Programowanie.NET / Programowanie aplikacji lokalnych

2025/26

Instrukcja domowa

Framework WPF/XAML

Prowadzący: Tomasz Goluch

Wersja: 1.0

I. WPF/XAML

Windows Presentation Foundation jest następcą Windows Forms i jeśli nie wymagamy kompatybilności ze starymi wersjami framework'a (wcześniejszymi niż .NET 3.0) to powinniśmy z niego skorzystać. Posiada wiele udoskonaleń w stosunku do poprzednika np. zagnieżdżanie kontrolek.

Interfejs graficzny definiowany jest przy pomocy deklaratywnego języka znaczników XAML i jest odseparowany od logiki. Należy pamiętać, że XAML uwzględnia wielkość liter (jest case-sensitive). Nazwy znaczników (takie jak `Button`) odpowiadają nazwom klas WPF, a atrybuty (takie jak `Content`) ich właściwościom.

Logika jest implementowana w plikach code-behind, które są dołączone do kodu znaczników XAML za pomocą częściowych definicji klas. Z plików XAML podczas kompilacji generowane są – w folderze: obj\<konfuguracja >\<wersja.NET>, np.: obj\Debug\net7.0-windows – pliki kodu z rozszerzeniem *.g.cs. Można sprawdzić, że po zamianie tych plików (wykluczając z projektu plik XAML i dodając do projektu odpowiadający mu plik *.g.cs) program nadal będzie się kompilował i nie zauważymy zmian w jego działaniu.

Domyślnie w aplikacji występują dwa pliki XAML:

- App.xaml – głównym znacznikiem jest `Application` wraz z atrybutem `x:Class` który zawiera nazwę klasy częściowej zawierającej logikę i zaimplementowanej w pliku code-behind, którego nazwa jest identyczna jak pliku XAML z dodanym suffixem .cs. Następnie podane są mapowane przestrzenie nazw CLR do XAML (atrybuty `xmlns xmlns:x xmlns:local`) oraz domyślne okno aplikacji (atrybut `StartupUri`). Ponadto we właściwości `Application.Resources` można zdefiniować zasoby widoczne w całej aplikacji.
- MainWindow.xaml – Domyślnie jest to główne okno aplikacji, ale tylko jeśli tak określono w App.xaml. Jako główny znacznik zawiera `Window`, który poza atrybutami opisanymi przy okazji znacznika `Application` posiada `Title` zawierający tekst wyświetlany w tytule okna oraz `mc:Ignorable` który określa prefiksy przestrzeni nazw XML, które będą ignorowane oraz. Przykładowo `mc:Ignorable="d"` pozwala na stosowanie elementów d:Height i d:Width, które są dostępne tylko w czasie projektowania i są ignorowane po standardowej kompilacji programu.). Ponadto we właściwości `Window.Resources` można zdefiniować zasoby widoczne lokalnie w oknie.

Przestrzenie nazw w plikach XAML można mapować na dwa sposoby w zależności od prefiksu[1]:

- `http://schemas` – wykorzystywany jest atrybut `XmlnsDefinition`, który używany do rozpoznawania typów przez moduł zapisujący obiekty XAML lub kontekst schematu XAML. Przyjmuje dwa parametry: nazwę przestrzeni nazw XML/XAML i nazwę przestrzeni nazw CLR. np.:
```
[assembly:
XmlnsDefinition("http://schemas.microsoft.com/winfx/2006/xaml/presentation"
, "<CLR_namespace>")]
```

- `clr-namespace` – pozwalają mapować przestrzenie nazw z assembly tak aby zdefiniowane tam klasy (np. konwertery) były widoczne bezpośrednio w XAML'u. Przykładowo, atrybut: `xmlns:local="clr-namespace:MyAppNS;assembly=MyApp"` mapuje przestrzeń nazw

[1] https://learn.microsoft.com/en-us/dotnet/desktop/wpf/advanced/xaml-namespaces-and-namespace-mapping-for-wpf-xaml

XAML `local` na przestrzeń nazw CLR `MyAppNS` z assembly `MyApp`. W przypadku kiedy przestrzeń nazw CLR jest zdefiniowana w tym samym assembly co kod XAML to część `assembly=MyApp` można pominąć. Klasy z tak zmapowanej przestrzeni są widoczne w kodzie XAML i można się do nich odwoływać z wykorzystaniem nazwy przestrzeni XAML: `<local:ExampleClass/>`.

Należy pamiętać, że zagnieżdżone przestrzenie nazw oraz klasy nie będą w ten sposób widoczne i trzeba dla nich zdefiniować nowe przestrzenie nazw XAML. Dzieje się tak ponieważ analizator składni XAML nie ma możliwości rozróżnienia, czy kropka jest częścią nazwy typu zagnieżdżonego, czy częścią nazwy przestrzeni nazw.

II. XAML Designer

Najprostszym i najbardziej intuicyjnym sposobem tworzenia GUI w WPF jest wykorzystanie designera XAML'a dostępnego w Visual Studio oraz w Blend for Visual Studio[2]. Pozwala on tworzyć interfejs użytkownika, przeciągając kontrolki z okna Toolbox (okno Assets w Blend for Visual Studio) i ustawiając właściwości w oknie Properties. Na bieżąco można podglądać i edytować wygenerowany tak kod XAML co pozwala na bieżąco poprawić w nim te aspekty, które z poziomu Designera mogą być trudne albo niemożliwe do wykonania. Aby móc pozycjonować i układać elementy wizualne należy wybrać jeden z dostępnych pojemników (ang. layout container/panel)[3], które można też w sobie zagnieżdżać. Najpopularniejsze z nich to:

- `Canvas` (płótno) – obszar, w którym można umieszczać obiekty podrzędne, używając współrzędnych względnych do obszaru płótna;
- `Grid` (siatka) – obszar domyślny dla nowych projektów, składa się z kolumn i wierszy. Elementy podrzędne siatki są mierzone i układane zgodnie z ich przypisaniem do wierszy/kolumn ustawionym za pomocą dołączonych właściwości Grid.Row i Grid.Column;
- `StackPanel` (stos) – Układa elementy podrzędne w pojedynczym wierszu, który może być zorientowany poziomo lub pionowo.
- `RelativePanel` (względny) – obszar, w którym można pozycjonować i wyrównywać obiekty podrzędne względem siebie lub panelu nadrzędnego. Przydatny do tworzenia GUI, które nie mają wyraźnego liniowego wzorca; to znaczy układów, które nie są zasadniczo ułożone w stosy, zawinięte lub tabelaryczne, czyli takie gdzie można by naturalnie użyć StackPanel lub Grid.
  Jeśli interfejs użytkownika składa się z wielu zagnieżdżonych paneli, RelativePanel jest dobrą opcją do rozważenia.

