Złożenie pracy online:
2014-03-30 17:48:50
Kod pracy:
11306
Kod załącznika:
11307

Krystian Talar
(nr albumu: 19973*INF/INŻ)

Praca inżynierska

Wykorzystanie zaawansowanych funkcji platformy WPF oraz wzorca projektowego MVVM w procesie tworzenia aplikacji biznesowej

Use the advanced features of the WPF platform and MVVM design pattern in the process of creating a business application

Wydział: Nauk Społecznych i Informatyki

Kierunek: Informatyka

Specjalność: inżynieria oprogramowania

Promotor: dr Włodzimierz Moczurad

Nowy Sącz, 2014

Streszczenie

Celem pracy inżynierskiej jest przedstawienie zaawansowanych funkcji platformy Windows Presentation Fundation w połączeniu z implementacją wzorca projektowego Model View View-Model w odniesieniu do procesu projektowania aplikacji biznesowych. Aplikacja hotelowa "DeLuxHotel" została napisana żeby przedstawić omawiane techniki na konkretnym przykładzie. Pisząc pracę inżynierską starałem się jak najdokładniej zaprezentować wszystkie najważniejsze mechanizmy platformy WPF ułatwiające budowę aplikacji biznesowych. Implementację wzorca MVVM oparłem na bazie biblioteki MVVM Light. W pracy zawarto projekt stylów oraz szablonów dla kontrolek systemu Windows, budowę zaawansowanych kontrolek użytkownika, niestandardowy layout, elementy 3D interfejsu użytkownika oraz zaawansowane animacje. Wszystkie te elementy wdrożono w przygotowany program "HotelDeLux" umożliwiający administrowanie i przeprowadzanie rezerwacji w ośrodku hotelowym.

Słowa kluczowe

aplikacja desktopowa, WPF, MVVM, .NET

Abstract

The main goal of this thesis is to introduce advanced function of Windows Presentation Fundation platform, along with implementation of Model View View-Model design pattern with respect to design busines applications. The "DeLuxHotel" application was created to show discussed techniques on exact example. The thesis is written ensuring the best accuracy in introducing all of the most crucial mechanisms of WPF platform taking into account the facilication of creating the busines applications. Implementation of MVVM design pattern is based on the MVVM Light library. The thesis includes project of styles and templates for Windows system controls, creation of advanced user controls, custom layout, 3D elements of user interface and advanced animations. All of those were implemented in designed program "HotelDeLux" which allows administration and performing bookings in hotel enviroment.

Keywords

desktop application, WPF, MVVM, .NET

Spis treści

1. Wstęp

Aplikacje biznesowe wymagają zarówno prawidłowego i wydajnego przetwarzania określonych danych, jak i intuicyjnej, prostej oraz atrakcyjnej formy wizualizowania interfejsu użytkownika. Jedną z najpopularniejszych i najnowszych technologii tworzenia nowoczesnych aplikacji desktopowych jest platforma Windows Presentation Fundation, która w połączeniu ze wzorcem projektowym MVVM daje bardzo potężne narzędzie pracy dla programistów. Wyżej wymienione technologie służą do budowy m. in. zaawansowanych aplikacji biznesowych w środowisku .NET.

Praca „Wykorzystanie zaawansowanych funkcji platformy WPF oraz wzorca projektowego MVVM w procesie tworzenia aplikacji biznesowej" ma na celu ukazanie zaawansowanych funkcji platformy WPF, które mogą zostać użyte przy projektowaniu dowolnej aplikacji biznesowej. Większość informacji dotyczących platformy WPF wykorzystanych przy budowie projektu załączonego do pracy opisowej bazuje na moich własnych doświadczeniach oraz jest dużym rozwinięciem wiedzy jaką nabyłem podczas studiów. W poniższym opracowaniu zostanie również pogłębiona idea wzorca projektowego MVVM na bazie biblioteki MVVM Light. Użycie wzorca zwiększa nakład pracy jaki musi włożyć programista w budowę projektu. Odpowiemy sobie na pytanie, w jakich sytuacjach wzorca MVVM powinno się używać i jakich sytuacji najlepiej unikać ze względów wydajnościowych.

W pracy zostanie omówiony przykład aplikacji biznesowej użyty do zaprezentowania ciekawych, niestandardowych interfejsów użytkownika. Następnie pokrótce zostaną opisane technologie, jakie zostały użyte przy budowie projektu. Zostanie przypomniana idea wzorca MVVM oraz opis możliwości biblioteki MVVM Light. Kolejnym krokiem będzie omówienie specjalnego mechanizmu platformy WPF o nazwie silnik łączenia, który umożliwił zaprojektowanie wzorca MVVM. Aplikacje biznesowe to aplikacje wielowarstwowe. Z reguły pierwszą warstwą w takiej aplikacji to warstwa bazy danych. Bez dobrze zaprojektowanej bazy nie można stworzyć poprawnie funkcjonującego programu. W jednym z rozdziałów będzie przedstawiona i opisana baza danych, która jest filarem logiki w przygotowywanej aplikacji. W następnym kroku zostanie przedstawiony główny layout programu oraz przepływ sterowania pomiędzy konkretnymi widokami. Na sam koniec po teoretycznym wstępnie będą zaprezentowane niestandardowe rozwiązania platformy WPF w celu zaprojektowania ciekawego interfejsu użytkownika, oraz efektywnego wykorzystania wzorca projektowego MVVM. Między innymi będą zaprezentowane:

- interfejs w technologii 3D;
- kontrolki imitujące okienka oraz zestaw paneli do zarządzania nimi. Wspominane kontrolki będą realizowały operacje biznesowe polegające na zarządzaniu podstawowymi zależnościami w bazie danych;
- widok architektoniczny w czasie rzeczywistym realizujący logikę biznesową.
- zaawansowana kontrolka kalendarzowa, wykorzystująca do wyrenderowania się niskiego poziomu elementy platformy WPF;
- zestaw szablonów dla kontrolek;
- animacje polepszające jakość wizualną prezentowanych użytkownikowi treści.

2. Studium przypadku – opis działania wybranej aplikacji biznesowej

Jako przykład do zaprezentowania zaawansowanych funkcji platformy WPF została wybrana aplikacja hotelowa. Jest to aplikacja biznesowa, wspierająca proces zarządzania modułem rezerwacji hotelowej. Głównym celem programu jest w sposób intuicyjny i przejrzysty umożliwić recepcjoniście dokonywanie rezerwacji klienckich oraz zarządzanie stenem pobytu klientów. Ponadto program ma umożliwić rozliczanie pobytu klienta w hotelu. Innowacją wprowadzoną do programu jest widok pozwalający na podgląd stanu pokojów w czasie rzeczywistym. W pokoju klientów zamontowano system sprawdzania obecności. Na bieżąco można sprawdzać, który pokój jest zajęty, a który wolny, czy wewnątrz są goście czy też nie. Cały widok jest przedstawiony w bardzo atrakcyjnej formie planu architektonicznego budynku hotelowego. Program posiada również specjalny interfejs umożliwiający administrowanie bazą danych.

Wybrałem przykład aplikacji hotelowej, ponieważ już wcześniej miałem doświadczenie przy projektowaniu tego typu programu. Wtedy jednak nie posiadałem bogatej wiedzy na temat platformy Windows Presentation Fundation oraz wzorca MVVM umożliwiających zaprojektowanie bogatej wizualnie oraz elastycznej aplikacji. Doświadczenie wynikające ze starciem się z logiką aplikacji hotelowych oraz rozległa wiedza na temat platformy WPF pozwoliły mi zaprojektować nowoczesną aplikację, którą wykorzystam jako tło do opisu ciekawych technologii w swojej pracy inżynierskiej.

3. Oprogramowanie

3.1. Visual Studio 2012

Projekt aplikacji został wykonany w programie Visual Studio 2012. Jest to bardzo zaawansowane środowisko programistyczne przystosowane do pracy nad dużymi projektami. Posiada ono szereg funkcji wspierających programistę podczas pracy.

3.2. SQL Server 2012

SQL Server 2012 jest to system zarządzania bazą danych wydany przez firmę Microsoft, który został wykorzystany w projekcie. Darmowa wersja Express umożliwia korzystanie z baz danych o pojemności do 10GB.

3.3. Expression Blend 4

Pracując nad efektami wizualnymi kontrolek pomocny okazał się program Expression Blend w wersji czwartej. Jest to narzędzie przeznaczone dla osób, które zajmują się układem interfejsu użytkownika. Pozwala ono np. na utworzenie zaawansowanych szablonów kontrolek oraz przeprowadzanie testów dynamicznie dołączanych danych do widoków.

4. Technologie

4.1. .Net Framework

Platforma programistyczna, która została opracowana przez firmę Microsoft. Główne komponenty platformy to środowisko uruchomieniowe (CLR) oraz zestaw bibliotek dostarczających podstawowych funkcjonalności dla aplikacji. Kod programu napisany pod platformę .NET jest kompilowany do kodu pośredniego, który z kolei jest tłumaczony na język maszynowy. Zaletą tego systemu jest multiplatformowość oraz możliwość korzystania z wielu języków dostosowanych do programowania pod platformę, z których pionierem jest C#. Aplikacja przygotowana jako projekt do pracy inżynierskiej wymaga zainstalowanej wersji czwartej platformy .NET. Wersja czwarta jest obecnie najpowszechniejszą wersją standardowo dostarczaną w systemach Windows XP, Vista, 7 oraz 8.

4.2. WPF (Windows Presentation Fundation)

Technologia platformy .NET, która jest przeznaczona do tworzenia aplikacji desktopowych z bogatym interfejsem użytkownika. W WPF-ie wszystkie komponenty użytkownika zostały przepisane ze standardowego kodu WinAPI na kod zarządzany platformy .NET. Zmodyfikowano podejście do projektowania kontrolek. Zamiast nadpisywać metody renderujące dany element piksel po pikselu zastępujemy tylko właściwość „Template" danej kontrolki specjalnym przygotowanym przez nas obiektem, przez co programista może nadawać kontrolkom dowolny wygląd przy niedużym nakładzie prazy. Zwiększono wydajność renderowania grafiki ,ponieważ proces ten wykorzystuje też GPU zamiast samego CPU. WPF oferuje rozwiązania layoutu niezależne od rozdzielczości klienta. Projektowanie wyglądu aplikacji odbywa się przy współpracy kodu znaczników języka XAML oraz języka C#. Przyjęto zasadę że kod XAML określa wygląd, natomiast w C# programista pisze logikę programu. Podejście to okazało się trafne i znalazło wielkie uznanie wśród programistów .NET. Technologia obsługuje grafikę wektorową oraz wiele rodzai multimediów. Więcej o WPF zostanie przedstawione w rozdziale poświęconym elementom, które sprawiły, że ta technologia jest tak potężna.

4.3. MVVM Light

Darmowa biblioteka umożliwiająca implementację wzorca projektowego Modem View View-Model w aplikacjach technologii WPF. Udostępnia zestaw bazowych klas które należy wykorzystać implementując wzorzec. Więcej informacji na ten temat znajduje się w rozdziale dotyczącym wzorca projektowego MVVM oraz jego implementacji w WPF.

4.4. iTextSharp

Darmowa biblioteka dostarczająca klasy i metody umożliwiające tworzenie i edytowanie dokumentów PDF. Bardzo prosta i efektywna, dostępna na wiele platform.

4.5. EntityFramework

System mapowania obiektowo-relacyjnego bazy danych. EF oparte jest na technologii ADO.NET i umożliwia w sposób obiektowy zarządzać elementami bazy danych. W tym podejściu nie wykonujemy zapytań SQL lecz operujemy na rekordach tabeli jak na obiektach. Mechanizm translacji akcji tłumaczy wykonywane przez nas operacje na zapytania SQL wysyłane do bazy. Technologia ta jest wolniejsza niż standardowe techniki łączenia z bazą danych, ale dużo bardziej wygodna dla programistów i bardziej odporna na błędy.

4.6. LINQ

Technologia wykonywania zapytań do obiektów w celu pozyskania z nich odpowiednich danych. Zapytania możemy wykonywać do obiektów implementujących interfejs IEnumerable, obiektów technologii Entity Framework oraz dokumentów XML. Wszechstronność i prostota LINQ sprawia, że technologia ta jest używana przez szerokie grono programistów i stała się nieodłączną częścią platformy .NET.

5. Implementacja wzorca MVVM za pomocą biblioteki MVVM Light

5.1. Wzorzec Model View View-Model

Jest to wzorzec projektowy warstwy prezentacji umożliwiający oddzielenie logiki biznesowej aplikacji od sposobu wizualizacji danych użytkownikowi. Wzorzec zalecany jest dla technologii WPF, Silverlight, ponieważ w naturalny sposób wykorzystuje mechanizmy obu freamworków, takie jak: silnik bindowania, szablony kontrolek, commandy, behavioury. Dodatkowymi zaletami wzorca jest łatwa testowalność aplikacji oraz czytelność i elastyczność przygotowanego przez programistę kodu.

5.1.1. Model

W warstwie modelu powinno implementować się całą logikę przygotowywanej aplikacji oraz mechanizmy odpowiedzialne za dostęp do danych. Przykładowo, w warstwie modelu można napisać klasy dostępu do bazy danych, połączeń z usługami sieciowymi, klasy pomocnicze służące jako forma nośnika przy przepływie informacji. Często spotykaną praktyką jest przygotowanie w warstwie modelu mapperów baz danych np. (nHibernate, Entity Framework). Takie właśnie podejście zostało zastosowane w przygotowanej przeze mnie aplikacji hotelowej.

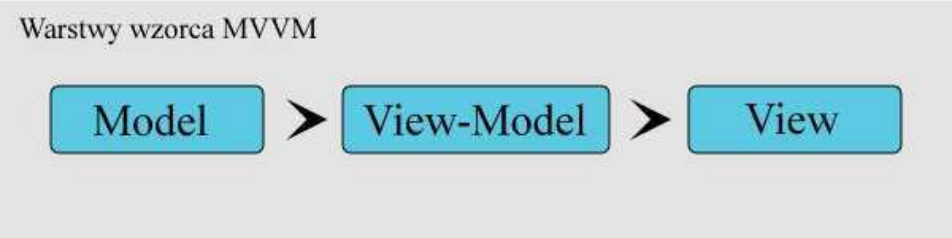
5.1.2. View-Model

Reprezentuje warstwę, która jest odpowiedzialna za dostarczanie danych do widoku. W tym zadaniu View-Model korzysta z klas warstwy Model. Co istotne, w View-Modelu nie ma żadnej referencji do widoku. Można więc powiedzieć, że omawiana warstwa jest całkowicie niezależna od warstwy View. Proces dostarczania danych do prezentacji odbywa się poprzez silnik łączenia danych oraz mechanizm komend, które znajdują się w widokach.

5.1.3. View

Warstwa prezentacyjna, w której określa się, jak powinny wyglądać dane pobrane z warstwy View-Model. Widoki tworzy się za pomocą kodu znaczników XAML, dzięki któremu programista ma możliwość stworzenia layoutu bez używania kodu zarządzanego np. C#. Przyjęto zasadę, że dobrze skonstruowany widok nie posiada praktycznie żadnego kodu code-behind (logiki np. napisanej w języku C#) oprócz konstruktora i metody inicjalizującej. Wyjątkami od tej reguły są jedynie widoki posiadające skomplikowaną animację.

View-Model dołączany jest do obiektu View poprzez dostarczenie go do właściwości DataContext. W technologii WPF oraz Silverlight każdy widok posiada tą właściwość, która jest miejscem, skąd bindowane są dane. Zgodnie z tym mechanizmem dane są bindowane od instancji klasy View-Modelu.



Rysunek 1 Zależność warstw wzorca MVVM

Jak wynika z powyższego rysunku, warstwa modelu nie powinna mieć styczności z widokiem. Widok jest odseparowany od logiki za pomocą warstwy View-Model, która odpowiada za dostarczanie do niego odpowiednich danych.

5.2. Opis MVVM Light

Biblioteka MVVM Light pomaga programiście w wydajny sposób wykorzystać wzorzec MVVM w aplikacjach typu WPF. Dostarcza ona zbiór klas i metod których wykorzystanie sprawia, że implementacja wzorca projektowego nie jest monotonna i nudna. Bez użycia wyżej wspominanej biblioteki programista w każdej klasie typu View-Model musiał implementować specjalny interfejs, dzięki któremu wprowadzano mechanizm notyfikacji danych. Jeśli wartość zmiennej zmieniła się w warstwie View-Model, informowany o tym był silnik bindingu i zmieniała się również renderowana wartość w warstwie View. Powszechny był również problem komunikacji pomiędzy warstwami View-Model. Nie istniał ujednolicony system komunikacji, jaki wprowadza biblioteka MVVM Light za pomocą klasy Messenger. Od momentu pojawienia się na rynku biblioteki MVVM Light programowanie aplikacji opartych na wzorcu MVVM okazało się naprawdę przyjemne. Poniżej zostanie omówiona struktura najistotniejszych, z punktu widzenia przygotowanego projektu, klas.

5.2.1. ObservableObject

Jest to bazowa klasa implementująca interfejs InotifyPropertyChanged, dzięki któremu możemy powiadamiać silnik bindingu o zmianie właściwości w warstwie View-Mode. Klasa

ta dostarcza szereg przydatnych metod, których celem jest odnotowanie zmiany wartości zmiennej. Jest to klasa bazowa dla wszystkich klas, które muszą posiadać funkcjonalność notyfikacji właściwości.

5.2.2. ViewModelBase

Klasa dziedzicząca po ObservableObject jest, jak sama nazwa wskazuje, bazowa dla wszystkich klas w hierarchii warstwy View-Model. Rozszerza zasób metod notyfikujących właściwości oraz dostarcza standardowego messengera (obiekt służący do komunikacji między obiektami we wzorcu MVVM). Implementuje interfejs ICleanup, który zawiera funkcję, gdzie programista powinien usunąć wszystkie użyte zasoby w momencie niszczenia obiektu. Klasa ta wystawia również zmienną informującą, czy projektowany obiekt znajduje się w trybie design, w którym programista, korzystający z programu Visual Studio 2012 ma możliwość zobaczyć przykładowo wyrenderowane dane.

5.2.3. Messenger

Klasa wystawiająca w postaci właściwości obiekt standardowego messengera, który jest singletonem. Metody tego obiektu umożliwiają rejestrację oraz wysyłanie komunikatów pomiędzy aktywnymi obiektami w aplikacji. Jest to kluczowa funkcjonalność w wypadku komunikacji, jaka często zachodzi w warstwie View-Model. Zaleca się, aby w formie komunikatów nie przesyłać dużych obiektów, jedynie wartości niezbędne do zrozumienia i zaimplementowania metody odbierającej komunikat.

5.3. Przykłady użycia

Poniżej zaprezentowany został przykład użycia dziedziczenia po klasie ObservableObject. KeyValueSelectedViewModel nie jest klasą przeznaczoną do przesłania danych konkretnego modelu, ale klasą pomocniczą, wystawiającą dowolne typy danych. Jej dodatkową funkcjonalnością jest możliwość zgłoszenia notyfikacji, kiedy dany obiekt został wybrany. Do tego była potrzebna klasa ObservableObject, która implementuje interfejs notyfikacji. Zastosowanie ViewModelBase niosłoby ze sobą implementacje wielu innych metod i właściwości, które tutaj nie byłyby potrzebne, ponieważ z założenia ta klasa musi być lekka. Jej zastosowanie to przesłanie danych do widoku z listą wyboru.

```
class KeyValueSelectedViewModel<K, V, D, D2> : ObservableObject
{
    public K Key { get; set; }
    public V Value { get; set; }
    public D AdditionalData { get; set; }
    public D2 AdditionalData2 { get; set; }

    public const string IsSelectedPropertyName = "IsSelected";
    private bool isSelected = true;
    public bool IsSelected
    {
        get
        {
            return  isSelected;
        }

        set
        {
            if ( isSelected == value)
            {
                return;
            }

            RaisePropertyChanging(IsSelectedPropertyName);
             isSelected = value;
            RaisePropertyChanged(IsSelectedPropertyName);
        }
    }
}
```

Rysunek 2 Przykład klasy dziedziczącej po ObservableObject i wykorzystanie funkcji notyfikacyjnych.

Kolejnym przykładem jest metoda, która ma za zadanie przygotować obiekt wybrany przez użytkownika z listy wyboru, a następnie wysłać komunikat o zamknięciu warstwy wyboru danych. Wysłanie komunikatu odbywa się za pomocą metody Send, przyjmującej za argumenty identyfikator kanału przesyłu komunikatów oraz wysyłany obiekt.

```
protected override void sendData()
{
    List<KeyAndValue> objectToSave = new List<KeyAndValue>();
    if (SelectedCollection.Count > 0)
    {
        foreach (var guest in SelectedCollection)
        {
            objectToSave.Add(new KeyAndValue { Key=guest.Id, Value=guest.Imie+" "+guest.Nazwisko });
        }
    }
    ChoiceStorage.ObjectFromChoice = objectToSave;
    Messenger.Default.Send<int>(1, "ChoiceLayerMSGClose");
}
```

Rysunek 3 Przykład wysłania komunikatu przez domyślnego messengera.

Ostatnim przykładem jest zarejestrowanie metody obsługującej komunikaty odebrane przez domyślnego messengera. Miejsce zarejestrowania metody to konstruktor klasy odpowiadającej za warstwę wyboru danych. Messegner zarejestrowany jest do instancji obiektu tej klasy, co gwarantuje odpowiednią żywotność procesu odbioru komunikatów,

dotyczących warstwy wyboru danych. Odbiór komunikatu obywa się za pomocą metody Register, która przyjmuje za argumenty unikalny token identyfikujący kanał, w którym będą przesyłane wiadomości, obiekt, do którego będzie zarejestrowana stacja odbierająca komunikat oraz metodę obsługującą obiekt przesłany w komunikacie.

```
public DataChoiceLayer()
{
    InitializeComponent();
    Messenger.Default.Register<TokenTypeMessage>(this, "ChoiceLayerMSGOpen", new Action<TokenTypeMessage>(p =>
    {
        addressToken = p.Tokem;
        var constructorInfo = p.ViewType.GetConstructor(new Type[] { });
        var control = constructorInfo.Invoke(new object[] { }) as UserControl;
        if (control != null)
        {
            ChoiceElement = control;
        }
        openLayer();
    }));
    Messenger.Default.Register<int>(this, "ChoiceLayerMSGClose", new Action<int>(
        (p) =>
        {
            Messenger.Default.Send<object>(ChoiceStorage.ObjectFromChoice, addressToken);
            closeLayer();
        }
        ));
}
```

Rysunek 4 Przykład odbioru i obsługi komunikatu przez domyślnego messengera.

