Adrian Jędrzejak, IS II

7. Tworzenie interfejsu użytkownika w WPF użyciem PRISM 5.0

Uniwersytet Mikołaja Kopernika

Wydział Fizyki, Astronomii i
Informatyki Stosowanej

patterns & practices
proven practices for predictable results

Sposoby tworzenia interfejsu:

All-in-one - marnie

- Wszystkie potrzebne kontrolki znajdują się w jednym pliku XAML.

Sekcje w kliku plikach

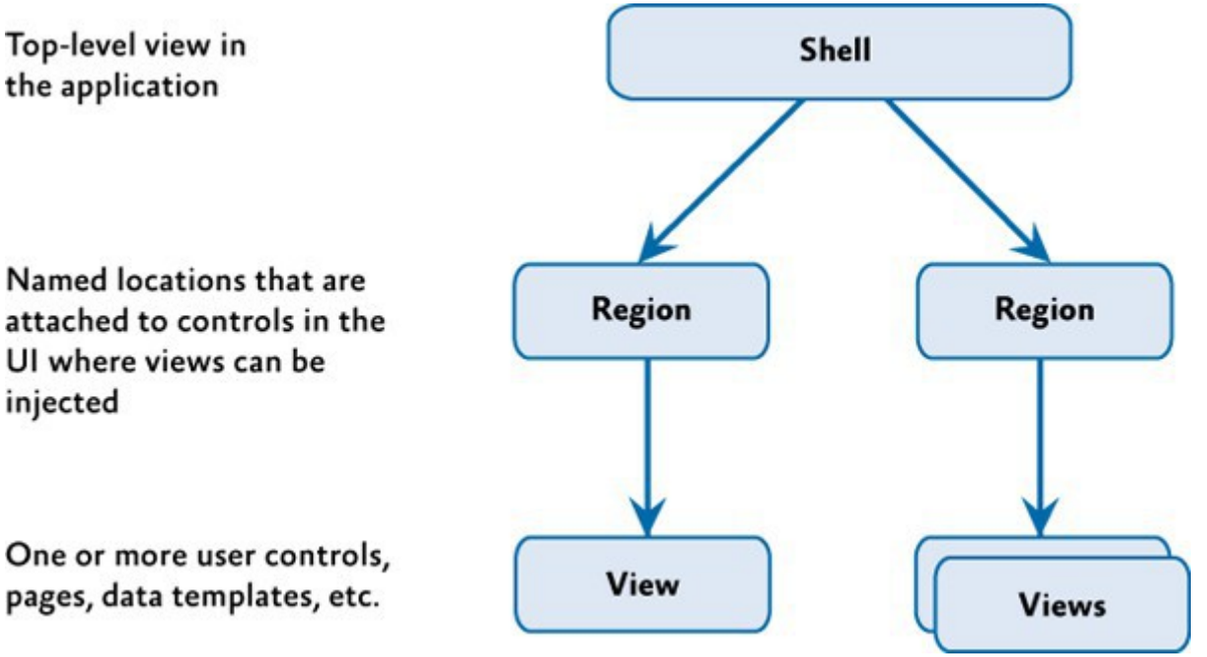
- Logiczne obszary folmularza są dzielone na części (kontrolki użytkownika). Odnośniki do nich znajdują się w głównym formularzu.

Sekcje – większa separacja

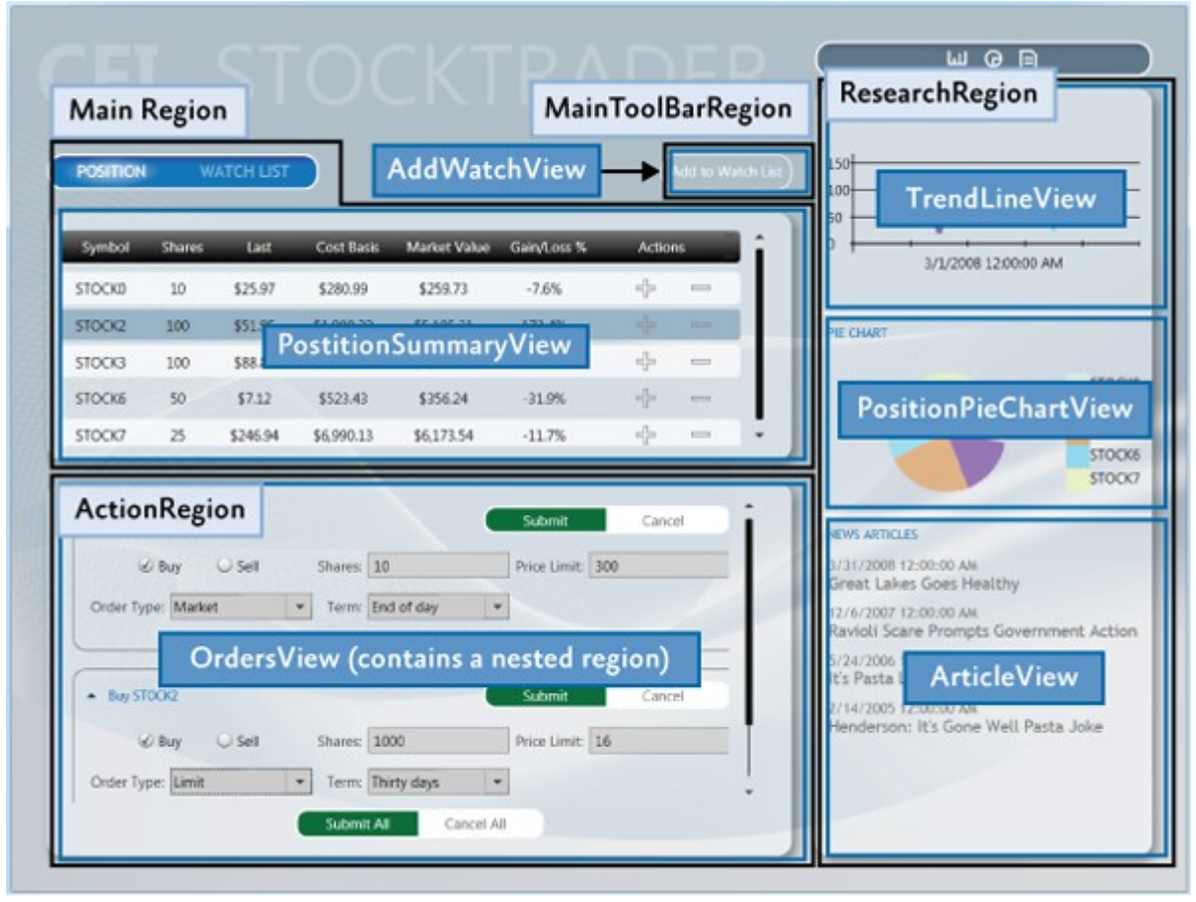
- Logiczne obszary folmularza są dzielone na części lecz definicje zawartości są **NIEZNANE** w formularzu – dynamicznie dodawanie w trakcie działania programu.
- Taki typ aplikacji to aplikacja kompozytowa używająca szablonów kompozycji interfejsu.

Kompozytowy interfejs użytkownika

- Komponenty są luźno ze sobą powiązane.
- Podział na tzw. widoki (views).
- Wyróżnione moduły.
- Potrzebna architektura pozwalająca na generowanie układu widoków i swobodną komunikację między nimi.



Przykład - Stock Trader Reference Impl.



Kompozycja kilku ładowanych dynamicznie widoków z odrębnych modułów, dodawane do regionów wyznaczonych przez powłokę (Shell).

Koncepcje układu interfejsu użytkownika

Powłoka (Shell)

- Obiekt nadrzędny (root).
- Zawiera w sobie regiony (miejsce na zawartość ładowaną dynamicznie).

Region

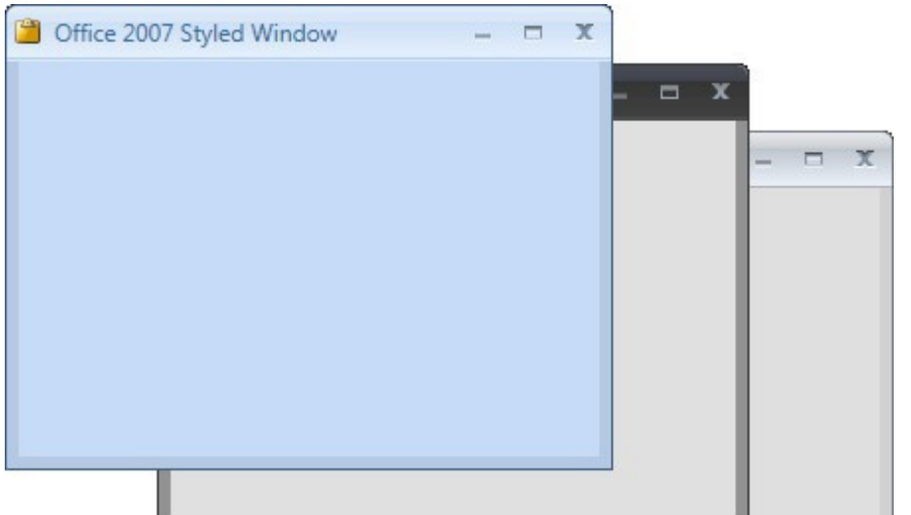
- Zarządza zawartością, ładowany dynamicznie lub na żądanie.
- Regionami mogą być typy bazowane na: **ContentControl**, **ItemsControl**, **TabControl**, lub **kontrolce niestandardowej.**

Widok (View)

- Zawartość regionu.
- Swobodny dostęp w **MEF** lub **Unity** przez nazwę lub typ widoku.

