Złożenie pracy online:
2021-03-12 15:55:09
Kod pracy:
8349/38899/CloudA

Jarosław Bochenek
(nr albumu: 23576)

Praca inżynierska

Intertegra - kompleksowy system zarządzania działalnością firmy stolarskiej

Intertegra

Wydział: Wydział Nauk Społecznych i Informatyki

Kierunek: Informatyka

Specjalność: programowanie aplikacji biznesowych

Promotor: dr Krzysztof Przybycień

Serdeczne podziękowania mojemu promotorowi dr. Krzysztofowi Przybycieniowi za konsultacje i wszelką pomoc okazaną w procesie tworzenia aplikacji i pisania pracy.

Streszczenie

Tematem ninejszej pracy było stworzenie systemu do kompeksowego zarządzania firmą stolarską. System składa sie z dwóch aplikacji. Pierwszą jest aplikacja internetowa, na której prezentowane są produkty firmy. Szczególny nacisk został położony na prostotę i przejrzystość strony, która ułatwi potencjalnym klientom zdecydowanie się na skorzystanie z usług oferowanych przez firmę. Drugą aplikacją jest program komputerowy przeznaczony do instalacji na komputerach pracowniczych. Z jego pomocą można zarządzać zamówieniami w firmie, włącznie z tworzeniem kosztorysów. Program ten został też wyposażony w system adnotacji - specjalnych przypomnień oraz podstawową obsługę magazynu. Do stworzenia projektu użyto technologii .NET Core oraz języka C#, HTML, JavaScript i XAML. Działanie aplikacji opiera się na realacyjnej bazie danych mssql.

Słowa kluczowe

programowanie aplikacji biznesowych, strona internetowa, zarządzanie zamówieniami, kosztorysy

Abstract

The subject of this thesis is to create the system to compresenhive managment carpentry company. System containg two applications. First is Web App, where companys products are presented. Particular emphasis has been placed on simplicity and clarity of the site. This helps customer to choose companys services. Second application is computer program designed to be installed on company computers. It allows to manage orders im company, including creating costs estimates. This program has been equipped with annotations system - special reminders and basic warehouse managment. To create this projekt the .Net Core technology has been uses and the following programming languages: C#, HTML, JavaScripts and XAML. Applications where created with realtive mssql database.

Keywords

business programming, website, orders managment, estimates

Spis treści

Rozdział 1. Wstęp

Celem mojej pracy było zaprojektowanie i stworzenie systemu wspomagającego działalność firmy stolarskiej zajmującej się produkcją tarasów i pergoli na zamówienie. System ten składa się z aplikacji internetowej oraz okienkowej przeznaczonej do użytku na komputerach firmowych. Strona internetowa została zaprojektowana z myślą o przyciągnięciu potencjalnych klientów. Głównym założeniem było zachowanie prostoty i estetyki, tak aby odwiedzająca ją osobą szybko się w niej odnalazła i nie została przytłoczona nadmiarem treści i elementów. Znaleźć tutaj można przykładowe realizacje produktów oferowanych przez firmę oraz dane kontaktowe. Przez specyfikę działalności nie było możliwe zaimplementowanie funkcjonalności sklepu internetowego – każde zamówienie wycenianie i realizowane jest indywidualnie. Podczas tworzenia tej aplikacje zostały zaimplementowane mechanizmy SEO. Aplikacja dzieli się na dwie strefy: publiczną dla klienta oraz kokpit dla pracownika, gdzie możliwa jest edycja treści wyświetlanych na stronie. Aplikacja okienkowa ma za zadanie wspomagać proces realizacji zamówienia. Umożliwia zarządzanie klientami, materiałami i ich dostawami oraz samymi zamówieniami. Na zarządzanie zamówieniami składa się także tworzenie kosztorysów, dzienników pracy i adnotacji. Na kosztorysy składają się ich dwa rodzaje: wycena oraz roboty dodatkowe. Ten drugi nie jest automatycznie tworzony i przeznaczony jest do wypełnienia po rozpoczęciu prac. Dodatkową pomocą jest tutaj system adnotacji, specjalnych przypomnień, które pojawią się po minięciu ustawionej daty. Pozwalają nie zapomnieć o ważnych wydarzeniach, jak na przykład zaplanowana wizyta u klienta. Obie aplikacje połączone są relacyjną bazą danych.

W pierwszym rozdziale przedstawione zostały najważniejsze i podstawowe informacje o technologii .Net Core, w której obie aplikacje zostały stworzone. Znajdują się tutaj też informacje na temat wzorców MVVM i MVC, według których zostały stworzone kolejno aplikacja internetowa oraz okienkowa. Znajdziemy tutaj także informacje na temat bazy danych użytej w projekcie oraz podejściu Database First. Rozdział kończą informacje na temat języków użytych przy programowaniu aplikacji. W następnym rozdziale opisane zostało środowisko programistyczne użyte przy tworzeniu aplikacji oraz jak prezentuje się struktura solucji (wraz z wyjaśnieniem pojęcia „solucja") tego projektu. Rozdział trzeci prezentuje strukturę bazy danych oraz wyjaśnienie wybranych tabel oraz ich relacji. Następne rozdziały szczegółowo opisują poszczególne projekty solucji oraz ich funkcjonalności.

Filmik z prezentacją działania systemu jest dostępny pod linkiem: https://youtu.be/C0sRIpcOC7E natomiast aplikacja internetowa dostępna jest pod adresem: https://intertegra.bochenekgroup.pl/ z danymi logowania:

login: kontakt@intertegra.com

hasło: Haslo123!

Filmik i strona internetowa zostały umieszczona pod tym adresem na potrzeby oceny tej pracy.

Rozdział 2. Technologie zastosowane w projekcie

1.1. Technologia .NET

.NET jest stworzoną przez Microsoft w całości darmową multiplatformową platformą programistyczną. Języki obsługiwane przez tą technologię to C#, F# ora Visual Basic. W tym projekcie został użyty język C#. Umożliwia ona pisanie aplikacji zarówno desktopowych jak i mobilnych i internetowych. Technologia jest multiplatformowa, a więc napisane w niej programy można uruchomić na systemach operacyjnych Windows, Linux i MacOS.[1]

Aplikacja internetowa została stworzona za pomocą ASP.NET, rozszerza ona technologię .NET o narzędzia i biblioteki do budowania aplikacji webowych.[2]

Aplikacja okienkowa została stworzona za pomocą WPF (Windows Presentation Fundation). Jest to graficzny system zaprojektowany przez Microsoft do tworzenia interfejsów użytkownika w aplikacjach okienkowych. Wspiera on projektowanie oparte na Modelu, bindowaniu danych, plikach resources i kontrolkach. Używa się tutaj języka XAML.[3]

2.1. Wzorzec projektowy MVVM

2.1.1. Ogólne informacje dotyczące wzorca MVVM

Tworzenie nowoczesnych i profesjonalnych aplikacji wymaga użycia wzorców projektowych. Umożliwiają one podział aplikacji na poszczególne moduły. Podział ten bazuje na funkcjach za jakie dany moduł odpowiada. Taki zabieg ułatwia zarządzanie całym projektem, systematyzuje pracę, porządkuje kod i znacząco usprawnia proces dodawania nowych funkcjonalności. W przypadku wystąpienia błędu nie ma potrzeby analizy całego projektu, a jedynie konkretnego modułu. MVVM jest powszechnie stosowanym i wykorzystanym w tym projekcie, wzorcem projektowym do budowania aplikacji okienkowych (desktopowych). Jego pełna nazwa brzmi Model-View-ViewModel. Dzieli on aplikację na trzy warstwy: model, view (widok) oraz viewmodel (widok-model).

2.1.2. Warstwa Model

Ta warstwa odpowiada za zarządzanie danymi. Zawiera klasy dostępowe do bazy danych jak i realizuje logikę biznesową aplikacji. Klasy tej warstwy odzwierciedlają strukturę bazy danych wraz z relacjami pomiędzy poszczególnymi danymi. Więcej o relacjach bazy

[1] https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet
[2] https://dotnet.microsoft.com/apps/aspnet
[3] https://docs.microsoft.com/en-us/visualstudio/designers/getting-started-with-wpf?view=vs-2019

danych znajduje się w rozdziale 1.4. Kilka słów o bazie danych Microsoft SQL i podejściu Database First. Realizuje się tutaj również walidacja danych. Każde pole klasy odpowiadające kolumnie tabeli z bazy danych posiada tzw. DataAnnotation. Są to specjalne informacje o warunkach, które dane pole musi spełniać. Są to:

- StringLength(x) – maksymalna długość pola tekstowego,
- Columne(TypeName = ”x”) – typ pola np. DateTime dla pola daty i czasu,
- Required – pole wymagane.

Jest to walidacja na poziomie bazy danych. Dodatkowo w tej warstwie umieszcza się klasy logiki biznesowej, które mogą dodatkowo walidować dane np.: czy tutuł jest unikalny w całej tabeli. Klasy te zawierają także funkcje, które przetwarzają dane. Jako przykład możemy podać obliczanie wartości kosztorysu, która obliczana jest na podstawie materiałów. W tym projekcie użyty tutaj został język C#.

2.1.3. Warstwa View

Odpowiada za interfejsy użytkownika, czyli jak sama aplikacja wygląda. Realizuje graficzną reprezentację danych z warstwy Model oraz odbiera interakcje ze strony użytkownika (kliknięcie myszą, wprowadzenie danych na klawiaturze) i przekazuje te interakcje do warstwy ViewModel. Do elementów interaktywnych zaliczamy przyciski, pola tekstowe, pole do wyboru daty, pola wyboru oraz listy rozwijane. Elementy te mogą posiadać atrybuty, które w porozumieniu z warstwą Model waliduje dane, zaliczamy do nich:

- required – wymusza wypełnienie pola danymi,
- maxlength – określa maksymalną długość pola tekstowego,
- max – maksymalna wartość pola liczbowego,
- min – minimalna wartość pola liczbowego.

Został tutaj użyty język XAML.

2.1.4. Warstwa ViewModel

Warstwa ta łączy i komunikuje ze sobą warstwy Model oraz View. Odbywa się tutaj bindowanie danych czy połączenia pól z warstwy Model z konkretnymi kontrolkami na widoku. Po wprowadzeniu danych przez użytkownika dane przekazywane są do Modelu. Decyduje się tutaj również jakie działania ma wykonać aplikacja po wciśnięciu przycisku. Służą do tego komendy (ang. command).Warstwa ta odpowiada również za to aby po wprowadzeniu zmian przez Model, zaktualizować widok. Używa się tutaj języka C#.

2.2. Wzorzec projektowy MVC

2.2.1. Ogólne informacje dotyczące wzorca MVC

MVC jest powszechnie stosowanym i wykorzystanym w tym projekcie, wzorcem projektowym do budowania aplikacji internetowych. W połączeniu z technologią ASP.NET oznacza to, że zapytanie wysłane przez klienta strony internetowej najpierw przetwarzane jest na serwerze, a później kod wynikowy przekazywany jest w odpowiedzi zwrotnej. Dzieli on aplikację na trzy warstwy: model, view (pol. widok) i controller (pol. kontroler).

2.2.2. Warstwa Model

Warstwa ta działa analogicznie do warstwy Model we wzorcu MVVM.

2.2.3. Warstwa View

Podobnie jak we wzorcu MVVM odpowiada za interfejsy użytkownika. W przypadku technologii ASP.NET używany jest tutaj silnik Razor. Bazuje on na języku HTML, do którego za pomocą znacznika @ można dodać elementy kodu C#. Każdy widok ma przydzieloną klasę z warstwy Model i z pomocą elementów języka C# można umieścić na stronie jej elementy. Tak jak we wszystkich stronach internetowych, widoki stalowane są za pomocą CSS oraz możliwe jest użycie języka JavaScript. Analogicznie jak w przypadku wzorca MVVM warstwa ta umożliwia walidację danych. Nie trzeba jednak ręcznie dodawać wszystkich atrybutów (required, maxlength) do konkretnych kontrolek. Przy podaniu jakiemu polu Modelu odpowiada dana kontrolka silnik Razor automatycznie doda te atrybuty na podstawie DataAnnotation w warstwie Model. Pliki zawierające kod Razor mają rozszerzenie .cshtml.

2.2.4. Warstwa Controller

Pośredniczy między dwiema pozostałymi warstwami oraz obsługuje zapytania klientów. Każdy adres internetowy ma odpowiadającą mu metodę w kontrolerze. O tym jaką metodę dany adres wywoła można określić w regułach routingu. Domyślnie adres www.mywebsite.pl/zamowienia/szczegoly wywoła metodę szczegoly w kontrolerze zamówienia. Każda metoda musi zwrócić widok, przekazując również Model, który następnie będzie obsługiwał silnik Razor

2.3. Kilka słów o bazie danych Microsoft SQL i podejściu Database First

Microsoft SQL jest relacyjną bazą danych stworzoną przez Microsoft. Baza podzielona jest na tabele. Każda zawiera pewne dane. Table Klienci może zawierać Imię, Nazwisko, DataUrodzenia, DataDołaczenia itd. Podstawowe typ danych to:

- bigint, int – numer,
- money – używane do pól pieniężnych, walutowych,
- date – data,
- datetime – data i czas,
- nchar(długość) – tekst zawierający znaki UNICODE określonej długości.
- nvarchar(długość) – analogicznie jak pole nchar. Różnica polega na tym, że miejsce zajmowane przez pole nvarchar jest dynamiczne, na podstawie długości już wprowadzonej danej, a pole nchar od razu zajmie maksymalną ilość miejsce, nawet jeśli wprowadzony tekst nie będzie maksymalnej długości.

Każda tabela powinna posiadać klucz główny, czyli kolumnę, która identyfikuje wiersz w całej tabeli. Następnie na jego podstawie tworzy się klucz obcy, którym można powiązać ze sobą dwie tabele. Rozróżniamy dwa główne typy relacji: jeden-do-wielu oraz wiele-do-wielu. W pierwszym przypadku umieszcza się klucz obcy. Na przykład tabela zamówienia może zawierać klucz obcy do tabeli klienci. Oznacza to, że zamówienie może mieć jednego klienta, natomiast klient może mieć wiele zamówień. Relacja wiele-do-wielu realizowana jest za pomocą trzeciej tabeli pośredniczącej. Jeśli mamy tabele klienci oraz wycieczki to możemy stworzyć tabelę klienciWycieczek, która będzie zawierała dwa klucze obce: do tabeli klienci oraz wycieczki. Dzięki temu wycieczka może być powiązana z wieloma klientami, a klient również może być powiązany z wieloma wycieczkami. Zapytania do bazy danych, również te tworzące tabele, pisane są w języku Transact-SQL.

Ten projekt został zrealizowany z podejściem DatabaseFirst. Oznacza ono, że najpierw w języku bazodanowym tworzona jest baza danych, a następnie na jej podstawie w warstwie Model aplikacji generowane są klasy dostępowe do bazy danych. Do wygenerowania tych klas użyto bibliotek EntityFrameworkCore, co pozwoliło automatycznie wygenerować także DataAnnotation określające typ danych, długość tekstów oraz czy pole jest wymagane. Przy takim modelu pracy nie można edytować klas bazodanowych w projekcie, ponieważ w przypadku zmian w bazie danych, zostaną one wygenerowane od nowa.

2.4. Język C#

Język stworzony przez Microsoft i wydany jako część .NET w 2002 r. Do dzisiaj doczekał się wielu poprawek. Jego głównymi założeniami była prostota, nowoczesność i powszechność opierające się na programowaniu obiektowym. Wyposażony jest w automatyczny Garbage Collection, który usuwa naużywane już dane z pamięci programu. Programy w nim napisane mogą być multiplatformowe, a od pewnego czasu mocno wspiera programowanie asynchroniczne.[45]

2.5. Język HTML i JavaScript i CSS

Języki HTML, JavaScript i CSS są powszechnie używane i obsługiwane przez wszystkie przeglądarki internetowe. Kod przetwarzany jest widok strony internetowej. HTML zawiera statyczny kod, który nadaje ram całej stronie, przechowywany jest w plikach .html. CSS (kaskadowe arkusze stylów) używany jest do sterowania wyglądem poszczególnych elementów określonych językiem HTML. Pliki tego języka przechowywane są w klasach .css. Język JavaScript jest wbudowany w język HTML. Pozwala on dynamicznie reagować na działania użytkownika. Po wprowadzeniu danych, lub kliknięciu przycisku może wywołać funkcję, która może nawet wysłać zapytanie do internetu (w tym także do kontrolera).

