POLSKO-JAPOŃSKA WYŻSZA SZKOŁA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Marek Drob

4061

MDGL - Deklaratywny język do tworzenia graficznych interfejsów użytkownika

Praca magisterska

Napisana pod kierunkiem

dr inż. Mariusza Trzaski

miejce, miesiąc, rok obrony

Spis Treści

1 Wstęp

Podczas tworzenia oprogramowania zwraca się obecnie ogromną uwagę nie tylko na szybkość realizacji zadania, ale również na jakość stworzonego przez programistę kodu. Coraz częściej czytelność i intuicyjność w wykorzystaniu API danej technologii są ważnymi argumentami decydującymi o wyborze danego rozwiązania. Kod, który jest łatwo rozszerzalny i w prosty sposób interpretowany przez nowego członka zespołu, stanowi inwestycję, która zwraca się bardzo szybko. W związku z tym, powstają łatwiejsze w użyciu biblioteki i technologie. Służą one jednocześnie zwiększeniu produktywności, ale także temu, aby uczynić rozwiązanie mniej skomplikowanym.

Powstało wiele koncepcji, m.in. wspierających budowę usług sieciowych *(ang. Web services)*, dostarczających gotowe szkielety aplikacji(Spring), które ujednolicają dostęp do bazy danych (Hibernate, iBatis, LinqToSql). Jednak obszar definiowania graficznych interfejsów użytkownika w sposób bardziej przyjazny wciąż nie dostarcza wielu rozwiązań. Dopiero stosunkowo niedawno nastąpił przełom w tworzeniu nowych idei. Powstanie takich technologii, jak JavaFX, XAML, UiBinder, GCL ukazuje kierunek rozwoju tej dziedziny – podejście deklaratywne.

1.1 Cel pracy

Celem pracy było przedstawienie obecnie wykorzystywanych technologii do tworzenia graficznych interfejsów użytkownika. Poruszany jest głównie kontekst łatwości tworzenia takich interfejsów przez programistę. Dodatkowym celem pracy było opracowanie nowej koncepcji języka służącego do budowy interfejsów użytkownika oraz stworzenie działającego prototypu owego języka. W założeniu język ten ma być deklaratywny i ukrywać przed programistą niskopoziomowe API, służące do tworzenia graficznych interfejsów użytkownika.

1.2 Rezultaty pracy

Rezultatem pracy jest koncepcja nowego języka służącego do budowy graficznych interfejsów użytkownika. Główną cechą tego języka jest czytelność kodu i łatwość w

generowaniu nowych elementów danego interfejsu użytkownika. Jest on również niezależny od platformy i jego wyrażenia mogą być w prosty sposób przenoszone pomiędzy obsługiwanymi platformami.

W ramach pracy stworzono także działający prototyp języka. Jest to implementacja w języku Java. Pozwala on na tworzenie graficznych interfejsów użytkownika w aplikacjach wykorzystujących Swing lub GWT.

1.3 Organizacja pracy

Niniejsza praca zawiera sześć rozdziałów. Kolejne opisują następujące zagadnienia:

1. Cel pracy i oczekiwany rezultat pracy.
2. Historie graficznych interfejsów użytkownika, opisano sposoby tworzenia. Przedstawiono również stan sztuki – czyli przegląd dostępnych obecnie deklaratywnych technologii, pozwalających generować graficzne interfejsy użytkownika.
3. Koncepcje nowego języka. Przedstawiono jego semantykę i podano przykładowe wykorzystanie w pseudokodzie.
4. Narzędzia wykorzystane podczas tworzenia implementacji prototypu języka.
5. Dokładny opis implementacji prototypu. Przedstawiono w nim wykorzystane wzorce projektowe. Opisano także wady i zalety stworzonego prototypu oraz zaproponowano plan rozwoju.
6. Podsumowanie pracy.

2 Graficzne interfejsy użytkownika

Graficzny interfejs użytkownika (ang. Graphical User Interface), często nazywany też środowiskiem graficznym, to sposób, w jaki użytkownik wchodzi w interakcje z systemem. Środowisko graficzne używa kombinacji graficznych elementów(przyciski, okna, menu) i urządzeń (myszka, klawiatura, ekran dotykowy) do zaprezentowania informacji w jak najbardziej naturalnej dla użytkownika postaci. Dzięki zastosowaniu tego typu interfejsu otrzymano produkt dużo bardziej przyjazny niż, np., CLI *(ang. Command Line Interface)* - wpisywanie komend w linii poleceń.

Oprócz tego, graficzne interfejsy użytkownika są często najważniejszą kwestią, decydującą o sukcesie, bądź porażce danego oprogramowania. Jest to pierwsza rzecz, na którą użytkownik zwraca uwagę przy kontakcie z danym systemem. Oprogramowanie, które posiada interfejs estetyczny, intuicyjny i przede wszystkim wygodny w użytkowaniu, ma dużo większą szansę na sukces.

2.1 Historia

Pierwszy interfejs, który później okazał się prekursorem dzisiejszych graficznych interfejsów użytkownika, powstał w latach 60. XX wieku, w Stanford Research Institute. Pracom tym przewodził Douglas Engelbart. Opracowano rozwiązanie, które wykorzystywało tekstowe łącza między dokumentami (ang. Hyperlinks). Do interakcji użyto znanej doskonale myszki komputerowej (także stworzona przez Engelbart'a). Pomysł wykorzystania tekstowych łączy został rozwinięty i rozszerzony przez firmę Xerox w jej centrum badawczym PARC *(ang. Palo Alto Research Center Incorporated)*. System powstał dla komputerów Xerox Alto. Interfejs graficzny budowano przy użyciu takich elementów jak:

- Okna
- Ikony
- Przycisk opcji
  - Radio button – wykluczające się opcje
  - Checkbox – niewykluczające się opcje
- Pasek menu

Poza klawiaturą, użytkownik dostał możliwość korzystania z myszki komputerowej. Współczesne systemy komputerowe opierają się na tych samych ideach, co system wykreowany w latach 70. XX wieku. Świadczy to o istotnym znaczeniu ówczesnych rozwiązań i jednocześnie pozwala zauważyć brak postępu w tej dziedzinie.

Pierwszym dostępnym dla każdego systemem z graficznym interfejsem użytkownika, który odniósł komercyjny sukces, był komputer Macintosh 128K z systemem „System 1.0".

2.2 Koncepcje budowania środowisk graficznych

Do niedawna tworzono głównie interfejsy w oparciu o paradygmat WIMP *(ang. Window, Icon, Menu, Pointing device)*. Urządzeniem wskazującym pozycje kursora jest najczęściej myszka, manipulator kulkowy *(ang. trackball)* lub panel dotykowy *(ang. touchpad)*. Wszystkie informacje są organizowane w ramach okien *(ang. Windows)* i reprezentowane przez ikony *(ang. Icons)*. Możliwe komendy, które dostarczał system, prezentuje się i skupia w ramach menu. Akcji dokonuje się przy pomocy wspomnianych wcześniej urządzeń peryferyjnych. Wszystkie te elementy graficznego interfejsu użytkownika są organizowane w ramach pulpitu *(ang. Desktop)*. Ten nowatorski sposób prezentowania informacji jest bardzo intuicyjny dla użytkowników, szczególnie tych mniej zaawansowanych technicznie.

Oczywiście istnieje wiele strategii organizowania okien w ramach pulpitu. Najbardziej znane i zarazem najbardziej ogólne to:

- SDI - Single Document Interface
- MDI - Multi Document Interface

SDI jest to sposób prezentowania interfejsu przez rozbicie go na odrębne okna, które są oddzielnie traktowane przez system operacyjny. MDI zaś przewiduje jedno główne okno programu i wiele okien podrzędnych. Oba podejścia mają swoich zwolenników jak i przeciwników. W nowych wersjach danego oprogramowania na przestrzeni lat często zmieniano podejście z MDI na SDI i odwrotnie, więc nie można go definitywnie przypisać do konkretnej kategorii. Często zdarza się, że obie strategie łączy się ze sobą. Niektóre elementy obsługuje się tak, jakby to był program SDI, a inne w myśl strategii MDI.

Ten rodzaj graficznych interfejsów użytkownika nie nadaje się jednak do mniejszych urządzeń, takich jak odtwarzacze MP3, telefony itp. Wielkość wyświetlacza w tych urządzeniach jest zbyt mała i klasyczne podejście zajmowało bardzo dużą przestrzeń.

Również nowe sposoby interakcji użytkownika z systemem, takie jak ekrany dotykowe z funkcją multidotykową *(ang. Multi-Touch)*, również przyczyniły się do odmiennego podejścia przy budowie takich interfejsów. Wiele akcji interfejsu, które były standardowo reprezentowane przez graficzne elementy, zostały zastąpione przez specjalne komendy, wydawane na ekranie multidotykowym. Na przykład, powiększanie zdjęć zastąpiono przez zbliżenie dwóch palców na ekranie. Często definiuje się także różne akcje interfejsu graficznego, jeżeli, np., użyliśmy dwóch palców, a inne jeżeli tylko jednego. Takie zabiegi pozwalają zaoszczędzić dużo przestrzeni na niewielkiej powierzchni wyświetlacza urządzenia mobilnego. Znacząco poprawiają też intuicyjność takiej aplikacji. Akcelerometr jest kolejnym sensorem w nowoczesnych urządzeniach przenośnych, który pomaga w znamienny sposób zoptymalizować budowę mobilnych interfejsów użytkownika. Dzięki takim rozwiązaniom można zaoszczędzić miejsce, które zostałoby zużyte na elementy przeznaczone do obsługi akcji, które teraz są realizowane przez nowe sensory.

2.3 Sposoby budowania interfejsów użytkownika

Współcześnie wytwarzając oprogramowanie, programista ma do wyboru trzy podejścia do wytwarzania graficznych interfejsów użytkownika.[1]

2.3.1 Ręczne pisanie kodu

Programista ma pełną kontrolę nad wyglądem interfejsu. Jednak zmuszony jest pisać stosunkowo niskopoziomowy i powtarzalny kod źródłowy. Każda platforma programistyczna posiada swoje rozwiązania w tej dziedzinie. Oczywiście nie są one zgodne ze sobą i wymagają od programisty szczegółowego poznania API *(ang. Application Programming Interface)* danej biblioteki, przeznaczonej do budowania graficznych interfejsów użytkownika. To podejście jest zdecydowanie najbardziej czasochłonne. Wymaga ono od programisty skupienia na problemach interfejsu graficznego użytkownika, a nie na pisaniu kodu odpowiedzialnego za logikę biznesową aplikacji.

2.3.2 Wizualne edytory

Pozwalają one tworzyć interfejsy w myśl idei WYSIWYG *(ang. What You See Is What You Get)*. Z tą koncepcją wiąże się oszczędność czasu, przeznaczonego przez programistę na tworzenie danego interfejsu w porównaniu do czasu ręcznego pisania

kodu. Jednak to rozwiązanie ma swoje wady. Edytowanie kodu wygenerowanego przez taki edytor, prowadzi do utraty możliwości dalszego korzystania z wizualnego tworzenia interfejsu użytkownika. Wyjściem są edytory, które wspierają tzw. inżynierię wahadłową *(ang. round-trip)*. Dzięki temu można zinterpretować zmiany, wprowadzone przez programistę w wygenerowanym kodzie i odpowiednio je nanieść w edytorze. Niezależnie od tego, bardzo wątpliwa jest jakość tak wykreowanego kodu źródłowego. Jego czytelność i ponowne użycie często jest praktycznie niemożliwe.

2.3.3 Podejście deklaratywne

W ogólnym przypadku podejście deklaratywne do danego problemu programistycznego charakteryzuje się tym, że nie określa się schematu postępowania przy wdrażaniu rozwiązania (nie definiujemy algorytmu). Opisuje się jedynie cel, który należy osiągnąć, a stworzeniem rozwiązania zajmuje się język programowania lub ogólnie komputer. Dla kontrastu, programowanie imperatywne wymaga od programisty zdefiniowania dokładnego algorytmu dla zadanego problemu.

Rozpatrując graficzne interfejsy użytkownika, można stwierdzić, że w podejściu deklaratywnym abstrahuje się od algorytmu tworzenia interfejsu użytkownika w danej technologii (rodzaj używanych komponentów, sposób rozmieszczenia w oknie itp.), a opisuje się jedynie efekt jaki chcemy uzyskać.

2.3.4 Rzeczywiste wykorzystanie

Przy rozwiązywaniu prawdziwych problemów wykorzystuje się jednak najczęściej kombinację użycia edytora WYSIWYG i ręcznego pisania kodu. Inną opcją jest zastosowanie dostępnej technologii wspierającej tworzenie deklaratywne interfejsów użytkownika. Później poprzez ręczne pisanie kodu źródłowego, odwołującego się do bezpośrednio do API konkretnej biblioteki, odbywa się szczegółowa edycja i wprowadzanie niestandardowych zachowań do interfejsu.

2.4 Istniejące rozwiązania

Większość bibliotek programistycznych, służących do budowania graficznych interfejsów użytkownika, nie wspiera podejścia deklaratywnego. Aczkolwiek obserwuje się znaczne zmiany – coraz więcej komercyjnych produktów opiera się właśnie na tym podejściu.

Takie rozwiązania firmy Microsoft jak WPF *(ang. Windows Presentation Foundation)* i Silverlight, korzystają z deklaratywnego języka opartego o XML *(ang. Extensible Markup Language)* o nazwie XAML [2] *(ang. Extensible Application Markup Language)*. Innym przykładem jest firma Google, która w swoim zestawie narzędzi do tworzenia aplikacji internetowych - GWT [3] *(ang. Google Web Toolkit)* od wersji 2.0, wprowadziła wsparcie dla podobnego rozwiązania znanego z produktu Microsoftu, a mianowicie UiBinder[4]. Pozwala on również za pomocą specjalnego dialektu XML'a definiować w sposób deklaratywny interfejs użytkownika.

2.4.1 XAML - Extensible Application Markup Language

XAML[2] to język deklaratywny stworzony przez Microsoft, oparty o XML. Służy on do opisywania interfejsów użytkownika, ale wykorzystuje się go również w produkcie Microsoft odpowiedzialnym za workflow. Język ten stworzono w celu lepszego odseparowania części kodu źródłowego, odpowiedzialnego za interakcje z użytkownikiem (animacje, rozkład kontrolek, itp.), od kodu zajmującego się logiką biznesową programu.

Przeznaczenie

Został on dodany do platformy .NET od wersji 3.0. Stosuje się go do definiowania elementów interfejsu, zasilania go danymi *(ang. data binding)*, podłączania kodu zarządzającego zdarzeniami oraz opisywania animacji. Jest wykorzystywany w takich technologiach giganta z Redmond jak:

- WPF – Windows Presentation Foundation
- SilverLight
- WF – Windows Workflow Foundation

```
1  <Window x:Class="WpfApplication1.MainWindow"
2         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="MainWindow" Height="252" Width="519">
5      <Grid Height="194" Width="493">
6       <StackPanel Height="133" HorizontalAlignment="Left"
7                   Margin="12,12,0,0
8                   Name="stackPanel1" VerticalAlignment="Top"
```

```
10                      FlowDirection="LeftToRight" Orientation="Vertical">
11          <Label Content="Label" Height="28" Name="label2" />
12          <Button Content="Button" Height="23" Name="button2" Width="75"/>
13          <CheckBox Content="CheckBox" Height="16" Name="checkBox1" />
14       </StackPanel>
15      </Grid>
16  </Window>
```

Listing 1. Przykładowy dokument XAML

Główną siłą XAML miała być możliwość synchronizacji pracy grafika, osoby odpowiedzialnej za wygląd aplikacji z programistą. Zadaniem programisty miało być tworzenie logiki aplikacji, a nie praca nad jej wyglądem. Oczywiście taki podział w realnych warunkach okazał się trudny do wyegzekwowania. Programista często staje się grafikiem i twórcą wyglądu aplikacji. Wymagania biznesowe także należy odpowiednio wyrazić w interfejsie użytkownika. Oczywiście wprowadzenie XAML przez Microsoft jest ogromnym postępem w dziedzinie tworzenia graficznych interfejsów użytkownika na platformę .NET. Graficzne edytory generują tylko kod XAML, pomijają zaś kod C# lub VisualBasic.NET (nieczytelny nawet dla programisty znającego dany język programowania).

Rysunek 1. Efekt działania pliku XAML

XAML wspiera również różne style. Dzięki temu możemy w bardzo łatwy sposób przygotować specjalne wersje danej aplikacji, nie modyfikując jej struktury, a jedynie dodając nowy szablon *(ang. template)*. Idea tych szablonów przypomina kaskadowe arkusze stylów *(ang. Cascading Style Sheet – CSS)*. Składnia oraz inne elementy, jak dziedziczenie stylów, są odmienne od podejścia znanego z CSS.

```
<Style TargetType="TextBlock">
            <Setter Property="HorizontalAlignment" Value="Center" />
            <Setter Property="FontFamily" Value="Comic Sans MS"/>
            <Setter Property="FontSize" Value="14"/>
</Style>
<Style BasedOn="{StaticResource {x:Type TextBlock}}"
    TargetType="TextBlock"
    x:Key="Header">
  <Setter Property="FontSize" Value="26"/>
  <Setter Property="Foreground">
  <Setter.Value>
      <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
        <LinearGradientBrush.GradientStops>
          <GradientStop Offset="0.0" Color="#90DDDD" />
          <GradientStop Offset="1.0" Color="#5BFFFF" />
        </LinearGradientBrush.GradientStops>
      </LinearGradientBrush>
    </Setter.Value>
  </Setter>
</Style>
```

Listing 2. Przykład budowy stylów w XAML.