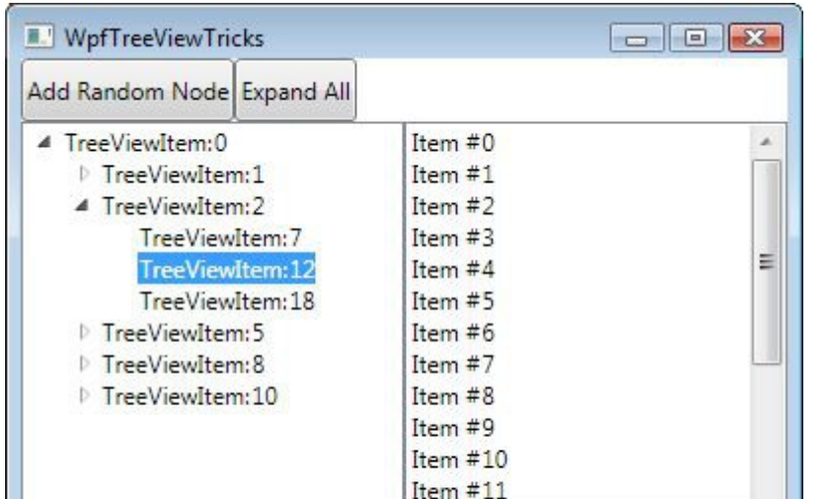
Powłoka (Shell)

- Dla **WPF** powłoka to klasa **Window**.
- Zawiera nazwane regiony.
- Definiuje ogólny wygląd aplikacji.
- Może definiować ogólne style, ramki, szablony, motywy, które będą zastosowane do załadowanych widoków.
- Projekt z powłoką nie musi posiadać referencji do projektów z widokami – mamy kilka metod dynamicznego ładowania.



Widok (View)

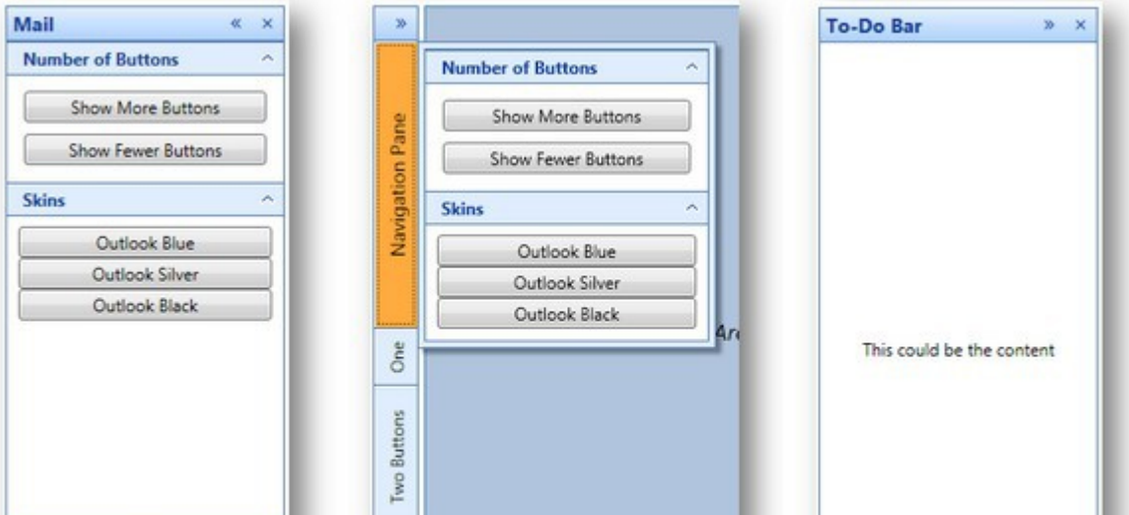
- Główny element konstrukcyjny interfejsu w aplikacji kompozytowej.
- Może być nim kontrolka użytkownika, strona(page), szablon danych i wiele innych.
- Traktujemy jako element będący oddzielonym jak to tylko możliwe od reszty aplikacji.
- **WPF** zapewnia możliwość definiowania widoku.
- **PRISM** pozwala na definiowanie regionu dla dynamicznego dodania widoku.
- Widok implementuje interfejs **IView.**



Widok kompozytowy (composite view)

Gdy widok staje się zbyt skomplikowany, zawiera wiele elementów składowych, możemy rozważyć dalszy jego podział

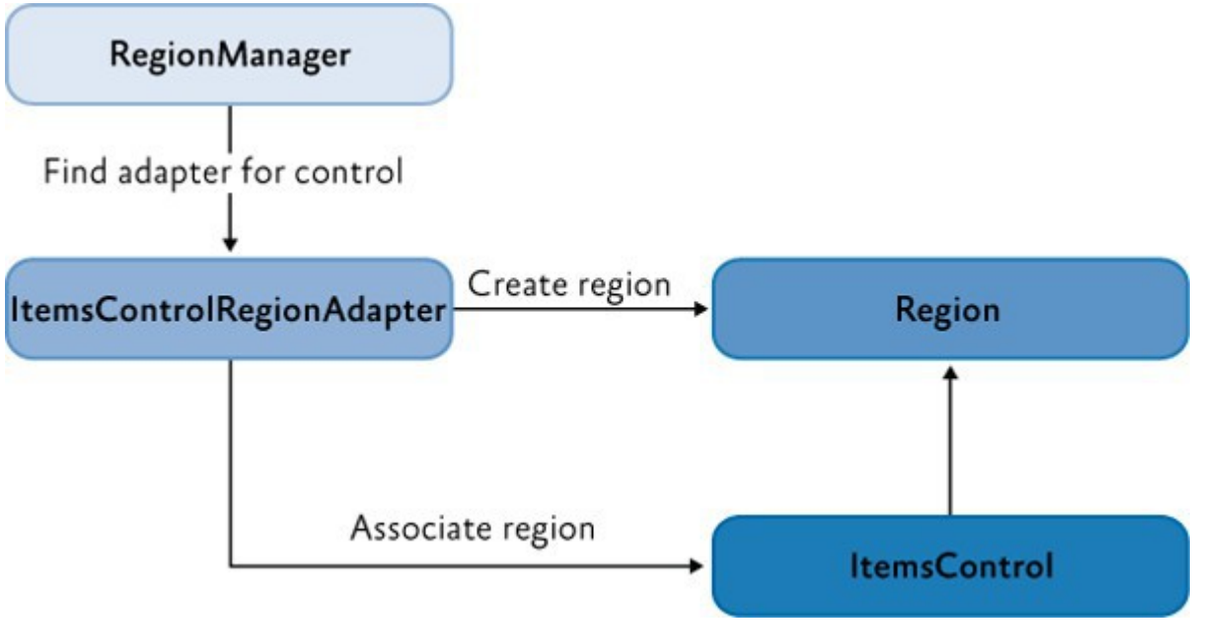
- Podział na mniejsze podrzędne widoki “dzieci”.
- Widok “rodzic” odpowiedzialny za ładowanie, współpracę między częściami oraz ich ewentualne usunięcie.
- Możliwe zastosowanie “rekurencyjnego” tworzenia widoku.



Region

Regionami zarządza klasa **RegionManager**

- Dostarcza adapter dla naszej kontrolki.
- Tworzy region dla adaptera.
- **WPF** zapewnia klasy kontenerów dla regionów.
- **PRISM** pozwala na zarządzanie regionami przez RegionManager.



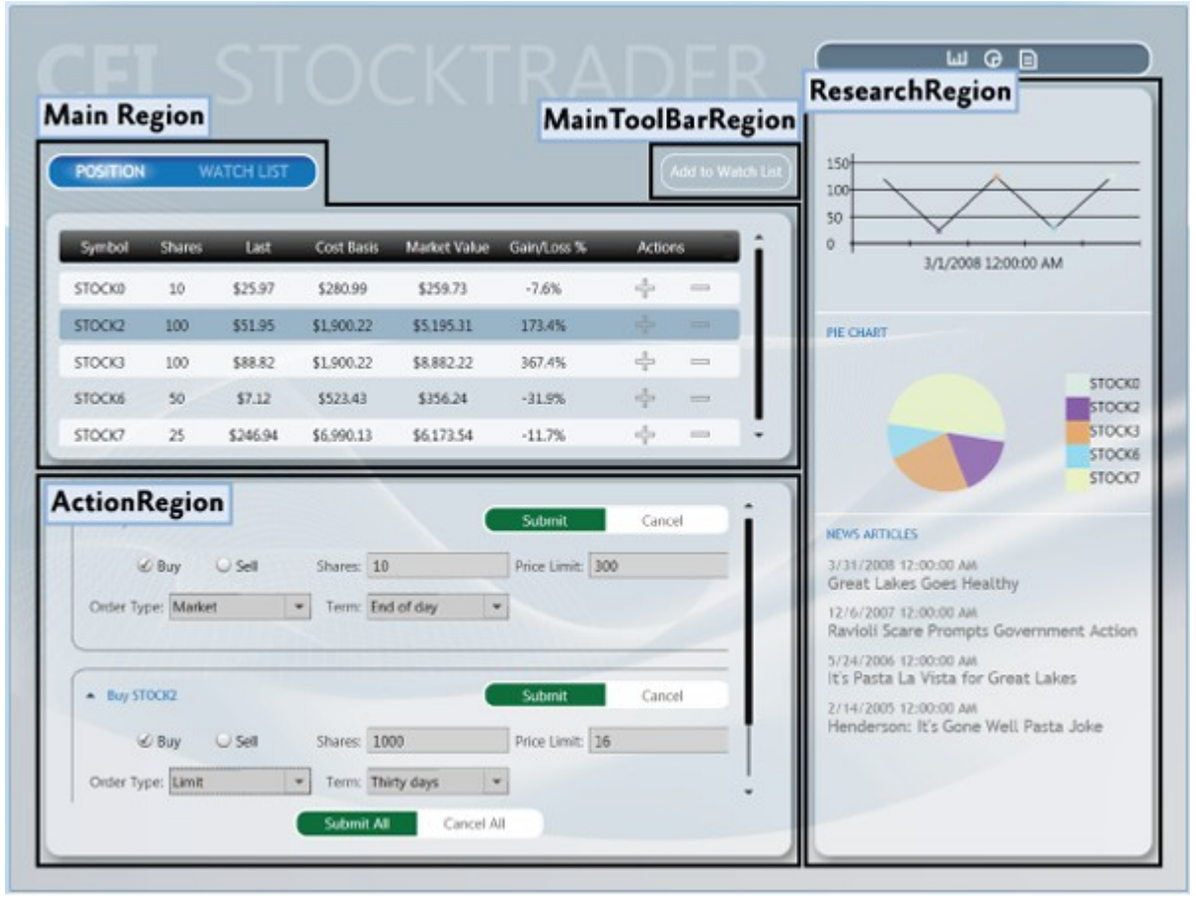
RegionManager

- Odpowiedzialny za tworzenie i zarządzanie kolekcją regionów dla kontrolek.
- Używa adaptera kontrolki i wiąże region z kontrolką.
- Można tworzyć region w kodzie lub XAML-u.
- **RegionManager.RegionName** - własność dołączona (attached property) nazywa region oraz służy przypisaniu kontrolce w jakim regionie ma się znajdować.
- Możliwość tworzenia wielu instancji typu RegionManager. Przydatne do przenoszenia kontrolki w drzewie interfejsu bez czyszczenia regionów.
- **RegionContext** – attached property, pozwala na udostępnianie wspólnych danych określonym regionom.