Po dodaniu kilku kontrolek można zmieniać ich położenie i kolejność przeciągając je kursorem myszy albo wybierając odpowiednie opcje (Kolejność/Wyrównaj/Układ/Grupuj w …) z menu kontekstowego okna Zarys dokumentu (*ang.* Document Outline). W przypadku nakładających się elementów aby ustalić ich kolejność nakładania się należy wykorzystać właściwość ZIndex. Ma on pierwszeństwo przed kolejnością elementów wyświetlanych w oknie Zarys dokumentu, a element o wyższej wartości ZIndex pojawia się na wierzchu.

III. Zasoby logiczne[4]

[2] https://learn.microsoft.com/en-us/visualstudio/xaml-tools/creating-a-ui-by-using-xaml-designer-in-visual-studio
[3] https://learn.microsoft.com/en-us/uwp/api/windows.ui.xaml.controls.panel#panel-derived-classes
[4] https://learn.microsoft.com/en-us/dotnet/desktop/wpf/systems/xaml-resources-overview

W WPF obok zwykłych zasobów binarnych .NET (takich jak: mapy bitowe, czcionki, ciągi znaków, które można osadzić w assembly lub spakować w pliku zasobów) dodatkowo wprowadzono zasoby logiczne. Umożliwiają one definiowanie obiektów w języku XAML, które nie są częścią drzewa wizualnego, ale można ich używać w interfejsie użytkownika. Jednym z przykładów zasobu logicznego jest `Brush`, który służy do zapewnienia schematu kolorów. Generalnie obiekty te definiuje się jako zasoby, z których korzysta wiele elementów aplikacji. Zasoby logiczne dzielimy na statyczne i dynamiczne:

- `StaticResource` – zostanie rozwiązany i przypisany do właściwości podczas ładowania kodu XAML, czyli przed uruchomieniem aplikacji. Wszelkie późniejsze próby zmiany tego zasobu w słowniku zasobów zostaną zignorowane. Definicja w XAML (przykład dotyczy kontrolki użytkownika, w przypadku okna należy zamienić `UserControl` na `Window`):

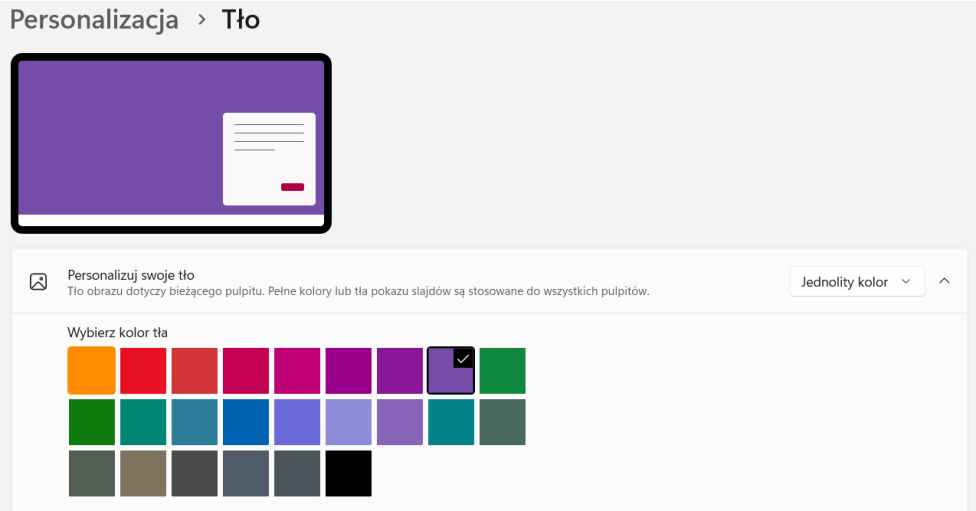
```
<UserControl.Resources>
    <SolidColorBrush x:Key="MyBrush" Color="#05E0E9"/>
</UserControl.Resources>
```

  Użycie zasobu w kodzie XAML: `Foreground="{StaticResource MyBrush}"`.

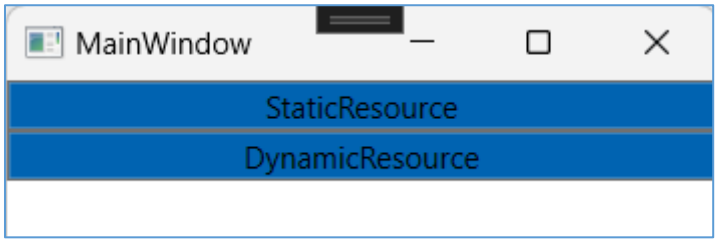
- `DynamicResource` – podczas początkowej kompilacji utworzone zostanie tymczasowe wyrażenie, co pozwoli odroczyć wyszukiwanie zasobów do momentu, gdy żądana wartość zasobu będzie faktycznie wymagana do skonstruowania obiektu. Może to potencjalnie nastąpić po załadowaniu strony XAML.

```
<StackPanel>
    <Button>
        <Button.Background>
            <SolidColorBrush
    Color="{StaticResource {x:Static
    SystemColors.DesktopColorKey}}" />
        </Button.Background>
        StaticResource
    </Button>
    <Button>
        <Button.Background>
            <SolidColorBrush
    Color="{DynamicResource {x:Static
    SystemColors.DesktopColorKey}}" />
        </Button.Background>
        DynamicResource
    </Button>
</StackPanel>
```

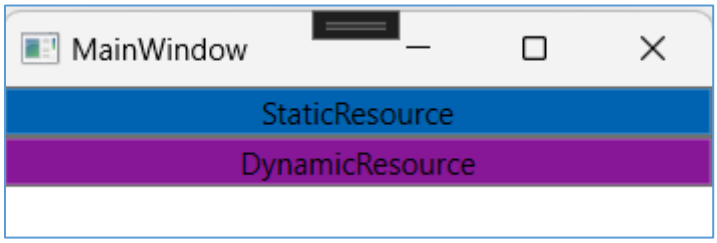
W powyższym przykładzie aby zobaczyć różnicę statycznym i dynamicznym należy po uruchomieniu programu zmienić kolor tła w systemie Windows: Ustawienia → Personalizacja → Tło i wybrać jednolity kolor jeśli nie było tak ustawione oraz zmienić kolor na inny.