Styl przedstawiony na Listingu 2 opiera się na stylu stworzonym dla typu TextBlock i definiuje atrybut x:Key. Dzięki temu można dokładnie przypisać omawiany styl konkretnej kontrolce. Przykład podpięcia tego stylu, widzimy na Listingu 3:

```
<TextBlock Style="{StaticResource Header}" Name="Header">
    Moj Naglówek
</TextBlock>
<TextBlock>Witam w moim œwiecie</TextBlock>
```

Listing 3. Aplikowanie stylów w dokumentach XAML

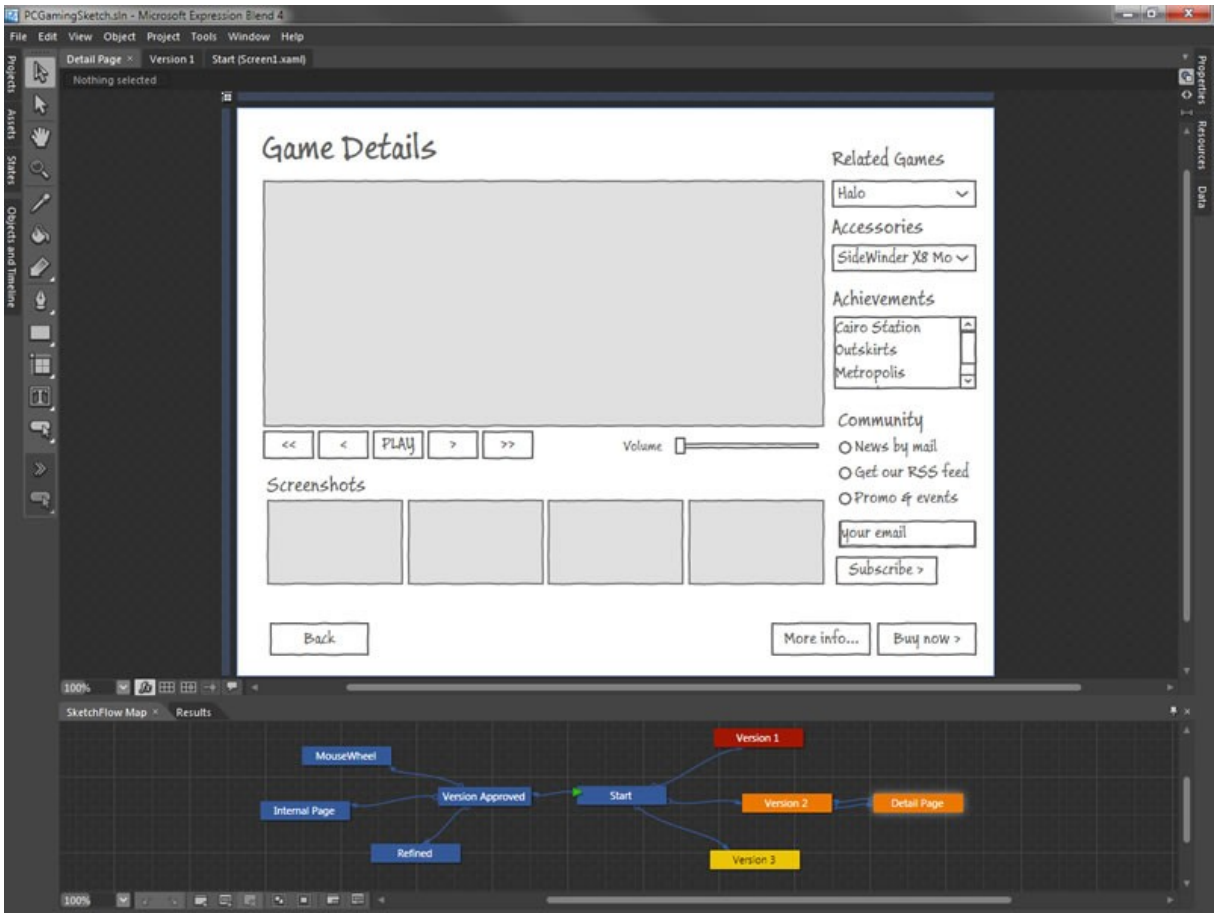
Dzięki takiemu rozdzieleniu struktury interfejsu od jego właściwości wizualnych dostajemy bardzo potężne narzędzie. Pozwala ono w łatwy sposób zarządzać wyglądem danych komponentów z jednego centralnego miejsca i współdzielić ich wygląd między sobą.

Rysunek 2: Rezultat działania stylów w aplikacji WPF

Praca z XAML

Wszystkie definicje elementów graficznych w XAML odnoszą się w stosunku jeden do jednego do obiektów CLR *(ang. Common Language Runtime)*. Atrybuty zaś przechodzą we właściwości *(ang. properties)* tych obiektów albo w zdarzenia *(ang. events)*.

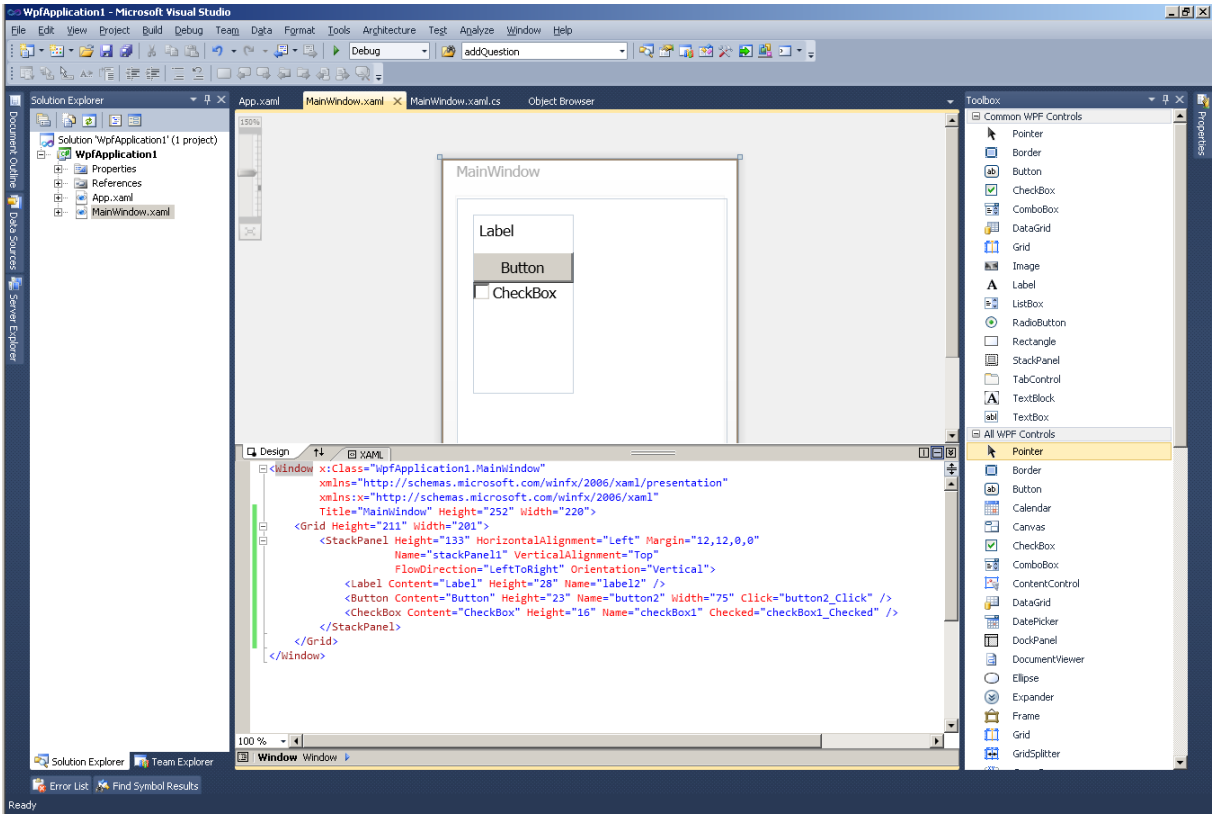
Pliki XAML edytuje się w specjalnych graficznych edytorach, przeznaczonych do pracy w trybie WYSIWYG. Do takich narzędzi należy Microsoft Expression Blend, który jest oddzielną aplikacją przeznaczoną do budowy zaawansowanych interfejsów użytkownika przy użyciu metody drag&drop. Dzięki niej można formułować animacje. Możliwe jest również proste zdefiniowanie Transformacji 3D, korzystając z danego produktu. Blend posiada też narzędzie do szybkiego prototypownia interfejsu o nazwie SketchFlow (Rysunek 3.). Służy on do szybkiego przedstawienia rozmieszczenia komponentów klientowi, ale również do prezentacji konkretnych sytuacji i przepływów. Taki prototyp jest prawdziwym projektem WPF lub SilverLight, dlatego można go z łatwością użyć w dalszej pracy nad interfejsem.



Rysunek 3: Microsoft Expression Blend - SketchFlow

Drugim narzędziem przeznaczonym przede wszystkim dla programistów, nie zaś grafików, jest wbudowany edytor drag&drop Visual Studio 2010. Nie dostarcza on tak zaawansowanych funkcji, jak Blend, jednak w początkowej fazie budowy oprogramowania zdecydowanie wystarcza. Visual Studio 2010 jest w IDE *(ang. Integrated development environment)* przeznaczony dla platformy .NET. Przykład działania widzimy na rysunku 4. Wspiera wytwarzanie oprogramowania w każdym etapie. Dostarcza rozwiązań przy analizie problemu, wytwarzaniu kodu i testowaniu (jednostkowe, obciążeniowe, itp.) oraz wdrożeniu gotowego produktu. Praca z XAML w trybie wizualnym w Visual Studio 2010 bardzo przypomina pracę z chwalonym edytorem przeznaczonym dla technologii WinForms. Edytor ten powinien być znany każdemu programiście, który kiedykolwiek pracował na platformie .NET. XAML zyskuje dużą popularność, więc korzystanie z zasobów nowej technologii staje się dużo łatwiejsze.

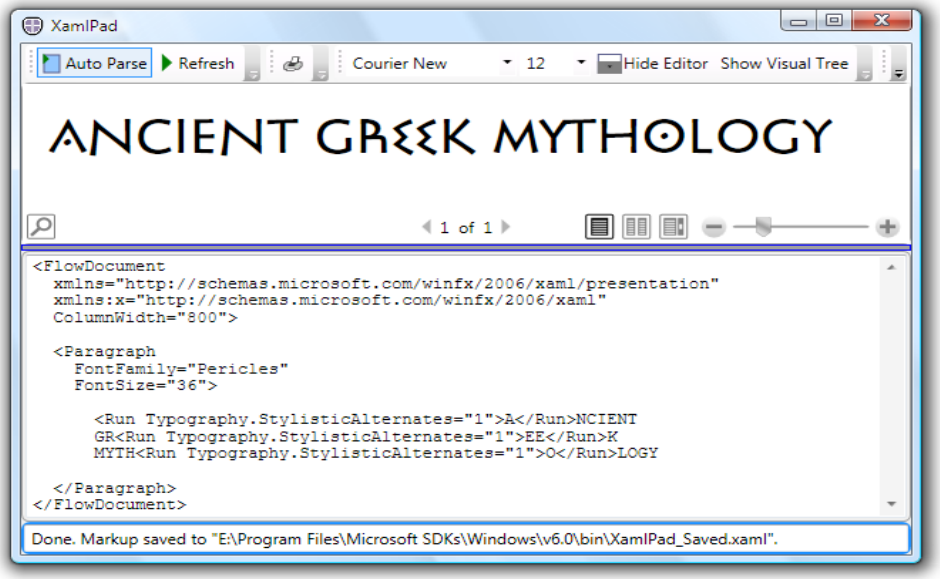
Największą zaletą obu produktów jest inżynieria wahadłowa *(ang. round -trip)*, która pozwala na edycję kodu XMAL bezpośrednio i oglądanie tych rezultatów w edytorze wizualnym.



Rysunek 4: Visual Studio 2010

Kolejnym sposobem pracy z XAML jest użycie tekstowych edytorów, takich jak XamlPad. Jak widać na rysunku 5., jest to prosty edytor instalowany wraz z Windows SDK. Otrzymuje się przyjemne w użytkowaniu oraz szybkie środowisko, które nadaje się do prototypowania, lecz kwestionuje się jego zastosowanie w budowie interfejsów użytkownika. Edytor umożliwia szybkie sprawdzenie poprawności kodu XAML. Wskazuje linie, które są niepoprawne. Otrzymuje się podświetlanie składni, co w znacznym stopniu zwiększa komfort pracy z programem. Pozwala również przedstawić kod w formie drzewa. Dzięki temu można z łatwością przeglądać właściwości elementów. Chociaż wykazuje brak takich oczywistych funkcji, jak:

- Otwórz plik – Jedynie przez podanie jako argumentu przy uruchamianiu xamlpad.exe możliwe jest załadowanie gotowego już pliku XAML.
- Zapisz jako – Zawsze zapisuje edytowany plik jako XamlPad_save.xaml



Rysunek 5: XamlPad

Zalety i Wady

XAML przyczynia się do ogromnego postępu przy budowie interfejsów użytkownika. Dzięki wykorzystaniu tej technologii można w prostszy sposób utrzymywać kod odpowiedzialny za interfejs użytkownika. Czytając kod XAML, dostrzega się ogólną wizję struktury i hierarchii tworzonego interfejsu. Dzięki takim technikom, jak style i szablony kontrolek *(ang. Control templates)*, można opracować łatwiejszą i rozszerzalną metodę budowania graficznych interfejsów użytkownika.

Największą wadą technologii XAML jest poziom abstrakcji, który uzyskuje programista decydujący się na użycie tej technologii. Musi on definiować dokładnie te same elementy, które umieściłby w kodzie C#. W związku z tym musi wykonać, praktycznie rzecz biorąc, tyle samo pracy niezależnie od tego, czy wykorzysta podejście deklaratywne (XAML), czy bezpośrednie użycie API przez kod C#. Więc główna zaleta podejścia deklaratywnego, czyli ekspresyjność języka jest tutaj wątpliwa. Wiąże się z tym kolejny problem wykorzystania XAML, pliki po pewnym czasie stają się bardzo nieczytelne.

Początkowo kod XAML był kompilowany do kodu pośredniego *(ang. Intermediate Language (IL))*. Jednak zrezygnowano z tego rozwiązania z takich powodów jak:

- Bezpieczeństwo wykonania takiego kodu.

- Wszelkie zmiany w czasie wykonania wymagałyby kompilacji na nowo. Chodzi tu przede wszystkim o lokalizowanie aplikacji.

Z racji tego wprowadzono BAML *(ang. Binary Application Markup Language)*, jest to pośrednia binarna forma, poddana tokenizacji (długie wyrażenia XAML zastępowane są krótszymi tokenami – plik .baml jest znacznie mniejszy od .xaml) wersja pliku XAML. Jest on interpretowany w czasie wykonania programu, przez to niektóre błędy ujawniają się dopiero na tym etapie. Również komunikaty są często mało zrozumiałe i nie pomagają w odnalezieniu przyczyny problemu.

Rozdzielenie prac programisty i grafika nie jest do końca przemyślane i poprawnie wykonane. Obaj muszą pracować na jednym pliku, dlatego często prowadzi to do niewłaściwego zrozumienia i zmian rzeczy, które nie należą do ich obowiązków.

Rownież narzędzia nie są dopracowane, przez co większość programistów pisze kod ręcznie.

2.4.2 JavaFX

JavaFX[5] jest to platforma, zbiór rozwiązań i technologii, przeznaczona głównie do budowy tak zwanych aplikacji typu RIA *(ang. Rich Internet Application)*. Została ona stworzona przez firmę Sun Microsystems (obecnie Oracle). Platforma konkuruje z takimi rozwiązaniami, jak Adobe Flex, Microsoft Silverlight, jeżeli rozważa się aplikacje webowe i Microsoft WPF, Adobe AIR oraz OpenLaszlo, które zajmują się aplikacjami uruchamianymi poza przeglądarką internetową. Aktualnie w skład tej platformy wchodzą dwa rozwiązania:

- JavaFX Script – Język skryptowy
- Java ME – System przeznaczony na urządzenia przenośne

JavaFX Script

JavaFX Script jest to deklaratywny, skryptowy język, wykorzystywany na platformie JavaFX. Dzięki statycznemu typowaniu i deklaratywnemu podejściu pomaga w wytwarzaniu oprogramowania. Statyczne typowanie pozwala odkryć dużo błędów już w czasie tworzenia kompilacji. Podejście deklaratywne stwarza dużo bardziej przejrzysty kod źródłowy, dzięki czemu osiąga się większą kontrolę nad danym programem. Poza tym, każdą aplikację JavaFX kompiluje się do bytecode Javy. W związku z tym, łatwo się domyślić, że aplikacje z platformy JavaFX uruchamiane są w ramach wirtualnej maszyny Javy. Wymagana jest

co najmniej wersja 5.0. Jednak, żeby w pełni dostrzec szybkość działania i całkowicie wykorzystać potencjał platformy JavaFX, zaleca się używanie najnowszej wersji Javy – 6.0.

```
Stage {
    title: "Aplikacja JavaFX Script"
    scene: Scene {
        content: [
            SwingButton {
                  text: "Przycisk"
            }
            Text {
                font : Font {
                          size : 16
                }
                x: 10
                y: 40
                content: "Witam w moim swiecie"
            }
        ]
    }
}
```

Listing 4. Kod źródłowy przykładowej aplikacji wykonanej w JavaFX Script

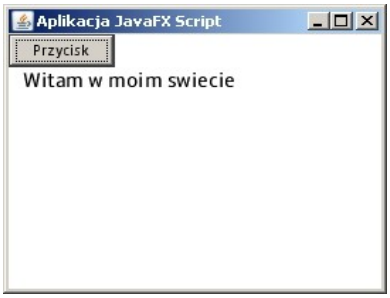
W listingu 4 przy użyciu JavaFX Script definiuje się okno, które posiada dwie kontrolki SwingButton i Text. W pierwszej chwili język wydaje się niejasny i mało intuicyjny. Przypomina trochę definicję obiektów w języku JavaScript w notacji JSON *(ang. JavaScript Object Notation)*. Jeżeli jednak ten sam interfejs opisze się w sposób „klasyczny" – wykorzystując programistyczne API. Taki sposób znany jest z programowania aplikacji m.in. WinForms na platformie .NET, jak również aplikacji Swing, które są tworzone w Javie. Okaże się, że deklaratywne podejście wykazuje dużą przejrzystość i ekspresyjność. Deklaratywny zapis JavaFX Script prowadzi do większej kontroli nad daną aplikacją. Listing 5. przedstawia kod napisany w sposób klasyczny:

```
var myStage:Stage  = new Stage ();
myStage.title = "Aplikacja Java FX";
var myScene:Scene = new Scene();
```

```
4  var myLabel:Text = new Text();
5  myLabel.content = "Witam w moim swiecie";
6  myLabel.x = 10;
7  myLabel.y = 40;
8  var myFont:Font = Font.font("", 16);
9  myLabel.font = myFont;
10 var myButton:SwingButton = new SwingButton();
11 myButton.text = "Przycisk";
12 myScene.content = [myLabel,myButton];
13 myStage.scene = myScene;
14 myStage.visible = true;
```

Listing 5. Kod JavaFX napisany w stylu programowania aplikacji w Javie dla Swing.

Kod z listingu 5. można lepiej sformatować, a nawet zmniejszyć jeszcze jego rozmiar. Jednak mimo tego, podejście deklaratywne jest bardziej czytelne i spójne. W związku z tym, ponowne odniesienie się przez programistę do tego kodu po pewnym czasie, będzie bardziej dogodne przy korzystaniu z wersji deklaratywnej. Z założenia wersja ta opisuje wygląd interfejsu graficznego, a nie szczegółowy schemat jego wytworzenia.

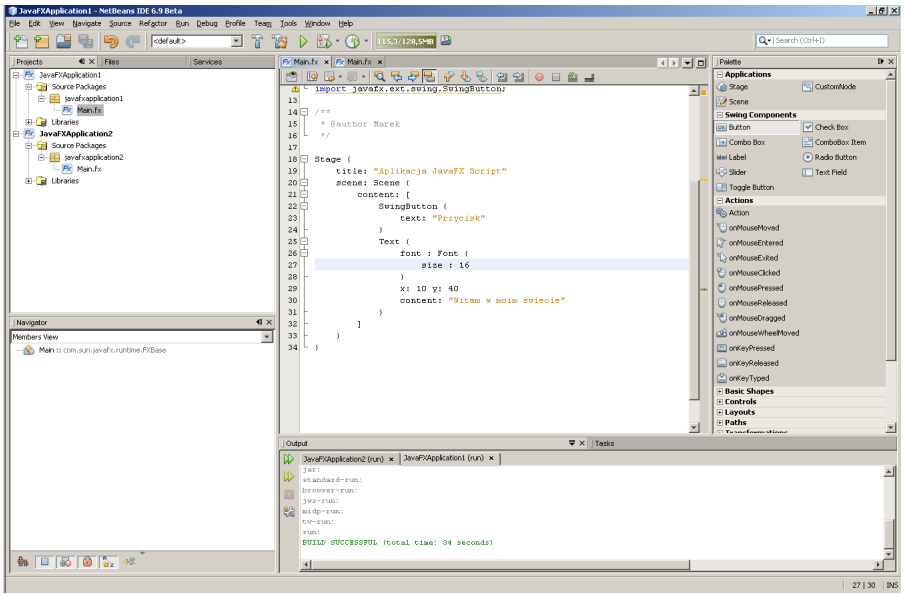


Rysunek 6: Aplikacja utworzona przy użyciu JavaFX Script

Rezultat działania obu rozwiązań będzie identyczny. Przedstawia go Rysunek 6. Oczywiste jest, że pierwszy sposób jest dużo bardziej praktyczny i zdecydowanie bardziej czytelny.