Implementacja regionu

- **IRegion** - interfejs, który należy zastosować implementując region.
- Możemy w nim przechowywać dowolny typ UI (kontrolka, kontrolka z szablonem danych, niestandardowa, kombinacja obu…).
- Może być pusty lub posiadać kilka elementów:
- Zależnie od ilości przechowywanych widoków bazujemy na różnych klasach:
  - **ContentControl** - wyświetla jeden obiekt widoku w regionie.
  - **ItemsControl** - wyświetla wiele widoków w regionie.

Przykład - Stock Trader Reference Impl.



Regiony z różnych modułów:

Main Region

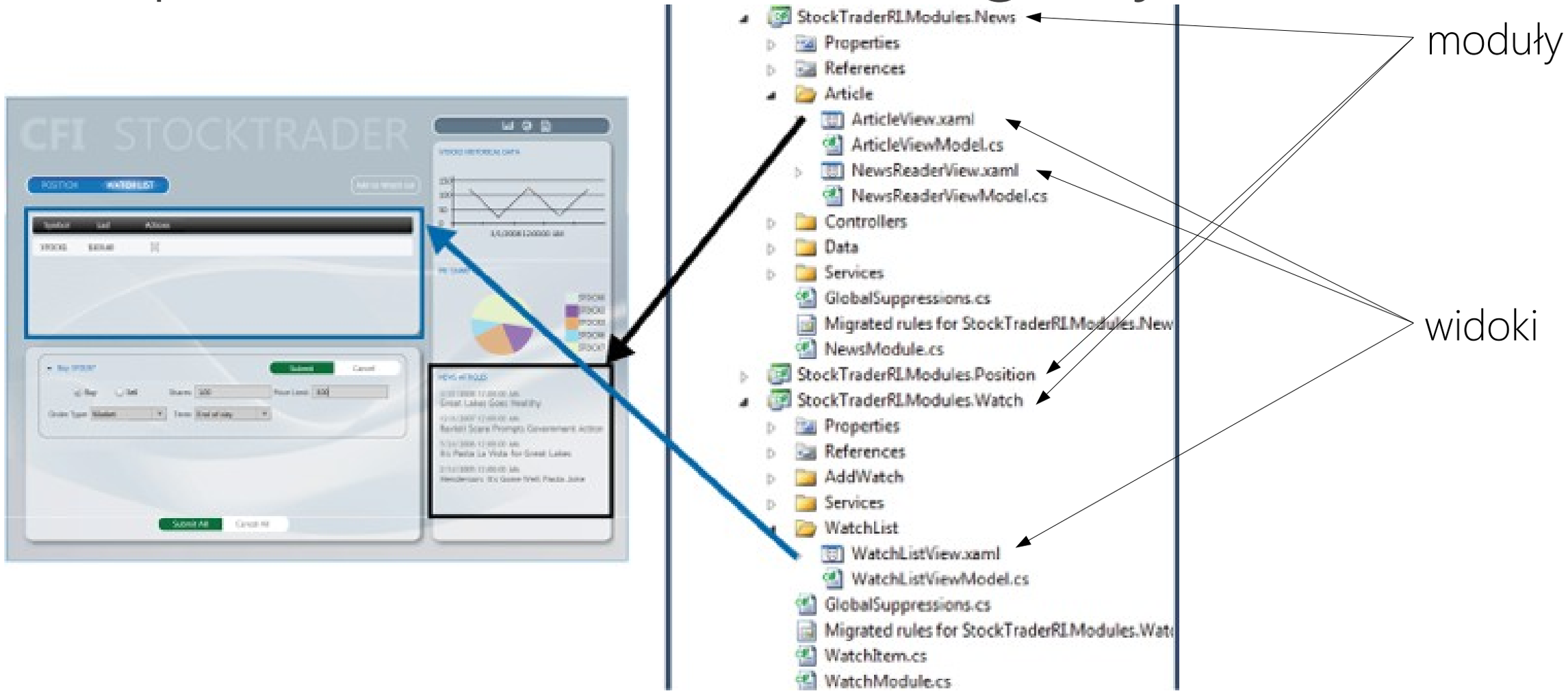
MainToolBarRegion

ActionRegion

ResearchRegion

Widoki regionów możemy zmieniać w dowolnym momencie

Mapowanie kontrolek na regiony



Schemat projektu jest ustalany w fazie projektowania

Regiony

- Wiązanie widoków z adapterami w regionie
- Rozszerzanie funkcjonalności
- Jak region lokalizuje i tworzy widoki ?

Adapter regionu

- Definiuje kontrolkę UI jako region.
- Implementuje interfejs **IRegion**.
- Każdy adapter regionu dostosowuje określony typ kontrolki UI:
  - **ContentControlRegionAdapter** - dostosowuje kontrolki typu **System.Windows.Controls.ContentControl** i klasy pochodne.
  - **SelectorRegionAdapter** - dostosowuje kontrolki dziedziczące po **System.Windows.Controls.Primitives.Selector**, takie jak kontrolka **TabControl**.
  - **ItemsControlRegionAdapter** - dostosowuje kontrolki typu **System.Windows.Controls.ItemsControl** i ich pochodne (listy, drzewa).

Komendy, Wyzwalacze UI, Akcje i Zachowania

Jeśli używamy MVVM, model widoku może bezpośrednio obsługiwać zdarzenia UI przez komendy, wyzwalacze, akcje lub zachowania.

Komenda (Command)

Separuje semantykę i obiekt wywołujący od wykonywanej logiki, pozwala na sprawdzenie możliwości uruchomienia komendy. Komendy interfejsu wiążemy z właściwościami typu **ICommand** w modelu widoku.

Komendy, Wyzwalacze UI, Akcje i Zachowania

Wyzwalacz UI (UI Trigger), Akcja (Action)

- Część przestrzeni **Microsoft.Expression.Interactivity** (Blend for Visual Studio 2013, Blend SDK).
- Dostarczają kompleksowego API do przechwytywania zdarzeń lub komend UI i kierowania ich do właściwości typu **ICommand** udostępnionych przez **DataContext**.

Zachowania regionu (Region behaviors):

- Dołączalne komponenty dodające funkcjonalność dla regionu. Wprowadzone dla wsparcia odkrywania widoków (view discovery) i kontekstu regionów oraz dla stworzenia API spójnego między **WPF** i **Silverlight**.
- Efektywny sposób na rozszerzenie implementacji regionów.

Przykłady zachowań regionów

- Gdy dołączymy klasę **AutoPopulateRegionBehavior**, tworzy ona automatycznie obiekty **ViewTypes** rejestrowane jako regiony z nazwą i monitoruje obiekt **RegionViewRegistry** w poszukiwaniu nowych rejestracji. Możemy zaimplementować własną za pomocą **IRegionViewRegistry** I **AutoPopulateRegionBehavior**.
- Zachowanie **RegionManagerRegistrationBehavior** może być odpowiedzialne za poprawną rejestrację regionów w obiekcie **RegionManager** kontrolki rodzica. Gdy kontrolka "dziecka" jest usuwana, zarejestrowany region jest usuwany.
- **RegionMemberLifetimeBehavior** jest odpowiedzialne za określenie czy element z kolekcji **ActiveViews** powinien zostać usunięty jeśli został deaktywowany. Sprawdza usunięte elementy przez interfejs **IRegionMemberLifetime**
- **SelectorItemsSourceSyncBehavior** - używany dla kontrolek pochodnych od klasy Selector. Synchronizuje aktywne widoki w regionie z elementami zaznaczonymi w selektorze.

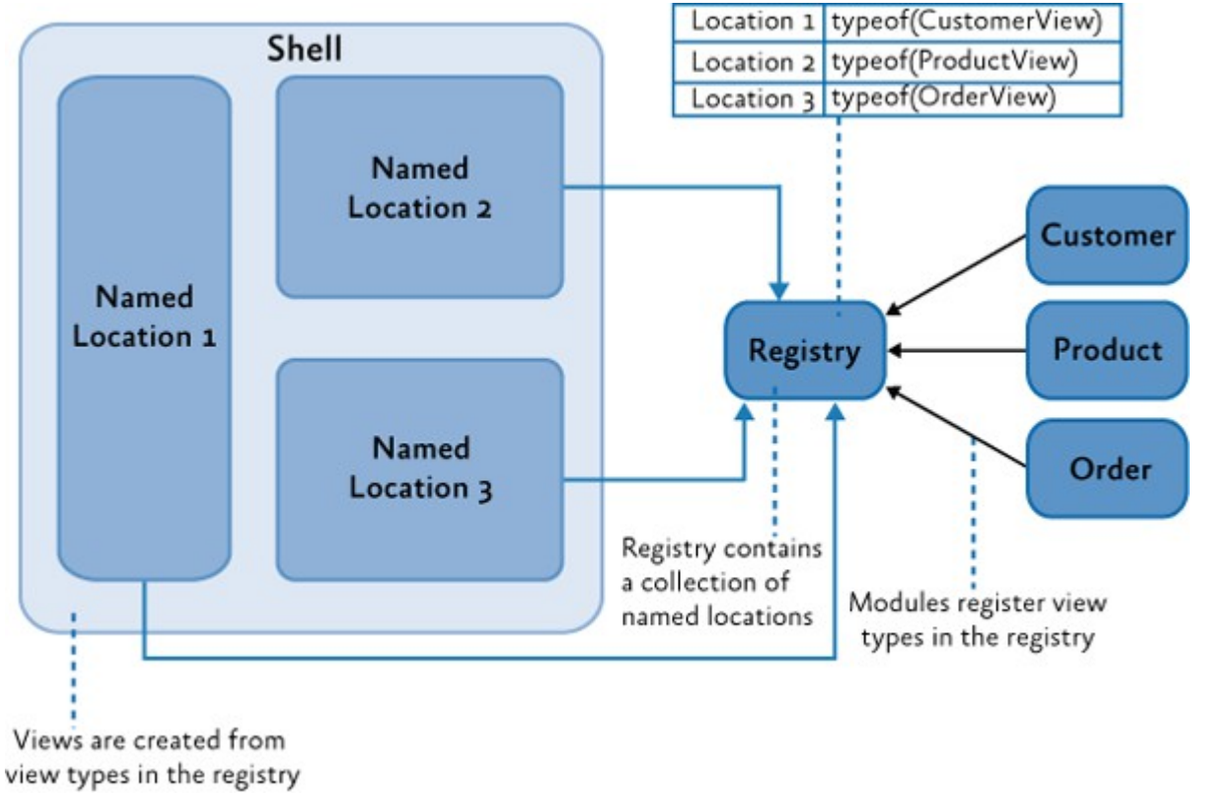
Komponowanie widoków

Widoki z różnych modułów muszą być wyświetlone w czasie działanie aplikacji w określonych lokacjach interfejsu użytkownika. Musimy określić te lokalizacje oraz sposób tworzenia i wyświetlania widoków za pomocą dostępnych metod:

- **Odkrywanie widoków** (view discovery)
- **Wstrzykiwanie widoków** (view injection) programowo

Odkrywanie widoków (view discovery)

Tworzymy powiązanie w **RegionViewRegistry** pomiędzy nazwami widoków i typem widoku. Gdy region jest tworzony, przeszukuje on obiekt **ViewTypes**, który jest do niego przypisany, a następnie tworzy instancje danych widoków. Dlatego w tym przypadku nie musimy jawnie kontrolować kiedy widoki mają zostać załadowane i prezentowane.