Zmiana koloru będzie widoczna jedynie w przypadku zasobu dynamicznego. Wygląd aplikacji przed:



i po zmianie koloru tła:



IV. Style

Stylizować można dowolne elementy GUI, wymaganiem jest to, ze muszą implementować `FrameworkElement` (lub `FrameworkContentElement`). Zwyczajowo style są deklarowane w plikach XAML w sekcji zasobów (ang. `Resources`). Style są zasobami i podlegają tym samym regułom zakresu, zatem to gdzie zostanie zadeklarowany styl określa gdzie może on zostać zastosowany. Największy zakres (całej aplikacji) będzie w korzeniu (element `Application`). Style możemy stosować na trzy sposoby:

- niejawnie – wykorzystując atrybut `TargetType` oznaczamy do jakiego typu kontrolek styl będzie miał zastosowanie. W poniższym przypadku będą to wszystkie pola tekstowe będące w jego zakresie.

```
<Style TargetType="TextBox">
    <Setter Property="HorizontalAlignment" Value="Center" />
    <Setter Property="FontFamily" Value="Comic Sans MS"/>
    <Setter Property="FontSize" Value="14"/>
```

```
</Style>
```

- jawnie – w tym celu do powyższego przykładu należy dodać atrybut `x:Key`, który oznacza, że tylko elementy posiadające jawne odwołanie do stylu będą go stosować.

```
<Style x:Key="TitleText" TargetType="TextBox">
                    ...
</Style>
```

Przykład wykorzystania w elemencie `TextBox`:

```
<TextBox Style="{StaticResource TitleText}"...
```

- programowo – należy pobrać styl z kolekcji zasobów i przypisać go do właściwości `Style` elementu:

```
InputNumberTextBox.Style = (Style)Resources["TitleText"];
```

Aby móc odwołać się do elementu w pliku code-behind musi on mieć zdefiniowaną nazwę w XAML'u (atrybut `x:Name`).

Style można rozszerzać (atrybut `BasedOn`) tworząc nowy styl oparty na już istniejącym rozbudowując go o dodatkową zawartość:

```
<Style BasedOn="{StaticResource {x:Type TextBox}}"
    TargetType="TextBox"
    x:Key="TitleText">
        ...
</Style>
```

Wykorzystanie w kodzie stylu rozszerzonego niczym nie różni się od wykorzystanie „zwykłego" stylu.

V. Szablony

W przypadku kiedy funkcjonalność zapewniana przez style staje się niewystarczające z pomocą przychodzą własne szablony wielokrotnego użytku, którymi można dostosować strukturę wizualną i zachowanie istniejących kontrolek. W designerze VS WPF można wygenerować szablon istniejącej kontrolki. W tym celu należy kliknąć prawym przyciskiem myszy wybraną kontrolkę i z menu kontekstowego wybrać: Edytuj szablon → Utwórz kopię. Szablon kontrolki przepisuje wygląd wizualny całej kontrolki i jest zaaplikowany przez ustawienie właściwości Control.Template, dlatego można użyć stylu, aby zdefiniować lub ustawić szablon. Poniższy przykład tworzy przycisk w postaci czerwonej elipsy wewnątrz siatki będącej korzeniem kontrolki. Aby móc wyświetlić tekst umieszczony pomiędzy znacznikiem otwierającym i zamykającym elementu `Button` należy jawnie zdefiniować `ContentPresenter` i zmienić jego domyślne wyrównanie z lewego górnego na wycentrowane.

```
<ControlTemplate TargetType="Button" x:Key="RoundedButton" >
    <Grid >
        <Ellipse Fill="Red"/>
        <ContentPresenter HorizontalAlignment="Center"
                          VerticalAlignment="Center"/>
    </Grid>
```

```
</ControlTemplate>
```

Jeżeli chcemy wykorzystać oryginalny kolor pierwszoplanowy oraz kolor tła z przycisku możemy wykorzystać `TemplateBinding`, który wiąże właściwość elementu z `ControlTemplate`, z publiczną właściwością zdefiniowaną przez kontrolkę:

```
<ControlTemplate TargetType="Button" x:Key="RoundedButton" >
      ...
         <Ellipse Fill="{TemplateBinding Background}"
                  Stroke="{TemplateBinding Foreground}"/>
      ...
</ControlTemplate>
```

To samo możemy uzyskać stosując rusując elisę przy pomocy klasy `Path`:

```
<ControlTemplate TargetType="Button" x:Key="RoundedButton" >
      ...
         <Path Stretch="Uniform"
               UseLayoutRounding="False"
               Fill="{TemplateBinding Background}"
               Stroke="{TemplateBinding Foreground}">
             <Path.Data>
                 <EllipseGeometry RadiusX="1" RadiusY="1"/>
             </Path.Data>
         </Path>
      ...
</ControlTemplate>
```

Wykorzystanie szablonu przycisku w kodzie:

```
<Button Height="100" Width="100" Template="{StaticResource RoundedButton}" >
    STOP
</Button>
```

Przycisk z zastosowanym szablonem nadal zachowuje się jak przycisk kiedy go naciśniesz, zostanie uruchomione zdarzenie `Click`. Jednak wszystkie wizualne interakcje, takie jak zmiana wyglądu po najechaniu myszką na przycisk przestają działać i muszę być ponownie zdefiniowane przez szablon. W tym celu należy dodać nowy `Trigger` to kolekcji `ControlTemplate.Triggers`. Aby w elemencie `Setter` były widoczna była właściwość której wartość chcemy ustawić należy nazwać element w którym się ona znajduje (w naszym przypadku `Path` albo `Ellipse`) i nazwę przypisać do atrybutu:

```
<Path x:Name="backgroundElement"...


     ...
    </Grid>
    <ControlTemplate.Triggers>
        <Trigger Property="IsMouseOver" Value="true">
            <Setter Property="Fill"
                    TargetName="backgroundElement" Value="AliceBlue"/>
        </Trigger>
    </ControlTemplate.Triggers>
</ControlTemplate>
```

VI. Drzewo logiczne i wizualne WPF[5]

XAML opisuje relacje pomiędzy elementami interfejsu użytkownika w postaci drzewa logicznego. To abstrakcyjne drzewo jest wzbogacone o niejawne elementy składni XAML. Przykładowo, kontrolka po `ListBox` zaraz utworzeniu będzie zawierała właściwość `ItemCollection` o nazwie `Items` czyli niejawnie `<ListBox.Items>`. Drzewo logiczne pozwala na:

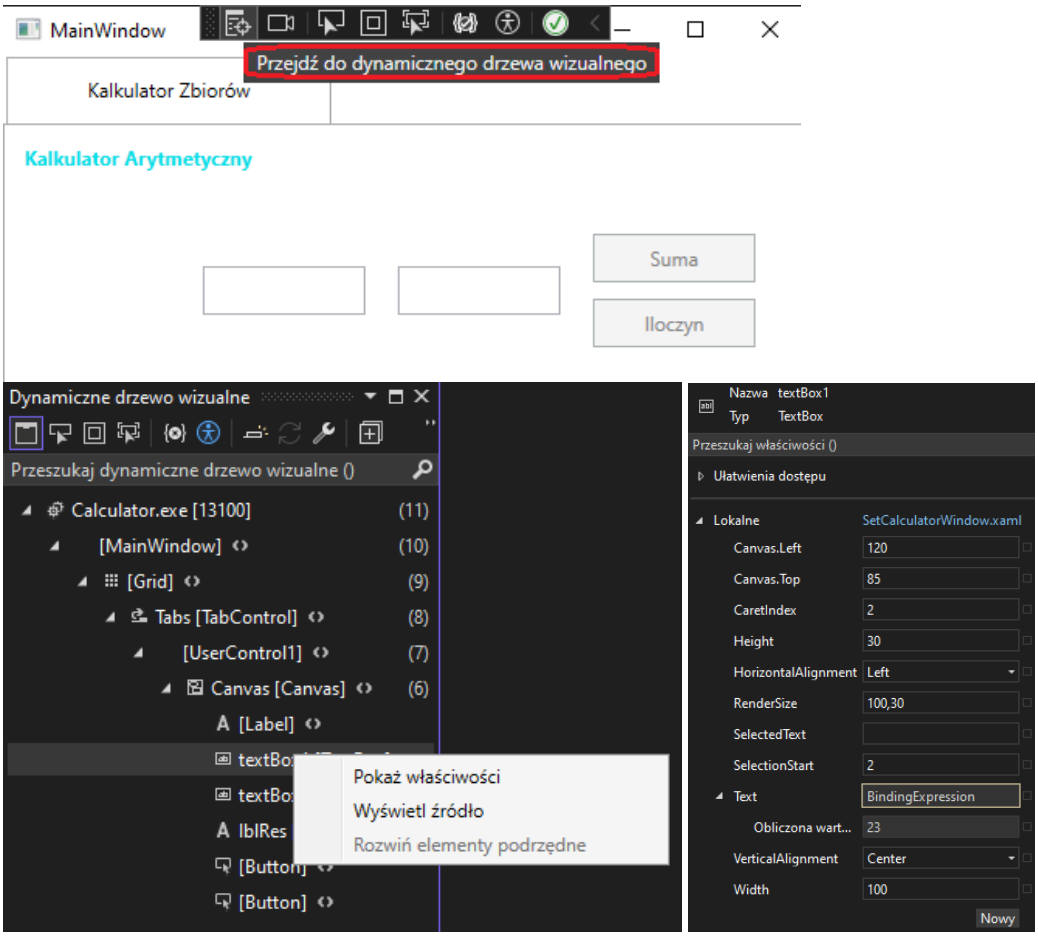
- dziedziczenie wartości DependencyProperty (opisane dalej w instrukcji) przez elementy potomne;
- rozwiązywanie odniesień do zasobów logicznych poprzez przeszukiwanie właściwości Resources zawierającej w ResourceDictionary poszukiwany klucz. Przeszukiwanie odbywa się w górę drzewa odwiedzając wszystkie `FrameworkElement` (lub `FrameworkContentElement`);
- wyszukiwanie nazw elementów dla powiązań;
- przekierowanie RoutedEvents.

Drzewo wizualne opisuje strukturę obiektów wizualnych dziedziczących po klasie bazowej `Visual`. Jest powiązane ze zdarzeniami kierowanymi (routed event), a ich trasy przebiegają głównie wzdłuż drzewa wizualnego, a nie logicznego. Ich zastosowanie łatwo opisać na przykładzie kiedy dodajemy do przycisku grafikę rozbudowując jego poddrzewo wizualne. Kliknięcie w pixele obrazka, który nie obsługuje zdarzenia kliknięcia, nie powinno zaburzyć działania samego przycisku. Jest to możliwe ponieważ zdarzenie `ButtonBase.Click` jest typu `RoutedEventHandler` i jest w stanie wywołać procedury obsługi na wielu elementach nasłuchujących w wizualnym drzewie elementów. W zależności od definicji kierowanego zdarzenia, jego propagacja w momencie wystąpienia w elemencie źródłowym może[6]:

- bąbelkować przez drzewo elementów od elementu źródłowego do elementu głównego, którym zazwyczaj jest strona lub okno;
- być tunelowana przez drzewo elementów od korzenia do elementu źródłowego;
- być ograniczona tylko do elementu źródłowego.

Drzewa wizualne WPF wraz z dependency properties można przeglądać, przy użyciu wizualizatora WPF.

[5] https://learn.microsoft.com/en-us/dotnet/desktop/wpf/advanced/trees-in-wpf
[6] https://learn.microsoft.com/en-us/dotnet/desktop/wpf/events/routed-events-overview#routing-strategies



Klasa `LogicalTreeHelper` pozwala na eksplorację drzewa logicznego, a `VisualTreeHelper` wizualnego.