2.6. Język XAML

Wykorzystywany przez WPF do tworzenia interfejsów użytkownika. Jest elementem technologii .NET. Kod w tym języku może być tworzony ręcznie lub przez edytory wizualne takie jak np. Blend for Visual Studio. Jest wyposażony w system obsługi komend, co w architekturze MVVM i technologii .NET umożliwia obsługę interakcji użytkownika w warstwie ViewModel. Jest językiem multiplatformowym. Prócz kodu w języku XAML widoku składają się z plików .cs, w których umieszczony jest tzw. Code Behind. Można tam umieścić kod wydarzeń obsługujących widok, chociaż architektura tego nie zaleca MVVM, w zamian proponując wykorzystanie ViewModel. Każdy widok posiada DataContext, czyli klasę z warstwy ViewModel, która dany widok obsługuje. Widok ma dostęp do jej właściwości i komend.

2.7. Visual Studio

Visual Studio jest środowiskiem programistycznym stworzonym przez Microsoft. Umożliwia pisanie m.in. aplikacji mobilnych, okienkowych, internetowych i konsolowych.

[4] https://docs.microsoft.com/pl-pl/dotnet/csharp/whats-new/csharp-version-history
[5] https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/

Zawiera edytor kodu wspierający IntelliSense czyli system podpowiedzi wspierający pracę programisty. Jego częścią jest także zintegrowany debugger, która działa na poziomie pisanego kodu jak i podczas działania programu w trybie debugowania. Umożliwia on m.in. podejrzenie i zmianę wartości zmiennych w czasie działania programu.

Visual studio działa w systemie solucji i projektów. Solucja to zbiór projektów, które powinny być ze sobą logicznie powiązane, oraz mogą korzystać z siebie nawzajem. Pojedynczym projektem może być aplikacja okienkowa, internetowa, konsolowa lub biblioteka klas.

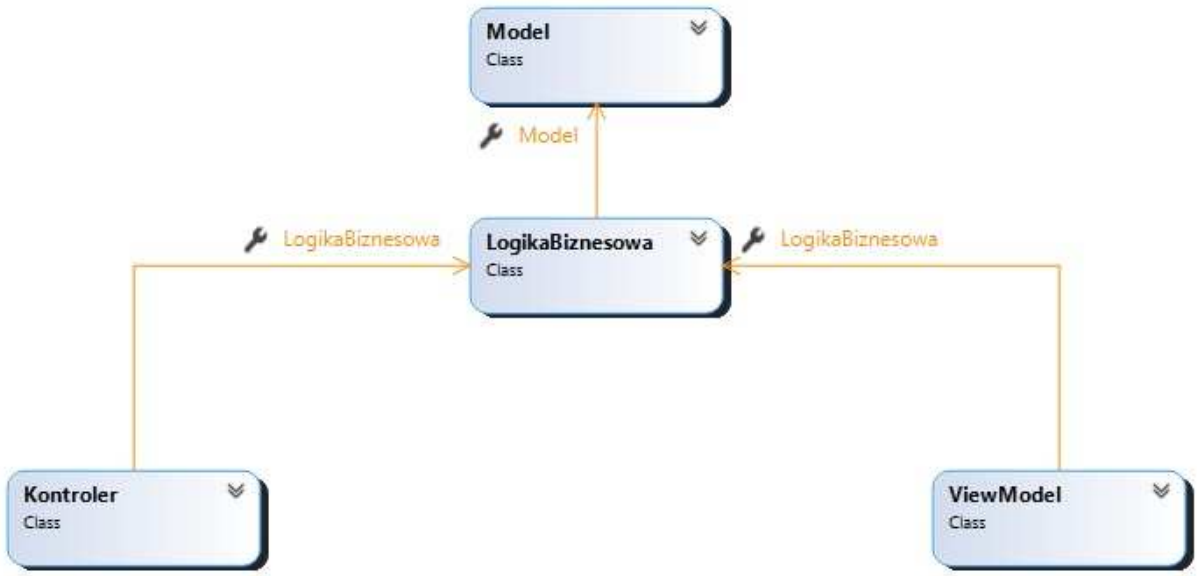
Rozdział 3. Wyjaśnienie struktury projektu jako solucji

Struktura Solucji wykorzystana w tej pracy prezentuje się następująco:

- Intertegra (cała solucja)
  - BusinessLogic (Logiki biznesowa, traktowana jako warstwa Model)
  - Intertegra (Aplikacja okienkowa (MVVM))
  - PortalWWW (Aplikacja internetowa (MVC))

W projekcie BusinessLogic znajduje się połączenie z bazą danych i klasy bazodanowe. Każdej z tych klas odpowiada jedna klasa logiki biznesowej. Aplikacja okienkowa i internetowa korzystają z tych klas aby komunikować się z bazą danych. Poniższy diagram w sposób uproszczony pokazuje powiązanie projektów.

3.1 Struktura solucji



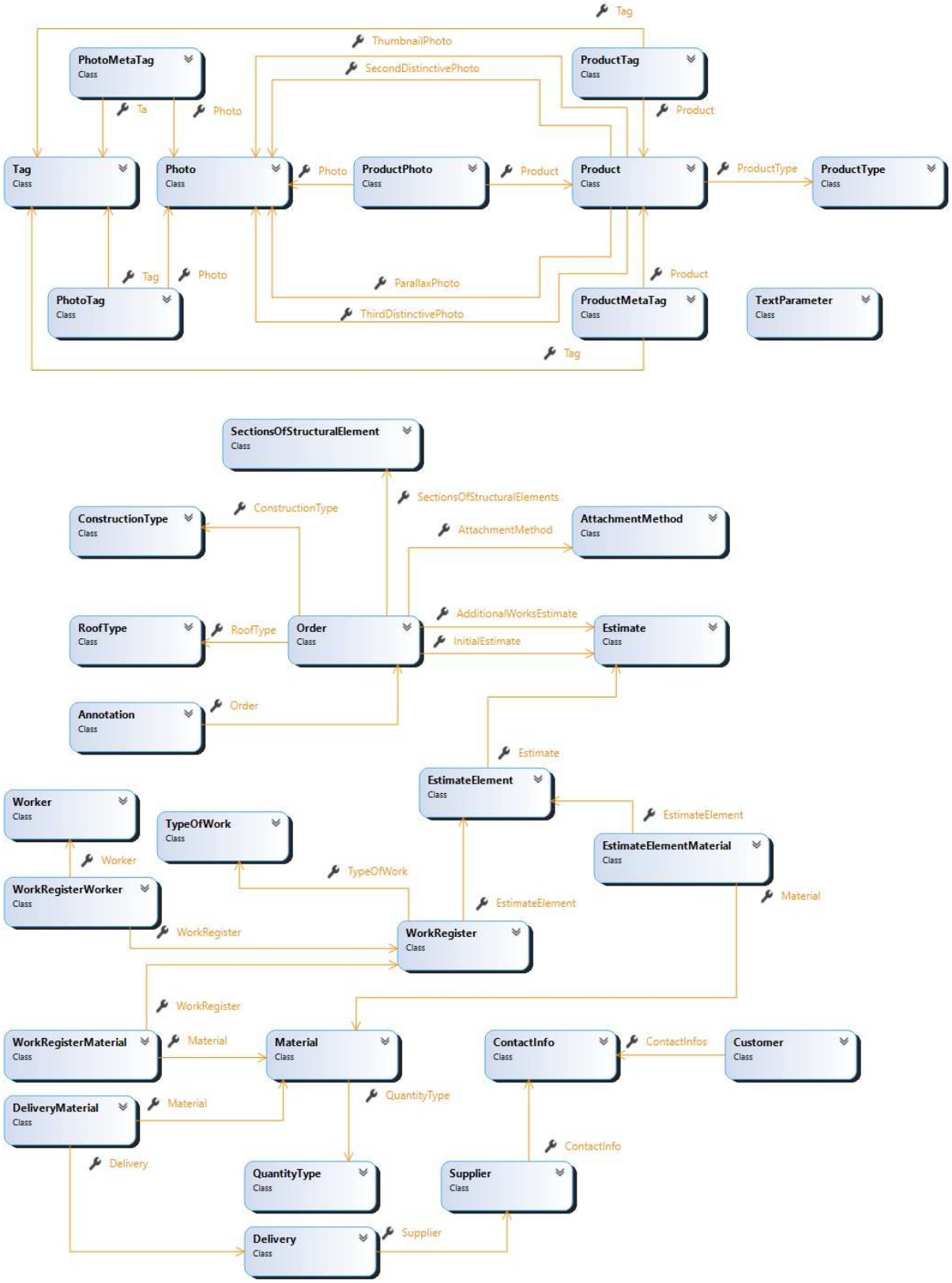
Źródło: opracowanie własne. Diagram wygenerowany przy użyciu Class designer Visual Studio

Klasa model przedstawia jedną z klas bazodanowych. Logika biznesowa odpowiada za pobieranie odpowiednich rekordów z bazy danych. Te dwie klasy znajdują się w projekcie BusinessLogic. Klasa Kontroler z projektu PortalWWW oraz ViewModel z projektu Intertegra korzystają w pełni z tej jednej klasy logiki biznesowej. Szczegółowo poszczególne projekty będą omawiane w następnych rozdziałach.

Rozdział 4. Baza danych użyta w projekcie

4.1. Diagram bazy danych

4.1 Diagram bazy danych z wykluczeniem tabel użytych do logowania



Źródło: opracowanie własne. Diagram wygenerowany przy użyciu Class designer Visual Studio

Powyższy diagram pokazuje strukturę bazy danych użytą w projekcie z wykluczeniem tabel użytych do logowanie w aplikacji internetowej. Obrazek został wygenerowany na podstawie klas bazodanowych w projekcie BusinessLogic, które to zostały wygenerowane na podstawie bazy danych SQL. Każda z tabel została wyposażona w dwie kolumny: *ID* i *IsActive*. Pierwsza z nich jest zawsze kluczem głównym typu int stworzonym z właściwością *identity*. Dzięki temu zabiegowi nie ma potrzeby wypełniania tego pola przy tworzeniu nowego rekordu, automatycznie przyjmie ono kolejną wartość. Kolumna *IsActive* spełnia funkcjonalność kosza. Przy tworzeniu bazy danych przyjęto strategię, która uniemożliwia użytkownikowi permanentne usunięcie rekordu. Może on zostać jedynie dezaktywowany, czyli umieszczony w koszu. Na diagramie pokazano tabele należące do schematu .intr, w bazie danych znajduje się drugi schemat .idt, gdzie umieszczone są tabele wykorzystane do logowania. Bazę danych można podzielić na dwie grupy. Ta widoczna na górze załączonej grafiki używana jest w PortaluWWW. Druga grupa przeznaczona jest dla aplikacji okienkowej. Zdecydowana większość relacji jest typu jeden-do-wielu. Tabele *PhotoMetaTag* oraz *PhotoTag* tworzą dwie relacje typu wiele-do-wielu pomiędzy tabelami Tag i Photo, tworząc jednocześnie podział tagów na dwie grupy: meta oraz zwykłe. Pierwsze umieszczane są w znacznikach <meta> w nagłówku, drugie przeznaczone są do wyszukiwania wewnątrz strony. Analogiczną sytuację możemy zauważyć w przypadku tabel Product, *ProductTag* i *ProductMetaTag*. Ciekawym powiązaniem jest też realcja pomiędzy Product i Photo, gdzie realcji jeden-do-wielu, do jednego Product przypisane są cztery szczególne Photo. Więcej informacji o pierwszej grupie tabel znaleźć można w rozdziale 6.

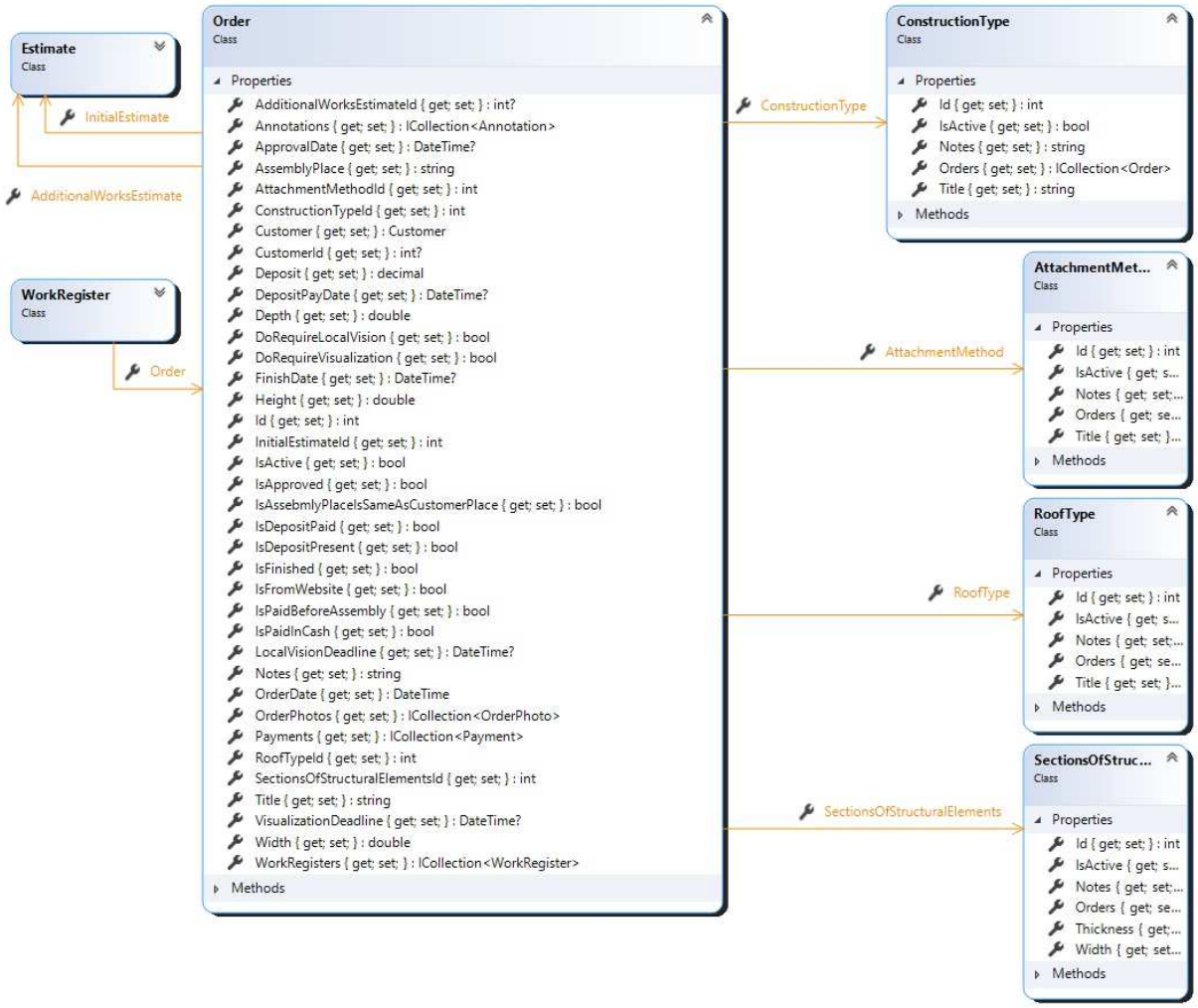
Do relacji wiele-do-wielu w drugiej grupie możemy zaliczyć:

- Worker-WorkRegister
- Material-WorkRegister
- Material-EstimateElement
- Material-Delivery

Każda z tych relacji posiada tabelę pośredniczącą. Na wyróżnienie zasługuje tutaj relacja pomiędzy Order i Estimate, gdzie do jednego Order należą dwa Estimate. Więcej o tej grupie można przeczytać w rozdziale 7.