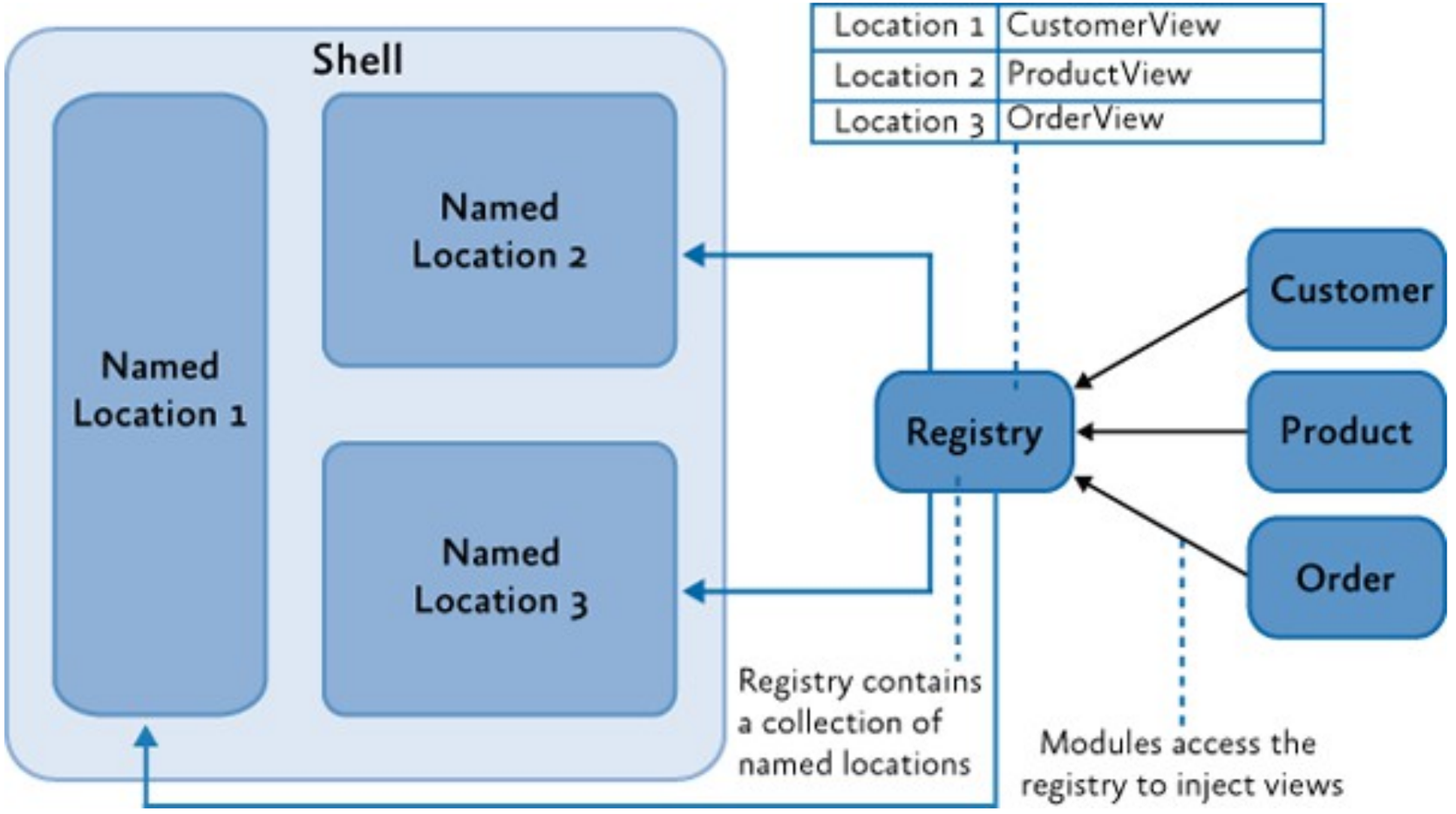


Wstrzykiwanie widoków (view injection)

Nasz kod otrzymuje referencję do regionu i musi sam dodać do niego odpowiednie już stworzone widoki (brak możliwości dodania niezainicjowanych widoków). Zwykle dochodzi to tego w trakcie inicjacji modułu lub jako wynik akcji użytkownika. Mamy również możliwość usuwania widoków z regionu.

Navigaton API w **PRISM** pozwala na uproszczenie procesu wstrzykiwania za pomocą przekazania **URI** do regionu. API samo utworzy nowy widok i doda go do regionu oraz uaktywni.

Wstrzykiwanie widoków (view injection)



Co kiedy stosować ?

Używamy odkrywania widoków gdy:

- Automatyczne ładowanie widoków jest zalecane lub wymagane.
- Pojedyncze instancje widoku będą ładowane do regionu.

Używamy wstrzykiwania widoków gdy:

- Twoja aplikacja używa Navigation API.
- Potrzebujemy jawnej kontroli nad miejscem tworzenia i wyświetlania widoku lub usunięcia widoku z regionu (rezultat logiki aplikacji lub nawigowania).
- Potrzebujemy wyświetlić wiele instancji tego samego widoku, które będą powiązane z róznymi kontekstami danych.
- Potrzebujemy kontrolować, do której instancji regionu zostanie dodany widok - na przykład chcemy dodać widok szczegółów klienta do określonego regionu klienta (potrzebna implementacja zagnieżdżonych regionów).

Scenariusze dla układów interfejsów użytkownika

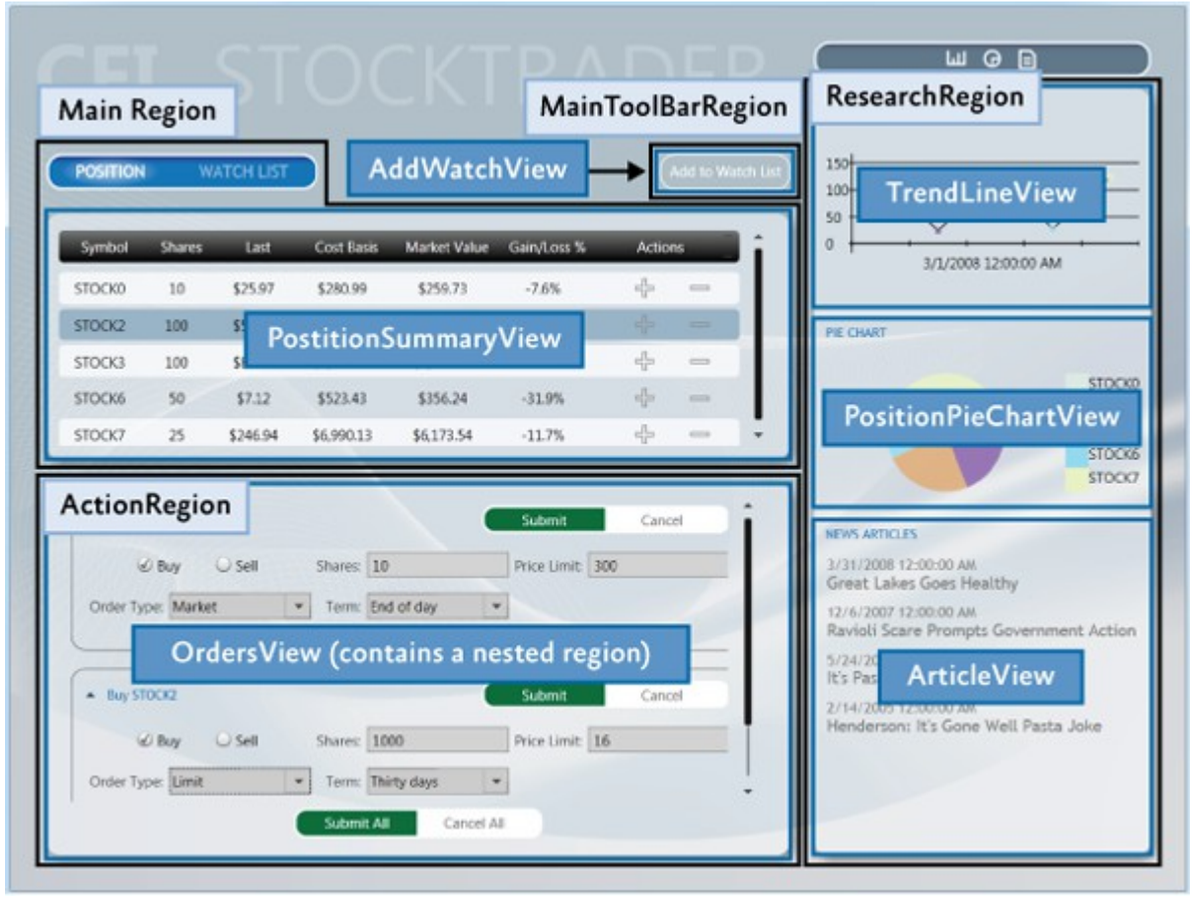
Opis bazowych scenariuszy na jakie natkniemy się przy tworzeniu aplikacji kompozytowej, na przykładzie aplikacji Stock Trader RI.

