak główny kod.

Czasami przyjmuje się, że minimalna liczba testów pozwalająca pokryć kod wynosi jeden dla liniowej ścieżki wykonywania programu. Dalej dla każdego wyrażenia `if`, `while`, `for` itd. należy dodać jeden test. Dodatkowo należy przewidzieć jeden dodatkowy test dla każdego przypadku `case` w wyrażeniu `switch-case`. Jest to jednak minimalny zbiór testów nie uwzględniający różnych kombinacji danych wejściowych.

Poza testami jednostkowymi istnieją również inne rodzaje testów.

Jednym z nich są testy integracyjne polegają na uruchomieniu dwóch lub większej liczby klas, pakietów, komponentów lub podsystemów. Ten rodzaj testów jest wykonywany zazwyczaj jak tylko dwie pierwsze klasy zostaną napisane i trwa do ukończenia projektu. Można je przeprowadzić np. przy wykorzystaniu oprogramowania Selenium, który automatyzuje testowanie aplikacji na poziomie przeglądarki internetowej. Zamiast ręcznie klikać poszczególne elementy UI, czynności te mogą być wykonywane automatycznie przez oprogramowanie testujące.

Drugim rodzajem są testy akceptacyjne podczas których klient sprawdza czy wymagania zostały poprawnie zrealizowane. Powinny być pisane przez osoby nieznające wewnętrznych mechanizmów systemu. Testy

jednostkowe zwykle pisane są przez samych programistów. Testy akceptacyjne zazwyczaj są automatyzowane, pisane w językach specyfikacji (np. FitNesse). Te testy również mają istotny wpływ na architekturę. Aplikacja musi zostać podzielona na odpowiednie komponenty umożliwiające przetestowanie jej. Interfejs użytkownika musi być oddzielony od logiki biznesowej.

Testowanie nie jest jednak panaceum na wszystkie problemy z oprogramowaniem. Ich stosowanie pozwala znaleźć niespełna 60% defektów[2]. Pomyślnie zakończone procedury testowania nie oznacza, że wszystkie błędy zostały znalezione. Brak błędów może również oznaczać, że testowanie było nieskuteczne. Zwiększanie liczby testów nie powoduje też, że kod staje się lepszy, konieczna jest również refaktoryzacja kodu.

Często testowana klasa posiada zależność od innego obiektu np. wstrzykiwanego przez konstruktor. Np. obiekt implementujący `IRepository` może być konieczny, w klasie usługi. Ponieważ zależy nam, aby testować tylko jedną klasę, tę drugą, wstrzykiwaną należy spreparować. W tym celu wykorzystuje się tzw. mockery. Pozwalają one na symulację działania zależności takich jak bazy danych, połączenia sieciowe czy operacje wejścia-wyjścia. Jeżeli testowana klasa wykorzystuje obiekt klasy czytającej informacje z bazy danych albo z web serwisu to nie powinna być przekazywana faktyczna implementacja tej klasy, a jednie klasa mockera implementująca wspólny interfejs z klasą „rzeczywistą". Testowana powinna być sama klasa, a nie klasa wraz w innymi obiektami, których potrzebuje do działania.

5.1 NUnit

NUnit jest frameworkiem do pisania testów jednostkowych. Jest on darmowy, również do zastosowania komercyjnego (licencja MIT). Posiada on bardzo prostą krzywą uczenia, bez większej znajomości technik testowania, każdy powinien być w stanie przetestować pisaną klasę.

5.1.1 Zadanie 1

Utwórz solucję zawierającą projekt `Class library` oraz `NUnit Test Project`. Pierwszy projekt nazwij `AbstractionDataTypes`, drugi `AbstractionDataTypesUnitTests`.

Do projektu `AbstractionDataTypesUnitTests` dodaj referencję do projektu `AbstractionDataTypes`. Aby to zrobić należy kliknąć PPM na projekt `AbstractionDataTypesUnitTests` i dalej `Add->Project references...` i wskazać `AbstractionDataTypes`.

Klikając `View->Test Explorer` albo za pomocą skrótu klawiszowego `Ctrl+E,T` otwórz widok eksploratora testów.

W tym miejscu mamy działające środowisko. W oknie `TestExplorer` możemy uruchomić testy klikając przycisk `Run All Tests`. Jedyny utworzony automatycznie test powinien się wykonać poprawnie.

Zanim rozpoczniemy pisać kod w bibliotece zgodnie z zasadą TDD należy przygotować pierwszy test, który zakończy się niepowodzeniem. Zamień metodę `Test1` na `CreateCustomStack` umieść w niej poniższy kod:

```
[Test]
public void CreateCustomStack()
{
  CustomStack sut = new CustomStack();
}
```

Kod nie przejdzie kompilacji ponieważ klasa `CustomStack` jeszcze nie istnieje. Utwórz tę klasę w projekcie `AbstractionDataTypes` i dodaj dyrektywę `using AbstractDataTypes` w pliku `CustomStackUnitTests.cs`. Solucja powinna przejść proces kompilacji.

Ponieważ utworzony stos powinien być pusty dodaj metodę asercji[15] na końcu metody `CreateCustomStack`:

```
[Test]
public void CreateCustomStack()
{
  CustomStack sut = new CustomStack();
  Assert.IsTrue(sut.IsEmpty());
}
```

W klasie `CustomStack` brakuje metody `IsEmpty()`, klikając na nazwę tej metody w wyrażeniu asercji PPM albo używając skrótu klawiszowego Alt+Enter, dodaj metodę do klasy `CustomStack`. Kod powinien przejść

[15] Asercje są predykatami (zwracają wartość `true` albo `false`), w zależności od wyniku wyrażenia danej asercji. W przypadku testów jednostkowych jeśli wynik asercji jest poprawny z założeniem, to test kończy się powodzeniem w przeciwnym przypadku dany test kończy się niepowodzeniem.

kompilację. Uruchom testy jednostkowe, zakończą się niepowodzeniem. Aby sprawić, że testy się wykonają zwróć wartość `true` w funkcji `IsEmpty()`, w metodzie TDD staramy się w pierwszej kolejności przejść testy po linii najmniejszego oporu. Dodatkowo możesz zmienić nazwę metody `CreateCustomStack` na np. `ShouldBeEmptyWhenCreated`, lepiej obrazującą co robi dany test. Zrób to korzystając z narzędzia refaktoryzacji ustawiając kursor myszki na nazwie metody i klikając `Ctrl+R+R`. Uruchom testy, powinny one wykonać się poprawnie.

Dodaj następnie kolejny test:

```
[Test]
public void ShouldNotBeEmptyWhenElementPushed()
{
  CustomStack sut = new CustomStack();
  sut.Push(0);
  Assert.IsFalse(sut.IsEmpty());
}
```

Jak wcześniej dodaj metodę `Push` do klasy `CustomStack`. Uruchom testy, zakończą się one niepowodzeniem. Umieść w klasie `CustomStack` poniższy kod, aby sprawić, żeby testy wykonały się poprawnie:

```
public class CustomStack
{
  private bool isEmpty = true;
  public bool IsEmpty() => isEmpty;
  public void Push(int element) => this.isEmpty = false;
}
```

Ponieważ testy wykonały się bez błędów można teraz wykonać prostą refaktoryzację. W klasie, która zawiera testy (`CustomStackUnitTests`), utwórz pole typu `CustomStack`. Zwróć uwagę, że automatycznie z klasa `CustomStackTests` dodana została metoda `Setup` z atrybutem `[SetUp]`[16]. Wewnątrz tej metody przypisz do utworzonego przed chwilą pola, instancję klasy `CustomStack` (wywołaj konstruktor tej klasy używając słowa kluczowego `new`). Usuń linijki kodu odpowiedzialne za tworzenie instancji klasy `CustomStack` w metodach testujących. Od teraz przed każdym uruchomieniem każdego z testów, zostanie wywołana metoda `Setup`. Utworzy ona nową instancję klasy stosu. Uruchom ponownie testy, powinny się one wykonać poprawnie.

