Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Piotr Nitecki
Nr albumu 7502

Framework do deklaratywnego tworzenia GUI

Praca magisterska
napisana pod kierunkiem:

dr inż. Mariusza Trzaski

Warszawa, luty 2012

Streszczenie

Praca skupia się na zagadnieniu tworzenie GUI. Omówione w niej zostały problemy towarzyszące programiście podczas pisania kodu odpowiedzialnego za interfejs użytkownika. Autor pracy przygląda się aktualnym technologiom wytwarzania GUI, skupiając się szczególnie na podejściu deklaratywnym oraz rozwiązaniach, które automatyzują cześć pracy programisty.

Autor przedstawia koncepcję deklaratywnego języka do tworzenia GUI, ze zwięzłą i samo tłumaczącą się składnią. Podejście deklaratywne pozwala na pracę na wyższym poziomie abstrakcji, co zarówno zmniejsza pracochłonność zadań programistycznych i jak i redukuję ilość miejsc, w których można popełnić błąd. Język do tworzenia GUI został zaimplementowany w C# i umożliwia stworzenie formularza z automatycznie wygenerowanymi kontrolkami na podstawie modelu przekazanego w formie instancji obiektu C#.

Spis Treści

1. Wstęp 5
1.1. Cel pracy 5
1.2. Rezultat pracy 5
1.3. Organizacja pracy 6
2. Programowanie GUI 7
2.1 Różne podejścia do programowania GUI 7
2.1.1 Ręczne pisanie kodu 7
2.1.2 Wizualne edytory WYSWIG 7
2.1.3 Podejście deklaratywne 8
2.2 Przegląd istniejących rozwiązań 8
2.2.1 Adobe Flex 8
2.2.2 Asp.Net Dynamic Data 10
2.2.3 WPF 12
2.2.4 SenseGUI 15
2.2.5 ASP.NET MVC 20
2. Koncepcja rozwiązania 23
3.1 Propozycja nowego rozwiązania 23
3.2 Motywacja przyjętego rozwiązania 23
3.3 Wymagania funkcjonalne 24
3.4 Opis rozwiązania 25
3.5 Składnia 27
3.6 Przykłady 27
3. Opis technologii użytych w pracy 32
4.1. .Net 33
4.2. Visual Studio 2010 34

4.3. C# ... 35
4.4. Wyrażenia lambda ... 37
4.5. Drzewa wyrażeń ... 37
4.6. Płynny interfejs ... 38
4.7. Refleksja ... 40
4.8. Windows Forms ... 41
4.9. JetBrains ReSharper 5. ... 42
5. Opis implementacji ... 44
5.1. Struktura projektu ... 44
5.2. Konfiguracja atrybutu modelu ... 46
5.3. Konfiguracja modelu ... 48
5.4. Walidacja ... 49
5.5. Walidatory ... 52
5.6. Tworzenie kontrolek ... 54
5.7. ControlBuildersRegistry ... 56
5.8. Klonowanie wartości ... 58
5.9. Silna kontrola typów ... 59
5.10. Budowniczy formularza ... 61
6. Podsumowanie ... 63
6.1. Wady i zalety rozwiązania ... 63
6.2. Proponowany plan rozwoju ... 63
6.3. Wnioski końcowe ... 64
7. Bibliografia ... 65

1. Wstęp

Istnieją trzy podstawowe podejścia do tworzenia interfejsów użytkownika. Kod można pisać ręcznie metodą imperatywną przy wykorzystaniu niskopoziomowego API. Można także skorzystać z edytorów WYSWYIG, które wygenerują niskopoziomowy kod za programistów. W końcu można budować interfejs użytkownika korzystając z podejścia deklaratywnego.

Deklaratywne podejście ukrywa niepotrzebne szczegóły implementacyjne, a programista koncentruje się na tym, co chce zrobić, a nie jak chce to osiągnąć. Takie podejście zwiększa wydajność programistów, i jednocześnie redukuje możliwość powstawania błędów podczas tworzenia komponentów GUI.

Koncepcja deklaratywnego języka jest implementowana na kilka sposobów. Najpopularniejsze z nich to użycie technologii opartych o XML, takich jak XAML w technologii WPF czy MXML w Adobe Flex, lub użycie adnotacji, jak w Asp.NET MVC oraz Dynamic Data. W końcu podejście deklaratywne można zaimplementować przy pomocy DSL jak to zostało zrobione w GCL.

1.1. Cel pracy

Autor postawił sobie za cel przedstawienie technologii aktualnie używanych do programowania graficznych interfejsów użytkownika. Przegląd rozwiązań koncentruje się na podejściu deklaratywnym, sposobach automatyzacji wytwarzania kontrolek oraz ilości pracy jaką musi włożyć programista do powstania interfejsu użytkownika przy zastosowaniu konkretnej technologii.

Kolejnym celem pracy było stworzenie założeń oraz opracowanie składni języka do tworzenia graficznych interfejsów użytkownika. Na podstawie założeń miała powstać działająca implementacja biblioteki wraz z językiem do deklaratywnego tworzenia GUI.

1.2. Rezultat pracy

Rezultatem pracy jest określenie podstawowych założeń języka do tworzenia interfejsów użytkownika. Powstał deklaratywny język ze zwięzłą i samo tłumaczącą się składnią.

Wynikiem pracy jest także biblioteka oraz implementacja języka. Rozwiązanie to zostało zaimplementowane w języku C# pod platformę .Net oraz Windows Forms. Stworzony język

zachowuje silną kontrolę typów, umożliwia internacjonalizację elementów GUI, dodawanie reguł walidacyjnych oraz minimalizuje nakład pracy dzięki wykorzystanie konwencji do tworzenia widgetów.

1.3. Organizacja pracy

Praca została podzielna na następujące rozdziały:

1. Wstęp, w którym opisano cel pracy i jej rezultaty,
2. Opis różnych podejść do tworzenia GUI oraz przegląd technologii do deklaratywnego tworzenia GUI,
3. Koncepcję rozwiązania, składnię oraz przykłady zastosowania języka,
4. Omówienie narzędzi i technologii użytych przy tworzeniu rozwiązania,
5. Opis implementacji, wady i zalety rozwiązania, a także plan rozwoju języka,
6. Podsumowanie pracy.

2. Programowanie GUI

2.1 Różne podejścia do programowania GUI

W tym rozdziale na podstawie pracy Mariusza Trzaski [1] zostaną omówione trzy najbardziej popularne podejścia do tworzenia interfejsu użytkownika. Po pierwsze zostanie omówiony najstarszy sposób programowania GUI czyli ręczne pisanie kodu, następnie najbardziej popularne podejście, czyli używanie wizualnych edytorów do budowania GUI. Na końcu autor pracy przyjrzy się deklaratywnemu podejściu do tworzenia GUI.

2.1.1 Ręczne pisanie kodu

Ręczne pisanie kodu jest najstarszym sposobem tworzenia GUI. Polega na wykorzystaniu odpowiedniego API (ang. Application Programming Interface) np. Swing dla Javy czy Windows Forms dla .Net do niskopoziomowego skonstruowania GUI. Takie podejście można zaliczyć do programowania imperatywnego – programista musi określić, nie tylko co ma być zrobione, ale także, jak każda kontrolka ma być stworzona oraz jak ma być umiejscowiona w layoucie. Takie podejście jest pracochłonne oraz skutkuje dużą ilością błędów. Często mała zmiana layoutu wymaga przepisania dużej części kodu odpowiedzialnego za GUI. Zaletą ręcznego pisania kodu jest natomiast pełna kontrola nad GUI. Ręczne pisanie kodu powinno być używane tylko wyjątkowo, gdy inne sposoby zawiodą.

2.1.2 Wizualne edytory WYSWIG

Wizualne edytory WYSWIG (ang. What You See Is What You Get), jak sama nazwa wskazuje, umożliwiają wizualne stworzenie GUI za pomocą myszki oraz gotowych komponentów. Ta metoda pozwala na szybkie tworzenie interfejsów i jest łatwa w nauce. Takie rozwiązanie ma jednak kilka wad. Przede wszystkim edytory wizualne posiadają skromniejsze możliwości i jest wiele sytuacji, w których programista będzie musiał ręcznie stworzyć kod. Po drugie, kod wygenerowany przez edytor WYSWIG jest daleki od ideału. Jest to szczególnie uciążliwe podczas tworzenia stron internetowych. Wynikowy kod html nie jest zgodny ze standardami W3C, zawiera inlinowe style CSS, bardzo często opiera się o tabele i niejednakowo się wyświetla w różnych przeglądarkach.

2.1.3 Podejście deklaratywne

Podejście deklaratywne do tworzenie GUI polega na tym, że programista nie skupia się na szczegółach implementacji, tylko opisuje to, co ma być zrobione. Deklaratywne programowanie znacznie skraca czas tworzenia GUI, lecz ze względu na pewne założenia i konwencje, nie pasuje do wszystkich przypadków w aplikacji. Ze względu na ten fakt programowanie deklaratywne często łączy się z ręcznym pisaniem kodu.

2.2 Przegląd istniejących rozwiązań

W tym rozdziale zostaną omówione wiodące technologie wykorzystujące deklaratywne podejście do tworzenia GUI. Autor przyjrzy się zarówno technologiom używanym do budowania aplikacji desktopowych, takim jak WPF, senseGUI, jak webowym, takim jak Adobe Flex, Dynamic Data oraz Asp.Net MVC.

2.2.1 Adobe Flex

Adobe Flex [2] jest opensourcowym frameworkiem do budowania aplikacji RIA, czyli wysoce interaktywnych aplikacji internetowych lub desktopowych. W Adobe Flex język MXML służy do tworzenia graficznego interfejsu użytkownika. ActionScript, oparty na ECMAScript, jest odpowiedzialny za kontrolę interakcji użytkownika z aplikacją. W skład technologii Flex, stworzonej prze firmę Adobe Systems, wchodzi także FLEX SDK, który zawiera wiele wbudowanych kontrolek m.in. przyciski, listy elementów (ang. listbox), widok drzewa (ang. treeview), tabele (ang. datagrid) oraz kontenery layoutu, przydatne podczas budowania interfejsu użytkownika. Adobe Flex jest konkurencją dla takich technologii jak Silverlight, JAVA FX oraz AJAX.

MXML jest deklaratywnym językiem znaczników opartym o XML, służącym do tworzenia interfejsu użytkownika. Każda aplikacja napisana w języku MXML zaczyna się od tagu deklarującego kodowanie Unicode oraz wersję XML. Następnie deklarowany jest główny kontener, w którym będzie zawarta reszta stworzonych kontrolek. W Listing 1 zaprezentowany jest kod, który tworzy obraz, etykietę oraz przycisk. Warto zwrócić uwagę na fakt, że tagi napisane w języku MXML są konwertowane na kod ActionScript.

Kontrolki stworzone przy pomocy MXML będą widoczne dla programisty z poziomu Action Script, co pozwoli na modyfikowanie wyglądu w trakcie wykonania programu. Język

MXML ze względu na deklaratywny paradygmat nie nadaje się do obsługi zdarzeń, a tym bardziej do implementacji logiki biznesowej, toteż programista pisząc aplikację w tym języku musi posiłkować się ActionScriptem. Funkcjonalność napisaną w Action Script można zarówno wpleść w kod MXML jak i umieścić w osobnym pliku. Druga metoda jest preferowana, gdyż umożliwia separację definicji wyglądu i logiki biznesowej.

Kolejną właściwością MXML jest możliwość definiowania wyglądu przy użyciu stylów CSS. Dzięki temu można uzyskać większą kontrolę nad wyglądem naszej aplikacji. Kod CSS może zostać napisany raz i zaaplikowany do wielu modułów, a co ważniejsze, kod style CSS napisany mogą być stworzone przez grafika, czyli osobę z lepszymi predyspozycjami do projektowania wyglądu aplikacji niż programista.

```
<?xml version="1.0" encoding="utf-8"?>

<mx:Application xmlns:mx=http://www.adobe.com/2006/mxml
backgroundColor="#676767"
<mx:Image source="images/exampleimage.jpg" />
<mx:Label text="Przykładowy obraz" />
<mx:Button label="Przysick" click="powiekszobraz(event)" />
</mx:Application>
```

Listing 1 Przykładowy kod MXML wyświetlający obraz wraz z podpisem oraz przysiekiem.

Język MXML ze względu na strukturę opartą na XML jest bardzo łatwy do opanowania. Jego czytelność i przejrzystość widać wyraźnie, gdy porówna się MXML z kodem Action Script widocznym Listing 2 użytym do implementacji takiego samego interfejsu użytkownika. Deklaratywny język MXML, zgodnie ze swoją specyfiką, opisuje właściwości GUI jakie chce uzyskać programista a kod Action Script skupia się na szczegółach implementacyjnych.

```
<mx:Script>
        <![CDATA[
    import mx.controls.Image;
    import mx.controls.Label;
    var image:Image = new Image();
      image.source = "images/exampleimage.jpg";
```

```
            this.addChild(photo);
            var label:Label = new Label();
            label.text = " Przykładowy obraz";
            this.addChild(label);
            var button:Button = new Button();
            button.label = " Przykładowy obraz";
            button.addEventListener(MouseEvent.CLICK, powiekszobraz);
            this.addChild(button);
             ]]>
</mx:Script>
```

Listing 2 Imperatywny kod Action Script odpowiedzialny za tworzenie GUI.

Podsumowując, główną zaletą języka MXML jest jego czytelność i przejrzystość oraz możliwość separacji logiki biznesowej od tworzenia interfejsu użytkownika. Kolejnym atutem jest możliwość używania kodu CSS, co daje większą kontrolę na wyglądam interfejsu oraz umożliwia wielokrotne użycie stylów.

Wadą technologii Adobe Flex jest brak dobrych darmowych IDE. Kolejnym problemem jest fakt, że Adobe Flex jest technologią służącą tylko do prezentacji i wymaga części serwerowej, którą zwyczaj tworzy się w technologii J2EE. To sprawia, że napisanie aplikacji w technologii Adobe Flex wymaga znajomości MXML, Action Script oraz Javy. Jest to poważna wada, szczególnie jeśli system jest tworzony przez mały zespół programistów. Przykładem technologii RIA, która nie wymaga znajomości dużej ilości języków programowania jest Silverlight firmy Microsoft. Interfejs użytkownika tworzony jest za pomocą deklaratywnego XAML, a język C# może być zastosowany zarówno po stronie klienckiej jak i serwerowej.

2.2.2 Asp.Net Dynamic Data

ASP.NET Dynamic Data [3] to framework, który umożliwia szybkie tworzenie aplikacji, które służą głownie do zarządzania danymi. Technologia ta umożliwia automatyczne generowania kontrolek służących do przeglądania, edycji, dodawania oraz usuwania danych. Dynamic Data używa meta modelu, na podstawie którego odczytuje właściwości obiektów

oraz relację pomiędzy obiektami. Dynamic Data ma wbudowaną funkcję obsługi meta modeli bazujących na LINQ to SQL lub Entity Framework.

Aplikację bazującą na LINQ to SQL można stworzyć w kilka chwil. Wystarczy podłączyć się do bazy danych przy pomocy Visual Studio i wybrać określone tabele. Odpowiednie klasy zostaną wygenerowane automatycznie. W ten sposób można otrzymać aplikację, która pozwoli wykonywać podstawowe operacie na danych, takie jak dodawanie, edycja, usuwanie oraz pobieranie danych. Dynamic Data różni się od typowych generatorów kodu rozbudowanymi możliwościami konfiguracji i rozszerzania. Pozwala tworzyć własne, oraz edytować istniejące szablony używane do wyświetlania i edycji danych obiektów. Dynamic Data umożliwia także konfigurowanie szablonów używanych do wyświetlania oraz edycji podstawowych typów danych, takich jak data, wartości numeryczne oraz tekstowe.