6. Silnik łączenia WPF

Silnik łączenia danych jest bardzo potężnym mechanizmem, który jest wykorzystywany w każdej zaawansowanej aplikacji WPF. W mojej pracy inżynierskiej bardzo często korzystam z wielu konfiguracji omawianego silnika. W prawie każdym przytoczonym kodzie XAML będą się pojawiały znaczniki informujące o połączeniu danych, przez co ważne jest dostarczenie podstawowych informacji o tym przydanym mechanizmie.

Silnik łączenia danych jest jednym z najistotniejszych składników platformy WPF, który musi znać każdy programista chcący stworzyć aplikację bardziej zaawansowaną niż popularny „Hello world". Jest to mechanizm opracowany z myślą o łatwym i intuicyjnym dostarczaniu danych do kontrolek, które mają je wyświetlać. Silnik łączenia po raz pierwszy pojawił się w aplikacjach desktopowych typu WinForms, jednak dopiero w platformie WPF został on rozwinięty i dostosowany do tego stopnia, że dzięki temu mechanizmowi mógł powstać wzorzec MVVM. Główna i najistotniejsza różnica polega na tym, że w starszej technologii samo łączenie musiało być napisane w kodzie zarządzanym, co było niekiedy dużym wyzwaniem dla początkujących programistów. Natomiast w WPF-ie połączenie z danymi odbywa się w deklaratywnym języku XAML, po stronie interfejsu użytkownika, a nie logiki działania programu.

```
Binding myBinding1 = new Binding("ActiveKey");
myBinding1.Mode = BindingMode.TwoWay;
myBinding1.ValidatesOnDataErrors = true;
myBinding1.Source = this.DataContext;
this.SetBinding(ActiveKeyProperty, myBinding1);
```

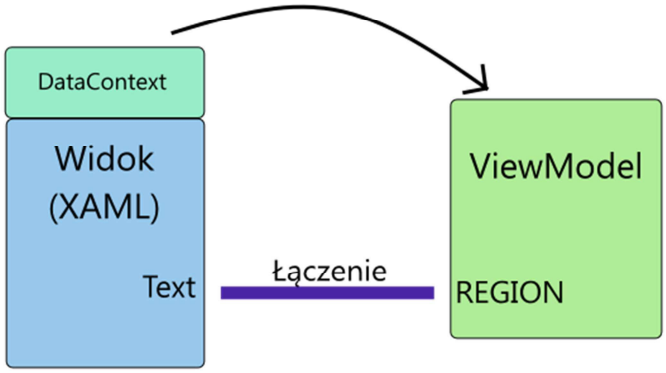
Rysunek 5 Przykład stworzenia łączenia w języku C#

```
Text="{Binding REGON, UpdateSourceTrigger=PropertyChanged, ValidatesOnDataErrors=True}"
```

Rysunek 6 Przykład tworzenia łączenia w języku XAML

Powyżej przedstawione są dwa sposoby tworzenia odniesienia do danych za pomocą mechanizmu silnika łączenia. Dużo bardziej intuicyjny i wygodny jest zapis w języku XAML. Ponadto XAML jest językiem opisu interfejsu użytkownika, a więc łączenia odbywa się po stronie widoku. Biorąc pod uwagę powyższy przykład, we właściwości" DataContext" widoku musi znajdować się typ ViewModel, który posiada właściwość „REGION". Wartość tej właściwości będzie wyświetlana jako tekst w pewnej kontrolce po stronie widoku. Największą zaletą wzorca MVVM jest to że, warstwa ViewModel nic nie wie o stanie

interfejsu użytkownika, pod który została podpięta. Tę ciekawą właściwość umożliwia właśnie silnik łączenia danych, który jest użyty po stronie widoku czerpiącego dane z udostępnionego mu obiektu warstwy ViewModel.



Rysunek 7 Diagram obrazujący najważniejsze składowe mechanizmu łączenia danych odnośnie wzorca MVVM

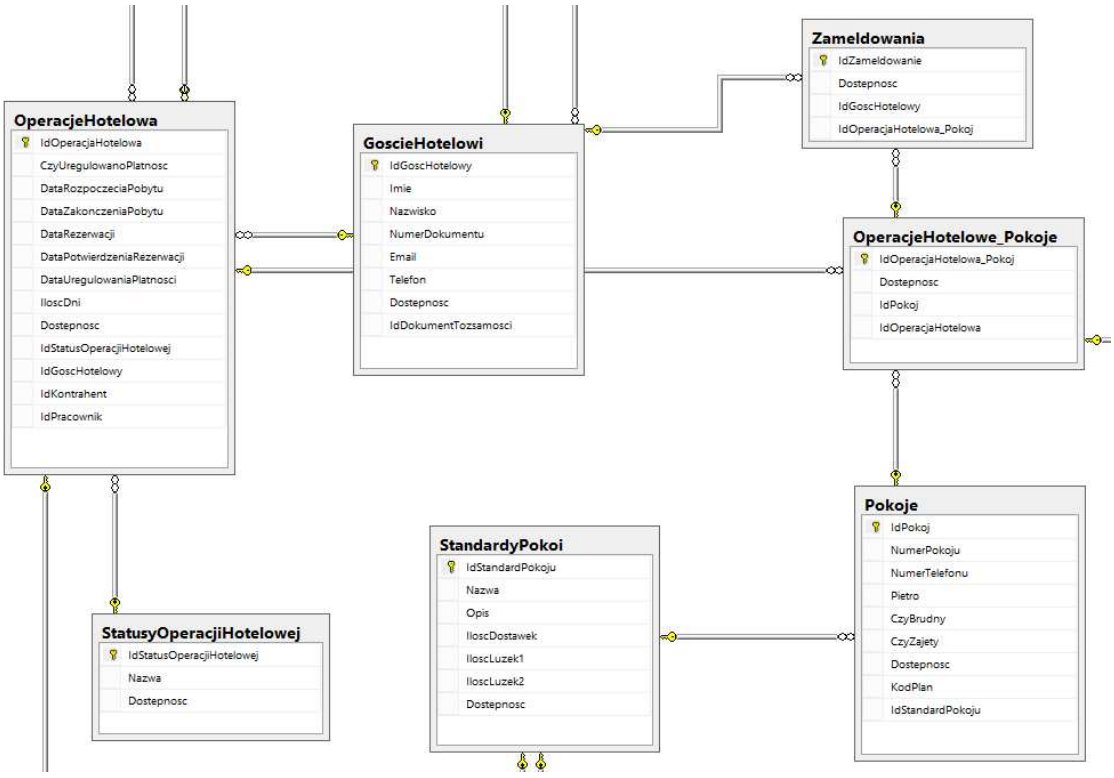
Łączenie można ustawić tylko pomiędzy specjalnymi konstrukcjami języka C# nazywanymi właściwościami. Gdyby np. połączenie zostało ustanowione ze zwykłą publiczną składową klasy, nie odniosłoby ono skutku. Jeśli podczas przesyłania danych wystąpią jakiekolwiek błędy a programista nie dostarczy sposobu ich obsługi, to zostaną one przed nim ukryte. Nie będzie wtedy zgłoszony żaden wyjątek. Projektując połączenie należy być bardzo ostrożnym. Łączenie danych polega na ustawieniu pewnej zależności między obiektem, który jest odbiorcą, a obiektem, który jest źródłem informacji. Można to sobie wyobrazić jako pomost pomiędzy właściwościami w programie. Jeśli jedna z tych właściwości się zmieni, informacja o tym jest przekazana do drugiej. Zmiana jest odnotowywana i zostają podjęte odpowiednie procedury jej obsługi. Zarówno kierunek komunikacji, jak i inne cechy mechanizmu łączenia danych można konfigurować.

7. Struktura bazy danych

Aby dokładnie zrozumieć logikę działania programu, należy omówić model domeny aplikacji, którym jest warstwa klas mapujących tabele bazy danych, zaprojektowana zgodnie z podejściem CodeFirst technologii EntityFramework. Wspomniane podejście umożliwia dużą elastyczność podczas przygotowywania aplikacji. Cała logika oparta jest ma klasach języka C#, dlatego wprowadzanie zmian sprowadza się do zmodyfikowania lub zaimplementowania nowych właściwości. Dodatkowo, jeżeli chcemy dokonywać bardziej skomplikowanych operacji, mamy do dyspozycji specjalne API, które nam to umożliwia. Fluent Api to zbiór klas i metod dla programisty chcącego zaimplementować funkcjonalność wykraczającą poza standardowy zbiór rozwiązań Entity Framework Code First.

W tym rozdziale zostaną przedstawione schematy połączeń najważniejszych tabel, będących podstawą danych dla realizacji określonej logiki działania. Zostanie również zaprezentowana przykładowa klasa C# użyta w mechanizmie mapowania obiektowo relacyjnego EntityFramework.

7.1. Operacje klienta



Rysunek 8 Schemat części bazy danych odpowiedzialnej za przechowywanie danych o operacjach klienta

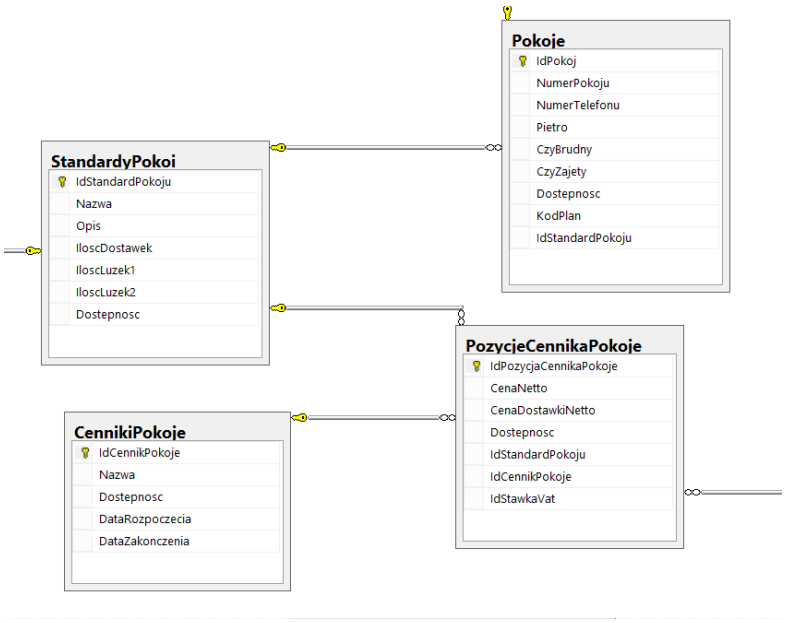
Poprzez operacje klienta rozumie się czynność dokonaną podczas pobytu lub rezerwacji dokonanej przez osobę chcącą skorzystać z usług hotelu. Przedstawiony powyżej schemat jest

odpowiedzialny za obsługę gościa hotelowego od momentu rezerwacji aż do rozliczenia finansowego odbytego pobytu. Tabela „OperacjaHotelowa” przechowuje podstawowe informacje o typie oraz stanie operacji dokonanej przez klienta. Status operacji hotelowej określa następujące stany:

- rezerwacja niepotwierdzona (klient dzwoni z chęcią rezerwacji pokoi);
- rezerwacja potwierdzona (klient potwierdza rezerwację);
- pobyt rozpoczęty (klient wprowadza się do hotelu);
- pobyt zakończony (klient wyprowadza się; pokoje wynajmowane zostają zwolnione);
- operacja anulowana (klient odwołuje rezerwację).

Do operacji hotelowej dołączone jest powiązanie z encją gościa hotelowego. To powiązanie jest odpowiedzialne za osobę, na którą została zabukowana operacja. Klient podaje swoje dane i bierze odpowiedzialność za rezerwację hotelową. Z operacją hotelową powiązane są pokoje jakie chce on wynająć. Do każdego pokoju może wprowadzić się kilkoro gości. Ich dane osobowe są przechowywane w tabeli „Zameldowania”. Każdy pokój ma swój standard, co jest zrealizowane połączeniem pomiędzy tabelami „Pokoje” oraz „StandardyPokoi”.

7.2. Cennik pokoi

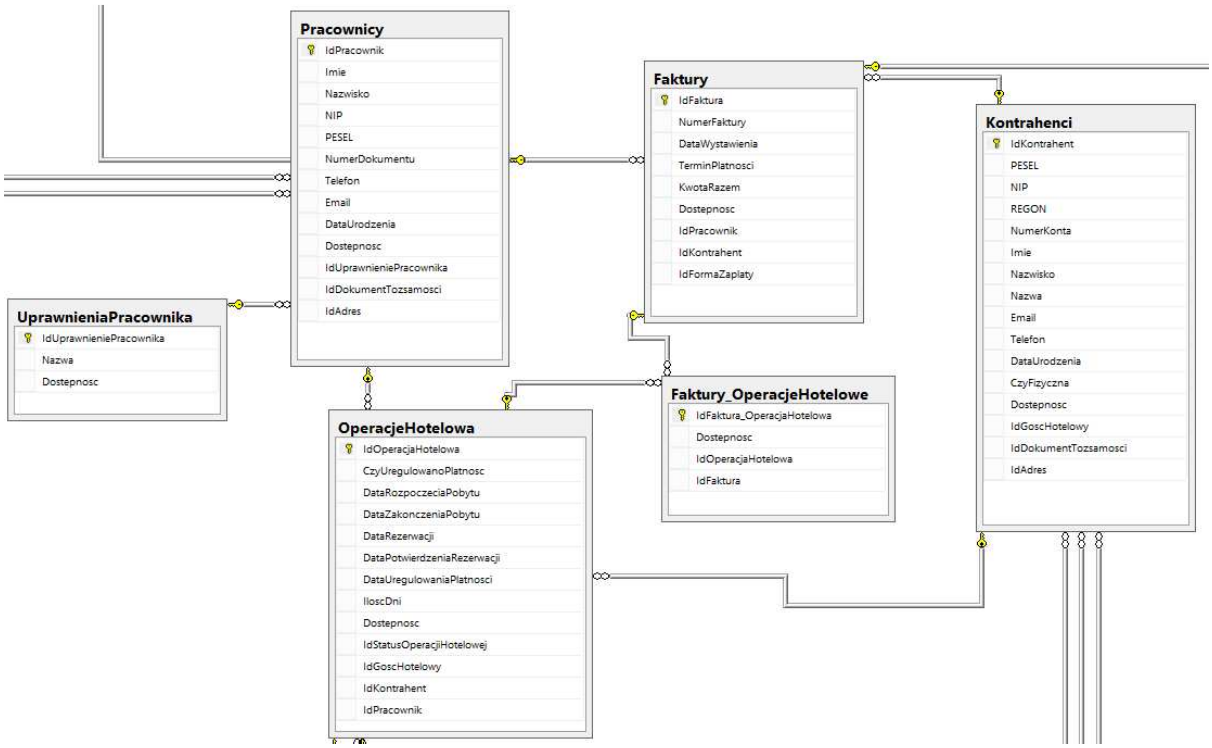


Rysunek 9 Schemat części bazy danych odpowiedzialny za przechowywanie informacji o zmianach cen pokoi

Ceny są wartościami zmiennymi w czasie. Nie można zaimplementować cen tak, aby były pamiętane tylko aktualne wartości danego produktu. Jest to ewidentnym błędem projektanta aplikacji. Ceny powinny być pamiętane dla każdej ich zmiany. Powyższy schemat określa

zależność pomiędzy pokojami, a ich standardami. Ceny zależą od standardu pokoju. Każdy standard powinien mieś swoją wartość ceny w każdym cenniku. Cenniki posiadają okres ważności. Aktualny jest ten cennik, który nie ma zdefiniowanej daty końcowej. Dzięki temu mechanizmowi w przygotowanej aplikacji został zaimplementowany mechanizm pamiętania wartości zmiennych w czasie, w konkretnym przypadku cen.

7.3. Wystawianie faktur



Rysunek 10 Schemat części bazy danych odpowiedzialny za przechowywanie informacji o wystawionych fakturach

Wystawianie dokumentów to bardzo ważny aspekt aplikacji biznesowych. W przygotowanej aplikacji również został on zaimplementowany. Gość hotelowy po zakończonym pobycie chce otrzymać fakturę do rozliczenia. Należy więc przechowywać i zorganizować wszystkie niezbędne informacje, aby mu takową fakturę móc wystawić. Powyższe tabele i połączenia między nimi realizują następującą funkcjonalność:

- pracownik jest osobą wystawiającą fakturę dla klienta z określonych pobytów;
- na fakturze może się znaleźć wiele operacji hotelowych;
- faktura jest wystawiona na określonego kontrahenta.

7.4. Przykład mapowania obiektowo relacyjnego

```
[Table("OperacjeHotelowa")]
public class OperacjaHotelowa
{
    [Key]
    public int IdOperacjaHotelowa { get; set; }
    public bool CzyUregulowanoPlatnosc { get; set; }
    public DateTime? DataRozpoczeciaPobytu { get; set; }
    public DateTime? DataZakonczeniaPobytu { get; set; }
    public DateTime? DataRezerwacji { get; set; }
    public DateTime? DataPotwierdzeniaRezerwacji { get; set; }
    public DateTime? DataUregulowaniaPlatnosci { get; set; }
    public int IloscDni { get; set; }
    public bool Dostepnosc { get; set; }

    public int IdStatusOperacjiHotelowej { get; set; }
    public virtual StatusOperacjiHotelowej StatusOperacjiHotelowej { get; set; }
    public int? IdGoscHotelowy { get; set; }
    public virtual GoscHotelowy GoscHotelowy { get; set; }
    public int? IdKontrahent { get; set; }
    public virtual Kontrahent Kontrahent { get; set; }
    public int? IdPracownik { get; set; }
    public virtual Pracownik Pracownik { get; set; }

    public virtual ICollection<OperacjaHotelowa_Pokoj> OperacjaHotelowa_Pokoj { get; set; }
}
```

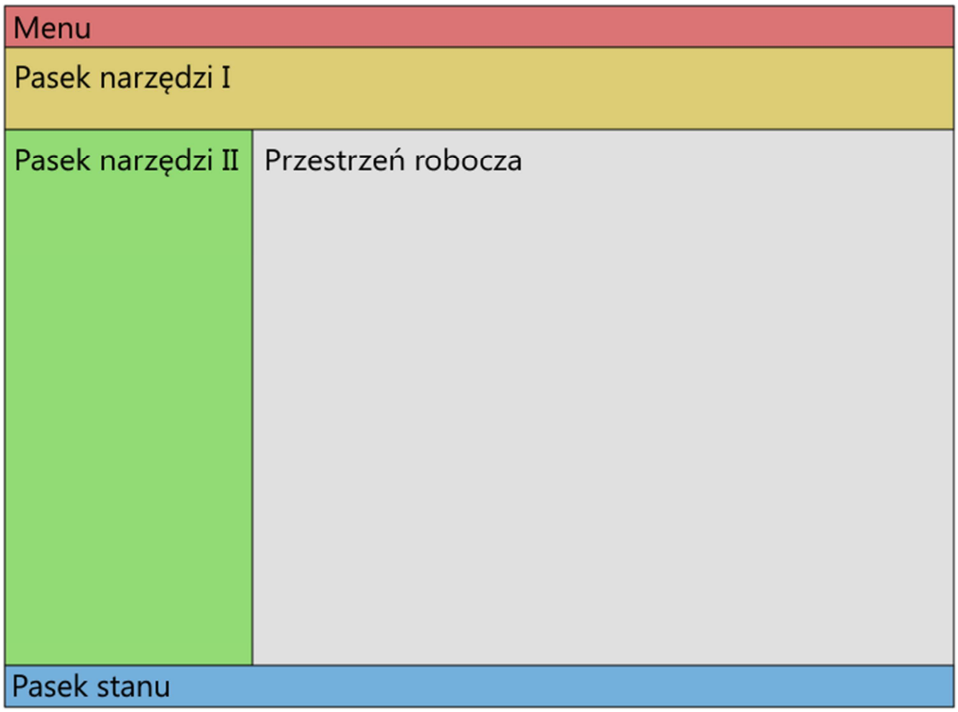
Rysunek 11 Przykładowa klasa modelu podejścia CodeFirst

Jak przedstawia rysunek, mapowanie obiektowo – relacyjne odbywa się w całkiem naturalny sposób. Nazwa klasy domyślnie odpowiada nazwie tabeli. Powyżej został zastosowany atrybut zmieniający nazwę na liczbę mnogą. Wszystkie właściwości klasy odpowiadają kolumnom w tabeli o określonym przez właściwość typie. Połączenia z innymi tabelami to referencje ze słowem kluczowym „virtual” do typów użytych w mapowaniu obiektowo – relacyjnym. W przedstawiony sposób bardzo łatwo można zmapować wszystkie podstawowe struktury bazy danych.

8. Główny layout aplikacji

8.1. Wprowadzenie do tematu

Layout aplikacji określa pozycję każdego elementu, jaki zostaje wyświetlony użytkownikowi. W głównym layoucie zostają wyświetlone wszystkie inne widoki wchodzące w skład prezentowanej treści. Aby przystosować layout do użytkownika możemy w nim wydzielić część dynamiczną, gdzie będą wyświetlane aktualne widoki, które użytkownik wybrał oraz tzw. paski narzędziowe, usprawniające nawigację. Cały projekt układu aplikacji zależy od specyfiki programu i wymagań użytkowników. Główną cechą interfejsu użytkownika powinna być prostota oraz przejrzystość, idąca w parze z pozytywnymi doznaniami wizualnymi.