4.2. Najważniejsze tabele

4.2 Diagram tabeli Order

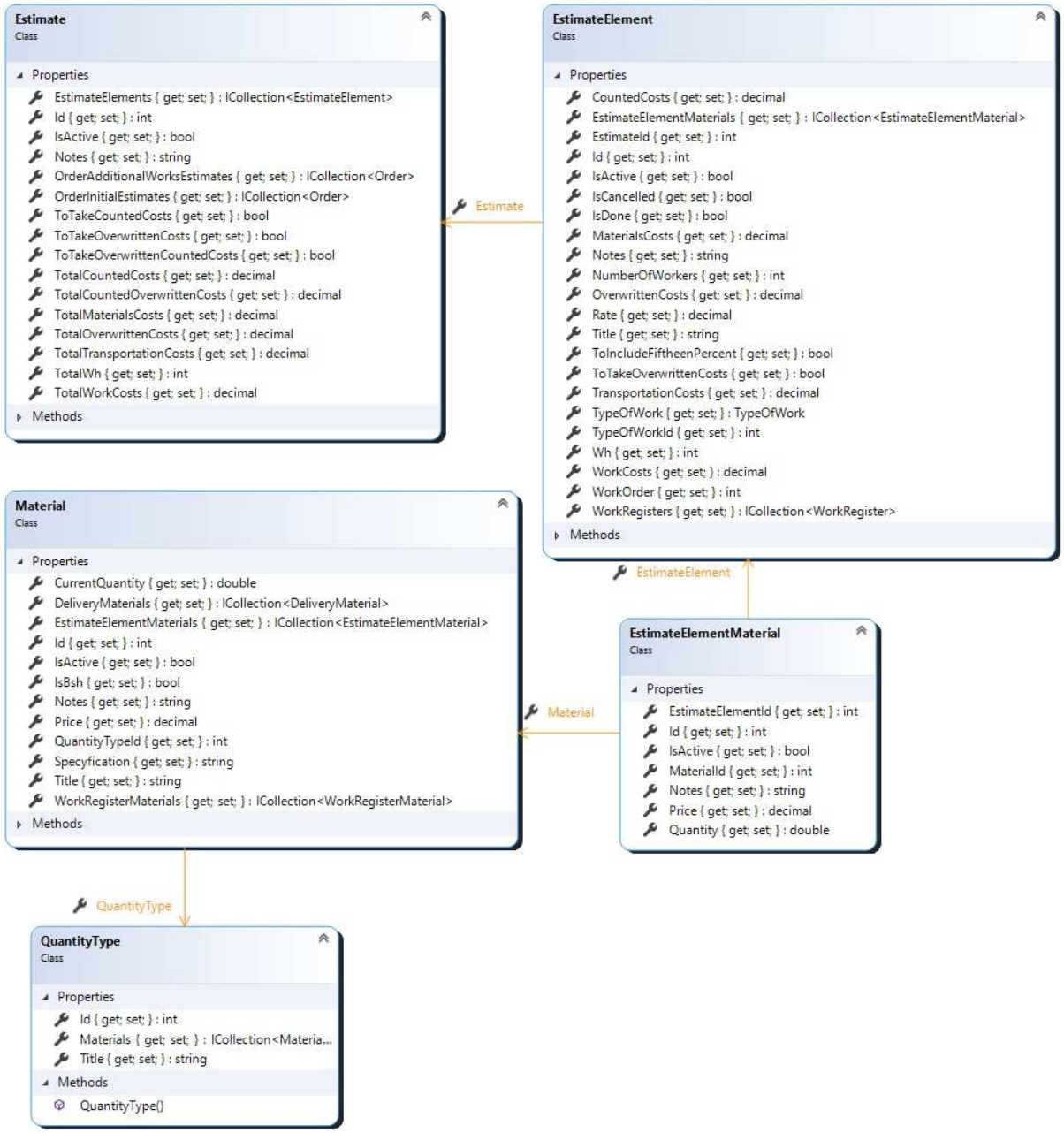


Źródło: opracowanie własne. Diagram wygenerowany przy użyciu Class designer Visual Studio

Tabela Order przedstawia zamówienia w firmie. Posiada dwa kosztorysy:

- InitialEstimate – wykorzystywany do wyceny wykonania usługi,
- AdditionalWorksEstimate – opcjonalny, umieszcza się tutaj prace, które wyniknęły w trakcie wykonywania usługi, a nie zostały uwzględnione we wcześniejszym kosztorysie. Mogą to być dodatkowe prace, których zażądał klient.

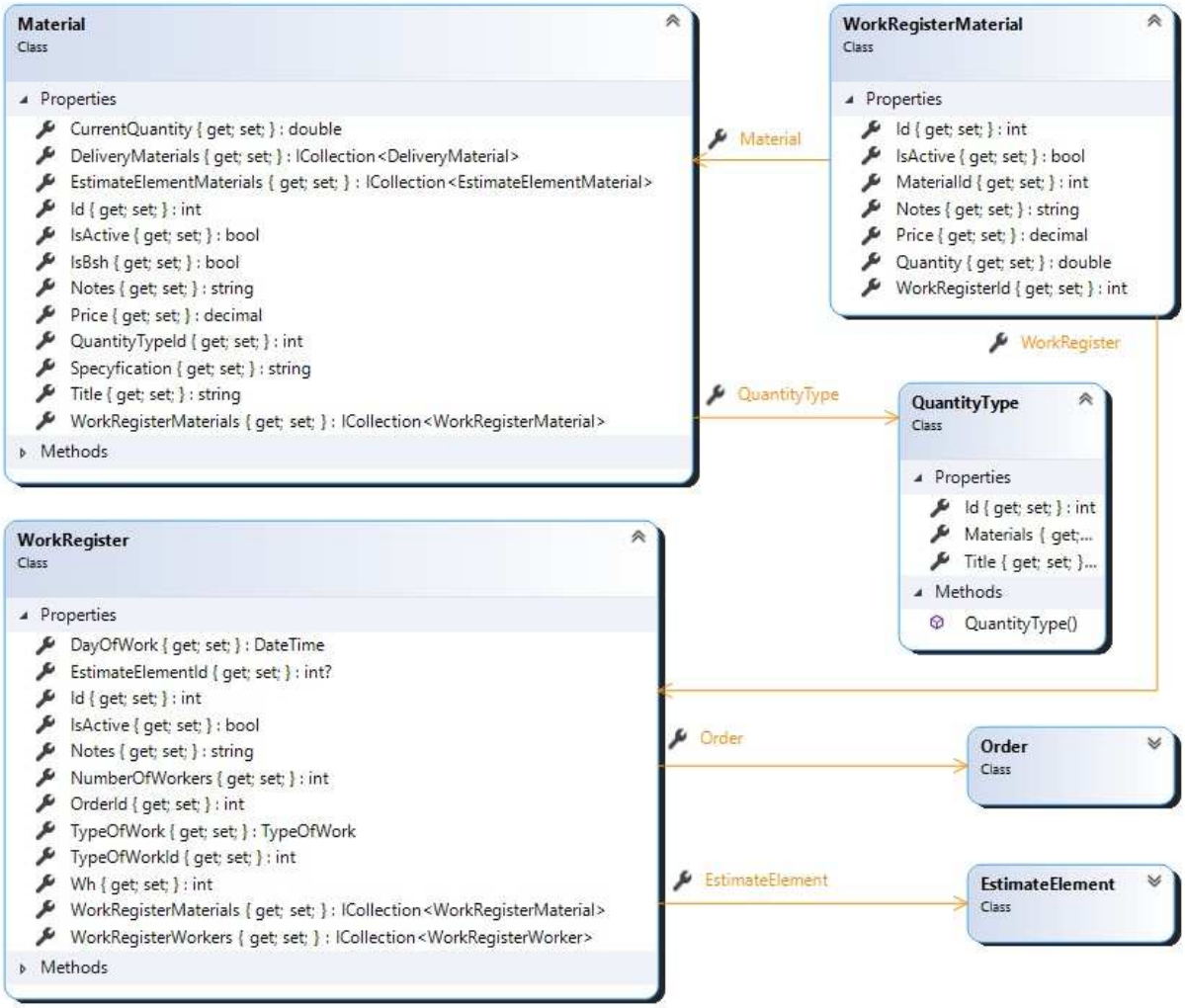
4.3 Diagram tabeli Estimate



Źródło: opracowanie własne. Diagram wygenerowany przy użyciu Class designer Visual Studio

Tabela Estimate zawiera wspomniane wyżej kosztorysy. Każdy składa się elementów kosztorysu (EstimateElements), które wyceniane są osobno. Posiadają swoje materiały, ilość pracowników, roboczogodziny i stawkę płacy. Przykładowymi elementami mogą być przygotowanie materiału lub montaż.

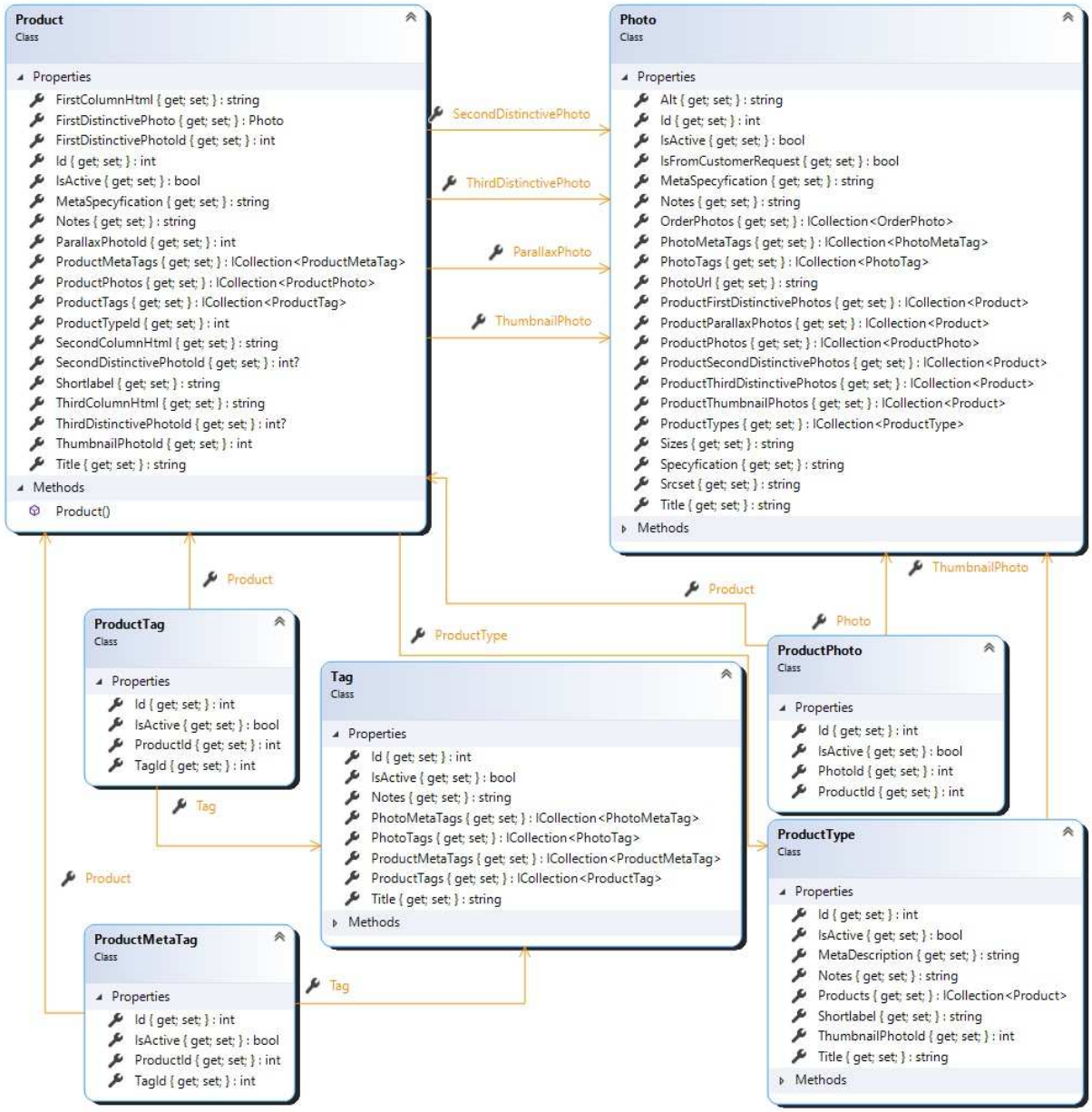
4.4 Diagram tabeli WorkRegister



Źródło: opracowanie własne. Diagram wygenerowany przy użyciu Class designer Visual Studio

WorkRegister to nic innego jak dziennik pracy. Informuje ile pracowników i przez ile godzin pracowało nad danym zamówieniem. Można także zaznaczyć jakiego elementu kosztorysu dziennik pracy dotyczył oraz dodać zużyte materiały.

4.5 Diagram tabeli Product



Źródło: opracowanie własne. Diagram wygenerowany przy użyciu Class designer Visual Studio

Tabela Product zawiera produkty, które potencjalny klient może zobaczyć na stronie internetowej. Są one podzielone na typy, zawierają trzy zdjęcia wyróżniające oraz miniaturkę wyświetlaną na liście. Za pośrednictwem tabeli ProductPhoto tworzona jest galeria zdjęć produktu. Każdy z nich posiada również meta opis, tagi (słowa kluczowe do wyszukiwania) i metatagi. Opis produktu zawiera się w trzech kolumnach HTML określonych w polach First-Secon-ThirdColumnHtml.

Rozdział 5. Projekt solucji BusinessLogic

5.1. Warstwa Model – kontekst bazy danych

W poprzednim rozdziale przedstawiony został diagram klas bazodanowych. Są one częścią warstwy Model dla pozostałych projektów. Wszystkie te klasy zostały wygenerowane na podstawie bazy danych. Użyto tutaj rozszerzeń z rodziny Entity Framework Core. Jest to Maper baz danych dla platformy .NET. Umożliwia korzystanie z zapytań LINQ, śledzenie zmian, aktualizacje i migracje schematu. Może pracować z wieloma silnikami baz danych w tym Microsoft SQL, MySQL, SQLite, Azure Cosmos DB i PostgreSQL.[6] Rozszerzenie do odzwierciedlenia bazy danych używa kontekstu i klas bazodanowych. Kontekst jest klasą, która łączy się z bazą danych i metody do zapisu zmian. Klasy bazodanowe odzwierciedlają strukturę poszczególnych tabel, a każdy pojedynczy obiekt tej klasy reprezentuje wiersz tabeli. Entity Framework Core umieszcza w klasach bazodanowych kolekcje obiektów, które odnoszą się do danego obiektu. Jeśli mamy tabelę Klient, a w niej klucz obcy do tabeli zamówienie, to w klasie bazodanowej zamówienie znajdziemy kolekcję Klienci, w której zostaną umieszczeni klienci według klucza obcego w tabeli Klienci. Do wygenerowania tych klas posłużyła komenda:

Scaffold-DbContext "<connection string>"
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -ContextDir
Models\Contexts -Context IntertegraContext -Schemas intr -
DataAnnotations -Force