Przedstawiona aplikacja oczywiście jest bardzo prosta. Jednak przy pomocy JavaFX Script można budować bogate w multimedia, animacje oraz efektowne kontrolki aplikacje.

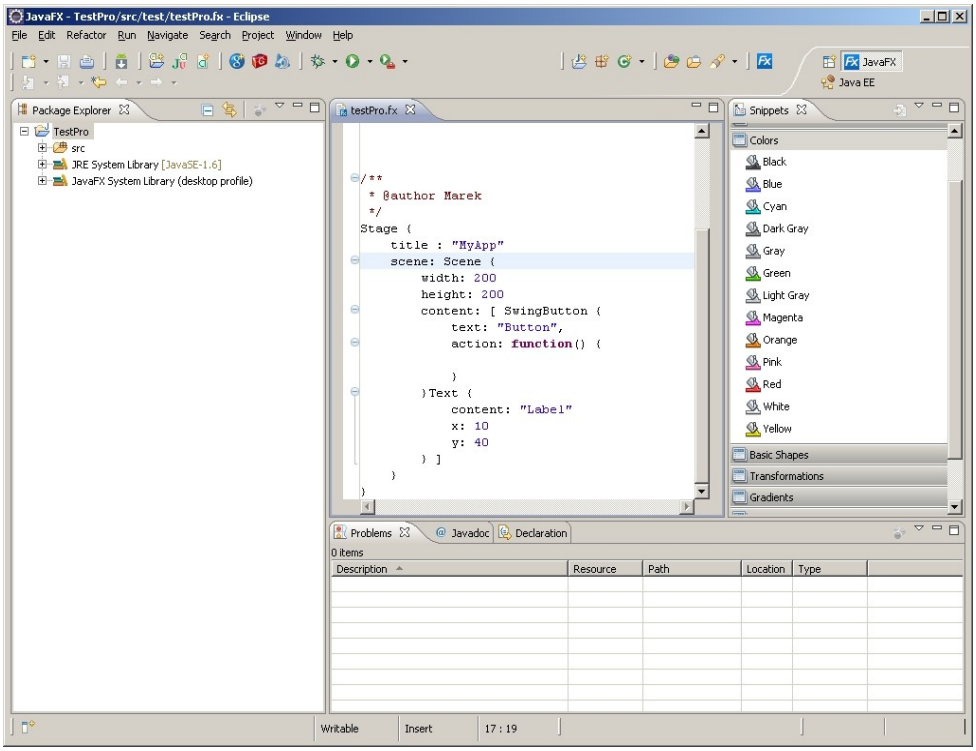
Praca z platformą JavaFX



Rysunek 7: NetBeans 6.9 i kod JavaFX Script

Platforma JavaFX nie posiada rozwiązania dedykowanego grafikowi. Inaczej jest w przypadku produktów firmy Adobe czy Microsoft. Zamiast tego, dostarcza wtyczki do popularnych edytorów graficznych, znanych doskonale każdemu, kto zawodowo tworzy grafikę. Do grona tych produktów można zaliczyć sztandarowe produkty firmy Adobe – Photoshop i Illustrator.

Zalecanym środowiskiem programistycznym dla platformy JavaFX jest NetBeans w wersji 6.9. Rysunek 7. przedstawia kod JavaFX Script w tym IDE. Jest to potężny edytor z podpowiadaniem i kolorowaniem składni dla języka JavaFX Script. Dodatkowo można skorzystać z wizualnego edytora służącego do budowania graficznych interfejsów użytkownika. Jednak dużą wadą tego rozwiązania jest brak pracy w trybie inżynierii wahadłowej *(ang. round-trip)*. Przez tą niedogodność nie jest możliwe edytowanie ręczne wygenerowanego kodu i oglądanie rezultatów w oknie wizualnego edytora.



Rysunek 8: Eclipse 3.5 wraz ze wsparciem dla JavaFX

Istnieje również inne środowisko pracy programisty związane z platformą JavaFX. Przy użyciu odpowiedniej wtyczki, można skorzystać z bardzo dobrego i popularnego IDE – Eclipse (Rysunek 8.). Jednak oficjalny plugin do Eclipse dostarczany przez twórcę platformy wspiera jedynie wersje 1.2.1 SDK. Obecnie dostępna jest już wersja 1.3 SDK, która naprawia wiele błędów i wprowadza ulepszenia w szybkości i stabilności działania aplikacji JavaFX.

Wtyczka do Eclipse nie dostarcza wizualnego edytora do tworzenia interfejsu graficznego. W zamian otrzymuje się przybornik z tak zwanymi „snippet'ami". Są to gotowe fragmenty kodu JavaFX Script, które, używając techniki drag&drop, można umieścić w danym kodzie źródłowym.

Zalety i Wady

Platforma JavaFX jest jeszcze bardzo młoda i przez to znacznie mniej popularna, niż inne konkurencyjne rozwiązania. Jakość narzędzi dostarczanych wraz z platformą także nie jest najlepsza, przez co wykazują one duże braki w stosunku do kompletnych rozwiązań dostępnych na rynku.

Ogromną zaletą jest możliwość łatwej integracji rozwiązań napisanych w Java z platformą JavaFX. W danej aplikacji możliwe jest stosowanie gotowych komponentów znanych ze świata Java.

2.4.3 UiBinder

UiBinder[4] stanowi nowy sposób na budowanie interfejsów użytkownika na platformie Google Web Toolkit. Jest to oparty o XML język, który w deklaratywny sposób pozwala na budowę graficznych interfejsów użytkownika. Ułatwia on adaptowanie gotowych szablonów zbudowanych w HTML/CSS do rozwiązań opartych o GWT. Wspiera budowanie aplikacji wielojęzykowych. Zwiększa także szybkość działania aplikacji. Budując interfejs przy użyciu UiBindera, dużo łatwiej wykorzystać „lekkie" elementy znane z HTML, jak div itp., niż stosować „ciężkie" komponenty GWT. Zwiększając odwołania do API dostarczanego przez GWT, zmniejsza się szybkość rysowania danego interfejsu przez przeglądarkę internetową.

Praca z UiBinderem

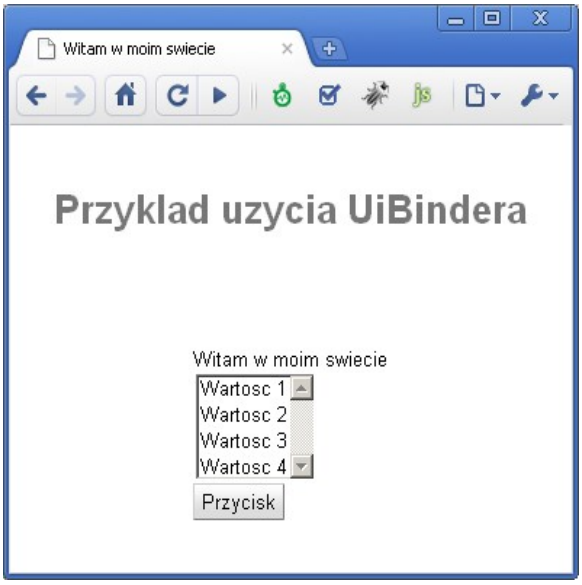
Stosując tę technologię, nie dostaniemy do pracy graficznych edytorów. Jedyną opcją jest korzystanie z podpowiedzi składni w trakcie pisania kodu w edytorze XML środowiska Eclipse. Jednak UiBinder oferuje dogodniejsze warunki pracy niż przy budowie interfejsów użytkownika na platformie GWT w tradycyjny sposób. Przypomina on programowanie aplikacji Swing/AWT w Javie.

```
1  <!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.ent">
2  <ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
3         xmlns:g="urn:import:com.google.gwt.user.client.ui">
4          <g:VerticalPanel>
5                  <g:Label>Witam w moim swiecie</g:Label>
6                  <g:ListBox ui:field="listbox"
7  visibleItemCount='2'></g:ListBox>
8                  <g:Button ui:field="button">Przycisk</g:Button>
9          </g:VerticalPanel>
10 </ui:UiBinder>
11 ...
```

```
5   ...
6   public class Layout extends Composite {
7       interface LayoutUiBinder extends UiBinder<Widget, Layout> {}
8       private static LayoutUiBinder uiBinder =
9   GWT.create(LayoutUiBinder.class);
10      @UiField
11      Button button;
12
13      @UiField
14      ListBox listbox;
15
16      public Layout() {
17            initWidget(uiBinder.createAndBindUi(this));
18            for (int i = 1; i < 5; i++) {
19                  listbox.addItem("Wartosc " + i, "Wartosc " + i);
20            }
21      }
22
23      @UiHandler("button")
24      void onClick(ClickEvent e) {
25        Window.alert("Hello!");
26      }
27  }
```

Listing 6. Przykładowy plik XML i klasa obsługująca wykorzystywane przez UiBinder – Layout.ui.xml i Layout.java

Jak widać na listingu 6., składnia UiBindera jest w swoim założeniu podobna do innych deklaratywnych języków opartych o XML, m.in. do XAML. Zaletą UiBindera jest to, że wszystkie style definiuje się, używając notacji CSS. Dostarcza to ogromnych możliwości wykorzystania gotowych już stylów w pracy nad wyglądem danego programu. Od początku ten sposób tworzenia graficznych interfejsów użytkownika był reklamowany przez Google jako połączenie HTML z komponentami GWT w jednym miejscu. UiBinder dostarcza medium, które jest bardziej zrozumiałe dla osób odpowiedzialnych za wygląd danej strony, niż kod źródłowy Javy. Graficy/projektanci pracują na poznanych przez nich technologiach, jak HTML i CSS. Dzięki temu w bardzo przejrzysty sposób rozdziela się deklaratywny interfejs od logiki aplikacji zawartej w kodzie napisanym w języku Java.



Rysunek 9: Przykład działania UiBindera

Połączenie deklaratywnego XML'a UiBindera z kodem Java odbywa się przez zdefiniowanie odpowiedniej klasy z zachowaniem poprawnego nazewnictwa. Klasa „łącząca" musi nazywać się tak samo jak plik .ui.xml. W tym przypadku plik UiBinder'a nazywa się Layout.ui.xml, więc klasa musi być zdefiniowana jako Layout. Listing 6. przedstawia przykładową implementację takiej klasy.

Warto zwrócić uwagę na zastosowane adnotacje, które są niezbędne i „łączą" kod XML z kodem Java. Przy użyciu adnotacji @UiField definiuje się zmienną, której nazwa odpowiada tej użytej w XML w atrybucie ui:filed elementu ListBox, Button itp. Dzięki temu można odwoływać się do tych kontrolek i je modyfikować. Tak jak ma to miejsce w kodzie konstruktora klasy Layout, gdzie kontrolka o nazwie listbox (typ ListBox) wypełniana jest danymi. Kolejną ważną adnotacją jest @UiHandler(String attr). Dzięki niej w bardzo łatwy i elegancki sposób definiuje się metody odpowiedzialne za obsługę zdarzeń dostępnych w ramach danej kontrolki - w tym konkretnym przypadku zdarzenie kliknięcia na kontrolce typu Button. Nazwa przekazana jak argument adnotacji @UiHnadler(Stirng attr) musi odpowiadać tej użytej w atrybucie ui:filed w pliku ui.xml.

```
1  VerticalPanel vp = new VerticalPanel();
2  ListBox listbox = new ListBox();
3  listbox.setVisibleItemCount(2);
```

```
4  Label lb = new Label();
5  lb.setText("Witam w moim swiecie");
6  Button btn = new Button();
7  btn.setText("Przycisk");
8  vp.add(lb);
9  vp.add(listbox);
10 vp.add(btn);
```

Listing 7. Tworzenie interfejsów graficznych w GWT bez wykorzystania UiBindera

Kod z listingu 7. jest analogiczny do zaprezentowanego na listingu 6. Jednak od razu można dostrzec, tak jak w przypadku JavaFX Script, sens zastosowania podejścia deklaratywnego w określaniu graficznego interfejsu użytkownika. Dzięki wykorzystaniu definiowania w sposób deklaratywny w pliku XML, otrzymuje się dodatkową przestrzeń na połączenie danej pracy z działaniem projektantów interfejsów, którzy nie muszą posiadać wiedzy na temat programowania w Javie. Rysunek 9. przedstawia działanie obu podejść w przeglądarce internetowej. Na obu listingach pominięto część wypełniania kontrolki ListBox wartościami.

Zalety i Wady

UiBinder to znakomite rozszerzenie możliwości platformy GWT. Zwiększa on szybkość tworzenia interfejsów graficznych. Pozwala na łatwiejszą integrację z istniejącymi stronami stworzonymi w technologii HTML/CSS. Dzięki UiBinderowi ponowne użycie stworzonych komponentów jest dużo prostsze, dlatego, że wygląd przechowuje się w oddzielnym miejscu, niż logika działania naszej aplikacji/komponentu. Również dzięki takiemu odseparowaniu kod źródłowy danej aplikacji staje się dużo bardziej przejrzysty i łatwiejszy w utrzymaniu. Wszystkie powtarzające się części, odpowiedzialne za wygląd i rozmieszczenie, umieszczone są w XML UiBindera.

2.4.4 GCL - GUI Creating Language

GCL[6] jest to deklaratywny DSL *(ang. Domain Specific Language)* stworzony w języku Java przez dr. inż. Mariusza Trzaskę. Język powstał w trakcie badań prowadzonych w Polsko-Japońskiej Wyższej Szkole Technik Komputerowych. Dotyczyły one automatyzacji budowania graficznych interfejsów użytkownika w oparciu o model danych.

W odróżnieniu od innych komercyjnych rozwiązań, GCL zbudowany jest w Javie, więc nie wymaga żadnych interpreterów, które będą go „parsować".Wszystko odbywa się to w bibliotece napisanej w Java, są to odpowiednio nazwane i połączone klasy. Jednak dzięki odpowiedniej architekturze otrzymano narzędzie bardzo wygodne w użyciu. Podobne właściwości można osiągnąć, modyfikując jedynie kompilator konkretnego języka. Mimo, że GCL napisano w Javie, jego składnia jest bardziej ogólna i wolna od Javy, przez co implementacja na innej platformie nie byłaby problemem.

Stworzony język różni się w znacznym stopniu koncepcją opisu interfejsu użytkownika od istniejących podejść, które wprowadzają deklaratywne interfejsy użytkownika. Nie wymaga to od programisty określania dokładnie z jakich komponentów ma być zbudowany interfejs. Stanowi to największą różnicę w porównaniu z komercyjnymi produktami. Innymi słowy, ukrywa szczegóły implementacji interfejsu graficznego przed programistą. W technologiach, takich jak XAML czy UiBinder, niezbędne jest dokładne określanie, z jakich komponentów ma być zbudowany interfejs użytkownika. Co tak naprawdę jest tylko inną notacją. W tych dwóch przypadkach przenosi się definicje z kodów źródłowych C# czy Javy do plików XML, które są jeden do jednego tłumaczone na odpowiednie odwołania API poszczególnych platform. W GCL abstrahuje się od implementacji i opisuje tylko wynik działania. Czyli można powiedzieć, że to podejście jest naprawdę deklaratywne.

Praca z GCL

Język GCL nie wymaga żadnego graficznego edytora. Do pracy z nim wystarczy środowisko wspierające tworzenie oprogramowania w Javie, które będzie przydatne przy podpowiedziach składni.

```
1  public class Person {
2  ...
3                  private String firstName;
4                  private String lastName;
5                  private Date birthDate;
6                  private boolean higherEducation;
7                  private String remarks;
8                  private int SSN;
9                  private double annualIncome;
```

```
10  ...
11  }
```

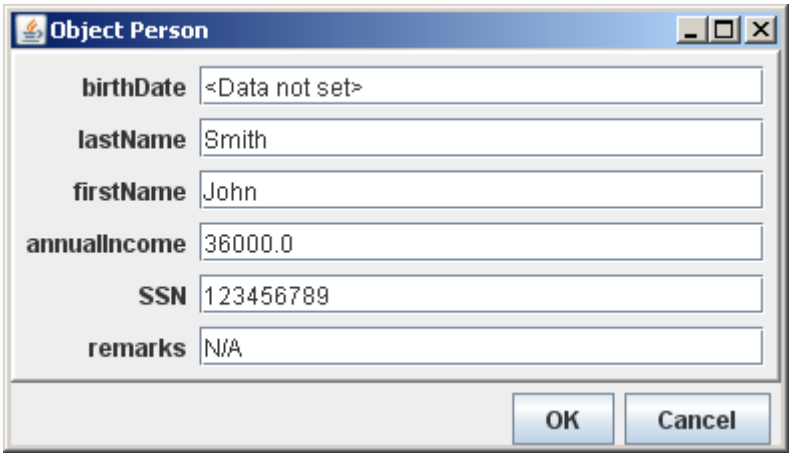
Listing 8. Klasa modelu danych użyta do generowania GUI przy pomocy GCL[7]

Listing 8. przedstawia klasę reprezentującą model danych. W przykładzie pominięto kod, za wyjątkiem atrybutów klasy. Jest to zwykła klasa Javy, zwana często POJO *(ang.Plain Old Java Object).* Główną siłą języka GCL jest to, że w trakcie korzystania z jego możliwości nie trzeba zmieniać danego modelu, np., przez dodanie specjalnych adnotacji.

```
1   JFrame frame1 =create.
2                  frame.
3                  usingOnly(person);
4
5   JFrame frame2 =create.
6                  frame.
7                  using(person).
8                  containing();
9
10  frame1.setVisible(true);
11  frame2.setVisible(true);
```

Listing 9. Proste użycie języka GCL do stworzenia formularza.[7]

Listing 9. prezentuje możliwie najprostsze użycie języka/biblioteki GCL. Biblioteka przy użyciu refleksji sama rozpoznaje jakie atrybuty posiada klasa, w tym przypadku siedem atrybutów. Korzystając z tych danych, sama rozmieszcza i generuje komponenty odpowiedzialne za prezentacje danych. Odpowiednio je nazywa i oczywiście również wypełnia je danymi przechowywanymi w instancji klasy Person. Efekt działania widać na rysunku.



Rysunek 10: Formularz wygenerowany przez bibliotekę GCL

Dodatkowo otrzymuje się automatyczne połączenie wygenerowanego interfejsu użytkownika z instancją klasy Person. Jeżeli zmieni się konkretną wartość w formularzu i zatwierdzi go przyciskiem „OK", zmiany te zostaną zapisane w danym modelu. Jest to bardzo przydatna i znacznie przyspieszająca wytwarzanie oprogramowania funkcja biblioteki GCL.

Etykiety przy polach tekstowych mają takie same nazwy jak atrybuty w klasie Person. GCL daje możliwość łatwej zmiany tych etykiet. Można również określić, które atrybuty, jak również metody, powinny się wyświetlić w danym interfejsie.

```
1  JFrame frame =
2            create.
3            frame.
4            using(person).
5            containing(
6                  attribute("firstName").as("First name"),
7                  attribute("lastName").validate(new ValidatorNotEmpty()),
8                  attribute("higherEducation"),
9                  method("getAge").as("Age"));
10 frame.setVisible(true);
```

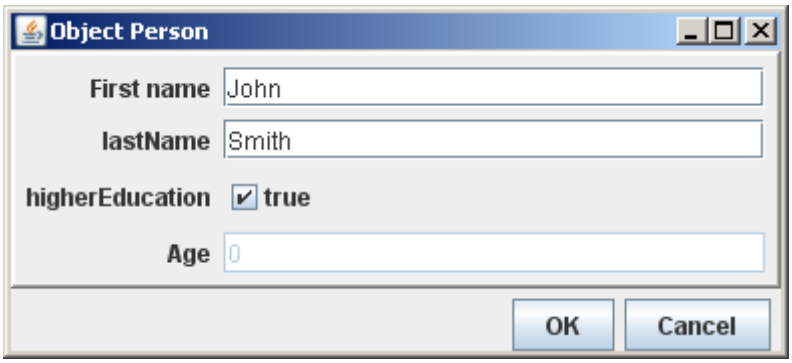
Listing 10. Bardziej zaawansowany przykład wykorzystania GCL[7]

Na listingu 10. widać, w jaki sposób można określać wygląd naszego formularza. Warto zauważyć, że nie jest ważny sposób, w jaki należy wyświetlić atrybut typu String-

fristName, a w jaki atrybut typu Boolean – higherEducation. Cały ten proces wykonuje GCL. Zapis jest zwięzły i bardzo ekspresyjny.