Jak zostało wspomniane wcześniej, meta model jest budowany automatycznie na podstawie struktury bazy danych. Jednak programista ma możliwość nadpisać go, używając klas częściowych (ang. partial class) oraz adnotacji. Na Listing 3 skonfigurowano klasę *Category* w taki sposób, aby jej id nie było wyświetlane, a właściwość *Description* była opatrzona nową etykietą „Przykładowy opis”.

```
[MetadataType(typeof(Categories))]
  public partial class Category {
    public class Categories {
      [ScaffoldColumn(false)]
      public object ID { get; set; }
      [DisplayName("Przyładkowy opis")]
      public string Description { get; set; }
    }
  }
```

Listing 3 Konfigurowanie modelu za pomocą adnotacji.

Dla każdej klasy wygenerowany jest zestaw metod częściowych (ang. partial method), takich jak OnCreated, OnChanging, OnChanged itd., które pozwalają „zaczepić” logikę biznesową do Dynamic Data. Takie rozwiązanie nie jest idealne, gdyż ma ściśle określone struktury i dostosowanie modelu do specyficznych potrzeb jest bardzo trudne.

Podsumowując, ASP.NET Dynamic Data umożliwia szybkie tworzenie aplikacji na podstawie LINQ to SQL oraz Entity Framework. Wydaje się to bardzo dobrym rozwiązaniem

przy tworzeniu prototypów oraz programów skupiających się głównie na danych oraz nie posiadających skomplikowanej logiki biznesowej. W przypadku bardziej skompilowanych systemów czas, który można zaoszczędzić korzystając z szybkiego rozwiązania jakim jest Dynamic Data, nie zrekompensuje czasu jaki trzeba poświęcić, aby dostosować je do specyficznych potrzeb. Mimo, że autor pracy nie uważa Dynamic Data za bardzo użyteczny framework, należy docenić pomysły, które zostały w nim zastosowane – generowanie UI na podstawie meta modelu oraz możliwość konfiguracji interfejsu użytkownika za pomocą adnotacji. Te koncepty zostały wykorzystane w ASP.NET MVC, i zostaną także omówione w dalszych częściach pracy.

2.2.3 WPF

WPF [4] to część .NET frameworka służąca do budowania aplikacji z zaawansowanym interfejsem użytkownika na platformę Windows. WPF renderuje grafikę wykorzystując DirectX, sprzętowo wspomagane graficzne API. Dzięki temu aplikacje napisane w WPF, nawet jeśli wykorzystują zaawansowane efekty graficzne, nie obciążają bardzo procesora, dzięki wykorzystaniu pracy karty graficznej. WPF jest następcą technologii Windows Forms. Jedną z głównych wad Windows Forms były silne zależności występujące pomiędzy kodem odpowiedzialnym za definiowanie interfejsu użytkownika, a implementacją logiki biznesowej. Problem ten wynikał między innymi z faktu, iż właściwości każdej kontrolki były opisane w automatycznie wygenerowanej klasie. Programista chcąc zmienić dynamicznie właściwości kontrolki ma możliwość wykonania tego tylko za pomocą kodu C#/VB.NET. Podczas tworzenia aplikacji w Windows Forms grafik ma bardzo ograniczone pole manewru. Może albo pracować z Visual Studio, które nie ma rozbudowanych funkcji służących do tworzenia grafiki, albo może wyeksportować swoją pracę to pliku bitmapowego, który zostanie użyty przez programistę. W WPF problem przeplatania się kodu służącego to definicji grafiki, efektów, animacji z kodem służącym do implementacji logiki biznesowej został rozwiązany dzięki wprowadzeniu języka XAML.

XAML to oparty na technologii XML deklaratywny język znaczników służący przede wszystkim do opisywania interfejsu użytkownika. Głównym zastosowaniem XAML jest uproszczenie tworzenia i inicjalizowania kontrolek oraz odwzorowania ich hierarchii.

XAML został stworzony z myślą o WPF, ale jest także używany do implementacji aplikacji w Silverlight, do opisywania dokumentów XPS oraz do definiowania przepływów pracy tworzonych przy użyciu Windows Workflow Foundation.

Składnia XAML oparta jest na XML a programuje się w tej technologii bardzo podobnie do tego, jak programuje się w HTML. Należy stworzyć główny element, zazwyczaj jest to okno (Window) lub strona (Page) i wewnątrz niego trzeba umieścić tagi reprezentujące pozostałe części interfejsu użytkownika, takie jak przyciski, etykiety czy pola tekstowe.

```
<Window x:Class="WPFTest.MainWindow" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="350" Width="525">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="60*" />
      <RowDefinition Height="251*" />
    </Grid.RowDefinitions>
    <StackPanel Grid.Row="0" Orientation="Horizontal">
      <Label Width="100"
          Content="Click me"
          Focusable="True"
          FontFamily="Tahoma"
          Foreground="#FF000064" />
      <Button Name="button"
           Height="30"
           Padding="3"
           VerticalAlignment="Top"
           Content="button"
           Click="OnclickHandler"
           BorderBrush="#FF004B00"
           Foreground="White"
           Background="#FF004B00" />
    </StackPanel>
  </Grid>
</Window>
```

Listing 4 Prezentujący przykładowy kod napisany w XAML.

Na Listing 4 Prezentujący przykładowy kod napisany w XAML zaprezentowany jest kod tworzący okno wraz kilkoma kontrolkami. Jak łatwo zauważyć zarówno rozmieszczenie

kontrolek, zawartość oraz wygląd zostały deklaratywnie zdefiniowane za pomocą ustawienia odpowiednich właściwości. Grid oraz StackPanel to kontenery przechowujące elementy UI. Są odpowiedzialne za zarządzanie rozmieszczeniem innych kontrolek. Warto zwrócić uwagę na sposób w jaki określany jest wygląd tych kontrolek. Został on zdefiniowany za pomocą atrybutów w tagach kontrolki, czyli w taki sam sposób w jaki definiowało się wygląd stron HTML, zanim powstał CSS. Technologia XAML wspiera również style, w których można definiować nie tylko wygląd, ale także zachowanie oraz animację elementów UI.

```
<Label  Style="{StaticResource LabelStyle}"
            Content="Click me"
             />
        <Button Name="button"
             Style="{StaticResource ButtonStyle}"
             Content="button"
             Click="OnclickHandler"   />
  <Application.Resources>
    <Style x:Key="LabelStyle" TargetType="Label">
      <Setter Property="Width" Value="100" />
      <Setter Property="Focusable" Value="True" />
      <Setter Property="FontFamily" Value="Tahoma" />
      <Setter Property="Foreground" Value="#FF000064" />
    </Style>

    <Style x:Key="ButtonStyle" TargetType="Button">
      <Setter Property="Height" Value="30" />
      <Setter Property="Padding" Value="3" />
      <Setter Property="VerticalAlignment" Value="Top" />
      <Setter Property="BorderBrush" Value="#FF004B00" />
      <Setter Property="Background" Value="#FF004B00" />
    </Style>
  </Application.Resources>
```

Listing 5 Definiowanie wyglądu kontrolek za pomocą stylów.

Na Listing 5 widoczny jest kod, który opisuje te same kontrolki co kod na Listingu 3 z tą różnicą, że tutaj style zdefiniowane są nie razem z kontrolkami tylko w wydzielonym miejscu. Takie podejście ma kilka zalet. Pierwszą z nich jest czytelność kodu, która jest znacznie lepsza w przypadku użycia nazwanych stylów. Kolejną zaletą jest możliwość zdefiniowania stylów, dodania ich do zasobów aplikacji, a później możliwość ich

wielokrotnego użycia. Aplikacje WPF mogą być tworzone zarówno za pomocą edycji pliku XAML, jak i za pomocą narzędzi wspierających tryb WYSIWYG. Do najpopularniejszych narzędzi przeznaczonych do tworzenia aplikacji WPF należą stworzone przez Microsoft Visual Studio oraz Expression Blend.

Oba narzędzia wspierają tryb WYSIWYG oraz drag&drop czyli możliwość umieszczania gotowych kontrolek w aplikacji, a później definiowania ich wyglądu oraz zachowania przy pomocy graficznych edytorów, bez konieczności pisania kodu XAML.

Expression Blend jest preferowane przez grafików ze względu na swoje rozbudowane funkcje graficzne oraz możliwość definiowania animacji, przejść i efektów graficznych bez konieczności pisania kodu. Visual Studio z kolei jest wybierane przez programistów, gdyż poza możliwością tworzenia prostego interfejsu użytkownika, pozwala pisać testy jednostkowe, posiada IntelliSense dla XAML oraz, przede wszystkim, jest narzędziem używanym do pracy z innymi technologiami .Net.

Podsumowując, XAML umożliwia szybkie i efektywne tworzenie atrakcyjnych pod względem grafiki aplikacji. Osiąga to poprzez separację kodu XAML od kodu C#. W ten sposób programiści mogą się zająć implementacją funkcjonalności, a graficy tworzeniem części graficznej, nie wchodząc sobie w drogę. Dzięki oddzieleniu kodu XAML i C# możliwe stało się stworzenie dedykowanych narzędzi zarówno dla programistów – Visual Studio, jaki i grafików – Microsoft Expression Blend. Separacja kodu XAML od stylów sprawia, że kod staje się czysty i czytelny, a raz zdefiniowane style mogą zostać wielokrotnie użyte w aplikacji.

Dużym minusem tej technologii jest składnia. Mimo tego, że XAML jest językiem deklaratywnym, programista musi napisać porównywalną ilość kodu jaką pisze programista używający języka imperatywnego.

2.2.4 SenseGUI

SenseGUI [1] jest deklaratywnym językiem dziedzinowym (ang. Domain Specific Language) opartym na Javie. Ideą przyświecającą temu pomysłowi jest stworzenie narzędzia, które ułatwi programiście zadanie tworzenia GUI dzięki automatyzacji tego procesu .

```
public class Person {
            private String firstName;
```

```
            private String lastName;
            private Date birthDate;
            private boolean higherEducation;
            private String remarks;
            private int SSN;
            private double annualIncome;
            public int getAge() {// [...]}
            }
```

Listing 6 Klasa użyta w senseGUI.

SenseGUI jest w stanie automatycznie wygenerować GUI bazując na modelu danych. Proces ten wygląda następująco: biblioteka skanuje obiekt języka Java przy użyciu refleksji i na podstawie jego atrybutów tworzy odpowiednie elementy GUI, takie jak pola tekstowe, etykiety, pole wyboru (ang. Combo box) itd. Gdyby tworzenie GUI było w pełni automatyczne, programista miałby bardzo nikłą kontrolę nad wynikowym interfejsem użytkownika.

W SenseGUI programista ma jednak duży wpływ na to, w jaki sposób biblioteka wygeneruje interfejs użytkownika. W wersji SenseGUI opisanej w artykule [1] konfiguracja bazuje na meta danych stworzonych za pomocą adnotacji. Na Listing 7 zaprezentowana jest klasa, która została udekorowana adnotacjami w celu utworzenia odpowiednich meta danych, które zostaną użyte do zbudowania interfejsu użytkownika.

```
@GUIGenerateAttribute(label ="First name", order = 1)
            private String firstName;
            @GUIGenerateAttribute(label ="Higher education",
widgetClass="mt.mas.GUI.CheckboxBoolean", order= 5)
            private boolean higherEducation;
            @GUIGenerateAttribute(label ="Remarks", order = 50,
widgetClass="javax.swing.JTextArea", scaleWidget=false)
            private String remarks;
            @GUIGenerateMethod(label = "Age",   showInFields = true, order =
4)
            public int getAge() { // ...}
```

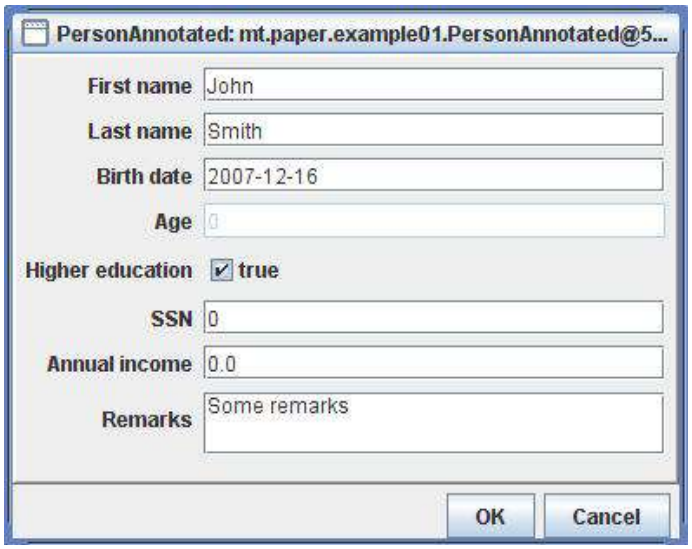
Listing 7 Klasa z adnotacjami senseGUI definiującymi GUI.

Na Rysunku 1 widoczna jest formatka utworzona przy pomocy adnotacji. Warto zauważyć, że część pól została wygenerowana na zasadzie konwencji, czyli etykieta pokrywa się z nazwą atrybutu, a JTextBox został użyty jako domyślna kontrolka do edycji. Reszta widgetów została wygenerowana na podstawie meta modelu, dla atrybutu reprezentującego wartość logiczną utworzono kontrolkę CheckboxBoolean, a dla długiej wartości tekstowej JTextArea.

Każdy atrybut klasy zostaje opatrzony informacją, jaką etykietę ma dostać kontrolka, jaka klasa ma być użyta do utworzenia odpowiedniej kontrolki oraz o kolejności w jakiej obiekt ma się pojawić na formularzu.

Cała biblioteka udostępnia 11 typów adnotacji. Poniżej znajdą się najważniejsze z nich:

- label, definiuje jaka etykieta zostanie dołączona do kontrolki reprezentującej obiekt,
- widgetClass, określa klasę, która będzie użyta do stworzenia kontrolki umożliwiającej edycję atrybutu lub metody,
- getMethod, setMethod definiuje metody umożliwiające czytanie oraz zapisywanie atrybutów. Domyślne wartość to nazwa atrybutu poprzedzona odpowiednim prefixem zgodnie z konwencją (get dla getterów oraz set dla setterów),
- order, określa wskazuje w jakiej pojawiają się kontrolki na formularzu,
- readOnly, ta adnotacja pozwala oznaczyć atrybut w wersji tylko do odczytu.



Rysunek 1 Formatka wygenerowana przy użyciu senseGUI.

W kolejnej wersji biblioteki – GUI Creation Langage [5] – zrezygnowano z adnotacji, zastępując je DSL. Według Mariusza Trzaski zastosowanie DSL ma kilka zalet. Przede wszystkim kod odpowiedzialny za tworzenie GUI znajduje się w jednym miejscu a nie, jak w przypadku adnotacji, w dwóch miejscach – w modelu oraz w miejscu, gdzie wywoływano API GCL. W przypadku użycia DSL nie ma potrzeby modyfikacji klas modelu domenowego, który po pierwsze może być niedostępny, a po drugie modyfikacja i wiązanie go z warstwą prezentacji jest niepożądane.

```
JFrame frame1 = create.frame.usingOnly(person);
```

Listing 8 Użycie języka GCL do stworzenia prostego formularza.