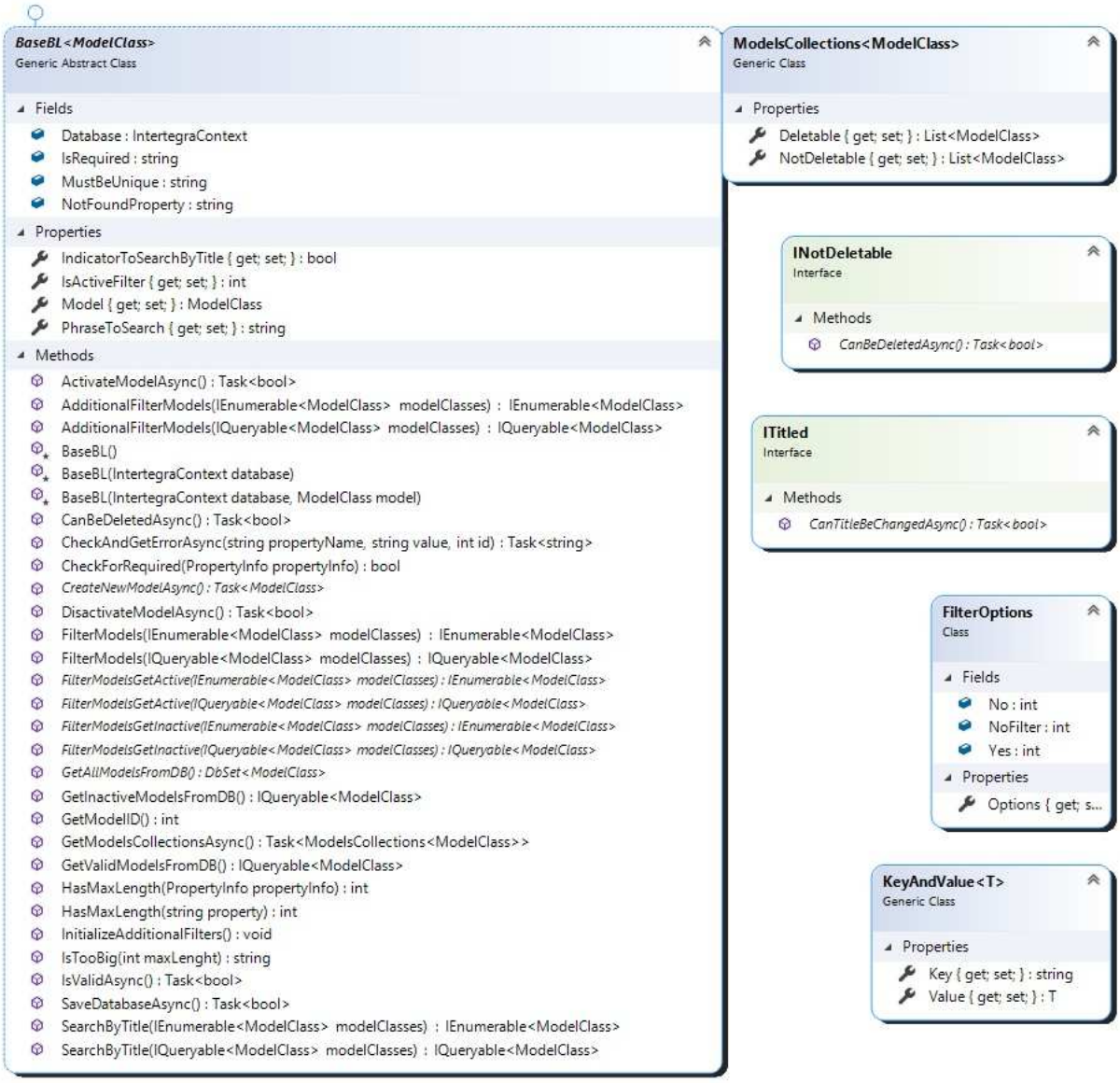
<connection string> zawiera dane do połączenia się z bazą danych. Następna część komendy określa rodzaj silnika bazy danych, a później kolejno: ścieżka, gdzie mają zostać utworzone klasy bazodanowe, ścieżka, gdzie ma zostać utworzony kontekst bazy danych, nazwa kontekstu bazy danych, schemat tabel, które mają być wykorzystane przy generowaniu klas. Dopisek *DataAnnotations* oznacza aby dołączyć do klas wspominane już wyżej DataAnnotation, natomiast *-Force* oznacza aby nadpisać ewentualnie istniejące już klasy. Ten ostatni atrybut jest konieczny gdy aktualizujemy warstwę Model po zmianach w strukturze bazy danych.

[6] https://docs.microsoft.com/en-us/ef/

W dalszej części dokumentu często używane będzie słowo *Model*, należy pod nim rozumieć obiekt jednej z klas bazodanowych, który reprezentuje rekord tabeli bazy danych. Jako *Modele* należy rozumieć zbiór tych obiektów. Kontekst bazy danych nazwano *IntertegraContext*. Przyjęto także pewne uproszczenie w nazewnictwie; zarówno pola jak właściwości (ang. properties) nazywane są po prostu polami.

5.2. Przykładowa klasa logiki biznesowej na podstawie OrderBL

5.1 Diagram klas i interfejsów logiki biznesowej



Źródło: opracowanie własne. Diagram wygenerowany przy użyciu Class designer Visual Studio

Każdy *Model* posiada odpowiadającą mu klasę logiki biznesowej. Wszystkie dziedziczą po abstrakcyjnej i generycznej klasie *BaseBL<ModelClass>*. Jako typ generyczny *ModelClass* podaje się *Model*, który ta logika biznesowa ma obsługiwać. Zaimplementowano tutaj funkcjonalności wyszukiwania i filtrowania, walidacji wprowadzonych danych, tworzenia

nowego *Modelu*, pobierania *Modeli* z i bez podziału na te, które można usunąć, aktywowania i dezaktywowania *Modelu*. Ta klasa implementuje interfejs *INotDeleteble*.

- Database – kontekst bazy danych, przez niego odbywa się pobieranie i zapis danych,
- IsRequired – wykorzystywane przy walidacji,
- MustBeUnique – wykorzystywane przy walidacji,
- NotFoundProperty – wykorzystywane przy walidacji,
- IndicatorToSearchByTitle – znacznik, który informuje, że wyszukiwanie musi odbyć się po tytule. Niektóre z klas dziedziczących mogą dodać więcej podobnych znaczników,
- IsActiveFilter – stan filtrowania po aktywności Modelu. Więcej o filtrowaniu znajduje się poniżej,
- Model – bezpośrednie odniesienie to klasy Model, reprezentującej jeden z wierszy tabeli bazy danych. To pole może przyjmować wartość null.

ModelsCollections zawiera w sobie dwie kolekcje Modeli, te które można usunąć oraz, te, których nie można. Takie rozwiązanie wykorzystane jest w aplikacji internetowej, gdzie druga grupa Modeli pozbawiona jest przycisku do usunięcia.

Interfejs *ITitled* służy do oznaczenia klas logiki biznesowej, których tytuł *Modelu* w pewnych przypadkach nie może być zmieniony. Wykorzystywane jest to m.in. w przypadkach kiedy tytuł musi być unikalny.

Celem usystematyzowania filtrów stworzono klasę *FilterOptions*. Wprowadza ona trzy wartości: tak, nie, bez filtru. Każda z tych opcji ma przypisany numer. Ponadto klasa ta zawiera słownik, którego kluczem jest numer, a wartością nazwa do wyświetlenia opcji. Ta kolekcja jest wykorzystywana jako elementy listy filtrów.

Każda z klas logiki biznesowej ma nazwę według wzoru: Model+BL. Poniżej przedstawiono logikę biznesową na podstawie *OrderBL*.

Konstruktory

5.2 Bazowa logika biznesowa - konstruktory

```
36          protected BaseBL()
37          {
38              if (typeof(ModelClass).GetProperty("Id") == null || typeof
                  (ModelClass).GetProperty("IsActive") == null)
39                  throw new NotImplementedException("Model nie zawiera pola Id
                        lub IsActive");
40              Database = new IntertegraContext();
41              IsActiveFilter = FilterOptions.Yes;
42              IndicatorToSearchByTitle = true;
43              InitializeAdditionalFilters();
44          }
45
46          protected BaseBL(IntertegraContext database)
47          {
48              Database = database;
49              IsActiveFilter = FilterOptions.Yes;
50              IndicatorToSearchByTitle = true;
51              InitializeAdditionalFilters();
52          }
53
54          protected BaseBL(IntertegraContext database, ModelClass model)
55          {
56              Model = model;
57              Database = database;
58              IsActiveFilter = FilterOptions.Yes;
59              IndicatorToSearchByTitle = true;
60              InitializeAdditionalFilters();
61          }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Tak wyglądają konstruktury klasy bazowej. Każdy z nich tworzy nowy kontekst bazy danych, przypisuje filtrowi *IsActive* wartość *Tak*, aby na początku wyświetlić tylko Modele nie znajdujące się w koszu, ustawia znacznik wyszukiwania po tytule. *InitializeAdditionalFilters* jest funkcją abstrakcyjną do nadpisania w klasach dziedziczących, jak nazwa wskazuje służy do zainicjowania dodatkowych filtrów analogicznie jak zostało to zrobione w przypadku *IsActiveFilter*.

5.3 Przykładowa logika biznesowa - konstruktory

```
17          public OrderBL() : base() { }
18
19          public OrderBL(IntertegraContext database) : base(database) { }
20
21          public OrderBL(IntertegraContext database, Order model) : base
              (database, model) { }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

W przypadku docelowej klasy logiki biznesowej konstruktory jedynie wywołują te bazowe.

Tworzenie nowego Modelu

5.4 Przykładowa logika biznesowa - tworzenie nowego Modelu

```
84          public override async Task<Order> CreateNewModelAsync()
85          {
86              Model = new Order()
87              {
88                  AttachmentMethod = await Database.AttachmentMethods.FirstAsync
                      (),
89                  ConstructionType = await Database.ConstructionTypes.FirstAsync
                      (),
90                  InitialEstimate = await new EstimateBL
                      (Database).CreateNewModelAsync(),
91                  IsAssebmlyPlaceIsSameAsCustomerPlace = true,
92                  IsPaidInCash = true,
93                  OrderDate = DateTime.Now,
94                  RoofType = await Database.RoofTypes.FirstAsync(),
95                  SectionsOfStructuralElements = await
                      Database.SectionsOfStructuralElements.FirstAsync(),
96                  IsApproved = true,
97                  ApprovalDate = DateTime.Now,
98                  IsActive = true
99              };
100             await Database.Orders.AddAsync(Model);
101             return Model;
102         }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Nowy *Model* zawsze tworzony jest przez przeznaczoną mu logikę biznesowej metodą asynchroniczną *CreateNewModelAsync()* wypełnianą w klasie dziedziczącej. Przypisywane są tutaj wartości domyślne oraz inicjalizowane niezbędne obiekty. Nowo utworzony obiekt zawsze przypisywany jest do pola *Model*. Jeśli występuje potrzeba utworzenia i przypisania innego Modelu, tworzy się na tą potrzebę logikę biznesową wykorzystując kontruktor z podaniem kontekstu bazy danych. Taki zabieg możemy zobaczyć na powyższej grafice w przypadku pola *InitialEstimate*. W tej metodzie pole *IsActive* zawsze ma przypisywaną wartość *true*. Po utworzeniu Model dodawany jest do kontekstu bazy danych oraz zwracany. Nie są zapisywane tutaj zmiany w bazie danych, należy to zrobić ręcznie używając metody *SaveDatabaseAsync()*. Zmiany nie są zwykle zapisywane bezpośrednio po wywołaniu powyższej metody. W wielu wypadkach wywołałoby to nawet błędy. Nie zawsze wypełnione zostaną tutaj także wszystkie wymagane pola. W tym przykładzie do poprawnego zapisu potrzeba uzupełnić *Title (pol. Tytuł)*, to zadanie należy najczęściej do użytkownika.

Aktywacja i dezaktywacja Modelu (usuwanie i przywracanie z kosza)

5.5 Przykładowa logika biznesowa - aktywacja i dezaktywacja Modelu

```
154         public async Task<bool> DisactivateModelAsync()
155         {
156             if (await CanBeDeletedAsync())
157             {
158                 Model.GetType().GetProperty("IsActive").SetValue(Model,
                      false);
159                 return true;
160             }
161             return false;
162         }
163
164         public async Task<bool> ActivateModelAsync()
165         {
166             if (await CanBeDeletedAsync())
167             {
168                 Model.GetType().GetProperty("IsActive").SetValue(Model, true);
169                 return true;
170             }
171             return false;
172         }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Aktywacja i dezaktywacja Modelu została zrealizowana w bazowej klasie logiki biznesowej dwoma metodami widocznymi na powyższej grafice. Te metody korzystają z założenia przyjętego podczas tworzenia bazy danych, które mówi, że każda tabela musi zawierać kolumnę *IsActive* typu logicznego (w języku SQL typu bit).

Filtrowanie i wyszukiwanie

Filtrowanie i wyszukiwanie odbywa się na podstawie ustawionych w logice biznesowych pól. W bazowej klasie znajduje się filtr aktywności i wyszukiwany tekst oraz znacznik wyszukiwania po tytule.

5.6 Przykładowa logika biznesowa - filtry i wyszukiwanie 1

```
177         public int IsActiveFilter { get; set; }
178
179         public bool IndicatorToSearchByTitle { get; set; }
180
181         public string PhraseToSearch { get; set; }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Omawiana klasa OrderBL dodaje kilka innych filtrów i znaczników.

5.7 Przykładowa logika biznesowa - filtry i wyszukiwanie 2

```
123         public bool IndicatorToSearchByCustomer { get; set; }
124
125         public int IsFinishedFilter { get; set; }
126
127         public int DoesFinishDatePassedFilter { get; set; }
128
129         public int HasCustomerFilter { get; set; }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Pierwszy z nich informuje, że wyszukiwanie powinno odbywać się po kliencie. Filtr *IsFinished*, decyduje o filtrowaniu po tym, czy zamówienie zostało zakończone, następny po tym, czy minęła planowana data zakończenia i ostatni z nich po tym czy zamówienie posiada klienta. W praktyce wyszukiwanie i filtrowanie odbywa się, najpierw w klasie bazowej.

5.8 Przykładowa logika biznesowa - filtry i wyszukiwanie 3

```
202         public IQueryable<ModelClass> FilterModels(IQueryable<ModelClass>
              modelClasses)
203         {
204             if (IsActiveFilter == FilterOptions.Yes)
205                 modelClasses = FilterModelsGetActive(modelClasses);
206             else if (IsActiveFilter == FilterOptions.No)
207                 modelClasses = FilterModelsGetInactive(modelClasses);
208             return !PhraseToSearch.IsNullOrEmpty() &&
                  IndicatorToSearchByTitle ? SearchByTitle(AdditionalFilterModels
                  (modelClasses)) : AdditionalFilterModels(modelClasses);
209         }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

W liniach 202-207 na podstawie wartości *IsActiveFilter* wybierane są aktywne, lub nieaktywne *Modele*. Następnie sprawdzane jest czy wprowadzona została fraza do wyszukiwania i czy zaznaczony jest znacznik do wyszukiwania po tytule. Jeśli tak wywoływane jest wyszukiwanie po tytule, jako argument podając metodę *AdditionalFilterModels*. Jest to metoda wirtualna, która domyślnie zwraca kolekcję modeli podana jako argument. Służy ona do ewentualnego nadpisania w klasie dziedziczącej jeśli występuje potrzeba dodatkowego filtrowania. Jeśli znacznik do wyszukania po tytule nie został zaznaczony, wywołana zostaje sama funkcja dodatkowego filtrowania. W obu przypadkach zwracane są Modele w pełni przefiltrowane. Aby cała operacja przebiegła pomyślnie trzeba oczywiście wcześniej ustawić wartości filtrów. Metoda ta została napisana w taki sposób, że można jej użyj dla dowolnej kolekcji Modeli, nie koniecznie muszą to być wszystkie Modele z bazy danych. Dzięki temu możemy przykładowo najpierw pobrać zamówienia należące do

klienta, a następnie poddać je filtrowaniu. Przy najczęstszym scenariuszu, kiedy znajdujemy się na widoku wszystkich Modeli, używana jest metoda *GetValidModels*, która wywołuje powyższą z podaniem wszystkich Modeli.

5.9 Przykładowa logika biznesowa - filtry i wyszukiwanie 4

```
140         public IQueryable<ModelClass> GetValidModelsFromDB() => FilterModels
              (GetAllModelsFromDB());
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Używana jest tutaj metoda abstrakcyjna *GetAllModelsFormDB*, która przeznaczona jest do wypełniania w klasie dziedziczącej i zwraca wszystkie *Modele*.