Jak widać na rysunku 11. atrybuty typu String zostały zaprezentowane przez pola tekstowe, zaś atrybut higherEducation typu Boolean reprezentuje kontrolka checkbox.

Również na listingu 10. przedstawiono bardzo użyteczną funkcjonalność biblioteki



Rysunek 11: Efekt działania kodu z listingu 10

GCL – mianowicie walidację. Do każdego pola w interfejsie można podłączyć obiekt walidacyjny, który będzie dbał o poprawność wprowadzonych danych. GCL dostarcza gotowe walidatory, jednak napisanie przez programistę nowego jest bardzo proste. Listing 11. przedstawia przykładowy kod klasy walidacyjnej.

```
1  public class ValidatorNotEmpty implements Validator {
2
3      public String isInvalid(String labelName, String enteredData) {
4            if(enteredData == null || enteredData.length() < 1) {
5                  return labelName + " cannot be empty";
6            }
7            return null;
8      }
9  }
```

Listing 11. Kod klasy walidacyjnej użytej w listingu 10 [7].

Zalety i Wady

GCL to nowatorski sposób tworzenia graficznych interfejsów użytkownika. Programista zyskuje silne narzędzie, które znacząco przyspiesza jego pracę. Dzięki takim możliwościom jak:

- Wielojęzyczność
- Wbudowana walidacja
- Automatyczna obsługa atrybutów złożonych – asocjacje między klasami modelu.

Możliwość stworzenia skomplikowanego interfejs użytkownika w zaledwie kilku linijkach, nie tracąc przy tym czytelności takiego kodu jest największą zaletą GCL. Składnia GCL jest bardzo prosta i jest „samo-tłumacząca”

Kolejnym ważnym udogodnieniem jest to, że GCL napisano całkowicie w Javie, więc dodatkowo zyskuje się całą moc znanych IDE takich jak Eclipse czy NetBeans.

2.4.5 Przyszłość – inne rozwiązania.

Branża IT jak też ośrodki akademickie szukają właściwych rozwiązań odnośnie interfejsów użytkownika. Podejście deklaratywne nie musi wcale okazać się najlepsze.

Podejścia mające na celu stworzenie tak zwanych „Bussines readable domain-specific-language” są warte uwagi. Jest to język, w którym osoby niezwiązane technicznie mogą bez problemu czytać i rozumieć, co chce wyrazić programista. Niewymagana jest także umiejętność pisania w omawianym języku. Tworzy się obszar, w którym może łączyć się programowanie ze światem biznesowym. Jednym z takich rozwiązań jest biblioteka napisana w języku Ruby – Cucumber[8]. Pozwala ona na tworzenie oprogramowania w myśl podejścia BDD[9] - „Behavior Driven Development„. BDD skupia się na opisywaniu wewnętrznej logiki aplikacji, więc bezpośrednio nie nadaje się do tworzenia graficznych interfejsów użytkownika. Jednak taki sposób, który zwiększa współpracę osób niezwiązanych technicznie (a znających biznesowe podłoże problemu) z programistami w celu lepszego wyspecyfikowania interfejsu wydaje się bardzo ciekawą, niestety – obecnie, jedynie teorią. Warto w tym miejscu zwrócić uwagę, że w pewien sposób składnia języka GCL jest zgodna z „Bussines readable domain-specific-language”.

3 *Koncepcja nowego rozwiązania*

Nowy język powinien w prosty sposób pozwalać na tworzenie graficznych interfejsów użytkownika, a w szczególności wspomagać tworzenie formularzy do wyświetlania i edytowania danych przechowywanych przez system. Język powinien opisywać interfejs użytkownika w sposób deklaratywny, abstrahować od platformy i być od nich niezależnym. Dzięki temu powinien być z łatwością implementowany na nowych platformach.

3.1 Główne założenia

W trakcie analizy i projektowania języka wyspecjalizowano jego główne założenia. Język powinien je spełniać w jak największym stopniu.

- Stworzenie interfejsu graficznego przez opisanie jego elementów, używając meta model interfejsu – prostych i znanych wszystkim elementów.
- Definiowanie podstawowych elementów meta model interfejsu, z których można budować dany interfejs (przyciski, etykiety, listy rozwijane itp.).
- Składnia i działanie w myśl podejścia deklaratywnego.
- Składnia musi być zrozumiała i maksymalnie „samo-tłumacząca” oraz bardzo intuicyjna.
- Język ma jedynie wprowadzić kolejny poziom abstrakcji w budowie interfejsu. W myśl podejścia deklaratywnego – powinien ukryć szczegóły implementacji specyficzne dla danej technologii. Dzięki temu kod źródłowy odpowiedzialny za opis interfejsu użytkownika ma być łatwo adoptowany na innych platformach, które będzie wspierał język.
- Rozwiązanie ma być w łatwy sposób rozszerzalne o kolejne elementy języka, jeżeli okaże się, że zbiór dostępnych możliwości jest niewystarczający.
- Również dzięki architekturze, której budowa będzie oparta na podmianie komponentów, dodanie kolejnych platform, na których będzie można używać języka, ma być maksymalnie proste.

- Wykorzystywanie dostępnych frameworków przeznaczonych do tworzenia graficznych interfejsów użytkownika.

3.2 Opis prototypowego języka

Język jest deklaratywny i w sposób niezależny od platformy końcowej opisuje graficzne interfejsy użytkownika. Składnia jest wzorowana na rozwiązaniu GCL.

3.2.1 *Semantyka*

Z racji tego, że język ma pracować niezależnie od platformy, musi wprowadzić swój meta model. Byłby on ogólną reprezentacją interfejsu użytkownika. Dzięki temu byłoby to rozwiązanie generyczne.

Język posiada słowa kluczowe i określoną dokładnie składnię. Początkowo miał on pozwalać na tworzenie interfejsu, który byłby reprezentowany przez trzy rodzaje głównych kontenerów:

- Frame/Window – Niezależne okno, obsługiwane przez system operacyjny indywidualnie., zgodnie z paradygmatem SDI.
- InternalFrame – Okno uruchamiane w kontekście innego okna (rodzica), zgodnie z paradygmatem MDI.
- Dialog – Wewnętrzne okno aplikacji, obsługiwane wraz z głównym oknem aplikacji przez system operacyjny.
- Panel – Obszar wypełniany widgetami, używany zarówno w Frame i Dialog.

Podczas prac analitycznych nad przeznaczeniem języka, zgodnie z jednym z głównych postulatów – generyczności i niezależności od platformy, zrezygnowano z natywnej obsługi tworzenia niezależnych przez język okien i okien dialogowych. Język wspiera jedynie tworzenie Paneli.

Ma on być niezależny od platformy. W związku z tym, powinien umożliwiać tworzenie interfejsów na klasyczne platformy – uruchamiane bezpośrednio w systemie operacyjnym i zarządzane przez jego menadżera okien. Jak również powinien także wspierać platformy uruchamiane w przeglądarce internetowej – aplikacje webowe. Te drugie nie posiadają w prostej linii odpowiedników Frame i Dialog. Chociaż istnieją pewne elementy w

niektórych technologiach przeznaczonych do tworzenia webowych, graficznych interfejsów użytkownika, które symulują jedynie działanie klasycznego menadżera okien, znanego z systemu operacyjnego i są najczęściej jedynie ciekawostką. Pokazuje to możliwości wykorzystania HTML/CSS/JS, a nie realne wykorzystanie tego paradygmatu w aplikacjach webowych.

Meta model interfejsu służy do jego definiowania. Abstrahuje on od technologii i wyciąga z nich część wspólną, więc nie odnajdzie się takich elementów, jak okno *(ang. Frame)* czy okno dialogowe *(ang. Dialog)*. Meta model składa się z następujących części:

- Panel – główny element, na którym rozmieszcza się widgety. Może on przechowywać wiele paneli, które posiadają już inne widgety. Zagnieżdżenie struktury jest dowolnie duże. Ogranicza ją jedynie czytelność interfejsu.
- Button – reprezentuje przycisk na formularzu.
- ComboBox – reprezentuje rozwijana lista , która przechowuje spis elementów.
- Label – reprezentuje etykietę.
- TextBox – pole tekstowe, które przyjmuje wartości z klawiatury.
- TextArea – pole tekstowe pozwalające na wprowadzanie tekstu wieloliniowego.

Dzięki meta modelowi można przechowywać definicje danego interfejsu. Do tworzenia odpowiedniej struktury przy użyciu meta modelu służy już sam język. Definiuje on kilka słów kluczowych.

Słowa kluczowe:

- create – Główny człon każdego wyrażenia.
- panel – Element języka, który zwiększa naturalną czytelność wyrażenia.
- title – Każda stworzona formatka musi posiadać tytuł.
- containing – Pozwala na określenie elementów składowych formatki. Kolejność elementów w tym wyrażeniu wprost przekłada się na kolejność w wygenerowanym interfejsie.
- textbox – Odpowiada za definiowanie pól tekstowych.
- textarea – Odpowiada za tworzenie wieloliniowych pól tekstowych.

- button – Odpowiada za tworzenie przycisków.
- combobox – Odpowiada za tworzenie rozwijanych list wyboru.
- as – Pozwala na zdefiniowanie programistycznego identyfikatora dla widgetu, użytego w interfejsie. Dzięki temu możemy łatwo się do nich odwoływać.
- $btn – Pozwala na wyszukanie w sposób hierarchiczny elementów typu Button na wygenerowanej formatce.
- $lb – Pozwala na wyszukanie w sposób hierarchiczny elementów typu Label na wygenerowanej formatce.
- $cbb – Pozwala na wyszukanie w sposób hierarchiczny elementów typu Combobox na wygenerowanej formatce.
- $tb – Pozwala na wyszukanie w sposób hierarchiczny elementów typu TextBox na wygenerowanej formatce.
- $ta – Pozwala na wyszukanie w sposób hierarchiczny elementów typu TextArea na wygenerowanej formatce.

Operacje, których nazwy zaczynają się od znaku '$', służą do wyszukiwania użytych elementów graficznych w wygenerowanej formatce. W odróżnieniu od innych słów kluczowych, operacje wyszukiwania nie są używane w wyrażeniu, a jedynie w kontekście wyniku zwróconego przez wyrażenie.

3.2.2 *Przykłady*

W celu lepszego zobrazowania działania nowego języka poniżej zaprezentowano przykładowe wyrażenia wraz z wynikiem ich działania. Napisano je w pseudokodzie, w którym, ze względu na utrzymanie prostoty, kontekst rozpoznawany jest na podstawie wcięć – odmiennie od kontekstu identyfikowanego przez użycie takich elementów jak „{ }" czy „( )". Elementy składniowe, takie jak słowa kluczowe, zaznaczone są na czerwono. Na niebiesko zaznaczano użyte identyfikatory programistyczne i opisy widgetów, które zostaną wyświetlone na wygenerowanym formularzu. Do reszty elementów w wyrażeniu użyto koloru czarnego.

```
1  form = create panel title Dane Osobowe
2                  containing
```

```
3                       textbox Imię as tb_name
4                       textbox Nazwisko as tb_surname
5                       button Ok as btn_ok
```

Listing 12. Przykładowe użycie nowego języka. Prosty przykład.

Przykład z listingu 12. pokazuje podstawowe wykorzystanie języka. Pseudokod powinien wygenerować formularz o nazwie „Dane Osobowe”, który będzie zawierał dwa widgety typu Textbox i jeden typu Button. Hierarchia formularza przedstawiona jest rysunku 12. Jest to bardzo prosty przykład.

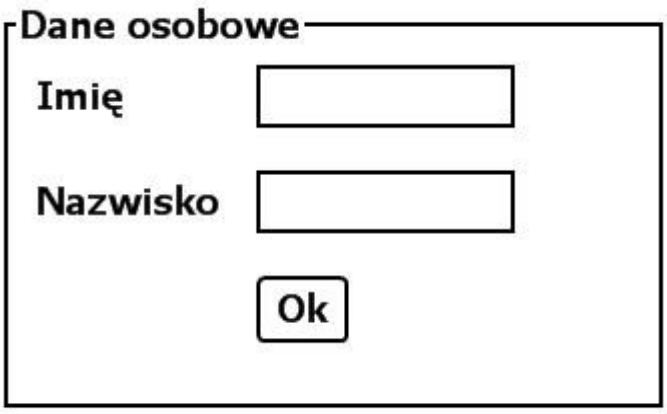
Rysunek 12: Hierarchia prostego formularza z listingu 12

Hierarchię definiuje się przez kolejność elementów po słowie kluczowym *containing*. Zagnieżdżenie uzyskuje się przez dodanie panelu do panelu rodzica. Hierarchia formularza jest bardzo ważna. Późniejsze odwołania do widgetów wykorzystanych w formularzu odbywają się przez użycie programistycznych identyfikatorów w kontekście hierarchii. W omawianym przypadku hierarchia jest jednopoziomowa, więc wystarczy podać jedynie identyfikator.

```
1   ( form $btn btn_ok ) addClickHandeler handler
```

Listing 13. Odwołanie do konkretnego elementu formularza.

Listing 13. pokazuje odwołanie do przycisku „Ok”, przy użyciu operacji $btn służącej do wyszukiwania przycisków w zwróconym formularzu. Do tak otrzymanego przycisku zostaje przypisany obiekt obsługujący zdarzenie kliknięcia na przycisk.



Rysunek 13: Wynik działania pseudokodu z listingu 13

Rysunek 13. przedstawia wygenerowany formularz. W myśl zasady *„konwencja nad konfiguracją”*, widgety rozmieszcza się jeden pod drugim. Opisy widgetów, które nie mają tak jak Button miejsca na wyświetlenie, są w postaci etykiet *(ang. Labels)* po lewej stronie, wyrównane do prawej krawędzi.

Kolejny przykład przedstawiony na listingu 14. wykorzystuje już zagnieżdżoną strukturę. Jednak ciągle używa podstawowych widgetów (przyciski i pola tekstowe).

```
1  form =
2   create panel title Dane Osobowe
3       containing
4         textbox Imię as tb_name
5         textbox Nazwisko as tb_surname
6         panel Adres as p_address containing
7            textbox Miasto  as tb_city
8            textbox Ulica as tb_street
9         button Ok as btn_ok
```

Listing 14. Wykorzystanie zagnieżdżonego panelu.

Hierarchia w tym przypadku jest już zagnieżdżona i nie można w prosty sposób odwołać się do pola tekstowego o identyfikatorze tb_street. W formie graficznej hierarchię widać na rysunku 14.

Rysunek 14: Hierarchia formularza z listingu 14

```
1  ( form $btn p_address.tb_city ) getValue
```

Listing 15. Odwołanie do konkretnego elementu formularza przy użyciu operatora hierarchicznego kropki.

Pseudokod przedstawiony na listingu 15. pokazuje, w jaki sposób pobrać wartość z pola tekstowego przeznaczonego do przechowywania nazwy miasta. Pole to zostało zawarte w ramach wewnętrznego panelu o identyfikatorze p_address. Główny panel o nazwie „Dane Osobowe" jest pomijany w zapytaniu, służącym do „wyciągnięcia" pola tekstowego tb_city. Kontekst w operacji wyszukiwania jest domyślnie ustawiony na główny panel.

Rysunek 15: Wynik działania pseudokodu z listingu 14

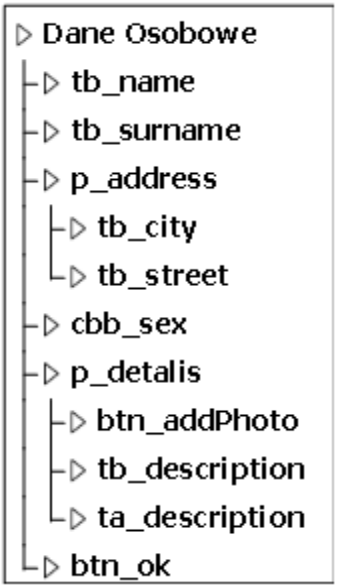
Wygenerowany formularz z rysunku 15. przedstawia bardziej skomplikowany przypadek. Wyrównanie elementów odbywa się oddzielnie dla każdego z paneli.

```
1  form = create panel title Dane Osobowe
2          containing
3            textbox Imię as tb_name
4            textbox Nazwisko as tb_surname
5            panel Adres as p_address containing
6              textbox Miasto as tb_city
7              textbox Ulica as tb_street
8            combobox Płeć as cbb_sex values { „Mężczyzna", „Kobieta" }
9            panel Szczegóły as p_details containing
10             button Dodaj zdjęcie as btn_addPhoto
11             textbox Krótki opis as tb_description
```

```
8               textarea Opis as ta_description
9             button Ok as btn_ok
```

Listing 16. Użycie list rozwijanych i wieloliniowych pól tekstowych.

Zgodnie z pseudokodem na listingu 16. obserwuje się wykorzystanie wszystkich dostępnych słów kluczowych języka. Tworząc element typu Combobox po wyrażeniu nadania identyfikatora programistycznego, stosuje się słówko values. Po nim podaje się listę wartości.

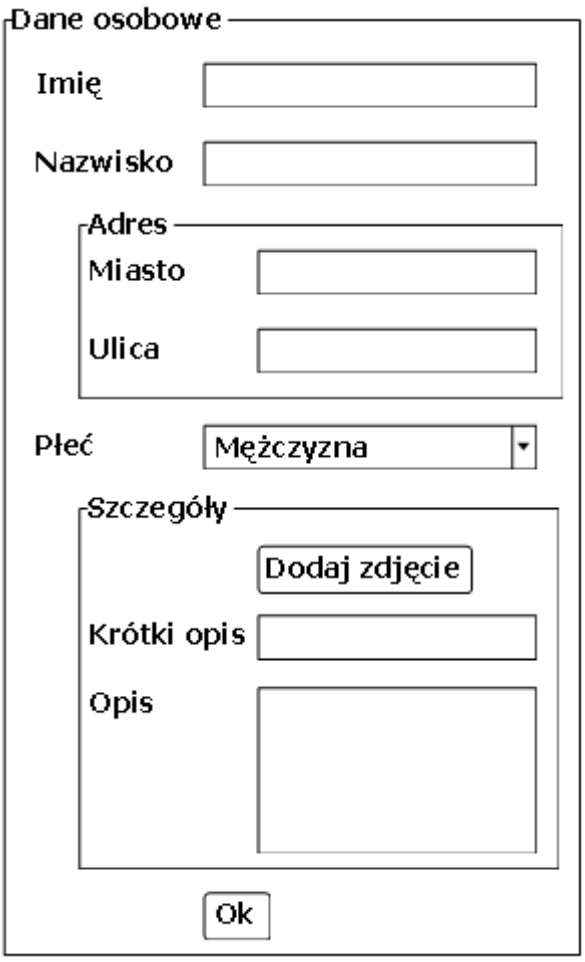


Rysunek 16: Hierarchia formularza z listingu 16

Hierarchia formularza z listingu 16. jest analogiczna do wcześniejszego przykładu. W formie graficznej przedstawiono ją na rysunku 16. Zaś gotowy formularz wygenerowany przez język obserwuje się na rysunku 17.