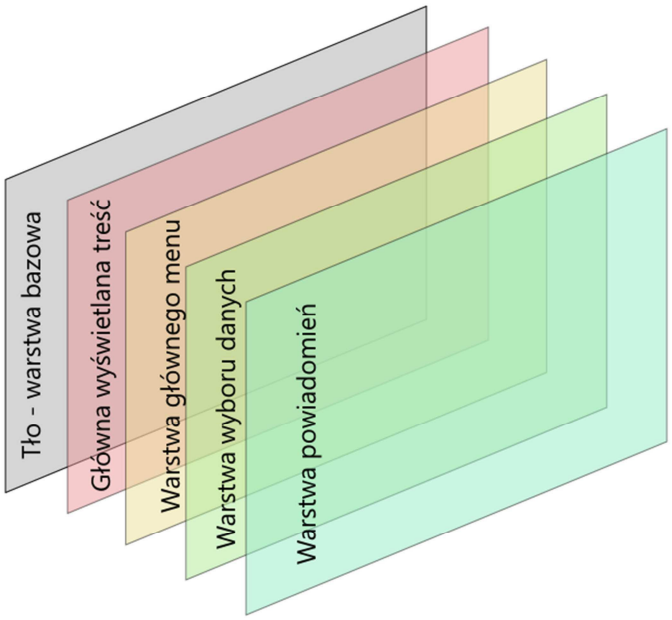


Rysunek 12 Przykładowy układ layoutu w aplikacji

Powyższy układ jest bardzo często spotykany w aplikacjach biznesowych. Przy dużych ilościach prezentowanych treści staje się jednak słabo czytelny i kłopotliwy w obsłudze. Nie wprowadza on żadnej sekwencji dla użytkownika, który bardzo łatwo może się w nim pogubić. Wyobraźmy sobie sytuację, kiedy uruchamiamy widok do obsługi pewnych elementów bazy danych, które wymagają wyboru innych elementów. Brzmi to dość prosto, jednak dla programisty może być to pewien kłopot z odpowiednim doborem układu. W momencie, gdy wszystko odbywa się w jednym panelu np. obsługującym zakładki, w bardzo

szybkim czasie otwiera się kilka, a nawet kilkanaście zakładek i tworzy się bałagan w przestrzeni roboczej. Na dodatek wyobraźmy sobie, że w międzyczasie powstał błąd i na ekranie pojawiają się okienka typu „MessageBox”. W takiej sytuacji powyższy układ z pewnością nie okaże się pomocny i przyjazny użytkownikowi.

W przygotowanej aplikacji zostało przyjęte całkiem inne podejście do budowy layoutu. Biorąc pod uwagę powyższe problemy, można przygotować prosty i funkcjonalny układ warstwowy.



Rysunek 13 Model warstwowy layoutu

Każda warstwa ma inny zakres odpowiedzialności i pełni inną funkcję w programie:

- tło – jest to warstwa bazowa, wyświetlana w momencie, gdy żadne inne warstwy nie przejmą pierwszeństwa;
- warstwa głównej treści – w tej warstwie znajduje się obiekt koordynujący wyświetlanie wybranych przez użytkownika treści;
- warstwa menu – w aplikacji przejścia, pomiędzy głównymi funkcjonalnościami realizowane są poprzez nadrzędne menu; menu to znajduje się w osobnej warstwie i jest sterowane kliknięciem prawego klawisza myszy;
- warstwa wyboru danych – w tej warstwie wyświetlane są wszystkie kontrolki umożliwiające niezależny wybór pewnych informacji;

- warstwa powiadomień – warstwa odpowiedzialna za wyświetlanie komunikatów; obsługuje różne typu komunikatów; API tej warstwy zostało wyprowadzone w postaci fasady.

Kolejność warstw nie jest przypadkowa. Warstwy powyżej przykrywają warstwy niższe. Z tego wynika, że warstwy umieszczone wyżej mają większy priorytet. Np. jesteśmy w warstwie wyboru danych i nastąpiła czynność, która spowodowała wysłanie komunikatu o wyświetleniu informacji użytkownikowi. Warstwa powiadomień musi być nad warstwą wyboru, aby użytkownik mógł zobaczyć wygenerowaną informację. W przeciwnym wypadku informacja zostałaby zakryta.

8.2. Realizacja warstw na zasadzie implementacji kontrolki

Większość z warstw została zdefiniowana jako niezależny „UserControl". Typ warstwy dziedziczy po wspominanej klasie. Klasa „UserControl", jak sama nazwa wskazuje, to kontrolka użytkownika. Jest ona bardzo popularną klasą bazową dla niestandardowych kontrolek w technologii WPF. Większość z warstw została zdefiniowana jako lekka kontrolka. W pliku XAML został opisany ich wygląd, natomiast w code-behind ich zasada działania. Większość z warstw odpowiada tylko za siebie, nie posiada informacji o współistnieniu innych. Do komunikacji z warstwami została wykorzystana domyślna implementacja klasy „Messenger" z biblioteki „MVVM Light". W konstruktorze warstwy zarejestrowano metodę obsługującą wysłany komunikat o specjalnym tokenie. Dla każdej warstwy jest inny token, dzięki czemu warstwy posiadają wiedzę, które komunikaty przetwarzać, a które nie.

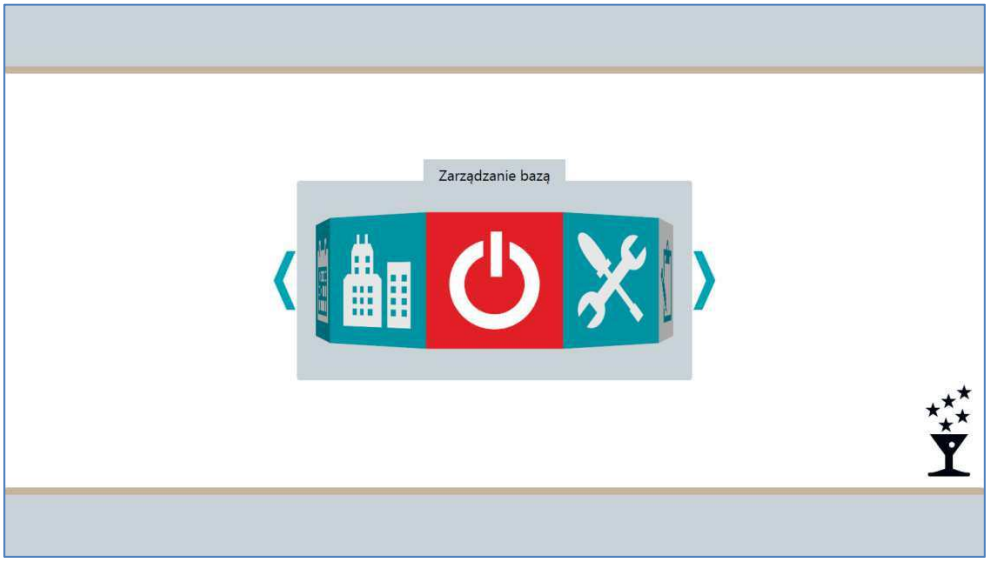
8.3. Realizacja warstwy z wykorzystaniem elementu ContentControl

Warstwa głównej treści została zaimplementowana z wykorzystaniem elementu „ContentControl". Jest to element w technologii WPF, którego najważniejszym zadaniem jest posiadanie i sterowanie wyglądem zawartości. Posiada on możliwość do wyświetlania dowolnej zawartości. Sterowanie tą warstwą odbywa się w pliku code-behind głównego okna aplikacji. Odbierane są komunikaty przekazujące informację jakiego rodzaju zawartość należy umieścić w elemencie „ContentControl". Zastosowanie tego elementu jest w tym przypadku całkiem naturalne. Pozostałe warstwy zostały zaimplementowane jako kontrolki użytkownika, głównie ze względu na niezależność ich działań, zaś zawartość elementu „ContentControl" ściśle wiąże się z zawartością głównego okna.

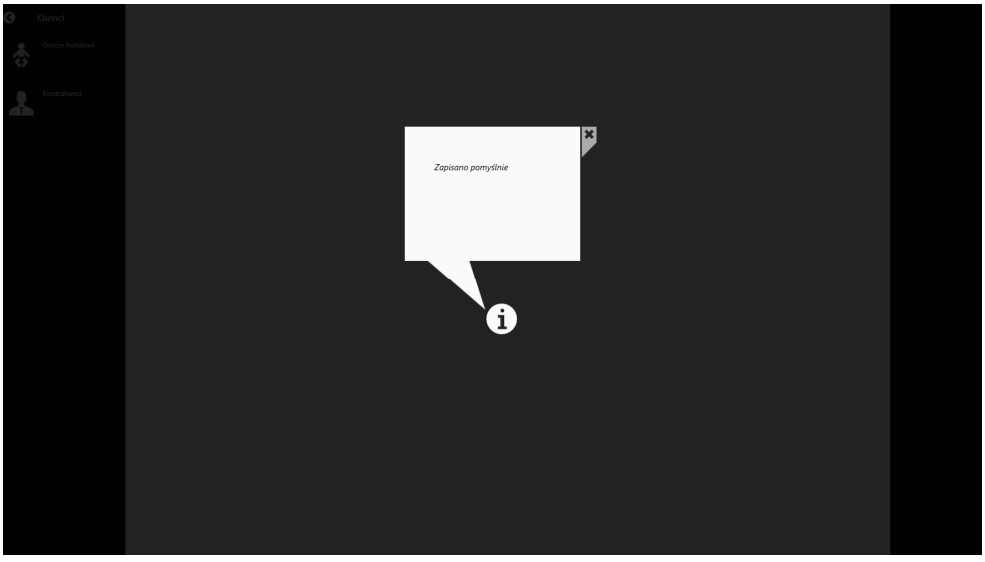
```
<Grid x:Name="LayoutRoot">
    <ContentControl Name="ContentControlMainContent"/>
    <con:Menu3D x:Name="Menu3D"/>
    <con:DataChoiceLayer LayerCollapsed="True"/>
    <con:ModernMessageBox MessageBoxCollapsed="True"/>
</Grid>
```

Rysunek 14 Kod zawarty w głównym oknie aplikacji. Cztery warstwy i pojemnik Grid jako tło

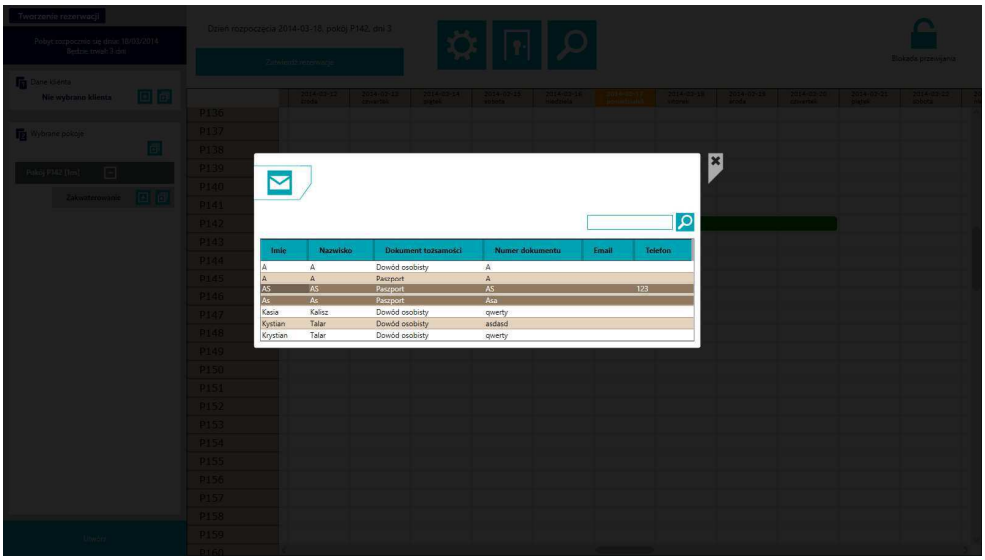
8.4. Prezentacja warstw



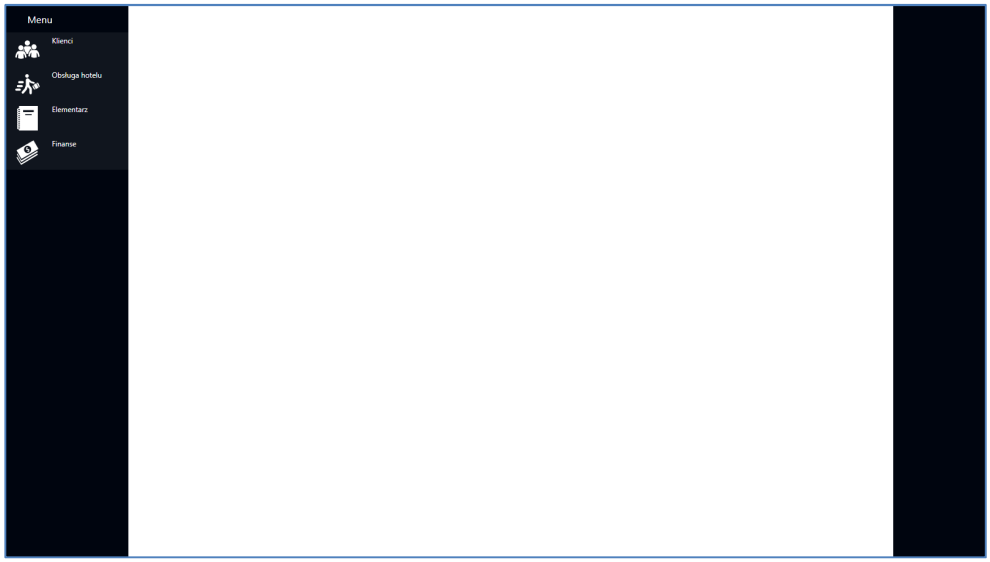
Rysunek 15 Warstwa głównego menu



Rysunek 16 Warstwa powiadomień prezentująca przykładową informację dla użytkownika o pomyślnym zapisaniu elementu do bazy



Rysunek 17 Warstwa wyboru danych z przykładową kontrolką prezentującą możliwych do wyboru gości hotelowych



Rysunek 18 Warstwa głównej treści z przykładowym widokiem panelu do zarządzania elementami bazy hotelowej

Warstwa tła nie została zaprezentowana, gdyż zawiera jedynie kolor biały. Można zauważyć, że niektóre warstwy stosują przyciemnianie tła dla zawartości warstw znajdujących się niżej. Funkcjonalność ta została zaimplementowana w celu skupienia większej uwagi użytkownika na głównej zawartości warstwy. Przykładowo, w przypadku pojawienia się komunikatu, użytkownik powinien skupić się wyłącznie na nim i na niego zareagować.

9. Wykorzystanie zaawansowanych funkcji platformy WPF

9.1. Style i szablony kontrolek

9.1.1. Wyjaśnienie teoretyczne

W technologii WPF całkowicie zmieniono podejście do renderowania wyglądu kontrolek. W starszych technologiach takich jak WinAPI czy WinForms, programista miał do wyboru zestaw bazowych kontrolek, które miały narzucony wygląd. Można było zmienić jedynie pewne parametry danej kontrolki, takie jak np. tło czy kolor tekstu. Nie można było w całości zredefiniować wyglądu komponentu. Wszelkie próby tego typu operacji były bardzo kosztowne i skomplikowane, ponieważ wymagały napisania kodu renderującego kontrolkę wraz z wszystkimi jej parametrami. Użycie grafik było pewnym rozwiązaniem w przypadku prostych kontrolek jak np. przyciski, lecz z pewnością nie prowadziło do wydajnego i elastycznego rezultatu.

WPF zmienił to standardowe podejście i dostarczył programiście możliwość dowolnej zmiany wyglądu kontrolki. Można np. zaprojektować przycisk aby wyglądał jak kamień szlachetny. Wszystkie operacje dzieją się bez utraty jakości, ponieważ WPF opiera się głównie na grafice wektorowej. Aby to było możliwe, technologia ta dostarcza wszystkim elementom dziedziczącym po klasie „FrameworkElement" właściwość „Style", oraz elementom dziedziczącym po klasie „Control" właściwość „Template". Warto tutaj wspomnieć, że klasa „Control" wywodzi się z klasy „FrameworkElement" co oznacza że wszystkie elementy z właściwością „Style" posiadają właściwość „Template", ale odwrotnie już ta zależność nie zachodzi.

Warto w tym miejscu postawić pytanie, jaka jest różnica pomiędzy stylem (funkcją właściwości „Style"), a szablonem (funkcją właściwości „Template")? Styl jest bardziej ogólny od szablonu. W stylu programista może nadać wartości różnym właściwościom opisującym wygląd elementu. Szablon natomiast określa, jak dany element ma wyglądać i jak ma interpretować właściwości opisujące wygląd kontrolki. W stylu można przypisać kontrolce szablon, ponieważ jest on właściwością. Zarówno w stylach jak i szablonach można definiować tzw. „Triggery". Są to akcje, na które reaguje element np. zmieniając swój wygląd. Dzięki temu mechanizmowi łatwo zaimplementować takie zachowanie jak przyciemnienie tła elementu po najechaniu na niego myszką.

9.1.2. Przykład stylu

Poniżej zademonstrowano prosty styl, który jest domyślny dla wszystkich elementów typu „TextBlock” w stworzonej aplikacji. Jest on domyślny ponieważ nie ma zdefiniowanego klucza po którym byłby identyfikowany. Zdefiniowanie identyfikatora wiąże się z koniecznością jego użycia, w przypadku chęci nadania kontrolce danego stylu.

```
<Style TargetType="TextBox">
    <Setter Property="Margin" Value="5,3"/>
    <Setter Property="Padding" Value="5,2"/>
    <Setter Property="Width" Value="200"/>
    <Setter Property="TextWrapping" Value="Wrap"/>
    <Setter Property="TextAlignment" Value="Left"/>
    <Setter Property="BorderBrush" Value="#00A3B4"/>
    <Setter Property="CaretBrush" Value="Black"/>
    <Setter Property="SelectionBrush" Value="#5BC9E1"/>
</Style>
```

Rysunek 19 Prosty styl bez klucza

```
<Style x:Key="DefaultTextBoxStyle" TargetType="TextBox">
    <Setter Property="Margin" Value="5,3"/>
    <Setter Property="Padding" Value="5,2"/>
    <Setter Property="Width" Value="200"/>
    <Setter Property="TextWrapping" Value="Wrap"/>
    <Setter Property="TextAlignment" Value="Left"/>
    <Setter Property="BorderBrush" Value="#00A3B4"/>
    <Setter Property="CaretBrush" Value="Black"/>
    <Setter Property="SelectionBrush" Value="#5BC9E1"/>
</Style>
```

Rysunek 20 Prosty styl z kluczem identyfikującym

Powyższy styl następująco modyfikuje element typu „TextBox”:

- ustawia lewy i prawy margines zewnętrzny na 5 jednostek; ustawia górny i dolny margines zewnętrzny na 3 jednostki;
- ustawia lewy i prawy margines wewnętrzny na 5 jednostek; ustawia górny i dolny margines wewnętrzny na 2 jednostki;
- ustawia szerokość elementu na 200 jednostek;
- ustawia zwijanie tekstu na szerokości elementu;
- ustawia wyrównanie tekstu do lewej strony;
- ustawia kolor obramowania;
- ustawia kolor karetki;
- ustawia kolor zaznaczonego tekstu;

Rysunek 21 Efekt dołączenia stylu do elementu "TextBox"

Jak zostało zaprezentowane, styl umożliwia zmianę istniejących właściwości w elemencie. Dzięki temu projektant może ujednolicić ogólny wygląd pewnej grupy kontrolek i co ważniejsze, łatwiej go modyfikować.

9.1.3. Przykład szablonu

Szablony określają wygląd elementu. Istnieją różne rodzaje szablonów wyróżniające się poziomem skomplikowania ich budowy. Zależą one on elementu, któremu projektant chce przypisać dany szablon. Niektóre elementy takie jak np. przyciski, nie mają żadnych ograniczeń co do budowy szablonu. Można to traktować jako pewnego rodzaju przyzwolenie, że do jego budowy dla danego przycisku, programista może użyć dowolnych obiektów zdefiniowanych w platformie WPF, służących do projektowania kontrolek. Poniżej znajduje się przykład szablonu dla standardowego przycisku użytego w aplikacji. Jest to przycisk, którego zawartością jest wyłącznie tekst.

```
<ControlTemplate x:Key="DynamicMenuButtonForText" TargetType="Button">
    <Border Name="border" Background="#00A3B4"
            Width="{TemplateBinding Width}"
            Height="{TemplateBinding Height}"
            Padding="{TemplateBinding Padding}">
        <TextBlock Foreground="White"
                   Text="{TemplateBinding Property=Content}"
                   HorizontalAlignment="Center"
                   VerticalAlignment="Center"/>
    </Border>
    <ControlTemplate.Triggers>
        <Trigger Property="IsMouseOver" Value="true">
            <Setter TargetName="border" Property="Background" Value="#00BBD6"/>
        </Trigger>
        <Trigger Property="IsPressed" Value="true">
            <Setter TargetName="border" Property="Background" Value="#F26921"/>
        </Trigger>
        <Trigger Property="IsEnabled" Value="false">
            <Setter TargetName="border" Property="Background" Value="DarkGray"/>
        </Trigger>
    </ControlTemplate.Triggers>
</ControlTemplate>
```

Rysunek 22 Przykład szablonu dla przycisku

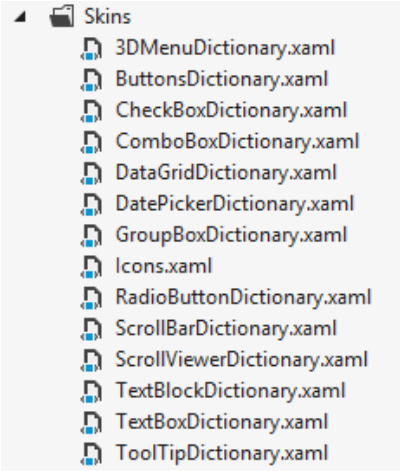
Szablon definiuje się używając znacznika „ControlTemplate”, gdzie w przeciwieństwie do stylu musi być zdefiniowany klucz. Właściwość „TargetType” określa nam typ elementu, do którego będzie przypisany szablon. Powyższy kod następująco zmienia wygląd przycisku:

- tłem przycisku jest element „Border”; jest to element posiadający atrybut „Name”, aby mógł być później użyty w sekcji triggerów; posiada stałe zdefiniowane jasno niebieskie tło; właściwości: „Width”, „Height” i „Padding” są powiązane za pomocą specjalnego rodzaju bindowania z właściwościami im odpowiednimi w strukturze kontrolki, które można ustawiać w stylu lub w samej kontrolce; ten mechanizm można traktować jak wystawienie punktów do modyfikacji na zewnątrz;
- tekst wyświetlany w przycisku znajduje się w elemencie typu „TextBlock”; tekst jest biały i wyśrodkowany; zawartość tekstu jest zbindowana do właściwości „Content”, przycisku, do którego powyższy szablon będzie dołączony;
- sekcja triggerów określa trzy podstawowe stany w jakich może znajdować się przycisk:
    - kursor myszy znajduje się nad elementem: zmiana koloru tła elementu „border” na jasno niebieski;
    - użytkownik naciska na przycisk lewym klawiszem myszy: zmiana koloru tła elementu „border” na jasno pomarańczowy;
    - element znajduje się w stanie nieaktywnym: zmiana koloru tła elementu „border” na ciemny szary.