Utwórz kolejną metodę testującą np. `ShouldThrowAnExceptionIfStackEmptyAndPopped`. Dodaj do niej wywołanie metody usunięcia elementu ze stosu `sut.Pop()`. Zakładając, że nowo stworzony stos jest pusty, to testowana metoda powinna zgłosić wyjątek np. `InvalidOperationException`. Można również napisać własny wyjątek dziedziczący po `Exception`. Dodaj implementację metody jedynie w zakresie niezbędnym do przejścia napisanego testu. Aby w `NUnit` sprawdź czy został poprawnie rzucony wyjątek należy posłużyć się generyczną metodą `Assert.Throws<>`, która jako argument przyjmuje delegat będący wywołaniem testowanej metody:

```
[Test]
public void ShouldThrowAnExceptionIfStackEmptyAndPopped()
{
  Assert.Throws<System.InvalidOperationException>(() => sut.Pop());
}
```

Testy powinny się zakończyć powodzeniem.

Następnie dodaj następny test, który ponownie zakończy się on niepowodzeniem. Niech sprawdza on czy po wrzuceniu i usunięciu elementu ze stosu kolekcja jest pusta:

```
[Test]
public void ShouldBeEmptyIfPushedAndPopped()
{
  sut.Push(0);
  sut.Pop();
  Assert.IsTrue(sut.IsEmpty());
}
```

[16] Atrybuty dodane do metod testujących pozwalają na umieszczenie dodatkowego „opisu" do metody/klasy. Będzie on podczas runtime'u uwzględniony przez framwork NUnit. Poza atrybutem `[SetUp]` wykorzystany może również zostać atrybut `[TearDown]`, definiuje on metodę, która zostanie wywołana po wykonaniu się każdego z testów. Dodatkowo wykorzystane mogą być atrybuty `[OneTimeSetUp]` oraz `[OneTimeTearDown]`, wywołane zostaną one jednorazowo przed i po wykonaniu się wszystkich testów.

Aby powyższy test został wykonany pomyślnie można zaproponować następująca „wersję” metody `Pop()`:

```
public int Pop()
{
  if(this.IsEmpty()) { throw new InvalidOperationException(); }
  this.isEmpty = true
  return -1;
}
```

Oczywiście powyższe rozwiązanie, już na pierwszy rzut oka, jest złe. Napisz teraz analogicznie test, pokazujący błędną implementację np. `ShouldNotBeEmptyWhenTwoPushedOnePopped()`. Jeszcze raz musi on zakończyć się niepowodzeniem. W klasie `CustomStack` dodaj pole `counter`, które przejmie role `isEmpty`. Przy umieszczaniu elementu na stosie inkrementuj ten licznik. Natomiast przy zdejmowaniu dekrementuj go. Doprowadź klasę `CustomStack` do takiego stanu, aby przechodziła napisany wcześniej test.

Kolejny test niech sprawdza czy po umieszczaniu elementu na stosie i zdjęciu go zwracana jest ta sama wartość:

```
[TestCase(-10)]
[TestCase(0)]
[TestCase(10)]
public void ShouldReturnPoppedElementWhenPoppedPushed(int element)
{
  sut.Push(element);
  Assert.AreEqual(element, sut.Pop());
}
```

Zwróć uwagę, że został wykorzystany inny atrybut dla metody testującej. Użycie `[TestCase]` pozwala na automatyczne wykonanie testu z różnymi argumentami, w tym przypadku zostaną wykonane trzy testy z wartościami odpowiednio: `-10,0,10`. Dodaj do klasy `CustomStack` prywatne pole `element`, przypisuj mu wartość przekazywanego argumentu w metodzie `Push()`. W metodzie `Pop()` zwracaj wartość tego pola. Uruchom testy, powinny zakończyć się pomyślnie.

Na koniec dodaj ostatni test `ShouldReturnPoppedElementWhenTwoPoppedTwoPushed`. I zmień implementację klasy `CustomStack` tak aby test zakończył się powodzeniem. Konieczne będzie dodanie tablicy typu `int[]`.

5.2 Moq

Moq jest frameworkiem do tworzenia mockerów, czyli obiektów które wstrzyknięte do testowanych klas udają wcześniej skonfigurowane „zachowania”. Framework jest darmowy, dystrybuownany na licencji BSD 3-Clause. Podobnie jak NUnit jest prosty i pomaga znacznie uprościć proces testowania.

Wykorzystując framework Moq można tworzyć mockery, które będą implementowały składowe definiowane w interfejsie albo będą przesłaniać metody wirtualne lub abstrakcyjne. Załóżmy, że posiadamy następujący interfejs:

```
public interface IFoo
{
  Bar Bar { get; set; }
  string Name { get; set; }
  int Value { get; set; }
  bool DoSomething(string value);
  bool DoSomething(int number, string value);
  Task<bool> DoSomethingAsync();
  string DoSomethingStringy(string value);
  bool TryParse(string value, out string outputValue);
  bool Submit(ref Bar bar);
  int GetCount();
  bool Add(int value);
}
```

Utworzenie generycznej klasy mocker'a odbywa się w następujący sposób:

```
var mock = new Mock<IFoo>();
```

Generyczny parametr `T` (w tym przypadku `IFoo`) jest typem, na podstawie którego chcemy utworzyć obiekt mocker'a. Zazwyczaj tworzenie tych obiektów odbywa się w konstruktorze klasy testującej albo w metodzie posiadającej atrybut `[SetUp]`.

Następnie taki mocker może zostać „wstrzyknięty" do testowanej klasy np. przez konstruktor. Zakładamy, że testowana klasa korzysta z obiektu implementującego `IFoo`, który jest do niej przekazywany przez technikę wstrzykiwania zależności.

Dalej w metodach testujących, metodzie `Setup` albo klasie bazowej (jeśli klasa z testami dziedziczy po jakimś innym typie) należy odpowiednio zdefiniować zachowanie mocker'a. Np. jeśli chcemy, aby obiekt po wywołaniu przez badaną klasę metody `GetCount()` zwrócił wartość `0` należy go skonfigurować w sposób pokazany poniżej:

```
mock.Setup(x => x.GetCount()).Returns(() => 0);
```

W przypadku chęci określenia zwracanej wartości jedynie dla konkretnego argumentu należy ten argument podać podczas konfigurowania mockera:

```
mock.Setup(x => x.DoSomething("abc")).Returns(() => true);
```

W tym przypadku jeśli testowana klasa, wywoła metodę `DoSomething` z argumentem `ąbc"`, zwrócona zostanie wartość `true`.