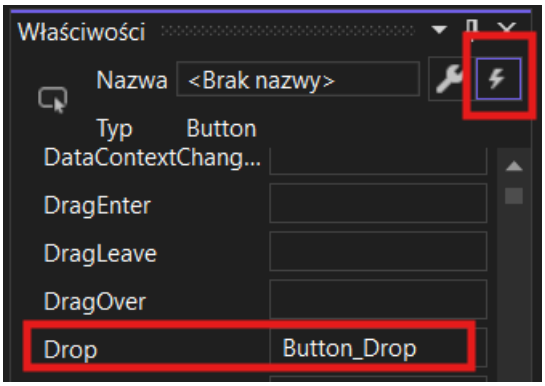
VII. Zdarzenia trasowane (ang. Routed Event[7])

Aby obsłużyć zdarzenie w XAML, wystarczy dodać w elemencie atrybutowi reprezentującemu nazwę zdarzenia wartość określającą nazwę procedury obsługi zdarzeń implementującej odpowiedni typ delegata. Najczęściej wykorzystywanym jest zdarzenie o nazwie W designerze jest to domyślnie obsługiwane tak, że po dwukrotnym kliknięciu na kontrolce przycisku następuje dodanie atrybutu `Click="Button_Click"` oraz handlera o korespondującej nazwie o sygnaturze delegata `RoutedEventHandler`:

```
delegate void RoutedEventHandler(object sender, RoutedEventArgs e);
```

Parametr `sender` jest identyczny dla wszystkich delegatów obsługi zdarzeń trasowanych i określa element, do którego dodawana jest obsługa zdarzeń. Parametr `e` określa dane dla zdarzenia. Aby dodać obsługę dla dowolnego zdarzenia trasowanego należy z menu kontekstowego wybrać: właściwości a następnie kliknąć przycisk z piorunem reprezentujący procedury obsługi zdarzenia dla wybranego elementu.

[7] https://learn.microsoft.com/en-us/dotnet/desktop/wpf/advanced/routed-events-overview



W kolejnym kroku podajemy nazwę handlera i zatwierdzamy enterem. W powyższym przykładzie dodaliśmy obsługę upuszczenia obiektu w granicach elementu. Musimy jeszcze ustawić możliwość upuszczania zmieniając domyślną wartość właściwości `AllowDrop="True"` elementu obsługującego zdarzenie. Kod obsługi zdarzenia będzie posiadał sygnaturę bardziej szczegółowego delegata ponieważ parametr `e` będzie typu `DragEventArgs` posiadającym właściwość `Data`, zawierającą zawartość schowka operacji przeciągania.

```
private void Button_Drop(object sender, DragEventArgs e)
{
    var data = e.Data;
}
```

Obsługi zdarzenia można dodać w kodzie:

```
b1.AddHandler(Button.DropEvent, new DragEventHandler(Button_Drop));
```

Lepiej wykorzystać operator `+=` poza krótszą składną sprawdzana jest zgodność typów dodawanych handlerów:

```
b1.Drop += new DragEventHandler(Button_Drop);
```

W obydwu przypadkach wymaga to nadania nazwy elementowi:

```
<Button x:Name="b1"...
```

Jeśli zdarzenie uważamy za obsłużone i nie zakładamy aby wymagane były dodatkowe działania przez inne elementy znajdujące się na drodze zdarzenia należy zaznaczyć je jako obsłużone:

```
e.Handled = true;
```

Pozwoli to na zabezpieczenie się przed jego ponownym, najprawdopodobniej błędnym obsłużeniem. Należy pamiętać, że propagowanie obsłużonego zdarzenia nie będzie zatrzymane i nadal będzie można je obsłużyć ustawiając parametr `handledEventsToo` na `true`:

```
b1.AddHandler(Button.DropEvent,
                new DragEventHandler(Button_Drop),
                handledEventsToo:true);
```

VIII. [Custom] Dependency Property, Dependency Object oraz Attached Property

Dependency property pozwalają między innymi na:

- nadawanie im wartości w stylach[8] i odniesieniach do zasobów dynamicznych (opisanych w poprzednim rozdziale)
- wiązanie za pomocą mechanizmu data binding[9].

Background, Width i Text to przykłady takich istniejących dependency property w klasach kontrolek WPF. Ich wartości mogą być ustawiane zarówno z poziomu XAML albo kodu jak zwykłe właściwości. Jeżeli obiekt nie ma zdefiniowanej dependency property wtedy używana jest wartość z elementu nadrzędnego (rodzica w drzewie logicznym). Może być „nadpisana" tj. zdefiniowana w elemencie potomnym.

Aby zaimplementować niestandardowe (custom) dependency property taka klasa podobnie jak wszystkie kontrolki w WPF musi dziedziczyć po klasie `DependencyObject`. Pozwala ona, na korzystanie z klasy `DependencyProperty`[10]. Pola klasy `DependencyProperty` muszą być widoczne w innych klasach zatem deklarowane są jako publiczne, a ponadto jako statyczne i tylko do odczytu. Stosując się do konwencji nazewniczej należy użyć nazw „zwykłych" właściwości i suffixu `DependencyProperty`[11]. Rejestracja odbywa się za pomocą statycznej metody `DependencyProperty.Register` którą najlepiej wywołać w statycznym konstruktorze. Przyjmuje ona następujące trzy parametry powiązane z „normalną" właściwością: jej nazwę, typ oraz typ klasy w którym jest zadeklarowana.

```
public class ExampleDependencyObject : DependencyObject
{
    static ExampleDependencyObject()
    {
        ReadyForBindingDependencyProperty = DependencyProperty.Register
            ("ReadyForBinding",
            typeof(string), typeof(ExampleDependencyObject));
    }

    public static readonly DependencyProperty ReadyForBindingDependencyProperty;
    public string ReadyForBinding
    {
        get { return (string)GetValue(ReadyForBindingDependencyProperty); }
        set { SetValue(ReadyForBindingDependencyProperty, value); }
    }
}
```

Na potrzeby XAML wyróżniamy jeszcze właściwość dołączoną (attached property), wyróżnia ją to że można jej wartość ustawić w innym obiekcie niż ten, który ją udostępnia. Przykładem jest właściwość `Dock` kontrolki `DockPanel`. Najłatwiej można utworzyć taką właściwość za pomocą atrybutu `[AttachedProperty]`.

---

[8] https://learn.microsoft.com/en-us/dotnet/desktop/wpf/controls/styles-templates-overview
[9] https://learn.microsoft.com/en-us/dotnet/desktop/wpf/data/how-to-bind-the-properties-of-two-controls
[10] https://docs.microsoft.com/en-us/dotnet/desktop/wpf/advanced/dependency-properties-overview
[11] W celu przyśpieszenia implementacji można wykorzystać fragment kodu wybierając w IntelliSense'ie: propdp i zatwierdzając dwukrotnie klawiszem tab.

```
<DockPanel>
    <CheckBox DockPanel.Dock="Top">Hello</CheckBox>
</DockPanel>
```