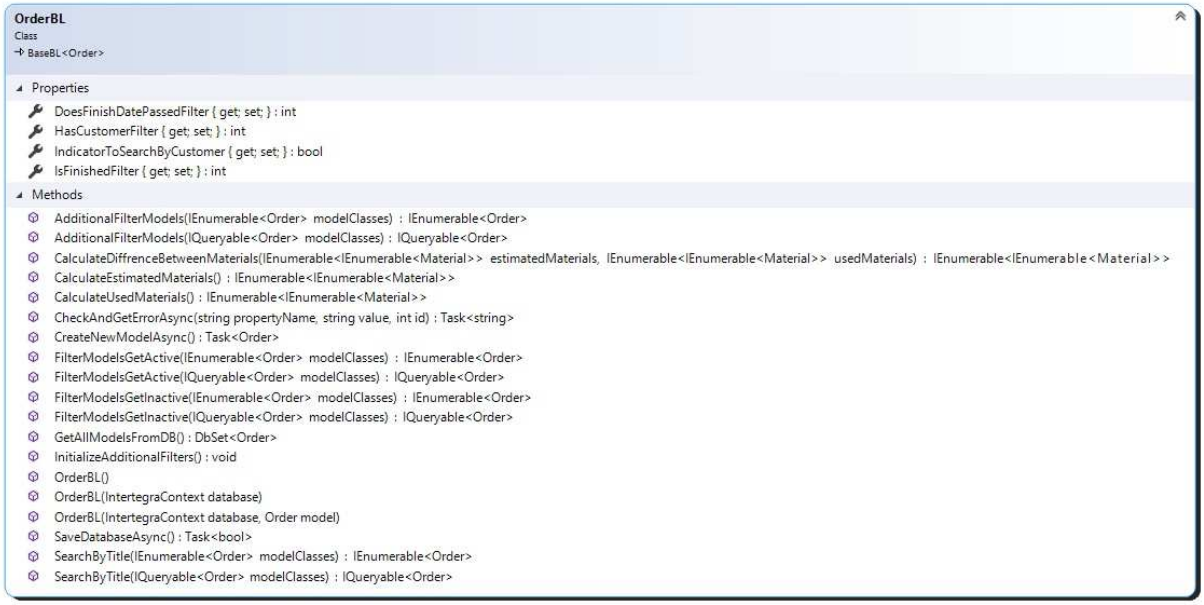
5.10 Przykładowa logika biznesowa - filtry i wyszukiwanie 5

```
133         public override IQueryable<Order> AdditionalFilterModels
              (IQueryable<Order> modelClasses)
134         {
135             if (DoesFinishDatePassedFilter == FilterOptions.Yes)
136                 modelClasses = modelClasses.Where(item => item.FinishDate <
                      DateTime.Today);
137             else if (DoesFinishDatePassedFilter == FilterOptions.No)
138                 modelClasses = modelClasses.Where(item => item.FinishDate >=
                      DateTime.Today);
139
140             if (HasCustomerFilter == FilterOptions.Yes)
141                 modelClasses = modelClasses.Where(item => item.Customer !=
                      null);
142             else if (HasCustomerFilter == FilterOptions.No)
143                 modelClasses = modelClasses.Where(item => item.Customer ==
                      null);
144
145             if (IsFinishedFilter == FilterOptions.Yes)
146                 modelClasses = modelClasses.Where(item => item.IsFinished);
147             else if (IsFinishedFilter == FilterOptions.No)
148                 modelClasses = modelClasses.Where(item => !item.IsFinished);
149
150             if (IndicatorToSearchByCustomer && !PhraseToSearch.IsNullOrEmpty
                  ())
151                 modelClasses = modelClasses.Where(item => item.Customer !=
                      null && ((item.Customer.FirstName != null &&
                      item.Customer.FirstName.ToLower().Contains(PhraseToSearch))
                      || (item.Customer.LastName != null &&
                      item.Customer.LastName.ToLower().Contains
                      (PhraseToSearch))));
152
153             return modelClasses;
154         }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Tak wyglądają dodatkowe filtrowanie w klasie *OrderBL*. Filtrowanie odbywa się tutaj po wyżej wspomnianych polach. Przyjrzyjmy się jeszcze na koniec całej klasie *OrderBL*.

5.11 Diagram klasy OrderBL



Źródło: opracowanie własne. Diagram wygenerowany przy użyciu Class designer Visual Studio

Zauważymy tutaj, że każda metoda filtrująca występuje w dwóch wersjach. Jedna przyjmuje jako argument kolekcję typu *IEnumerable*, a druga *IQueryable*. Jest to spowodowane pewnymi różnicami w metodach tych klas. Nie byłoby dobrym pomysłem rzutowanie *Modeli* do jednego typu. Jeśli *IEnumerable*, rzutujemy na *IQueryable*, nawet przy użyciu *AsQueryable*, wywoła to błędy przy użyciu metod asynchronicznych. Podczas odwrotnego zabiegu spowodujemy przedwczesne pobranie modeli z bazy. *IQueryable* ma tą zaletę, że Modele mogą zostać pobrane dopiero po ich przefiltrowaniu. Jeśli baza danych będzie posiadała wiele rekordów z pewnością wpłynie to pozytywnie na wydajność. Warto też tutaj wspomnieć, że metoda *SearchByTitle* jest metodą wirtualną klasy bazowej. Powinna być zawsze nadpisana w klasie dziedziczącej.

Walidacja

Walidacja zrealizowana jest w trakcie wprowadzania danych, oczywiście za pośrednictwem logiki biznesowej. Każdorazowo po zmianie danych są one wysyłane do sprawdzenia. Następnie zwracany jest pusty tekst lub komunikat z błędem. To pierwsze oczywiście oznacza, że walidacja przebiegła pomyślnie i można zapisać zmiany. Dla każdego pola ta walidacja odbywa się osobno. Funkcja nigdy nie może zwracać null, wywołałoby to różne komplikacje.

5.12 Bazowa logika biznesowa - walidacja

```
302         public virtual async Task<string> CheckAndGetErrorAsync(string
              propertyName, string value, int id)
303         {
304             PropertyInfo propertyInfo = typeof(ModelClass).GetProperty
                  (propertyName);
305             if (propertyInfo == null)
306                 return await Task.FromResult(NotFoundProperty);
307             if (CheckForRequired(propertyInfo) && (value == null || value.Trim
                  () == ""))
308                 return await Task.FromResult(IsRequired);
309             int maxLenght = HasMaxLength(propertyInfo);
310             if (value != null && maxLenght != 0 && value.Length > maxLenght)
311                 return await Task.FromResult(IsTooBig(maxLenght));
312             return "";
313         }
314
315         public bool CheckForRequired(PropertyInfo propertyInfo)
316         {
317             Attribute attribute = propertyInfo.GetCustomAttribute(typeof
                  (RequiredAttribute));
318             return attribute != null;
319         }
320
321         public int HasMaxLength(PropertyInfo propertyInfo)
322         {
323             Attribute attribute = propertyInfo.GetCustomAttribute(typeof
                  (StringLengthAttribute));
324             return attribute != null ? ((StringLengthAttribute)
                  attribute).MaximumLength : 0;
325         }
326
327         public int HasMaxLength(string property) => HasMaxLength(GetType
              ().GetProperty(property));
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Do metody walidującej przekazywana jest nazwa pola, jego wartość rzutowana do *string* oraz id *Modelu*. Na początku sprawdzane jest, czy w ogóle znajduje się pole o takiej nazwie. Następnie na podstawie *DataAnnotation Modelu*, wygenerowanych przy generowaniu warstwy *Model* sprawdza się czy pole jest wymagane i czy posiada maksymalną długość, jeśli tak to czy nie została przekroczona. W przypadku sprawdzania czy pole posiada maksymalną długość, zwracane jest 0 jeśli jej nie posiada. Metoda ta jest wirtualna, aby można było przeprowadzić dodatkowe filtrowanie w klasie dziedziczącej. Zostało to zrobione w omawianej teraz klasie *OrderBL*.

5.13 Przykładowa logika biznesowa - walidacja

```
195         public override async Task<string> CheckAndGetErrorAsync(string
              propertyName, string value, int id)
196         {
197             string error = await base.CheckAndGetErrorAsync(propertyName,
                  value, id);
198             if (error != "")
199                 return error;
200             return propertyName switch
201             {
202                 nameof(Model.Title) => (await Database.Orders.Where(item =>
                      item.Title.Trim() == value.Trim() && item.Id !=
                      id).FirstOrDefaultAsync()) != null ? MustBeUnique : "",
203                 _ => "",
204             };
205         }
```

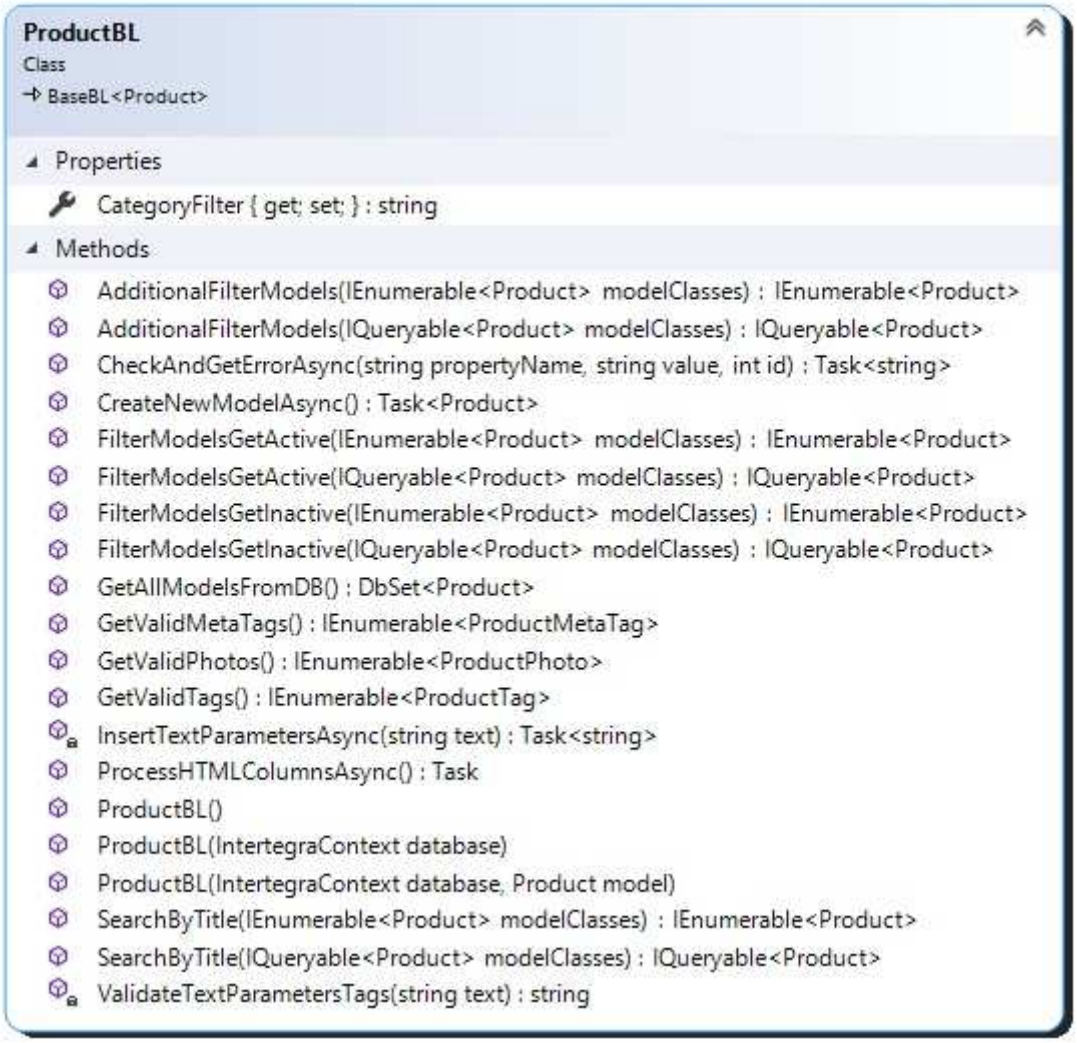
Źródło: opracowanie własne. Wydrukowano z Visual Studio

Podczas stosowania dodatkowej walidacji, na początku zawsze sprawdzane jest czy ta podstawowa nie pokazała błędu. Jest on ewentualnie od razu zwracany. W tym przypadku sprawdzane jest czy Tytuł jest unikalny. Dlatego też podczas sprawdzania potrzebne było podanie Id Modelu.

Klasa *OrderBL* posiada jeszcze metody obliczające zużyte i planowane materiały, oraz różnice między nimi.

5.3. Logika biznesowa ProductBL

5.14 Diagram klasy ProductBL



Źródło: opracowanie własne. Diagram wygenerowany przy użyciu Class designer Visual Studio

Model Product przedstawia produkt w aplikacji internetowej. Logika biznesowa wyróżnia się walidacją i przetwarzaniem kolumn HTML.

Przetwarzanie kolmn HTML

Ciekawą funkcją przy wypełnianiu kolumn HTML produktu jest możliwość wstrzyknięcia zawartości parametru. Parametry znajdują się w tabeli *TextParameter*. Składa się on z tytułu i zawartości (oraz oczywiście tak jak każda tabela z kolumn Id i IsActive). Aby umieścić taki parametr w treści należy napisać znacznik: {#Nazwa parametru#} lub {#Id parametru#}. Z uwagi na fakt, że tytuł można zmienić, zaleca się korzystanie z opcji zawierającej Id. Cała operacje realizowana jest poniższą metodą.

5.15 Logika biznesowa Product - przetwarzanie kolumn HTML

```
52          private async Task<string> InsertTextParametersAsync(string text)
53          {
54              if (text.IsNullOrEmpty())
55                  return text;
56              int parametrStart, parametrEnd, position = 0;
57              string parametrTitle;
58              parametrStart = text.IndexOf("{#", position);
59              parametrEnd = text.IndexOf("#}", position);
60              TextParameterBL textParameterBL = new TextParameterBL(Database);
61              // Ograniczenie wywołania pętli do 100 razy na wypadek zapętlenia
                  wstrzykiwanych parametrów.
62              int i = 0;
63              IQueryable<TextParameter> textParameters =
                  textParameterBL.FilterModelsGetActive
                  (textParameterBL.GetAllModelsFromDB());
64              while (position < text.Length && parametrStart != -1 &&
                  parametrEnd != -1 && i < 100)
65              {
66                  i++;
67                  // Jeśli znacznik początkowy jest dalej niż znacznik końcowy
                      lub jeśli długość tekstu pomiędzy znacznikami jest większy
                      niż 32 (długość nazwy parametru)
68                  if (parametrStart > parametrEnd || parametrEnd - parametrStart
                      - 2 > 32)
69                  {
70                      position = parametrEnd + 3;
71                  }
72                  else
73                  {
74                      parametrTitle = text[(parametrStart + 2)..parametrEnd];
75                      if (int.TryParse(parametrTitle, out int id))
76                          text = text.Replace("{#" + parametrTitle + "#}", (await
                           textParameters.FirstOrDefaultAsync(item => item.Id ==
                          id))?.Content);
77                      text = text.Replace("{#" + parametrTitle + "#}", (await
                          textParameters.FirstOrDefaultAsync(item => item.Title ==
                          parametrTitle))?.Content);
78                      position = parametrStart;
79                  }
80                  parametrStart = text.IndexOf("{#", position);
81                  parametrEnd = text.IndexOf("#}", position);
82              }
83              return text;
84          }
85
86          public async Task ProcessHTMLColumnsAsync()
87          {
88              Model.FirstColumnHtml = await InsertTextParametersAsync
                  (Model.FirstColumnHtml);
89              Model.SecondColumnHtml = await InsertTextParametersAsync
                  (Model.SecondColumnHtml);
90              Model.ThirdColumnHtml = await InsertTextParametersAsync
                  (Model.ThirdColumnHtml);
91          }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Jako parametr przyjmuje się tekst, który zawiera znaczniki wstrzyknięcia parametru, jeśli jest pusty, od razu się go zwraca. Na początku tworzy się kilka zmiennych:

- parametrStart – pozycja w tekście, na której zaczyna się znacznik parametru,
- parametrEnd – pozycja w tekście, na której kończy się znacznik parametru,
- position – pozycja w tekście, od której zaczyna się wyszukiwanie,
- parametrTitle – tytuł lub Id znalezionego parametru,
- textParameterBL – logika biznesowa służącą do pobierania znalezionych parametrów,
- i – licznik bezpieczeństwa.

Wszystkie pozycje ustawiane są na zero. Wyszukiwanie i wstrzykiwanie parametrów odbywa się w pętli while, która sprawdza czy pozycja wyszukiwania (zmienna position) nie jest większa od długości tekstu, oraz czy licznik bezpieczeństwa, który jest zawsze zwiększany na początku wywołania pętli, nie przekroczył wartości 100. Bez takiego limitu, jeśli parametr zawierałby znacznik odwołujący się do samego siebie, pętla nigdy by się nie skończyła. Na początku pętli (za pierwszym razem dzieje się to przed pętla while) znajdowana jest pozycja początkowa i końcowa znacznika parametru. Następnie sprawdzane jest czy te pozycje nie mijają się, czyli czy koniec jest dalej od początku znacznika, oraz czy nazwa nie przekracza 32 znaków, ten limit ustawiony jest w bazie danych. Jeśli któryś z tych warunków nie został spełniony pozycja wyszukiwania ustawiana jest za końcem parametru, tak aby przy następnym wywołaniu ten znacznik został pominięty. Jeśli wszystko się zgadza, wydobywany jest tytuł tekst pomiędzy początkiem i końcem parametru. Za pomocą funkcji *TryParse* sprawdzane jest czy znaleziony tekst jest cyfrą, jeśli tak wyszukiwany jest parametr na podstawie Id, w przeciwnym wypadku na podstawie tytułu i od razu cały znacznik parametru zamieniany jest na zawartość znalezionego parametru (lub na pusty string jeśli parametr nie został znaleziony). Na samym końcu wyszukiwane są pozycje początku i końca znacznika parametru i petla wywołuje się jeszcze raz. Po jej zakończeniu zwracany jest tekst ze wstrzykniętymi już parametrami. Ta metoda wywoływana jest dla każdej kolumny HTML.

Walidacja

5.16 Logika biznesowa Product - walidacja 2

```
155         public override async Task<string> CheckAndGetErrorAsync(string
              propertyName, string value, int id)
156         {
157             string error = await base.CheckAndGetErrorAsync(propertyName,
                  value, id);
158             if (error != "")
159                 return error;
160             switch (propertyName)
161             {
162                 case nameof(Model.Title):
163                     if (value.Trim().ToLower() == "typ")
164                         return "Ta nazwa jest zabroniona";
165                     if ((await GetValidModelsFromDB().Where(item =>
                        item.Title.Trim() == value.Trim() && item.Id !=
                        id).FirstOrDefaultAsync()) != null)
166                         return MustBeUnique;
167                     return "";
168                 case nameof(Model.FirstColumnHtml):
169                     return ValidateTextParametersTags(value);
170                 case nameof(Model.SecondColumnHtml):
171                     return ValidateTextParametersTags(value);
172                 case nameof(Model.ThirdColumnHtml):
173                     return ValidateTextParametersTags(value);
174                 default:
175                     return "";
176             }
177         }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Walidacja na początku sprawdza czy jako tytuł nie ustawia się „typ”. Ta nazwa jest zabroniona ze względu na konflikt w adresach aplikacji internetowej. Produkty widoczne są pod linkiem */nazwaProduktu*. Natomiast pod linkiem */typ* znajduje się strona z kategoriami produktów. Następnie, jeśli walidowana jest kolumna HTML, wywoływana jest poniższa metoda.