Można również określić, zachowanie mocker'a dla dowolnego argumentu danego typu albo można dodatkowo podać wyrażenie lambda, które stanie się warunkiem zwrócenia danej wartości

```
mock.Setup(x => x.DoSomething(It.IsAny<string>())).Returns(true);
mock.Setup(x => x.DoSomething(It.Is<string>(y => y.StartsWith("S")))).
    Returns(() => true);
```

Istnieje także możliwość określenia przedziału wartości w jakim powinien znaleźć się przekazywany argument:

```
mock.Setup(x => x.Add(It.IsInRange<int>(0, 10, Range.Inclusive))).Returns
    (() => true);
```

Jeśli zależy nam, aby wywołana metoda zgłosiła wyjątek, należy użyć metody `Throws`:

```
mock.Setup(x => x.DoSomething(string.Empty)).Throws<
    InvalidOperationException>();
```

W analogiczny sposób jak w przypadku metod można konfigurować właściwości, albo użyć metody `SetupProperty`, która dodatkowo będzie śledzić liczbę wywołań akcesorów `get` oraz `set`:

```
mock.Setup(x => x.Name).Returns("bar");
// or
mock.SetupProperty(x => x.Name, "foo");

mock.Verify(x => x.Name, Times.Once());
```

Co więcej, wykorzystując framework Moq można także ustawić zdarzenia generowane przez klasę mocker'a:

```
mock.Setup(x => x.Submit()).Raises(x => x.Sent += null, EventArgs.Empty);
mock.Raise(x => x.FooEvent += null, new FooEventArgs(fooValue));

// Raise passing the custom arguments expected by the event delegate
mock.Raise(x => x.MyEvent += null, 25, true);
```

Jeśli chcemy przechować argumenty, które testowana klasa przekazuje do obiektu mocker'a albo policzyć liczbę wywołań danej metody mocker'a możemy skorzystać z metody `Callback`:

```
var calls = 0;
var callArgs = new List<string>();

// alternate equivalent generic method syntax
mock.Setup(x => x.DoSomething(It.IsAny<string>()))
  .Callback<string>(s => callArgs.Add(s))
```

```
  .Returns(() => true);

// access arguments for methods with multiple parameters
mock.Setup(x => x.DoSomething(It.IsAny<int>(), It.IsAny<string>()))
  .Callback<int, string>((i, s) => callArgs.Add(s))
  .Returns(() => true);

mock.Setup(foo => foo.DoSomething("abc"))
  .Callback(() => calls++)
  .Returns(() => true);
```

Sprawdzenie liczby wywołań danej metody można również sprawdzić przy użyciu metody `Verify`:

```
mock.Verify(x => x.DoSomething(It.IsAny<int>(), It.IsAny<string>()), Times
    .Exacly(4));
//or
mock.Verify(x => x.DoSomething(It.IsAny<int>(), It.IsAny<string>()), Times
    .Once());
```

5.2.1 Zadanie 2

Z udostępnionego repozytorium pobierz materiały na zajęcia laboratoryjne. W folderze `Lab5` powinny znajdować się projekty `Core`, `Infrastructure` oraz `Web`. Dwa pierwsza są zestawami bibliotek ostatni natomiast projektem ASP.NET Core, który zawiera szablon prostej strony www.

W projekcie `Core` znajduje się folder `Aggregates`, a w nim klasa `Project`. Posiada ona kilka właściwości oraz przechowuje referencję do listy obiektów typu `ToDoItem`. Sprawdź jakie składowe zawierają te klasy. W tym folderze został również dodany folder `Interfaces`, w którym umieszczono interfejs `IRepository<>`. Implementacja tego interfejsu (zdefiniowana w innej warstwie/projekcie) będzie odpowiedzialna za komunikację ze źródłem danych np. bazą danych.

Project `Core` jedynie zawierał interfejs `IRepository<>`, natomiast jego implementacja znajduje się w projekcie `Infrastructure`. W celu uproszczenia samego przykładu implementacja zawiera jedynie na sztywno zwracany obiekt agregatu `Project`.

Ostatnim zestawem znajdującym się w folderze jest `Web`. Jest to aplikacja webową ASP.NET Core. Posiada ona widoki `Views`, będące plikami `*.cshtml`. Zawierają one znaczniki Razor'a[17] i służą do generowania plików HTML przekazywanych do przeglądarki internetowej. Dodatkowo w projekcie umieszczony został folder `Controllers`, który zawiera klasy kontrolerów. Żądania są przekazywane do kontrolerów, które komunikują się z innymi warstwami (np. z warstwą infrastruktury) i zwracają widoki, które następnie są wyświetlane w oknie przeglądarki. Jeśli widok wymaga jakiś danych do uzupełnienia strony, należy je przekazać w formie modelu. Przykłady modeli znajdują się w folderze `Models`.

Zwróć uwagę, że kontroler `ProjectController` zwraca widok, do którego przekazuje obiekt DTO[18]. Z repozytorium pobierana jest encja projektu, która następnie musi zostać zmapowana do obiektu DTO. Dodatkowo w kontrolerze został dodany warunek w którym sprawdzany jest obiekt zwrócony przez repozytorium. Jeśli zmienna posiada wartość `null` to następuje wywołanie metody `NotFound` i zwrócenie obiektu `NotFoundObjectResult`.

Tak przygotowane projekty posłużą do pokazania zasady działania mocker'ów. Jeśli chcielibyśmy przetestować kod kontrolera, to konieczne okazuje się przekazanie do niego obiektu typu `IRepository<>`. Ponieważ testowana jest klasa kontrolera, nie jest dobrym pomysłem przekazywanie faktycznego obiektu repozytorium, który komunikowałby się z bazą danych. Właściwym podejściem jest wykorzystanie mocker'ów.

Umieść w folderze `Lab5` kolejny projekt testowy (nazwij go `WebTests`) analogicznie jak było to pokazane w zadaniu pierwszym5.1.1.

Dodaj do projektu `WebTests` za pomocą menadżera pakietów NuGet pakiet `Moq`. Aby to zrobić kliknij PPM na nazwę projektu i wybierz opcję "Manage NuGet Packages". Ustaw również zależność projektu

[17] Razor jest językiem znaczników który pozwala na umieszczenie (po stronie serwera) kodu C# na stronach webowych. W momencie żądania strony przez przeglądarkę, serwer wykonuje kod znajdujący się na stronie zanim zwróci tę stronę do przeglądarki.

[18] DTO jest obiektem, który zawiera informacje do przekazanie pomiędzy warstwami aplikacji. Pozwala to na ukrycie pewnych informacji, które są zawarte w encjach np. w wewnętrznych warstwach, a nie chcemy ich przekazywać dalej do warstw zewnętrznych. Dodatkowo usunięcie pewnych informacji może być korzystne ponieważ pozwala na zmniejszenie rozmiaru danego obiektu i umożliwia luźne powiązanie warstw np. warstwy serwisów oraz infrastruktury.

z testami od zestawów Core, Infrastructure oraz Web, klikając PPM na nazwę projektu i następnie Add->Project References..., w zakładce Projects/Solution zaznacz odpowiednie projekty.

Po wykonaniu tych czynności zmień nazwę klasy w projekcie z `UnitTest1` na `ProjectControllerTests`. Aby to zrobić użyj narzędzia do refaktoryzacji poprzez ustawienie kursora na nazwie klasy i użycie skrótu klawiszowego Ctrl+R+R. Zmień również nazwę samego pliku z poziomu eksplorera solucji.

Dalej dodaj do klasy z testami dyrektywę `using Moq`, aby móc skorzystać z przed chwilą dodanego do projektu pakietu. Następnie w klasie `ProjectControllerTests`, dodaj prywatne pola: `ProjectController sut` oraz `Mock<IRepository<Project» projectRepositoryMock = new();`.

Następnie w metodzie `Setup()` (z atrybutem `[SetUp]`) przypisz do pola `sut` instancję `ProjectController` przekazując jako argument konstruktora obiekt mocker'a:

```
[SetUp]
public void Setup()
{
  projectRepositoryMock.Reset();
  this.sut = new ProjectController(projectRepositoryMock.Object);
}
```

Dodatkowo w tym miejscu można zresetować obiekt mocker'a jeśli jest to konieczne.