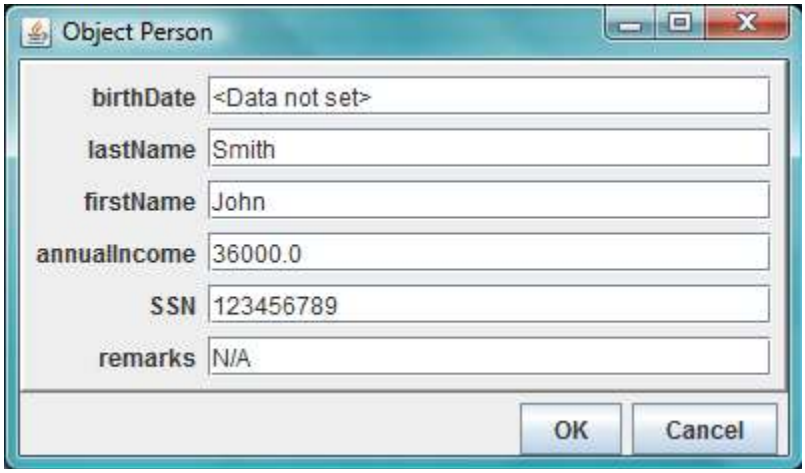
Na Listing 8 widoczny jest jednoliniowy kod tworzący formularz. Jest to możliwe dzięki użyciu odpowiednio dobranych wartości domyślnych. Jak można zaobserwować na Rysunek 2 formularz zawiera dla każdego atrybutu klasy Person etykietę z wartością odpowiadającą jego nazwie oraz pole tekstowe umożliwiające edycję jego wartości.

```
JFrame frame = create.
            frame.
            using(person).
```

```
            containing(
            attribute("firstName").as("First name"),
            attribute("lastName").validate(new ValidatorNotEmpty()),
            attribute("higherEducation"),
            method("getAge").as("Age"));
```

Listing 9 Użycie języka GCL do stworzenia formularza.

GCL zawiera także zbiór wyrażeń, które pozwalają programiście konfigurować sposób w jaki generowane jest GUI. Oprócz funkcjonalności zaimplementowanej w senseGUI czyli m.in. definiowane etykiet, klas kontrolek, które będą użyte do prezentowania danych w GCL programista ma także możliwość dodawania walidacji oraz prostego zaimplementowania lokalizacji.



Rysunek 2 Formularz wygenerowany przy użyciu kodu z Listing 8.

GCL znacznie podniósł poziom abstrakcji w porównaniu do wcześniej przedstawionych języków deklaratywnych MXML oraz XAML. Przy użyciu tej biblioteki programista nie jest zobligowany do dokładnego określenia jakie kontrolki mają zostać użyte do budowy GUI, musi on jedynie określić efekty, których oczekuje. Toteż spora część szczegółów implementacyjnych zostaje przed nim ukryta, co stanowi istotę języka deklaratywnego.

Ważną zaletą GCL jest odpowiednie zastosowanie refleksji oraz możliwość dobrania odpowiednich wartości domyślnych, dzięki czemu można stworzyć interfejs użytkownika za pomocą kilku linijek kodu. GCL posiada wbudowaną obsługę wielojęzyczności oraz

walidację, a ponadto łatwo konfigurowalną obsługę asocjacji pomiędzy klasami. Podczas tworzenia języka dziedzinowego dla GCL zrezygnowano z podejścia opartego na ciągach znaków, na rzecz DSL opartego na API. Takie podejście ma szereg zalet, między innymi mocną kontrolę typów oraz wsparcie dla wykrywania błędów podczas kompilacji programu. Mimo to GCL nie udało się całkowicie pozbyć użycia ciągów znaków w API. Na Listingu10 można zobaczyć, że nazwa atrybutu klasy Person jest przekazana jako ciąg znakowy, co często może prowadzić do błędów popełnianych przez programistów. Listing 10 przedstawia także podejście z użyciem wyrażeń lambda i języka C#, które eliminuje konieczność użycia łańcuchów znaków. Takie rozwiązanie nie było możliwe w GCL, gdyż JAVA nie wspiera wyrażeń lambda.

```
using(person).containing(  attribute("firstName") );

using(person).containing(  attribute( person=> person.firstName) );
```

Listing 10 Porównanie składni z ciągiem znaków oraz lambdą jako argumentem.

2.2.5 ASP.NET MVC

Jedną z podstawowych nowości wprowadzonych do ASP.NET MVC 2 [6] były szablony oraz meta model oparty na atrybutach. Jest to funkcjonalność podobna do Dynamic Data, opisanego w rozdziale 2.2.2. Funkcjonalność polega na tym, że Asp.Net jest w stanie automatycznie wygenerować kontrolki do wyświetlania i edycji modelu. Informacje definiujące GUI, czyli meta model, są zdefiniowane za pomocą atrybutów, co można zobaczyć na Listing 11 Klasa udekorowana atrybutami UI.

```
public class NewUserModel
    {
        [Required]
        [DisplayName( "Nazwa użytkownika")]
        public string UserName { get; set; }

        [Required]
        [DataType(DataType.EmailAddress)]
        [DisplayName( "Adres email")]
        public string Email { get; set; }
```

```
        [Required]
        [DataType(DataType.Password)]
        [DisplayName("Hasło")]
        public string Password { get; set; }

        [DataType(DataType.Password)]
        [DisplayName("Potwiedź hasło")]
        public string ConfirmPassword { get; set; }

        [DisplayName("Czy firma")]
        public bool IsCompany{ get; set; }
    }
```

Listing 11 Klasa udekorowana atrybutami UI.

Pierwsze podejście wykorzystujące meta model polega na użyciu metod LabelFor, EditorFor oraz ValidationMessageFor dla każdego atrybutu obiektu, które wygenerują odpowiednio etykietę, kontrolkę adekwatną dla typu oraz, w razie potrzeby, błędy walidacji. Jak widać na Listingu 12 takie podejście, nie oszczędza programiście pisania dużych ilości kodu, aczkolwiek daje możliwość dowolnego ostylowania strony przy pomocy CSS.

```
            <div class="editor-label">
                <%: Html.LabelFor(model => model.UserName) %>
            </div>
            <div class="editor-field">
                <%: Html.EditorFor(model => model.UserName)%>
                <%: Html.ValidationMessageFor(model => model.UserName) %>
            </div>
                .
                . ominięto 30 analogicznych lini kodu
                .
            <div class="editor-label">
                <%: Html.LabelFor(model => model.IsCompany) %>
            </div>
            <div class="editor-field">
                <%: Html.EditorFor(model => model.IsCompany)%>
                <%: Html.ValidationMessageFor(model => model.IsCompany)%>
            </div>
```

Listing 12 Okrojony kod widoku aspx.

Drugie podejście polega na użyciu metody EditorForModel, która generuje to samo, co programista w pierwszym podejściu, czyli etykietę, kontrolkę do edycji oraz błędy walidacyjne dla każdego atrybutu modelu. Jak widać na Listing 13, jedna linijka zastąpiła kilkadziesiąt linijek kodu generującego UI. Ograniczaniem takiego podejścia jest fakt, że trudno zaaplikować zawansowane style CSS do tak wygenerowanego kodu. Jednak takie rozwiązanie nadaje się do typowych formularzy do edycji i wyświetlania danych, gdzie nie ma dużego nacisku na wygląd aplikacji, a wręcz prosty layout bywa pożądany.

```
<%: Html.EditorForModel()%>
```

Listing 13 Kod widoku opierający się o konwencje.

Jak zostało wcześniej wspomniane w tym rozdziale, dla każdego typu generowana jest odpowiednia kontrolka do edycji oraz wyświetlania danych. Domyślnie w Asp.Net jest zdefiniowanych kilka szablonów:

- Pole tekstowe dla stringów,
- Hiperłącze dla adresów Url,
- Pole tekstowe dla wartości liczbowych z wartością sformatowaną odpowiednio dla ustawień językowych użytkownika,
- Checkbox dla wartości logicznych.

Jest to rozwiązanie bardzo ubogie i niewystarczające dla większości rozwiązań. Na szczęście twórcy Asp.Net umożliwią bardzo łatwe dodanie obsługi nowych typów danych. Wystarczy w odpowiednim folderze utworzyć plik widoku o nazwie odpowiadającej typowi np. DateTime.aspx dla daty. W takim pliku należy zdefiniować jak ma być wyświetlany oraz edytowany dany typ. Używając ostatniego przykładu, typu data, dla edycji można by zastosować kontrolkę jQuery UI Datepicker, a dla wyświetlania można użyć tagu z odpowiednim formatowaniem.

2. Koncepcja rozwiązania

W niniejszym rozdziale opisano motywację przyjętego rozwiązania oraz wyszczególniono wymagania funkcjonalne, które powstały po analizie słabych i mocnych stron rozwiązań opisanych w rozdziale 2.2. Następnie opisano, w jaki sposób zostały zaimplementowane wymagania rozwiązania. Rozdział zawiera także opis składni języka do deklaratywnego tworzenia GUI oraz kilka przykładów jego zastosowania.

3.1 Propozycja nowego rozwiązania

W tej pracy autor proponuje stworzenie frameworka do deklaratywnego budowania GUI, z możliwością rozbudowy jego funkcjonalności za pomocą rejestrowania rozszerzeń do obsługi różnych typów danych. Framework miałby zastosowanie przy prezentacji oraz edycji dowolnych typów danych, a w szczególności nadawałby się do programowania danocentrycznych (ang. data-centric) aplikacji biznesowych. Rozwiązanie umożliwiałoby łatwą implementację rozszerzeń do obsługi nowych typów danych, dzięki czemu programista mógłby dostosować framework do specyficznych wymagań aplikacji.

Oprócz deklaratywnego tworzenia odpowiednich widgetów framework pozwalałby na łatwe dodawanie reguł walidacyjnych oraz internacjonalizację interfejsu użytkownika.

3.2 Motywacja przyjętego rozwiązania

Praca programisty składa się zarówno z kreatywnej część polegającej na projektowaniu architektury i implementacji przeróżnych algorytmów, jak i mniej kreatywnej, czyli programowania interfejsu użytkownika. W szczególności odnosi się to do komponentów

prezentujących oraz umożliwiających edycję danych. Celem pracy jest stworzenie frameworka, który umożliwi szybkie, deklaratywne tworzenie graficznego interfejsu użytkownika.

Deklaratywne podejście do tworzenia interfejsu użytkownika to także skrócenie czasu potrzebnego do stworzenia aplikacji, jak i redukcja błędów, które programista może popełnić podczas imperatywnego dodawania dużej ilości kontrolek w celu zbudowania komponentów GUI. Proponowane rozwiązanie wprowadzi także warstwę abstrakcji pomiędzy logiką biznesową a interfejsem użytkownika. Takie rozwiązanie uniezależnia aplikację od platformy, gdyż programista tworząc GUI, nie odnosi się do konkretnej technologii tylko korzysta z deklaratywnego języka, który ukrywa szczegóły implementacji specyficzne dla danego API. Dzięki wprowadzeniu poziomu abstrakcji przed GUI, istnieje jedno miejsce, w którym są definiowane konwencje i globalne ustawienia decydujące jak będzie wyglądał interfejs użytkownika. Dzięki temu zmiana wyglądu całej aplikacji będzie znacznie mniej czasochłonna.

3.3 Wymagania funkcjonalne

Po analizie mocnych i słabych stron technologii do deklaratywnego tworzenia GUI nakreślono wymagania funkcjonalne i nie funkcjonalne frameworka:

- Język ma być zgodny z podejściem deklaratywnym.
- Zwięzła i samo tłumacząca się składnia.
- Język ma zachować silną kontrolę typów i nie może używać wpisanych na stałe wartości (ang. hard-coded).
- Możliwość dodawania reguł walidacyjnych.
- Możliwość internacjonalizacji elementów GUI.
- Język ma umożliwiać tworzenie interfejsu z najprostszymi elementami GUI (pola tekstowe, listy rozwijane, przyciski itd.).
- Wykorzystanie konwencji do tworzenia widgetów, aby zminimalizować nakład pracy programisty.
- Ma pozwalać na rejestrację własnych rozszerzeń, jeśli możliwości rozwiązania okażą się niewystarczające.

3.4 Opis rozwiązania

W poprzednim rozdziale zidentyfikowano podstawowe wymagania funkcjonalne i nie funkcjonalne. W trakcie prac projektowych napotkano na następujące problemy koncepcyjne:

- Jak odczytać strukturę klasy?
- Jak sprawić, aby ilość wymaganych konfiguracji, była sprowadzona do minimum?
- Jak połączyć generowane kontrolki z danymi?
- Jak zachować silną kontrolę typów i nie używać przekazywanych w obiekcie String wartości?

Pierwszy problem wygląda w sposób następujący. Po pierwsze programista przekazuje model do biblioteki, gdzie zostaje on przeanalizowany pod względem publicznych danych. Po drugie model musi zostać sklonowany, aby framework, nie operował na oryginalnym obiekcie tylko na jego kopii. Ten problem został rozwiązany przy pomocy mechanizmu refleksji, czyli zdolności programu do odczytywania i modyfikowania swojego stanu w trakcie wykonania.

Jak wcześniej wspomniano framework powstał w celu skrócenia czasu programowania, toteż redukcja koniecznej ilość konfiguracji do minimum jest zadaniem o najwyższym priorytecie. Zazwyczaj podczas budowania formatki programista tworzy odpowiednie kontrolki dla poszczególnych pól modelu biznesowego. W trakcie dobierania kontrolki do odpowiedniego pola modelu programiści kierują się zdrowym rozsądkiem, a nie dokumentacją techniczną, w której bardzo często nie ma takich szczegółów. Wydaje się więc, że jest to odpowiednie pole do zastosowania strategii konwencji ponad konfigurację *(ang. Convention over configuration)* polegającej na stworzeniu domyślnych kontrolek dla odpowiednich typów danych, co wygląda następująco:

- Pole tekstowe dla wartości tekstowych.
- Przycisk wyboru (ang. checkbox) dla wartość logicznych.
- Lista wyboru (ang. combobox) dla typu wyliczeniowego (ang. enumeration).
- Datepicker dla typów oznaczających datę.
- Tabela dla listy wartości.

Za pomocą konwencji został także zaimplementowany mechanizm internacjonalizacji etykiet. Rozwiązanie nie wymaga od programisty podawania klucza definiującego tłumaczenie. Wystarczy podać typ zasobu, a biblioteka wyszuka zasób używając nazwy atrybutu modelu. Jeśli zasób zostanie znaleziony, będzie on użyty jako etykieta. W innym przypadku wartość domyślna, czyli nazwa atrybutu, zostanie użyta jako etykieta.

Zastosowanie konwencji sprawia, że rozwiązanie jest generyczne i może być wykorzystane w dużej części przypadków. Jednak nie zawsze programista chce wyświetlić wszystkie pola modelu, często potrzebne są reguły walidacyjne. Kolejny problem do rozwiązania to sposób definiowania GUI, gdy ustawienia domyślne nie są adekwatne. Kwestię budowania definicji interfejsu można rozwiązać na kilka sposobów. Można to zrobić za pomocą zewnętrznego pliku konfiguracyjnego w XML, adnotacji lub dedykowanego API.

Główną charakterystyką adnotacji jest bliskość składowania definicji GUI i modelu, a także fakt, że nie potrzeba dodatkowych plików. Pojawienie się adnotacji w pliku modelu może być odczytywane zarówno jako przydatny komentarz. jak niepotrzebne zaśmiecanie warstwy biznesowej informacjami o GUI. Rozwiązanie z użyciem XML pozwala odseparować warstwę biznesową od definicji GUI, poprzez umieszczenie konfiguracji w zewnętrznym pliku. Plik XML jest rozwiązaniem, które sprawia, że dla jednego modelu można mieć tylko jedną definicję GUI, co byłoby niepożądanym efektem. Kolejną wadą zastosowania XML jest kiepskie wsparcie popularnych IDE np. Visual Studio dla edycji XML co wyraża się w braku IntelliSense.