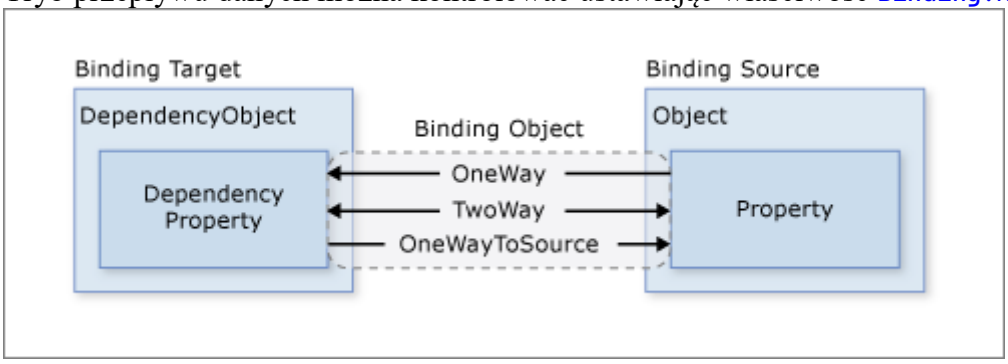
IX. Mechanizm wiązania danych (ang. Data Binding)

Proces wiązania danych polega na ustanowieniu komunikacji pomiędzy elementami GUI zaimplementowanymi w postaci Dependency Property i stanowiącymi cel, a obiektami .NET, XML, oraz obiektami elementów XAML, reprezentującymi źródło wyświetlanych danych. Domyślny obiekt będący źródłem danych jest ustawiany we właściwości DataContext każdej klasy `FrameworkElement`. Jest on dziedziczony od obiektu nadrzędnego, chyba że zostanie jawnie ustawiony.

Wyróżniamy cztery tryby przepływu danych:

- `OneWay` – od źródła do celu odpowiedni, jeśli powiązana kontrolka jest domyślnie tylko do odczytu.
- `TwoWay` – w obydwu kierunkach odpowiedni dla edytowalnych formularzy lub innych w pełni interaktywnych scenariuszy interfejsu użytkownika.
- `OneWayToSource` – od celu do źródła odpowiedni, gdy wymagane jest jedynie ponowne ustawienie wartości źródłowej z poziomu interfejsu użytkownika.
- `OneTime` – właściwość źródłowa jedynie inicjuje właściwość docelową i nie propaguje późniejszych zmian odpowiedni, dla statycznych danych oraz gdy inicjalizujemy właściwość docelową jakąś wartością z właściwości źródłowej, a kontekst danych nie jest znany z góry. Zapewnia lepszą wydajność od `OneWay` w przypadkach, gdy wartość źródłowa nie ulega zmianie.

Tryb przepływu danych można kontrolować ustawiając właściwość `Binding.Mode`.



W poniższym przykładzie możemy wyróżnić dwa wiązania danych:

```
<Window x:Class="<namespace>.MainWindow"
            ...
        DataContext="{Binding RelativeSource={RelativeSource Self}}">
    <StackPanel>
    <TextBox x:Name="InputNumberTextBox"
                Text="{Binding InputNumber, Mode=TwoWay}"/>
    <Slider Value="{Binding Text, ElementName=InputNumberTextBox}"/>
    </StackPanel>
</Window>
```

W elemencie `TextBox` występuje powiązanie dwukierunkowe pomiędzy dependency property `Text` a właściwością `InputNumber` zdefiniowanej w klasie `MainWindow`. Aby możliwe było

wiązanie z właściwościami klasy `MainWindow` musi być odpowiednio ustawiony `DataContext`. W przykładzie `RelativeSource` opisuje lokalizację źródła wiązania względem pozycji celu, gdzie `Self`[12] oznacza `Window` czyli klasę `<namespace>.MainWindow`.

```
    public partial class MainWindow : Window
    {
        public String InputNumber { set; get; } = "0";
        public MainWindow()
        {

            InitializeComponent();
        }
    }
```

Drugie wiązanie jest domyślnie (jednostronne od źródła do celu) pomiędzy dependecy property `Value` elementu `Slider` a dependency property `Text` zdefiniowaną w elemencie `TextBox` o nazwie `InputNumberTextBox`.

X. Wiązanie danych do kolekcji

Wiązanie danych do kolekcji w WPF pozwala na automatyczne odzwierciedlenie zawartości zbioru obiektów w interfejsie użytkownika (np. w kontrolkach takich jak ListBox, ComboBox, ListView, DataGrid). Dzięki temu po dodaniu, usunięciu lub zmianie elementu w kolekcji — interfejs zostaje automatycznie zaktualizowany, bez potrzeby ręcznej ingerencji.

Najczęściej wykorzystywane są dwie kolekcje:

1. `ObservableCollection<T>` – generuje zdarzenia powiadamiające UI o zmianach (dodanie, usunięcie elementu, itp.);
2. `CollectionView` / `ListCollectionView` – pozwala tworzyć widoki danych (np. sortowanie, filtrowanie, grupowanie).

Zwykła kolekcja `List<T>` nie powiadamia UI o zmianach, dlatego nie powinna być używana jako źródło danych w kontrolkach, jeśli zależy nam na dynamicznej aktualizacji.

Przykład użycia `ObservableCollection`:

Model danych:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

ViewModel z kolekcją:

```
using System.Collections.ObjectModel;
```

[12] https://learn.microsoft

```
public class MainViewModel
{
    public ObservableCollection<Person> People { get; set; }

    public MainViewModel()
    {
        People = new ObservableCollection<Person>()
        {
            new Person() { Name = "Jan", Age = 25 },
            new Person() { Name = "Anna", Age = 30 },
            new Person() { Name = "Tomasz", Age = 40 }
        };
    }
}
```

Powiązanie w XAML:

```
<Window x:Class="MyApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ObservableCollection Demo"
        Height="300" Width="300">
    <Window.DataContext>
        <local:MainViewModel/>
    </Window.DataContext>

    <Grid>
        <ListBox ItemsSource="{Binding People}" DisplayMemberPath="Name" />
    </Grid>
</Window>
```

Kontrolka ListBox wyświetli listę imion. Po dodaniu nowego elementu do kolekcji People (np. w kodzie C#), interfejs automatycznie się odświeży:

```
People.Add(new Person() { Name = "Ewa", Age = 22 });
```

Aby wyświetlić złożone dane, można wykorzystać `DataTemplate`:

```
<ListBox ItemsSource="{Binding People}">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <TextBlock Text="{Binding Name}" FontWeight="Bold"
Margin="5"/>
                <TextBlock Text=" - " />
                <TextBlock Text="{Binding Age}" />
            </StackPanel>
        </DataTemplate>
```

```
    </ListBox.ItemTemplate>
</ListBox>
```