5.17 Logika biznesowa Product - walidacja 1

```
127         private static string ValidateTextParametersTags(string text)
128         {
129             if (text.IsNullOrEmpty())
130                 return "";
131             int parametrStart, parametrEnd, position = 0;
132             parametrStart = text.IndexOf("{#", position);
133             parametrEnd = text.IndexOf("#}", position);
134             if (parametrEnd == -1 && parametrStart == -1)
135                 return "";
136             while (position < text.Length)
137             {
138                 if (parametrEnd == -1 && parametrStart == -1)
139                     return "";
140                 if (parametrStart != -1 && parametrEnd == -1)
141                     return "Odnaleziono znacznik początkowy wstrzykiwania
                    parametru ale nie odnaleziono odpowiadającego mu znacznika
                     końcowego. Poprawny zapis: {#Nazwa parametru#}.";
142                 if (parametrStart == -1 && parametrEnd != -1)
143                     return "Odnaleziono znacznik końcowy wstrzykiwania
                    parametru ale nie odnaleziono odpowiadającego mu znacznika
                     początkowego. Poprawny zapis: {#Nazwa parametru#}.";
144                 if (parametrStart > parametrEnd)
145                     return "Znaczniki początkowy i końcowy wszykiwania
                    parametru się mijają! Poprawny zapis: {#Nazwa
                    parametru#}.";
146                 if (parametrEnd - parametrStart - 2 > 32)
147                     return "Nazwa parametru pomiędzy znakami wstrzykiwania
                    parametru nie może być dłuższa niż 32 znaki.";
148                 position = parametrEnd + 2;
149                 parametrStart = text.IndexOf("{#", position);
150                 parametrEnd = text.IndexOf("#}", position);
151             }
152             return "";
153         }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Sprawdzana jest poprawność znaczników wstrzykiwania parametru:

- czy odnaleziono pasujące pary początku i końca znacznika,
- czy koniec i początek znacznika mijają się,
- czy nazwa parametru w znaczniku nie przekracza 32 znaków

5.4. Logika biznesowa kosztorysów – obliczanie kosztów

5.18 Logika biznesowa Estimate - obliczanie kosztów

```
26          private void CalculateCountedCosts() => Model.TotalCountedCosts = new
              EstimateElementBL(Database).FilterModels(Model.EstimateElements).Sum
              (item => item.CountedCosts);
27
28          private decimal CalculateOverwrittenCountedCosts() =>
              Model.TotalCountedCosts = new EstimateElementBL
              (Database).FilterModels(Model.EstimateElements).Sum(item =>
              item.ToTakeOverwrittenCosts ? item.OverwrittenCosts :
              item.CountedCosts);
29
30          private decimal CalculateTotalWorkCosts() => Model.TotalWorkCosts = new
               EstimateElementBL(Database).FilterModels(Model.EstimateElements).Sum
              (item => item.WorkCosts);
31
32          private decimal CalculateTotalTransportationCosts() =>
              Model.TotalTransportationCosts = new EstimateElementBL
              (Database).FilterModels(Model.EstimateElements).Sum(item =>
              item.TransportationCosts);
33
34          private decimal CalculateTotalWH() => Model.TotalWh = new
              EstimateElementBL(Database).FilterModels(Model.EstimateElements).Sum
              (item => item.Wh);
35
36          public void CalculateCosts()
37          {
38              if (Model.ToTakeCountedCosts)
39              {
40                  CalculateCountedCosts();
41              }
42              else if (Model.ToTakeOverwrittenCountedCosts)
43              {
44                  CalculateOverwrittenCountedCosts();
45              }
46              else
47                  Model.TotalCountedCosts = Model.TotalOverwrittenCosts;
48              CalculateTotalTransportationCosts();
49              CalculateTotalWH();
50              CalculateTotalWorkCosts();
51          }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Przy próbie zapisu zamówienia w aplikacji okienkowej lub innych jego kosztów, koszty całkowite obliczane są od nowa. W metodzie *CalculateOverwrittenCountedCosts* brane są pod uwagę koszty obliczone automatycznie lub te nadpisane w zależności od tego, jaka opcja jest zaznaczona. Ten mechanizm lepiej opisany został w rozdziale 6. Przy próbie zapisy elementu kozstorysu lub jego materiału, również obliczane są jego koszty.

5.19 Logika biznesowa EstimateElement - obliczanie kosztów

```
39          public void CalculateTotalCosts()
40          {
41              Model.WorkCosts = Model.NumberOfWorkers * Model.Wh * Model.Rate;
42              Model.MaterialsCosts = new EstimateElementMaterialBL
                  (Database).FilterModels
                  (Model.EstimateElementMaterials.AsQueryable()).Sum(item =>
                  (decimal)item.Quantity * item.Price);
43              Model.CountedCosts = Model.TransportationCosts +
                  Model.MaterialsCosts + Model.WorkCosts;
44              new EstimateBL(Database, Model.Estimate).CalculateCosts();
45          }
46
47          public override Task<bool> SaveDatabaseAsync()
48          {
49              CalculateTotalCosts();
50              return base.SaveDatabaseAsync();
51          }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Rozdział 6. Projekt solucji PortalWWW

6.1. Wstęp

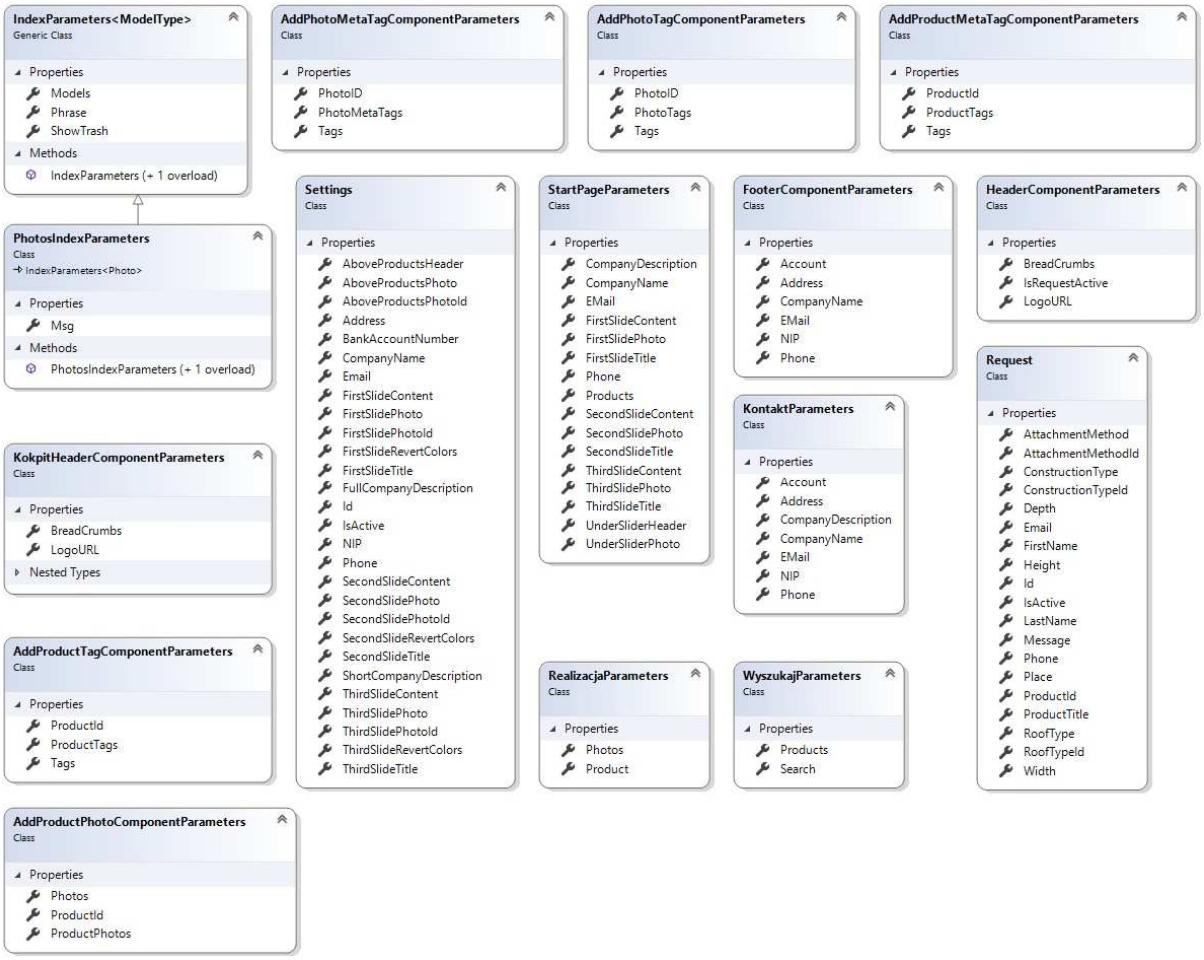
Aplikacja internetowa została podzielona na dwie części: publiczną strefę dla klienta oraz kokpit dla pracownika, do którego dostęp wymaga zalogowania. Na potrzeby oceny pracy aplikacja została umieszczona pod adresem: Strefa klienta, znajdująca się pod adresem głównym witryny zawiera stronę główną, kontakt, stronę z typami realizacji oraz ze szczegółami realizacji. Wszystkie operują na jednym kontrolerze. Oprócz tego po przejściu na stronę produktu na dole można zobacz przycisk „wyślij zapytanie". Przenosi on nas do formularza, który doda nowe zamówienie widocznie potem w aplikacji okienkowej. Aby przejść do kokpitu należy ręcznie do adresu dopisać /kokpit i zalogować się danymi podanymi przez administratora. Celowo na stronie głównej strony nie został umieszczony żaden przycisk lub inny odnośnik do kokpitu, aby ukryć to przed przeglądającymi stronę klientami. Po zalogowaniu się w nagłówku pojawi się odnośnik do kokpitu. Wykorzystano tam wiele kontrolerów, a dostęp do każdego z nich wymaga zalogowania.

W podrozdziale *Warstwa Model* pokazane zostaną klasy wykorzystane w ramach tej warstwy. *Warstwa Controller* przedstawia diagram klas kontrolerów wraz z większością kodu. *Warstwa View* zaprezentuje kod widoków strefy dla klienta oraz tych, które zarządzają Produktem od strony pracownika. Reszta widoków jest zbudowana na podobnej zasadzie. W następnych rozdziałach znajdują się przypadki użycia prezentujące aplikację od strony użytkowej wraz ze zrzutami ekranów.

6.2. Warstwa Model

Znaczą cześć warstwy Model pełnią tutaj klasy z logiki biznesowej. Poniżej przedstawiono diagram klas z Modelami stworzonymi specjalnie dla tego projektu.

6.1 Aplikacja internetowa - Modele



Źródło: opracowanie własne. Diagram wygenerowany przy użyciu Class designer Visual Studio

IndexParameters

Jest to klasa generyczna wykorzystywana na stronach z listą *Modeli*. Zawiera ich kolekcję, logiczną zmienną sygnalizującą czy pokazać kosz, która domyślnie ustawiona jest na *false* i frazę do wyszukiwania. Dziedziczy po niej *PhotosIndexParameters*, który dodatkowo zawiera wiadomość do wyświetlenia po dodaniu zdjęcia. Więcej o tym procesie znajduje się w kolejnym podrozdziale

TagComponentParameters

Modele zawierające w nazwie tą frazę wykorzystywany jest przy dodawaniu Tagów do produktów i zdjęć.

6.3. Warstwa Controller

6.2 Aplikacja internetowa - diagram kontrolerów

Źródło: opracowanie własne. Diagram wygenerowany przy użyciu Class designer Visual Studio

Podstawowym kontrolerem, po którym dziedziczą wszystkie inne, jest *BaseController*. Zawiera pole *MetaTags*, które zawiera meta tagi oddzielone przecinkami, które zostaną umieszczone w znaczniku meta. Tagi te można dodawać za pomocą metody *AddMetaTag*. *SetDefaultMetaDescription* ustawia jako meta opis strony opis firmy, który można zmienić w ustawieniach w kokpicie. Można jednak ustawić inny opis za pomocą *SetMetaDescription*.Kod tych metod można zobaczyć na poniższej grafice.

6.3 Aplikacja internetowa - BaseController

```
20          protected void AddMetaTag(Tag tag)
21          {
22              MetaTags += ", " + tag.Title;
23              ViewData["MetaTags"] = MetaTags;
24          }
25
26          protected void SetMetaDescription(string metaDescription)
27          {
28              ViewData["MetaDescription"] = metaDescription;
29          }
30
31          protected async Task SetDefaultMetaDesciptionAsync()
32          {
33              SetMetaDescription((await new TextParameterBL().GetAllModelsFromDB
                  ().FindAsync(3)).Content
34                  + " telefon: " + (await new TextParameterBL
                      ().GetAllModelsFromDB().FindAsync(4)).Content
35                  + " adres e-mail: " + (await new TextParameterBL
                      ().GetAllModelsFromDB().FindAsync(5)).Content);
36              TagBL tagBL = new TagBL();
37
38              if (MetaTags == null)
39                  MetaTags = "";
40
41              foreach (Tag tag in new TagBL().GetAllModelsFromDB().Where(item =>
                    new int[] { 1, 2, 3, 5, 8, 10, 11, 14, 15 }.Contains(item.Id)))
42              {
43                  if (!MetaTags.Contains(tag.Title))
44                      AddMetaTag(tag);
45              }
46          }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

BaseDBController jest klasą generyczną, która korzysta z logiki biznesowej. Dziedziczące po niej kontrolery przeznaczone są do wykorzystanie w kokpicie. Zawiera metody do usuwania (*Delete*) oraz przywracania (*Restore*) *Modelu* z kosza oraz wirtualną metodę Index, która wystarczy do zdecydowanej większości przypadków. Jeśli jednak jakiś kontroler wymaga niestandardowych operacji przy wyświetlaniu listy lub przypisanie dodatkowych filtrów, możliwe jest jej nadpisanie. Po wybraniu filtrów i wpisaniu frazy wyszukiwania na widoku, akcja *Index* wywoływana jest jeszcze raz. Aby filtrowanie i wyszukiwanie zadziałało prawidłowo należy uzupełnić filtry w logice biznesowej. *FillDefaultIndexParametersAsync* wypełnia filtr dotycząca aktywności oraz frazę wyszukiwania. W przypadku większej ilości filtrów nadpisuje się akcję *Index*. Metoda *Validate* wykorzystywana jest podczas walidacji; wywołuje metodę *CheckAndGetErrorsAsync* z logiki biznesowej. Kod tej klasy prezentuje się następująco.