Oba powyższe rozwiązania nie oferują wsparcia dla refaktoringu kodu ani wykrywania błędów składniowych w trakcie kompilacji. Kolejne rozwiązanie – dedykowane API – łączy zalety poprzednich rozwiązań. Pozwala odseparować warstwę biznesową od warstwy prezentacyjnej oraz daje możliwość kilku definicji GUI dla jednego modelu. Ponadto API daje takie zalety jak możliwość refaktoringu kodu oraz wykrywania bledów w trakcie kompilacji.

Wykrywanie błędów w trakcie kompilacji kodu, a nie w trakcie wykonania programu, jest możliwe dzięki zachowaniu silnej kontroli typów i zrezygnowaniu z przekazywania argumentów za pomocą stringów. By to osiągnąć rozwiązanie korzysta z wyrażeń lambda oraz drzewa wyrażeń (ang. expression tree), aby przekazać nazwę atrybutu jako argument do API.

Zwięzła składnia oraz czytelność została osiągnięta, tak samo jako w GLC, przy zastosowaniu tak zwanego płynnego interfejsu (ang. Fluent interface). Rozwiązanie te polega na łączeniu wywołań metody w łańcuch metod, co jest możliwe dzięki odpowiedniemu zaprojektowaniu API, gdzie każda metoda wykonująca operację, zawraca this zamiast void.

3.5 Składnia

Składnia ma charakteryzować się przede wszystkim prostotą i czytelnością. Można wyróżnić następujące słowa kluczowe:

- Window – tworzenie kontrolek w niezależnym oknie.
- Panel – tworzenie kontrolek w panelu.
- Using – służy do przekazania modelu do biblioteki.
- Containg – służy do określenia atrybutów, które pojawią się na formatce.
- Attribute – służy do określenia, które pole modelu jest aktualnie definiowane.
- Label – służy do określenia etykiety pola.
- Range – walidator zakresu.
- Required –walidator dla pola wymaganego.
- MaxLenght _ walidator maksymalnej długości pola.
- Regex – walidator wyrażeń regularnych.
- Email – walidator dla pola email.
- Must – walidator, przyjmujący delegat jako argument.
- Show – służy do pokazania okna z kontrolkami.
- Build – służy do stworzenie panelu z kontrolkami.
- UseResources – służy do wskazania klasy z zasobami.

3.6 Przykłady

W tym rozdziale zostaną zaprezentowane przykłady zastosowania deklaratywnego frameworka do tworzenia GUI. Wszystkie przykłady zostały stworzone przy zastosowaniu dwóch klas – Person oraz Address widocznych na Listing 14.

```
public class Person {
        public string FirstName { get; set; }
        public string LastName { get; set; }
```

```
        public int? Age { get; set; }
        public bool IsMarried { get; set; }
        public Education Education { get; set; }
        public DateTime DataOfBirth { get; set; }
        public IList<Address> Address { get; set; }
        public Image Photo { get; set; }
    }
    public class Address {
        public string City { get; set; }
        public string Street { get; set; }
        public int Number { get; set; }
        public bool IsHomeAddress { get; set; }
    }
```

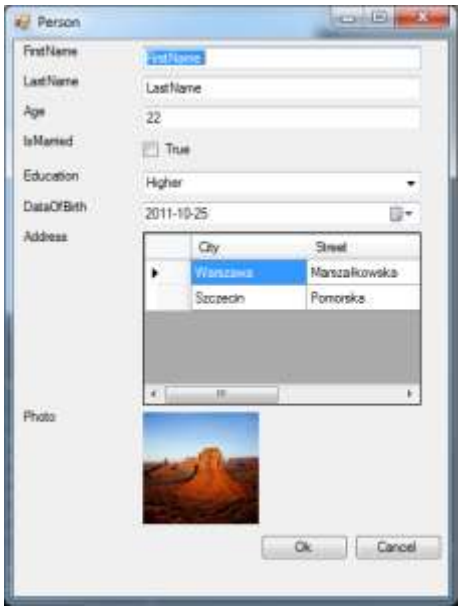
Listing 14 Modele użyte do przykładów zastosowań biblioteki.

Kod z Listing 15 przedstawia najprostszy sposób wykorzystania biblioteki. Polecenie rozpoczyna się od słowa Window, powodującego powstanie kontrolek w niezależnym oknie. W kolejnym wyrażeniu – Using – przekazywany jest model, na podstawie którego tworzy się kontrolki. Ostatnia komenda – Show – oznacza zakończenie procesu definiowania GUI.

```
        Window
        .Using(person)
        .Show();
```

Listing 15 Najprostszy przykład użycia biblioteki.

Na Rysunku 3 widoczne jest okno powstałe po wykonaniu kodu z Listing 15. Należy zwrócić uwagę, że dla wszystkich atrybutów klasy powstaje etykieta oraz odpowiednia kontrolka do edycji danych. W tym przykładzie widoczne są pola tekstowe dla wartości tekstowych, checkbox dla wartości logicznej, lista wyboru dla atrybutu wyliczeniowego oraz kontrolka do wyboru daty. Dwie kolejne kontrolki to tabela dla kolekcji oraz obraz dla typu Image. Na samym dole okna widoczne są dwa domyślne przyciski służące do akceptowania lub anulowania zmian dokonanych na edytowanym obiekcie.

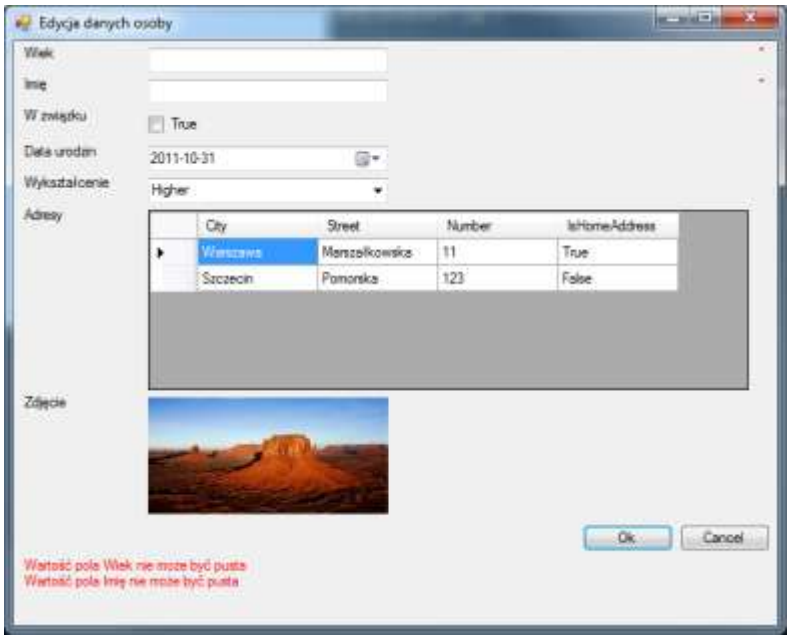


Rysunek 3 Formatka powstała w wyniku wykonania kodu Listing 15.

```
Window
.Using(person)
.Containing(
    e => e.Attribute(a => a.Age).Label("Wiek").Range(0, 120).Required(),
    e => e.Attribute(a =>
a.FirstName).Label("Imię").MaxLength(10).Required(),
    e => e.Attribute(a => a.IsMarried).Label("W związku małżeńskim"),
    e => e.Attribute(a => a.DataOfBirth).Label("Data urodzin"),
    e => e.Attribute(a => a.Education).Label("Wykształcenie"),
    e => e.Attribute(a => a.Address).Label("Adresy").Width(500),
    e => e.Attribute(a => a.Photo).Label("Zdjęcie"))
.Title("Edycja danych osoby")
.Show();
```

Listing 16 Kod przedstawiający zaawansowane użycie walidatorów.

Na Listing 16 przedstawione są kolejne możliwości biblioteki. Metoda Containing pozwala przekazać listę atrybutów modelu, które będą widoczne na formularzu. Wszystkie etykiety zostały zdefiniowane przy pomocy metody Label. W powyższym przykładzie zostały użyte trzy walidatory. Walidator zakresu stworzony został przy użyciu metody Range, walidator maksymalnej długości pola był dodany metodą MaxLength, a walidator dla pola wymaganego stworzono przy pomocy wyrażenia Required.



Rysunek 4 Formatka powstała w wyniku wykonania kodu z Listing 16.

Na Rysunek 4 widoczna jest formatka powstała po wykonaniu kodu z Listing 16. Wszystkie kontrolki zostały zbudowane na podstawie konwencji. Jedynie długość tabeli została dostosowana przy pomocy metody Width, oraz tytuł okna przy pomocy metody Title. Po wciśnięciu przycisku Zapisz, przed faktycznym zapisaniem danych, następuje walidacja. Błędy walidacyjne są wyświetlana na dwa sposoby. Po pierwsze, wiersz z błędnymi danymi jest oznaczany czerwoną gwiazdką, po drugie, na dole formatki wyświetlany jest komunikat błędu przypisany do walidatora.

```
Window
.Using(person)
.Must(model => model.IsMarried && model.Age < 18 ?
"Za młody na ślub" : null)
.UseResources<Resource>()
.Show();
```

Listing 17 przedstawia zaawansowane użycie piku zasobów i walidatora biznesowego.

Na Listing 17 widoczne jest zastosowanie walidatora reguł biznesowych, który został dodany za pomocą metody Must. Walidator ten umożliwia zdefiniowanie przy użyciu wyrażenia lambda dowolnej reguły dla modelu oraz podanie komunikatu błędu, który

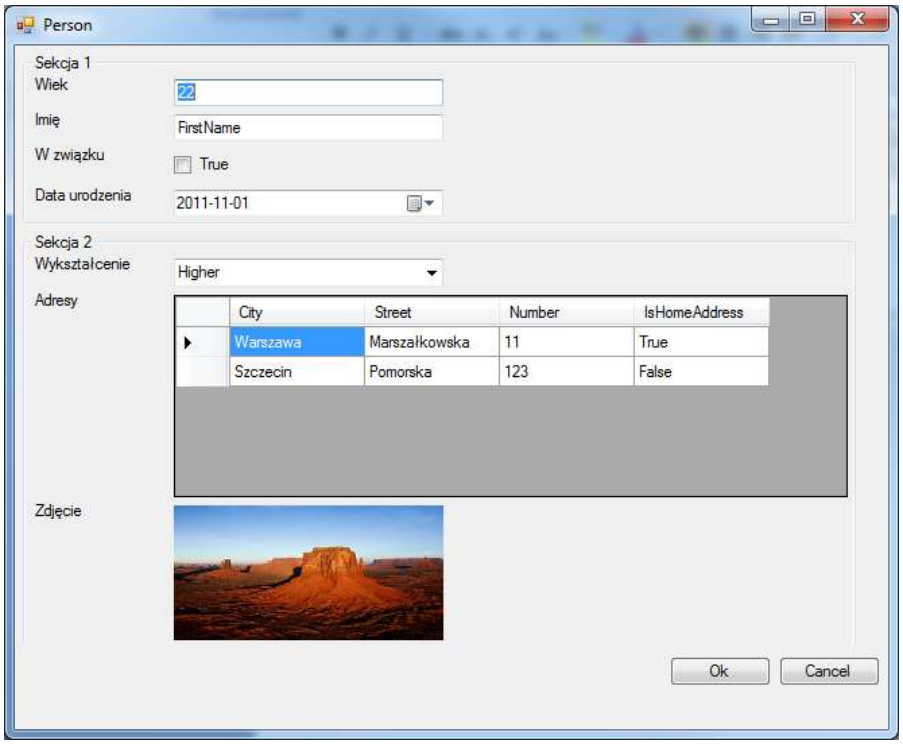
zostanie wyświetlony w dolnej części ekranu. Przy użyciu metody UseResources została określona klasa zasobów, która będzie użyta do internacjonalizacji interfejsu użytkownika.

```
Window.Using(person)
.Group(
  "Sekcja 1",
  e => e.Attribute(a => a.Age).Range(0, 120).Required(),
  e => e.Attribute(a => a.FirstName).MaxLength(10).Required(),
  e => e.Attribute(a => a.IsMarried),
  e => e.Attribute(a => a.DataOfBirth)
 )
.Group(
  "Sekcja 2",
  e => e.Attribute(a => a.Education),
  e => e.Attribute(a => a.Address).Width(500),
  e => e.Attribute(a => a.Photo)
)
.UseResources<Resource>()
.Show();
```

Listing 18 kod prezentujący grupowanie kontrolek w sekcje.

Na Listing 18 widoczny jest przykład wykorzystania biblioteki do grupowania kontrolek w nazwane sekcje. Sekcja taka jest stworzona przy pomocy metody Group przyjmującej w argumencie nazwę sekcji oraz atrybuty modelu, które wejdą w jej skład.

Tak jak widać na Rysunek 5, funkcjonalność grupowania sekcji, służy do wizualnego podzielenia tematycznie pasujących do siebie kontrolek. Kolejność jest zgodna z kolejnością zdefiniowania sekcji w kodzie.



Rysunek 5 przedstawiający pogrupowane kontrolki.

```
ControlBuilderRegistry.Instance.Register(
typeof(Image), new ImageControlBuilder());
```

Listing 19 prezentuje rozszerzania rozwiązania o nowe kontrolki.

Kolejną ważną funkcjonalnością, która została zaprezentowana na Listing 19 jest możliwość dodawania własnych rozszerzeń. Dokonać tego można wywołując metodę Register oraz podając typ oraz klasę, która będzie odpowiedzialna za tworzenie kontrolki do edycji oraz wyświetlana typu.

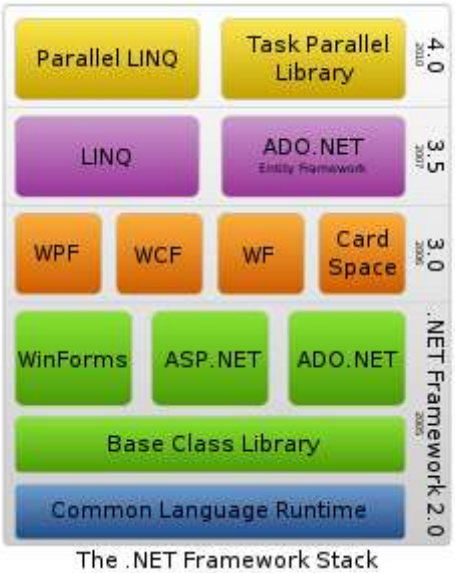
3. Opis technologii użytych w pracy

W tym rozdziale zostały przedstawione użyte podczas implementacji rozwiązania:

- Technologie – .Net, C# Windows Forms.
- Narzędzia – Visual Studio, Resharper.
- mechanizmy – refleksja, płynny interfejs, drzewo wyrażeń.

4.1. .Net

.Net Framework to biblioteka wspierająca tworzenie oprogramowania do systemów z rodziny Windows. Składa się ze środowiska uruchomieniowego Common Language Runtime, standardowej biblioteki Base Class Library zawierającej podstawowe funkcje, takie jak operacje I/O, kolekcje, wielowątkowość, refleksja czy manipulacja plikami XML. Zgodnie z Rysunek 6 w skład wersji .Net 2.0 wchodziły także biblioteka Windows Forms do tworzenia aplikacji okienkowych, ASP.Net do programowania aplikacji internetowych oraz ADO.NET umożliwiający dostęp do baz danych. Wersja .Net 3.0 została uzupełniona o silnik graficzny Windows Presentation Foundation, usługi sieciowe Windows Communication Fundation oraz, wycofany już, system do zarządzana tożsamością Card Space. Kolejna wersja .Net 3.5 Framework została wzbogacona o silnik zapytań do obiektów LINQ oraz mapper obiektowo-relacyjny Entity Framework. W wersji 4.0 .Net Framework pojawiły się rozszerzenia do programowania równoległego Parallel LINQ oraz Task Parallel Library.