Implementacja Powłoki (Shell)

Nie potrzebujemy posiadania oddzielnej Powłoki jako części architektury aplikacji,aby używać biblioteki PRISM.

- Jeśli tworzymy zupełnie nową aplikację kompozytową, implementacja powłoki dostarcza dobrze zdefiniowany obiekt nadrzędny i szablon inicjacji interfejsu użytkownika.
- Jeśli dodajemy elementy biblioteki **PRISM** do istniejącej aplikacji, nie musimy zmieniać bazowej architektury aby dodać Powłokę. Zamiast tego możemy zacząć od definicji regionów w istniejącym interfejsie.
- Możemy mieć więcej niż jedną Powłokę w aplikacji. Jeśli jest ona zaprojektowana do otwierania więcej niż jednego okna głównego, każde takie okno działa jako Powłoka dla zawiarejących się w nim elementów.

Przykład - Stock Trader Reference Impl.



Główne okno jako Powłoka, zawiera wszystkie widoki, definiuje regiony widoków i kilka elementów UI wyższego poziomu (tytuł, WatchList tear-off banner).

Jak to wygląda pod maską ?

Kod XAML szablonu aplikacji kompozytowej

```
<!--Shell.xaml (WPF) -->
<Window x:Class="StockTraderRI.Shell"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <!--shell background -->
    <Window.Background><ImageBrush ImageSource="Resources/background.png" Stretch="UniformToFill"/></Window.Background>
    <Grid>
        <!-- logo -->
        <Canvas x:Name="Logo" >
            <TextBlock Text="CFI" /><TextBlock Text="STOCKTRADER" />
        </Canvas>

        <!-- main bar -->
        <ItemsControl x:Name="MainToolbar" prism:RegionManager.RegionName="MainToolBarRegion">
        </ItemsControl>

        <!-- content -->
        <Grid><Controls:AnimatedTabControl x:Name="PositionBuySellTab"
                prism:RegionManager.RegionName="MainRegion"/>
        </Grid>

        <!-- details -->
        <Grid><ContentControl x:Name="ActionContent" prism:RegionManager.RegionName="ActionRegion">
        </ContentControl></Grid>

        <!-- sidebar -->
        <Grid x:Name="SideGrid"><Controls:ResearchControl prism:RegionManager.RegionName="{x:Static inf:RegionNames.ResearchRegion}">
        </Controls:ResearchControl></Grid>
    </Grid>
</Window>
```

Powłoka (Shell) jest eksportowana, więc gdy bootstrapper tworzy ją, jej zależności zostaną rozwiązane przez MEF. Tutaj powłoka ma jedną zależność – ShellViewModel, która zostanie wstrzyknięta podczas tworzenia Powłoki.

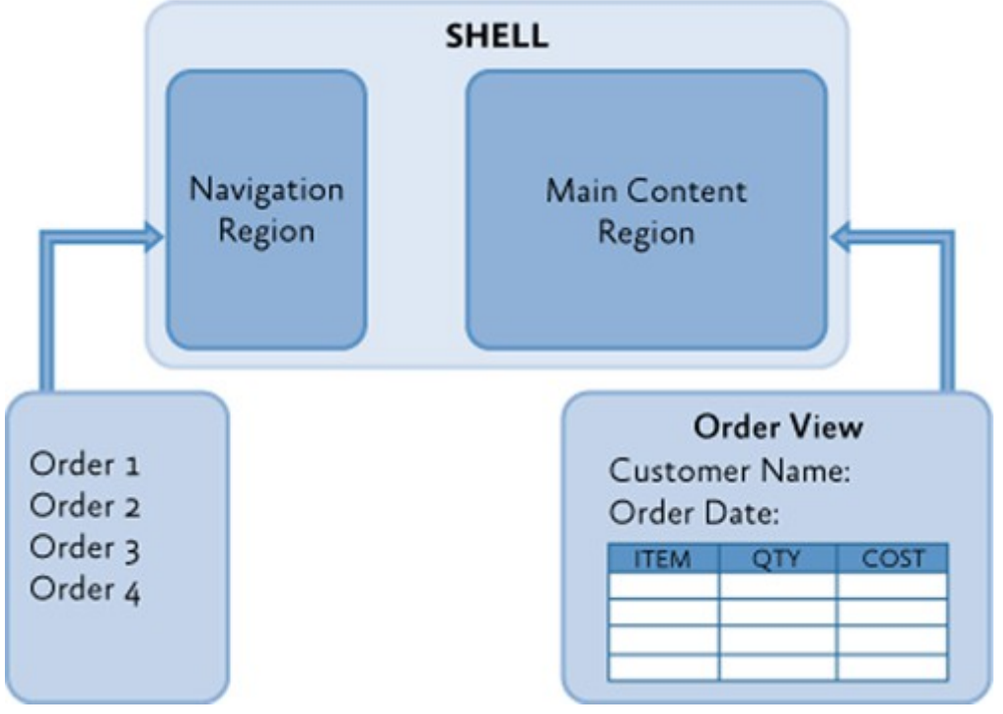
```
// Shell.xaml.cs
[Export]
public partial class Shell : Window
{
      public Shell()
      {
            InitializeComponent();
      }

      [Import]
      ShellViewModel ViewModel
      {
            set
            {
                  this.DataContext = value;
            }
      }
}
```

```
// ShellViewModel.cs
[Export]
public class ShellViewModel : BindableBase
{
// Tutaj powinna znajdować się cała logika dla Powłoki
}
```

Definiowanie Regionu

Regiony definiujemy przez przypisanie nazwy regionu do kontrolki WPF w kodzie XAML lub C#. Jest on wtedy dostępny dla nas przez tą właśnie nazwę. Możemy np. wydzielić region dla przycisków nawigacji oraz zawartości.

```
prism:RegionManager.RegionName="MainContentRegion”
```



Implementacja Regionu - IRegion

Region jest klasą implementującą interfejs **IRegion** pokazany niżej:

```
public interface IRegion : INavigateAsync,
INotifyPropertyChanged
{
IViewsCollection ActiveViews { get; }
IRegionBehaviorCollection Behaviors { get; }
object Context { get; set; }
string Name { get; set; }
IRegionNavigationService NavigationService {
get; set; }
IRegionManager RegionManager { get; set; }
.
.
.
```

```
.
Comparison<object> SortComparison { get;
set; }
IViewsCollection Views { get; }
void Activate( object view );
IRegionManager Add( object view );
IRegionManager Add( object view, string
viewName );
IRegionManager Add( object view, string
viewName, bool createRegionManagerScope);
void Deactivate( object view );
object GetView( string viewName );
void Remove( object view );
}
```

Wytyczne projektowania UI

W aplikacjach kompozytowych, zawartość paneli jest dynamiczna więc nieznana podczas projektowania. To zmusza nas do tworzenia struktur kontenerów oddzielnie oraz zaprojektowania elementów, które mogą się tam znaleźć osobno.

Jako projektant lub grafik musimy więc myśleć o kompozycji kontenerów i kompozycji regionów w kontenerach.

Układ kontenerów

Układ kontenerów jest rozszerzeniem modelu ramek, który dostarcza WPF . Termin **kontener** może oznaczać każdy element, włączając w to okno, stronę, kontrolkę użytkownika, panel, kontrolkę niestardardową, kontrolkę szablonu, szablon danych, które mogą zawierać w sobie elementy.

Tworząc wiele projektów interfejsów, pewne motywy zaczynają się powtarzać:

Okno > Strona > Kontrolki użytkownika zawierające stałą lub dynamiczną zawartość.

Rozpatrujemy następujące kwestie

Gdy tworzymy układ dla kontenerów aplikacj:

- Czy masz jakiekolwiek limity rozmiaru, które ograniczą jak duża może być zawartość ? Jeśli tak, rozważ użycie kontenerów z paskiem przewijania.
- Przemyśl użycie kombinacji Expander-ScrollViewer w sytuacji, gdy duża ilość dynamicznej zawartości musi zmieścić się w ograniczonej przestrzeni.
- Przeanalizuj, jak zawartość powiększa się wraz ze wzrostem dostępnej powierzchni prezentacji, aby zapewnić dobry odbiór treści w każdej rozdzielczości

Podgląd aplikacji kompozytowej w fazie projektowania

Każdy element interfejsu z racji dynamicznego ładowania musimy zaprojektować poza projektem kontenerów. To utrudnia przewidzenie jak okno aplikacji będzie wyglądało w trakcie działania.

W tym celu możemy utworzyć projekt testowy z projektem kontenerów oraz wszystkimi elementami interfejsu do sprawdzenia.

Możemy również użyć próbek danych w programie Blend lub Visual Studio (design-time sample data). Opcja pomocna przy pracy z tabelami, grafami czy kontrolkami typu lista.

Układ aplikacji

Rozpatrz poniższe kwestie gdy projektujesz układ aplikacji kompozytowej:

- Regiony głównej powłoki powinny być puste. Nie umieszczamy tam zawartości, ponieważ zostanie ona załadowana w trakcie działania.
- Powłoka powinna zawierać tło, tytuły oraz stopkę. Pomyśl o powłoce jak o głównej stronie witryn internetowej .
- Kontrolki kontenerów, działające jako regiony są oddzielone od widoków które przechowują. Dlatego powinieneś mieć możliwość zmiany rozmiaru widoków bez modyfikacji jej kontenera oraz zmiany rozmiaru kontenera bez zmiany widoków. Należy przemyśleć następujące punkty, definiując rozmiar widoku:

  …

Układ aplikacji – rozmiary widoków

- Jeśli widok będzie używany w wielu regionach lub jeśli nie można określić gdzie będzie się znajdował, zaprojektuj go ze zmienną szerokością lub wysokością.
- Jeśli widoki mają stałe rozmiary, regiony powłoki powinny używać dynamicznej zmiany rozmiaru i odwrotnie, ponieważ nie wiemy ile elementów znajdzie się w regionie.
- Widoki mogą wymagać stałej wysokości oraz zmiennej szerokości (przykładem widok **PositionPieChart** w panelu bocznym **Stock Trader RI**)
- Inne widoki mogą wymagać zmiennej wysokości I szerokości. Na przykład widok **NewsReader** w panelu bocznym STRI. Wysokość zależy od długości tytułu, a szerokość powinna dostosować się do rozmiaru regionu (szerokości panelu). Podobna rzecz ma zastosowanie do widoku **PositionSummaryView**, gdzie szerokość siatki powinna adaptować się do rozmiaru ekranu, a wysokość do liczby wierszy siatki.