6.4 Aplikacja internetowa - BaseDBController

```
24          public async Task<IActionResult> Restore(int id)
25          {
26              try
27              {
28                  ModelBL.Model = await ModelBL.GetAllModelsFromDB().FindAsync
                      (id);
29                  await ModelBL.ActivateModelAsync();
30                  await ModelBL.SaveDatabaseAsync();
31              }
32              catch (Exception) { }
33              return RedirectToAction(nameof(Index));
34          }
35
36          public async Task<IActionResult> Delete(int? id)
37          {
38              ModelBL.Model = await ModelBL.GetAllModelsFromDB().FindAsync(id);
39              if (ModelBL.Model == null)
40                  return NotFound();
41              await ModelBL.DisactivateModelAsync();
42              await ModelBL.SaveDatabaseAsync();
43              return RedirectToAction(nameof(Index));
44          }
45
46          public virtual async Task<IActionResult> Index
              (IndexParameters<ModelType> parameters)
47          {
48              if (parameters == null)
49                  return View(new IndexParameters<ModelType>(await
                      ModelBL.GetModelsCollectionsAsync()));
50              await FillDefaultIndexParametersAsync(parameters);
51              return View(parameters);
52          }
53
54          public async Task FillDefaultIndexParametersAsync
              (IndexParameters<ModelType> parameters)
55          {
56              ModelBL.PhraseToSearch = parameters.Phrase;
57              ModelBL.IsActiveFilter = parameters.ShowTrash == false ?
                    FilterOptions.Yes : FilterOptions.No;
58              parameters.Models = await ModelBL.GetModelsCollectionsAsync();
59          }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Dodawanie tagów

Kontroler produktów umożliwia dodanie tagów i zdjęć produktów. W następnej części rozdziału zostanie pokazany widok formularza do wyboru zdjęć i tagów. Kod prezentuje się w następujący sposób.

6.5 Aplikacja internetowa - dodawanie tagów i zdjęć produktu

```
178         public async Task<RedirectToActionResult> DeleteTagAsync(int id, int
              tagId)
179         {
180             await new ProductTagBL(ModelBL.Database, await
                  ModelBL.Database.ProductTags.FindAsync
                  (tagId)).DisactivateModelAsync();
181             await ModelBL.SaveDatabaseAsync();
182             return RedirectToAction(nameof(Details), new { id });
183         }
184
185         public async Task<RedirectToActionResult> AddMetaTagAsync(int id, int
              tagId)
186         {
187             ProductMetaTag productMetaTag = await new ProductMetaTagBL
                  (ModelBL.Database).CreateNewModelAsync();
188             productMetaTag.ProductId = id;
189             productMetaTag.TagId = tagId;
190             await ModelBL.SaveDatabaseAsync();
191             return RedirectToAction(nameof(Details), new { id });
192         }
193
194         public async Task<RedirectToActionResult> DeleteMetaTagAsync(int id,
              int tagId)
195         {
196             await new ProductMetaTagBL(ModelBL.Database, await
                  ModelBL.Database.ProductMetaTags.FindAsync
                  (tagId)).DisactivateModelAsync();
197             await ModelBL.SaveDatabaseAsync();
198             return RedirectToAction(nameof(Details), new { id });
199         }
200
201         public async Task<RedirectToActionResult> AddPhotoAsync(int id, int
              photoId)
202         {
203             ProductPhoto productPhoto = await new ProductPhotoBL
                  (ModelBL.Database).CreateNewModelAsync();
204             productPhoto.ProductId = id;
205             productPhoto.PhotoId = photoId;
206             await ModelBL.SaveDatabaseAsync();
207             return RedirectToAction(nameof(Details), new { id });
208         }
209
210         public async Task<RedirectToActionResult> DeletePhotoAsync(int id, int
               photoId)
211         {
212             await new ProductPhotoBL(ModelBL.Database, await
                  ModelBL.Database.ProductPhotos.FindAsync
                  (photoId)).DisactivateModelAsync();
213             await ModelBL.SaveDatabaseAsync();
214             return RedirectToAction(nameof(Details), new { id });
215         }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Produkty

6.6 Aplikacja internetowa – produkty w strefie klienta

```
44          public async Task<IActionResult> Realizacje(string id)
45          {
46              if (id != null)
47              {
48                  ProductBL productWebBL = new ProductBL();
49                  productWebBL.Model = await productWebBL.GetAllModelsFromDB
                      ().FirstOrDefaultAsync(item => item.Title == id);
50                  if (productWebBL.Model == null)
51                      return NotFound();
52                  List<Tag> tags = new List<Tag>(productWebBL.GetValidMetaTags
                      ().ToList().Select(item => item.Tag));
53                  PhotoBL photoWebBL = new PhotoBL(productWebBL.Database,
                      productWebBL.Model.FirstDistinctivePhoto);
54                  tags.AddRange(photoWebBL.GetValidMetaTags().Select(item =>
                      item.Tag));
55                  photoWebBL.Model = productWebBL.Model.SecondDistinctivePhoto;
56                  tags.AddRange(photoWebBL.GetValidMetaTags().Select(item =>
                      item.Tag));
57                  photoWebBL.Model = productWebBL.Model.ThirdDistinctivePhoto;
58                  tags.AddRange(photoWebBL.GetValidMetaTags().Select(item =>
                      item.Tag));
59                  photoWebBL.Model = productWebBL.Model.ParallaxPhoto;
60                  tags.AddRange(photoWebBL.GetValidMetaTags().Select(item =>
                      item.Tag));
61                  foreach (Tag item in tags)
62                  {
63                      AddMetaTag(item);
64                  }
65                  SetMetaDescription(productWebBL.Model.MetaSpecyfication);
66                  await productWebBL.ProcessHTMLColumnsAsync();
67                  ViewData["Photos"] = new ProductPhotoBL
                      (productWebBL.Database).FilterModels
                      (productWebBL.Model.ProductPhotos).Select(item =>
                      item.Photo);
68                  return View("Realizacja", productWebBL.Model);
69              }
70
71              AddMetaTag(new Tag() { Title = "Realizacje" });
72              await SetDefaultMetaDesciptionAsync();
73              ProductTypeBL productTypeWebBL = new ProductTypeBL();
74              List<ProductType> model = await
                  productTypeWebBL.GetValidModelsFromDB().ToListAsync();
75              return View(model);
76          }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Powyższa grafikę prezentuje kod akcji pokazującej produkty. Id oznacza tutaj nazwę produkty. Jeśli nie została podana (adres /realizacje) pokazywana jest lista typów produktów. Jeśli została podana i znaleziono produkt. Następnie dodawane są meta tagi produktu, zdjęć wyróżniających oraz paralaksy do znacznika meta w nagłówku przy użyciu metody *AddMetaTag*. Jako meta opis przypisywany jest meta opis produktu. Po wypełnieniu

wymaganych przez widok ViewBag'ów (Photo) zwracany jest widok o nazwie „Realizacja", nie ten domyślny.

6.4. Warstwa View

Warstwa widoku została stworzony Przy użyciu Material Design. Jest to stworzony i zaprojektowany przez Google język projektowania, który łączy klasyczne zasady projektowania z innowacyjnością i technologią. Celem Google było stworzenie systemu, który ujednolici doświadczenia użytkowników wśród wszystkich jego produktów[7]. W projekcie zostały użyte pliki css oraz javascript pobrane z ofilanej strony Materialize: https://materializecss.com/getting-started.html

Każda ze stron została zaprojektowana responsywnie. Oznacza, to że położenie elementów na różnych wielkościach ekranów różni się. Podczas gdy na dużych monitorach wiele kilka elementów umieszonych jest w jednym wierszu, na ekranie telefonu zdecydowana większość znajduje się jeden pod drugim. W pierwszej kolejności zostaną przedstawione kody należące do stron kokpitu. Można tutaj wyróżnić kilka zakładek: zdjęcia, parametry, produkty, ustawienia, kategorie produktów. Na każdą z wyjątkiem ustawień składają się cztery strony: lista, tworzenie edycja i szczegóły. Dwa ostatnie są bliźniaczo podobne do tworzenia. Te widoki zostaną przedstawione na podstawie Parametru. Wszystkie widoki w aplikacji zostały wykonane w sposób bardzo podobny do tego, co zostało pokazane poniżej.

6.7 Aplikacja internetowa - widok 1

```
1  @model IndexParameters<BusinessLogic.Models.TextParameter>
2
3  @{
4      ViewData["Title"] = "Parametry";
5      Layout = "~/Views/Shared/_KokpitLayout.cshtml";
6  }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Elementem wspólnym wszystkich widoków jest określenie używanego na nich *Modelu*. Następnie definiuje się tytuł wyświetlany w zakładce przeglądarki i używany *Layout. Layout* jest to jest to swojego rodzaju szablon. W nim sprecyzowany jest wygląd nagłówki i stopki. Dzięki temu na stronach wystarczy wypełnić jedynie ich część główną.

[7] https://materializecss.com/getting-started.html

6.8 Aplikacja internetowa - widok wyszukiwania

```
1  <div class="row">
2      @*Input for Phrase*@
3      <div class="input-field col s12">
4          <i class="material-icons-round prefix palette-text-secondary-
             color">search</i>
5          <input id="Phrase" name="Phrase" type="text" class="palette-primary-
             text-color" value="@Model.Phrase" />
6          <label for="Phrase" class="palette-text-secondary-
             color">@BusinessLogic.Display.ViewResources.Searching</label>
7      </div>
8  </div>
9  <div class="row">
10     <div class="switch col s12 m6 l6">
11         <label class="palette-text-secondary-color">
12             @BusinessLogic.Display.ViewResources.HideTrash
13             <input name="ShowTrash" type="checkbox" value="true"
                 checked="@Model.ShowTrash">
14             <span class="lever palette-secondary-color"></span>
15             @BusinessLogic.Display.ViewResources.ShowTrash
16         </label>
17     </div>
18     <div class="col s12 m6 l6">
19         <button class="btn waves-effect waves-light palette-secondary-color
             palette-secondary-text-color right" type="submit">
20             @BusinessLogic.Display.ViewResources.Filter
21             <i class="material-icons-round right">filter_alt</i>
22         </button>
23     </div>
24
25 </div>
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Każdy widok listy zawiera pole do wprowadzenia wyszukiwanej frazy, przełącznik to pokazania kosza oraz sam przycisk do filtrowania. W linii 10 możemy zobaczyć realizację responsywości. W architekturze Material Design każdy wiersz podzielony jest na 12 cześci. W przed chwilą wspomnianym wierszu określone jest, że definiowana kolumna ma zająć: na dużych ekranach 6 części, na średnich 6 części, na małych 6 części. Następna kolumna opisana jest w taki sam sposób. Skutkuje to tym, że na dużych i średnich ekranach te dwie kolumny znajdą się obok siebie, natomiast na małych pod sobą. Powyższy kod należy do widoku częściowego, który wywoływany jest na widokach list następującym kodem: `@await Html.PartialAsync("_IndexSearchAndTrashBar")`.

6.9 Aplikacja internetowa - widok listy 1

```
21      <table>
22          <thead class="text-secondary-color">
23              <tr>
24                  <th>
25                      @BusinessLogic.DisplayNames.TitleDisplayName
26                  </th>
27                  <th></th>
28              </tr>
29          </thead>
30          <tbody class="text-primary-color">
31              @foreach (var item in Model.Models.NotDeletable)
32              {
33                  <tr>
34                      <td>
35                          @item.Title
36                      </td>
37                      <td class="right-align">
38                          <a asp-action="Edit" asp-route-id="@item.Id" class="
                        tooltipped" data-position="top" data-tooltip="Edytuj"><i
                        class="material-icons-round palette-text-secondary-
                        color">edit</i></a>
39                          <a asp-action="Details" asp-route-id="@item.Id" class="
                         tooltipped" data-position="top" data-tooltip="Więcej"><i
                        class="material-icons-round palette-text-secondary-
                        color">info</i></a>
40                      </td>
41                  </tr>
42              }
43              @foreach (var item in Model.Models.Deletable)
44              {
45                  <tr>
46                      <td>
47                          @item.Title
48                      </td>
49                      <td class="right-align">
50                          <a asp-action="Edit" asp-route-id="@item.Id" class="
                        tooltipped" data-position="top" data-tooltip="Edytuj"><i
                        class="material-icons-round palette-text-secondary-
                        color">edit</i></a>
51                          <a asp-action="Details" asp-route-id="@item.Id" class="
                         tooltipped" data-position="top" data-tooltip="Więcej"><i
                        class="material-icons-round palette-text-secondary-
                        color">info</i></a>
52                          @if (Model.ShowTrash)
53                          {
54                              <a asp-action="Restore" asp-route-id="@item.Id"
                        class=" tooltipped" data-position="top" data-
                        tooltip="@BusinessLogic.Display.ViewResources.Restore"><i
                        class="material-icons-round red-text">restore_from_trash</
                        i></a>
55                          }
56                          else
57                          {
58                              <a asp-action="Delete" asp-route-id="@item.Id"
                        class=" tooltipped" data-position="top" data-
                        tooltip="@BusinessLogic.Display.ViewResources.Delete"><i
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

6.10 Źródło: opracowanie własne. Wydrukowano z Visual Studio 2

```
                    class="material-icons-round red-text">delete</i></a>
59                      }
60                  </td>
61              </tr>
62          }
63      </tbody>
64  </table>
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Na powyższych grafikach pokazano realizację listy. W linii 25 widzimy wykorzystanie plików resources z logiki biznesowej. Są one powszechnie używane w aplikacji, co ułatwia kontrolę nad tekstem wyświetlanym użytkownikowi. W pierwszej kolejności Pokazywane są *Modele*, które nie mogą zostać usunięte, a więc pozbawione są umożliwiającego to przycisku. Następnie pokazywane są te z drugiej grupy i tutaj w zależności od tego czy widoczny jest kosz, pokazywana jest ikona usunięcia lub przywrócenia z kosza.

6.11 Aplikacja internetowa - widok 2

```
70  @{await Html.RenderPartialAsync("_Scripts");}
71
72  <script>
73      $(document).ready(function () {
74          $('.tooltipped').tooltip();
75      });
76  </script>
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Wiele komponentów takich jak slajder lub tekst podręczny pokazywany po najechaniu myszą wymagają zainicjalizowania. Taki zabieg widoczny jest w liniach 72-76 na powyższej grafice. Natomiast w linii 70 pokazano wywołanie widoku częściowego.

6.12 Aplikacja internetowa - lista zdjęć

```
 99      <div class="row flex-container">
100          @foreach (BusinessLogic.Models.Photo item in
               Model.Models.NotDeletable)
101          {
102              <div class="col s6 m4 l3 flex-item">
103                  <div class="card palette-primary-color palette-primary-text-
                       color small-margin">
104                      <div class="card-image">
105                          <img src="@item.PhotoUrl"
                         alt="@BusinessLogic.Display.ViewResources.WrongURL">
106                      </div>
107                      <div class="card-content">
108                          <span class="card-title palette-primary-text-
                         color">@item.Title</span>
109                          <a asp-action="Edit" asp-route-id="@item.Id" class="
                         tooltipped" data-position="top" data-
                         tooltip="@BusinessLogic.Display.ViewResources.Edit"><i
                         class="material-icons-round palette-text-secondary-
                         color">edit</i></a>
110                          <a asp-action="Details" asp-route-id="@item.Id"
                         class=" tooltipped" data-position="top" data-
                         tooltip="@BusinessLogic.Display.ViewResources.More"><i
                         class="material-icons-round palette-text-secondary-
                         color">info</i></a>
111                      </div>
112                  </div>
113              </div>
114          }
115      </div>
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Wyjątkiem spośród list są zdjęcia. Tutaj inaczej niż w innych przypadkach nie jest widoczna tabela, ale siatka zdjęć, również responsywna. Wykorzystywany tutaj jest mechanizm flex, dzięki czemu elementy nie mieszczące się w jednej linii umieszczane są poniżej. Klasa stylów *flex-container* zawiera atrybuty: *display: flex; flex-wrap: wrap;*. Natomiast klasa flex-item: *margin-left: 0px !important; margin-right: 0px !important;*.

6.13 Aplikacja internetowa - widok pola formularza

```
22          <div class="input-field col s12 m6 l6">
23              <i class="material-icons-round prefix palette-text-secondary-
                  color">label</i>
24              <input id="@Html.DisplayNameFor(model => model.Title)" asp-
                  for="Title" class="validate palette-primary-text-color"
                  onfocusout="Validate('@Html.DisplayNameFor(model =>
                  model.Title)')" />
25              <label for="@Html.DisplayNameFor(model => model.Title)"
                  class="palette-text-secondary-
                  color">@BusinessLogic.DisplayNames.TitleDisplayName</label>
26              <span id="@Html.DisplayNameFor(model => model.Title) span"
                  class="helper-text"></span>
27          </div>
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Każda pole do wprowadzenia danych przez użytkownika zostało stworzone na podstawie instrukcji na oficjalnej stronie Materialize: https://materializecss.com/ Tak jak w przypadku listy wykorzystuje się tutaj pliki resources. W powyższy sposób tworzy się pola na stronach tworzenia oraz edycji *Modelu*, natomiast na stronie szczegółów dodaje się atrybut *readonly*, tak aby nie można było edytować wyświetlonej wartości. Ponadto zaimplementowano tutaj mechanizm walidacji. Do niezbędnych jej elementów zalicza się:

- Określenie id pola z wartością, zawsze powinna to być nazwa pola, linia 24,
- Dodanie wydarzenia *onfocusout*, które wywołuje funkcję *Validate*, które bezpośrednio waliduje pole, linia 24,
- Dodatkowa pole span o strukturze *Nazwa pola* + *span* z klasą *helper-text*, tutaj wyświetlany jest komunikat błędu, linia 26.