Rysunek 6 .Net Framework.

W środowisku .Net języki [7] są kompilowane do języka pośredniego Microsoft Intermediate Language. W czasie wykonania programu, Common Language Runtime zamienia kod pośredni na kod maszynowy procesora, na którym wykonywana jest aplikacja.

Środowiskiem uruchomieniowym dla .Net jest Common Language Runtime. CLR jest odpowiedzialny za:

- zarządzanie wyjątkami,
- zarządzanie pamięcią przy użyciu mechanizmu odśmiecania pamięci (ang. garbage colector),
- kontrolę typologiczną (ang. type safety).

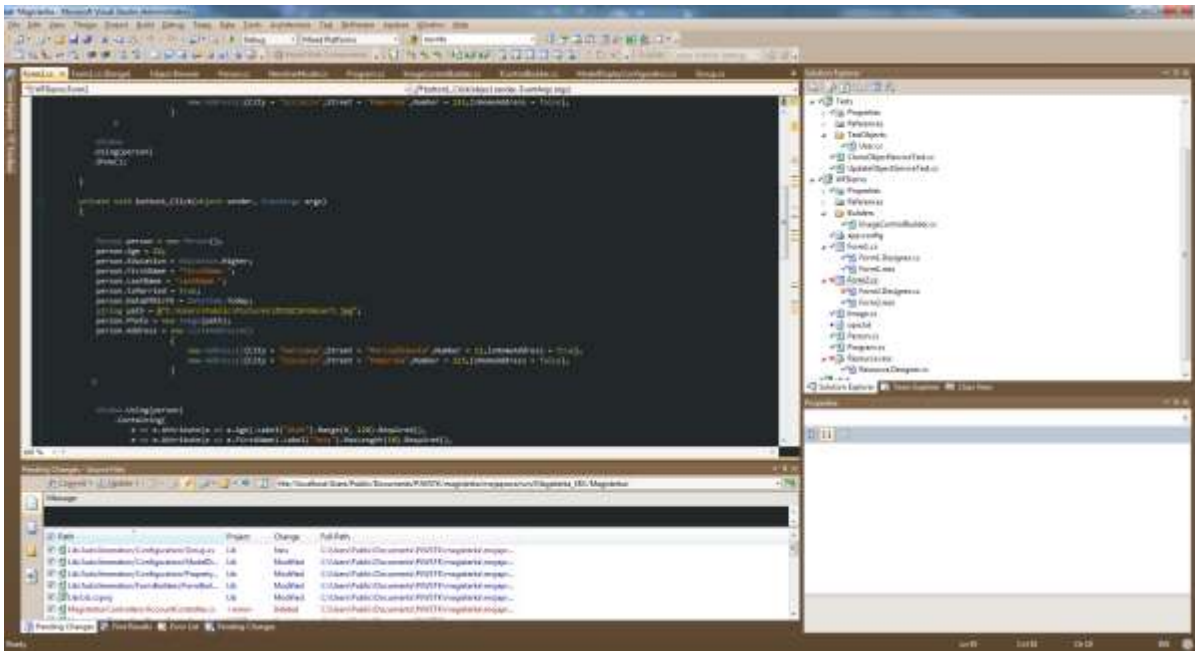
4.2. Visual Studio 2010

Visual Studio 2010 widoczne na Rysunek 7 to zintegrowane środowisko programistyczne (ang. Integrated Development Environment, IDE) służące do tworzenia aplikacji opartych o technologię .Net oraz języki C#, C++, VB.Net, IronPython i IronRuby.

Visual Studio wspiera następujące typy projektów:

- aplikacje internetowe – ASP.NET, ASP.NET MVC, ASP.NET Dynamic Data oraz Silverlight, a także web serwisy WCF (Windows Communication Foundation),
- aplikacje desktopowe dla systemu Windows – Windows Forms oraz WPF (Windows Presentation Fundation),
- rozszerzenia dla rodziny produktów Microsoft Office,
- aplikacje mobilne Windows Mobile oraz Windows Phone.

Visual Studio 2010 posiada edytor kodu z funkcją podpowiadania kodu IntelliSense, która znacznie zwiększa wydajność pracy programistów. Visual Studio daje także możliwość pisania i uruchamiania różnych rodzajów testów m.in. jednostkowych, obciążeniowych oraz testów UI dla WPF. Visual Studio współpracuje z wieloma systemami kontroli wersji m.in. Team Fundation Server, Visual SourceSafe oraz z SVN, poprzez wtyczkę AnkhSVN widoczną na Rysunek 7.



Rysunek 7 Visual Studio 2010.

4.3. C#

C# to obiektowy język programowania stworzony przez zespół Microsoft pod kierownictwem Andersa Hejlsberga. C# charakteryzuje się brakiem wielodziedziczenia, z hierarchią klas o jednym elemencie nadrzędnym klasy System.Object. Programista piszący w C#, w przeciwieństwie do programisty C++ nie musi zajmować się zarządzaniem pamięcią, gdyż C# korzysta z garbage collectora, mechanizmu wbudowanego w CLR do odśmiecania pamięci.

Język C# posiada delegaty oraz zdarzenia (ang. event), odpowiedniki wskaźników na funkcję C+. Delegaty w C# różnią się od wskaźników tym, że zapewniają bezpieczeństwo typów. Delegaty osiągają to poprzez składowanie następujących informacji:

- adres metody do której się odnoszą,
- informację o parametrach metody,
- typ zwracany przez metodę.

Podczas projektowania języka C# położono nacisk na produktywność programistów i minimalizację liczby miejsc, w których można popełnić błędy. Poza wspomnianym zarządzaniem pamięcią C# ułatwia życie programistom dzięki poniższym funkcjom:

- silna kontrola typów,
- wykrywanie próby użycia niezainicjalizowanych zmiennych,
- sprawdzanie zakresu (ang. array bounds checking),
- wykrywanie martwego kodu (ang. unreachable code).

Najnowsza wersja języka C# to 4.0. Kolejne wersje języka pojawiały się razem z nowymi wersjami .Net frameworka.

Wersja 2.0 wprowadziła następujące ulepszenia:

- typy oraz kolekcje generyczne (ang. generics),
- klasy częściowe (ang. partial class),
- typy nullowalne,
- metody anonimowe.

Kolejna wersja 3.0 wprowadziła jeszcze więcej zmian, oto kilka z nich:

- typy domniemane (ang. Implicitly typed local variable) oraz wyrażenie var,
- typy anonimowe,
- metody rozszerzające (ang. Extension methods),
- wyrażenia lambda,
- drzewa wyrażeń (ang. Expression trees),
- LINQ, czyli możliwość tworzenia zapytań do obiektów.

Aktualna wersja 4.0 została rozszerzona o:

- obsługę kowariancji oraz kontra wariancji,
- opcjonalne parametry,
- obsługę języków dynamicznych oraz wyrażenie dynamic.

4.4. Wyrażenia lambda

Wyrażenia lambda (ang. lambda Expression) to anonimowe funkcje ze zwięzłą składnią, które mogą być użyte do tworzenia delegatów oraz drzew wyrażeń. Wyrażenia lambda są następcami funkcji anonimowych (ang. anonymous functions) korzystających z wyrażenia delegate. Na Listing 20 zaprezentowany jest delegat przyjmujący dwie liczby w parametrach oraz zwracający wartość logiczną. Zostaje on zainicjalizowany na dwa sposoby, poprzez funkcję anonimową oraz przez wyrażenie lambda. Jedyną zaletą wyrażenia lambda jest znaczne skrócenie składni, która składa się z operatora => oraz lewej strony oznaczającej parametry, oraz prawej, zawierającej wyrażenie lub blok kodu.

```
private delegate bool  Delegate(int x, int y);
Delegate myDelegate = delegate(int x, int y) { return x > y; };
Delegate myDelegate2 = (x, y) => x > y;
```

Listing 20 Porównanie delegatów i wyrażeń lambda.

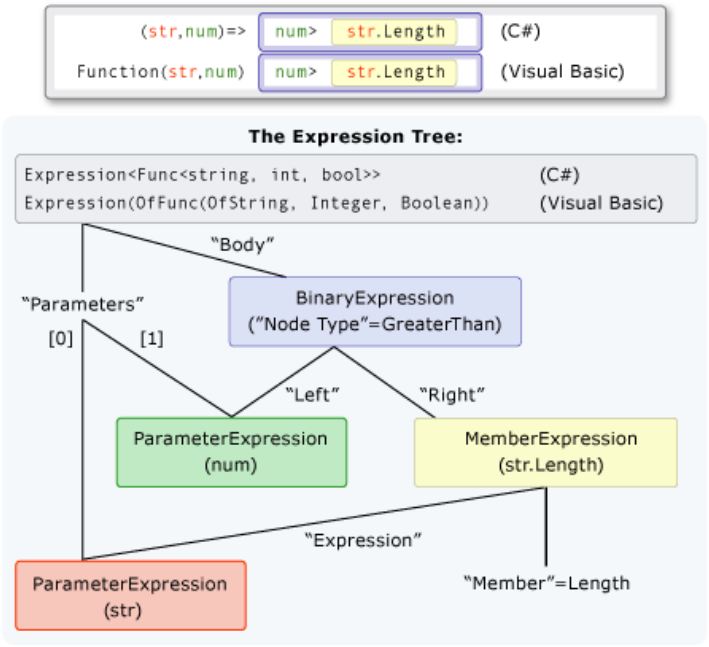
4.5.Drzewa wyrażeń

Drzewo wyrażeń to struktura danych, gdzie każdy węzeł jest wyrażeniem, jak na przykład wyrażenie binarne x>y. Na Listing 21 widoczne jest drzewo wyrażeń składające się z lambdy będącej funkcją, która sprawdza czy liczba podana w argumencie jest większa od długości tekstu przekazanego jako drugi argument.

```
Expression<Func<string, int, bool>> lambda =
(str, num) => num > (str.Length);
```

Listing 21 Przykład prostego drzewa wyrażeń [8].

Wyrażenia takie można budować dynamicznie, kompilować, a następnie wykonywać jak zwykłe wyrażenia lambda. Drugą operacją, którą można wykonać na wyrażeniu jest rozebranie go na części pierwsze. Jak widać na Rysunek 8 wyrażenie składa się z parametrów oraz ciała wyrażenia. Same ciało jest wyrażeniem binarnym „większe niż”, gdzie po lewej stronie jest stała wartość, a po prawej wyrażenie zwracające atrybut obiektu (ang. member access).



Rysunek 8 Graficzna reprezentacja drzewa wyrażeń [8].

Taka możliwość przejścia po drzewie wyrażeń jest ciekawą alternatywą dla mechanizmu refleksji. Zamiast przekazywać nazwę atrybutu klasy w ciągu znaków i używać refleksji do pobrania metadanych, można użyć wyrażenia polegającego na dostępie do atrybutu, co widać na Listing 22.

```
SomeMethod("SomeProperty");
SomeMethod(e => e.SomeProperty);
```

Listing 22 Porównanie zastosowania refleksji i drzewa wyrażeń.

4.6. Płynny interfejs

Płynny interfejs (ang. fluent interface) to interfejs programistyczny, który sprawia, że kod jest bardziej czytelny i zwięzły. Jak widać na Listing 23 płynny interfejs implementuje się wykonując zamierzoną operację wewnątrz metody, a następnie zwracając this.

```
public class Person
{
    private string _name;
    private int _age;
```

```
    public static Person New()
    {
        return new Person();
    }

    public Person Name(string name)
    {
        _name = name;
        return this;
    }

    public Person Age(byte age)
    {
        _age = age;
        return this;
    }

    public Person Describe()
    {
        Console.WriteLine("Person: name - {0} and {1} years old.", _name,
_age);
        return this;
    }
}
```

Listing 23 Implementacja płynnego interfejsu.

Na Listing 24 zaprezentowany jest kod, który tworzy nowy obiekt, ustawia dwa atrybuty na nowej instancji oraz wywołuje jedną metodę. Klasyczna implementacja zajmuje cztery linijki kodu, podczas gdy, implementacja z płynnym interfejsem tylko jedną linię. Minusem zastosowania płynnego interfejsu jest to, że cały ciąg poleceń jest jednym wywołaniem, przez co debugowanie takiego kodu jest znacznie utrudnione.

```
var  person = new Person();
  person.Name = "Andrzej";
  person.Age  = 22;
  person.Describe();
```

```
var personFluent = Person.New().Name("Andrzej").Age(22).Describe();
```

Listing 24 Porównanie zwykłego kodu i płynnego interfejsu.

4.7. Refleksja

Refleksja to zdolność kodu do identyfikacji i modyfikacji struktury oraz zachowania programu w trakcie jego wykonania. Gdy kod .Net jest kompilowany, meta dane o przestrzeniach nazw (ang. namespace), typach, oraz metodach są wytwarzane i składowane w assembly. Net posiada rozbudowane API, znajdujące się w System.Reflection, które pozwala pobrać assembly w trakcie działania programu i wykonać na nim wiele operacji m.in. pobrać wszystkie znajdujące się w nim typy. Na Listing 25 widać przykład mechanizmu późnego bindowania (ang. late binding), który pozwala na stworzenie instancji klasy, nie wiedząc w trakcie kompilacji jaki będzie jej typ. Na Listing 25 zaprezentowane jest dynamiczne wywołanie metody, przy pomocy funkcji invoke, która otrzymuje nazwę metody w argumencie.

```
Assembly myAssembly = Assembly.GetExecutingAssembly();
Type personType = myAssembly.GetType("MyNamespace.Person");
object person = Activator.CreateInstance(personType);
MethodInfo methodInfo= classType.GetMethod("GetName");
string personName= (string) mi.Invoke(person, null);
```

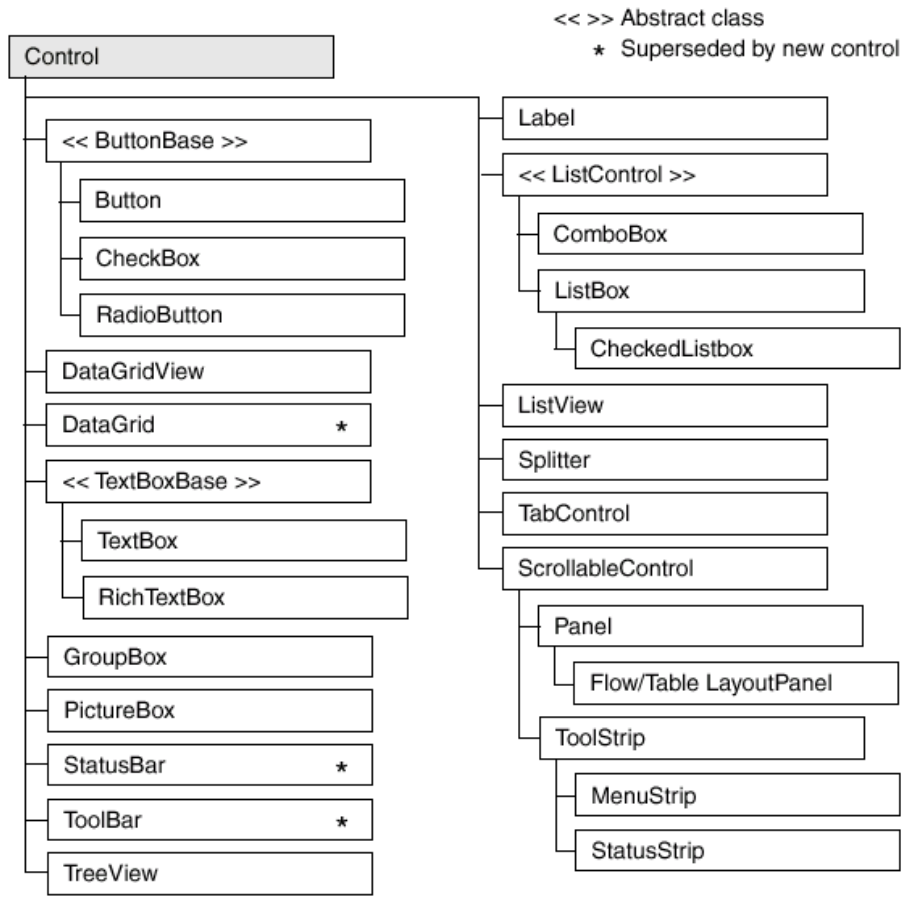
Listing 25 Wykorzystanie refleksji do tworzenia obiektów.