Układ aplikacji, cd.

- Widoki powinny zwykle posiadać przezroczyste tło, pozwalając powłoce na zapewnienie takowego.
- Zawsze używaj wartości nazwanych do przypisania kolorów, pędzli, czcionek i ich rozmiarów zamiast podawania bezpośredniej wartości własności w XAML. Pozwala to na łatwiejszą konserwację aplikacji a także reagowanie na zmiany tych wartości w słownikach zasobów w trakcie działania aplikacji.

Optymalizacje w fazie run-time i design-time

Dla działającej aplikacji:

- Umieść wspólne zasoby w pliku **App.xaml** lub wspólnym katalogu w celu uniknięcia duplikacji styli.

Dla aplikacji w fazie projektowania:

- **Duże projekty z wieloma plikami XAML –** czas ładowania visual designera może zostać spowolniony przez dużą ilość pliku XAML do parsowania. Przeniesienie plików XAML do oddzielnego projektu, kompilacja i dodanie referencji do projektu oryginalnego unika fazy parsowania i poprawia działanie visual designera. Przy eksporcie rozważny udostępnienie kluczy komponentów zasobu (**ComponentResourceKeys**).

Optymalizacje w fazie design-time

Dla aplikacji w fazie projektowania:

- **XAML Assets –** rozważmy stworzenie zestawu assetów w pliku XAML zamiast dołączać je jako graficzne pliki .png, .jpeg, .ico . Zapewnia to uniezależnienie się od rozdzielczości ekranu. Jeśli posiadamy dużą ilość tego typu elementów, przeniesienie ich do oddzielnego projektu bilbioteki **.dll** poprawia wydajność.

Skutkiem ubocznym przeniesienia zawartości XAML oraz assetów do bibliotek **.dll** jest brak ich listowego wyświetlenia przez **Blend 2013** lub **VS 2013**. Nie będzie możliwości korzystania z graficznych narzędzi do wybierania elementów. Zamiast tego będziemy musieli podawać nazwę potrzebnego zasobu !

8. Nawigacja w WPF z użyciem PRISM 5.0

Nawigacja

Proces przechodzenia pomiędzy różnymi układami widoków w regionie.

Dodatkowe funkcjonalności:

- Możliwość nawigowania do poprzedniego stanu,
- Potwierdzenie procesu nawigacji przez użytkownika

PRISM wspiera dwa modele nawigacji i udostępnia kilka interfejsów dla usystematyzowania procesu.

Nawigacja bazująca na stanach (state-base)

Widok prezentowany w interfejsie jest odświeżany przez zmiany stanów w modelu widoku lub przez interakcję użytkownika w samym widoku. Zamiast zastępowania się widoków, zmieniamy ich stan.

Ten styl nawigacji pasuje do poniższych sytuacji:

- Widok potrzebuje wyświetlić te same dane lub funkcje w różnych formach.
- Widok ma zmienić swój układ lub styl bazując na stanach modelu widoku.
- Widok potrzebuje zainicjować ograniczoną blokującą lub nieblokującą reszty interakcję z użytkownikiem w kontekście danego widoku.

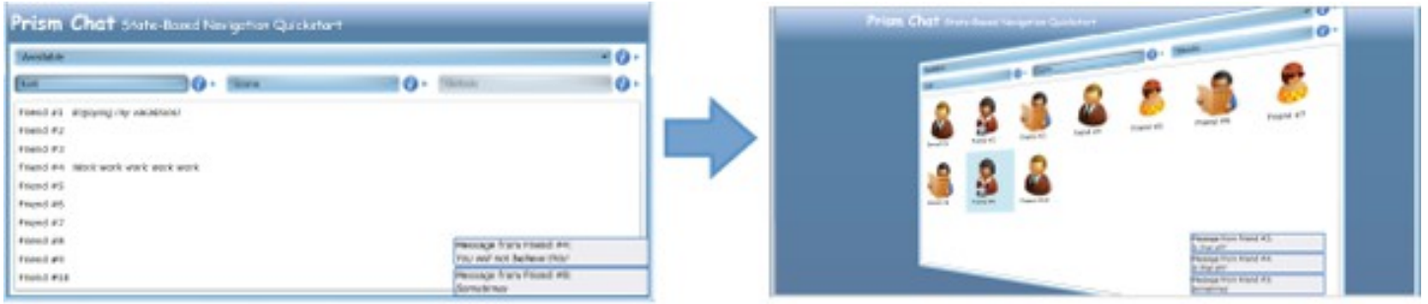
Nawigacja bazująca na stanach (state-base)

Ten styl nawigacji nie nadaje się w przypadku:

- Interfejs musi przestawić zupełnie różne dane użytkownikowi lub użytkownik musi przeprowadzić inne zadanie. Lepiej wtedy zaimplementować oddzielne widoki reprezentujące dane oraz zadanie, przełączanie opierając na widokach (view-based).
- Ilość zmian stanów potrzebna do implementacji jest zbyt skomplikowana ze względu na wielkość i trudność zmiany logiki i struktury stanu (lepiej view-based navigation).

Wyświetlanie danych w różnych formach

- Zmiana reprezentacji danych nie wymaga działania modelu widoku. Możemy określić “nawigację” (transformację) w XAML-u.
- **DataStateBehavior** programu Blend dobrze się do tego nadaje.




```
<ei:DataStateBehavior Binding="{Binding IsChecked,
ElementName=ShowAsListButton}"
                      Value="True"
                      TrueState="ShowAsList"
                      FalseState="ShowAsIcons"/>
```

Stany wizualne

Odzwierciedlanie stanów aplikacji

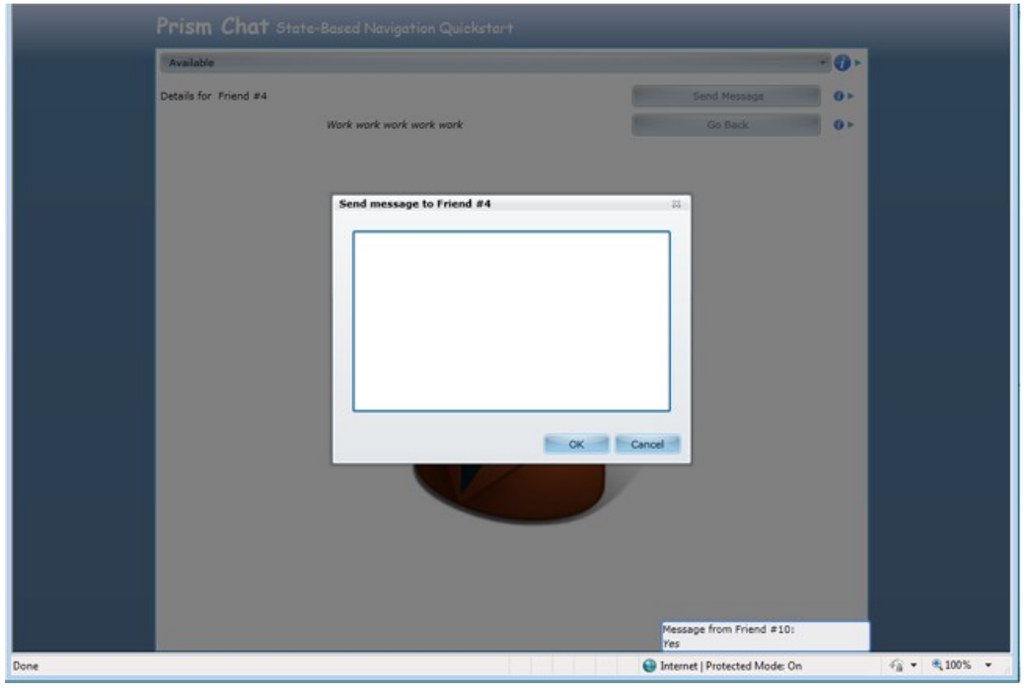
- Symulacja nawigacji do stanu “niedostępny” przez zmianę stanu wizualnego
- Powiadomienie o zmianie stanu w modelu widoku (np. isOffline) samego widoku może skutkować dużą zmianą wizualną symulującą nawigację

```
<ei:DataStateBehavior Binding="{Binding ConnectionStatus}"
                                  Value="Available"
                                  TrueState="Available" FalseState="Unavailable"/>
```

Interakcja z użytkownikiem

- Prośba o dane przez wyświetlenie okna popup
- Gdy nawigacja do nowego okna odbywa się w ramach tego samego kontekstu danych, możemy ją zaimplementować jako nawigacja oparta o zmianę stanu
- Implementacja komendy SendMessage

```
<prism:InteractionRequestTrigger
SourceObject="{Binding SendMessageRequest}">
    <prism:PopupWindowAction IsModal="True">
        <prism:PopupWindowAction.WindowContent>
            <vs:SendMessagePopupView />
        </prism:PopupWindowAction.WindowContent>
    </prism:PopupWindowAction>
</prism:InteractionRequestTrigger>
```



Nawigacja bazująca na widokach (view-base)

Częściej będziemy zamieniać widoki między sobą, a więc zastosujemy nawigację opartą o widoki (view-base).

Ten typ nawigacji może być dość złożony i wymagać od nas kontroli za pomocą logiki.

Istnieje kilka spraw do rozpatrzenia przy implementacji tą metodą:

View-based navigation - Cel

Nasz cel nawigacji (kontener lub kontrolka wielu widoków) może obsługiwać ją w inny sposób lub inaczej prezentować wizualnie.

Cel może być prostą ramką (**Frame**), kontrolką zawartości (**ContentControl**) – wtedy nawigowany widok będzie normalnie w nich wyświetlony.

Cel może być też typu **TabControl**, **ListBox**, itp. Nawigacja może wymagać aktywacji, wybrania istniejącego lub dodania nowego widoku w specyficzny sposób.

View-based navigation - Identyfikacja

Musimy określić sposób w jaki będziemy wskazywać kolejny widok.

Identyfikacja celu do którego się odnosimy – **URI**, **nazwa typu**, **lokalizacja zasobu** i wiele innych metod do dyspozycji.

Widoki w aplikacji kompozytowej będą często znajdować sie w oddzielnych modułach. Potrzeba więc identyfikacji niewymagającej ścisłych powiązań czy występowania zależności między nimi.

View-based navigation – inne własności

Zapewnienie dostępu do tego samego kontekstu danych możemy zapewnić z wykorzystaniem np. **MEF** czy **Unity Framework**.