Rysunek 23 Efekt zastosowania szablonu dla przycisku

9.1.4. Kontekst aplikacji

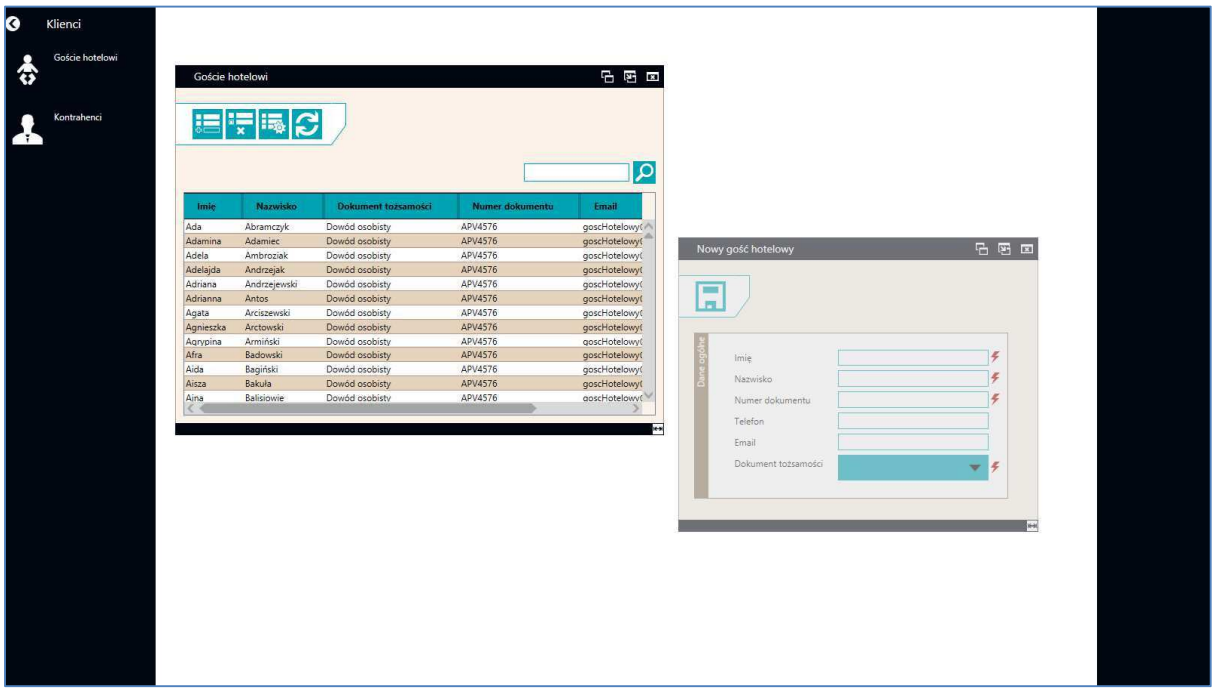
W przygotowanej aplikacji został przygotowany i użyty zestaw stylów i szablonów dla całej gamy kontrolek. Dzięki takiemu podejściu całkowicie zmieniono domyślny wygląd programu. Style i szablony dla każdej kontrolki zostały umieszczone w osobnym pliku o typie „ResourceDictionary”. Każdy z tych plików został podpięty do zasobów na poziomie aplikacji, dzięki czemu wszystkie style i szablony są dostępne w każdym miejscu programu. To podejście ma pewne konsekwencje, mianowicie w momencie zastosowania stylu domyślnego (bez klucza) przyjmują go wszystkie kontrolki o wskazanym przez styl typie w całym programie.



Rysunek 24 Katalog „Skins" w którym znajdują się wszystkie pliki typu „ResourceDictionary"

```
<Application.Resources>
    <!--Global View Model Locator-->
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="Skins/Icons.xaml" />
            <ResourceDictionary Source="Skins/ButtonsDictionary.xaml" />
            <ResourceDictionary Source="Skins/ScrollBarDictionary.xaml" />
            <ResourceDictionary Source="Skins/DataGridDictionary.xaml" />
            <ResourceDictionary Source="Skins/ToolTipDictionary.xaml" />
            <ResourceDictionary Source="Skins/GroupBoxDictionary.xaml" />
            <ResourceDictionary Source="Skins/TextBlockDictionary.xaml" />
            <ResourceDictionary Source="Skins/TextBoxDictionary.xaml" />
            <ResourceDictionary Source="Skins/ComboBoxDictionary.xaml" />
            <ResourceDictionary Source="Skins/ScrollViewerDictionary.xaml" />
            <ResourceDictionary Source="Skins/DatePickerDictionary.xaml" />
            <ResourceDictionary Source="Skins/3DMenuDictionary.xaml" />
            <ResourceDictionary Source="Skins/RadioButtonDictionary.xaml" />
            <ResourceDictionary Source="Skins/CheckBoxDictionary.xaml" />
        </ResourceDictionary.MergedDictionaries>
        <vm:ZarzadzanieHotelemLocator x:Key="ZHLocator" d:IsDataSource="True" />
        <vm:ViewModelLocator x:Key="BaseLocator" d:IsDataSource="True" />
        <vm:PlanHotelLocator x:Key="PHLocator" d:IsDataSource="True" />
        <vm:ReservationLocator x:Key="ReservationLocator" d:IsDataSource="True"/>
        <vm:DaneObiektuHotelowegoLocator x:Key="DaneObiektuLocator" d:IsDataSource="True"/>
    </ResourceDictionary>
</Application.Resources>
```

Rysunek 25 Podpięcie plików zawierających style i szablony na poziomie zasobów aplikacji



Rysunek 26 Efekt użycia styli i szablonów w aplikacji

9.2. Animacje

9.2.1. Wyjaśnienie teoretyczne

Platforma WPF posiada bardzo rozbudowany system animacji, dzięki któremu programista ma możliwość tworzenia efektów wizualnych, wzbogacających interfejs użytkownika. Animacja w technologii WPF polega na zmianie właściwości obiektu w czasie. Warunkiem jest, aby właściwość była „właściwością zależną" (ang. „Dependency Property"), która umożliwia użycie mechanizmów takich jak m. in.:

- bindowanie danych;
- animacje;
- style i szablony.

Właściwość musi być również typu umożliwiającego przeprowadzenie samego procesu animacji. Większość właściwości w kontrolkach dostarczonych z platformą .NET jest typu „Dependency Property". W związku z tym, programista ma szeroki wachlarz możliwości tworzenia efektów wizualnych związanych z interfejsem użytkownika. Animacje mogą być napisane w języku C# lub w kodzie znaczników XAML. Jeśli projektant zakłada pewną statyczną animację, rozsądnie będzie ją zadeklarować w kodzie XAML i w następnej kolejności programistycznie ją wywoływać. Zapis za pomocą znaczników jest bardziej elegancki i czytelny. Jeśli natomiast animacja bazuje na pewnych dynamicznie zmieniających się zmiennych, musi być stworzona i wywołana w języku C#.

Platforma WPF udostępnia kilka rodzajów animacji:

- podstawowe, określające jednolitą zmianę pewnej właściwości w czasie;
- oparte na klatkach, gdzie element może być poddany różnym animacjom w różnych przedziałach czasu;
- oparte na ścieżce ruchu;
- od źródła napisane przez programistę (najtrudniejsza technika).

Interfejs użytkownika nie powinien być przepełniony różnymi efektami specjalnymi. W projekcie zostały więc zastosowane głównie podstawowe animacje doskonale spełniające swoją rolę w tworzeniu prostych efektów wzbogacających działanie kontrolek.

9.2.2. Przykład animacji napisanej w kodzie XAML

W przygotowanym projekcie głównie wykorzystywane są dynamiczne animacje napisane w kodzie C#. Jest jednak kilka ich przykładów w kodzie XAML. Poniższa animacja dotyczy obrotu elementu o 36 stopni nie uwzględniając żadnych innych parametrów.

```
<UserControl.Resources>
    <Storyboard x:Key="NextStory">
        <DoubleAnimation Storyboard.TargetName="rotateY" Storyboard.TargetProperty="Angle" By="36" Duration="0:0:0.3"/>
    </Storyboard>
    <Storyboard x:Key="EarlierStory">
        <DoubleAnimation Storyboard.TargetName="rotateY" Storyboard.TargetProperty="Angle" By="-36" Duration="0:0:0.3"/>
    </Storyboard>
</UserControl.Resources>
```

Rysunek 27 Animacje w kodzie XAML

Powyżej zaprezentowano kod dwóch animacji, mających za zadanie obrót elementu w lewo lub w prawo. Ich kod jest niezwykle prosty i czytelny. Animowana zostaje właściwość „Angle" elementu o nazwie „rotateY", która jest typu „double". W związku z tym zostaje zastosowana animacja typu „DoubleAnimation". Właściwość „By" określa, o ile należy zmienić animowaną wartość od stanu aktualnego. Właściwość „Duration" określa czas trwania całego procesu. Animacje znajdują się w znacznikach „Storyboard", który jest specjalnie dla nich przygotowanym pojemnikiem. W kodzie powyżej każdy z pojemników został nazwany, aby można było go wywołać w kodzie code-behind.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    earlierStoryboard = this.Resources["EarlierStory"] as Storyboard;
    earlierStoryboard.Begin();
}
```

Rysunek 28 Przykład wywołania animacji za pomocą języka C# z kodu XAML

Wywołanie animacji z kodu XAML jest bardzo proste. Pierwszym krokiem jest pozyskanie jej obiektu ze zbioru zasobów, a następnie jej wykonanie za pomocą wywołania metody „Begin".

9.2.3. Przykład animacji w kodzie C#

Animacja do napisania w kodzie C# jest dużo trudniejsza do napisania niż w kodzie XAML, ale daje dużo większe możliwości. Programista może np. dynamicznie zmieniać pewne zachowanie elementu w zależności od czynności wybranej przez użytkownika oraz animować pojawiające się okienko w punkcie obliczonym dopiero podczas działania programu. Poniżej przykład animacji lewego paska menu w zakładce „Zarządzanie bazą". W zależności od

wyboru użytkownika, stary panel zanika ustępując miejsca nowemu. Wszystko odbywa się z lekkim przesunięciem w lewo. Powoduje to złudzenie napływania menu.

```
private void animateMenu(Panel from, Panel to)
{
    ThicknessAnimation lightTouchAnimation = new ThicknessAnimation
    {
        From = new Thickness(5, 0, -5, 0),
        To = new Thickness(0, 0, 0, 0),
        Duration = new Duration(TimeSpan.FromSeconds(0.3)),
        EasingFunction = new CircleEase()
    };
    DoubleAnimation fromOpacityAnimation = new DoubleAnimation
    {
        From = 1,
        To = 0,
        Duration = new Duration(TimeSpan.FromSeconds(0.2)),
        EasingFunction = new CircleEase()
    };
    DoubleAnimation toOpacityAnimation = new DoubleAnimation
    {
        From = 0,
        To = 1,
        Duration = new Duration(TimeSpan.FromSeconds(0.3)),
        EasingFunction = new CircleEase()
    };
    Storyboard fadeInStoryboard = new Storyboard();
    fadeInStoryboard.Children.Add(toOpacityAnimation);
    fadeInStoryboard.Children.Add(lightTouchAnimation);
    Storyboard.SetTarget(lightTouchAnimation, to);
    Storyboard.SetTargetProperty(lightTouchAnimation, new PropertyPath(Panel.MarginProperty));
    Storyboard.SetTarget(toOpacityAnimation, to);
    Storyboard.SetTargetProperty(toOpacityAnimation, new PropertyPath(Panel.OpacityProperty));
    fromOpacityAnimation.Completed += (o, e) =>
    {
        from.Visibility = System.Windows.Visibility.Hidden;
        to.Visibility = System.Windows.Visibility.Visible;
        activePanelMenu = to;
        fadeInStoryboard.Begin();
    };
    fadeInStoryboard.Completed += (o, e) => { enableButtonsInMenu(); };
    disableButtonsInMenu();
    from.BeginAnimation(Panel.OpacityProperty, fromOpacityAnimation);
}
```

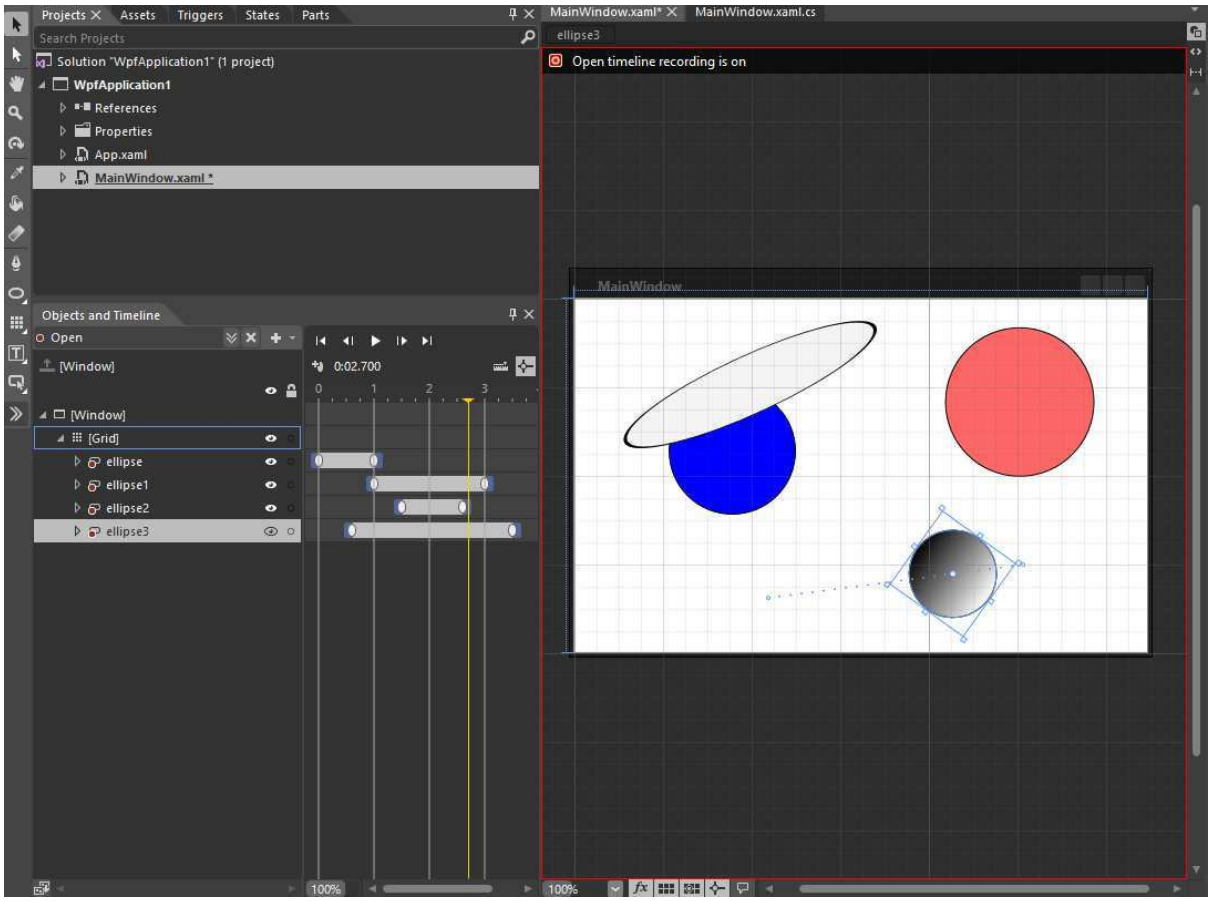
Rysunek 29 Kod animacji zmiany menu

Powyższa metoda zostaje wywoływana podczas wyboru nowej zakładki menu przez użytkownika. Za parametry przyjmuje ona stary panel do zastąpienia oraz nowy panel, który ma się pojawić po zaniknięciu starego panelu. Przygotowane zostały trzy obiekty animacji. Dwa z nich dotyczą zmiany przezroczystości elementu, pozostały odnosi się do zmiany marginesu zewnętrznego. Przezroczystość będzie odpowiadać za zanikanie starego i pojawienie się nowego panelu, z kolei zmiana marginesu spowoduje złudzenie ruchu wpływania nowego menu. Wszystkie obiekty animacji są skonfigurowane na zasadzie „From” – „To”. Można to traktować jak sztywne przejście „od” pewnej wartości „do” pewnej wartości. Każda z zadeklarowanych animacji dzieje się szybko w czasie od 0.2 do 0.3 sekundy.

Animację można wywołać na różne sposoby. Powyższy kod prezentuje dwa z nich. Pierwszy sposób wywołuje grupę animacji, dotyczących panelu, który ma się pojawić. Należy stworzyć obiekt typu „Storyboard" i dodać do niego przygotowane animacje. Następnie za pomocą statycznych metod „SetTarget" oraz „SetTargetProperty" należy ustalić, jakie właściwości, jakiego obiektu będą animowane. Po zakończeniu procesu inicjalizacji wywołuje się metodę „Begin" rozpoczynająca animację. Drugi sposób dotyczy pojedynczej animacji. W powyższym przypadku jest to animacja odpowiadająca za zanikanie starego panelu. Wówczas nie trzeba tworzyć obiektu „Storyboard". Wystarczy wywołać metodę „BeginAnimation" na rzecz animowanego obiektu. Parametrami tej metody jest właściwość obiektu, jaka będzie animowana oraz przygotowany obiekt animacji.

9.2.4. Skomplikowane statyczne animacje

W przypadku tworzenia skomplikowanych statycznych animacji programista ma do dyspozycji potężne narzędzie o nazwie „Expression Blend". Za pomocą tego programu można w sposób wizualny przygotowywać bardzo złożone animacje. Program posiada specjalistyczny zestaw narzędzi ułatwiający pracę. Większość operacji odbywa się na zasadzie „przeciągnij i upuść" albo „narysuj i nadaj odpowiednie właściwości". Program zamienia wykonane czynności na kod XAML, który można po zakończeniu prac importować do własnego projektu.



Rysunek 30 Screen programu Expression Blend z narzędziami do tworzenia animacji

9.3. Interfejs 3D

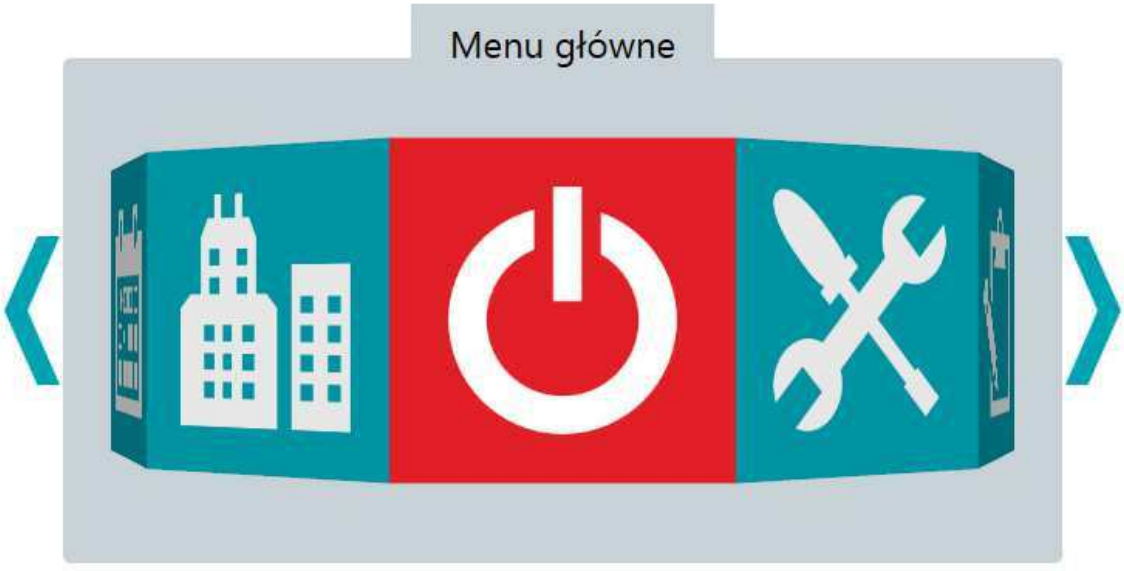
Platforma WPF umożliwia tworzenie interfejsów użytkownika zorientowanych w przestrzeni trójwymiarowej. Takie podejście daje bardzo wiele możliwości w dziedzinie prezentacji danych użytkownikowi.

9.3.1. Menu 3D

W przygotowanej aplikacji w technologii 3D zostało wykonane menu główne, znajdujące się na osobnej warstwie, które przełącza użytkownika pomiędzy modułami programu:

- rezerwacji;
- widoku architektonicznego;
- zarządzania bazą danych;
- ustawień obiektu.

Menu główne umożliwia również zamknięcie programu. Jest ono aktywowane i zamykane poprzez naciśnięcie prawego klawisza myszy w każdym miejscu programu.



Rysunek 31 Główne menu wykonane w technologii 3D

9.3.2. Schemat

Wsparcie oferowane przez platformę w budowaniu interfejsów 3D jest bardzo bazowe. Nie ma żadnych gotowych komponentów wykorzystujących technologię „przeciągnij i upuść". Każdy element interfejsu należy zaprojektować samemu, korzystając z bazowych

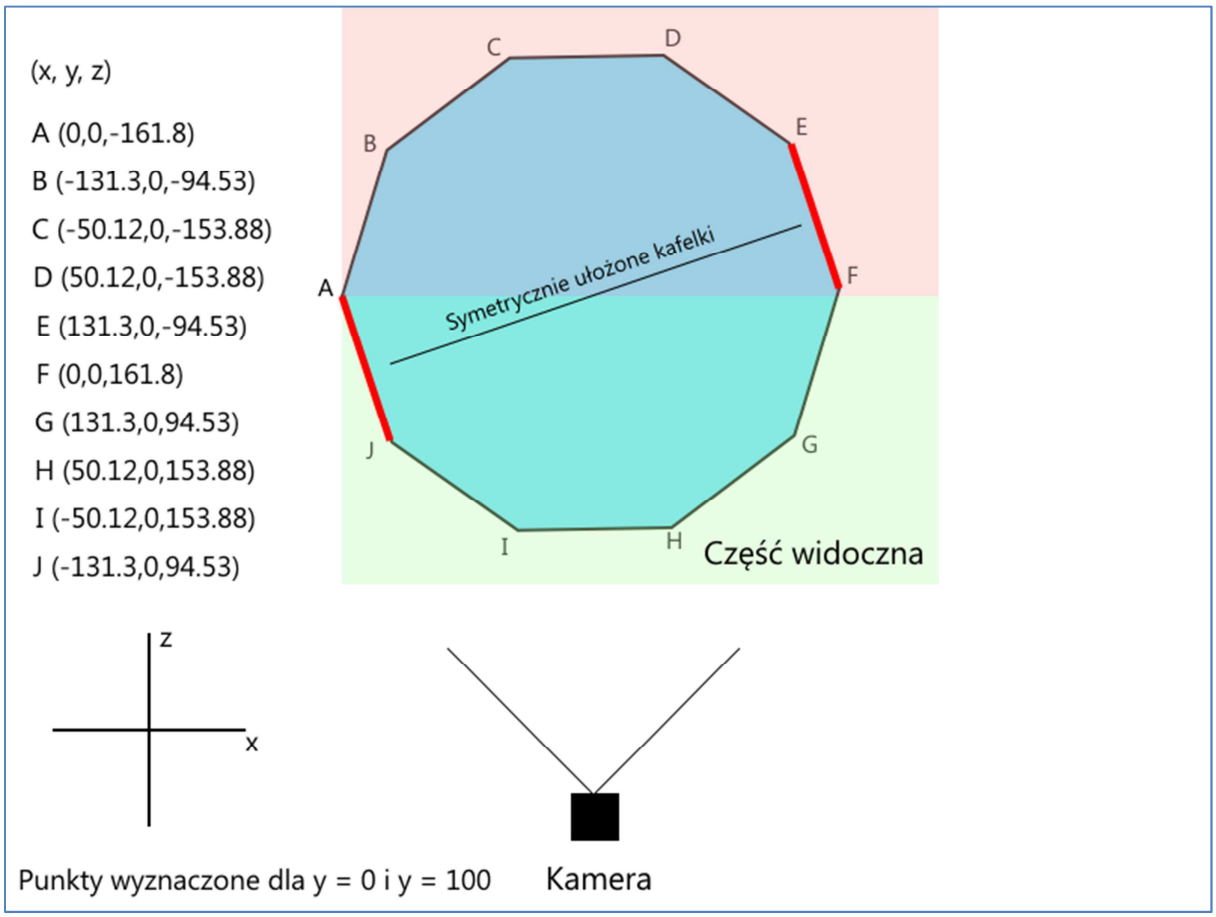
funkcjonalności typu: wyznaczanie trójkątów w przestrzeni, nakładanie tekstur, umieszczenie kamery w przestrzeni i obliczenie wielkości sceny, na której model 3D będzie prezentowany.

Rozpoczynając pracę nad menu 3D, należało zastanowić się nad jego celem i możliwościami jakie ma oferować. Jak zostało wspomniane wyżej, główne menu służy przełączaniu pomiędzy głównymi modułami aplikacji i oferuje możliwość wyłączenia programu. W założeniu, projektowane menu nie było otwarte na rozbudowę o kolejne kafelki. Otrzymano więc pięć funkcjonalności, które główne menu ma dostarczać: przejście do czterech głównych modułów aplikacji oraz możliwość zamknięcia programu. Należało skonstruować obiekt z pięcioma składnikami, który powinien być atrakcyjny wizualnie i łatwy w obsłudze dla użytkownika. Ważne, aby kafelki menu były zawsze widoczne, ponieważ występują w niewielkiej ilości. Aktualnie wybrany kafelek powinien być możliwie najlepiej zaprezentowany użytkownikowi.

Taki funkcjonalny opis bardzo dobrze spełnia prostopadłościan oparty na podstawie dziesięciokąta foremnego. Bryła jest widoczna tylko z przodu, więc użytkownik dostrzega pięć kafelków. Aktualnie wybrany kafelek jest widoczny najlepiej, ponieważ nie znajduje się pod kątem. Po bokach bryły umieszczono dwa przyciski umożliwiające jej obrót. Kafelki są umieszczone na ścianach symetrycznie, dlatego podczas obrotu, kiedy z jednej strony jeden z nich znika, po drugiej stronie pojawia się jego odpowiednik. Dzięki takiemu mechanizmowi użytkownik stale widzi możliwe do wyboru wszystkie przyciski głównego menu. Każdy kafelek posiada dynamiczna teksturę, zmieniającą się podczas najechania na niego kursorem myszy. Posiada również akcję wykonywaną w momencie kliknięcia na nim przez użytkownika. Cały skomplikowany proces tzw. hit-testingu obsłużony jest za pomocą specjalnych klas udostępnionych przez platformę WPF.

Pierwszą fazą zaprojektowanego menu w technologii 3D było schematyczne nakreślenie działania oraz obliczenie wszystkich wierzchołków bryły i punktu położenia kamery.

Istnieją narzędzia graficzne umożliwiające projektowanie modeli 3D i konwertowanie ich do kodu XAML. Te narzędzia nie występują jednak w wersji darmowej, dlatego każdy programista chcący coś zaprojektować w przestrzeni trójwymiarowej, musi włożyć spory nakład pracy i wszystko wykonać ręcznie. Poniżej przedstawiono schematyczny rysunek rozmieszczenia punktów bryły i kamery w przestrzeni. Taki rysunek nie jest niezbędny, jednak bardzo pomaga wyobrazić sobie położenie trójkątów na których bazują modele 3D.



Rysunek 32 Schemat obrazujący położenie punktów bryły w przestrzeni 3D

9.3.3. Wykonanie modelu w kodzie XAML

Cały projekt należy przenieść do kody XAML. Platforma WPF udostępnia specjalną kontrolkę, która służy do renderowania elementów 3D. Kontrola ta nosi nazwę „Viewport3D" i służy do odrysowania całej sceny 3D przygotowanej przez programistę. Posiada również właściwość nazywającą się „Camera" i odpowiada za umieszczenie w odpowiednim miejscu kamery. W WPF-ie mamy do dyspozycji dwa podstawowe rodzaje kamer:

- perspektywistyczną
- ortogonalną

W projekcie została użyta kamera perspektywistyczna, która powoduje złudzenie oddalenia się krawędzi obiektu. Obraz jaki powstaje przy użyciu tego typu kamery nie jest płaski i nie posiada krawędzi prostopadłych, jak w przypadku użycia kamery ortogonalnej. W elemencie typu „Viewport3D" muszą znajdować się elementy odpowiadające za model lub grupę modeli 3D. W przypadku projektowanego menu będą to elementy typu "ModelVisual3D", które będą zawierały w sobie modele 3D ścian bryły oraz światło. Każda ściana bryły definiowana jest w

znaczniku „GeometryModel3D”, w którym można zadeklarować punkty przestrzeni trójwymiarowej wchodzące w skład kształtu oraz połączenia tych punktów w trójkąty. Bardzo istotna jest kolejność łączenia punktów, aby kształt nie miał pomylonego frontu z tyłem. Cały proces budowy kody zaprezentowany jest na obrazku poniżej.