Poza powyżej przedstawionymi możliwościami refleksji w .Net używa się do:

- pobierania informacji o konstruktorach obiektu, ich parametrach oraz widoczności (public, private, protected) używając ConstructorInfo,
- pobierania informacji o polach przy pomocy FiledInfo,
- pobierania nazwy i typu oraz pobierania, i ustawiania wartości propert przy pomocy PropertyInfo. Często używana jest informacja o atrybutach, którymi udekorowana jest properta. Jest to używane m.in. we wspomnianym wcześniej Asp.Net MVC oraz Dynamic Data,
- tworzenia nowych typów w trakcie wykonania programu przy pomocy Reflection.Emit.

4.8. Windows Forms

Windows Forms to API, które umożliwia zarządzanie elementami interfejsu graficznego Windows. Powstał jako uproszczony następca opartego o C++ Microsoft Foundation Classes. W przeciwieństwie do MFC oraz Java SWING w Windows Forms nie zaimplementowano wzorca MVC, tylko oparto się na programowaniu sterowanym zdarzeniami (ang. Event-driven programming). Z jednej strony takie podejście, połączone z tworzeniem GUI przy pomocy Visual Studio Forms Designera, ułatwia pracę programiście, z drugiej strony, nie sprzyja to separacji kodu odpowiedzialnego z GUI i logikę biznesową.



Rysunek 9 Hierarchia klas Windows Forms [9]

Na Rysunek 9 widoczna jest hierarchia klas w Windows Forms. Klasą bazową jest klasa Control. Klasa ta zapewnia podstawową funkcjonalność do wyświetlania użytkownikowi informacji oraz pozwala obsłużyć zdarzenia w odpowiedzi na akcję

klawiatury oraz myszki, wykonaną przez użytkownika. Każda kontrolka posiada swoje granice na ekranie – pozycję oraz wymiary. Kontrolka stara się dopasować do swojego otoczenia, w taki sposób, że dziedziczy podstawowe właściwości, takie jak styl kursora, rodzinę, kolor czcionki i kolor tła, po swoim rodzicu.

Podstawowe kontrolki to:

- Button, odpowiedzialny za wyzwalanie zdarzeń w dopowiedzi na zdarzenia myszki i klawiatury,
- CheckBox, pozwala na zaznaczenie jednej lub kilku opcji,
- Textbox, pozwala na przekazanie wartości tekstowych przez użytkownika,
- Label, służy do wyświetlania informacji tekstowych,
- Listbox, wyświetla listę elementów, z których jeden lub więcej może być zaznaczony,
- RadioButton pozwala na zaznaczenie jednej z wielu opcji,
- ListView pozwala na wyświetlanie listy dowolnych kontrolek,
- MenuStrip pozwala na dodanie menu do aplikacji,
- ProgressBar wyświetla informację zwrotną dotyczącą postępu określonej operacji,
- TreeView wyświetla dane w formie drzewa,
- DataGridView służy do wyświetlania tabelarycznych danych, wspiera bindowanie do bazy danych,
- GroupBox służy do grupowania kontrolek,
- Panel, oraz FlowPanelLayout i TablePanelLayout pełni rolę kontenera na inne kontrolki.

4.9. JetBrains ReSharper

ReSharper to dodatek do Visual Studio służący do refaktoringu i zwiększania produktywności programisty. Jego podstawową funkcją jest statyczna analiza kodu – wykrywanie błędów bez potrzeby kompilacji kodu. Resharper oferuje także rozbudowaną

funkcję uzupełniania kodu, wiele opcji refaktoringu kodu oraz możliwość uruchamiania testów jednostkowych NUnit.

5. Opis implementacji

W tym rozdziale zostanie zaprezentowana implementacja prototypu rozwiązania. Najpierw zostanie opisana struktura projektu, następnie jego poszczególne komponenty, takie jak budowniczy kontrolek, budowniczy formularza, walidatory, klonowanie obiektów itd.

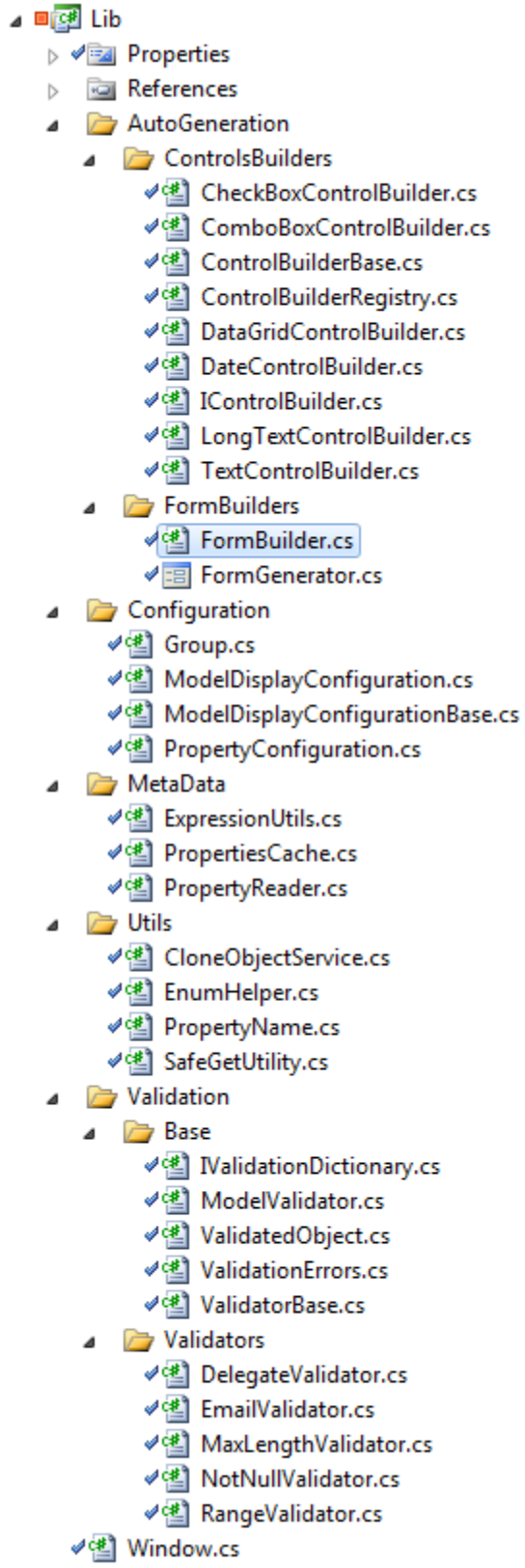
5.1.Struktura projektu

Implementacja została podzielona na komponenty, które jak widziać na Rysunek 10, mają odwzorowanie w strukturze projektu i namespaceach.

Cały projekt został podzielony na następujące namespcacy:

- ControlBuilders – zawiera klasy odpowiedzialne za generowanie kontrolek do wyświetlania i edycji danych,
- FormBuilders – zawiera klasy odpowiedzialne za budowanie formularza zgodnie z konfiguracją dostarczoną przez programistę,
- Configuration – zawiera klasy służące do konfiguracji UI, poszczególnych atrybutów modelu, jaki i całego modelu,
- MetaData – zawiera klasy pomocnicze do odczytywania oraz cachowania publicznych atrybutów modeli,
- Utils – Zawiera różne klasy pomocnicze m.in. do klonowania obiektów,
- Validation/Base zawiera klasy bazowe dla mechanizmu walidacji,
- Validation/Validators – zawiera implementacje poszczególnych walidatorów m.in. email walidator, walidator zakresu, walidator maksymalnej długości pola,
- Klasy statyczne Windows oraz Panel, które są słowami kluczowymi, od których rozpoczyna się definiowanie GUI.

Poza projektem biblioteki do deklaratywnego tworzenia GUI rozwiązanie zawiera, dwa dodatkowe projekty. W pierwszym umieszczono kilka testów jednostkowych. Drugi projekt to aplikacja Windows Forms, w której stworzono kilka przykładowych formularzy opisanych w rozdziale 3.6.



Rysunek 10 Struktura projektu.

5.2.Konfiguracja atrybutu modelu

Za konfigurację atrybutów w modelu odpowiedzialna jest klasa PropertyConfiguration. Na Listing 26 widoczny jest przykład tworzenia GUI dla jednego atrybutu modelu. W przykładzie tym ustawiana jest etykieta, wymiary kontrolki, zmieniany jest domyślny budowniczy na MaskedTextControl oraz dodawane są walidatory – pola wymaganego, makasymalnej długości oraz walidator z delegatem.

```
     e => e.Attribute(a => a.FirstName).Label("Imię")
.Width(30).Height(30).Use<MaskedTextControl>()
.MaxLength(10).Required()
.Must<Person>(p => !validNames.Contains(p.FirstName) ? "Niepoprawne imię" :
null),
```

Listing 26 Konfiguracja atrybutu modelu.

W Tabela 1 zostały przedstawione podstawowe funkcjonalności klasy PropertyConfiguration, czyli ustawianie etykiety oraz długości i szerokości kontrolki, a także dodawanie walidatorów do kontrolki.

Metoda	Opis
Label	Konfiguracja etykiety dla pola.
Width	Konfiguracja szerokości kontrolki.
Height	Konfiguracja wysokości kontrolki.
ReadOnly	Ustawienie kontrolki w trybie tylko do odczytu.
Use<T>	Konfiguracja budowniczego, który zostanie użyty do stworzenia kontrolki dla pola. Metoda jest używana, gdy programista chce użyć innej kontrolki niż domyślna np. gdy powinno być użyte pole tekstowe z maską zamiast zwykłego pola tekstowego.
Required	Dodaje walidator pola wymaganego do kontrolki.
Must<T>	Dodaje walidator, który używa delegata do weryfikacji poprawności

	pola.
Range	Dodaje walidator zakresu do pola.
MaxLength	Dodaje walidator na maksymalną długość pola.
Email	Dodaje walidator sprawdzający czy pole jest poprawnym adresem email.

Tabela 1 Najważniejsze metody klasy PropertyConfiguration.

W klasie PropertyConfiguration został zaimplementowany mechanizm do internacjonalizacji etykiet przy pomocy konwencji. Programista nie podaje klucza zasobu a jedynie typ. ResourceManager szuka zasobów używając nazwy atrybutu jako klucza. Jeśli zasób zostanie odnaleziony, użyty będzie jako etykieta. Jeśli nie zostanie znaleziony jako etykieta zostanie użyta wartość domyślna, czyli nazwa atrybutu.

```
            public string DisplayName
        {
            get
            {
                if (_displayName != null)
                  return _displayName;
                if (ResourceManager != null)
                {
                    var                      resource                      =
ResourceManager.GetString(PropertyInfo.Name);
                    if (resource != null)
                    {
                        _displayName = resource;
                        return resource;
                    }
                }
                _displayName = PropertyInfo.Name;
                return _displayName;
            }
            set
```

```
            {
                _displayName = value;
            }
        }
```

Listing 27 Mechanizm do internacjonalizacji etykiet.

5.3.Konfiguracja modelu

Za konfigurację modelu odpowiedzialne są dwie klasy – generyczna ModelDisplayConfiguration <T> oraz abstrakcyjna ModelDisplayConfiguration. Na Listing 28 widoczne jest użycie tej klasy do wygenerowania formatki dla klasy person. W poniższym przykładzie określona jest lista atrybutów, które mają być wyświetlone na formatce, ustawiony jest tytuł oraz klasa zasobów, oraz dodany walidator dla całego modelu.

```
Window.Using(person)
.Containing(
e => e.Attribute(a => a.Age).Range(0, 120).Required(),
                    ...
e => e.Attribute(a => a.Photo)
.Must(model => model.IsMarried && model.Age < 18 ? "Komunikat" : null)
.Title("Edycja danych osoby")
.UseResources<Resource>()
.Show();
```

Listing 28 Konfiguracja modelu.

W Tabela 1 prezentowane są najważniejsze metody służące do konfiguracji formatki, czyli ustawianie tytułu, pliku zasobów, grup, które mają zostać wyodrębnione na formatce oraz wyszczególnienie, atrybutów które mają się pojawić na formatce, a także, polecenie Show. Metoda ta pojawia się jako ostatnia w całym wywołaniu i służy do wyświetlania formatki.

Metoda	Opis
Title	Ustawienia tytułu formatki.
Attribute	Wskazania, który atrybut jest aktualnie definiowany.

Show	Wywołuje operację budowania formatki.
UseResources	Wskazuje, która zasoby mają zostać użyte.
Group	Służy do definiowania grup na formatce.
Containing	Metoda w argumencie dostaje konfigurację atrybutów modelu.

Tabela 2 Najważniejsze metody służące do konfiguracji formatki

Podczas inicjalizacji obiektu typu ModelDisplayConfiguration <T> tworzone są domyślne ustawienia dla modelu. Polega to na pobraniu wszystkich publicznych atrybutów dla typu T i stworzeniu dla każdego atrybutu konfiguracji z domyślnymi wartościami. Na Listing 29 widoczna jest metoda, która służy do nadpisywania domyślnych ustawień. Metoda ta najpierw czyści domyślną konfigurację a następnie tworzy nową, wykorzystując listę funkcji przekazanych w parametrze.

```
public ModelDisplayConfiguration<T> Containing(params
Func<ModelDisplayConfiguration<T>, PropertyConfiguration>[] args)
        {
            ClearDefaultSettings();
            foreach (var createPropertyConfigurationFunction in args)
            {
                var propertyConfiguration =
createPropertyConfigurationFunction(this);
                ModelConfiguration.Add(propertyConfiguration.PropertyInfo,
propertyConfiguration);
            }
            return this;
          }
```

Listing 29 Metoda nadpisująca konfigurację atrybutów.

5.4.Walidacja

Walidacja we frameworku umożliwia dodawanie zarówno reguł dla poszczególnych atrybutów jak i całego obiektu. Zostało to zaimplementowane w taki sposób, że konfiguracja

przetrzymuję listę walidatorów dla całego modelu – ClassValidators, a dodatkowo każdy atrybut posiada własną listę walidatorów – PropertyValidators.

Błędy walidacji wyświetlane są w dwóch miejscach. Obok pola edycyjnego wyświetlana jest informacja, czy wyświetlona wartość jest poprawa. Dodatkowo na dole formatki znajduje się podsumowanie, w którym wyróżnione są zarówno błędy atrybutów, jak i błędy modelu.