W tym miejscu trzeba zwrócić uwagę, że przy niewielkiej ilości kodu programista jest w stanie stworzyć większość typowych formularzy. W odróżnieniu od obecnych technologie wymagają, aby pisać powielający się kod, który jest trudny w utrzymaniu i pielęgnacji.

Nowy język ma opisywać jedynie interfejs użytkownika, więc musi być jak najbardziej ekspresyjny i czytelny. Wszystkie elementy, które mogą być pominięte są wyłączone ze składni języka.



Rysunek 17: Wygenerowany formularz z listingu 16

3.2.3 *Przyjęte rozwiązania*

Po pierwsze – nie należy definiować identyfikatora programistycznego dla głównego panelu. Takie założenie zwiększa czytelność kodu. Gdyby wynikiem działania wyrażenia miał być główny panel, identyfikator byłby całkowicie nieużyteczny. W związku z tym jest on niepotrzebny.

Nie ma możliwości definiowania zachowania elementów interfejsu w wyrażeniu budującym jego strukturę. Jest to celowe założenie chroniące przed zaśmiecaniem takiego wyrażenia zbędnymi na tym etapie informacjami. Dzięki prostym w użyciu metodom wyszukującym, definiowanie zachowania formularza odbywa się w równie prosty sposób.

4 *Opis narzędzi zastosowanych w pracy*

Działający prototyp stworzono na platformie Java. Aktualnie istnieje implementacja dla bibliotek Swing i GWT. W czasie wytwarzania użyto opisanych w tym rozdziale technologii.

4.1 Java 6.0

Jest to kolejna wersja obiektowego języka programowania. Język jest kompilowany do kodu bajtowego *(ang. byte code)* i interpretowany przez wirtualną maszynę. Dzięki takiemu podejściu możliwe jest przenoszenie aplikacji między platformami systemowymi. W myśl zasady *„napisz raz, uruchom gdziekolwiek”* (slogan promujący język Java, wykreowany przez firmę Sun) wymaga się więc odpowiedniej implementacji maszyny wirtualnej na danej platformie systemowej.

W wersji 6.0 nie dodano żadnych nowych elementów języka do tych, które wystąpiły w wersji 5.0. Wprowadzono między innymi typy generyczne, adnotacje, słówko kluczowe enum. Prace skupiono głównie nad maszyną wirtualną oraz bezpieczeństwem.

Aktualnie dostępne są trzy główne platformy Java.

- Java ME – do obsługi urządzeń przenośnych.
- Java SE – do obsługi stacji roboczych
- Java EE – do obsługi rozproszonych systemów klasy enterprise

4.2 Java Swing

Swing[10] jest to biblioteka służąca do budowy graficznych interfejsów użytkownika na platformie Java. Dodano ją do Java SE w wersji 1.2 tej platformy. Swing został stworzony w celu umożliwienia budowy bardziej zaawansowanych i przyjaznych dla oka interfejsów użytkownika, które wcześniej budowano przy użyciu biblioteki AWT. W odróżnieniu od AWT, które delegowało rysowanie elementów do natywnych narzędzi systemowych, Swing nie wymaga użycia natywnych zasobów systemowych.

Dzięki temu, że Swing nie używa natywnych elementów platformy systemowej, jest w pełni od niej niezależny. Zarówno w formie wyrazu – używa niezależnego języka Java, jak też przy rozważaniu jego implementacji. Komponenty Swing same odpowiadają za swój wygląd. Swing do „rysowania" widgetów używa standardowego Java 2D API, a nie natywnych narzędzi systemowych jak ma to miejsce, m.in., w SWT czy AWT. W połączeniu z mechanizmem „Look and Feel" można uzyskać niemal identyczny wygląd omawianej aplikacji w systemach Linux, MacOS czy Windows. Mimo, że Swing jest niezależny od platformy, to jednak wykorzystuje w sposób przejrzysty dla programisty bibliotekę AWT w celu obsługi interakcji z użytkownikiem.

Swing jest bardzo rozszerzalną i w pełni konfigurowalną biblioteką. Dzięki odpowiedniej architekturze (m.in. wykorzystanie wzorca projektowego MVC), daje programiście możliwość użycia własnej implementacji wybranych elementów składniowych komponentu.

4.3 Google Web Toolkit 2.0

Google Web Toolkit[3] to stworzony przez firmę Google zestaw narzędzi, przeznaczony do wytwarzania aplikacji webowych w technologii AJAX *(ang. Asynchronous JavaScript and XML).* W skład tego zestawu wchodzą:

- Kompilator Java-to-JavaScript
- Development mode
- Biblioteki

Największą zaletą i tym, co wyróżnia GWT od innych technologii przeznaczonych do tworzenia aplikacji webowych jest to, że pisane są wyłącznie w języku Java. Programista, który nie zna JavaScript/HTML/CSS, może w bardzo łatwy sposób stworzyć działającą aplikację webową. Wykorzystuje ona najnowsze standardy internetowe.

Dzięki takiemu podejściu, wykorzystuje się gotowe narzędzia przeznaczone dla platformy Java. Rozpoczęcie pracy z GWT przez osoby znające i mające doświadczenie w programowaniu jest bardzo przyjemne i proste.

Równie ważnym czynnikiem przemawiającym za GWT jest to, że pisząc jedną wersję aplikacji, otrzymuje się od razu wsparcie dla praktycznie wszystkich przeglądarek internetowych. Nie ma potrzeby pisania specjalnej wersji danego kodu JavaScript czy CSS,

który będzie działał na określonej przeglądarce. Odpowiada za to GWT w momencie kompilacji, a następnie w momencie uruchomienia danej aplikacji.

Praca z GWT

Dzięki wykorzystaniu GWT, programista może pisać część serwerową i część przeznaczoną dla klienta w tym samym języku. Jest to duży plus tego rozwiązania. Obie części projektu mogą być więc z łatwością poddane refaktoryzacji. W przypadku pisania aplikacji JavaScript jest to bardzo trudne do osiągnięcia, z powodu dynamicznego oblicza języka JavaScript.

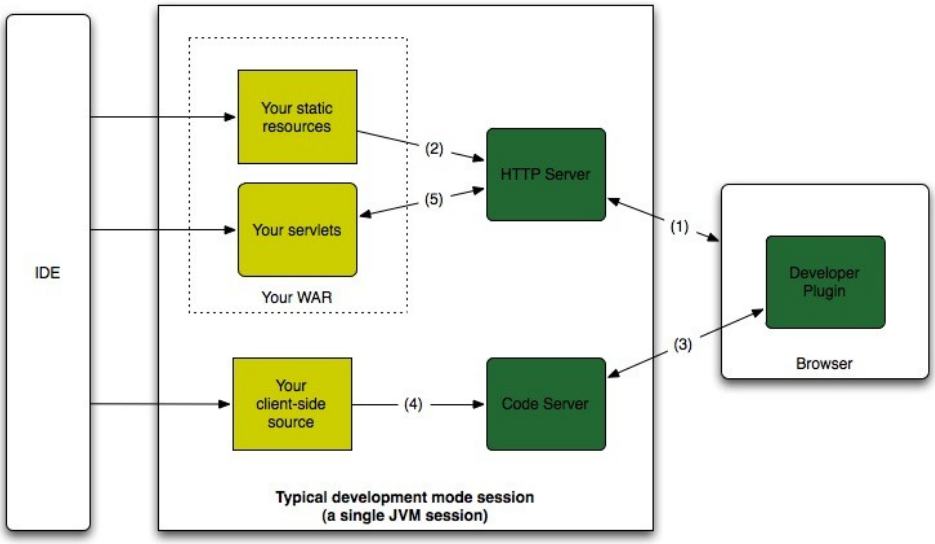
Styl opisywania interfejsu użytkownika przypomina pisanie kodu aplikacji desktopowych przy użyciu Swing API. Co jak już wcześniej udowodniono w tej pracy nie jest najlepszym sposobem. Powstały też inne rozwiązania, jak UiBinder (opisany w rozdziale 2.4.3), który pozwala na opisywanie interfejsu w sposób deklaratywny. Znacząco zwiększa przy tym produktywność programisty.

Podstawowy sposób pisania aplikacji z wykorzystaniem GWT ma w założeniu dwie bardzo ważne rzeczy:

- Bezstanowy serwer
- Stanowy klient

Serwer nie przechowuje stanu aplikacji, dlatego w bardzo łatwy sposób można rozbudować infrastrukturę serwerową bez obawy o to który serwer przyjmie żądanie od klienta. Dzięki temu równoważenie obciążenia (*ang. Load balancing*) jest dużo prostszy w wykonaniu. Ponieważ niezależnie od tego, który serwer dostanie żądanie od klienta, jest w stanie je obsłużyć. Przeniesienie logiki aplikacji na stację klienta zmniejsza wymagania serwerowe. Dużą część obliczeń wykonuje się po stronie klienta.

Możliwe jest także debugowanie znane ze standardowych aplikacji Java SE.



Rysunek 18: Schemat debagowania aplikacji GWT.[3]

Rysunek 18. przedstawia schemat działania debugera GWT. Aplikację uruchamia się w przeglądarce. Dzięki odpowiedniej wtyczce do przeglądarki, która komunikuje się z danym serwerem, możliwe jest tłumaczenie akcji na kod źródłowy Javy. Warto w tym miejscu zaznaczyć, że, zmieniając kod klienta naszej aplikacji, nie wymaga się restartowania sesji „development mode". Wystarczy odświeżyć stronę w naszej przeglądarce, a kod zostanie przeładowany automatycznie. Dzięki wtyczkom do praktycznie wszystkich znanych przeglądarek możliwe jest debugowanie w realnych warunkach działania aplikacji oraz zastosowanie takich narzędzi, jak wtyczka FireBug do przeglądarki FireFox. FireBug pozwala na śledzenie komunikacji typu AJAX, podglądanie generowanej struktury DOM danej strony, edycja stylów CSS i podgląd na żywo wyników, a także wiele innych pomocnych dla każdego programisty stron www funkcji.

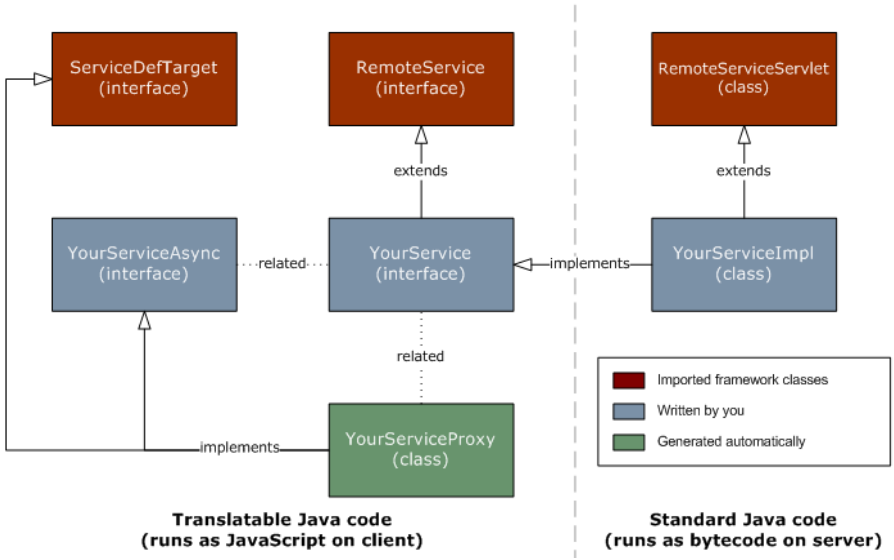
Komunikacja Klient – Serwer

GWT dostarcza dwa rozwiązania do komunikacji Klient – Serwer, przy użyciu protokołu HTTP.

Pierwszą możliwością jest użycie klas z pakietu com.google.gwt.http.client.*. Dzięki nim osiąga się pełną kontrolę nad danym żądaniem. Zastosowanie żądań HTTP nie różni się znacząco od używanych w innych językach czy bibliotekach. Jednak występują pewne ważne różnice. Wszystkie wołania są asynchroniczne, więc programista musi używać obiektów typu

callback do obsługi wołania HTTP. Dzieje się tak, ponieważ kod kliencki uruchamiany jest w przeglądarce w ramach jednego wątku. Długie wołanie synchroniczne może spowodować „zawieszenie" się tej aplikacji. Z tego właśnie powodu twórcy GWT udostępnili jedynie komunikację asynchroniczną. Drugie ograniczenie wołań HTTP jest związane z bezpieczeństwem w samych przeglądarkach internetowych. Mianowicie, nie ma możliwości wykonania wołania HTTP do innego hosta, niż tego, który wysłał daną stronę.

Innym sposobem na komunikację z serwerem jest zastosowanie mechanizmu GWT RPC. Zapewnia on przezroczysty model komunikacji asynchronicznej z serwerem. Mechanizm ten zajmuje się w całości niskopoziomową obsługą wołania. Jest on zalecany przez twórców GWT, głównie ze względu na jego prostotę i szybkość implementacji. W tym przypadku zapewniona jest serializacja i deserializacja obiektów oraz ich transport przez HTTP. Komunikacja odbywa się między specjalnym servletem Java, odpowiedzialnym za obsługę zadanego żądania, a wygenerowanym automatycznie przez GWT proxy po stronie klienta. Rysunek 19. z oficjalnego poradnika GWT przedstawia ten mechanizm szczegółowo.



Rysunek 19: Mechanizm GWT RPC[3].

Kompilator

Jest to najbardziej zaawansowany element GWT. Nie jest on zwykłym translatorem Java do JavaScript. Kompilator działa na źródłach Java, więc do kompilacji nie wystarczą skompilowane klasy Java.

Kompilator wspiera większą część języka Java. GWT emuluje większość klas JRE. Jeżeli zastosuje się niewspieraną klasę, występuje błąd kompilacji. Klasy z pakietu java.io.* nie są, dla przykładu, emulowane przez GWT z wiadomych ograniczeń języka JavaScript.

Kompilator przygotowuje zestaw aplikacji pod każdą przeglądarkę osobno i w każdej wersji językowej (GWT wspiera internacjonalizację). Dla przykładu, jeżeli występują cztery przeglądarki i dwie wersje językowe aplikacji, kompilator przygotuje aż osiem wersji aplikacji. Odpowiednia wersja aplikacji ładowana jest przez główny skrypt ładujący, decydujący o tym, którą część ściągnąć z serwera. Jest to o tyle ważne rozwiązanie, że klient nie musi ściągać wersji kod JavaScript do przeglądarki, np., Internet Explorer 6, jeżeli używa Firefox. Taki mechanizm dynamicznego ładowania jest praktycznie nie wykonalny przy użyciu standardowych frameworków JavaScript i jest kolejną bardzo ważną i unikalną własnością GWT.

Twórcy chwalą się, że nie można napisać szybciej działającego kodu JavaScript, pisząc go w sposób klasyczny – ręcznie. Dzieje się tak, ponieważ kompilator produkuje bardzo zoptymalizowany kod JavaScript. Stosuje też zaawansowane sposoby optymalizacji wynikowego kodu JavaScript, zmniejszając jego rozmiar, jak również samą szybkość wykonania. Jednak nie jest to prosta optymalizacja polegająca na skracaniu nazw zmiennych i usuwanie białych znaków z kodu źródłowego. Kod źródłowy Javy jest sprawdzany pod względem dostępności (czy jest możliwe jego wykonanie). Te części, które nigdy nie będą użyte, nie są brane pod uwagę w czasie kompilacji. Tyczy się to, m.in., całej biblioteki widgetów dostarczanych przez GWT. Jeżeli nie stosuje się w kodzie, np., elementów typu Combobox, to nie zostaną one w ogóle dodane do kodu wynikowego. Dobrym przykładem zaawansowania kompilatora GWT jest biblioteka GwtQuery, wersja bardzo popularnej JavaScriptowej biblioteki jQuery. Napisano ją w GWT. Wersja JavaScript to zawsze (wersja 1.4.2) 24KB, niezależnie od tego, jakich funkcji się używa. Natomiast używając, np., tylko jednego selektora CSS z GwtQuery, uzyskuje się nawet mniej niż 1KB. W kodzie wynikowym nie występuje cała biblioteka, a jedynie fragment niezbędny do poprawnego jej działania. Pisząc kod JavaScript ręcznie, nie ma możliwości uzyskania takiej funkcjonalności.

4.4 Eclipse IDE

Jest jednym z najpopularniejszych zintegrowanych środowisk programistycznych *(ang. Integrated development environment)* do wytwarzania aplikacji. Eclipse IDE zbudowano przy użyciu platformy Eclipse. Jest to bardzo rozbudowana platforma służąca do wytwarzania aplikacji typu desktop. Istnieje szereg wersji Eclipse IDE, które są dostarczane standardowo z każdą kolejną wersją platformy. Główne jej założenia, to budowa oprogramowania na zasadzie wtyczek do głównego produktu. IDE, dostarczane standardowo, jest gotowym pakietem wtyczek.

W wersji 3.6 o kodowej nazwie Helios Fundacja Eclipse dostarcza następujące produkty[11]:

- Eclipse IDE for Java EE Developers
- Eclipse IDE for Java Developers
- Eclipse Classic 3.6.0
- Eclipse IDE for C/C++ Developers
- Eclipse for PHP Developers
- Eclipse IDE for JavaScript Web Developers
- Eclipse for RCP and RAP Developers
- Pulsar for Mobile Developers
- Eclipse Modeling Tools (includes Incubating components)
- Eclipse IDE for Java and Report Developers
- Eclipse SOA Platform for Java and SOA Developers (includes Incubating components)

Za rozwój platformy odpowiada Fundacje Eclipse. Jest to organizacja typu non-profit. Platforma Eclipse zbudowana jest praktycznie w całości przy użyciu języka Java, występuje więc na wszystkich platformach systemowych wspierających język Java, a w szczególności jego wirtualną maszynę. Wymagana jest też obecność, stworzonej również przez Fundację Eclipse, biblioteki graficznej SWT. Jest ona odpowiedzią na istniejące w ekosystemie Javy biblioteki AWT i Swing. SWT napisano przy użyciu języka Java, ale SWT również używa natywnych dla danego systemu elementów. Dlatego niezbędna jest oddzielna implementacja dla danej platformy systemowej.

Sam w sobie edytor nie wspiera żadnej technologii. Jednak dzięki bardzo rozbudowanemu systemowi wtyczek *(ang. pliugins)*, możliwa jest obsługa takich języków i technologii jak:

- Ruby On Rails
- C/C++
- PHP
- JavaScript
- Java

Dzięki tak dużej elastyczności Eclipse IDE doskonale sprawdza się w zadaniach od analizy, przez proces wytwarzania, aż do końcowego wdrożenia systemu. Świadczyć może o tym ogromne zainteresowanie świata biznesu. Firmy, takie jak IBM czy Adobe, tworzą swoje flagowe produkty w oparciu o platformę Eclipse. Stworzenie takich programów jak IBM Rational Software Architect czy Adobe Flex Builder bez platformy Eclipse byłyby zdecydowanie dużo bardziej czasochłonne. W związku z tym, wymagałyby większych nakładów finansowych.

Firmy często decydują się na stworzenie jedynie wtyczki *(ang. plugin)*. Wtyczka ma pomagać podczas pracy z ich produktem, zaś największym jej atutem jest to, że świetnie się integruje z aktualnie wykorzystywanymi technologiami. Dobrym przykładem jest firma Google, która do swoich produktów tworzy jedynie wtyczki. Daje to dużą swobodę programiście. Może on korzystać z dostosowanej do jego wymagań wersji Eclipse IDE.

Platformę Eclipse wykorzystuje się również do tworzenia produktów, które nie są skierowane, ani do programistów, ani do całej branży IT. Programy, takie jak Lotus Notes 8 czy Lotus Sametime 8, w ogóle nie przypominają środowiska programistycznego. Jednak z sukcesem przeniesiono je na platformę Eclipse.