Zamiast przekazywać do kontrolera faktyczną implementację repozytorium, przekazywany jest obiekt pobrany za pomocą właściwości `Object` mocker'a. Od teraz w metodach testowych, będzie możliwe zmieniane zachowań mocker'a tak, aby sprawdzić różne zachowania klasy `ProjectController`.

W pierwszej kolejności napisz metodę testującą, która sprawdzi czy metoda `Index`, wywoła metodę `NotFound()` i zwróci obiekt klasy `NotFoundResults`. Aby to zrobić dodaj metodę testującą i skonfiguruj w niej obiekt mocker'a w następujący sposób:

```
[Test]
public void ShouldReturnNotFoundIfRepositoryReturnNull()
{
  //Arrange
  projectRepositoryMock.Setup(x => x.GetById(It.IsAny<int>()))
    .ReturnsAsync(() => null);

  //Act
  var ret = this.sut.Index().Result as NotFoundResult;

  //Assert
  Assert.AreEqual(404, ret.StatusCode);
}
```

Ponieważ kontroler jest metodą asynchroniczną konieczne jest skonfigurowanie mocker'a tak, aby również był asynchroniczny za pomocą `.ReturnsAsync(() => null)`. Jeśli testowana metoda byłaby nieasynchroniczna wystarczyłoby napisać `.Returns(() => null)`.

Dodaj do klasy `ProjectControllerTests` kolejną metodę. Zanim skonfigurujesz mocker'a w analogiczny jak poprzednio sposób, utwórz obiekt klasy `Project`. Dodaj do niego kilka obiektów `ToDoItem` np. tak jak pokazano poniżej:

```
[Test]
public void ShouldReturnViewIfRepositoryReturnsValidProject()
{
  //Arrange
  Project project = new Project("Foo") { Id = 1 };
  project.AddItem(new ToDoItem() {Id = 1, Title = "Bar", Description = "
   Empty" });

  projectRepositoryMock.Setup(x => x.GetById(It.IsAny<int>()))
  .ReturnsAsync(() => project);

  //...
}
```

Zwrócić uwagę, że mocker został skonfigurowany tak aby, po wywołaniu metody `GetById` zwrócić referencję do obiektu wskazywanego przez zmienną `project`.

Dalej wywołaj metodę kontrolera i sprawdź czy zwracany widok posiada odpowiednio zmapowany model:

```
[Test]
public void ShouldReturnViewIfRepositoryReturnsValidProject()
{
  //...

  //Act
  var retView = this.sut.Index().Result as ViewResult;
  var retModel = (retView.Model as ProjectModel);

  //Assert
  Assert.AreEqual(1, retModel.Id);
  //...
}
```

Dodaj inne asserty tak, aby sprawdzić poprawność zwracanego widoku.

W przypadku, gdy chcielibyśmy sprawdzić jakie argumenty zostały przekazane do metody mocker'a konieczne jest skonfigurowanie tzw. „callback'ów". Podczas ich ustawiania istnieje możliwość, aby przekazywany argument przypisać do zmiennej albo dodać przekazywane argumenty do listy w celu ich późniejszej weryfikacji. Aby to sprawdzić dodaj kolejną metodę testującą:

```
[Test]
public void ShouldCallRepositoryWithProjectIdOneIfArgumentNotSpecified()
{
  //Arrange
  int capturedId = -1;
  projectRepositoryMock.Setup(x => x.GetById(It.IsAny<int>()))
    .Callback((int id) => capturedId = id);

  //Act
  _ = this.sut.Index();

  //Assert
  Assert.AreEqual(1, capturedId);
}
```

Jeśli chcielibyśmy sprawdzić czy metoda mocker'a została wywołana, można posłużyć się metodą `Verify`:

```
[Test]
public void ShouldReturnNotFoundWhenRepositoryReturnNull()
{
  //...

  //Assert
  projectRepositoryMock.Verify(x => x.GetById(It.IsAny<int>())),  Times.
   Once());
}
```

Analogicznie można sprawdzić czy metoda nie została wywołana albo została wywołana określoną liczbę razy, zmieniając `Times.Once()`, na `Times.Never()` albo `Times.Exactly()`.

Aby przetestować różnicę pomiędzy statyczną metodą `It.Any<int>()`, a `It.Is<int>(Predicate<T>)`, gdzie ta druga pozwala na dodanie warunku dla przekazywanego argumentu zmień w pokazany poniże sposób kod metody `ShouldReturnNotFoundIfRepositoryReturnNull`:

```
[Test]
public void ShouldCallRepositoryExactlyOnce()
{
  //Act
  _ = this.sut.Index();
```

```
6
7     //Assert
8     projectRepositoryMock.Verify(x => x.GetById(It.Is<int>(x => x == 1)),
       Times.Once());
9 }
```

Metoda `Verify` sprawdzi czy metoda `GetById` została wywołana z argumentem jeden tylko raz. W analogiczny sposób można ustawiać warunki dla zwracanych wartości. Np. zwrócenie jakieś wartości tylko jeśli przekazywany argument spełnia dany warunek.

6 Refaktoryzacja

Refaktoryzacja jest procesem zmiany **działającego** kodu, bez wpływania na jego zachowanie (kod przed i po refaktoryzacji powinien działać tak samo). Poprawie ulegają jego wewnętrzne struktury. Podczas procesu refaktoryzacji kod powinien być cały czas sprawny.

Refaktoryzacja powinna być rzeczą naturalną podczas rozwijania oprogramowania. Pozwala ona na zwiększenie czytelności kodu. Czytelny, łatwy do zrozumienia kod jest ważny zarówno dla kolegów z zespołu jak i dla nas samych wracających po pewnym czasie to kodu nad którym wcześniej pracowaliśmy. Refaktoryzacja polepsza projekt, ułatwia jego modyfikowanie i rozszerzanie. Dodatkowo sprawia, że znalezienie błędów staje się prostsze, a samo programowanie szybsze i sprawniejsze.

6.1 „Zapachy" w kodzie

W kontekście refaktoryzacji często pojawia się termin „zapach" kodu. „Zapaszki" sugerują miejsca, gdzie w naszym kodzie może dziać się coś złego. Wskazują na fragmenty, które potencjalnie powinny zostać podane refaktoryzacji. W książce[3] Martin Fowler wyróżnia następujące „zapachy":

- Tajemnicza nazwa - nazwy metod, klas, pól powinny jasno określać za co dana składowa jest odpowiedzialna. Dzięki narzędziom refaktoryzacyjnym dostępnym w środowiskach IDE, zmiana nazw jest bardzo prosta i warto z niej często korzystać.

- Duplikowany kod - jeśli fragment kodu pojawia się w wielu miejscach jego utrzymanie staje się trudne. Konieczne jest pamiętanie o wprowadzaniu zmian w kilku miejscach, niezbędne jest również czytanie tych samych fragmentów kodu w różnych plikach.

- Długa funkcja - im dłuższa jest metoda tym jej zrozumienie staje się trudniejsze. Warto dekomponować długie metody na kilka mniejszych stosując ekstrakcję funkcji.

- Długa lista parametrów - jeśli funkcja ma wiele parametrów trudno jest z niej korzystać. Mimo, że współczesne IDE ułatwiają proces wywoływania metod przez wyświetlanie jej parametrów, to funkcje z mniejszą ich liczbą są czytelniejsze. Zmniejszyć listę parametrów można np. przez zebranie funkcji w klasę i stosowanie pól klasy, zamiast parametrów.

