Programowanie funkcyjne

Materiały z wykładu i laboratorium

Łukasz Bartczuk

11. Języki domenowe w C#

Na tych zajęciach zapoznacie się z wzorcami, jakie możecie wykorzystać w celu utworzenia kodu, który będzie czytelny i poprowadzi użytkownika krok po kroku w celu poprawnego stworzenia jakiegoś obiektu. Wzorce te najczęściej wykorzystywane są podczas tworzenia bibliotek, z których mogą korzystać osoby z małym doświadczeniem programistycznym.

Łącznie techniki te możemy określić mianem płynnego interfejsu (ang. *Fluid Interface*), które również określane są mianem wewnętrznych języków domenowych (ang. *internal domain specific languages*). Języki takie wykorzystują mechanizmy języka, w którym są zaimplementowane.

11.1 Wzorzec Łączenie Metod

Załóżmy, że chcemy stworzyć aplikację, która pozwoli nam i/lub innym osobom zamodelować problem rozmieszczenia w autobusie beaconów i pasażerów. W tym celu musimy oczywiście stworzyć klasy opisujące pasażera, beacona oraz autobus:

```
class Pasazer {
  public int Pozycja { get; internal set; }
  public int WysokoscUmieszczeniaTelefonu { get; internal set; }
}

class Beacon {
  public int Pozycja { get; internal set; }
}

class Autobus
```

```
  {
    List<Pasazer> pasazerowie;
    List<Beacon> beacons;

    public Autobus()
    {
      pasazerowie = new List<Pasazer>();
      beacons = new List<Beacon>();
    }

    public void DodajPasazera(Pasazer p)
    {
      pasazerowie.Add(p);
    }

    public void UstawBeacon(Beacon b)
    {
      beacons.Add(b);
    }
  }
```

Klasy te są bardzo proste i same w sobie nie wymagają tłumaczenia. Poniższy kod pokazuje sposób ich wykorzystania:

```
  class Program
  {
    static void Main(string[] args)
    {
      var autobus = new Autobus();
          autobus.DodajPasazera(new Pasazer());
          autobus.DodajPasazera(new Pasazer());
          autobus.UstawBeacon(new Beacon());
          autobus.UstawBeacon(new Beacon());
    }
  }
```

Zauważmy, że w wywołując poszczególne metody za każdym razem musimy określić obiekt na którym chcemy je wywołać. Oczywiście jest to konieczne, gdyż metodę w programowaniu obiektowym wywołujemy na rzecz określonego obiektu. Szczególnie, jeżeli wywołanie tej metody zmienia jego stan. Jednak może to być dosyć niewygodne oraz zaburzać rytm zapisywania i odczytywania kodu programu.

Jeżeli przeanalizujemy kod klasy Autobus to zobaczymy, że metody DodajPasazera oraz UstawBeacon sa metodami typu void. Oczywiście uwzględniając, jednoznaczność poszczególnych rodzajów podprogramów (funkcje - wykonują obliczenia i zwracają ich wynik, a komendy modyfikują stan programu lub obiektu i nic nie zwracają) wszystko jest zapisane poprawnie. Jednak nie będzie to zaburzeniem tego porządku jeżeli komenda jawnie zwróci obiekt, którego stan zmodyfikowała. Czyli w naszym przypadku metody DodajPasazera oraz UstawBeacon mogą zwrócić obiekt klasy Autobus, na rzecz którego zostały wywołane:

```
public Autobus DodajPasazera(Pasazer p)
{
  pasazerowie.Add(p);
  return this;
}

public Autobus UstawBeacon(Beacon b)
{
  beacons.Add(b);
  return this;
}
```

Ta drobna zmiana spowoduje, że kolejne metody możemy wywoływać na rzecz rezultatu ostatnio wywołanej metody:

```
class Program
{
  static void Main(string[] args)
  {
    var autobus = new Autobus()
                  .DodajPasazera(new Pasazer())
                  .DodajPasazera(new Pasazer())
                  .UstawBeacon(new Beacon())
                  .UstawBeacon(new Beacon());
  }
}
```

Powoduje to, że czytelność naszego programu znacznie się poprawia. Zastosowany tutaj wzorzec projektowy określany jest mianem Łańcucha Metod (ang. *method chaining*).