Wersja – IDE for Java EE Developers – dostarcza gotowego i pełnego zestawu narzędzi, niezbędnych do wytwarzania oprogramowania na platformie Java EE. Właśnie tę wersję wykorzystano w procesie tworzenia prototypu języka.

5 *Opis stworzonego prototypu*

Implementacja referencyjna przedstawionej w rozdziale 3 . koncepcji nowego języka została stworzona w języku Java.

Z powodu braku dobrych pomysłów na nazwę nowego języka, pierwsza implementacja otrzymała kodową nazwę MDGL. Skrót rozwija się bardzo trywialnie - „Marek Drob GUI Language".

Obecnie prototyp wspiera dwie platformy do tworzenia graficznych interfejsów użytkownika:

- Swing
- Google Web Toolkit

Od początku język miał być niezależny od platformy. W związku z tym, pierwszy prototyp miał być zaimplementowany właśnie na tych dwóch platformach. Filozofia tworzenia interfejsów użytkownika w Swing i GWT jest bardzo zbliżona. Można powiedzieć wprost, że ten drugi był wzorowany na Swing. Jednak GWT pozwala na pisanie aplikacji działających w przeglądarce, uruchamianych jako JavaScript/HTML/CSS. Natomiast Swing skupia się na tworzeniu interfejsów dla klasycznych stacji roboczych. Aplikacje napisane w Swing uruchamiana się w kontekście systemowego menadżera okien. Pokazanie implementacji, która ujednolici tworzenie interfejsów użytkownika na tych dwóch, jednak bardzo różnych, platformach, było jednym z głównych celów tej pracy.

5.1 Opis aktualnych możliwości

Prototyp aktualnie wspiera w całości zaproponowaną w rozdziale 3 . składnię języka. Oczywiście, żeby zaimplementować jak najbardziej zbliżoną wersję zaproponowaną pseudokodem, niezbędna byłaby zmiana kompilatora Java i GWT. Jednak takie podejście niesie za sobą ogromny nakład pracy.

Po pierwsze, zaimplementowanie własnych operatorów w Java nie jest rzeczą trywialną. Dodatkowo niezbędna będzie ingerencja w kompilator każdej platformy, w ramach której wprowadzi się obsługę zaproponowanego języka. Ponadto wymusza na programiście

użycie nieoficjalnego kompilatora. Jest to bardzo niewygodne rozwiązanie, które może prowadzić do wielu komplikacji i trudności w użytkowaniu języka.

Kolejnym czynnikiem, który przemawia przeciwko modyfikacji kompilatora, jest fakt obsługi takiego języka przez środowiska programistyczne. Podpowiadanie i kolorowanie składni nie byłoby możliwe bez dodatkowej pracy w postaci tworzenia wtyczek do popularnych IDE. Wsparcie w tego typu produktach, często jest jednym z najważniejszych czynników decydujących o sukcesie danego rozwiązania.

Innym rozwiązaniem było napisanie własnego parsera do zaproponowanego języka. W tym przypadku nie byłoby problemu kompatybilności z kompilatorem danego języka/technologii. Jednak takie rozwiązanie ma również swoje wady. Kod odpowiedzialny za wygląd musiałby być przechowywany w oddzielnych plikach lub w specjalny sposób zaznaczony w kodzie (w komentarzach). Zmniejszałoby to swobodę programisty. Wsparcie w IDE byłoby również niemożliwe bez dodatkowej pracy.

Następną możliwością było stworzenie języka typu DSL w czystej Javie. Jest to najbardziej uniwersalna i najłatwiejsza w stworzeniu implementacja języka MDGL. Jednocześnie nie wymuszająca na twórcy dużych ustępstw w składni.

5.2 Tworzenie Domain-specific language (DSL) w Java

Istnieje wiele podejść, dzięki którym można stworzyć DSL w Java. Każdy ze sposobów ma swoje zalety i wady. Na przestrzeni lat stworzono wiele technik od używania łańcuchów znaków przetwarzanych przez specjalny parser do projektowania API, które mogłoby być używane zgodnie z naszą gramatyką.

5.2.1 *Zewnętrzny DSL – implementacja oparta na łańcuchach znaków (String)*

Technika ta nie ma żadnych ograniczeń, jeżeli chodzi o gramatykę nowego języka. Dzieje się tak dlatego, że tak naprawdę niezbędne jest napisanie leksera i parsera. Działają one, praktycznie rzecz biorąc, jak kompilator. Jako, że nie ma żadnych ograniczeń w gramatyce języka może być on przedstawiony w dowolnej formie. Brak ograniczenia w formie wyrazu, jak to ma miejsce w wewnętrznych DSL (5.2.2), jest główną zaletą tej techniki.

Tworząc ten typ języka na platformie Java, najczęściej wykorzystuje się dwa produkty. Jeden do tworzenia leksera, który dzięki odpowiednim wyrażeniom regularnym

„potnie” łańcuch na tak zwane tokeny. Drugi będzie definiował gramatykę - czyli będzie łączył tokeny w większe wyrażenia. Pierwszym produktem jest JLex[12], generator lekserów dla języka Java napisany w języku Java. Drugim zaś CUP[13] - generator parserów, który doskonale współpracuje z JLex. Oba produkty wykorzystywane są m.in. w projekcie ODRA[14].

Przykład plików .lex[15] i .cup[16] systemu ODRA, które pokazują wykorzystanie leksera i parsera w dużym działającym systemie, można znaleźć w repozytorium systemu.

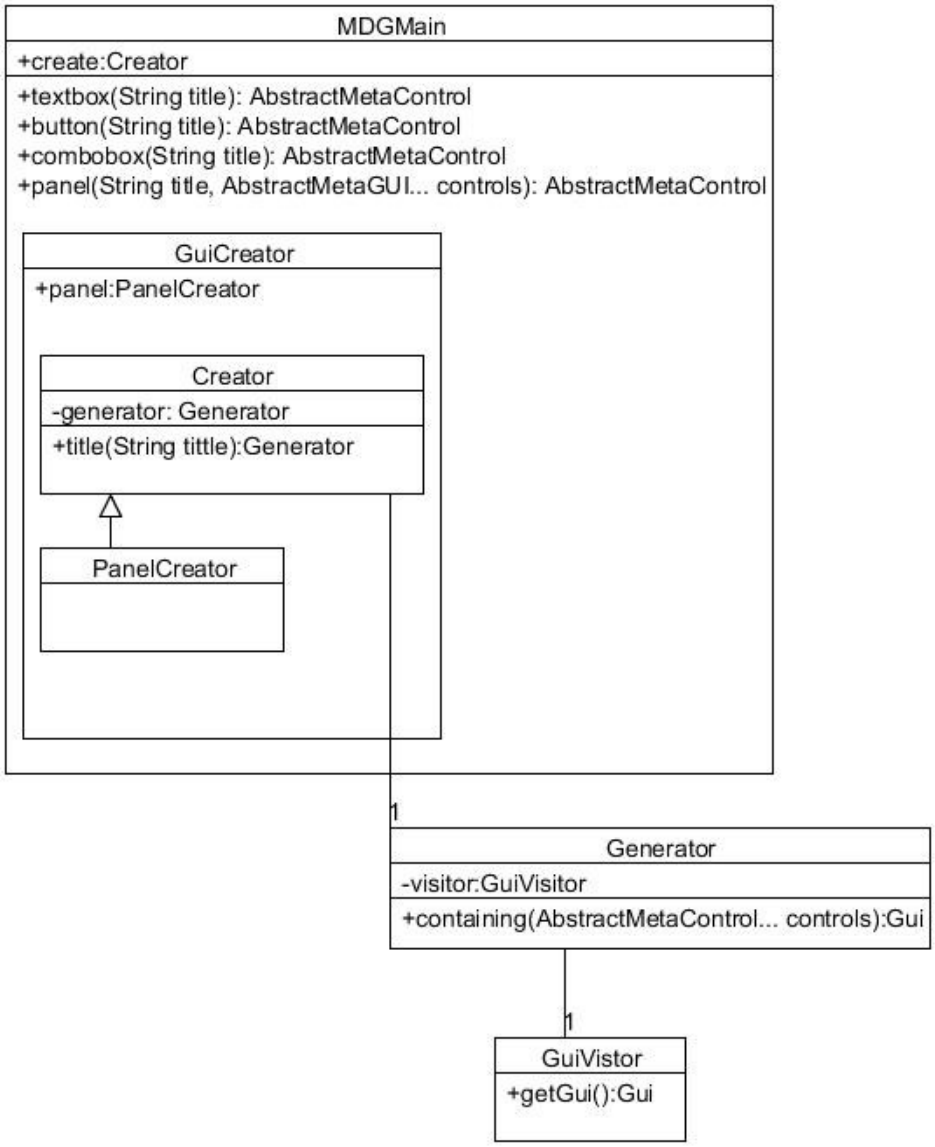
Największą wadą takiego rozwiązania jest brak wsparcia w podpowiadaniu składni nowego języka w popularnych IDE. Oczywiście jest to możliwe w wykonaniu jednak wymaga dużych nakładów pracy podczas pisania rozszerzeń dla IDE. Również to, że błędy składniowe zostaną odkryte dopiero w czasie wykonania programu mogą przysporzyć wielu problemów.

5.2.2 *Wewnętrzny DSL – na przykładzie prototypu MDGL*

Rozpatrując powyższe problemy, przyjęto ideę napisania języka przy użyciu środków udostępnianych przez język Java. Taki sposób określa się wewnętrznym DSL *(ang. Internal DSL)*. Sam pomysł zaczerpnięto z języka GCL, opisanego w rozdziale 2.4.4 .

Jednym ze sposobów tworzenia wewnętrznych DSL w języku Java jest podejście sprowadzające się do odpowiedniego zaprojektowania API. W odpowiedni sposób stworzone API, daje możliwość „naturalnego” czytania takiego kodu. [17].

Idea jest bardzo prosta i często ciężko stwierdzić, czym tak naprawdę różni się DSL od zwykłego API. Wszystko rozgrywa się na poziomie samej gramatyki języka i jego czytelności. Sama architektura API może być doskonała, jednak nieprzemyślane nazwy klas, zmiennych czy metod mogą sprawić, że użycie takiego API nie będzie przypominało czytelnego i przejrzystego DSL.



Rysunek 20: Poglądowy diagram klas implementacji języka MDGL

Na rysunku 20. widzimy uproszczony diagram klas API zaproponowanego w rozwiązaniu MDGL. Pierwsze spojrzenie na poglądowy diagram nie ukazuje, w jaki sposób używać dane API. Klasa `MDGMain` przechowuje definicję metod pomocniczych, pozwalających na tworzenie odpowiednich elementów interfejsu.

```
1  public static AbstractMetaControl textbox(GUIEnum guiEnum) {
2      return new MetaTextBox(guiEnum.getName());
3  }
4  public static AbstractMetaControl textbox(String name) {
```

```
5       return new MetaTextBox(name);
6   }
7   public static AbstractMetaControl button(String name) {
8       return new MetaButton(name);
9   }
10  public static AbstractMetaControl combobox(String name) {
11       return new MetaComboBox(name);
12  }
13  public static AbstractMetaControl combobox(String name,List<String[]>
14  values) {
15              return new MetaComboBox(name,values);
16  }
17  public static AbstractMetaGUI panel(String title, AbstractMetaGUI... g)
18  {
19              return createPanel(title, g);
20  }
```

Listing 17. Pomocnicze metody z klasy MDGMain

Jak widzimy na listingu 17. metody mają bardzo prostą implementację. Dzięki nim można tworzyć meta strukturę interfejsu użytkownika. Abstrahuje ona od technologii i platformy. W związku z tym, unifikuje nazewnictwo i pozwala na interpretację takiego meta interfejsu na konkretnej platformie.

Kolejnym ważnym elementem, za który odpowiada `MDGMain`, to przechowywanie odpowiednio nazwanego obiektu typu wewnętrznej klasy `GuiCreator`.

```
1   public class MDGMain {
2   ...
3   public static GuiCreator create = new GuiCreator();
4   ...
5   public static class GuiCreator {
6     ...
7     public class Creator {
8       ...
9       public GuiGenerator title(String string) { ... }
10      ...
11    }
```

```
12    public class PanelCreator extends Creator { ... }
13    ...
14    public PanelCreator panel = new PanelCreator();
15  }
16  }
17  ...
18  create.panel.title("Szczegóły")
```

Listing 18. Referencja „create” w klasie MDGMain.

Jak widać na listingu 18. nazwa zmiennej jest równocześnie słowem kluczowym zaproponowanego języka. Natomiast w klasie `GuiCreator` występuje obiekt typu `PanelCreator`, nazwany panel. Ta łatwa struktura klas pozwala zobaczyć, jak powinno wyglądać początkowe wyrażenie języka. Jest ono zaprezentowane w 14. linijce w listingu 18. Jedyna różnicą między tym wyrażeniem, a proponowanym w pseudokodzie jest to, że spację zastąpiono znakiem kropki. Również identyfikatory i nazwy, które były reprezentowane jako zwykłe słowa w pseudokodzie, zostały zaimplementowane przy użyciu obiektów klasy `String`. Nie wpływa to jednak na czytelność takiego wyrażenia.

Metoda `title(Stirng title)` z klasy `Creator` daje możliwość podłączenia do tego wyrażenia obiektu klasy Generator. Klasa ta dzięki zastosowaniu podmiany komponentów, wprowadza odpowiednią implementację obiektu typu `GuiVisitor` – reprezentuje ona wykorzystanie wzorca projektowego Vistor, odpowiedzialnego za wygenerowanie odpowiedniego dla danej platformy interfejsu użytkownika.

```
1   public abstract class AbstractGuiGenerator {
2       private AbstractMetaContainer guiContainer;
3       GuiVisitor visitor;
4   ...
5       public void populateGui(AbstractMetaGUI... c) {
6         guiContainer.addControls(c);
7         guiContainer.accept(visitor);
8     }
9   }
10  public class GuiGenerator extends AbstractGuiGenerator {
11  ...
```

```
13   public MDGGWTAbstractPanel containing(AbstractMetaGUI... c) {
14         populateGui(c);
15         return ((MDGVisitor) visitor).getGui();
16      }
17   ...
18   }
```

Listing 19. Przykładowa implementacja klasy Generator

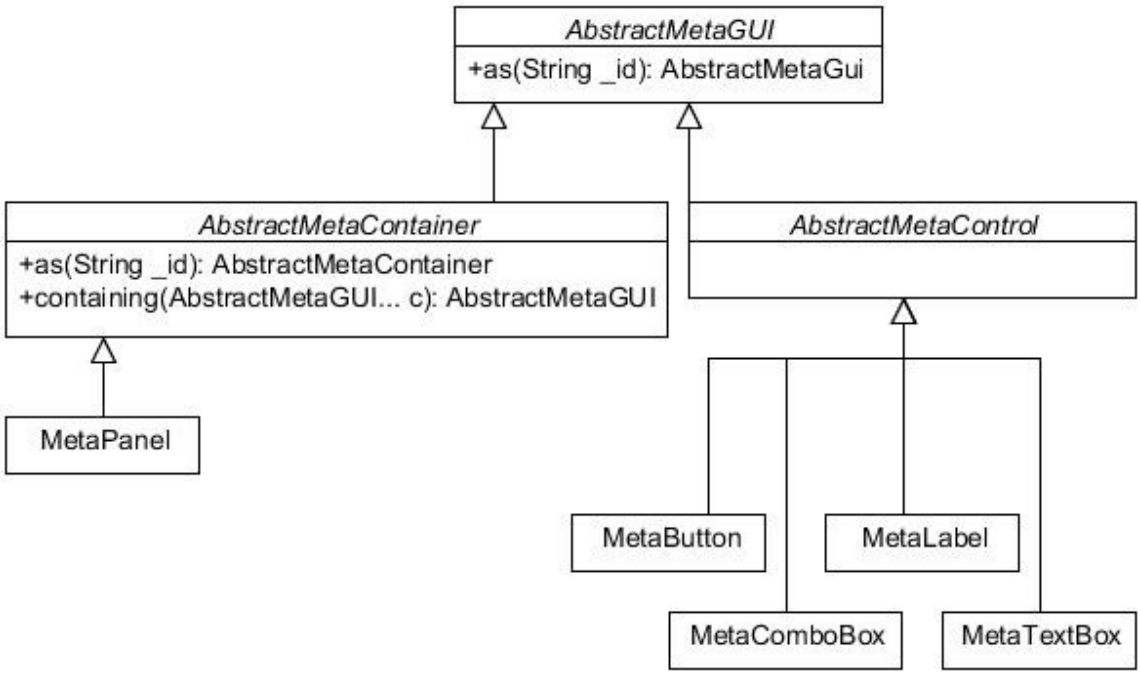
Na listingu 19. przedstawiono użycie `GuiGenerator` i `GuiVisitora` w celu wygenerowania odpowiedniego interfejsu użytkownika. Metoda `containing` w klasie `GuiGenerator` zadeklarowana jest jako metoda typu varargs – pozwala na przyjmowanie dowolnie dużej ilości argumentów tego samego typu. Ta własność została wykorzystana w celu użycia sekwencji funkcji, generujących podstawowe meta kontrolki.

Przedstawione w tym rozdziale listingi pozwalają na użycie praktycznie pełnej składni zaproponowanego języka. Poniższy listing przedstawia przykładowe wyrażenie wykorzystujące dotychczas zaprezentowane klasy.

```
1  create.panel.title("PanelMain")
2        .containing(
3            textbox("Textbox1"),
4            panel("Inner"),
5                  textbox("Textbox2"),
6                  textbox("Textbox3"));
```

Listing 20. Przykład MDGL stworzonego w Javie.

Kolejne elementy składniowe języka zaimplementowano w klasach meta modelu. Są to odpowiednio nazwane metody typu setter, które dodatkowo nie są typu `void`. Zwracają referencję do samego siebie. Dzięki takiemu zabiegowi możliwe jest wywoływanie metod w sposób „łańcuchowy” *(ang. Method chaining)*. Tę technikę zastosowano już w klasie `Creator` w metodzie `title(String title)`. Jednak metoda ta zwraca referencję do obiektu typu `GuiGenerator`, a nie do obiektu, na rzecz którego została wywołana.



Rysunek 21: Diagram MetaModel

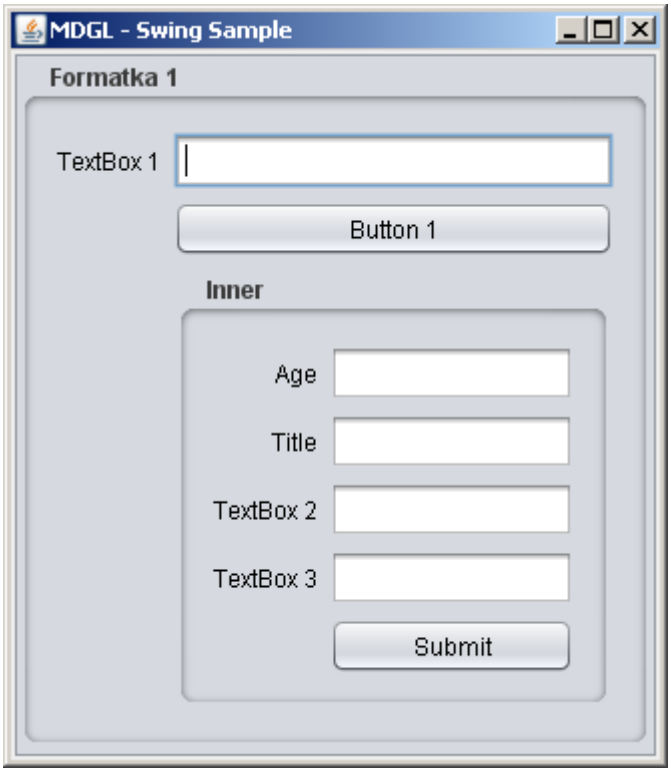
Na rysunku 21. przedstawiono hierarchię dziedziczenia klas odpowiedzialnych za przechowywanie meta modelu interfejsu użytkownika. Dla prostoty zostały zaznaczone tylko te metody, które są użyte podczas budowania wyrażania. Metoda `as(String _id)` została zaimplementowana w abstrakcyjnej klasie `AbstractMetaGUI`. W przypadku klas z gałęzi `AbstractMetaContainer`, wymagane jest wywołanie po niej metody `containing(AbstractMetaGUI... c)`. W kontekście klas dziedziczących po `AbstractMetaControl` byłoby to niezamierzonym efektem, a nawet poważnym błędem składniowym. Dlatego metoda `as(String _id)` w klasie `AbstractMetaContainer` zostaje przesłonięta i w nowej wersji zwraca wartość typu `AbstractMetaContainer`. Dzięki temu, że `AbstractMetaContainer` rozszerza `AbstractMetaGUI`, taka konstrukcja jest możliwa. Z tego właśnie powodu mogła ona być współdzielona między `AbstractMetaControl` i `AbstractMetaContainer` przez umieszczenie jej w klasie `AbstractMetaGUI`.