- Dane globalne - stosowania zmiennych globalnych czy klas (singletonów) należy unikać, nie wiadomo kiedy i gdzie zostaną one zmienione. Błędy z nimi związane są bardzo ciężkie do debugowania.

- Dane mutowalne - problem z danymi zmiennymi jest podobny do tego z danymi globalnymi. Tam gdzie to możliwe lepiej jest stosować klasy niezmienne. W C# od wersji 9.0 można korzystać z rekordów.

- Rozbieżne zmiany - powstają w skutek nieprzestrzegania zasady SRP1.1. „Zapach" ten pojawia się jeżeli jedna metoda/klasa jest modyfikowana z różnych powodów, zmian funkcjonalności. Np. zmiana bazy danych czy rozliczeń finansowych pociąga za zmiany kilku metod danej klasy.

- Fala uderzeniowa - jest odwrotnością rozbieżnych zmian. Problem powstaje jeżeli jedna konkretna zmiana w programie pociąga za sobą konieczność wielu innych zmian, w innych fragmentach kodu. Jest ona podobna do rozbieżnych zmian, które odnoszą się do wielu zmian z pojedynczej klasie, natomiast fala uderzeniowa odnosi się do sytuacji kiedy, jedna zmiana generuje zmiany w wielu innych miejscach.

- Zazdrosne funkcjonalności - problem pojawia się, gdy dana metoda odwołuje się w większości do metod z innej klasy. Może to oznaczać, że powinna ona być częścią tej klasy. W takiej sytuacji należy przenieść tę funkcję w miejsce, do którego bardzie pasuje.

- Stada danych - „zapach" powstaje jeśli pola często występują blisko siebie. Pogrupować je można poprzez ekstrakcję do nowej klasy.

- Opętanie typami prostymi - często warto zamiast stosować proste typu: `int`, `double`, `bool` itd. zamienić je własnymi typami. Np. nr telefonu lepiej jest umieścić w osobnej klasie, zamiast stosować łańcuchy znaków.

- Powtarzane instrukcje warunkowe - mogą zostać zastąpione polimorfizmem. Klauzule `if-else` oraz `switch-case` często można całkowicie usunąć. W przypadku wielu takich instrukcji warunkowych, może to znacząco poprawić czytelność kodu.

- Pętle - zamiast je stosować warto rozważyć użycie potoków. W przypadku C# dobrym rozwiązaniem jest wykorzystanie wyrażeń LINQ.
- Leniwa klasa - „zapach" występuje jeśli klasa nie wykonuje nic ponad delegowanie zadań do innych klas. Rola tej klasy mogła się zmniejszyć np. w wyniku wcześniejszych refaktoryzacji.
- Spekulacyjne uogólnienia - pojawia się jeśli kod jest nadmiarowy, zaprojektowany na przypadki, które być może kiedyś wystąpią, tak na wszelki wypadek.
- Pole tymczasowe - przypadek, w którym pewna wartość jest przypisywana do zmiennej tylko w pewnych sytuacjach. Osoba czytająca kod zwykle oczekuje, że obiekt potrzebuje wszystkich swoich zmiennych.
- Łańcuchy komunikatów - pojawia się jeśli klient żąda pewnego obiektu, natomiast ten obiekt żąda innego obiektu, a ten inny obiekt jeszcze innego. Każda zmiana struktury nawigacji wymaga zmian w kodzie klienta.
- Pośrednik - występuje jeśli dana klasa jedyne co robi to deleguje do innych klas. Zamiast korzystać z tej klasy, często lepiej jest odwołać się bezpośrednio do obiektu, który wie co robić.
- Niestosowna bliskość - ma miejsce jeśli dwie klasy wymieniają ze sobą dużą liczbę informacji. Np. jeśli jedna klasa korzysta z klasy danych to być może część tych operacji mogłaby zostać do niej przeniesiona.
- Duża klasa - „zapach" pojawia się jeśli klasa posiada wiele pól, metod czy po prostu linii kodu. Często taka klasa powinna zostać podzielona na kilka mniejszych.
- Alternatywne klasy z różnymi interfejsami - występuje, gdy dwie klasy, które mogłyby być stosowane zamienne, posiadają różne interfejsy.
- Klasa danych - posiada jedynie pola i metody dostępowe do nich (gettery i settery). Nie posiadają one żadnego dodatkowego zachowania, a klasy kliencie jedynie manipulują na danych w nich zawartych. Wyjątkiem od tej zasady jest klasa zwracająca rekord danych.
- Odmowa przyjęcia spadku - ma miejsce, gdy klasy pochodne, korzystają jedynie z części składowych klasy rodzica. Może to świadczyć o źle zaprojektowanej hierarchii klas. Nie zawsze ten „zapach" oznacza coś złego, często można spotkać się z sytuacją w której tworzy się klasę pochodną, aby wykorzystać tylko fragment klasy bazowej.

6.2 Zadanie 1

Z repozytorium pobierz materiały do zadania. W folderze `Lab6` znajdują się dwa projekty. Pierwszy z nich jest zestawem biblioteki .NET Framework, drugi natomiast zawiera projekt z testami NUnit. Podczas tego zadania będziemy przeprowadzać refaktoryzację klasy `Statement22`. Metoda `CreateStatement` wspomnianej klasy pobiera argumenty wejściowe i na ich podstawie przygotowuje raport. Konkretnie będzie to raport generowany w oparciu o dostępne spektakle teatralne oraz faktury.

```
internal static class Statement
{
  public static string CreateStatement(Invoice invoice, Dictionary<string,
    Play> plays)
  {
    var totalAmount = 0;
    var volumeCredits = 0;
    var results = $"Rachunek dla {invoice.Customer}";

    CultureInfo pl = new CultureInfo("pl-PL");

    foreach (var perf in invoice.Performances)
    {
      var play = plays[perf.PlayID];
      var thisAmount = 0;

      switch (play.Type)
      {
```

```
        case "tragedia":
        thisAmount = 40000;
        if (perf.Audience > 30)
        {
          thisAmount += 1000 * (perf.Audience - 30);
        }
        break;

        case "komedia":
        thisAmount = 30000;
        if (perf.Audience > 20)
        {
          thisAmount += 10000 + 500 * (perf.Audience - 20);
        }
        thisAmount += 300 * perf.Audience;
        break;

        default:
        throw new ArgumentException($"Nieznany typ przedstawienia: ${play.Type}");
      }

      // Award bonus points
      volumeCredits += Math.Max(perf.Audience - 30, 0);
      // Award an additional promotional point for every 5 viewers of the comedy
      if ("komedia" == play.Type)
      {
        volumeCredits += (int)Math.Floor((decimal)perf.Audience / 5);
      }

      // Create statement row
      results += $"{play.Name}: {(thisAmount / 100).ToString("c", pl)} (liczba miejsc: {perf.Audience}" + Environment.NewLine;
      totalAmount += thisAmount;
     }

     results += $"Naleznosc: {(totalAmount / 100).ToString("c", pl)}" + Environment.NewLine;
     results += $"Punkty promocyjne: {volumeCredits}";

     return results;
  }
}
```

Listing 22: Klasa statyczna `Statement`

W projekcie z folderze `DTOs` znajdują się również klasy wykorzystywane do deserializacji danych typu JSON z plików wejściowych. Konkretnie dla modeli spektakli (ang. plays) oraz faktur (ang. invoces).

```
public class Performance
{
  [JsonProperty("playID")]
  public string PlayID { get; set; }
  [JsonProperty("audience")]
  public int Audience { get; set; }
}
```