11.2 Funkcje fabrykujące

Skomplikujmy teraz trochę nasze zadanie, a mianowicie chcemy mieć pewność, że autobus zawsze będzie miał ustawiony przynajmniej jeden beacon i/lub jednego pasażera. Niestety aktualna budowa klasy Autobus nam tego nie zapewnia, ponieważ metody DodajPasazera i UstawBeacon mogą być przez programistę zignorowane i może on ich nie wywołać:

```
class Program
{
  static void Main(string[] args)
  {
    var autobus = new Autobus()
  }
}
```

Spowoduje to stworzenie obiektu, który jest w niepoprawnym stanie (w stosunku do wymagań naszej aplikacji), przez co łatwo może doprowadzić do błędów w naszej aplikacji.

Najprostszym rozwiązaniem tego problemu byłoby stworzenie konstruktora, który będzie przyjmował odpowiednie, wymagane parametry. Skorzystanie z takiego kon-

struktora może jednak zmniejszyć czytelność naszego kodu, a nie zapominajmy, że naszym celem jest stworzenie kodu czytelnego i łatwego do zastosowania dla osób, które średnio znają się na programowaniu. Uwzględniając to lepszym pomysłem będzie zmiana widoczności konstruktora domyślnego dla klasy Autobus z publicznej na prywatną (lub chronioną), przez co próba bezpośredniego utworzenia obiektu za pomocą operatora new nie powiedzie się. Dlatego konieczne jest również utworzenie odpowiednich metod fabrykujących (kolejny wzorzec projektowy), które przyjmą określony parametr i zanim zwrócą tworzony obiekt odpowiednio go skonfigurują:

```
class Autobus
{
  List<Pasazer> pasazerowie;
  List<Beacon> beacons;

  protected Autobus()
  {
    pasazerowie = new List<Pasazer>();
    beacons = new List<Beacon>();
  }

  // metody DodajPasazera i UstawBeacon pozostaja
  // jak w porzedniej wersji

  public static Autobus ZPasazerem(Pasazer pasazer)
  {
    var autobus = new Autobus();
    autobus.DodajPasazera(pasazer);
    return autobus;
  }

  public static Autobus ZBeaconem(Beacon beacon)
  {
    var autobus = new Autobus();
    autobus.UstawBeacon(beacon);
    return autobus;
  }
}
```

Zmiana ta spowodowała, że mamy 100% pewność, że nasz autobus zawsze będzie miał przynajmniej jednego pasażera, lub ustawiony jeden beacon:

```
class Program
{
  static void Main(string[] args)
  {

    var autobus = Autobus.ZBeaconem(new Beacon())
                  .UstawBeacon(new Beacon())
                  .DodajPasazera(new Pasazer());
  }
}
```

11.3 Wzorzec Budowniczy

Kod z ostatniego przykładu bardzo dobrze się sprawdza, jednak powoduje, że klasa Autobus ma teraz za dużo odpowiedzialności. Oprócz opisywania autobusu jest odpowiedzialna również za poprawne tworzenie jej instancji. Z tego powodu funkcje fabrykujące często przenosi się do osobnej klasy, której jedyną odpowiedzialnością jest stworzenie poprawnego obiektu. Jest to tzw. wzorzec Budowniczego (ang. *Builder*). Powoduje on, że jedyną dodatkową odpowiedzialnością klasy, której obiekt ma zbudować jest utworzenie budowniczego. Przedstawia się ona następująco:

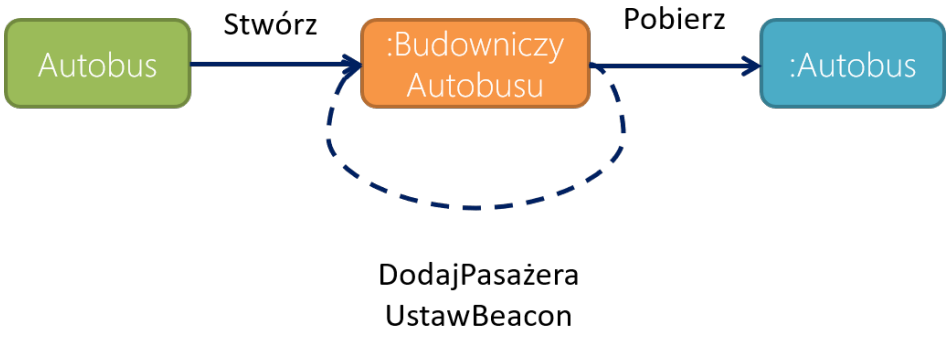
```
class BudowniczyAutobusu {

  private Autobus autobus;

  public BudowniczyAutobusu(Autobus autobus) {
    autobus = autobus;
  }

  public BudowniczyAutobusu UstawBeacon(Beacon b) {
    autobus.UstawBeacon(b);
    return this;
  }

  public BudowniczyAutobusu DodajPasazera(Pasazer p) {
    autobus.DodajPasazera(p);
    return this;
  }

  public Autobus Pobierz() => autobus;
}
```

W klasie Autobus dodałem jedną statyczną metodę Stworz, która jest odpowiedzialna za stworzenie budowniczego i zwrócenie go:

```
class Autobus
{
  List<Pasazer> pasazerowie;
  List<Beacon> beacons;

  protected Autobus()
  {
    pasazerowie = new List<Pasazer>();
    beacons = new List<Beacon>();
  }

  public void DodajPasazera(Pasazer p)
  {
    pasazerowie.Add(p);
  }

  public void UstawBeacon(Beacon b)
  {
```

```
      beacons.Add(b);
    }

    public static BudowniczyAutobusu Stworz() {
      var autobus = new Autobus();
      return new BudowniczyAutobusu(autobus);
    }
  }
```

Zauważmy, że metoda Stworz tworzy obiekt klasy Autobus i przekazuje go do budowniczego. Jest to konieczne, gdyż chcemy mieć pod kontrolą sposób tworzenia obiektów tej klasy. Wykorzystanie budowniczego możemy przedstawić na diagramie, przypominającym trochę diagram stanów z języka UML:



Odczytamy go w następujący sposób: ”rozpoczynamy od klasy Autobus na której wywołujemy metodę Stworz. Uzyskujemy w ten sposób instancję klasy BusBuilder, na rzecz której możemy wywołać zero lub więcej razy metody DodajPasazera lub UstawBeacon w dowolnej kolejności. Wywołując metodę Pobierz uzyskujemy odpowiednio skonfigurowaną instancję klasy Autobus”.

Stosując napisane klas zgodnie z powyższym opisem uzyskamy następujący kod:

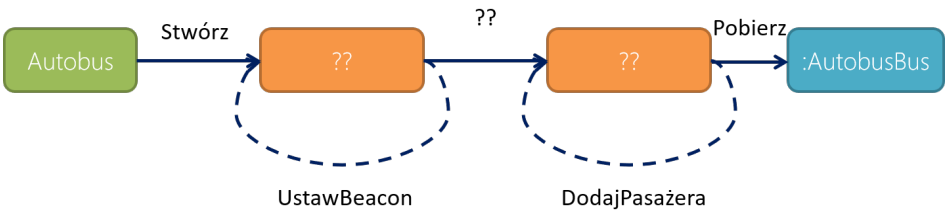
```
class Program
{
  static void Main(string[] args)
  {
    Autobus.Stworz()
       .DodajPasazera(new Pasazer())
       .UstawBeacon(new Beacon())
       .DodajPasazera(new Pasazer())
       .UstawBeacon(new Beacon())
       .Pobierz();
  }
}
```

Zwróćmy uwagę, ze budowniczy stworzony w tej postaci nie gwarantuje utworzenia obiektu, który będzie miał ustawiony choćby jeden beacon lub dodanego pasażera. To co będziemy chcieli uzyskać to strukturę, która wymusi na programiście dodanie beacona lub pasażera. Oczywiście nie będziemy chcieli się zbytnio przemęczać. Musimy

więc to zrobić jak najmniejszym wysiłkiem. Zrobimy to małymi krokami.