```
<Viewport3D Name="viewport3DROOT" Width="640" Height="320">
    <ModelVisual3D>
        <ModelVisual3D.Content>
            <Model3DGroup>
                <GeometryModel3D x:Name="btnPower1" Material="{StaticResource powerNormal}"
                                 BackMaterial="{StaticResource back}">
                    <GeometryModel3D.Geometry>
                        <MeshGeometry3D TextureCoordinates="0 1, 1 1, 0 0
                                                            1 1, 1 0, 0 0"
                        Positions="-50.12 0 153.88, 50.12 0 153.88, -50.12 100 153.88
                                   50.12 0 153.88, 50.12 100 153.88, -50.12 100 153.88"
                        TriangleIndices="0 1 2, 3 4 5"/>
                    </GeometryModel3D.Geometry>
                </GeometryModel3D>
                <GeometryModel3D x:Name="btnSettings1" Material="{StaticResource settingsNormal}" BackMaterial="{Stat
                <GeometryModel3D x:Name="btnHotelPanel1" Material="{StaticResource hotelPanelNormal}" BackMaterial="
                <GeometryModel3D x:Name="btnHotelArch1" Material="{StaticResource hotelArchNormal}" BackMaterial="{St
                <GeometryModel3D x:Name="btnHotelReception1" Material="{StaticResource hotelReceptionNormal}" BackMa
                <GeometryModel3D x:Name="btnPower2" Material="{StaticResource hotelPanelNormal}" BackMaterial="{Stat
                <GeometryModel3D x:Name="btnSettings2" Material="{StaticResource settingsNormal}" BackMaterial="{Stat
                <GeometryModel3D x:Name="btnHotelPanel2" Material="{StaticResource powerNormal}" BackMaterial="{Stat
                <GeometryModel3D x:Name="btnHotelArch2" Material="{StaticResource hotelArchNormal}" BackMaterial="{St
                <GeometryModel3D x:Name="btnHotelReception2" Material="{StaticResource hotelReceptionNormal}" BackMa
                <Model3DGroup.Transform...>
            </Model3DGroup>
        </ModelVisual3D.Content>
    </ModelVisual3D>
    <ModelVisual3D>
        <ModelVisual3D.Content>
            <Model3DGroup>
                <AmbientLight Color="Gray"/>
                <DirectionalLight Color="Gray" Direction="0 0 -1"/>
            </Model3DGroup>
        </ModelVisual3D.Content>
    </ModelVisual3D>
    <ModelVisual3D...>
    <Viewport3D.Camera>
        <PerspectiveCamera Position="0 50 800" LookDirection="0 0 -1" UpDirection="0 1 0" FieldOfView="25"/>
    </Viewport3D.Camera>
</Viewport3D>
```

Rysunek 33 Kod XAML sceny 3D

W powyższym kodzie widoczne jest opisane wcześniej hierarchiczne ułożenie elementów. Jednostką, która zawiera dane do wyświetlenia jest „GeometryModel3D”. Każdy obiekt tego typu został nazwany, aby był dostępny w kodzie code-behind w celu przeprowadzenia hit-testingu na reakcję użytkownika. „GeometryModel3D” definiuje materiał, jakim pokrywany jest kształt od frontu oraz widziany od tyłu. Dla każdego kafelka zostały przygotowane specjalne tekstury z ikoną odpowiadającą jego działaniu. Definiuje również właściwość „Geometry”, która przyjmuje obiekt typu „MeshGeometry3D”, gdzie można podać koordynaty kształtu w przestrzeni, punkty łączeń trójkątów, a także koordynaty tekstury.

W scenie zdefiniowane jest również światło stałe oraz kierunkowe. Pierwsze z nich powoduje jednakowe oświetlenie wszystkich kształtów, natomiast drugie nadaje efekt cienia dla

kształtów, które znajdują się pod kątem. Poprzez to model 3D wydaje się bardziej realistyczny.

Kamera została zdefiniowana na sam koniec kodu. Jej ułożenie decyduje o tym, z jakiej strony cała scena będzie widziana przez użytkownika. Bardzo istotne jest takie rozmieszczenie kamery, aby swoim zasięgiem obejmowała całą scenę. Nie można pozwolić na ucięcie jakiegokolwiek elementu wyświetlanego użytkownikowi.

9.3.4. Hit testing

Aby menu było w pełni funkcjonalne musi wchodzić w interakcję z użytkownikiem. Warto udostępnić użytkownikowi funkcjonalność klikania w kafelki i podejmowania pewnej akcji z kliknięty kafelkiem związanej. Kafelki wyglądają jak przyciski, jednak w rzeczywistości nimi nie są. Programista nie ma więc wystawionego zdarzenia umożliwiającego reakcję na kliknięcie w element. Cały ten proces należy przeprowadzić za pomocą klas udostępnionych przez platformę .NET służących do walidacji interakcji w przestrzeni 3D. Użytkownik klika w określonym punkcie w obszarze elementu „Viewport3D" i powinno się oszacować czy ten punkt nie znajdował się na jakimś elemencie, który w założeniu miał być interaktywny. Jeśli punkt znajdował się w takim miejscu to powinno się sprawdzić jaki to element i podjąć akcję z nim związaną.

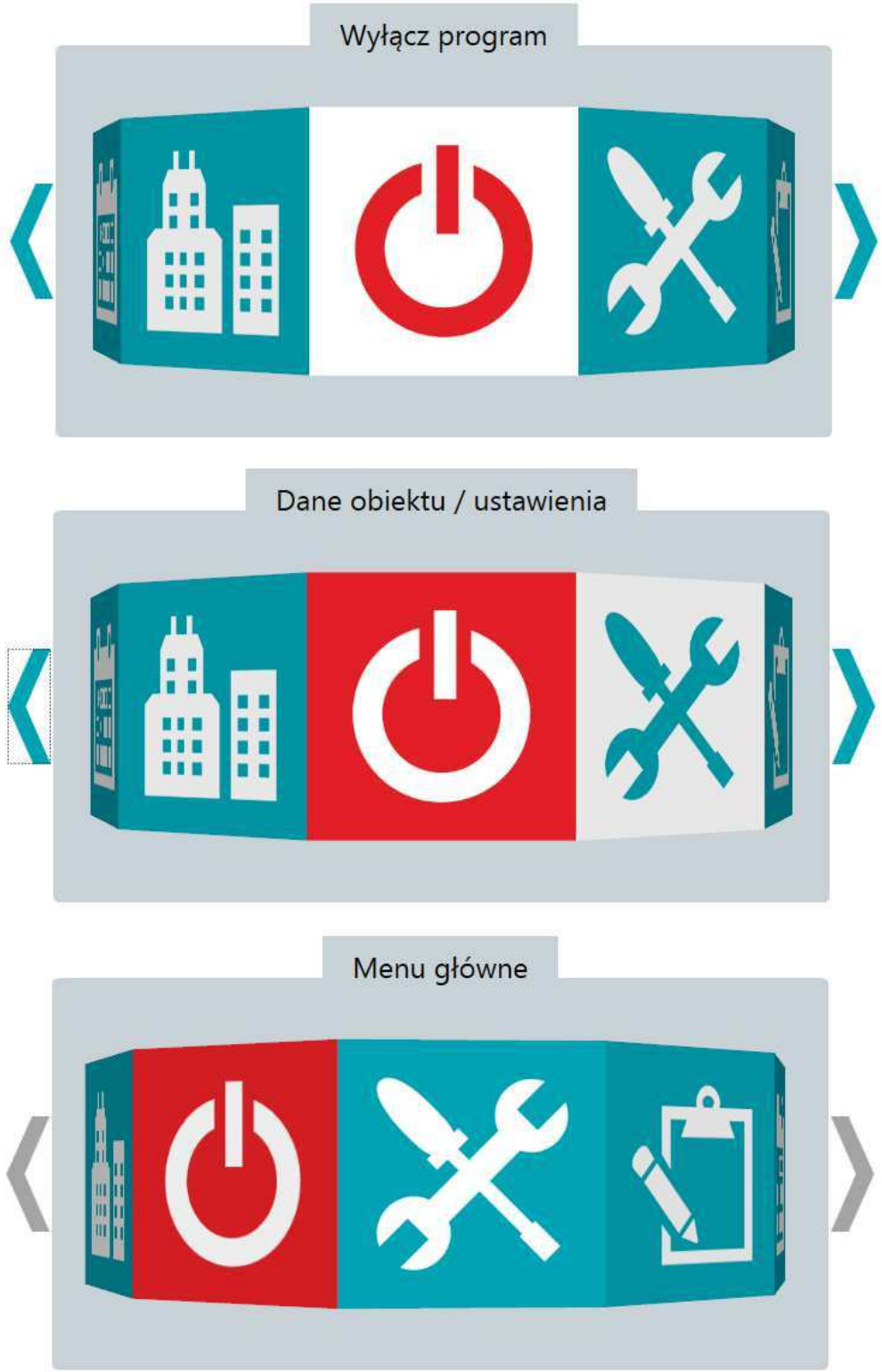
Algorytm działania zastosowanej w projekcie walidacji polega na oszacowaniu położenia kursora myszy przy każdym jego przesunięciu względem obiektu „Viewport3D". Jeśli zostanie wykryta kolizja z elementem interaktywnym, do pewnej zmiennej zostaje zapisany jego identyfikator. Jeśli użytkownik zdecyduje się na kliknięcie myszką w tym miejscu, zostanie wysłany komunikat o podjęciu akcji związanej z klikanym kafelkiem. Po każdym sprawdzeniu walidacji zostają sprawdzone zmienne z tablicy, w której trzymane są dane o stanie każdego z kafelków. Jeśli zostanie wykryta zmiana w tablicy, wówczas wpłynie ona na wygląd przycisku o odpowiednim indeksie.

```
//akcja związana z poszczególnym kafelkiem
for (int i = 0; i < 10; i++)
{
    if (resultMesh.ModelHit == tabButton[i])
        isMouseOver[i] = true;
    else
        isMouseOver[i] = false;
}
//porównanie wartości w tablicach boolowskich i wyrenderowanie odpowiedniej textury
selectedIndex = -1;
LabelDisplayTitle.Content = "Menu główne";
for (int i = 0; i < 10; i++)
{
    if (isMouseOver[i])
    {
        selectedIndex = i;
        LabelDisplayTitle.Content = tabDisplayTitle[i];
    }
    if (isMouseOver[i] != isMouseOverPreviev[i])
    {
        if (isMouseOver[i])
        {
            tabButton[i].Material = (DiffuseMaterial)Application.Current.Resources[tabNameOfTexturasMouseOver[i]];
        }
        else
        {
            tabButton[i].Material = (DiffuseMaterial)Application.Current.Resources[tabNameOfTexturasNormal[i]];
        }
    }
    isMouseOverPreviev[i] = isMouseOver[i];
}
```

Rysunek 34 Część kodu odpowiedzialnego za przeprowadzenie hit testingu

Wszystkie obiekty typu „GeometryModel3D” odpowiedzialne za wyrenderowanie kafelków zostają za pomocą nadanej im nazwy odnalezione i zapisane do tablicy „tabButton” w konstruktorze. Właściwość „ModelHit” zmiennej „resultMesh” przechowuje model sceny, na który wskazuje kursor myszy. W pętli zostaje on porównany z wszystkimi elementami interaktywnymi w menu. Jeśli okaże się, że referencje na obiekt są zgodne wówczas w tablicy typu logicznego zostaje odnotowane, że element o danym indeksie powinien przejść w stan tzw. „mouse over”. Następnie porównuje się stan tablic sprzed i po akcji i jeśli zmienne spod tego samego indeksu różnią się, wykonywane jest odpowiednie działanie.

9.3.5. Prezentacja głównego menu



Rysunek 35 Główne menu podczas różnych interakcji. Kolejno od góry: zaznaczenie kafelka wyłączającego program, zaznaczenie kafelka zmiany modułu, menu podczas animacji obrotu

9.4. Widok architektoniczny

9.4.1. Problematyka

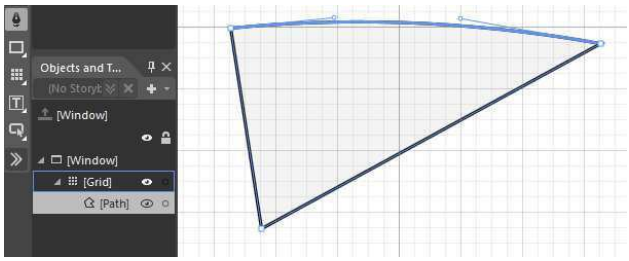
Jednym z czterech głównych modułów aplikacji jest widok architektoniczny hotelu, który przedstawia przekrój budynku na każdym piętrze, umożliwiający wgląd w rozmieszczenie pokoi. Po kliknięciu na pokój wyświetla się informacja o gościach aktualnie go zamieszkujących. Zakłada się, że nie można edytować stanu pokoi w hotelu, który jest ściśle powiązany z systemem.

Funkcjonalność wydaje się prosta, jednak dla programisty może okazać się ona kłopotliwa. Należy zaprojektować widok podobny do rysunku architektonicznego, który ma być estetyczny i prosty. Rozwiązaniem może być zastosowanie grafiki i wyznaczenie w niej pewnych aktywnych obszarów oznaczających pokoje. Po kliknięciu przez użytkownika w pewne miejsce grafiki należałoby zbadać, czy został kliknięty obszar oznaczający pokój czy nie. Rozwiązanie to jest bardzo skomplikowane i mało efektywne. Zwykła grafika nie będzie odporna na zmiany rozdzielczości na ekranie użytkownika. Bardziej estetyczne rozwiązanie opierałoby się o grafikę wektorową. Pokój powinien być kontrolką przechowującą pewne informacje i wyrenderowaną w odpowiedni sposób. Dzięki temu zostanie odseparowana logika pokoju od całej logiki wyświetlania widoku. Pomysł ten jest o wiele lepszy od poprzedniego. Powstaje jednak pewien istotny problem, jak stworzyć kontrolę, która na bazie punktów w układzie 2D sama określi swój kształt i będzie łatwa w implementacji. Żaden programista nie chce ręcznie obliczać i wprowadzać wszystkich punktów. Bardzo przydatna byłaby metoda graficzna dzięki, której można byłoby ręcznie narysować dany kształt i nazwać go kontrolką. Narzędziem dla grafików w technologii WPF jest ExpressionBlend. Jednak ten program nie posiada narzędzia mogącego rysować dowolne kontrolki, które nie zostały jeszcze wymyślone w momencie jego stworzenia. Ten program jest jednak na tyle pomocny, że przy niedużej ilości pomysłowości bardzo łatwo można przedstawiony wyżej problem rozwiązać.

9.4.2. Stworzenie widoku architektonicznego

W programie ExpressionBlend zostało zaprojektowanych wiele narzędzi do rysowania grafiki wektorowej. Jednym z najpotężniejszych jest pióro dające możliwość wykreowania dowolnego kształtu. Kształt ten w pliku XAML jest zapisany jako obiekt klasy „Path”, która wywodzi się ze specjalnej linii elementów technologii WPF nazywanej kształtami. Kształty to elementy o bardzo lekkiej konstrukcji. Bardzo często bywają częściami bardziej

skomplikowanych kontrolek. Kształt typu „Path” jest z nich wszystkich najbardziej zaawansowany. Przechowuje kolekcję punktów, na bazie których wyrenderowany jest element. Takich elementów z reguły się nie pisze, ale używa programu ExpressionBlend, aby je narysować.



Rysunek 36 Część widoku programu ExpressionBlend z aktywnym narzędziem pióro

```
<Grid>
    <Path Data="M105,80 L115.00425,145.00425 225.00425,85.004246 C179.45221,76.868332 138.5043,76.590293 105,80 z"
    Fill="#FFF4F4F5" HorizontalAlignment="Left" Height="68.065" Margin="105,77.939,0,0" Stretch="Fill" Stroke="Black"
    VerticalAlignment="Top" Width="121.004"/>
</Grid>
```

Rysunek 37 Kod XAML wygenerowany przez program ExpressionBlend

Jak widać jest to bardzo prosta metoda rysowania dowolnie skomplikowanych kształtów. Można też zauważyć podobieństwo do opisanego wcześniej problemu rysowania kontrolki na bazie punktów.

Kolejnym krokiem w przygotowanie widoku będzie stworzenie kontrolki reprezentującej pokój. Kontrolka zostanie stworzona na bazie klasy „UserControl”. Do kodu klasy będzie dodana specjalna właściwość odpowiedzialna za przechowanie kolekcji punktów określających kształt kontrolki.