Listing 23: Performance.cs

```
public class Invoice
{
  [JsonProperty("customer")]
  public string Customer { get; set; }
  [JsonProperty("performance")]
  public List<Performance> Performance { get; set; }
}
```

Listing 24: Invoice.cs

oraz

```
public class Play
{
  [JsonProperty("name")]
  public string Name { get; set; }
  [JsonProperty("type")]
  public string Type { get; set; }
}
```

Listing 25: Play.cs

Ponieważ podczas refaktoryzacji zastanego kodu, z którym nie mieliśmy wcześniej doświadczenia istnieje duże ryzyko wprowadzenia nowego błędu, dobrze jest zacząć proces refaktoryzacji od napisania porządnego zestawu testów. W celu maksymalnego uproszczenia tego zadania, w projekcie z testami został dodany tylko jeden test korzystający z plików wejściowych oraz oczekiwanego wyniku, również w takiej formie - w rzeczywistości powinniśmy przygotować zdecydowanie więcej plików, dla różnych sytuacji. Lepiej byłoby przygotować pełne testy refaktoryzowanej klasy, bez korzystania z plików wejściowych, ale pokazane podejście pozwoli na sprawniejsze przeprowadzenie ćwiczenia. W folderze `GoldenFiles` zostały umieszczone pliki JSON zawierające listę spektakli oraz przykładowe „dane do" faktury.

```
{
  "hamlet": {
    "Name": "Hamlet",
    "Type": "tragedia"
  },
  "as-like": {
    "Name": "Jak wam sie podoba",
    "Type": "komedia"
  },
  "othello": {
    "Name": "Otello",
    "Type": "tragedia"
  }
}
```

Listing 26: plays.json

oraz

```
{
  "customer": "BigCo",
  "performance": [
  {
    "playID": "hamlet",
    "audience": 55
  },
  {
    "playID": "as-like",
    "audience": 35
  },
  {
    "playID": "othello",
```

```
    "audience": 40
  }
  ]
}
```

Listing 27: invoices.json

W jedynej klasie w projekcie z testami zwróć uwagę na metodę testującą. W pierwszej kolejności następuje deserializacja plików JSON na obiekty klas `Plays` oraz `Invoice`. Do deserializacji został wykorzystany pakiet `Newtonsoft.Json`. Dalej do zmiennej `expected` przypisywany jest obiekt typu `string`, który zawiera oczekiwany wynik działania testowanej metody. Następnie jest wywoływana wspomniana metoda oraz porównywane są wyniki. Uruchom testy, powinny zakończyć się one wynikiem pozytywnym.

Pora teraz na przyjrzenie się klasie, którą należy w tym zadaniu poprawić. Na pewno klasa łamie zasadę pojedynczej odpowiedzialności1.1 oraz jednocześnie jest dość długa, co pogarsza jej czytelność. Metoda zarówno przygotowuje raport, jak jest odpowiedzialna za naliczania rabatów czy liczenie kosztów całych przedstawień. Sama w sobie nie jest źle napisana, ale na pewno można w nie sporo zmienić.

W pierwszym kroku dokonaj ekstrakcji wyrażenia `switch` zaznaczając całe wyrażenie i wciskając `Alt+Enter` oraz wybierając opcję `Extract method`, nazwij nową metodę `AmountFor`. Zamiast tworzyć zmienną `thisAmount` i przypisywać jej najpierw wartość zero, przypisz jej od razu wartość zwracaną przez `AmountFor`. Zmień za pomocą `Ctrl+R+R` nazwę zmiennej `thisAmount` wewnątrz utworzonej przed chwilą metody `AmountFor` na nazwę lepiej ją opisującą, informującą że jest to wartość zwracana przez tę funkcję np. `result`. Nazwa parametru `perf` również nie jest najszczęśliwsza, analogicznie zmień ją na bardziej opisową. Uruchom testy, aby sprawdzić czy nie zostało nic zepsute w między czasie.

Wewnątrz klasy `Statement` dodaj prywatną metodę `GetPlay`:

```
Play GetPlay(Dictionary<string, Play> plays, Performance perf) => plays[
    perf.PlayID];
```

Listing 28: Dodana metoda `GetPlay`

Następnie usuń zmienną `play` wykonując „Zastąpienie Zmiennej Tymczasowej Zapytaniem”. Zmień wszystkie wykorzystania zmiennej `play` na wywołanie metody `GetPlay(plays, perf)` wykonując „Wchłonięcie Zmiennej”. Usuń zmienną `play` z **wywołania** funkcji `AmountFor`, zamień ją odpytaniem metody `GetPlay()`. Uruchom testy jednostkowe i sprawdź czy wykonują się one poprawnie.

Refaktoryzację „Wchłonięcie zmiennej” wykonaj analogicznie dla `thisAmount`, zamień ją wywołaniem prywatnej metody `AmountFor`. Usuń zmienną `thisAmount`.

Wyodrębnij fragment kodu odpowiedzialny za liczenie rabatu do osobnej metody:

```
private static int VolumeCreditsFor(Dictionary<string, Play> plays,
    Performance perf)
{
  // Award bonus points
  var volumeCredits = Math.Max(perf.Audience - 30, 0);
  // Award an additional promotional point for every 5 viewers of the
    comedy
  if ("komedia" == GetPlay(plays, perf).Type)
  {
    volumeCredits += (int)Math.Floor((decimal)perf.Audience / 5);
  }

  return volumeCredits;
}
```

Dokonaj zmian w pętli `foreach` tak, aby do zmiennej `volumeCredits` była dodawana wartość zwracana przez funkcję `VolumeCreditsFor`. Zmień nazwy zmiennych wewnątrz funkcji `VolumeCreditsFor` na bardziej opisowe.

Analogicznie wyodrębnij funkcję formatującą waluty. Utwórz folder `Extensions` i dodaj do niego klasę statyczną `CurrencyExtensions`. Następnie dodaj do tej klasy metodę formatującą ciąg znaków. Możesz skorzystać z metody rozszerzającej, która została pokazana w rozdziale 3.3.

```
namespace RefactorExample.Extensions
{
```

```
  internal static class CurrencyExtensions
  {
    internal static string ToPolishZloty(this int value) => (value / 100).
   ToString("c", new CultureInfo("pl-PL"));
  }
}
```

Zmień kod w miejscach, gdzie były formatowane waluty, korzystając z metody rozszerzającej. Usuń zmienną lokalną `pl` typu `CultureInfo`.

Teraz, aby wchłonąć zmienną `volumeCredits` zastosuj „Podział pętli" - rozdziel jedną pętle na dwie odpowiedzialne odpowiednio za wyliczanie ceny przedstawienia oraz punktów rabatowych. Możesz również zmienić nazwę zmiennej w pętli `foreach`.

```
static string CreateStatement(Invoice invoice, Dictionary<string, Play>
   plays)
{
  //...

  foreach (var performance in invoice.Performances)
  {
    volumeCredits += VolumeCreditsFor(plays, performance);
  }

  foreach (var performance in invoice.Performances)
  {
    // Create statement row
    results += $"{GetPlay(plays, performance).Name}: {AmountFor(
   performance, GetPlay(plays, performance)).ToPolishZloty()} (liczba
   miejsc: {performance.Audience}" + Environment.NewLine;
    totalAmount += AmountFor(performance, GetPlay(plays, performance));
  }

  //...
}
```

Następnie przenieś pętle w której wykonywane jest liczenie rabatu do osobnej metody prywatnej o nazwie `TotalVolumeCredits` wraz ze zmienną lokalną `volumeCredits`. Teraz usuń z funkcji `CreateStatement` zmienną `volumeCredits` i w miejscu jej użycia umieść wywołanie utworzonej przed chwilą metody.