Informacja o błędach walidacji znajduje się w klasie implementującej interfejs IValidationDictionary, który jest widoczny na Listing 30. Metoda IsPropertValid zwraca informację czy atrybut modelu posiada poprawną wartość. Metoda GetAllErrors zwraca listę wszystkich komunikatów błedów, a właściwość IsValid wskazuje czy wszystkie atrybuty modelu posiadją prawidłowe wartości.

```
public interface IValidationDictionary
        {
            void AddError(string key, string errorMessage);
            bool IsValid { get; }
            void AddClassError(string errorMessage);
            bool IsPropertValid(string property);
            List<string> GetAllErrors();
          }
```

Listing 30 Interfejs do obsługi słownika błędów.

Klasa ModelValidator, jest odpowiedzialna za walidację modelu. Jak widać na Listing 31 w metodzie Validate, zaimplementowana jest logika walidacyjna. Najpierw następuje zagnieżdżona iteracja po wszystkich atrybutach i ich kolekcjach walidatorów PropertyValidators. Dla każdego walidatora uruchamiana jest walidacja. W przypadku braku sukcesu walidacji, dodawany jest komunikat błędu do słownika błędów atrybutów. Następnie uruchamiane są walidatory klasowe, a błędy walidacji dodawane są do listy błędów całego modelu.

```
public class ModelValidator {
      public ModelDisplayConfigurationBase ModelConfiguration { get; set; }
```

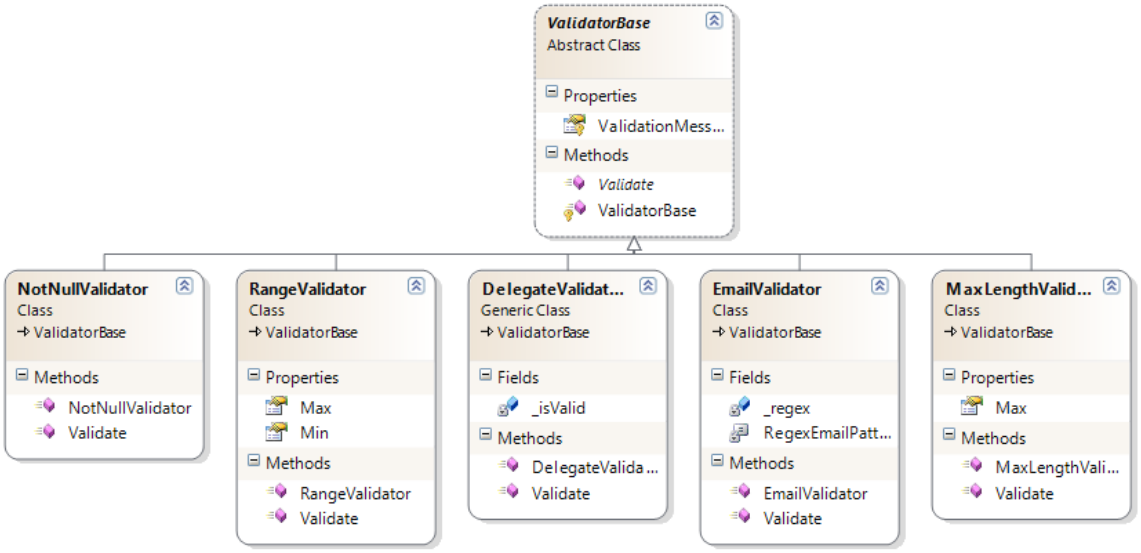
```
        public ModelValidator(ModelDisplayConfigurationBase
modelDisplayConfiguration) {
            ModelConfiguration = modelDisplayConfiguration;
        }

        public bool Validate(object model) {
            var validationDictionary = ModelConfiguration.ValidationErrors;
            foreach (PropertyConfiguration propertyConfiguration in
ModelConfiguration.ModelConfiguration.Values) {
                foreach (var validator in propertyConfiguration.PropertyValidators)
{
                    var validationResult = validator.Validate(new ValidatedObject()
{
                        TargetName = propertyConfiguration.DisplayName,
                        TargetValue =
propertyConfiguration.PropertyInfo.GetValue(model, null),
                        Model = model
                    });
                    if (validationResult != null) {

validationDictionary.AddError(propertyConfiguration.PropertyInfo.Name,
validationResult);
                    }
                }
            }

            foreach (var classValidator in ModelConfiguration.Classalidators) {
                var validationResult = classValidator.Validate(new
ValidatedObject() {
                    TargetName = null,
                    TargetValue = null,
                    Model = model
                });
                if (validationResult != null) {
                    validationDictionary.AddClassError(validationResult);
                }
            }

            return validationDictionary.IsValid;
        }
    }
```

Listing 31 ModelValidator – klasa odpowiedzialna za walidację modelu.

5.5. Walidatory

We frameworku do deklaratywnego tworzenia GUI stworzono podstawowe walidatory, tak aby programista nie był zmuszony do ciągłego pisania reguł walidacyjnych. Na Rysunek 11 widać hierarchię klas walidatorów. Podstawowe walidatory, które zostały zaimplementowane to:

- NotNullValidator – walidator pola wymaganego,
- RangeValidator – walidator zakresu,
- DelegateValidator – uniwersalny walidator, przyjmujący delegat w argumencie, który rozstrzyga czy obiekt jest poprawny,
- EmailValidator – walidator wiadomości email,
- MaxLengthValidator – walidator długości pola tekstowego.



Rysunek 11 Hierarchia klas walidatorów.

Na Listing 32 widać klasę abstrakcyjną ValidatorBase, która jest klasą bazową dla wszystkich walidatorów. Zawiera ona metodę Validate, która musi zostać zaimplementowana przez konkretne implementacje walidatorów. Metoda ta weryfikuje czy obiekt ma poprawną wartość. Jeśli wśród walidatorów zaimplementowanych w bibliotece nie ma odpowiedniego,

programista może zaimplementować swój walidator. Utworzona klasa musi dziedziczyć po ValidatorBase. Następnie należy dodać ją do kolekcji PropertyValidators lub ClassValidators.

```
public abstract class ValidatorBase
        {
            protected ValidatorBase(string message)
            {
                ValidationMessage = message;
            }
            protected string ValidationMessage { get; set; }

            public abstract string Validate(ValidatedObject
validatedObject);
        }
```

Listing 32 Bazowy walidator.

Na Listing 33 widać jedną z implementacji walidatora – RangeValidator. Walidator ten parsuje obiekt walidowany do liczby i weryfikuje czy znajduje się on w podanym wcześniej zakresie. Jeśli walidacja nie kończy się sukcesem walidator zwraca komunikat błędu, który następnie jest wyświetlony na formatce.

```
      public class RangeValidator : ValidatorBase
    {
        public int Min { get; private set; }

        public int Max { get; private set; }

        public RangeValidator(int min, int max, string message = "Pole {0}
musi mieć wartość pomiędzy {1} i {2}")
            : base(message)
        {
            Min = min;
            Max = max;
        }

        public override string Validate(ValidatedObject validatedObject)
        {
            int value;
```

```
            if (validatedObject.TargetValue == null) return null;

            if (!int.TryParse(validatedObject.TargetValue.ToString(), out
value))
                return null;

            if (value < Min || value > Max)
                return string.Format(ValidationMessage,
validatedObject.TargetName, Min, Max);

            return null;
        }
    }
```

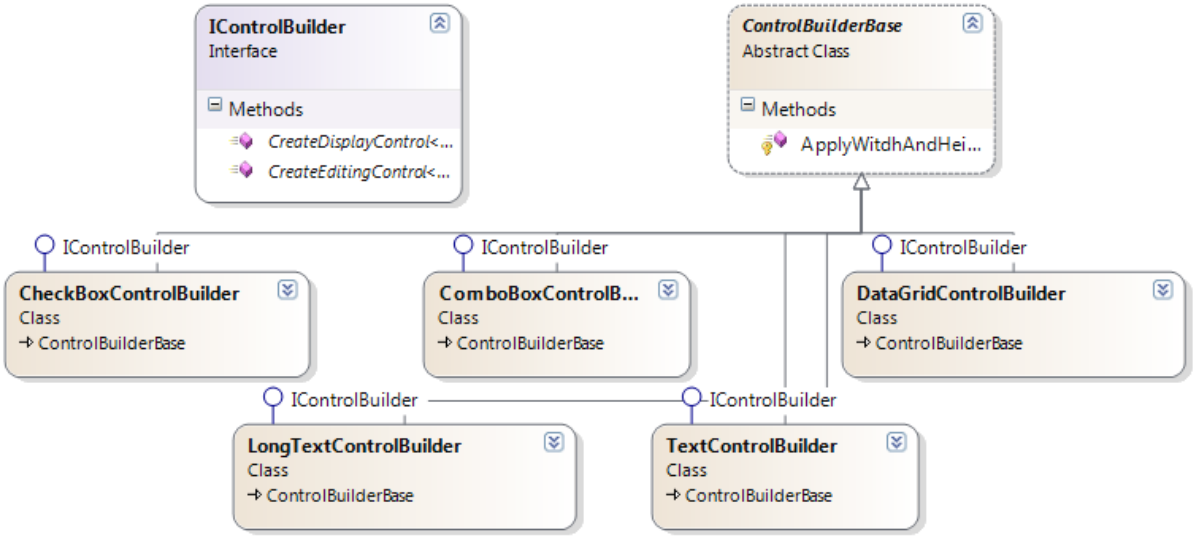
Listing 33 Walidator zakresu.

5.6.Tworzenie kontrolek

Za tworzenie kontrolek odpowiedzialne są klasy implementujące interfejs IControlBuilder- widoczny na Listing 34. Klasy te muszą zaimplementować dwie metody. Pierwsza, CreateDisplayControl, tworzy widet w trybie tylko do odczytu, druga, CreateEditingControl, tworzy kontrolkę do edycji danych. Obie metody w argumencie dostają model oraz konfigurację wyświetlanego pola, która zawiera takie informacje, jak wymiary oraz nazwę pola, które będzie zbindowane do kontrolki.

```
  public interface IControlBuilder
{
            Control CreateDisplayControl<TModel>(TModel model,
PropertyConfiguration propertyName);

            Control CreateEditingControl<TModel>(TModel model,
PropertyConfiguration propertyName);
}
```

Listing 34 Interfejs IControlBuilder.



Listing 35 Hierarchia klas budowniczych kontrolek.

Na Listing 35 widać wszystkie klasy odpowiedzialne za tworzenie kontrolek:

- CheckBoxControlBuilder, tworzący kontrolkę przycisk wyboru,
- ComboBoxControlBuilder, tworzący kontrolkę pole wyboru,
- DataGridControlBuilder, tworzący kontrolkę do wyświetlania danych tabelarycznych,
- TextControlBuilder, tworzący kontrolkę pole tekstowe,
- Long TextControlBuilder, tworzący kontrolkę do edycji i wyświetlania sformowanego tekstu.

Jednym z wymagań biblioteki do deklaratywnego tworzenia GUI była możliwość dodawania widgetów. W celu dodania nowej kontrolki należy zaimplementować interfejs IControlBuilder i zarejestrować taką klasę jako domyślną do wyświetlania i edycji typu, co zostanie opisane w rozdziale 5.7. Na Listing 36 widać implementację klasy ImageControlBuilder, która będzie odpowiedzialna za wyświetlanie obrazów.

```
public class ImageControlBuilder : ControlBuilderBase, IControlBuilder
        {
            public Control CreateDisplayControl<TModel>(TModel model,
PropertyConfiguration propertyName)
            {
                var pictureBox = new PictureBox();
                pictureBox.SizeMode = PictureBoxSizeMode.StretchImage;
```

```
                var filename = (Image)propertyName.PropertyInfo.GetValue(model,
null);
                var path = filename.Path;
                if (path != null) {
                    var image = new Bitmap(path);
                    pictureBox.ClientSize = new Size(propertyName.WidthValue ??
100, propertyName.HeightValue ?? 100);
                    pictureBox.Image = (System.Drawing.Image)image;
                }
                var panel = new Panel();
                panel.Controls.Add(pictureBox);
                return panel;
            }
            public Control CreateEditingControl<TModel>(TModel model,
PropertyConfiguration configuration)
            {
                return CreateDisplayControl(model, configuration);
            }
        }
```

Listing 36 Tworzenie kontrolki do wyświetlania obrazów.

5.7.ControlBuildersRegistry

Framework do deklaratywnego tworzenia GUI miał na celu skrócenie czasu potrzebnego do programowania GUI poprzez użycie konwencji do tworzenia widgetów. Zostało to osiągnięte poprzez użycie rejestru budowniczych kontrolek ControlBuildersRegistry, który jest singletonem i służy do wiązania typu z konkretnym budowniczym kontrolki. W trakcie budowania formularza ControlBuildersRegistry jest używany do pobierania budowniczego dla wszystkich atrybutów modelu. Na Listing 37 widać rejestrowanie domyślnych budowniczych dla typów.

```
  protected  void RegisterDefaultBuilders()
        {
            Register(typeof(string), new TextControlBuilder());
            Register(typeof(bool), new CheckBoxControlBuilder());
            Register(typeof(bool?), new CheckBoxControlBuilder());
            Register(forType => typeof(Enum).IsAssignableFrom(forType), new
ComboBoxControlBuilder());
            Register(typeof(DateTime), new DateControlBuilder());
```

```
            Register(typeof(DateTime?), new DateControlBuilder());
}
```

Listing 37 Rejestrowanie domyślnych budowniczych.

Framework został przystosowany do łatwego rozszerzania jego możliwości. W poprzednim rozdziale zostało opisane w jaki sposób można utworzyć nowego budowniczego. Na Listing 38 widoczny jest proces rejestrowania nowych budowniczych. Pierwsze polecenie rejestruje ImageControlBuilder dla wszystkich pól o typie Image. Drugie polecanie przypisuje DataGridControlBuilder wszystkim typom, które można przypsać do typu IEnumerable.

```
ControlBuildersRegistry.Instance.Register(
typeof(Image), new ImageControlBuilder());

ControlBuildersRegistry.Instance.Register(
type => typeof(IEnumerable).IsAssignableFrom(type),
 new DataGridControlBuilder());
```

Listing 38 Rejestrowanie budowniczych.

Wiązanie typu z budowniczym można wykonać na dwa sposoby. Pierwszy to proste dodanie wpisu do słownika z kluczem typ i wartością budowniczy. Drugi to dodanie funkcji, która rozstrzygnie czy budowniczy powinien zostać użyty dla typu. Na Listing 39 widać jak zaimplementowane jest rozstrzyganie, który budowniczy ma zostać użyty dla typu. Najpierw szukany jest wpis w słowniku typ – budowniczy, następnie ewaluowane są wszystkie funkcje weryfikujące czy typ pasuje do budowniczego. Jeśli dwa pierwsze warunki nie są spełnione zwracany jest domyślny budowniczy.

```
public IControlBuilder GetBuilder(Type forType)
        {
            if (Map.ContainsKey(forType))
            {
                return Map[forType];
            }

            foreach (var constraint in MapConstraints)
            {
                Func<Type, bool> constraintFunction = constraint.Key;
```

```
                var match = constraintFunction(forType);
                if (match) return constraint.Value;
            }

            return DefaultBuilder;
        }
```

Listing 39 Pobieranie budowniczego dla typu.

5.8.Klonowanie wartości

Typowe zastosowanie biblioteki wygląda tak, że do formularza przekazywany jest model, który następnie będzie edytowany przez użytkownika. Użytkownik po skończonej edycji może wykonać dwie czynności: wcisnąć przycisk „Ok” i tym samym zaakceptować zmiany, lub wcisnąć przycisk „Cancel”, aby wycofać zmiany. Przy wycofaniu zmian pojawia się problem, polegający na tym, że nie jesteśmy w stanie wycofać zmian zrobionych bezpośrednio na modelu.