Dzięki tym wszystkim zabiegom udało się uzyskać składnię bardzo podobną do pseudokodu. Zaimplementowano ją w całości w Javie, jako wewnętrzny rodzaj języków typu DSL. Na listingu 21. widać końcowy efekt jaki udało się uzyskać.

```
1  create.panel.title("Formatka 1").
2       containing(
3            textbox("TextBox 1"),
4            button("Button 1"),
5              panel("Inner").as("inner").
6               containing(
7                  textbox("Age").as("tb_age"),
8                  textbox("Title").as("tb_title"),
9            textbox("TextBox 2"),
10           textbox("TextBox 3"),
11           button("Submit")));
```

Listing 21. Końcowa forma DSL – MDGL

Na rysunku 22. widać zakładany wynik działania wyrażenia z listingu powyżej na platformie Swing.



Rysunek 22: Wynik działania kodu z listingu 21 - MDGL na platformie Swing

Podsumowanie głównych technik, które zostały zastosowane w celu uzyskania efektu końcowego.

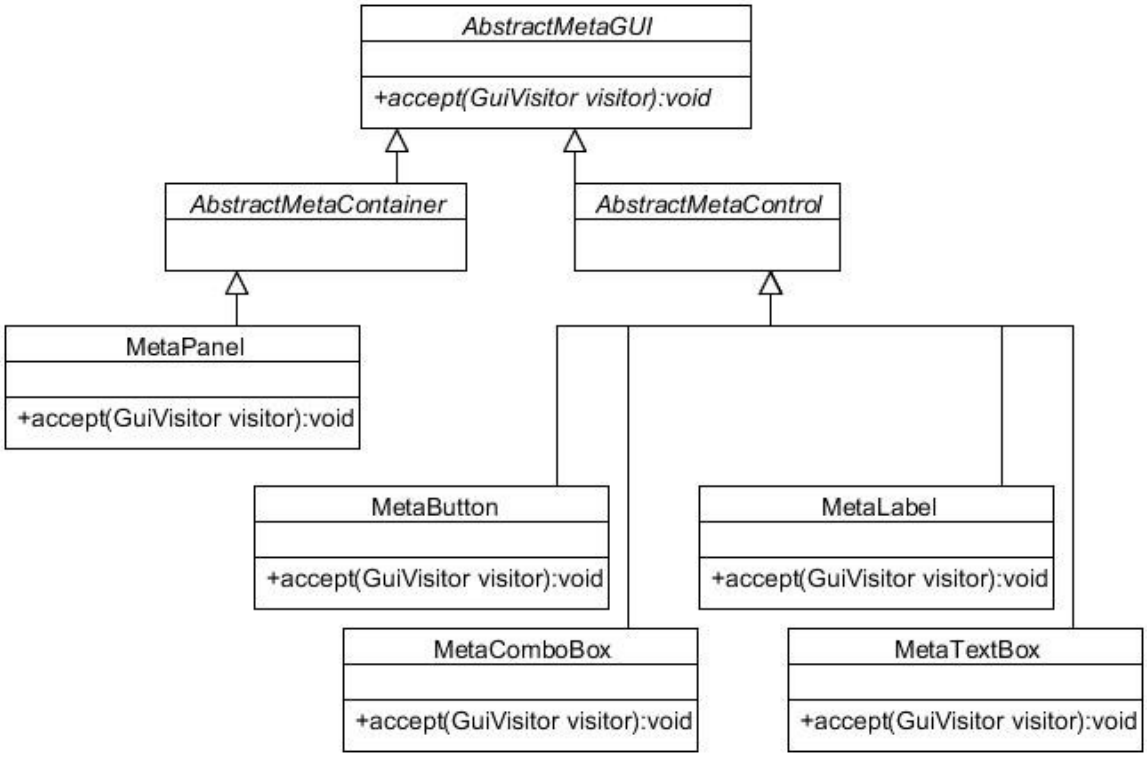
- Zagnieżdżone funkcje *(ang. Nested functions)*[17] – Czyli przekazywanie funkcji jako parametr innej funkcji. Typ zwracany funkcji zagnieżdżonej musi się zgadzać z typem argumentu w funkcji zewnętrznej. Funkcje są wywoływane od prawej do lewej, czyli najpierw wywoływana jest funkcja najbardziej zagnieżdżona.
- Łańcuchowe wywołanie metod *(ang. Method Chaining)*[17] – Wywoływanie funkcji jedna po drugiej w ciągu jest możliwe, jeżeli kolejne metody zwracają referencje do obiektu, na rzecz którego została ona wywołana lub inny obiekt, który może zostać wykorzystany w takim ciągu. W Java taka metoda nie może zwracać typu prymitywnego. W tym przypadku kolejność wywoływania jest następująca – od lewej do prawej.
- Sekwencja funkcji *(ang. Function Sequence)*[17] – Dzięki wprowadzonej w Java5 deklaracji metod ze zmienną ilością argumentów*(ang. varargs)*. Możliwe jest wywoływanie metod jako argumentów metody dowolnie często.
- Dzięki wprowadzonemu w Java5 mechanizmowi „covariant return”[18] dostajemy możliwość przesłonięcia metody z klasy nadrzędnej w hierarchii dziedziczenia, na podstawie zwracanego typu. Nowa metoda musi zwracać typ bardziej szczegółowy niż jej klasa nadrzędna. Zwracany typ jest bardziej szczegółowy w klasie, która jest niżej w hierarchii dziedziczenia.

5.3 Wykorzystane wzorce projektowe

W prototypowej implementacji nowego języka wykorzystano głównie dwa wzorce projektowe. Zostaną opisane na przykładzie tej implementacji w poniższym rozdziale.

5.3.1 *Wizytator (Visitor pattern)*

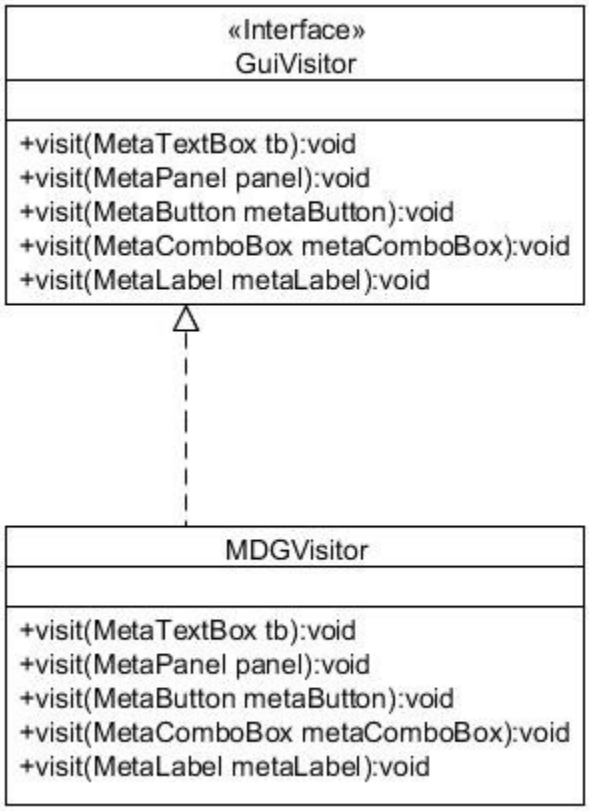
Wzorzec projektowy wizytator służy do odseparowania struktury danych od algorytmu, który na niej operuje. Daje też możliwość łatwego przełączania między algorytmami bez żadnych zmian w strukturze danych.



Rysunek 23: Meta Model interfejsu użytkownika - Visitor Pattern

W prototypie języka wzorzec został wykorzystany do interpretowania meta modelu interfejsu użytkownika. Dzięki dużej swobodzie i możliwości tworzenia wielu implementacji danego wizytatora, powstała możliwość łatwego adaptowania języka na nowe platformy. Wystarczy napisać nową implementację na nową platformę w celu skorzystania z dobrodziejstw nowego rozwiązania.

Na rysunku 23. widzimy meta model interfejsu użytkownika uwzględniający jedynie niezbędne elementy do implementacji wzorca projektowego wizytator. Każda klasa struktury meta modelu, która nie jest klasą abstrakcyjną, musi implementować metodę `accept(GuiVisitor visitor):void`. Dzięki niej wizytator może „odwiedzać” strukturę danych.



Rysunek 24: Wizytator - GuiVisitor

Rysunek pokazuje strukturę klasy, która może być wizytatorem. Każda taka klasa musi implementować interfejs `GuiVisitor` i przedstawić implementację każdej z wymaganych w interfejsie metod.

Struktura danych, w omawianym przypadku meta model interfejsu użytkownika, musi implementować metodę `accept(GuiVisitor visitor)`. Ciało metody tej będzie identyczne dla każdej klasy z hierarchii dziedziczenia meta modelu. Jednak ze względu na to, że niezbędne jest wywołanie konkretnej metody z obiektu implementującego interfejs `GuiVisitor`, konieczna jest implementacja tej metody w klasach z niższych poziomów hierarchii dziedziczenia. Słówko kluczowe `this`, wykorzystane w tej metodzie jest oczywiście w każdym przypadku referencją innego typu. Pozwala to na wywołanie odpowiedniego przeciążenia metody `visit` obiektu wizytatora. Przykładowa metoda widoczna jest na listingu 22. Właśnie w tym miejscu widać, w jaki sposób wzorzec projektowy wizytator, pozwala na odseparowanie logiki algorytmu, budującego w tym przypadku interfejs użytkownika, od meta modelu. Zmiana wizytatora, czyli tak naprawdę algorytmu, nie wymaga żadnej zmiany w meta modelu.

```
1  @Override
2  public void accept(GuiVisitor visitor) {
3     visitor.visit(this);
4  }
```

Listing 22. Przykładowa metoda accpet(GuiVisitor visitor), implementowana w każdej klasie meta modelu.

W aktualnej wersji prototypu języka powstały dwie implementacje GuiVisitora. Ze względu na wykorzystanie innego wzorca projektowego(podmiana komponentów – opisanego w rozdziale 5.3.2) obie klasy implementujące nazywają się tak samo – `MDGVisitor`. Na listingu 23. widać skróconą wersję wizytatora dla platformy GWT.

```
1  public class MDGVisitor implements
2                          GuiVisitor<MDGGWTAbstractPanel>  {
3
4    Stack<MDGGWTAbstractPanel> containers = new
5    @Override
6    public void visit(MetaTextBox tb) {
7       MDGGWTAbstractPanel tempContainer = containers.pop();
8       tempContainer.add(tb);
9       containers.push(tempContainer);
10   }
11   @Override
12   public void visit(MetaPanel metaPanel) {
13      MDGGWTAbstractPanel panel;
14      if (metaPanel.isCollapsible()) {
15        panel = new MDGCollapsiblePanel();
16      } else {
17        panel = new MDGLayoutPanel();
18        panel.setStyleName("MDGPanel");
19      }
20       panel.addTitle(metaPanel.getTitle());
21      containers.push(panel);
22      for (AbstractMetaGUI meta : metaPanel.controls()) {
23          meta.accept(this);
24      }
```

```
13          if (containers.size() > 1) {
14              MDGGWTAbstractPanel currentPanel = containers.pop();
15              containers.peek().add(currentPanel);
16          }
17      }
18  ...
19  }
```

Listing 23. Implementacja wizytatora dla platformy GWT.

Jak widać na listingu 23., na przykładzie metody obsługującej „odwiedzanie" MetaPanel'u, od implementacji wizytatora zależy kolejność przechodzenia po strukturze danych. W przypadku meta modelu interfejsu użytkownika wykorzystanego w implementacji prototypu, struktura danych jest bardzo prosta. Implikuje to na wygląd samego algorytmu „odwiedzania". Występuje tylko schodzenie w dół, w przypadku zagnieżdżonego elementu typu `MetaPanel`. W linijkach 22-24, widać sterowanie „odwiedzaniem" przez wywoływanie metody `accept(GuiVisitor visitor)` na kolejnych elementach zawartych w MetaPanel'u.

5.3.2 *Podmiana komponentów*

Wynikiem działania wyrażenia w nowym języku ma być gotowa klasa odpowiadająca idei MetaPanel na konkretnej platformie. W przypadku Swing, będzie to klasa rozszerzająca standardową klasę z Java SE API – JPanel. Natomiast na platformie GWT wykorzystano rozszerzoną o odpowiednią funkcjonalność klasę VerticalPanel. Najniższym wspólnym elementem w hierarchii dziedziczenia dla obu klas jest klasa Object. W związku z tym, to samo wyrażenie na dwóch różnych platformach musi zwracać wynik innego typu. Jak wiadomo, w języku Java nie ma możliwości przeciążania metod na podstawie zwracanego typu(chociaż sama maszyna wirtualna JVM obsługuje takie konstrukcje – np. Język Scala). W prototypie tę funkcjonalność zastąpiono podmianą odpowiednich komponentów. Podmiana polega na tworzeniu klas o identycznej nazwie, z metodami o takich samych sygnaturach(mogą być już inne zwracane typy), przechowywanych w identycznym pakiecie ale w innych fizycznych miejscach. Teraz dzięki odpowiednim wskazaniom ścieżki(*ang. classpath)* do tych klas możliwe jest, przezroczyste dla reszty kodu, zmienianie implementacji bez zmian samego kodu.

W prototypie rozwiązania podmiana komponentów wykorzystana jest na rzecz klasy `GuiGenerator`. Jest ona odpowiedzialna między innymi za użycie odpowiedniego obiektu implementującego interfejs `GuiVisitator`. Klasa ta wprowadza implementacje metody `containing (AbstractMetaGUI...c)`, odpowiedzialną za wygenerowanie interfejsu na daną platformę. Metoda ta jest wywoływana jako ostatnia w prototypie języka MDGL. Powoduje to, że musi zwracać w zależności od platformy wynik odpowiedniego typu.

```
1  package pl.marekdrob.mdg.generators;
2  ...
3  public class GuiGenerator extends AbstractGuiGenerator {
4  public GuiGenerator(AbstractMetaContainer metaContainerGUI) {
5            super(metaContainerGUI);
6            visitor = new MDGVisitor();
7  }
8  public MDGGWTAbstractPanel containing(AbstractMetaGUI... c) {
9            populateGui(c);
10           return ((MDGVisitor) visitor).getGui();
11 }
12 }
```

Listing 24. Implementacja klasy GuiGenerator wykorzystana na platformie GWT.

```
1  package pl.marekdrob.mdg.generators;
2  ...
3  public class GuiGenerator extends AbstractGuiGenerator {
4  public GuiGenerator(AbstractMetaContainer metaContainerGUI) {
5            super(metaContainerGUI);
6            visitor = new MDGVisitor();
7  }
8  public MDGGWTAbstractPanel containing(AbstractMetaGUI... c) {
9            populateGui(c);
10           return ((MDGVisitor) visitor).getGui();
11 }
12 }
```

Listing 25. Implementacja klasy GuiGenerator wykorzystana na platformie Swing.

Na listingach 24. i 25., widać dwie identyczne klasy. Różnią się tylko i wyłącznie zwracanym typem w metodzie `containing(AbstractMetaGUI... c)`. Odpowiednie manipulowanie classpath(argument przekazywany przy uruchamianiu konkretnej klasy lub wskazany przez zmienną środowiskową) stanowi bardzo łatwy i efektywny sposób, na obejście braku przeciążania metod na podstawie zwracanego typu. Takie rozwiązanie powoduje, że niezbędne jest budowanie oddzielnych plików jar dla danej platformy. Nie można po prostu połączyć obu implementacji klasy `GuiGenerator` i `MDGVisitor` w ramach jednego pliku jar.

5.4 Implementacja

Dzięki podziałowi kodu na część wspólną dla wszystkich platform, dodając nową platformę, która będzie obsługiwana przez prototyp języka, należy stworzyć jedynie następujące rzeczy:

- Klasę implementacją interfejs GuiVisitor, która odpowiada za interpretację struktury meta modelu i właściwe wygenerowanie na tej podstawie gotowego interfejsu użytkownika
- Implementację klasy GuiGenerator – jest ona niezbędna do „podpięcia” do części wspólnej odpowiedniego wizytatora.
- Implementację klasy odpowiadającej idei MetaPanel'u(kontener układający widgety w sposób wertykalny – jeden pod drugim). Przechowuje ona także implementację metod wyszukujących zadeklarowane elementy interfejsu użytkownika.

```
1  create.panel.title("Formatka 1").
2            containing(
3                  textbox("TextBox 1").as("tb_1"),
4                  button("Button 1").as("btn_1"),
5                  panel("InnerPanel").as("panel").
6                  containing(
7                        textbox("Age").as("tb_age"),
8                        textbox("Title").as("tb_title"),
9                        combobox("test",values),
10                       panel("InnerPanel2").as("panel").
```

```
                        containing(
                    textbox("Age").as("tb_age"),
                    textbox("Title").as("tb_title"))),
    textbox("TextBox 2").as("tb_2"),
    textbox("TextBox 3").as("tb_3"),
    button("Submit"));
```

Listing 26. Przykładowy kod MDGL – generujący formularz.

Zgodnie z założeniami nowego języka wyrażenie MDGL opisuje tylko wygląd nowego interfejsu użytkownika. Zostaje kwestia obsługi kontrolek w wygenerowanym interfejsie. Implementacja języka MDGL wprowadza następujące metody, które mogą być wywołana na rzecz, zwróconego przez wyrażenie, obiektu.