```
public partial class PokojPlanView : UserControl
{
    public PokojPlanView()
    {
        InitializeComponent();
    }

    public Geometry Points
    {
        get { return (Geometry)GetValue(PointsProperty); }
        set { SetValue(PointsProperty, value); }
    }

    public static readonly DependencyProperty PointsProperty =
        DependencyProperty.Register("Points", typeof(Geometry), typeof(PokojPlanView), new PropertyMetadata(null));
}
```

Rysunek 38 Kod klasy reprezentującej pokój w widoku architektonicznym

Właściwość jest typu „DependencyProperty” ponieważ istotny jest mechanizm bindowania danych. Punkty, na bazie których wygeneruje się kształt będą dostarczone z zewnątrz i podłączone do właściwości ”Points” typu „Geometry”. Typ „Geometry” jest wymagany, ponieważ reprezentuje on kolekcję punktów deklarowanych w języku XAML. Plik typu „UserControl” składa się również z pliku XAML określającego wygląd kontrolki. Poniższy kod XAML jest kompilowany w metodzie wywoływanej w domyślnym konstruktorze.

```
<Canvas>
    <Button Command="{Binding EditCommand}">
        <Button.Template>
            <ControlTemplate TargetType="Button">
                <Grid>
                    <Path Name="pathRoom" Data="{Binding ElementName=ROOT, Path=Points}"
                          Stretch="Fill" Stroke="Black" StrokeThickness="1"
                          Height="{Binding ElementName=ROOT, Path=Height}"
                          Width="{Binding ElementName=ROOT, Path=Width}"/>
                    <Path Name="pathRoomBack" Data="{Binding ElementName=ROOT, Path=Points}"
                          Fill="Transparent" Stretch="Fill" Height="{Binding ElementName=ROOT, Path=Height}"
                          Width="{Binding ElementName=ROOT, Path=Width}"/>
                    <TextBlock Text="{Binding Path=NumerPokoju}" Canvas.Top="20" Canvas.Left="20"
                               HorizontalAlignment="Center" VerticalAlignment="Center"/>
                </Grid>
                <ControlTemplate.Triggers>
                    <Trigger Property="IsMouseOver" Value="true">
                        <Setter TargetName="pathRoomBack" Property="Fill" Value="#40FFFFFF"/>
                    </Trigger>
                    <Trigger Property="IsPressed" Value="true">
                        <Setter TargetName="pathRoomBack" Property="Fill" Value="#80FFFFFF"/>
                    </Trigger>
                    <DataTrigger Binding="{Binding Stan}" Value="0">
                        <Setter TargetName="pathRoom" Property="Fill" Value="#C7B29D"/>
                    </DataTrigger>
                    <DataTrigger Binding="{Binding Stan}" Value="1">
                        <Setter TargetName="pathRoom" Property="Fill" Value="#799754"/>
                    </DataTrigger>
                    <DataTrigger Binding="{Binding Stan}" Value="2">
                        <Setter TargetName="pathRoom" Property="Fill" Value="#00A3B4"/>
                    </DataTrigger>
                    <DataTrigger Binding="{Binding Stan}" Value="3">
                        <Setter TargetName="pathRoom" Property="Fill" Value="#E21F26"/>
                    </DataTrigger>
                </ControlTemplate.Triggers>
            </ControlTemplate>
        </Button.Template>
    </Button>
</Canvas>
```

Rysunek 39 Kod XAML kontrolki reprezentującej pokój w widoku architektonicznym

Wygląd kontrolki składa się z przycisku, do którego podłączona jest pewna komenda, która powinna znajdować się w obiekcie ukrytym we właściwości „DataContext” kontrolki. Tym obiektem jest ViewModel dostarczony do kontrolki w późniejszym etapie projektowania. Wygląd przycisku został całkowicie zmieniony. Składa się on z omawianych wcześniej elementów „Path”, których kolekcja punktów podbindowana jest do właściwości „Points”.Oba elementy typu „Path” będą odpowiadać kształtowi, jaki zostanie przypisany pokojowi. Elementy typu „TextBlock” będzie wyświetlał nazwę pokoju. W szablonie zastosowano dwa typy triggerów. Pierwszy rodzaj to triggery odpowiedzialne za stan kontrolki względem jej domyślnego zachowania. Jeśli kursor muszy znajdzie się nad

elementem lub użytkownik kliknie element, zmieni się tło drugiego z elementów „Path”. To zachowanie oznacza, że kontrolka jest interaktywna. Drugi rodzaj triggerów o nazwie „DataTrigger” pozwala podejmować akcję względem podbindowanych danych do kontrolki. Tło pierwszego z elementów „Path” będzie się zmieniało względem właściwości „Stan”, która określa czy pokój jest zajęty. Np. cyfra zero oznacza, że pokój jest wolny. Kontrolka będzie na tyle domyślna, aby w zależności od swojego stanu zmienić swój wygląd.

Można teraz zauważyć podobieństwo wygenerowanego w ExpressionBlend elementu „Path” i zaprojektowanej kontrolki „PokojPlanView”. Kluczowym elementem obu klas jest właściwość z danymi punktów, które określają jak element ma wyglądać. Rozwiązaniem jest narysowanie całego planu hotelowego w programie ExpressionBlend za pomocą elementów „Path”. Następnie przy wykorzystaniu edytora tekstu należy pozamieniać niektóre nazwy właściwości, aby odpowiadały kontrolce „PokojPlanView”.

```
<Path Data="M-5,0 L139.5,0.5 139.5,69.5 -5,70 z" Height="72.333" Canvas.Top="150" Width="145.5"/>
<Path Data="M-5,0 L139.5,0.5 139.5,69.5 -5,70 z" Height="73.875" Canvas.Top="224" Width="145.5" />
<Path Data="M-5,0 L139.5,0.5 139.5,69.5 -5,70 z" Height="63.875" Canvas.Top="299.375" Width="145.5" />
<Path Data="M-5,0 L139.5,0.5 139.5,69.5 -5,70 z" Height="62.875" Canvas.Top="365.5" Width="145.5" />
<Path Data="M-5,0 L139.5,0.5 139.5,69.5 -5,70 z" Height="75" Canvas.Top="430.5" Width="145.5" />
```

Rysunek 40 Kod XAML przed zamianą

```
<local:PokojPlanView DataContext="{Binding [P01_1]}" Points="M-5,0 L139.5,0.5 139.5,69.5 -5,70 z"
                     Height="72.333" Canvas.Top="150" Width="145.5"/>
<local:PokojPlanView DataContext="{Binding [P02_1]}" Points="M-5,0 L139.5,0.5 139.5,69.5 -5,70 z"
                     Height="73.875" Canvas.Top="224" Width="145.5" />
<local:PokojPlanView DataContext="{Binding [P03_1]}" Points="M-5,0 L139.5,0.5 139.5,69.5 -5,70 z"
                     Height="63.875" Canvas.Top="299.375" Width="145.5" />
<local:PokojPlanView DataContext="{Binding [P04_1]}" Points="M-5,0 L139.5,0.5 139.5,69.5 -5,70 z"
                     Height="62.875" Canvas.Top="365.5" Width="145.5" />
<local:PokojPlanView DataContext="{Binding [P05_1]}" Points="M-5,0 L139.5,0.5 139.5,69.5 -5,70 z"
                     Height="75" Canvas.Top="430.5" Width="145.5" />
```

Rysunek 41 Kod XAML po zamianie

Zamieniono ze sobą następujące elementy:

- „Path” na „local:PokojPlanView”;
- „Data” na „Points”.

Całą operację można bardzo szybko wykonać przy pomocy narzędzia w VisualStudio, które nazywa się „Znajdź i zamień”. W ten sposób do właściwości „Points” zostały podbinowane dane punktów, które z kolei będą odrysowane w kontrole wewnątrz przycisku.

Wszystkie elementy zostały umieszczone w panelu o nazwie „Canvas", który układa znajdujące się w nim elementy względem parametrów kartezjańskich. Jedyna różnica jest taka, że dodanie wartości osi OY skierowane są w dół. „Canvas" jest najmniej skomplikowanym i kosztującym operacji procesora panelem dostarczanym wraz z platformą WPF. Panel i elementy znajdujące się w nim posiadają na stałe ustalone rozmiary. Jednak elementy te są reprezentowane poprzez grafikę wektorową i podczas skalowania nie tracą one na jakości.

Kontrola reprezentująca widok architektoniczny wyświetla się, aczkolwiek nie pokazuje żadnych informacji i nie można z nią dokonać interakcji. Należy dostarczyć do niej klasę typu ViewModel.

9.4.3. Tworzenie ViewModelu

Klasa „PokojPlanView" wymaga ViewModelu dostarczającego informacji o stanie pokoju i komendy związanej z kliknięciem. Poniżej znajduje się kod klasy „PokojPlanViewModel", która spełnia te wymogi.

```
public class PokojPlanViewModel
{
    public int Id { get; set; }
    public string NumerPokoju { get; set; }
    public string StandardNazwa { get; set; }
    public int Stan { get; set; }
    public string KodPlan { get; set; }

    private RelayCommand editCommand;
    public RelayCommand EditCommand
    {
        get...
    }
}
```

Rysunek 42 Kod klasy ViewModelu do widoku architektonicznego pokoju

Widok architektoniczny to zbiór pokoi, dlatego ViewModel dla takiego widoku powinien być kolekcją odpowiednio zainicjalizowanych obiektów typu „PokojPlanViewModel".

```
public class HotelPlanViewModel : Dictionary<string, PokojPlanViewModel>
{
    private HotelDbContext db = new HotelDbContext();
    private int pietro;
    public HotelPlanViewModel(int pietro)
    {
        this.pietro = pietro;
        var kolekcjaPokoi = from pokoj in db.Pokoje
                            where pokoj.Pietro == pietro && pokoj.Dostepnosc
                            select pokoj;
        int stan = 0;
        int index = 0;
        foreach (var pokoj in kolekcjaPokoi)
        {
            DateTime dataTerazniejsza = GeneralHelper.GetSimplyActualDate();
            var czyRezerwacja = db.OperacjeHotelowe_Pokoje.Where(poh => poh.IdPokoj ==
                pokoj.IdPokoj && pokoj.Dostepnosc && poh.OperacjaHotelowa.DataRozpoczeciaPobytu <= dataTerazniejsza &&
                poh.OperacjaHotelowa.DataZakonczeniaPobytu >= dataTerazniejsza).Any();
            PokojPlanViewModel model = new PokojPlanViewModel();
            model.Id = pokoj.IdPokoj;
            model.NumerPokoju = pokoj.NumerPokoju;
            model.StandardNazwa = pokoj.StandardPokoju.Nazwa;
            model.KodPlan = pokoj.KodPlan;
            if (czyRezerwacja)
            {
                stan = (index % 5 == 0) ? 1 : 2;
            }
            else
            {
                stan = 0;
            }
            model.Stan = stan;
            index++;
            this.Add(model.KodPlan, model);
        }
    }
}
```

Rysunek 43 Kod klasy ViewModelu dla całego widoku architektonicznego

Klasa „HotelPlanViewModel” dziedziczy po sparametryzowanej klasie „Dictionary”, która jest słownikiem przechowującym elementy na zasadzie klucza i wartości. Kluczem w słowniku jest specjalny kod umożliwiający łatwe podbindowane elementów w widoku. Kod ten jest wartością pobieraną z bazy danych i jest indywidualny dla każdego pokoju. Wartością jest odpowiedni obiekt klasy „PokojPlanViewModel”.

Tak przygotowany obiekt ViewModel można dostarczyć do widoku za pomocą mechanizmu zasobów. Następnie dla każdej kontrolki „PokojPlanView” należy podbindować jej właściwość „DataContext” do odpowiedniego obiektu w słowniku.

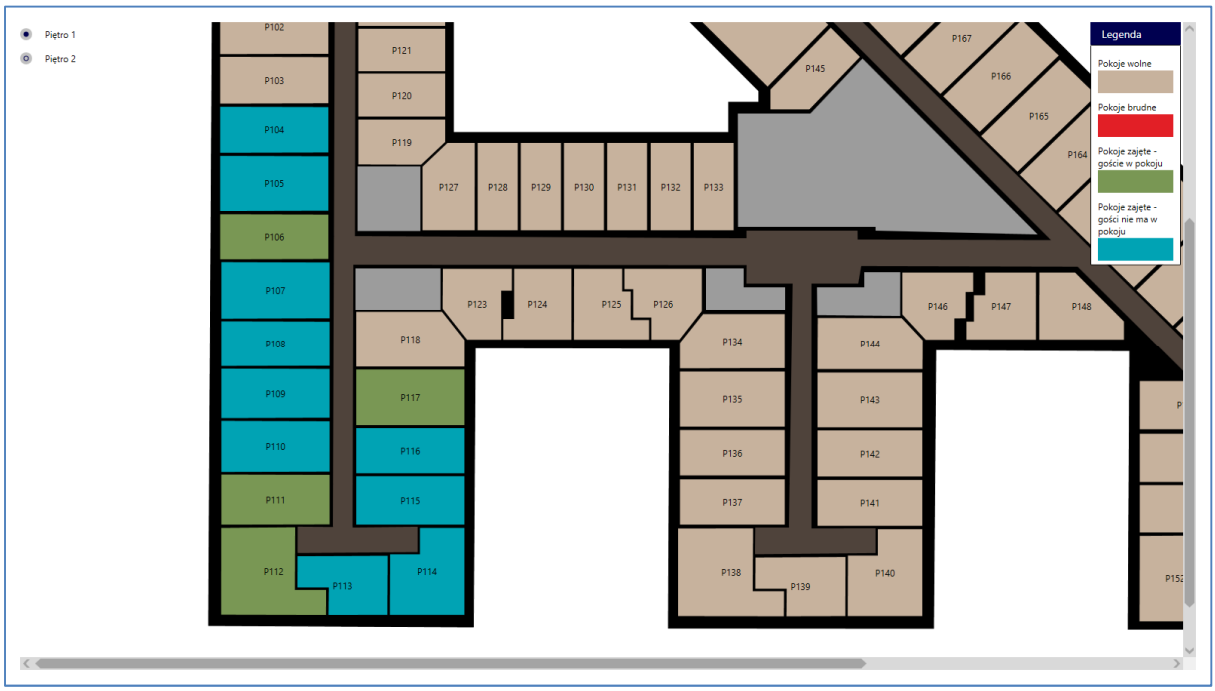
```
<UserControl x:Class="ZarzadzanieHotelem.View.PlanHotelView.HotelPlanView"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:ZarzadzanieHotelem.View.PlanHotelView"
             DataContext="{Binding HotelPlan1, Source={StaticResource PHLocator}}">

    <Canvas Height="1300" Width="2100">
        <Path x:Name="T_PATH_KONTUR" Data="M255,265 L604.5,269.5 604.5,349.5 574.5,349.5 574.5,529.5 939.5,529.5 939.5,489.5 979.5
        <Canvas x:Name="T_CANVAS_POKOJE" Height="1051" Canvas.Left="264.5" Canvas.Top="129.5" Width="1647.333">
            <local:PokojPlanView DataContext="{Binding [P01_1]}" Points="M-5,0 L139.5,0.5 139.5,69.5 -5,70 z" Height="72.333" Canv
            <local:PokojPlanView DataContext="{Binding [P02_1]}" Points="M-5,0 L139.5,0.5 139.5,69.5 -5,70 z" Height="73.875" Canv
            <local:PokojPlanView DataContext="{Binding [P03_1]}" Points="M-5,0 L139.5,0.5 139.5,69.5 -5,70 z" Height="63.875" Canv
            <local:PokojPlanView DataContext="{Binding [P04_1]}" Points="M-5,0 L139.5,0.5 139.5,69.5 -5,70 z" Height="62.875" Canv
            <local:PokojPlanView DataContext="{Binding [P05_1]}" Points="M-5,0 L139.5,0.5 139.5,69.5 -5,70 z" Height="75" Canvas.T
            <local:PokojPlanView DataContext="{Binding [P06_1]}" Points="M-5,0 L139.5,0.5 139.5,69.5 -5,70 z" Height="61.625" Canv
            <local:PokojPlanView DataContext="{Binding [P07_1]}" Points="M-5,0 L139.5,0.5 139.5,69.5 -5,70 z" Height="76.958" Canv
            <local:PokojPlanView DataContext="{Binding [P08_1]}" Points="M-5,0 L139.5,0.5 139.5,69.5 -5,70 z" Height="62.187" Canv
            <local:PokojPlanView DataContext="{Binding [P09_1]}" Points="M-5,0 L139.5,0.5 139.5,69.5 -5,70 z" Height="67.812" Canv
            <local:PokojPlanView DataContext="{Binding [P10_1]}" Points="M-5,0 L139.5,0.5 139.5,69.5 -5,70 z" Height="69.895" Canv
            <local:PokojPlanView DataContext="{Binding [P11_1]}" Points="M-5,0 L139.5,0.5 139.5,69.5 -5,70 z" Height="68.645" Canv
            <local:PokojPlanView DataContext="{Binding [P12_1]}" Points="M364.49974,1136.75 L405.99963,1136.75 405.83297,1171 265.
            <local:PokojPlanView DataContext="{Binding [P13_1]}" Points="M408.1875,1171 L487.25,1170.75 487.625,1091.375 367.125,1
            <local:PokojPlanView DataContext="{Binding [P14_1]}" Points="M489.5,1170.7502 L589.5,1170.8124 589.5,1054.5 529.5,1054
            <local:PokojPlanView DataContext="{Binding [P15_1]}" Points="M0.5,0.5 L145,0.5 L145,69.458 L0.5,69.458 z"  Height="66.
            <local:PokojPlanView DataContext="{Binding [P16_1]}" Points="M0.5,0.5 L145,0.5 L145,69.458 L0.5,69.458 z"  Height="61.
            <local:PokojPlanView DataContext="{Binding [P17_1]}" Points="M0.5,0.5 L145,0.5 L145,69.458 L0.5,69.458 z"  Height="75.
            <local:PokojPlanView DataContext="{Binding [P18_1]}" Points="M0.5,0.5 L115,0.5 145,38.839552 145,75.812 0.5,75.812 z"
            <local:PokojPlanView DataContext="{Binding [P19_1]}" Points="M0.5,0.5 L116.75,0.5 116.75,28.953541 84.5,57.524659 0.5,
            <local:PokojPlanView DataContext="{Binding [P20_1]}" Points="M0.5,0.5 L116.75,0.5 L116.75,58.375 L0.5,58.375 z"  Heigh
```

Rysunek 44 Cześć kodu widoku architektonicznego

W elemencie „UserControl” właściwość „DataContext” przyjmuje obiekt typu „HotelPlanViewModel”. Do obiektów typu „PokojPlanView” do właściwości „DataContext” zostaje podbinowany odpowiedni element słownika. Klucze zostały tak skonstruowane, aby proces podłączania ViewModelu trwał możliwie krótko.

9.4.4. Prezentacja modułu widoku architektonicznego

Z tak przygotowanym mechanizmem dostarczania odpowiednich danych, kontrolka wyświetlająca widok architektoniczny działa szybko i ma estetyczny wygląd. Kontrolka posiada jeszcze dodatkową legendę opisującą znaczenie kolorów , które przyjmują pokoje. Hotel, który używa aplikacji, posiada dwa piętra. Przełączanie pomiędzy nimi odbywa się za pomocą przycisków typu „CheckBox”. Dla każdego piętra zostaje pobrany inny zestaw pokoi. Właściwość ta jest możliwa dzięki sparametryzowanemu konstruktorowi klasy „HotelPlanViewModel”. Parametr konstruktora przyjmuje wartość numeru piętra dla, którego należy przygotować zestaw pokoi, na bazie których zostanie stworzony słownik.



Rysunek 45 Widok architektoniczny piętra pierwszego

Rysunek 46 Wynik operacji kliknięcia na dany pokój

9.5. Panel zarządzający okienkami

Moduł, którego zadaniem jest realizowanie podstawowych operacji na bazie danych hotelu został zaprojektowany jako specjalny panel. Informacje w tym panelu są prezentowane za pomocą kontrolek przypominających i zachowujących się podobnie do okienek systemu „Windows". Zaleta takiej implementacji polega na dowolnej manipulacji zachowania się okienek w zależności od sytuacji na panelu. Należy pamiętać, że okienko jest w programie obiektem typu „UserControl". W związku z tym, programista może dokonać z nim wiele bardzo atrakcyjnych wizualnie operacji w stosunkowo łatwy sposób.

9.5.1. Konstrukcja okienka

Podstawowa konstrukcja okienka opiera się na specjalnie przygotowanej do tego zadania kontrolce o nazwie „ElastycznyRozciaganyVoewBase". Jak sama nazwa wskazuje, kontrolka implementuje takie funkcje jak przeciąganie po panelu oraz powiększanie/zmniejszanie swojego rozmiaru. Jest ona bazowym komponentem w każdej kontrolce, która ma pełnić funkcję okienka.

Rysunek 47 Koncepcyjny schemat okienka

Pasek umożliwiający przeciąganie

W środowisku WPF istnieje specjalny element reagujący na wszelkie operacje typu „drag and drop" wykonywane przez użytkownika. Kontrolka ta nazywa się „Thumb". W celu implementacji specjalnego paska umożliwiającego przeciąganie okienka po panelu została stworzona klasa dziedzicząca po klasie „Thumb" i dodająca metodę do zdarzenia

wywoływanego w momencie przesunięcia elementu. Wszystkie dane dotyczące zmiany położenia elementu programista znajdzie w argumencie typu „DragDeltaEventArgs" dołączanej metody. Wyprowadzona klasa nazywa się „DragThumb" i jej typu jest pasek odpowiedzialny za prawidłowe przeciąganie okienka w przestrzeni roboczej.