W bibliotece zostało przyjęte rozwiązanie polegające, na przekazywaniu do formatki klonu modelu zamiast oryginalnego modelu. Jeśli użytkownik akceptuje zmiany, zostają one naniesione z klonu obiektu na oryginalny obiekt. Gdy użytkownik postanowi wycofać zmiany, wtedy są one ignorowane.

Metoda Clone<T> widoczna na Listing 40 jest odpowiedzialna za klonowanie modelu. Proces klonowania zaczyna się od stworzenia nowego obiektu przy pomocy metody CreateInstance obiektu Assembly. Następnie przy pomocy refleksji kopiowane są wartości wszyskich pól, które będą wyświetlane i edytowane na formatce.

```
public static T Clone<T>(T toBeCloned, Dictionary<PropertyInfo,
PropertyConfiguration> modelConfiguration) where T : class
        {
            var type = toBeCloned.GetType();
            T clonedObject = type.Assembly.CreateInstance(type.ToString())
as T;
            foreach (var propertyInfo in modelConfiguration.Keys)
            {
                if (propertyInfo.CanWrite)
```

```
                {
                    var newValue = propertyInfo.GetValue(toBeCloned, null);
                    propertyInfo.SetValue(clonedObject, newValue, null);
                }
            }

            return clonedObject;
        }
    }
```

Listing 40 Metoda klonująca obiekt.

Proces nanoszenia zmian z obiektu sklonowanego na oryginalny jest widoczny na Listing 41. Jest to proces odwrotny do klonowania. Dla wszystkich pól wyświetlonych na formularzu wartości są kopiowane, pod warunkiem, że pole nie jest tylko do odczytu.

```
public static void UpdateFrom<T>(T from, T to, Dictionary<PropertyInfo,
PropertyConfiguration> modelConfiguration)
        {

            foreach (var propertyInfo in modelConfiguration.Keys)
            {
                var configuration = modelConfiguration[propertyInfo];
                if (propertyInfo.CanWrite && !configuration.IsReadOnly)
                {
                    var newValue = propertyInfo.GetValue(from, null);
                    propertyInfo.SetValue(to, newValue, null);
                }
            }
        }
```

Listing 41 Metoda UpdateFrom.

5.9.Silna kontrola typów

Jednym z założeń projektu była silna kontrola typów oraz uniknięcie przekazywania nazw atrybutów w stringach. Zostało to osiągnięte poprzez użycie drzewa wyrażeń oraz wyrażeń lambda widocznych na Listing 42.

```
e => e.Attribute(a => a.Age)
```

Listing 42 Konfiguracja pola przy pomocy lambda.

Na Listing 43 zaprezentowany jest sposób w jaki wyłuskiwana jest informacja o atrybucie z wyrażenia lambda. Do metody GetMember jest przekazywane ciało wyrażenia lambda np. a=>a.Age. Ciałem tego wyrażenia jest MemberExpresion {a.Age}, z którego można pobrać pełne informację o atrybucie, takie jak typ, nazwa, flaga czy pole jest tylko do odczytu itd.

```
public static MemberInfo GetMember(Expression expression)
        {
            if (expression is MemberExpression)
            {
                var memberExpression = (MemberExpression)expression;
                if (memberExpression.Expression.NodeType ==
ExpressionType.MemberAccess)
                {
                    return GetMember(memberExpression.Expression);
                }
                return memberExpression.Member;
            }

            if (expression is UnaryExpression)
            {
                var unaryExpression = (UnaryExpression)expression;
                if (unaryExpression.NodeType != ExpressionType.Convert)
                {
                    throw new ArgumentException(expression.ToString());
                }
                return GetMember(unaryExpression.Operand);
            }

            throw new ArgumentException(expression.ToString());
        }
```

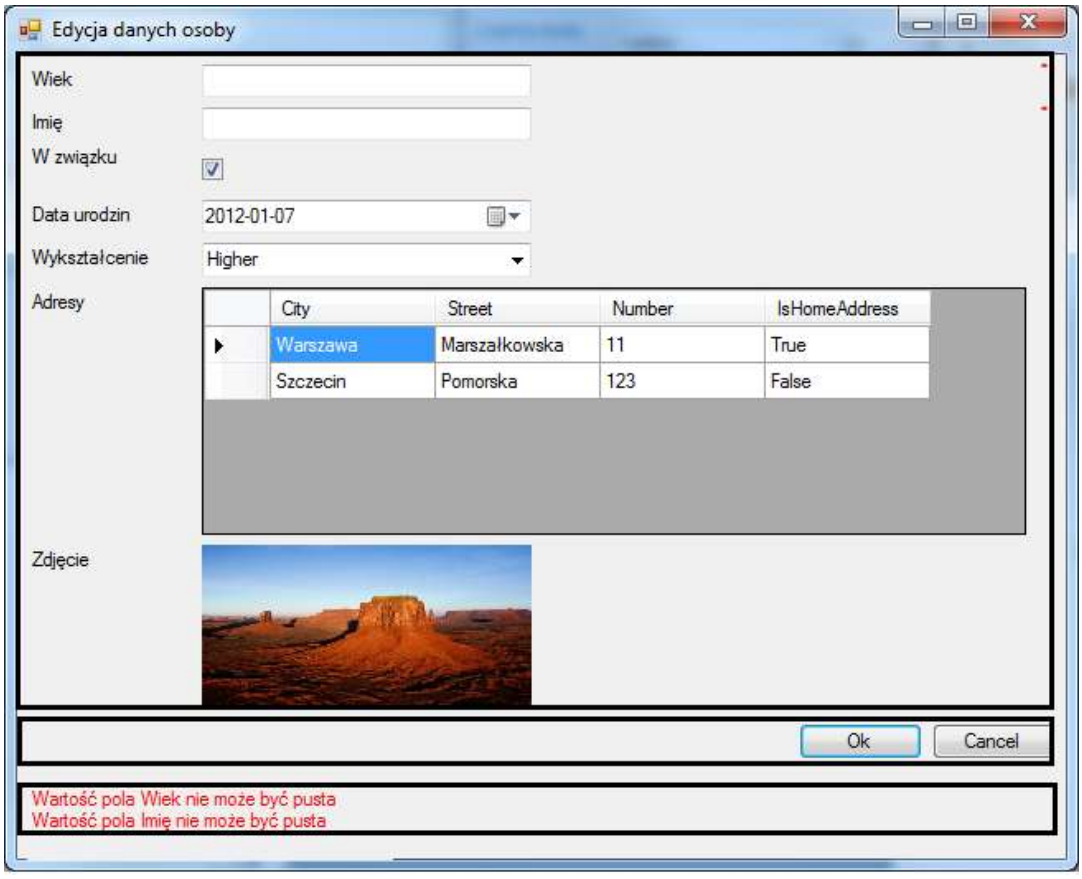
Listing 43 Wyłuskiwanie informacji o atrybucie z wyrażenia lambda.

5.10. Budowniczy formularza

W poprzednich rozdziałach wytłumaczono jak programista może konfigurować GUI oraz jak tworzona jest kontrolka na podstawie konfiguracji. W tym rozdziale zostanie opisane jak tworzone jest całe okno wraz z wszystkim polami formularza i przyciskami oraz kontrolkami wyświetlającymi błędy walidacji.

Za generowanie formularza jest odpowiedzialna klasa FormBuilder. Najpierw tworzony jest formularz i dodawane są do niego trzy panele, które będą przechowywały elementy znajdujące się na każdym formularzu. Jak widać na Rysunek 12 są to:

- etykiety i pola do wprowadzania danych oraz czerwone gwiazdki, wskazujące, że w pole została wprowadzona nieprawidłowa wartość,
- przyciski „Ok” i „Cancel”,
- pole z pełnymi opisami błędów walidacyjnych.



Rysunek 12 Przykładowy formularz.

Następnie tworzone są GroupBoxy dla każdej grupy kontrolek. Ostatnim etapem tworzenia formularza jest generowanie kontrolek dla każdego atrybutu modelu, co zaprezentowane jest na Listing 44. Dla każdego pola obiektu tworzona jest etykieta. Później z rejestru ControlBuildersRegistry pobierany jest budowniczy, odpowiednio dobrany do typu atrybutu. W zależności od tego czy cała formatka lub samo pole zostało ustawione tylko do odczytu, generowana jest kontrolka do wyświetlania lub edycji pola. Na samym końcu dodane są gwiazdki, dla atrybutów, które nie spełniają wymagań walidatorów przypisanych do nich.

```
foreach (var property in keyCollection)
      {
       string propertyName = property.Name;
       var configuration =
ModelDisplayConfiguraion.ModelConfiguration[property];
         CreateLabelAndAddToForm(configuration.DisplayName,
controlsContainer, rowIndex, 0);
                Control control = null;
            var builder = configuration.CurrentControlBuilder
                ??
ControlBuildersRegistry.Instance.GetBulder(property.PropertyType);

            if (ModelDisplayConfiguraion.IsReadOnly ||
configuration.IsReadOnly)
            {
     control =
builder.CreateDisplayControl(ModelDisplayConfiguraion.WorkingCopy,
configuration);
            }
            else
            {
                control =
builder.CreateEditingControl(ModelDisplayConfiguraion.WorkingCopy,
configuration);
            }

                if
(!ModelDisplayConfiguraion.ValidationErrors.IsPropertValid(propertyName))
                {
                    AddValidationError(controlsContainer, rowIndex);
                }
                controlsContainer.Controls.Add(control, 1, rowIndex);
                rowIndex++;
                propertyPanelMap.Add(property, controlsContainer);
              }
```

Listing 44 Tworzenie kontrolek dla wszystkich atrybutów modelu.

6. Podsumowanie

W tym rozdziale autor dokonał podsumowania koncepcji i implementacji deklaratywnego języka do tworzenia GUI.

6.1. Wady i zalety rozwiązania

Podstawową wadą rozwiązania jest brak elastyczności. Biblioteka jest przeznaczone do tworzenia formularzy o sztywno określonej strukturze, żadne odstępstwa nie są możliwe. Jeśli chcielibyśmy stworzyć formularz, na którym kontrolki byłyby wyświetlane w dwóch wierszach, musielibyśmy skorzystać z API Windows Forms i stworzyć ekran od podstaw.

Niewątpliwą zaletą języka jest użycie wyrażeń lambda i pozbycie się konieczności używania identyfikatorów przekazywanych w obiektach String. Takie rozwiązanie pozwala zachować mocną kontrolę typów i w związków z tym pozwala wykrywać część błędów w trakcie kompilacji. Dodatkowo użycie wyrażeń lambda pozwala na wykorzystanie narzędzi do refaktoringu, które są bezużyteczne w przypadku przechowywania nazw atrybutów w Stringu.

Kolejną zaletą jest zminimalizowanie konieczności konfigurowania, dzięki użyciu podejścia „konwencja ponad konfigurację” oraz automatyzacja generowania kontrolek. Dzięki temu można stworzyć cały formularz za pomocą jedynie kilku linijek kodu.

Rozwiązanie jest przystosowane do obsługi kilku typów danych oraz dodawania zaledwie kilku rodzajów walidatorów. Jeśli programista stwierdzi, że potrzebuje dodatkowych funkcjonalności, biblioteka jest przystosowana do rozszerzania jej możliwości.

6.2. Proponowany plan rozwoju

Elementy, które można poprawić:

- Rozszerzenie języka o obsługę nowych typów danych oraz generowanie nowych widgetów np. selektor godziny.
- Poszerzenie możliwości języka przez dostarczenie nowych możliwości konfiguracji elementów GUI, np. marginesy, wyrównanie tekstu, możliwość wyświetlania kontrolek w kilku kolumnach.

- Stworzenie rozwiązania webowego na platformę Asp.Net. Rozwiązanie takie wymagałoby jedynie napisania nowych funkcjonalności do tworzenia kontrolek i formularzy. Reszta biblioteki jest napisana w sposób niezależna od konkretnej platformy.
- Stworzenie rozwiązania, które umożliwiałoby za pomocą kilku linijek generowanie aplikacji do przeglądania, edycji i dodawania danych. Taka biblioteka bazowałoby na relacjach pomiędzy obiektami biznesowymi. Dodatkowo konfiguracja UI oraz walidacji byłaby robiona dla każdego obiektu biznesowego, a nie dla formatki jak ma to miejsce teraz.

6.3. Wnioski końcowe

Programowanie GUI to część pracy programisty, która bywa pracochłonna i monotonna. Jest to także miejsce, gdzie powstaje duża cześć błędów. Celem pracy była analiza istniejących rozwiązań, które ułatwiają i automatyzują proces tworzenia graficznych interfejsów użytkownika.

Wynikiem analizy jest koncept deklaratywnego języka, charakteryzującego się prostą, zwięzłą i samo tłumaczącą się składnią. Rozwiązanie to nie tylko przyśpiesza pracę programisty, ale także zmniejsza ilość potencjalnych błędów, dzięki temu, że pisząc w deklaratywnym języku programista skupia się na tym co ma osiągnąć, a nie na korzystaniu z niskopoziomowego API do tworzenia GUI.

Zaprezentowany prototyp języka zrealizował stawiane przed nim założenie automatyzacji procesu, dzięki zastosowaniu rozwiązań bazujących na użyciu konwencji. Najprostszy formularz można napisać za pomocą jednej linii kodu.

Język został stworzony w taki sposób, aby programista mógł go w łatwy sposób dostosowywać do własnych potrzeb oraz dopisywać do niego nowe funkcjonalności. Proponowany kierunek rozwoju rozwiązania to implementacja wersji webowej przeznaczona dla technologii Asp.Net.

7. Bibliografia

1. **Trzaska, Mariusz.** *senseGUI – A DECLARATIVEWAY OF GENERATING.* Porto, : Proceedings of the Third International Conference on Software and Data Technologies (ICSOFT 2008)., 2008. strony 71 - 76. 978-989-8111-51-7.

2. **Joseph Balderson, Peter Ent, Jun Heider, Todd Prekaski, Tom Sugden, Andrew Trice, David Hassoun, Joe Berkovitz.** *Professional Adobe Flex 3.* brak miejsca : John Wiley and Sons, 2011, 2011. 1118059484.

3. **Matthew MacDonald, Adam Freeman, Mario Szpuszta.** *Pro ASP.Net 4 in C# 2010.* Apress, 2010. 1430225297.

4. **MacDonald, Matthew.** *Pro WPF in C# 2010.* brak miejsca : Apress, 2010. 1430272058.

5. **Trzaska, Mariusz.** *GCL – An Easy Way for Creating Graphical User Interfaces.* Orlando, : Proceedings of the 14th World Multi-Conference on Systemics, Cybernetics and Informatics:, 2010. strony 403 - 410. 978-1-934272-98-5.

6. **Jeffrey Palermo, Ben Scheirman, Jimmy Bogard, Matthew Hinze, Eric Hexter.** *ASP.NET MVC 2 in Action.* Manning Publications, 2010. Manning Publications.

7. .NET Framework Developer Center. [Online] 11 12 2011. http://msdn.microsoft.com/pl-pl/netframework/cc511286.

8. Microsoft Developer Network. [Online] 12 12 2011. http://msdn.microsoft.com/en-us/library/bb397951(v=vs.90).aspx.

9. Windows Forms Controls. *Codeguru.* [Online] 9 1 2006. [Zacytowano: 18 12 2011.] http://www.codeguru.com/csharp/sample_chapter/article.php/c11213.