Aby umożliwić dodawanie/usuwanie elementów z interfejsu, można powiązać zdarzenia z metodami w ViewModelu:

```
<StackPanel>
    <ListBox ItemsSource="{Binding People}" DisplayMemberPath="Name" />
    <Button Content="Dodaj osobę" Click="AddPerson_Click"/>
</StackPanel>
```

```
private void AddPerson_Click(object sender, RoutedEventArgs e)
{
    if (DataContext is MainViewModel vm)
        vm.People.Add(new Person { Name = "Nowy", Age = 20 });
}
```

Do bardziej zaawansowanego zarządzania kolekcjami można użyć widoku danych (CollectionView):

```
var view = CollectionViewSource.GetDefaultView(People);
view.SortDescriptions.Add(new SortDescription("Age",
ListSortDirection.Ascending));
view.Filter = p => ((Person)p).Age >= 25;
```

W ten sposób można dynamicznie wpływać na sposób prezentacji danych (sortowanie, filtrowanie i grupowanie), nie modyfikując samej kolekcji źródłowej.

XI. Konwertery

W celu zachowania przejrzystości właściwości kodowane są jako proste stringi a XAML posiada bogatą pulę konwerterów. Przykładowo jeśli w klasie `MainWindow` dodamy właściwość:

```
public String ColorName { set; get; } = "Yellow";
```

a w elemencie `TextBox` dodamy wiązanie do dependency property `Background`:

```
Background="{Binding Path=ColorName}"
```

Pomimo iż `ColorName` jest typu `String` a `Background` jest typu `Brush` to konwersję tę zapewnia obecność konwertera pozwalającego przekonwertować wartość `String` na `Brush`. Ponadto nic nie stoi na przeszkodzie aby dodać własny, musi on jedynie dziedziczyć po interfejsie `TypeConverter`.

Konwersji można dokonywać nawet w przypadku kiedy domyślna konwersja działa ale chcemy zmienić wartość powiązanych danych. Załóżmy, że chcemy aby `Slider` z poprzedniego

rozdziału generował wartości rosnące wykładniczo zamiast liniowo i logarytmicznie w przeciwną stronę. W takim przypadku musimy dodać konwerter

```
public class LogScaleConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
CultureInfo culture)
    {
        var stringValue = value.ToString();
        if (string.IsNullOrWhiteSpace(stringValue)) return null;

        var intValue = int.Parse(stringValue);
        return Math.Log(intValue);
    }

    public object ConvertBack(object value, Type targetType, object parameter,
CultureInfo culture)
    {
        return (int)Math.Exp((double)value);
    }
}
```

Oraz obsługę konwertera w istniejącym wiązaniu w elemencie `Slider`:

```
<Slider Value="{Binding Text,
    ElementName=InputNumberTextBox,
    Converter={StaticResource LogScaleConverterInstance}}"/>
```

Zastosowanie konwerterów może okazać się bardzo przydatne jeśli dane powinny być wyświetlane inaczej, w zależności od kultury albo kiedy wiele kontrolek lub właściwości kontrolek jest powiązanych z tymi samymi danymi ale wykorzystują je w różny sposób. W przypadku kiedy wartość docelowa jest generowana z wielu powiązanych danych źródłowych (MultiBinding[13]) można zaimplementować `IMultiValueConverter`.

XII. Dispatcher

Klasa `DependencyObject` dziedziczy po klasie `DispatcherObject` pozwala to uniknąć problemów związanych z współbieżnym dostępem do obiektu, ponieważ bezpośredni dostęp do obiektów dziedziczących po klasie `DispatcherObject` dostępny jest tylko z wątku w którym zostały utworzone. W wyniku wszystkie kontrolki użytkownika dziedzicząc po klasie `DispatcherObject` działają w jednym wątku UI[14]. Takie podejście często doprowadza do problemu jakim jest aktualizacja stanu kontrolki z innego wątku niż wątek UI w którym została ona stworzona[15]. Z pomocą przychodzi metoda `ControlDispatcher.Invoke`, przyjmująca jako argument akcję (typ `Action<T>`) w której umieszczamy kod wymagający dostępu do obiektów reprezentujących kontrolki WPF. Trafia on do kolejki zadań Dispatchera obsługiwanej przez wątek UI. W przypadku niepewności czy kod wywoływany będzie z poziomu wątku UI czy z innego możemy wykorzystać właściwość `InvokeRequired` każdej kontrolki odpowiadającą na pytanie czy musimy skorzystać z metody `Invoke` klasy

[13] https://learn.microsoft.com/en-us/dotnet/api/system.windows.data.multibinding
[14] https://cezarywalenciuk.pl/blog/programing/mvvm-wpf-i-silverlight-dependencyproperty-i-dependency-object-1
[15] http://www.pzielinski.com/?p=77

`ControlDispatcher`. Ponadto jeśli kod wywoływany jest w klasie można go uprościć korzystając z rozszerzeń, oto implementacja metody `InvokeIfRequired`:

```
public static class ControlExtensions
{
   public static void InvokeIfRequired(this Control control, Action action)
   {
       if (System.Threading.Thread.CurrentThread!=control.Dispatcher.Thread)
           control.Dispatcher.Invoke(action);
       else
           action();
   }
   public static void InvokeIfRequired<T>(this Control control, Action<T> action, T
parameter)
   {
       if (System.Threading.Thread.CurrentThread!=control.Dispatcher.Thread)
           control.Dispatcher.Invoke(action, parameter);
       else
           action(parameter);
   }
}
```

oraz wołanie w klasie:

```
this.InvokeIfRequired((value) => bar.Value = value, 10);
```

Możemy sprawdzić czy użycie Dispatcher'a jest wymagane:

```
/// <summary>
/// Invokes the specified action on the dispatcher, if necessary.
/// </summary>
/// <param name="action">The action.</param>
private void BeginInvoke(Action action)
{
    if (!this.Dispatcher.CheckAccess())
    {
        this.Dispatcher.BeginInvoke(DispatcherPriority.Loaded, action);
    }
    else
    {
        action();
    }
}
```