```
private void DragThumb_DragDelta(object sender, DragDeltaEventArgs e)
{
    Control designerItem = new ControlHelper().FindParent<IOkno>(this) as Control;
    Panel workspaceCanvas = designerItem.Parent as Panel;
    if (designerItem != null && workspaceCanvas != null)
    {
        //zastosowanie zmiennych int w celu poprawienia dokładności obliczeń
        //typ double powodował rożnice w porównywaniu zmiennych na poziomie tysiecznych części
        //poprzez co warunki kolizji się nie spełniały poprawnie
        int xWsp = (int)Canvas.GetLeft(designerItem);
        int yWsp = (int)Canvas.GetTop(designerItem);
        int controlWidth = (int)designerItem.ActualWidth;
        int controlHeight = (int)designerItem.ActualHeight;
        int xWspL = 0;
        int yWspL = ApplicationSettings.HeightOfAlarmArea;
        int layoutWidth = (int)workspaceCanvas.ActualWidth - ApplicationSettings.WidthOfPassiveArea;
        int layoutHeight = (int)workspaceCanvas.ActualHeight;
        Rect rectControl = new Rect(xWsp+e.HorizontalChange, yWsp+e.VerticalChange, controlWidth, controlHeight);
        Rect layoutControl = new Rect(xWspL, yWspL, layoutWidth, layoutHeight);
        layoutControl.Intersect(rectControl);
        if (layoutControl.Width == rectControl.Width && layoutControl.Height == rectControl.Height)
        {
            Canvas.SetTop(designerItem, Canvas.GetTop(designerItem) + e.VerticalChange);
            Canvas.SetLeft(designerItem, Canvas.GetLeft(designerItem) + e.HorizontalChange);
        }
        else
        {
            //okienko przechodzi lewą krawędź pola
            if (xWsp < xWspL && yWsp >= yWspL && yWsp <= yWspL + layoutHeight && yWsp + controlHeight >= yWspL &&
            {
                Canvas.SetTop(designerItem, yWsp);
                Canvas.SetLeft(designerItem, xWspL);
            }else
            //okienko przechodzi prawą krawędź pola
            if (xWsp + controlWidth > xWspL + layoutWidth && yWsp >= yWspL && yWsp <= yWspL + layoutHeight && yWsp
            {
                Canvas.SetTop(designerItem, yWsp);
```

Rysunek 48 Część metody odpowiedzialnej za walidacę położenia okienka podczas przeciągania

Bardzo ważnym elementem powyższej metody jest określenie pojemnika, w jakim znajduje się pasek. Należy posiadać referencję do obiektu okna, aby można było określić jego parametry położenia i dokonać odpowiedniej decyzji, czy element można przesunąć o daną wartość w pionie lub poziomie i czy należy go zatrzymać. Aby wyszukać kontrolkę odpowiedzialną za okienko, napisano specjalną sparametryzowaną metodę „FindParent" w klasie „ControlHelper". Metoda ta ma za zadanie wyszukać element w hierarchii wizualnego drzewa WPF, który implementuje specjalny interfejs „IOkno". Każde okienko w programie musi implementować ten interfejs w celu poprawnego działania.

```
public class ControlHelper
{
    public T FindParent<T>(DependencyObject child)
        where T : class
    {
        DependencyObject parentObject = VisualTreeHelper.GetParent(child);
        if (parentObject == null) return null;
        T parent = parentObject as T;
        if (parent != null)
            return parent;
        else
            return FindParent<T>(parentObject);
    }
```

Rysunek 49 Kod metody FindParent

Kiedy kontrolka implementująca funkcje okienka zostanie wyszukana, zostają pobrane jej dane odnośnie położenia w panelu, w którym się znajduje i sprawdzone są warunki mówiące o kolizji z obszarem dozwolonym. Obszar dozwolony to płaszczyzna, po której element może być przeciągany. W przypadku np. przeciągnięcia okienka za dowolną krawędź obszaru dozwolonego, następuje automatyczna korekta położenia i kontrolka zostaje wyrównana do zagrożonej krawędzi.

Element umożliwiający zmianę rozmiaru

Kod elementu służącego do zmniejszania/zwiększania okna jest bardzo podobny do kodu elementu przeciągającego. Również tworzy się nowy typ kontrolki dziedziczący po klasie „Thumb”. Jednak w metodzie wywoływanej podczas interakcji z użytkownikiem zmienia się nie położenie okna, ale jego rozmiary. Jednocześnie następuje walidacja uniemożliwiająca stworzenie okienka większego niż obszar dozwolony.

Zawartość okna

Cała zawartość jaką okienko ma w sobie zawierać, zrealizowana jest jako wyprowadzenie właściwości zależnej. Każda kontrolka realizująca funkcję okna musi umieścić kod odpowiedzialny za jej wygląd w tej właściwości.

```
public partial class ElastycznyRozciaganyViewBase : UserControl
{
    public object MainContent
    {
        get { return (object)GetValue(MainContentProperty); }
        set { SetValue(MainContentProperty, value); }
    }
    public static readonly DependencyProperty MainContentProperty =
        DependencyProperty.Register("MainContent", typeof(object), typeof(ElastycznyRozciaganyViewBase),
        new PropertyMetadata(null));

    public string WindowTitle
    {
        get { return GetValue(WindowTitleProperty).ToString(); }
        set { SetValue(WindowTitleProperty, value); }
    }
    public static readonly DependencyProperty WindowTitleProperty =
        DependencyProperty.Register("WindowTitle", typeof(string), typeof(ElastycznyRozciaganyViewBase),
        new PropertyMetadata(""));

    public ElastycznyRozciaganyViewBase()
    {
        InitializeComponent();
    }
}
```

Rysunek 50 Kod klasy ElastycznyRozciaganyViewBase

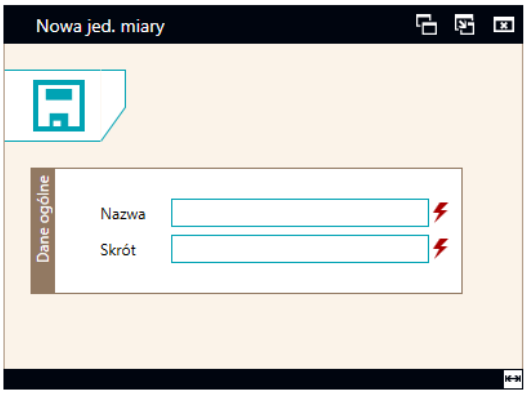
Powyższy kod implementuje dwie właściwości zależne. Jedna z nich odpowiada za wyżej wspomnianą zawartość okna. Druga jest napisem wyświetlającym się na górnym pasku okna, odpowiedzialnym za jego przeciąganie. Jest to jednocześnie tytuł okna.

9.5.2. Przykładowa implementacja okienka

```
<thmb:ElastycznyRozciaganyViewBase WindowTitle="{Binding Title}">
    <thmb:ElastycznyRozciaganyViewBase.MainContent>
        <ScrollViewer>
            <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition Height="auto"/>
                    <RowDefinition Height="auto"/>
                    <RowDefinition Height="auto"/>
                    <RowDefinition />
                </Grid.RowDefinitions>
                <controls:DynamicMenu Grid.Row="0"
                                      SaveCommand="{Binding SaveCommand}"/>
                <GroupBox Grid.Row="1" Header="Dane ogólne">
                    <Grid>
                        <Grid.ColumnDefinitions>
                            <ColumnDefinition Width="auto"/>
                            <ColumnDefinition Width="auto"/>
                            <ColumnDefinition/>
                        </Grid.ColumnDefinitions>
                        <Grid.RowDefinitions>
                            <RowDefinition />
                            <RowDefinition />
                        </Grid.RowDefinitions>
                        <TextBlock Grid.Column="0" Grid.Row="0" Text="Nazwa"/>
                        <TextBox Grid.Column="1" Grid.Row="0" Text="{Binding Nazwa, UpdateSourceTrigger=PropertyChanged,