Najpierw zaczniemy od odrobinę prostszego problemu. Chcemy mieć pewność, że w autobusie będzie na pewno co najmniej jeden pasażer, beacon będzie opcjonalny. Wymaga to rozbicia procesu tworzenia autobusu na dwa osobne kroki. W pierwszym będziemy mogli dodać zero lub więcej beaconów, a w drugim co najmniej jednego pasażera. Przy okazji możemy trochę uporządkować kod wymuszając najpierw ustawienie beaconów, a później pasażerów autobusu.

Sytuacje taką możemy przedstawić na następującym diagramie:



Rozpoczynamy oczywiście od wywołania metody Stwórz na klasie Autobus. W rezultacie powinniśmy uzyskać strukturę, która umożliwi nam ustawienie w autobusie beacona lub przejść do kroku dodawania pasażerów. Jeżeli zdecydujemy się ustawić beacon, to później możemy kolejno dodać kolejnego beacona lub dodać pasażera. Jeżeli przejdziemy do kroku dodawania pasażerów, to będziemy mogli ich ustawiać lub uzyskać gotowy, skonfigurowany autobus.

Pozostaje pytanie jak zamodelować oba konieczne do wykonania kroki oraz przejście pomiędzy nimi. Pamiętajmy jednocześnie, że nie będziemy chcieli się zbyt dużo napracować, a uzyskać maksymalne zyski.

Zanim odpowiemy na to pytanie wróćmy na chwilę do interfejsów. Załóżmy, że mamy dane dwa interfejsy oraz klasę, która je implementuje:

```
interface IInterface1
{
  void Metoda1a();
  void Metoda1b();
}

interface IInterface2
{
  void Metoda2a();
  void Metoda2b();
}

class Przyklad : IInterface1, IInterface2
{
  public void Metoda1a() { }
  public void Metoda1b() { }
  public void Metoda2a() { }
  public void Metoda2b() { }
```

```
    }
```

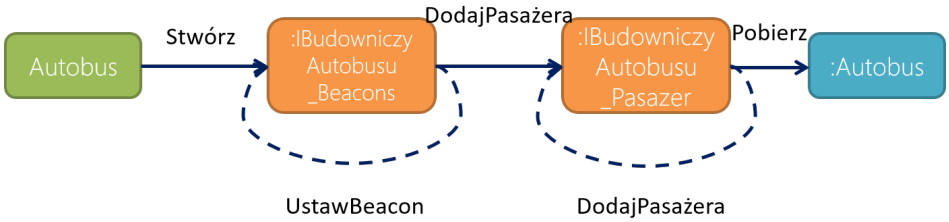
Teraz jeżeli stworzymy obiekt tej klasy, to możemy go przypisać do zmiennych o typie IInterface1 lub IInterface2:

```
class Program
{
  static void Main(string[] args)
  {
    Przyklad przyklad = new Przyklad();
    IInterface1 int1 = przyklad;
    IInterface2 int2 = przyklad;

    int1.Metoda1a();
    int2.Metoda2a();
  }
}
```

Pomimo, że obiekt przykład posiada wszystkie cztery metody, to poprzez przypisanej go do zmiennej o określonym interfejsie, możemy skorzystać tylko z tych, które są zdefiniowane w tym interfejsie (oczywiście bez dokonywania dodatkowego rzutowania).

Przykład ten pokazuje, że interfejsy możemy wykorzystać aby spojrzeć na dany obiekt z różnej perspektywy.