- $btn – Pozwala na wyszukanie w sposób hierarchiczny elementów typu Button na wygenerowanej formatce.
- $lb – Pozwala na wyszukanie w sposób hierarchiczny elementów typu Label na wygenerowanej formatce.
- $cbb – Pozwala na wyszukanie w sposób hierarchiczny elementów typu Combobox na wygenerowanej formatce.
- $tb – Pozwala na wyszukanie w sposób hierarchiczny elementów typu TextBox na wygenerowanej formatce.
- $ta – Pozwala na wyszukanie w sposób hierarchiczny elementów typu TextArea na wygenerowanej formatce.

```
containing.$tb("panel.panel.tb_age").setText("Wprowadź wiek");

containing.$tb("panel.panel.tb_age").addFocusListener(new
FocusListener() {
   @Override
   public void focusLost(FocusEvent e) {...}
   @Override
   public void focusGained(FocusEvent e) {...}
});
```

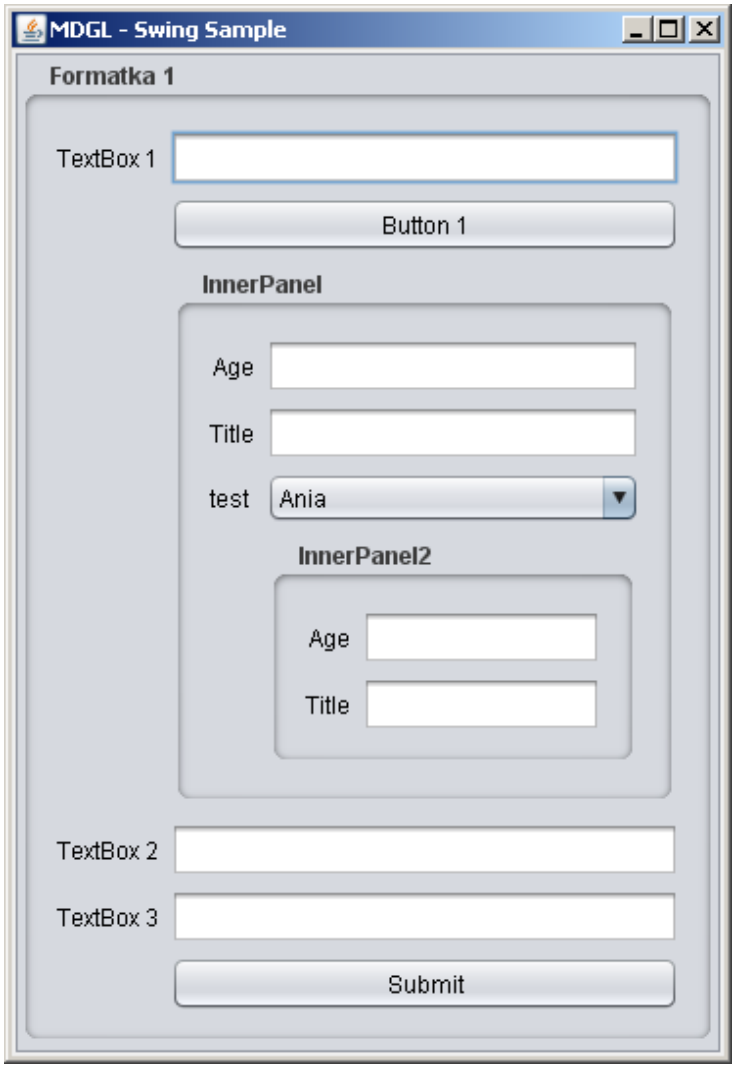
Listing 27. Przykład odwołania do zagnieżdżonych elementów GUI z listingu 26.

Jak widzimy oba dodane panele mają taki sam identyfikator, jednak odwołanie do nich możliwe jest tylko w kontekście hierarchii zagnieżdżenia więc taka konstrukcja jest jak najbardziej poprawna. Dochodzi tu oczywiście kwestia obsługi przypadków kiedy do metody wyszukującej zostanie wprowadzone błędne zapytanie. W obecnej implementacji metoda zwraca null, więc niezbędne jest sprawdzenie w czasie uruchomienia czy posiadamy odpowiednią referencje.

Swing

Podczas tworzenia implementacji dla platformy Swing niezbędne było stworzenie następujących elementów:

- pl.marekdrob.mdg.generators.GuiGenerator – klasa wykorzystana przy podmianie komponentów.
- pl.marekdrob.mdg.visitor.MDGVisitor – klasa „wizytatora", odpowiada za budowę interfejsu użytkownika na platformę Swing.
- pl.marekdrob.swing.controls.MDGJPanel – klasa dziedziczy po javax.swing.JPanel i odpowiada za właściwe rozmieszczenie elementów interfejsu użytkownika.
- layout.SpringUtilities – klasa użyta w Java Sun Tutorial. Pozwala w łatwy sposób układać elementy na JPanel. Wykorzystano ją w MDGJPanel.



Rysunek 25: Efekt działania kodu z listingu 26. na platformie Swing

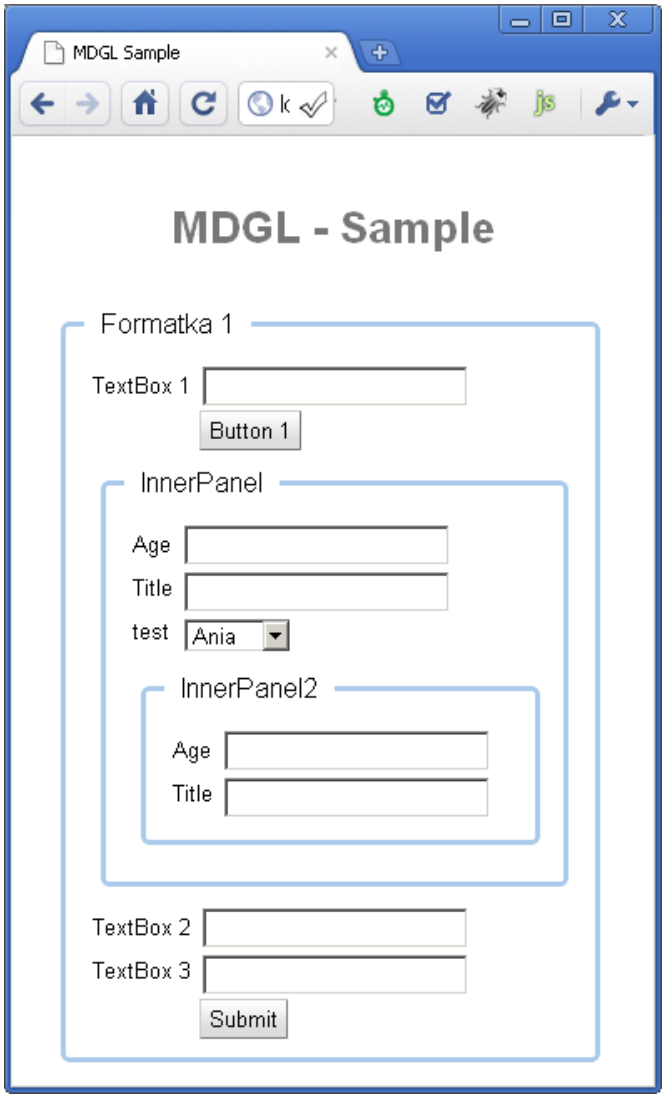
Przykład działania języka MDGL na platformie Swing widać na rysunku 25. Przedstawia on efekt uruchomienia kodu z listingu 26.

Google Web Toolkit

Analogicznie do implementacji dla Swing w ramach rozwiązania dla platformy GWT stworzono następujące elementy:

- pl.marekdrob.mdg.generators.GuiGenerator – klasa wykorzystana przy podmianie komponentów.
- pl.marekdrob.mdg.visitor.MDGVisitor – klasa „wizytatora”, odpowiada za budowę interfejsu użytkownika na platformę GWT.

- MDGGWTAbstractPanel i MDGLayoutPanel z pakietu pl.marekdrob.gwt.controls – klasy odpowiedzialne za odpowiednie rozmieszczenie elementów interfejsu użytkownika.
- pl.marekdrob.gwt.controls.gwt.xml – plik konfiguracyjny, wskazujący kompilatorowi miejsce kodu źródłowego nowych elementów graficznych.



Rysunek 26: Efekt działania kodu z listingu 26. na platformie GWT

Przykład działania języka MDGL na platformie GWT widać na rysunku 26. Przedstawia on efekt uruchomienia kodu z listingu 26.

Wnioski

Obie przygotowane implementacje, dzięki dobrze zaprojektowanej architekturze rozwiązania, współdzielą rdzeń. Stanowi on niezmienną część wspólną. Dzięki temu utrzymanie i pielęgnacja kodu jest dużo prostsza.

Stworzenie identycznego w wyglądzie interfejsu wygenerowanego dzięki listingowi 26. byłoby bardziej pracochłonne i dostarczałoby powtarzalny kod. Jeden ze sposobów utworzenia takiego interfejsu dla platformy Swing przedstawiono na listingu 27.

```
1  MDGJPanel main = new MDGJPanel();
2  main.setTitle("Formatka 1");
3  JTextField tb_1 = new JtextField(10);
4  main.add(new JLabel("TextBox 1"));
5  main.add(tb_1);
6  JButton btn_1 = new JButton("Button 1");
7  main.add(new JLabel(" "));
8  main.add(btn_1);
9  MDGJPanel panel = new MDGJPanel();
10 panel.setTitle("InnerPanel");
11 JTextField tb_age = new JTextField(10);
12 panel.add(new Jlabel("Age"));
13 panel.add(tb_age);
14 JTextField tb_title = new JTextField(10);
15 panel.add(new Jlabel("Title"));
16 panel.add(tb_title);
17 JComboBox combo = new JComboBox();
18 for(String[] s : values){
19     combo.addItem(s[0]);
20 }
21 panel.add(new Jlabel("test"));
22 panel.add(combo);
23 MDGJPanel panel2 = new MDGJPanel();
24 panel2.setTitle("InnerPanel2");
25 JTextField tb_age2 = new JTextField(10);
26 panel2.add(new Jlabel("Age"));
27 panel2.add(tb_age2);
28 JTextField tb_title2 = new JTextField(10);
29 panel2.add(new Jlabel("Title"));
```

```
30  panel2.add(tb_title2);
31  main.add(panel);
32  JTextField tb_2 = new JTextField(10);
33  main.add(new JLabel("TextBox 2"));
34  main.add(tb_2);
35  JTextField tb_3 = new JTextField(10);
36  main.add(new JLabel("TextBox 3"));
37  main.add(tb_3);
38  JButton submit = new JButton("Submit");
39  main.add(new JLabel(" "));
40  main.add(submit);
41  panel.add(panel2);
42  SpringUtilities.makeCompactGrid(panel2, 2, 2, 6, 6, 6, 6);
43  SpringUtilities.makeCompactGrid(panel, 4, 2, 6, 6, 6, 6);
44  SpringUtilities.makeCompactGrid(main,   6, 2, 6, 6, 6, 6);
```

Listing 28. Przykład stworzenia interfejsu z rysunku 25. w sposób tradycyjny na platformie Swing. W MDGL kod zapisano w listingu 26.

Łatwo zauważyć, że nawet przy tak prostym interfejsie programista zmuszony jest napisać dużą ilość kodu. Jest on bardzo nieczytelny i trudny w pelengacji. W porównaniu do deklaratywnego podejścia w języku MDGL z przykładu z listingu 26., który generuje ten sam interfejs użytkownika, kod z listingu 28. jest nieporównywalnie mniej ekspresyjny. Stwierdzenie na podstawie kodu, jak powinno wyglądać GUI, jest praktycznie niemożliwe. W tak prostym przypadku programista musi zwracać uwagę na bardzo niskopoziomowe API. Mimo, że jest bardzo potężne, to w tym wypadku jest zwyczajnie zbędne. Próba zmiany kolejności wyświetlania danego elementu przy takim kodzie jest bardzo niewygodna i może prowadzić do większego nieporządku w kodzie. W tym konkretnym przypadku MDGL sprawdza się idealnie. Nie dochodzi do utraty jego siły wyrazu, a zmiana sprowadza się do zmiany kolejności występowania konkretnych linijek.

5.5 Problemy przy implementacji

Początkowym zadaniem MDGL było wspieranie tworzenia interfejsu użytkownika również na podstawie modelu danych – podobnie jak GCL. Przy użyciu refleksji miał odczytywać strukturę danych i samodzielnie decydować, jakiego elementu graficznego użyć do wyświetlenia. Próba implementacji pierwszej wersji prototypu zakończyła się jednak niepowodzeniem. Dwie platformy wybrane do próbnej implementacji – Swing, GWT, pomimo, że bardzo podobne pod względem tworzenia interfejsów użytkownika, to w znacznym stopniu różnią się odnośnie środowiska uruchomienia.

5.5.1 Brak mechanizmu refleksji

Platforma GWT w części klienckiej nie wspiera mechanizmu refleksji. Próby obejścia tego braku sprowadzają się do tego, że niezbędne jest „zaznaczenie" klasy modelu danych. Dzięki takiemu zaznaczeniu można wykorzystać dwie techniki omijające brak refleksji po stronie klienta GWT. Takie zaznaczenie może być zrealizowane przez implementację konkretnego, ustalonego interfejsu programistycznego przez klasę takiego modelu.

- Taki obiekt, dzięki mechanizmowi GWT RPC, można wysłać na serwer, otrzymując dostęp do refleksji. W tym momencie można zinterpretować i odesłać taki wynik do części klienta. Największym argumentem przeciwko temu rozwiązaniu była wydajność. Ten sposób wymagałby ciągłego odwoływania się do serwera. Wystąpiłaby zależność od połączenia internetowego.
- Kolejnym pomysłem było wykorzystanie mechanizmu wiązania podczas kompilacji kodu Java do JavaScript przez kompilator GWT i dodanie własnego generatora. W takim generatorze istnieje dostęp do podobnego, ale uboższego mechanizmu, przypominającego klasyczną refleksję znaną z Java. Odpowiadałby on za stworzenie właściwych struktur. Na tej podstawie strona klienta mogłaby wygenerować interfejs bez użycia refleksji. Rozwiązanie to uchodzi za bardziej wydajne, niż pomysł z wysyłaniem obiektów na serwer. Pozostaje jednak „zaznaczanie" obiektów modelu danym interfejsem programistycznym. Dzieje się tak ponieważ, zarówno wołania GWT, jak i wywołanie własnego generatora, odbywa się na podstawie typów i nie może być użyte dla obiektu typu Object.

5.6 Zalety i wady przyjętych rozwiązań

Największą wadą jest obecność identyfikatorów przechowywanych w obiektach klasy String. Podczas wyszukiwania danego elementu interfejsu użytkownika, np., w celu dodania do niego dodatkowej obsługi lub pobrania jego aktualnej wartości, błąd w nazwie zostanie odkryty dopiero podczas działania danego programu. Również wykorzystanie standardowo dostępnych w IDE mechanizmów do refaktorowania takich identyfikatorów, przechowywanych w obiektach typu String, jest niemożliwe. Pozostaje jedynie bardzo niebezpieczny sposób, z użyciem takich funkcji, jak „znajdź i zamień na".

Obecnie język pozwala na budowanie prostych interfejsów, a elementy rozmieszczane są jeden pod drugim. W tym względzie jest on bardzo nieelastyczny. Narzuca duże uogólnienia. W związku z tym, wykorzystanie stworzonego języka może zostać znacząco ograniczone do typowych i prostych formularzy.

Język jest w tym momencie przeznaczony praktycznie tylko do budowy formularzy o określonej strukturze. Nie jest możliwe w tym momencie, np., umieszczenie dwóch paneli obok siebie, bez wykorzystania API konkretnej platformy.

5.7 Plan rozwoju

Koncepcja języka jest bardzo młoda i wymaga udoskonaleń na wielu płaszczyznach. Również sama implementacja prototypu powinna być udoskonalona ze szczególną uwagą na wydajność języka.

Główne elementy wymagające poprawy to:

- Rozszerzenie języka, dzięki czemu można będzie obsługiwać większą ilość elementów graficznych.
- Zwiększenie jego elastyczności przez dostarczenie większej możliwości decydowania o wyglądzie końcowym, ale nie w niskopoziomowy sposób. Wiąże się to z wprowadzeniem nowych konstrukcji, które umożliwiłyby właściwe opisanie rozłożenia widgetów.
- Dodanie możliwości automatycznego generowania interfejsu użytkownika na podstawie modelu danych – podobny mechanizm do języka GCL. W tym momencie jest to niemożliwe, ze względu na brak mechanizmu refleksji po stronie kodu klienta na platformie GWT.
- Stworzenie implementacji języka na platformę mobilną Android. Bardzo ważna jest optymalizacja klasy wizytatora, ponieważ urządzenia mobilne posiadają zdecydowanie mniejsze zasoby sprzętowe. Potrzebna będzie także nowa idea budowy takiego interfejsu. Zagnieżdżone w sobie panele nie sprawdzą się na tak małym ekranie.

6 *Podsumowanie*

W niniejszej pracy opisano nową koncepcję tworzenia graficznych interfejsów użytkownika. Owa koncepcja opiera się na podejściu deklaratywnym do definiowania struktury interfejsu użytkownika. Język pozwala na pisanie kodu dużo bardziej zrozumiałego dla innego członka zespołu programistycznego, ze względu na jego prostotę i wykorzystanie ogólnie znanej terminologii. Osoby, które nie specjalizują się w tworzeniu graficznych interfejsów użytkownika, powinny dostrzec ogólną strukturę interfejsu, czytając jedynie kod nowego języka.

Przedstawioną w pracy przykładową implementację, zaproponowanego w rozdziale 3 . języka, można z łatwością użyć w istniejących aplikacjach. Obecnie istnieje wersja na platformy Swing i GWT w języku Java. Wprowadzenie obsługi kolejnych platform jest bardzo proste. Architekturę tego rozwiązania projektowano głównie z myślą wprowadzenia implementacji na jak największej ilości platform.

Dalsze możliwości rozwoju obejmują, między innymi, implementację rozwiązania w języku C#. Dzięki czemu istniałaby szansa użycia jej w aplikacjach tworzonych na platformie .NET.

Bibliografia

[1] Mariusz Trzaska, Automatyczne generowanie graficznych interfejsów użytkownika, 2008, X Krajowa Konferencja Inżynierii Oprogramowania 2008 (KKIO'2008) ISBN: 978-83-206-1703-0 pp. 125-134
[2] Adam Nathan, WPF 4 Unleashed , 2010, ISBN: 978-0672331190
[3] Google Web Toolkit, http://code.google.com/webtoolkit/
[4] Declarative Layout with UiBinder
http://code.google.com/webtoolkit/doc/latest/DevGuideUiBinder.html
[5] http://javafx.com/
[6] Mariusz Trzaska, GCL – An Easy Way for Creating Graphical User Interfaces, WMSCI 2010. Orlando, USA. 2010. pp. 403 - 410. ISBN-13: 978-1-934272-98-5
[7] dr inż. Mariusz Trzaska, GCL Official site, http://code.google.com/p/gcl-dsl/
[8] Cucumber Official site, http://cukes.info/
[9] http://en.wikipedia.org/wiki/Behavior_Driven_Development
[10] Cay S. Horstmann , Gary Cornell, Core Java, Volume II - Advanced Features, 2008
[11] http://www.eclipse.org/downloads/
[12] JLex: A Lexical Analyzer Generator for Java
http://www.cs.princeton.edu/~appel/modern/java/JLex/
[13] CUP: LALR Parser Generator in Java, , http://www2.cs.tum.edu/projects/cup/
[14] ODRA (Object Database for Rapid Application development) Description and Programmer Manual, http://sbql.pl/various/ODRA/ODRA_manual.html
[15] https://dev.kis.p.lodz.pl:8443/svn/egovbus/ODRA2/trunk/res/lexer.lex
[16] https://dev.kis.p.lodz.pl:8443/svn/egovbus/ODRA2/trunk/res/parser.cup
[17] Martin Fowler, Implementing an Internal DSL, 2007
http://martinfowler.com/dslwip/InternalOverview.html
[18] Katherine Sierra, Bert Bates , SCJP Sun Certified Programmer for Java 5 Study Guide (Exam 310-055) (Certification Press), 2005

Listingi

Rysunki

Słownik

WPF - Windows Presentation Foundation, technologia wykorzystywana w tworzeniu graficznych interfejsów użytkownika na platformie .NET
XML - Extensible Markup Language
XAML - Extensible Application Markup Language
GWT – Google Web Toolkit
UiBinder – Deklaratywny język opisujący GUI w używany w GWT.
.NET - stworzona przez Microsoft platforma programistyczna
SilverLight – Webowy odpowiednik WPF. Wykorzysutje XMAL.
AWT – Abstract Window Toolkit,
Widget – element graficznego interfejsu użytkownika.
GUI – Grafical User Interface – graficzny interfejs użytkownika
Kontrolka – element graficznego interfejsu użytkownika