                        <TextBlock Grid.Column="0" Grid.Row="1" Text="Skrót"/>
                        <TextBox Grid.Column="1" Grid.Row="1" Text="{Binding Skrot, UpdateSourceTrigger=PropertyChanged,
                    </Grid>
                </GroupBox>
            </Grid>
        </ScrollViewer>
    </thmb:ElastycznyRozciaganyViewBase.MainContent>
```

Rysunek 51 Przykład implementacji zachowania okna

Powyższy kod obrazuje przykład zastosowania kontrolki „ElsatycznyRozciaganyViewBase". Jest ona głównym kontenerem dla całej zawartości okna, która znajduje się we właściwości „MainContent". Poniżej zaprezentowano efekt przedstawionego powyżej kodu.



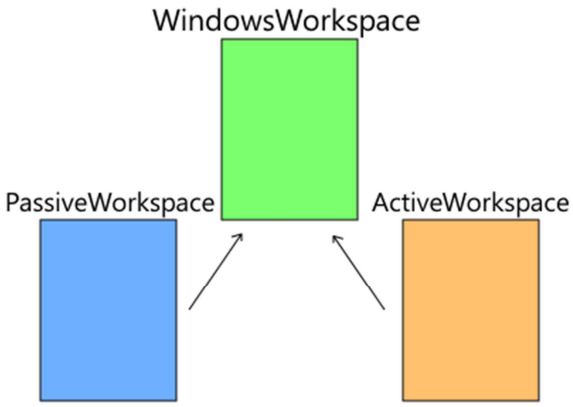
Rysunek 52 Przykład okienka

9.5.3. Kontener zarządzający oknami

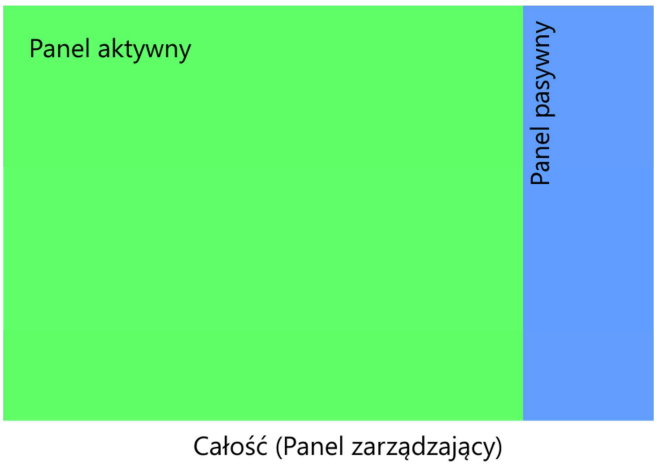
Kontener, którego zadaniem jest odpowiednie umieszczenie okienek i kontrola nad nimi jest zaimplementowany za pomocą dwóch współpracujących ze sobą paneli:

- panel aktywny wyświetlający okna w pełni ich rozmiarów; jest to panel, na którym znajdują się okienka, z których aktualnie korzysta użytkownik;
- panel pasywny ukrywający chwilowo niepotrzebne okienka w postaci eleganckich kafelków.

Komunikacja pomiędzy panelami odbywa się poprzez klasę nadrzędną, która zawiera w sobie zmienne przechowujące po jednym egzemplarzu wyżej wspomnianych paneli. Panel roboczy należy traktować jako całość a nie rozdzielne dwa obszary. Użytkownik klikając ikonę nowego okna danego typu zgłasza zapotrzebowanie na stworzenie kontrolki umożliwiającej pracę w określonym zakresie. To, gdzie i jak zostanie kontrolka reprezentująca okno stworzona, określa panel zarządzający.



Rysunek 53 Zależności pomiędzy klasami paneli



Rysunek 54 Rozmieszczenie paneli

Implementacja polega na stworzeniu klas „PassiveWorkspace” oraz „ActiveWorkspace” jako prywatnych w przestrzeni nazw „WindowsWorkspace”. Klasy te nie powinny być dostępne poza określony pakiet, gdyż samoistnie nie pełnią one żadnej roli. Klasa panelu nadrzędnego zwiera po egzemplarzu obiektu klas panelu aktywnego i pasywnego. Panel nadrzędny pełni kontrolę nad opisanymi panelami za pomocą następujących metod:

- dodanie okna: najpierw zostaje wykonana próba umieszczenia okna w panelu aktywnym; w przypadku kiedy nie ma w nim miejsca, zostaje podjęta próba dodania kafelka reprezentującego okno w panelu pasywnym; kiedy panel pasywny również będzie pełny, użytkownik zobaczy komunikat informujący go o sytuacji i poradę;
- umieszczenie okna na pierwszym planie: w przypadku wielu okien w panelu aktywnym tylko jedno z nich może być otwarte na interakcję z użytkownikiem; pozostałe nieaktywne okna są ukryte za bladą powłoką; funkcja umieszczenia okna na pierwszym planie informuje panel aktywny o przesunięciu na pierwszy plan okna o odpowiednim identyfikatorze i usunięciu z niego bladej powłoki;
- zminimalizowanie okna: użytkownik może wyrazić chęć umieszczenia aktywnego okna na panelu pasywnym poprzez kliknięcie odpowiedniej ikony; panel zarządzający sprawdza czy w panelu pasywnym znajduje się miejsce na dodatkowy kafelek; jeśli operacja wykona się pomyślnie, okno zostanie zminimalizowane, jeśli nie to użytkownik zobaczy odpowiedni komunikat i poradę;
- maksymalizacja okna: jest to proces odwrotny do minimalizacji.

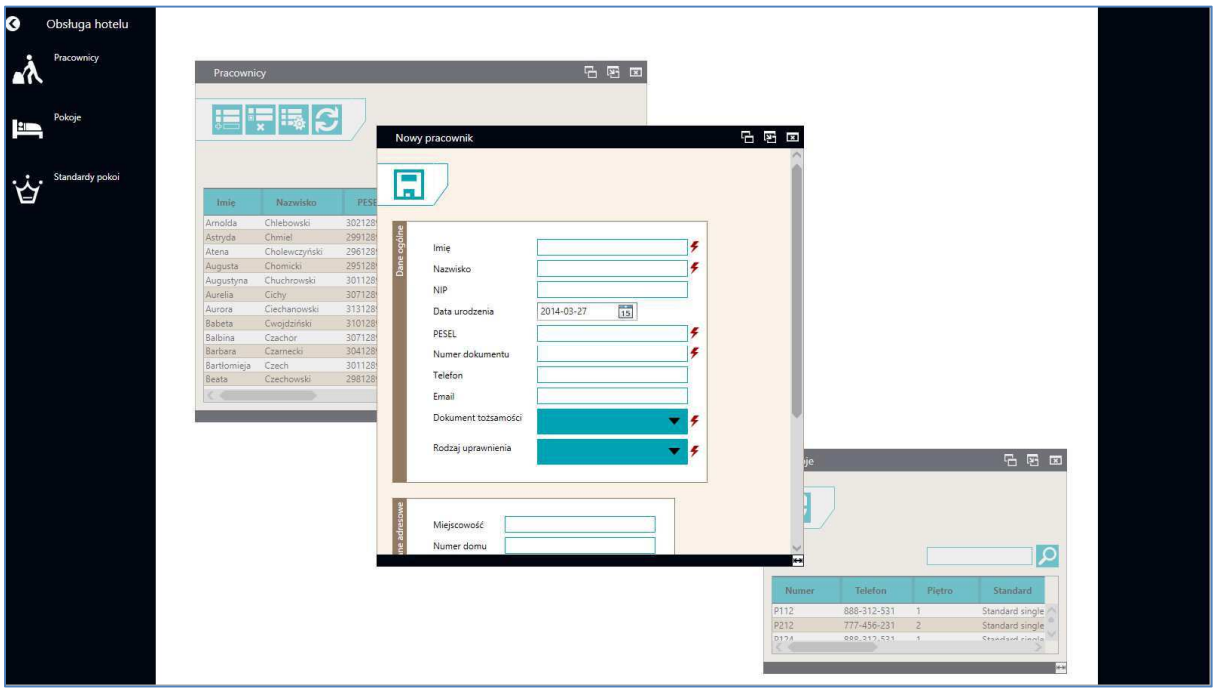
Wszystkie operacje na okienkach w panelu są wyposażone w dodatkowe atrakcyjne efekty wizualne. W tym panelu zaimplementowano szereg skomplikowanych animacji, takich jak

wpływanie pojawiającego się okienka czy minimalizacja okna do kafelka jako płynne przejście pomiędzy dwoma różnymi kształtami znajdującymi się w różnych położeniach. Animacje te musiały zostać wykonane za pomocą kodu C#, gdyż cechuje je pełna dynamika. Nie jest się w stanie przewidzieć np. w jakim położeniu będzie okienko kiedy nastąpi jego minimalizacja i pod który z kolei kafelek ma się ono ukryć.

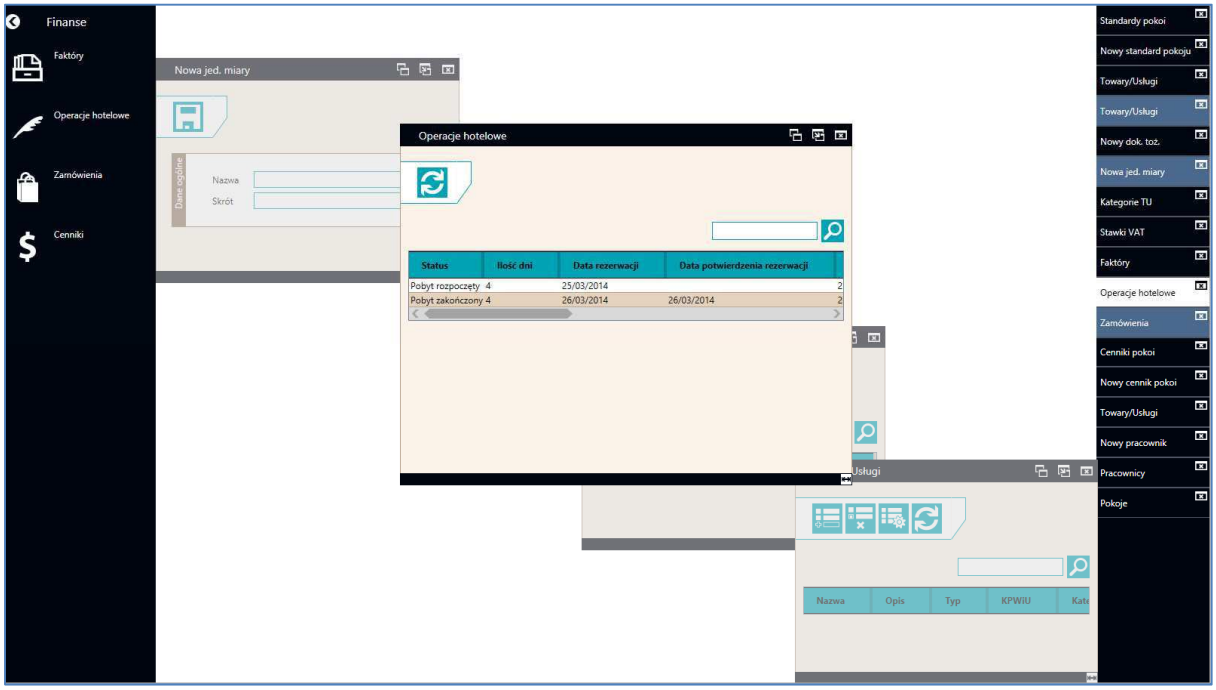
W panelu pasywnym stan kafelków jest ściśle powiązany z okienkami znajdującymi się w panelu aktywnym. Okienko gotowe do interakcji z użytkownikiem posiada białe tło jako kafelek. Pozostałe okna znajdujące się w panelu aktywnym jako kafelki posiadają tło jasnoniebieskie. Natomiast kafelki bez reprezentacji okien posiadają tło granatowe. Kafelek został zaimplementowany jako kontrolka dynamicznie zmieniająca swój wygląd w zależności od pewnych dołączonych do niej danych.

9.5.4. Przykłady użycia

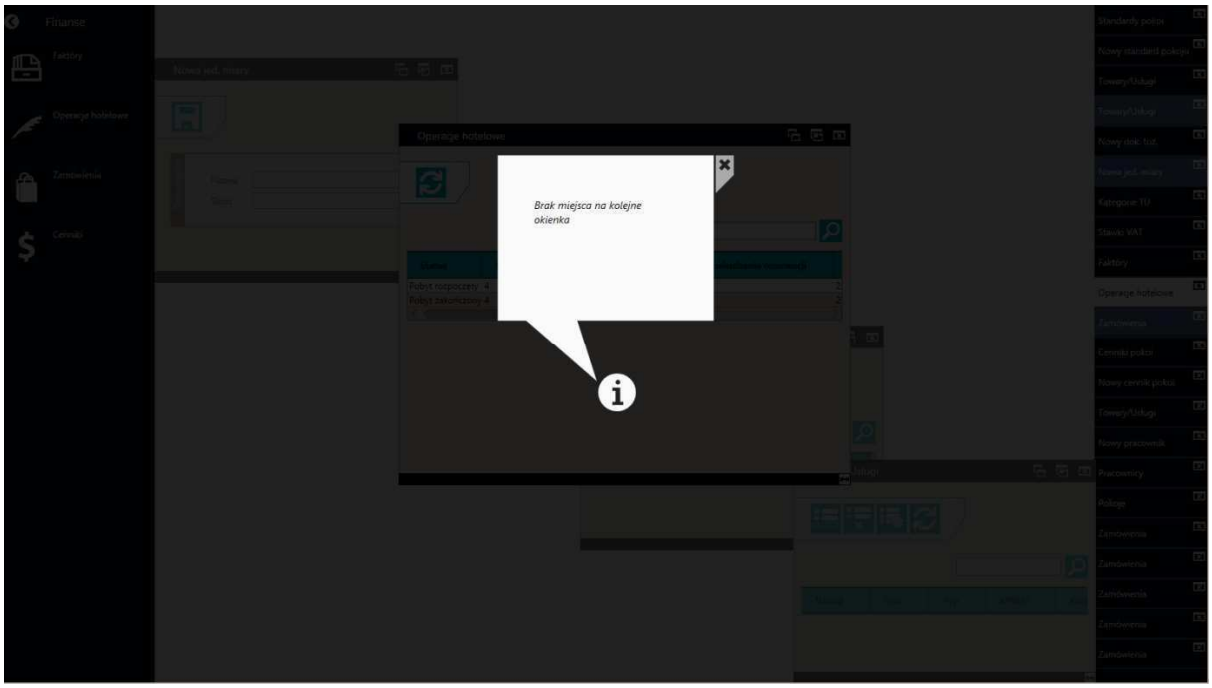
Rysunek 55 Pusty panel roboczy



Rysunek 56 Zapełniony panel aktywny z oknem głównym



Rysunek 57 Zapełniona przestrzeń robocza. Wybór okien aktywnych z panelu pasywnego

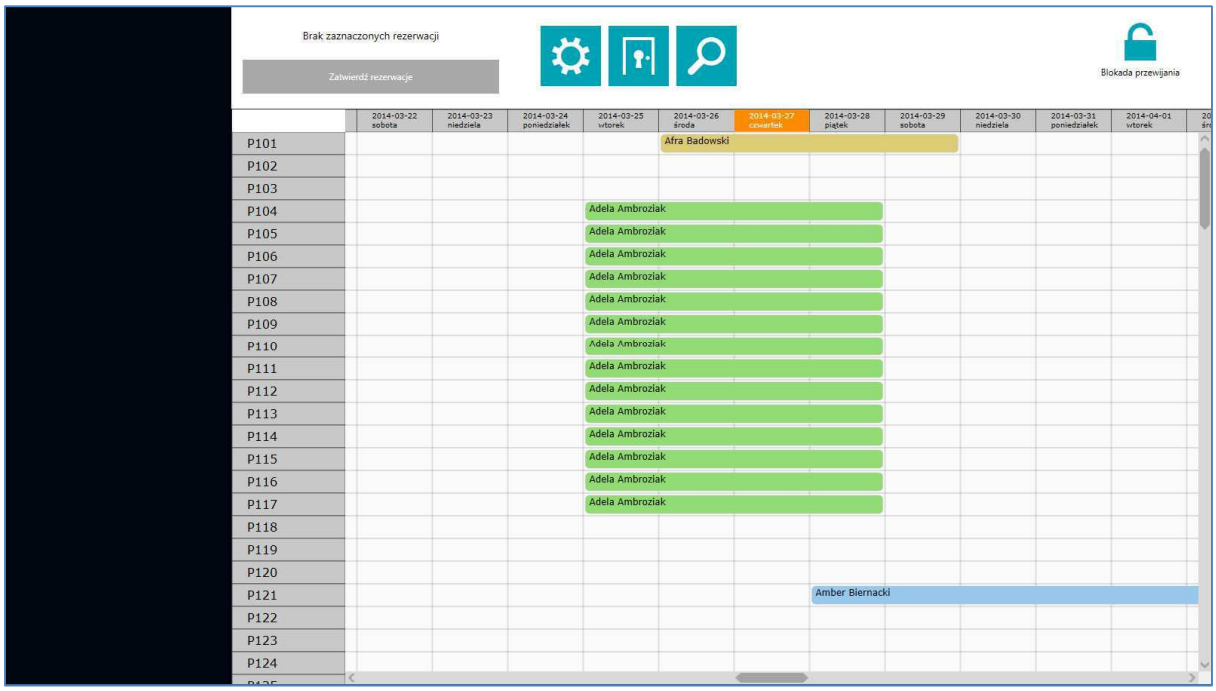


Rysunek 58 Komunikat informujący o braku miejsca w panelu roboczym na kolejne okna

9.6. Moduł rezerwacji

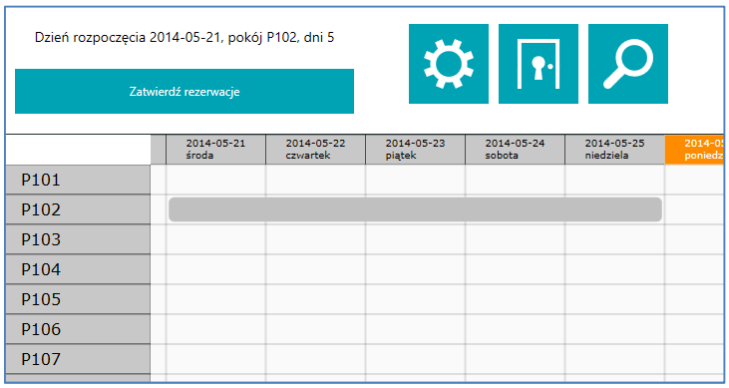
9.6.1. Prezentacja działania kontrolki kalendarzowej

Cały moduł rezerwacji, w skład którego wchodzą dwa główne komponenty, czyli pasek boczny oraz kontrolka kalendarzowa, został zaprojektowany w celu jak najwygodniejszego i najszybszego obsługiwania klienta. Na pasku bocznym wyświetlają się pracownikowi recepcji formularze, które może uzupełnić w związku z dokonaną przez siebie operacją. Główną część obszaru ekranu zajmuje kontrolka kalendarzowa wyświetlająca graficzną reprezentację rezerwacji hotelowych. Kontrolka ta wyświetla dane w zależności zarezerwowanych pokoi od czasu.



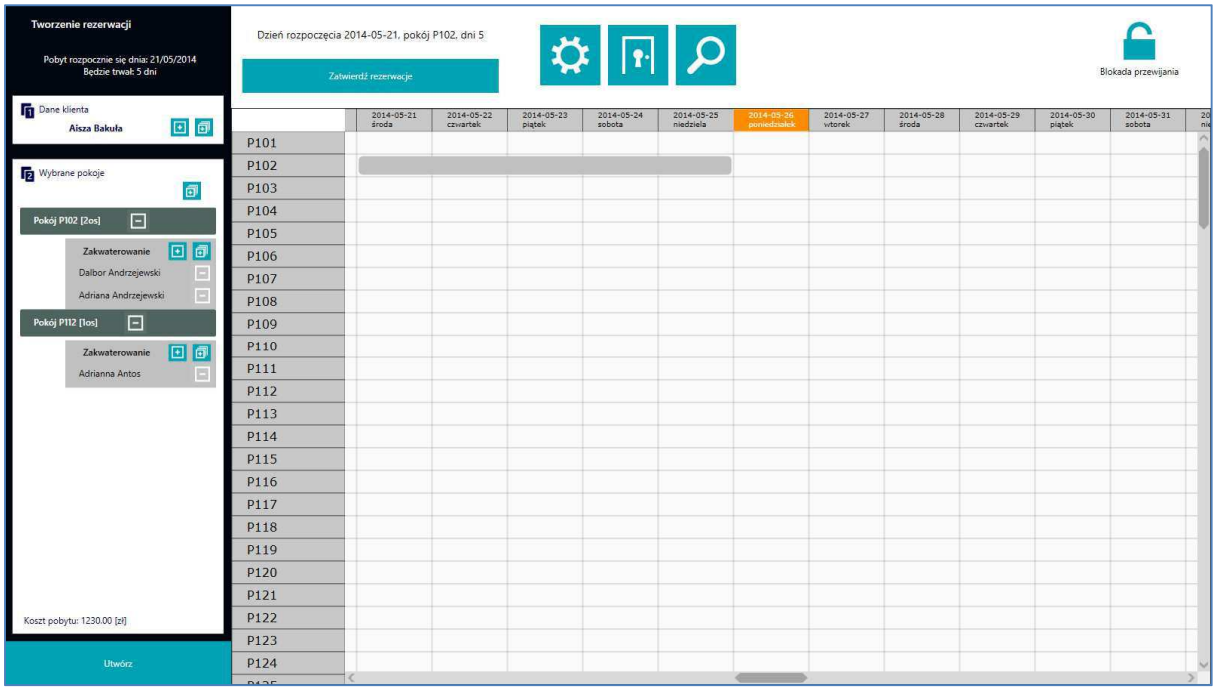
Rysunek 59 Moduł rezerwacji: po lewej miejsce na formularze, główną część ekranu zajmuje kontrolka kalendarzowa z opcjami

Utworzenie rezerwacji dla klienta jest bardzo intuicyjne. Należy zaznaczyć w kalendarzu, trzymając wciśnięty lewy przycisk myszy, przedział czasu w pokoju, którym klient jest zainteresowany. Kolor paska zaznaczonego przedziału powinien być srebrny. Po zaznaczeniu przdziału czasu aktywny staje się przycisk „Zatwierdź rezerwację”.

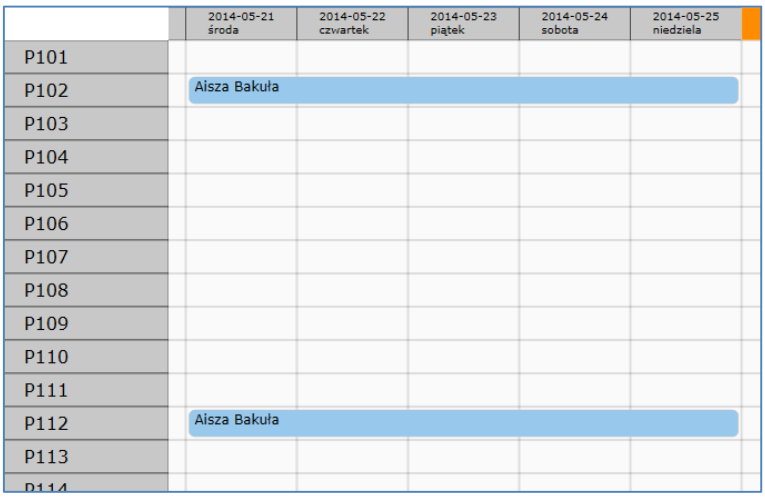


Rysunek 60 Zaznaczenie okresu rezerwacji

Przycisk zatwierdzający rezerwację powoduje pojawienie się formularza tworzenia rezerwacji. Formularz udostępnia opcje wyboru osoby, na którą rezerwacja będzie zatwierdzona. Można także zarezerwować więcej pokoi i zameldować do nich osoby. W oknie wyboru wielu pokoi pojawiają się tylko te, które są wolne w zaznaczonym przedziale czasu. Po wprowadzeni danych należy wybrać opcję „Utwórz rezerwację". Wówczas dane zostaną wprowadzone do bazy a w kalendarzu pojawi się wpis dotyczący rezerwacji niepotwierdzonej. Kolor paska rezerwacji będzie jasno-niebieski.

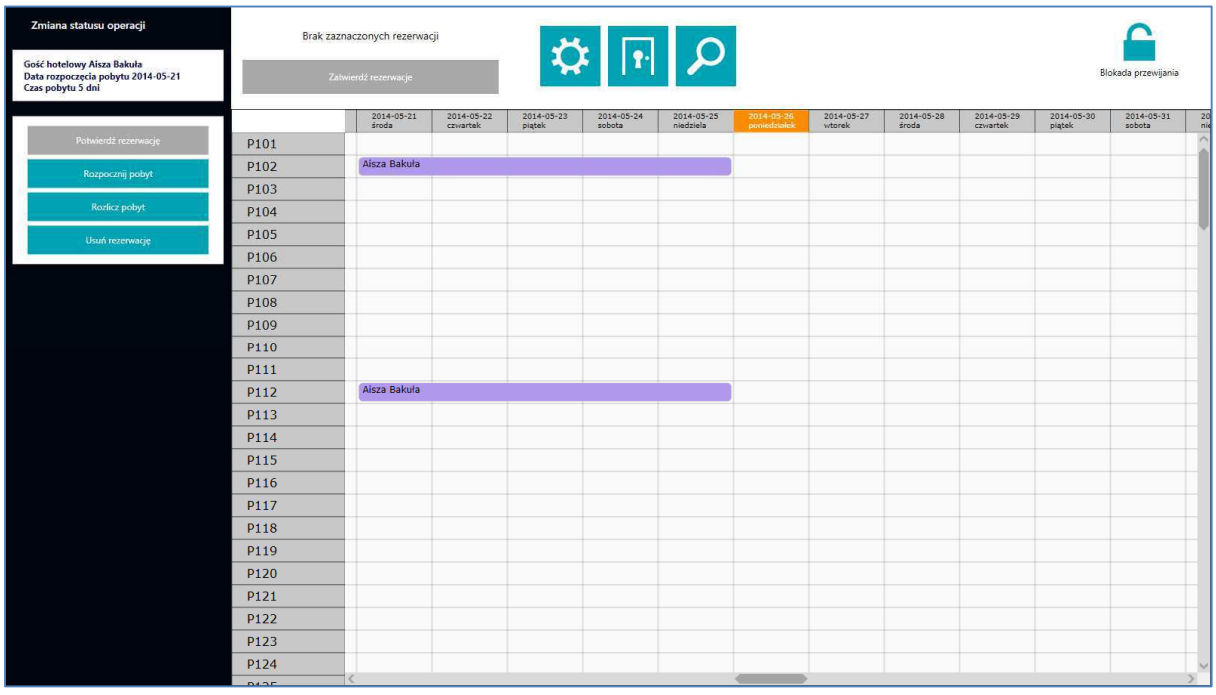


Rysunek 61 Formularz pobierający dane o rezerwacji



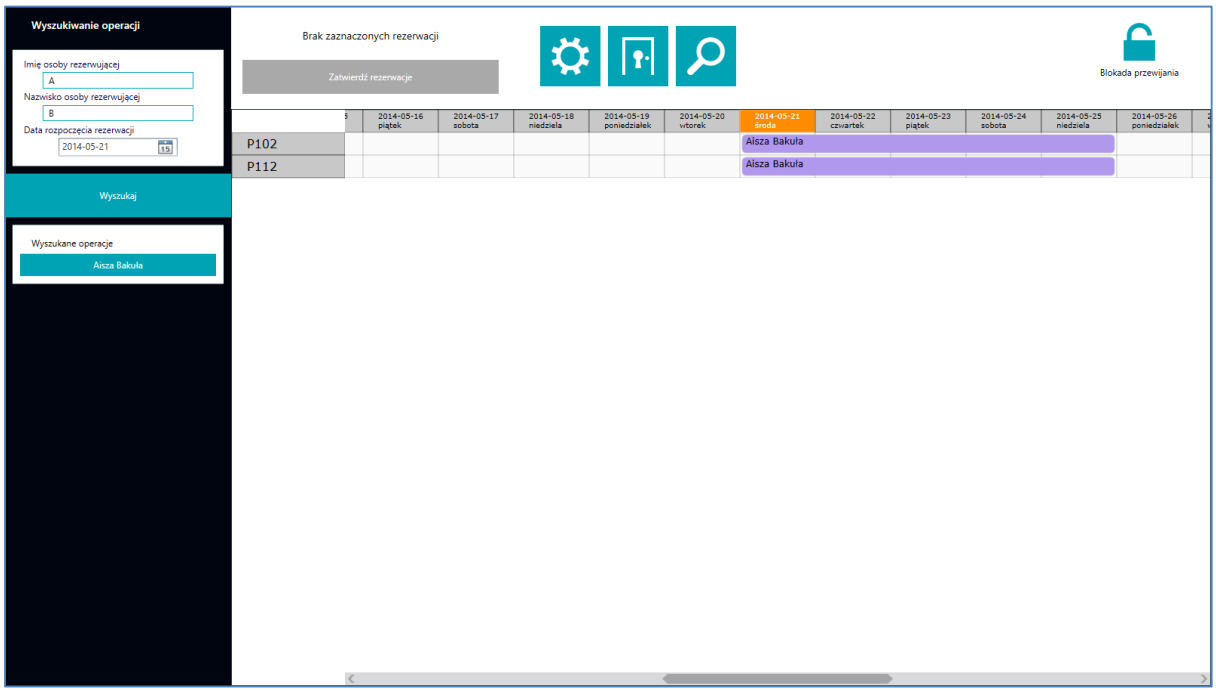
Rysunek 62 Zmiana wyglądu paska rezerwacji po jej zatwierdzeniu. Jak widać pojawiają sie dane osoby, która dokonała rezerwacji

Mając utworzona rezerwację można wchodzić z nią w szereg interakcji. Klikając na pasek rezerwacji lewym klawiszem myszy udostępnia się formularz mogący zmieniać stan rezerwacji. Stan rezerwacji odpowiada kolorowi jej paska w kontrolce kalendarzowej.



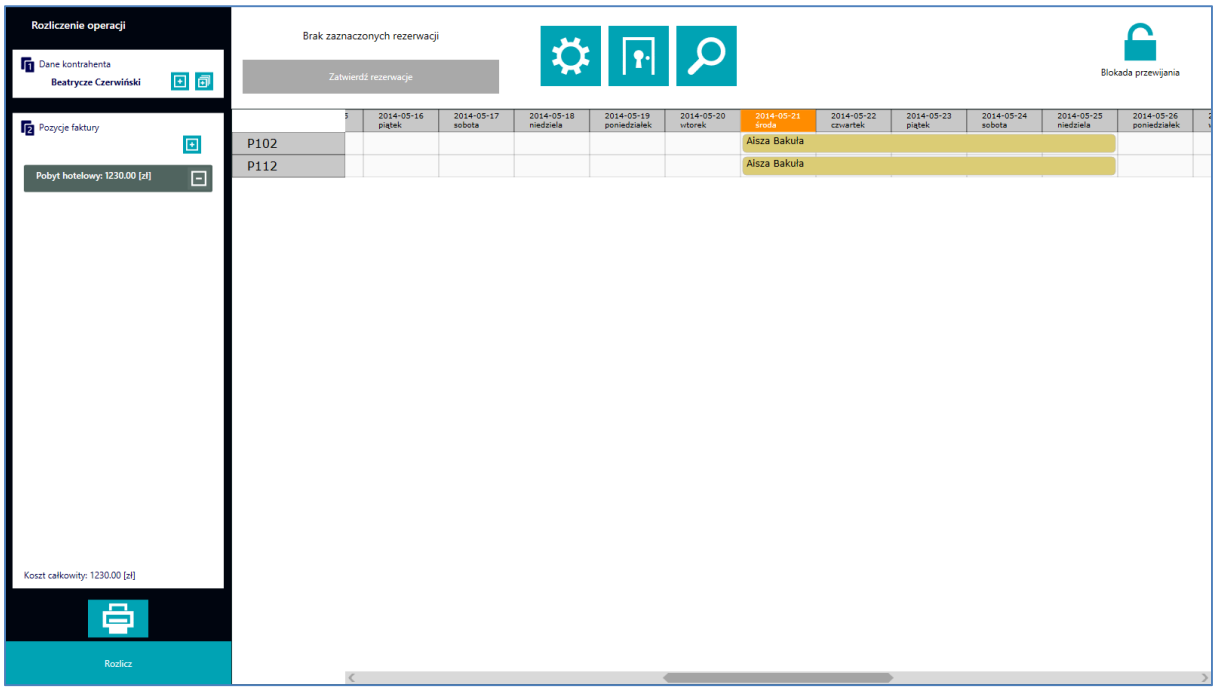
Rysunek 63 Zmiana stanu rezerwacji na potwierdzoną

Klient kończąc swój pobyt chce dokonać rozliczenia. Recepcjonista otwiera wyszukiwarkę operacji hotelowych i wpisuje dane osoby zakładającej rezerwację w specjalnie przygotowane do tego pola. Po kliknięciu „Wyszukaj” pokoje oraz przedział czasu w kalendarzu dostosują się do wyszukiwanego pobytu.



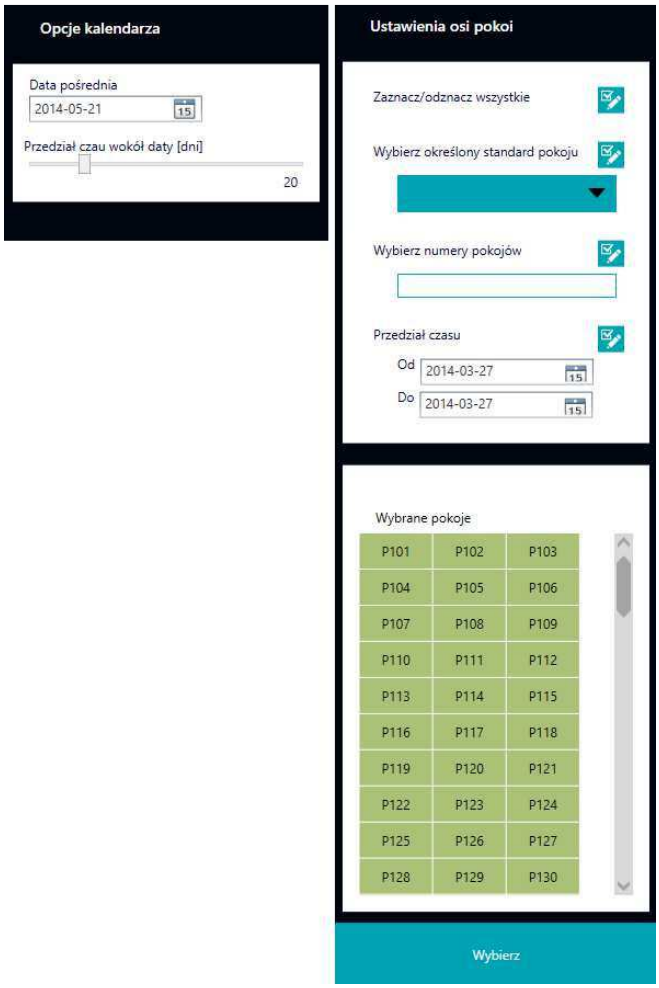
Rysunek 64 Wyszukiwanie rezerwacji

Kiedy rezerwacja zostanie znaleziona, użytkownik klika na nią lewym klawiszem myszy i z opcji zmiany stanu wybiera „Rozlicz pobyt". Pojawia się formularz, który wymaga podania danych kontrahenta do faktury. Umożliwia on też dodanie nowych pozycji faktury. Po rozliczeniu pobytu dokument można wydrukować do formatu PDF.



Rysunek 65 Rozliczanie pobytu hotelowego

W celu wygody korzystania z kontrolki kalendarzowej można dowolnie dostosowywać jej część służącą do wyświetlania danych. Udostępniają to panele opcji wywoływanie po kliknięciu jednego z kilku większych przycisków w górnym menu kontrolki kalendarzowej. Można dostosować pokoje, jakie będą wyświetlanie na osi pionowej. Można też wybrać datę, wokół której będą wyświetlane dane rezerwacji w określonym przedziale dni, który również jest modyfikowany.



Rysunek 66 Panele umożliwiające sterowanie kontrolką kalendarzową

10. Zakończenie

Niniejsza praca służy przybliżeniu zaawansowanych możliwości technologii „Windows Presentation Fundation" wraz z połączeniem wzorca projektowego „Model View View-Model" w procesie tworzenia aplikacji biznesowej. Projekt zrealizowany na potrzeby pracy jest aplikacją hotelową wyposażoną w podstawową funkcjonalność, umożliwiającą przeprowadzenie procesu rezerwacji oraz zarządzanie obiektem hotelowym. Projekt jest podzielony na moduły, które zostały zaprojektowane tak, aby można było zademonstrować potęgę platformy WPF w połączenie z wzorcem MVVM. Zaawansowane zagadnienia wykorzystane w projekcie to m. in.

- budowa niestandardowych styli oraz szablonów dla kontrolek systemowych;
- zastosowanie zaawansowanych animacji;
- wykorzystanie narzędzia „ExpressionBlend" w celu zaprojektowania złożonego widoku architektonicznego;
- projektowanie własnych kontrolek użytkownika;
- projektowanie kontrolek bazujących na renderowaniu elementów niskiego poziomu;
- wykorzystanie interfejsów 3D;
- wykorzystanie biblioteki „MVVM Light" w celu implementacji wzorca MVVM.

Spis rysunków