Wykonaj analogiczną refaktoryzację ze zmienną `totalAmount` - nowo tworzoną metodę nazwij `TotalAmount`. Wcześniej zastosuj ponownie podział pętli na jedną aktualizującą `results` oraz tę aktualizującą `totalAmount`.

```
public static string CreateStatement(Invoice invoice, Dictionary<string,
   Play> plays)
{
  var results = $"Rachunek dla {invoice.Customer}";

  foreach (var performance in invoice.Performances)
  {
    // Create statement row
    results += $"{GetPlay(plays, performance).Name}: {AmountFor(
   performance, GetPlay(plays, performance)).ToPolishZloty()} (liczba
   miejsc: {performance.Audience}" + Environment.NewLine;
  }

  results += $"Naleznosc: {TotalAmount(invoice, plays).ToPolishZloty()}" +
    Environment.NewLine;
  results += $"Punkty promocyjne: {TotalVolumeCredits(invoice, plays)}";

  return results;
}
```

```
16
17 private static int TotalAmount(Invoice invoice, Dictionary<string, Play>
       plays)
18 {
19   var totalAmount = 0;
20
21   foreach (var performance in invoice.Performances)
22   {
23     totalAmount += AmountFor(performance, GetPlay(plays, performance));
24   }
25
26   return totalAmount;
27 }
```

Wykonując powyższe kroki udało się poprawić czytelność kodu, ale jego funkcjonalność może zostać ulepszona. Jeśli chcielibyśmy np. zmienić funkcję, aby formatowała oświadczenie do formatu HTML, konieczne byłoby przekopiowanie dużej części kodu. Spróbujmy rozwiązać ten problem.

Przenieś cały kod metody `CreateStatement` do osobnej metody (wykonaj „Ekstrakcję Funkcji") o nazwie `RenderPlainText`. Teraz utwórz w projekcie folder `Statements`, a nim klasę `StatementData`, która będzie służyła za pośrednika danych między metodami `CreateStatement` oraz `RenderPlainText`. Ta refaktoryzacja nazywa jest „Wprowadzenie Obiektu Parametrycznego". W metodzie `CreateStatement` utwórz obiekt typu `StatementData` i przekaż go do wnętrza metody `RenderPlainText`:

```
1 static string CreateStatement(Invoice invoice, Dictionary<string, Play>
      plays)
2 {
3   StatementData statementData = new StatementData();
4
5   return RenderPlainText(statementData, invoice, plays);
6 }
7
8 private static string RenderPlainText(StatementData data, Invoice invoice,
      Dictionary<string, Play> plays)
9 {
10   var results = $"Rachunek dla {invoice.Customer}";
11
12   //...
13 }
```

Do klasy `StatementData` dodaj publiczne właściwości `List<Performance> Performances` oraz `string Customer`. Możesz, też dodać prywatne pole, a kolekcję obiektów `Performance` zwracać jako kolekcję tylko do odczytu, tak jak pokazano to na 6.2.Teraz w funkcji `CreateStatement` zmień wywołanie konstruktora i przekaż do niego odpowiednie wartości z właściwości obiektu `invoice`. Wszystkie użycia `invoice` w metodzie `RenderPlainText` zamień użyciem nowego obiektu pośredniczącego. Z sygnatury metody `RenderPlainText` usuń parametr typu `Invoice`, od teraz wszystkie potrzebne dane będą przekazywane za pomocą obiektu pośredniego.

```
1 private class StatementData
2 {
3   public string Customer { get; }
4
5   private List<Performance> performances;
6
7   public StatementData(string customer, List<Performance> performances)
8   {
9     this.Customer = customer;
10    this.performances = performances;
11  }
12
13  public IEnumerable<Performance> Performances => this.performances.
      AsReadOnly();
```

```
}

static string CreateStatement(Invoice invoice, Dictionary<string, Play>
    plays)
{
  StatementData data = new StatementData(invoice.Customer, invoice.
    Performances);

  return RenderPlainText(data, plays);

}

private static string RenderPlainText(StatementData data, Dictionary<
    string, Play> plays)
{
  var results = $"Rachunek dla {data.Customer}";

  //...
}
```

Konieczna okaże się również zmiana sygnatury funkcji `TotalVolumeCredtis` oraz `TotalAmount`. Zamiast przyjmować argument typu `Invoice` niech przyjmują argumenty typu `IEnumerable<Performance>`.

Klasa `Performance` służyła do deserializacji pliku JSON. Wydaje się dobrym pomysłem, aby dodać więcej informacji do klasy opisującej przedstawienia np. obiekt typu `Play`, cenę za przedstawienie oraz punkty rabatowe. Dodaj do projektu folder `Entities`, a w nim umieść klasę `EnrichedPerformance` w osobnym pliku:

```
namespace RefactorExample.Entities
{
  public class EnrichedPerformance
  {
    public EnrichedPerformance(string playId, int audience, int amount,
    Play play, int volumeCredits)
    {
      PlayId = playId;
      Audience = audience;
      Amount = amount;
      Play = play;
      VolumeCredits = volumeCredits;
    }

    public string PlayId { get; }
    public int Audience { get; }
    public int Amount { get; }
    public Play Play { get; }
    public int VolumeCredits { get; }
  }
}
```

Zanim wykonamy następne kroki, najwyższa pora usunąć słowo kluczowe `static` znajdujące się przed klasą `Statement`. Po jego usunięciu, dodaj do klasy konstruktor przyjmujący dwa argumenty: jeden typu `Invoice` drugi natomiast typu `Dictionary<string, Play> plays`. Oba przekazywane argumenty przypisz do prywatnych pól:

```
internal class Statement
{
  private readonly Invoice invoice;
  private readonly Dictionary<string, Play> plays;

  public Statement(Invoice invoice, Dictionary<string, Play> plays)
```

```
  {
    this.invoice = invoice;
    this.plays = plays;
  }

  //...
}
```

Teraz usuń słowo kluczowe `static` z sygnatury metody `CreateStatement` oraz usuń oba jej parametry. Od tego momentu można będzie posługiwać się obiektami przekazywanymi jako argumenty konstruktora. Konieczna okaże się również zmiana w projekcie `RefactorExample.Tests`. Zmień ciało metody zawierającej testy:

```
[Test]
public void ShouldGenerateStatementIfValidInput()
{
  //Arrange
  var inputPlays = JsonConvert.DeserializeObject<Dictionary<string, Play
   >>(File.ReadAllText(Path.Combine(baseDirectory, "GoldenFiles\\
   inputPlays.json")));
  var inputInvoice = JsonConvert.DeserializeObject<Invoice>(File.
   ReadAllText(Path.Combine(baseDirectory, "GoldenFiles\\inputInvoices.
   json")));
  var _sut = new Statement(inputInvoice, inputPlays);

  var expected = File.ReadAllText(Path.Combine(baseDirectory, "GoldenFiles
   \\expectedOutput.txt"));

  //Act
  var actual = _sut.CreateStatement();

  //Assert
  Assert.AreEqual(expected, actual);
}
```

Uruchom ponownie testy, powinny się one zakończyć „na zielono".