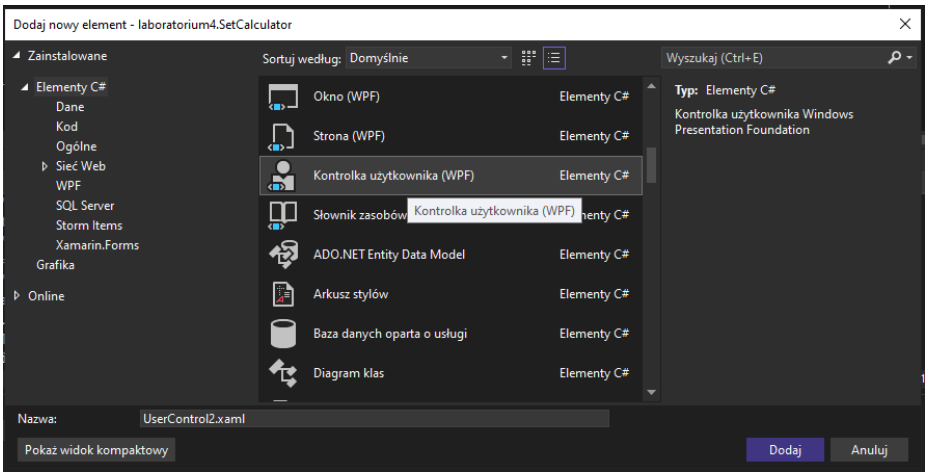
`this.BeginInvoke(this.Render);` - wołanie w klasie.

Czasami wykorzystanie metody synchronicznej w wątku UI powodującej jego zamrożenie może być pożądane, np. podczas wyświetlenia zapytania do użytkownika i w zależności od udzielonej odpowiedzi (przeważnie TAK/NIE) wykonanie warunku.

XIII. Kontrolki użytkownika WPF[16]

[16] https://learn.microsoft.com/en-us/dotnet/api/system.windows.controls.usercontrol

Najprostszym sposobem utworzenia nowej kontrolki WPF jest dziedziczenie po klasie UserControl. UserControl to ContentControl[17], co oznacza, że może zawierać pojedynczy dowolny obiekt CLR (takiego jak String, DateTime) lub obiekt UIElement (Rectangle lub Panel). Jest to dobry sposób jeśli część GUI chcemy dołączyć z innego assembly. W takim przypadku definiujemy je w nim w postaci kontrolki użytkownika. Dodanie pliku kontrolki:



Wyglądu takiej kontrolki nie można dostosować za pomocą szablonów DataTemplate i ControlTemplate oraz niemożliwa będzie obsługa różnych motywów (themes). Jeśli musimy spełnić te wymagania konieczne będzie utworzenie niestandardowej kontrolki dziedziczącej po klasie Control zamiast UserControl. Jeśli, natomiast potrzebujemy czegoś więcej niż prostej kompozycji elementów WPF to można rozbudować klasę FrameworkElement. Pozwala ona na renderowanie bezpośrednie oraz na niestandardową kompozycję elementów WPF. Każdy z kolejnych scenariuszy jest bardziej elastyczny ale wymaga od nas dużo więcej wysiłku[18].

właściwości nawigacji były wirtualne, w innym przypadku nie ma takiej potrzeby.

XIV. Okno konsoli

Okno konsoli w projekcie WPF domyślnie jest niewidoczne. Jeśli chcemy je wykorzystać należy wykorzystać funkcje z natywnych bibliotek systemowych Kernel32.dll. Z dostępnych funkcji konsoli[19] potrzebne będą :

- `AllocConsole()` – przydziela nową konsolę dla procesu wywołującego;
- `FreeConsole()` – odłącza proces wywołujący od konsoli;
- `GetConsoleWindow()` – pobiera uchwyt okna konsoli używany przez konsolę powiązaną z procesem wywołującym;
- `SetConsoleTitle(string title)` – ustawia tytuł bieżącego okna konsoli.

[17] https://learn.microsoft.com/en-us/dotnet/api/system.windows.controls.contentcontrol
[18] https://learn.microsoft.com/en-us/dotnet/desktop/wpf/controls/control-authoring-overview
[19] https://learn.microsoft.com/en-us/windows/console/console-functions

W tym celu przydatne będą funkcje z natywnych bibliotek systemowych User32.dll[20]:

- `ShowWindow(IntPtr hWnd, int nCmdShow)` – ustawia stan określonego okna konsoli. Pierwszy parametr – `hWnd` – to uchwyt okna, a drugi to typ wyliczeniowy określający jego stan, np. `SW_HIDE`. ukrywa a `SW_SHOW` aktywuje okno;
- `GetSystemMenu(IntPtr hWnd, bool bRevert)` – pobiera uchwyt do menu okna konsoli. Pierwszy parametr – `hWnd` – to uchwyt okna, a drugi to rodzaj działania:
  - FALSE – zwrot uchwytu;
  - TRUE – reset menu okna do stanu domyślnego;
- `DeleteMenu(IntPtr hMenu, int nPosition, int wFlags)` – Usuwa element z określonego menu. Pierwszy parametr – `hWnd` – to uchwyt menu okna, a drugi to element menu, który ma zostać usunięty (np. `SC_CLOSE` oznacza przycisk zamykający okno konsoli), i jest interpretowany zgodnie z trzecim parametrem – `wFlags` – który musi mieć jedną z wartości:
  - MF_BYCOMMAND – oznacza, że `nPosition` podaje identyfikator pozycji menu
  - MF_BYPOSITION– oznacza, że `nPosition` podaje względną pozycję elementu menu liczoną od zera.

Zauważmy, że zamknięcie okna konsoli powoduje zamknięcie całego procesu a tego byśmy nie chcieli. Należy zatem albo deaktywować przycisk albo zmienić jego funkcjonalność.

Musimy pamiętać o zwolnieniu niezarządzanych zasobów[21]. Można to uzyskać wykorzystując wzorzec Dispose[22]. Dobrym miejscem na wywołanie metody Dispose jest metoda wołana podczas zdarzenia zamykania okna:

```
<Window ...
        Closing="Window_Closing">
```

```
private void Window_Closing(object sender, CancelEventArgs e)
{
    _cm.Dispose();
}
```

Można tez wywołać metodę `SuppressFinalize()`[23] aby zapobiec zbędnemu wołaniu GC.

Zawsze możemy odpytać o ostatni błąd wykorzystując funkcję `GetLastError()`. W celu jego interpretacji potrzebna będzie lista kodów błędów systemowych[24].

[20] https://learn.microsoft.com/en-us/windows/win32/api/winuser
[21] https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/unmanaged
[22] https://learn.microsoft.com/en-us/dotnet/standard/design-guidelines/dispose-pattern
[23] https://learn.microsoft.com/en-us/dotnet/api/system.gc.suppressfinalize
[24] https://learn.microsoft.com/en-us/windows/win32/debug/system-error-codes