Musimy mieć możliwość czystego odseparowania zachowania nawigacyjnego poprzez widok i model widoku.

Możliwość przechodzenia w przód lub tył między widokami. Należy rozważyć historię lub dziennik dla takiej funkcjonalności.

Podstawowa nawigacja w regionie

- Nowy widok zostanie wyświetlony w tym samym regionie.
- Wykorzystanie interfejsu **INavigateAsync** – metody **RequestNavigate**.
- Klasa **Region** implementuje **INavigateAsync**.

```
IRegion mainRegion = ...;
mainRegion.RequestNavigate(new Uri("InboxView", UriKind.Relative));
```

- Klasa **RegionManager** również pozwala rozpocząć nawigację.

```
IRegionManager regionManager = ...;
regionManager.RequestNavigate("MainRegion",
                              new Uri("InboxView", UriKind.Relative));
```

- **NavigationResult** dostarcza informacji o powodzeniu operacji.

Modele i Modele widoków podczas nawigacji

- Czasem widoki muszą zmienić swój stan podczas nawigacji.
- Interfejs **INavigationAware** to umożliwia.

```
public interface INavigationAware
{
    bool IsNavigationTarget( NavigationContext navigationContext );
    void OnNavigatedFrom( NavigationContext navigationContext );
    void OnNavigatedTo( NavigationContext navigationContext );
}
```

- **OnNavigatedFrom** – wywołany przed nawigacją, pozwala zmienianemu widokowi zapisać stan, usunąć się z regionu, itp.
- **OnNavigatedTo –** wywołany po zakończeniu nawigacji. Pozwala nowo wyświetlonemu widokowi zainicjować swój stan.
- **IsNavigationTarget** – wykorzystywany w późniejszym przypadku.

Przekazywanie parametrów podczas nawigacji

- Przesłanie parametrów przy wywołaniu RequestNavigate()

```
Employee employee = Employees.CurrentItem as Employee;
if (employee != null)
{
    var parameters = new NavigationParameters();
    parameters.Add("ID", employee.Id);
    parameters.Add("myObjectParameter", new ObjectParameter());
    regionManager.RequestNavigate(RegionNames.TabRegion,
        new Uri("EmployeeDetailsView", UriKind.Relative), parameters);
}
```

- Odebranie parametrów z kontekstu nawigacji

```
public void OnNavigatedTo( NavigationContext navigationContext )
{
    string id = navigationContext.Parameters["ID"];
    ObjectParameter myParameter = navigationContext.Parameters["myObjectParameter"];
}
```

Nawigowanie do istniejących widoków

Często właściwsze dla widoków w aplikacji jest ich ponowne użycie, odświeżenie lub aktywacja zamiast zamiana przez nową instancję.

Chcemy np. pokazać widok, który jest już załadowany, ale z innymi danymi.

Prism wspiera takie scenariusze przez metodę **IsNavigationTarget** interfejsu **INavigationAware**.

Implementacja metody **IsNavigationTarget** może użyć kontekstu nawigacji do określenia czy widok może obsłużyć żądanie. Jeśli zwróci **true**, niezależnie od parametrów nawigacji, instancja widoku zostanie użyta ponownie.

Pozwala to na stworzenie reużywalnych obiektów.

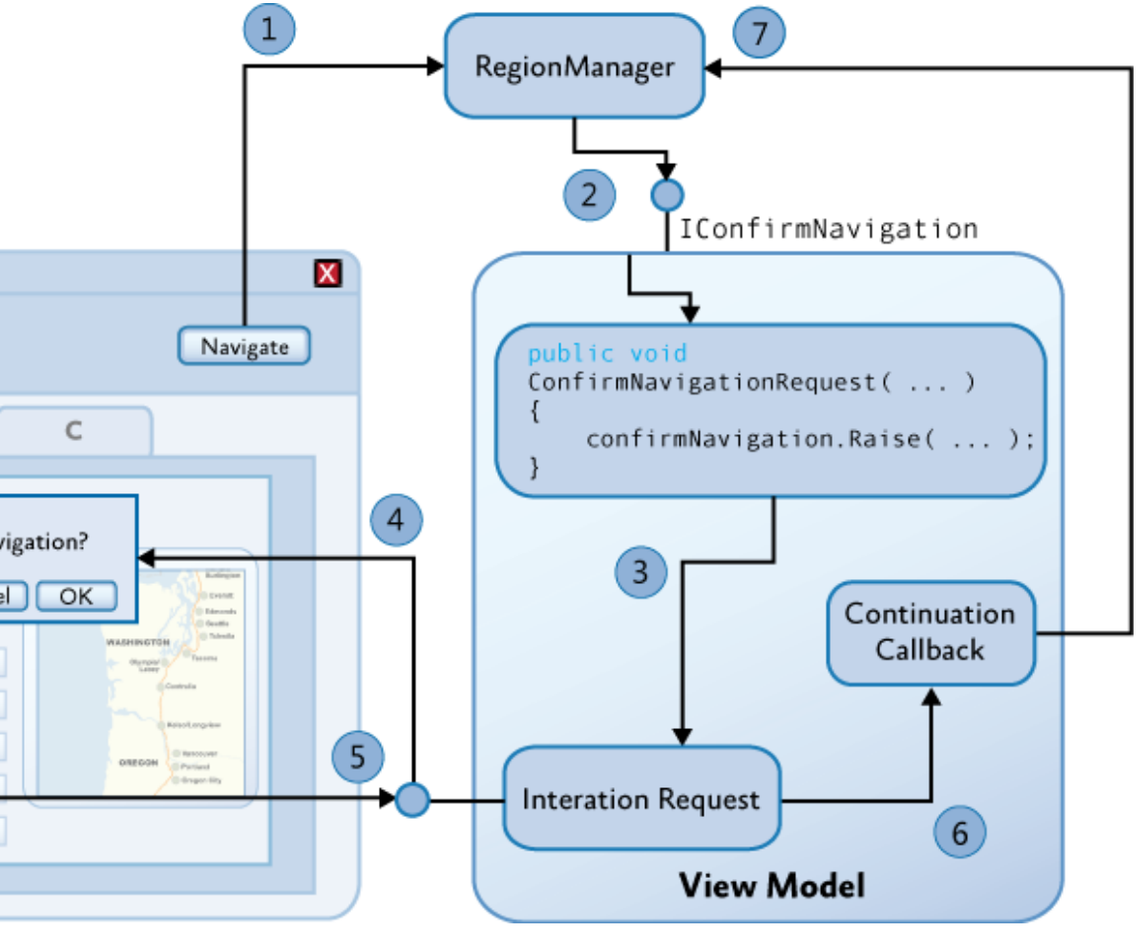
Potwierdzanie lub przerywanie nawigacji

Użytkownik może próbować nawigować w trakcie edytowania danych. Powinniśmy potwierdzić taką operację i powiadomić o możliwej utracie danych.

Interfejs **IConfirmNavigationRequest** dziedziczący po **INavigationAware** dodaje metodę **ConfirmNavigationRequest.** Implementacja jej pozwala na uczestniczenie modelu lub modelu widoku w wyświetleniu monitu.

Możemy wtedy kazać wyświetlić okno z pytaniem i możliwością interakcji w celu potwierdzenia lub przerwania działania procesu nawigowania.

Potwierdzanie lub przerywanie nawigacji



1. Inicjacja metodą **RequestNavigate**
2. Jeśli widok implementuje **IConfirmNavigation**, wywołaj **ConfirmNavigationRequest**.
3. Model widoku uruchamia zdarzenie żądania interakcji.
4. Widok wyświetla okno pop-up i czeka na reakcję.
5. Wywołanie zwrotne jest wywołane po zamknięciu okna.
6. Metoda kontynuująca jest wywołana aby zwrócić informację o decyzji użytkownika.
7. Operacja nawigacji jest zakończona.

Dziennik Nawigacji (Navigation Journal)

Serwis nawigacji w **NavigationContext** implementuje interfejs **IRegionNavigationService** z własnością **Journal.**

Własność **Journal** zapewnia dostęp do dziennika nawigacji powiązanego z regionem. Dziennik ten implementuje poniższy interfejs **IRegionNavigationJournal:**

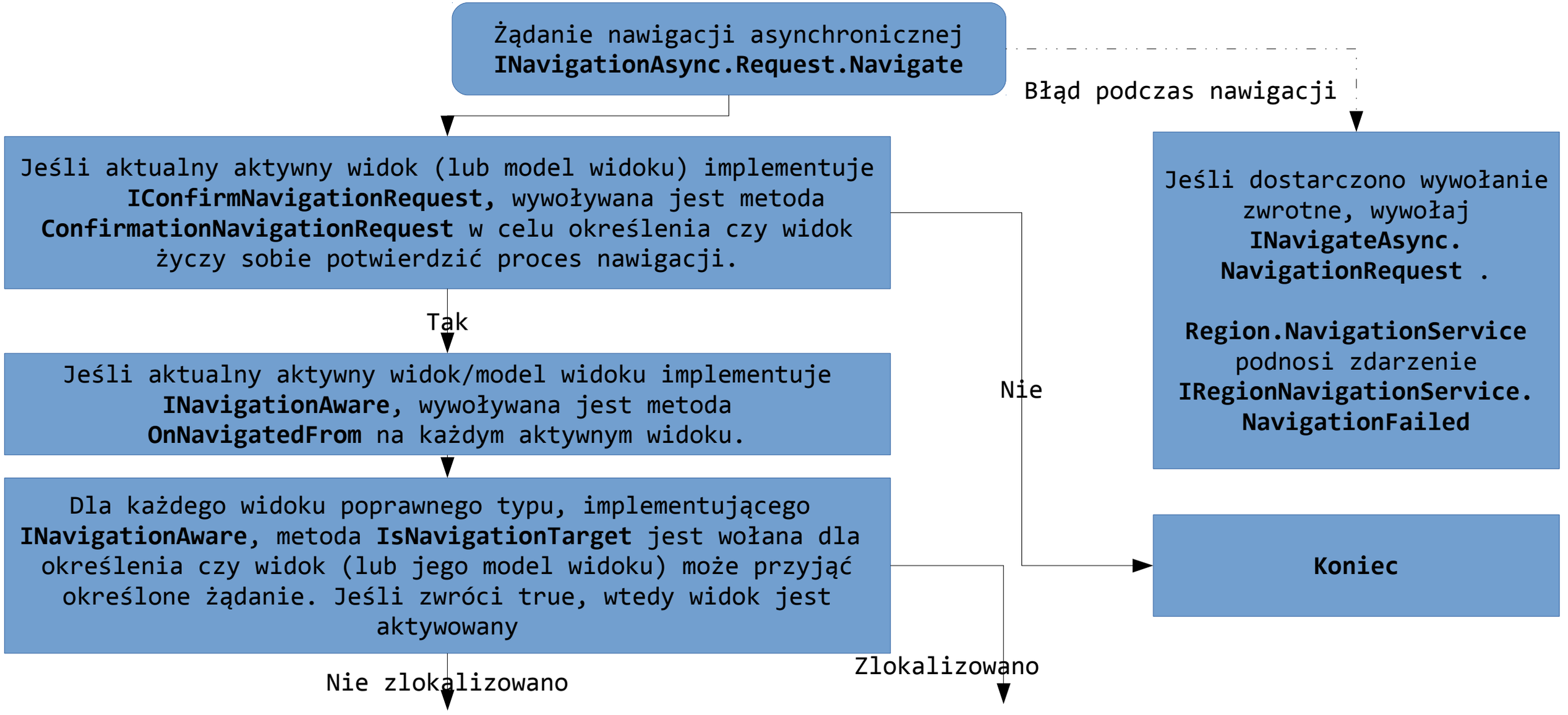
```
public interface IRegionNavigationJournal
{
    bool CanGoBack { get; }
    bool CanGoForward { get; }
    IRegionNavigationJournalEntry CurrentEntry { get; }
    INavigateAsync NavigationTarget { get; set; }
    void Clear();
    void GoBack();
    void GoForward();
    void RecordNavigation( IRegionNavigationJournalEntry entry );
}
```