Następnie dodaj do klasy `Statement` metodę, która będzie odpowiedzialna za mapowanie obiektów `Performance` na `EnrichedPerformance`:

```
public EnrichedPerformance EnrichPerformance(Performance performance)
{
  return new EnrichedPerformance(
    performance.PlayID,
    performance.Audience,
    AmountFor(performance, GetPlay(plays, performance)),
    GetPlay(plays, performance),
    VolumeCreditsFor(plays, performance));
}
```

Zamień typ przechowywany w liście `Performances` klasy `StatementData` na `EnrichedPerformance`. Podczas tworzenia obiektu typy `Statement` w metodzie `StatementData` zastosuj napisaną przed chwilą metodę w celu zmapowania obiektu `Performance` na `EnrichedPerformance`:

```
public string CreateStatement()
{
  StatementData statementData = new StatementData(invoice.Customer,
    invoice.Performances.Select(x => EnrichPerformance(x)).ToList());

  return RenderPlainText(statementData, plays);
}
```

Zamień sygnatury metod: `TotalAmount` oraz `TotalVolumeCredtis` tak, aby przyjmowały obiekty typu `EnrichedPerformance`. W metodach `RenderPlainText`, `TotalAmount` oraz `TotalVolumeCredits` pojawiły się błędy. Popraw je wykorzystując właściwości `VolumeCredits` oraz `Amount` kolekcji `Performances` obiektu `StatementData`.

W metodzie `EnrichPerformance` dodaj przypisanie wartości kolejnej właściwości:

```
private EnrichedPerformance EnrichPerformance(Performance performance)
{
  return new EnrichedPerformance(
    performance.PlayID,
    performance.Audience,
    AmountFor(performance, GetPlay(plays, performance)),
    GetPlay(plays, performance),
    VolumeCreditsFor(plays, performance));
}
```

Zwróć uwagę, że z funkcji `TotalAmount` oraz `TotalVolumeCredtis` możesz usunąć parametr `Dictionary<string, Play> plays`. Popraw błędy w metodzie `RenderPlainText`.

Dalej usuń słowo kluczowe `static` ze wszystkich statycznych metod w klasie `Statement`. Usuń z ich sygnatur argument typu `Dictionary<string, Play> plays`. Ponownie popraw błędy, które się pojawiły i uruchom jeszcze raz testy.

Następnie wydaje się możliwe, aby przenieść metody `TotalAmount` oraz `TotalVolumeCredtis` do klasy `StatementData`, jednocześnie usuwając ich jedyny parametr `IEnumerable<EnrichedPerformance> performances`. Możesz również pętle w metodach `TotalAmount` oraz `TotalVolumeCredits` zamienić na potok:

```
private int TotalAmount(StatementData statementData)
{
  var totalAmount = 0;
    performances.ForEach(x => totalAmount += x.Amount);


  return totalAmount;
}
```

Ponownie popraw błędy, które się pojawiły i uruchom testy.

Do projektu dodaj folder `Render`, a w nim klasę `PlainTextRenderer`. Przenieś do niej metodę `RenderPlainText` z klasy `Statement`, zmień jej nazwę na `Render` oraz modyfikator dostępu z `private` na `public`. Metodę `CreateStatement` zmień w następujący sposób:

```
public string CreateStatement()
{
  StatementData statementData = new StatementData(invoice.Customer,
    invoice.Performances.Select(x => EnrichPerformance(x)).ToList());

  return new PlainTextRenderer().Render(statementData);
}
```

Możesz również utworzyć w folderze `Render` interfejs `IRenderer`, który będzie implementowany przez klasę `PlainTextRenderer`:

```
public interface IRenderer
{
  string Render(StatementData data);
}
```

Na sam koniec spróbujmy pozbyć się instrukcji `switch` z metod `AmountFor` oraz `VolumeCreditsFor`. Najpierw utwórz folder `Calculators`. W nim natomiast interfejs `IPerformanceCalculator`:

```
interface IPerformanceCalculator
{
  int AmountFor(int audience);
  int VolumeCreditsFor(int audience);
}
```

Dalej do folderu dodaj dwie klasy `ComedyCalculator` oraz `TragedyCalculator`. Obie niech implementują utworzony przed chwilą interfejs. Następnie przenieś kod z metod `AmountFor` oraz `VolumeCreditsFor` do odpowiednich klas:

```
class TragedyCalculator : IPerformanceCalculator
{
  public int AmountFor(int audience)
  {
    int result = 40000;
    if (audience > 30)
    {
      result += 1000 * (audience - 30);
    }

    return result;
  }

  public int VolumeCreditsFor(int audience)
  {
    return Math.Max(audience - 30, 0);
  }
}
```

oraz

```
class ComedyCalculator : IPerformanceCalculator
{
  public int AmountFor(int audience)
  {
    int result = 30000;
    if (audience > 20)
    {
      result += 10000 + 500 * (audience - 20);
    }
    result += 300 * audience;

    return result;
  }

  public int VolumeCreditsFor(int audience)
  {
    var volumeCredits = Math.Max(audience - 30, 0);
    volumeCredits += (int)Math.Floor((decimal)audience / 5);

    return volumeCredits;
  }
}
```

Dalej dodaj do folderu `Calculators` klasę fabryki z metodą wytwórczą w następujący sposób:

```
class FactoryCalculator
{
  public static IPerformanceCalculator Create(Play play)
  {
    switch(play.Type.ToLowerInvariant())
    {
      case "tragedia":
        return new TragedyCalculator();
      case "komedia":
        return new ComedyCalculator();
      default:
```

```
12            throw new ArgumentException($"Unknown type of play");
13        }
14    }
15 }
```

Tylko w jednym miejscu - w klasie fabryki - będzie tworzona odpowiednia instancja klasy, dalej będziemy korzystać z mechanizmu polimorfizmu. Skorzystaj z dodanych klas w metodzie `EnrichPerformance`:

```
1  private EnrichedPerformance EnrichPerformance(Performance performance)
2  {
3    IPerformanceCalculator calculator = FactoryCalculator.Create(GetPlay(
       performance));
4
5    return new EnrichedPerformance(
6      performance.PlayID,
7      performance.Audience,
8      calculator.AmountFor(performance.Audience),
9      GetPlay(performance),
10     calculator.VolumeCreditsFor(performance.Audience));
11 }
```

Omawiany przykład można by zapewne refaktoryzować dalej, być może w trakcie prac nad kodem okazałoby się konieczne „pójście" w inna stronę. Tym nie mniej z dość chaotycznej pojedynczej metody udało nam się wprowadzić strukturę, która dobrze tłumaczy kod i pozwala na jego rozszerzanie. Jeśli przyjdzie potrzeba zmienić sposób renderowania rachunku to wystarczy dodać nową klasę implementującą `IRenderer` i ją wykorzystać. Jeśli pojawi się potrzeba dodania nowego sposobu wyliczania rabatów wystarczy dodać nowy kalkulator.

Bibliografia

[1] Gary McLean Hall.
Adaptywny kod. Zwinne programowanie, wzorce projektowe i SOLID-ne zasady. WYdanie II. Helion, 2017. ISBN: 9788328338708.

[2] Steve McConnell. Kod doskonały. Jak tworzyć oprogramowanie pozbawione błędów. Helion, 2017. ISBN: 9788328334893.

[3] Martin Fowler. Refaktoryzacja. Ulepszanie struktury istniejącego kodu. Wydanie II. Helion, 2019. ISBN: 9788328355637.

[4] Micah Martin Robert C. Martin.
Agile. Programowanie zwinne: zasady, wzorce i praktyki zwinnego wytwarzania oprogramowania w C#. Helion, 2018. ISBN: 9788328355675.

[5] Refactoring.Guru. URL: `https://refactoring.guru/`.