Powyższa uwaga na temat interfejsu rozwiązuje nasze problemy. Musimy zdefiniować dwa interfejsy, które odpowiadają wcześniej opisanym krokom. Pierwszy z nich będzie zawierał metody, które pozwalają dodawać beacony oraz przejść do kroku drugiego, a drugi metody pozwalające dodawać pasażerów oraz otrzymać gotowy obiekt. Ponieważ ustaliliśmy wcześniej, że dodanie co najmniej jednego pasażera jest obowiązkowe, to właśnie metoda pozwalająca go dodać powinna przenosić nas z jednego interfejsu do drugiego. Czyli poprzedni diagram możemy uzupełnić w następujący sposób:



Definicja naszych interfejsów będzie następująca:

```
interface IBudowniczyAutobusu_Beacons
{
  IBudowniczyAutobusu_Beacons UstawBeacon(Beacon b);
  IBudowniczyAutobusu_Pasazer DodajPasazera(Pasazer p);
}

interface IBudowniczyAutobusu_Pasazer
```

```
{
  IBudowniczyAutobusu_Pasazer DodajPasazera(Pasazer p);
  Autobus Pobierz();
}
```

Oba interfejsy mogą być przypisane do tej samej klasy BudowniczyAutobusu:

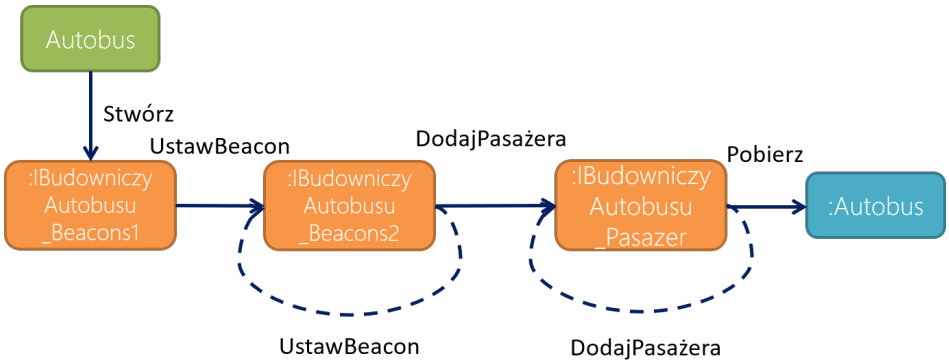
```
class BudowniczyAutobusu :
   IBudowniczyAutobusu_Pasazer,
   IBudowniczyAutobusu_Beacons
{
  private Autobus autobus;

  public BudowniczyAutobusu(Autobus autobus) { autobus = autobus; }

  public IBudowniczyAutobusu_Beacons UstawBeacon(Beacon b) {
    autobus.UstawBeacon(b);
    return this;
  }

  public IBudowniczyAutobusu_Pasazer DodajPasazera(Pasazer p) {
    autobus.DodajPasazera(p);
    return this;
  }

  public Autobus Pobierz() => autobus;
}
```

Musimy pamiętać, że w tym przypadku metoda Stworz klasy Autobus, powinna zwracać obiekt dający dostęp do pierwszego kroku:

```
class Autobus
{
  List<Pasazer> pasazerowie;
  List<Beacon> beacons;

  protected Autobus()
  {
    pasazerowie = new List<Pasazer>();
    beacons = new List<Beacon>();
  }

  public void DodajPasazera(Pasazer p)
  {
    pasazerowie.Add(p);
  }

  public void UstawBeacon(Beacon b)
  {
    beacons.Add(b);
  }

  public static IBudowniczyAutobusu_Beacons Stworz() {
```

```
      var autobus = new Autobus();
      return new BudowniczyAutobusu(autobus);
    }
  }
```

W powyższym przykładzie dodanie beacona było opcjonalne. Co jednak powinniśmy zrobić, jeżeli będziemy chcieli, aby był on obowiązkowy? Musimy postąpić jak z dodawaniem pasażera - stworzyć następny krok, którego nie będzie się dało objeść:



W kodzie będzie się to przedstawiało następująco:

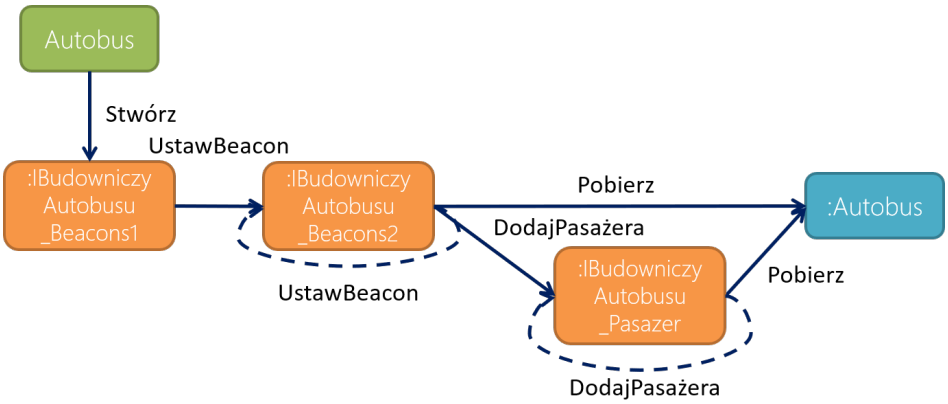
```
interface IBudowniczyAutobusu_Beacons1
{
  IBudowniczyAutobusu_Beacons2 UstawBeacon(Beacon b);
}

interface IBudowniczyAutobusu_Beacons2
{
  IBudowniczyAutobusu_Beacons2 UstawBeacon(Beacon b);
  IBudowniczyAutobusu_Pasazer DodajPasazera(Pasazer p);
}

class BudowniczyAutobusu : IBudowniczyAutobusu_Pasazer, IBudowniczyAutobusu_Beacons1
    , IBudowniczyAutobusu_Beacons2
{
  private Autobus autobus;

  public BudowniczyAutobusu(Autobus autobus)
    { autobus = autobus; }

  public IBudowniczyAutobusu_Beacons2 UstawBeacon(Beacon b) {
    autobus.UstawBeacon(b);
    return this;
  }

  public IBudowniczyAutobusu_Pasazer DodajPasazera(Pasazer p) {
    autobus.DodajPasazera(p);
    return this;
  }
```

```
    public Autobus Pobierz() => autobus;
  }
```

Oczywiście, musimy pamiętać o poprawie metody Stworz klasy Autobus w następujący sposób:

```
  public static IBudowniczyAutobusu_Beacons1 Stworz() {
    var autobus = new Autobus();
    return new BudowniczyAutobusu(autobus);
  }
```

Kolejnym krokiem jaki możemy podjąć, jest zmiana naszej struktury poleceń, tak aby dodanie pasażera było opcjonalne:



Na szczęście taka zmiana jest bardzo prosta do zastosowania. Wystarczy dopisać do interfejsu IBudowniczyAutobusu_Beacons2 metodę Pobierz zwracającą skonfigurowany autobus:

```
  interface IBudowniczyAutobusu_Beacons2
  {
    IBudowniczyAutobusu_Beacons2 UstawBeacon(Beacon b);
    IBudowniczyAutobusu_Pasazer DodajPasazera(Pasazer p);
    Autobus Pobierz();
  }
```

W tym przypadku do klasy BudowniczyAutobusu nie musimy dopisywać żadnego kodu, ponieważ metoda Pobierz została już zaimplementowana wcześniej, aby spełnić wymagania interfejsu IBudowniczyAutobusu_Pasazer.

Skomplikujmy nasz problem jeszcze bardziej (ale dzisiaj już po raz ostatni) dodając do niego możliwość konfiguracji obiektów klasy Beacon i Pasazer tak aby dało się wykonać następujące wywołania:

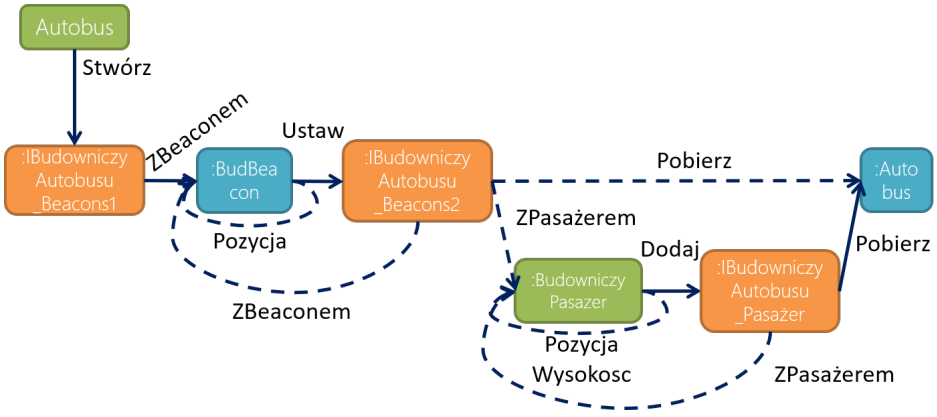
```
  class Program
  {
    static void Main(string[] args)
    {
      var autobus = Autobus.Stworz()
                           .ZBeaconem()
                              .Pozycja(1)
```

```
                .Ustaw()
            .ZBeaconem()
                .Pozycja(2)
                .Ustaw()
            .ZPasazerem()
                .Pozycja(3)
                .Wysokosc(4)
                .Dodaj()
            .Pobierz();
    }
}
```

Wynika z niego, że metoda statyczna Stworz klasy Autobus powinna, tak jak do tej pory, zwrócić obiekt klasy BudowniczyAutobusu. Skoro tak, to będziemy musieli do niego (i do odpowiednich interfejsów) dołączyć dwie nowe metody: ZBeaconem oraz ZPasazerem. Ponieważ nie chcemy łamać zasady pojedynczej odpowiedzialności, będziemy musieli stworzyć dwie nowe klasy BudowniczyBeacon oraz BudowniczyPasazer o odpowiednich metodach pozwalających na ustawienie określonych właściwości.

W przypadku klasy BudowniczyAutobusu w celu pobrania skonfigurowanego obiektu musieliśmy wywołać metodę Pobierz. W klasach BudowniczyBeacon oraz BudowniczyPasazer również potrzebujemy podobnych metod, jednak ich działanie będzie trochę inne. Będziemy chcieli, aby po ich wywołaniu utworzony obiekt został dodany do autobusu, oraz abyśmy mogli powrócić do konfiguracji autobusu. Czyli metody Dodaj i Ustaw powinny zwracać wartość odpowiadającą określonemu interfejsowi definiującemu klasę BudowniczyAutobusu.

Całość koniecznych do wykonania działań możemy przedstawić na poniższym diagramie:



Aby klasy BudowniczyBeacon oraz BudowniczyPasazer mogły zwrócić z metod Ustaw i Stworz obiekt BudowniczyAutobusu i na dodatek ten sam, który został utworzony w metodzie Stworz w klasie Autobus, najprościej przekazać go poprzez konstruktor.

Kod nowych klas jest następujący:

```
class BudowniczyBeacona
```

```
  {
    Beacon beacon;

    private BudowniczyAutobusu budowniczy;

    public BudowniczyBeacona(BudowniczyAutobusu budowniczy)
    {
      this.budowniczy = budowniczy;
      beacon = new Beacon();
    }

    public BudowniczyBeacona Pozycja(int pozycja)
    {
      beacon.Pozycja = pozycja;
      return this;
    }

    public IBudowniczyAutobusu_Beacons2 Ustaw()
    {
      budowniczy.UstawBeacon(beacon);
      return budowniczy;
    }
  }

  class BudowniczyPasazera
  {
    Pasazer pasazer;
    private BudowniczyAutobusu budowniczy;

    public BudowniczyPasazera(BudowniczyAutobusu budowniczy)
    {
      this.budowniczy = budowniczy;
      pasazer = new Pasazer();
    }

    public BudowniczyPasazera Pozycja(int pozycja)
    {
      pasazer.Pozycja = pozycja;
      return this;
    }

    public BudowniczyPasazera Wysokosc(int wysokosc)
    {
      pasazer.WysokoscUmieszczeniaTelefonu = wysokosc;
      return this;
    }

    public IBudowniczyAutobusu_Pasazer Dodaj()
    {
      budowniczy.DodajPasazera(pasazer);
      return budowniczy;
```

```
  }
}
```