Pozwala użytkownikowi nawigować z wnętrza widoku. Przykłady: cofanie z obecnego do poprzedniego widoku za pomocą **GoBack()** lub implementacja aplikacji w formie kreatora z użyciem metody **GoForward()** .

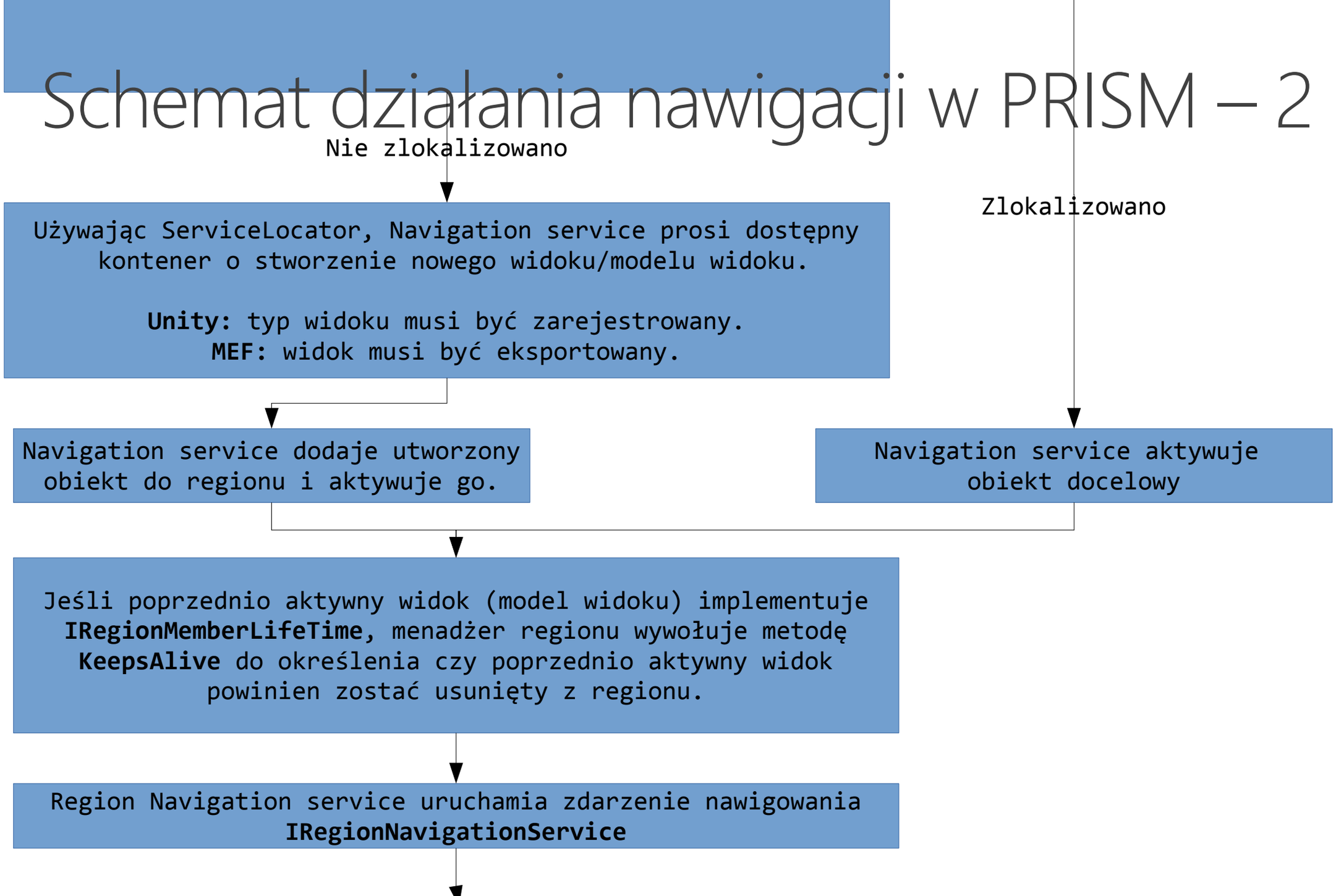
Użycie WPF Navigation Framework

- Przy użyciu z regionami pozwala na zmianę wyglądu aplikacji bez wpływu na strukturę nawigacji. Wspiera nawigację pseudosynchroniczną.
- Nawigacja w regionach została zaprojektowana do użycia razem z **WPF Navigation Framework**, a nie w celu jego zastąpienia.
- Oparcie na kontrolce **Frame** dające funkcjonalność dziennika nawigacji.
- Trudne wykorzystanie przy wsparciu MVVM oraz jednoczesnym wstrzykiwaniu zależności. Można wtedy korzystać razem z PRISM jednak łatwiejsze może być oparcie wszystkiego na nawigacji w regionach.

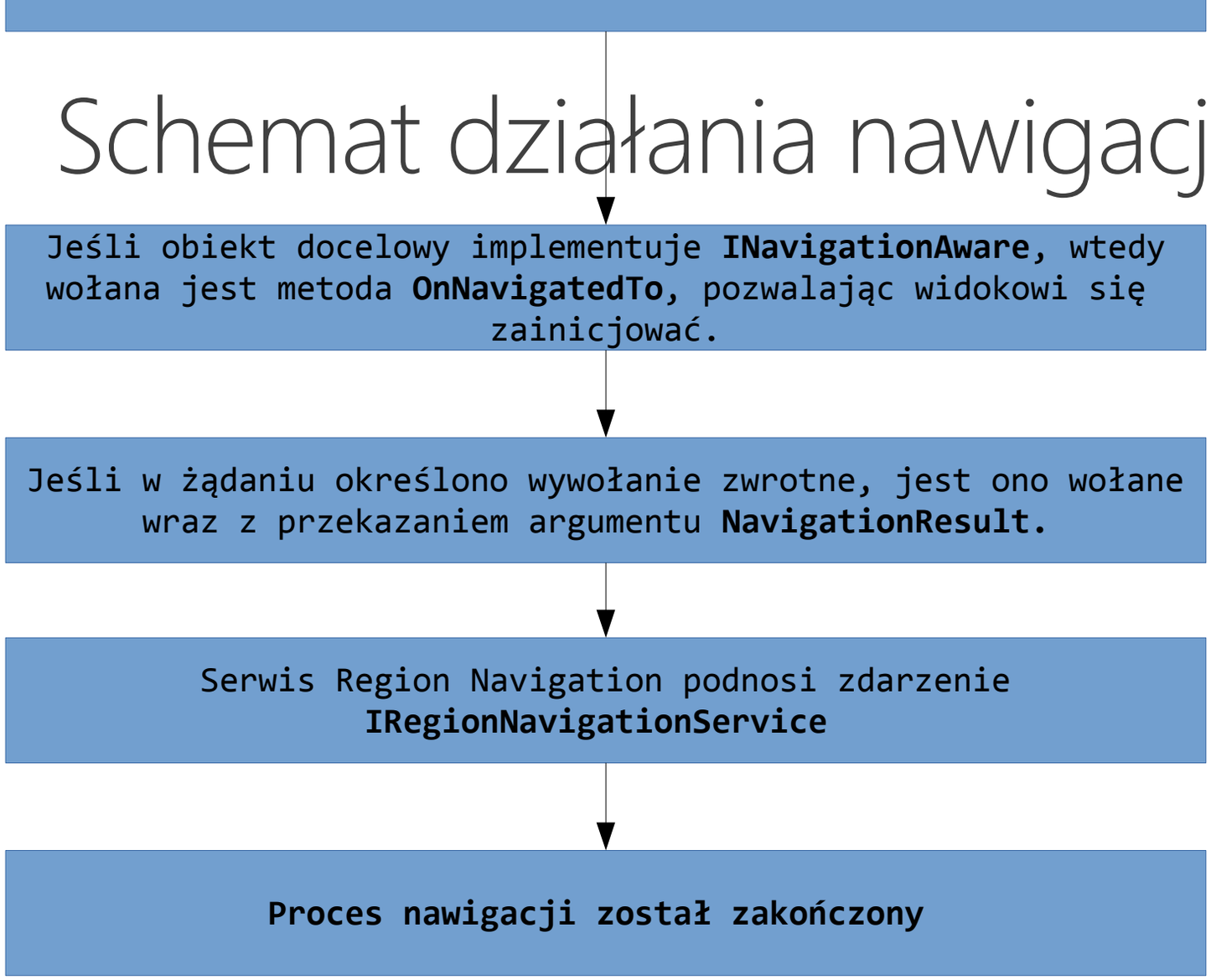
Schemat działania nawigacji w PRISM - 1



Schemat działania nawigacji w PRISM – 2



Schemat działania nawigacji w PRISM – 3



Dziękuję za uwagę !