Zmodyfikowane interfejsy oraz klasa BudowniczyAutobusu:

```
interface IBudowniczyAutobusu_Beacons1
{
  BudowniczyBeacona ZBeaconem();
}

interface IBudowniczyAutobusu_Beacons2
{
  BudowniczyBeacona ZBeaconem();
  BudowniczyPasazera ZPasazerem(Pasazer p);
}

interface IBudowniczyAutobusu_Pasazer
{
  BudowniczyPasazera ZPasazerem(Pasazer p);
  Autobus Pobierz();
}

class BudowniczyAutobusu :
      IBudowniczyAutobusu_Beacons1,
      IBudowniczyAutobusu_Beacons2,
      IBudowniczyAutobusu_Pasazer
{
  private Autobus autobus;

  public BudowniczyAutobusu(Autobus autobus) {
    autobus = autobus;
  }

  public BudowniczyBeacona ZBeaconem()
  {
    return new BudowniczyBeacona(this);
  }

  public BudowniczyPasazera ZPasazerem()
  {
    return new BudowniczyPasazera(this);
  }

  public void UstawBeacon(Beacon b) {
    autobus.UstawBeacon(b);
  }


  public void DodajPasazera(Pasazer p) {
    autobus.DodajPasazera(p);
  }
```

```
    public Autobus Pobierz()
    {
      return autobus;
    }
  }
```

11.4 Zadania do samodzielnego wykonania

Zadanie 11.1 Napisz klasę Rysunek w taki sposób, aby umożliwiała zapisanie następującego wywołania:

```
var rysunek =
  new Rysunek()
    .Prostokat(x:1,y:1,
               szerokosc:40,wysokosc50,
               wypelnienie:"czerwony", gruboscLinii:2)
    .Kolo(x:1,y:5, promien:5,wypelnienie:"zielony")
    .Linia(x1:1,y1:1,x2:5,y2:10,linia:"zielony")
```

W przypadku tego programu kolejność oraz liczba wywołań poszczególnych funkcji nie ma znaczenia.

Zadanie 11.2 Załóż, że tworzysz aplikację do wystawiania faktur. Musisz w niej stworzyć klasę Faktura, która będzie zawierała następujące pola:

1. Number faktury
2. Data wystawienia faktury
3. Data sprzedazy
4. Sprzedawca
5. Nabywca
6. Pozycje:
   (a) Nazwa produktu
   (b) Jednostka
   (c) Ilość
   (d) Cena jednostkowa netto
   (e) Stawka Vat

Upewnij się, że każda faktura ma ustawione wszystkie dane i posiada przynajmniej jedną pozycję.

Zadanie 11.3 Wykorzystaj wzorzec budowniczego w celu uzyskania następującego kodu:

```
Faktura
  .NumerFaktury(...)
  .DataWystawieniaFaktury(...)
  .DataSprzedazy(...)
  .Sprzedawca(...)
  .Nabywca(...)
  .Pozycja(...)
  .Pozycja(...)
  .Generuj();
```

Oczywiście w miejsce kropek wstawiamy odpowiednie dane zgodnie z zapisem z poprzedniego zadania.

Zadanie 11.4 Rozszerz poprzednie zadanie tak, aby każda z metod oczywiście oprócz DodajProdukt mogła być wywołana tylko raz w określonej kolejności (np takiej jak pokazałem powyżej).

Zadanie 11.5 Rozszerz poprzednie zadanie tak, aby możliwy był następujący zapis:

```
Faktura
.NumerFaktury(...)
.DataWystawieniaFaktury(...)
.DataSprzedazy(...)
.Sprzedawca(...)
.Nabywca(...)
.Pozycja(...)
  .Nazwa(...)
  .Jedostka(...)
  .Ilosc(...)
  .CenaJednostkowa(...)
  .Vat(...)
  .Dodaj(...)
.Pozycja(...)
  .Nazwa(...)
  .Jedostka(...)
  .Ilosc(...)
  .CenaJednostkowa(...)
  .Vat(...)
  .Dodaj(...)
.Generuj();
```