Aby jednak walidacja działała na każdej stronie umieszczono widok częściowy *_Validation*, którego kod prezentuje się następująco.

6.14 Aplikacja internetowa - widok walidacji 1

```
<script type="text/javascript">
    function Validate(property) {
        var element = document.getElementById(property);
        if (element == null)
            return;
        var value = document.getElementById(property).value;
        $.ajax({
            url: "@Url.Action("Validate")",
            data: { property: property, value: value, id: @Model.Id},
            dataType: "text",
            success: function (error) {
                ContinueValidation(property, value, error);
            }
        });
    }

    function ContinueValidation(property, value, error) {
        document.getElementById(property + " span").setAttribute("data-error",
          error);
        var input = document.getElementById(property)
        if (error == "") {
            input.classList.remove("invalid");
            input.classList.add("valid");
        }
        else {
            input.classList.add("invalid");
            input.classList.remove("valid");
        }
    }

    function ValidateAll() {
        var error = true;
        @foreach (System.Reflection.PropertyInfo item in Model.GetType
          ().GetProperties())
        {
            if (item.Name != "Id" && item.Name != "LazyLoader")
            {
                @:Validate("@item.Name");
            }
        }
    }

    function TrySubmit() {
        @foreach (System.Reflection.PropertyInfo item in Model.GetType
          ().GetProperties())
        {
            if (item.Name != "Id")
            {
                @:if (document.getElementById("@item.Name") !== null)
                    @:if (document.getElementById
                    ("@item.Name").classList.contains("invalid"))
                        @:return false;
            }
        }
        return true;
    }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

6.15 Aplikacja internetowa - widok walidacji 2

```
53
54      ValidateAll();
55
56      var textareas = document.getElementsByTagName('textarea');
57      var count = textareas.length;
58      for (var i = 0; i < count; i++) {
59          textareas[i].onkeydown = function (e) {
60              if (e.keyCode == 9 || e.which == 9) {
61                  e.preventDefault();
62                  var s = this.selectionStart;
63                  this.value = this.value.substring(0, this.selectionStart) +
                      "\t" + this.value.substring(this.selectionEnd);
64                  this.selectionEnd = s + 1;
65              }
66          }
67      }
68  </script>
```

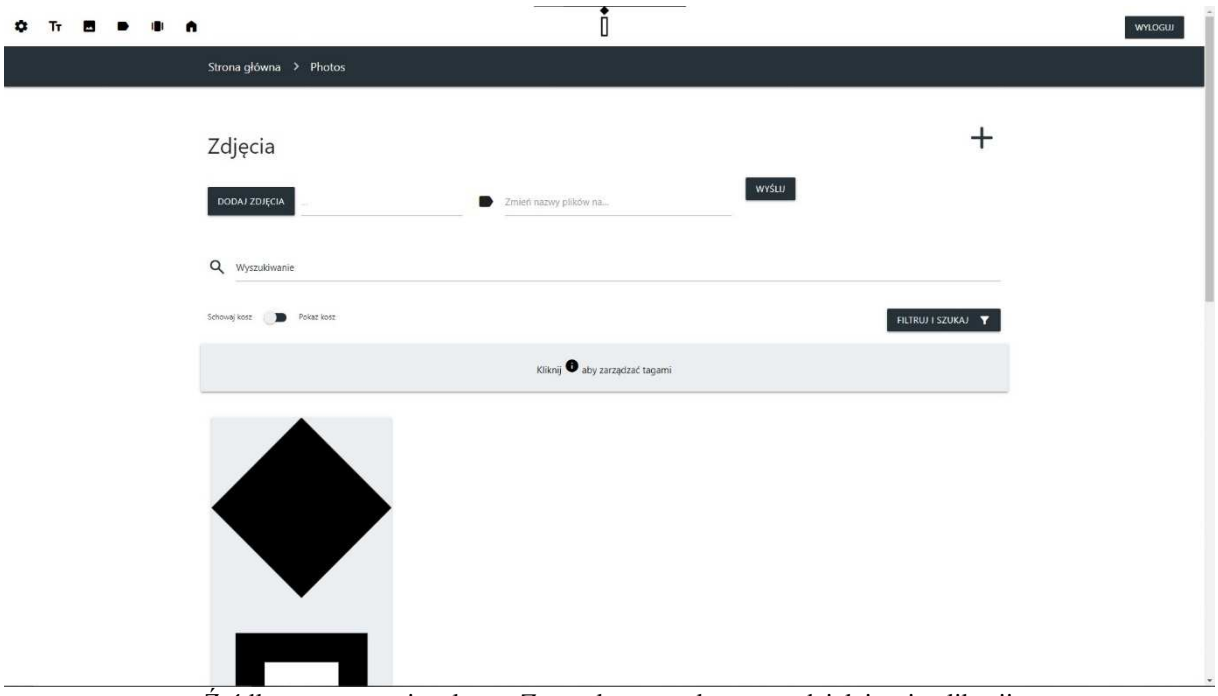
Źródło: opracowanie własne. Wydrukowano z Visual Studio

Pierwsza funkcja Validate przyjmuje jako argument nazwę pola, która ma zostać sprawdzone. Operacja jest przerywana jeśli widok nie zawiera znacznika o takim id. Następnie zapytaniem *Ajax*, które wysyła zapytanie do metody *Validate* w kontrolerze pobierana jest ewentualna zawartość błędu. Po otrzymaniu wyniku zapytania operacja kontynuowana jest metodą *ContinueValidation*. Tutaj jeśli otrzymano jakiś błąd, jest on pokazywany na widoku, a zawierający ją element otrzymuje specjalne klasy stylów, które informują, że błąd wystąpił. Naciśnięcie przycisku zapisu wywołuje metodę *TrySubmit*, która na podstawie pól *Modelu* przeszukuje widok w poszukiwaniu wcześniej pokazanych błędów. Tylko, jeśli nie odnaleziono żadnych pozwala się wykonać zapis. *ValidateAll* zawiera kod do walidacji wszystkich pól, który jest uruchamiany po pokazaniu widoku. Od linii 56 dodano kod, który umożliwia użycia tabulatora w polach textarea.

6.5. Procedura dodania nowego zdjęcia z pliku

1. Udaj się do kokpitu i kliknij ikonę zdjęcia w lewym górnym rogu.

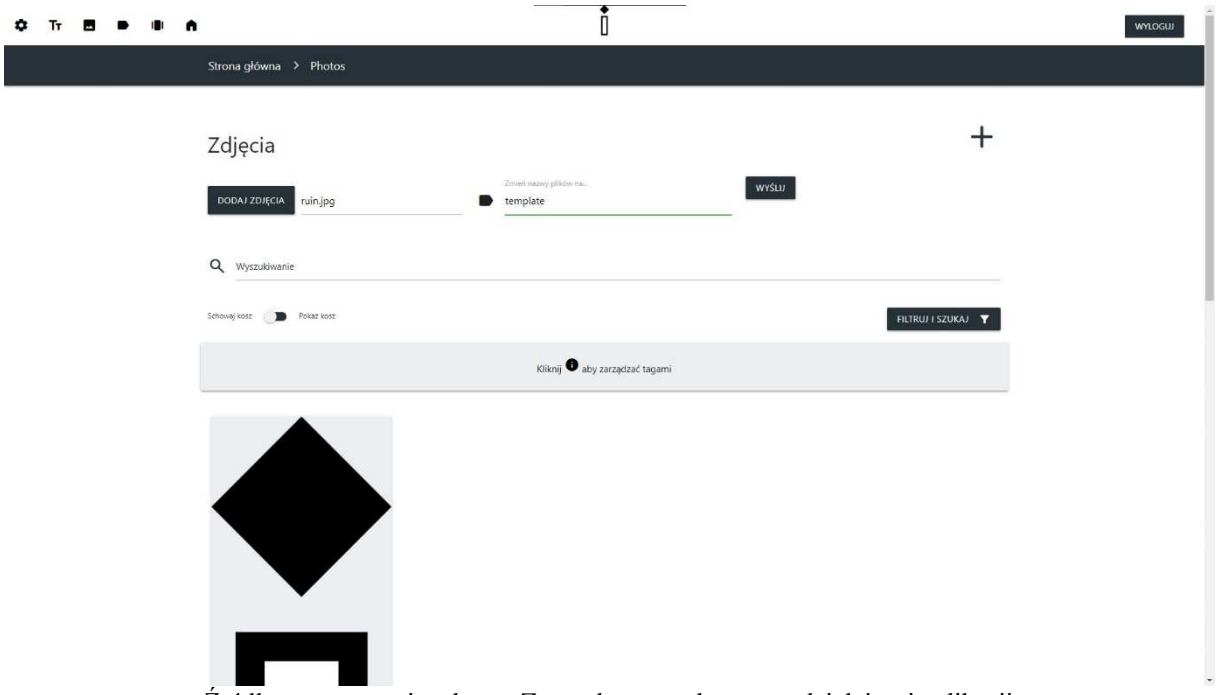
6.16 Procedura dodania nowego zdjęcia z pliku 1



Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

2. Kliknij przycisk *DODAJ ZDJĘCIA* i ewentualnie wypełnij pole *Zmień nazwy plików na…*, aby zmienić nazwy wysłanych plików.

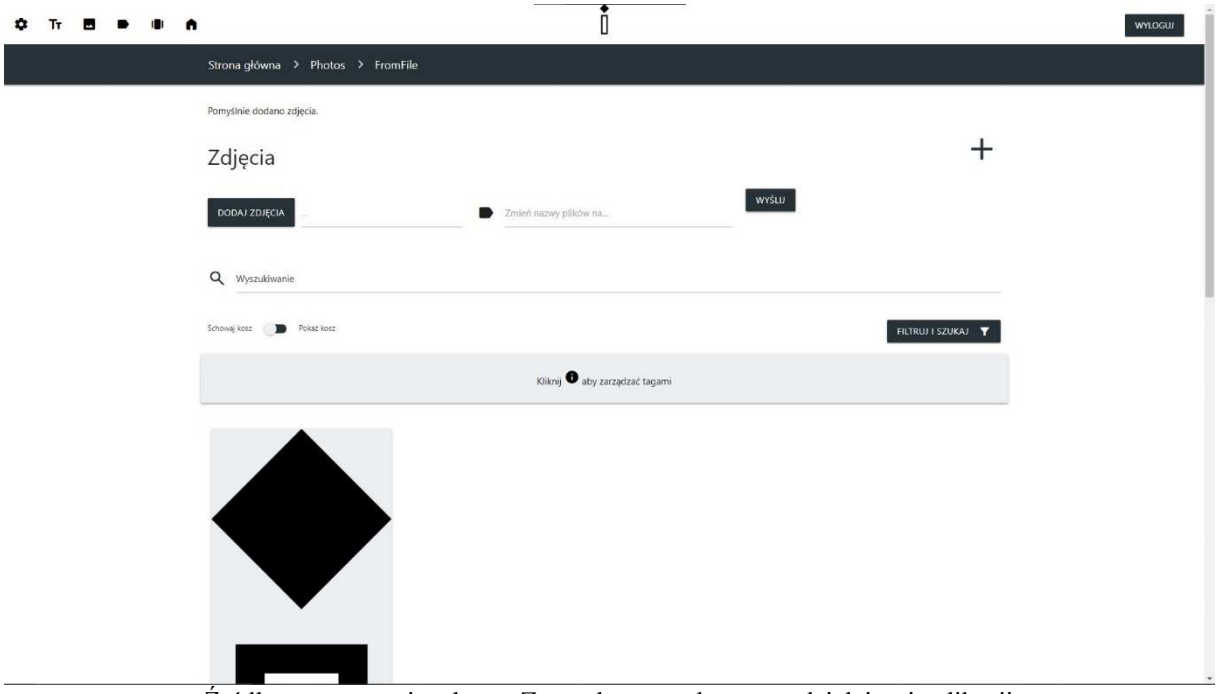
6.17 Procedura dodania nowego zdjęcia z pliku 2



Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

3. Wciśnij przycisk *WYŚLIJ* i poczekaj na dodanie zdjęcia. Na górze strony wyświetli się komunikat.

6.18 Procedura dodania nowego zdjęcia z pliku 3



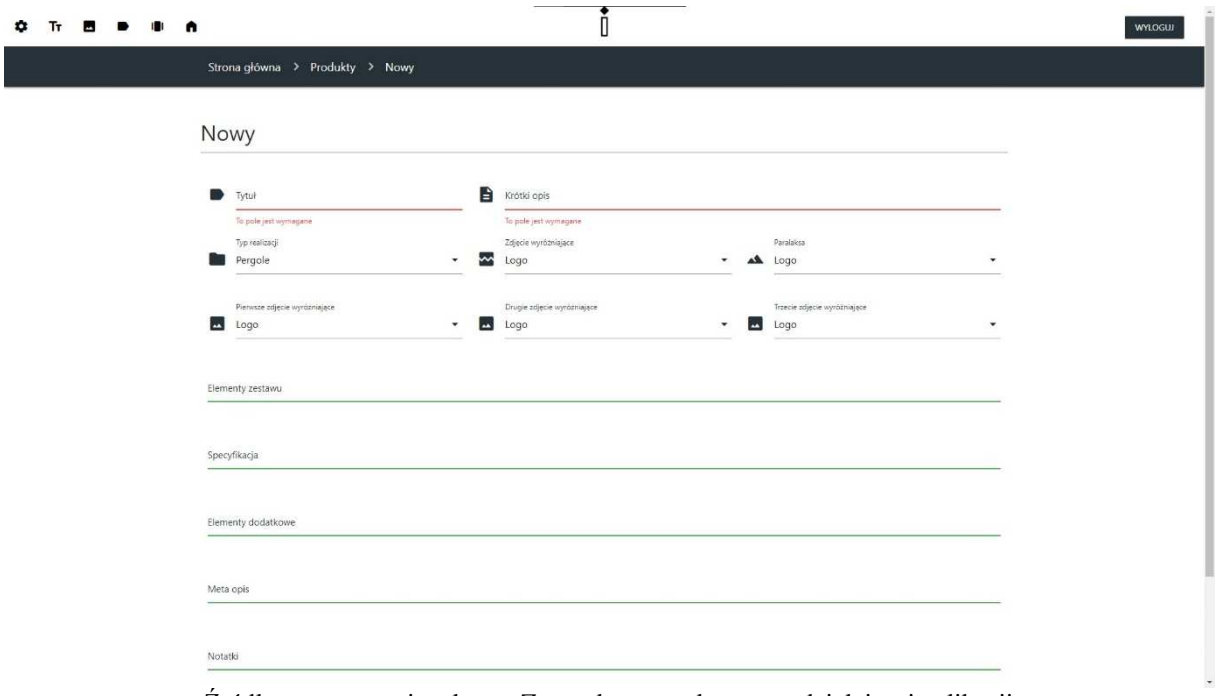
Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

6.6. Procedura dodania nowego produktu

W podobny sposób można dodać nowe zdjęcie ręcznie wpisując adres URL.

1. Przejdź do kokpitu (automatycznie otworzą się produkty) i kliknij ikonę + w górnym prawym rogu.

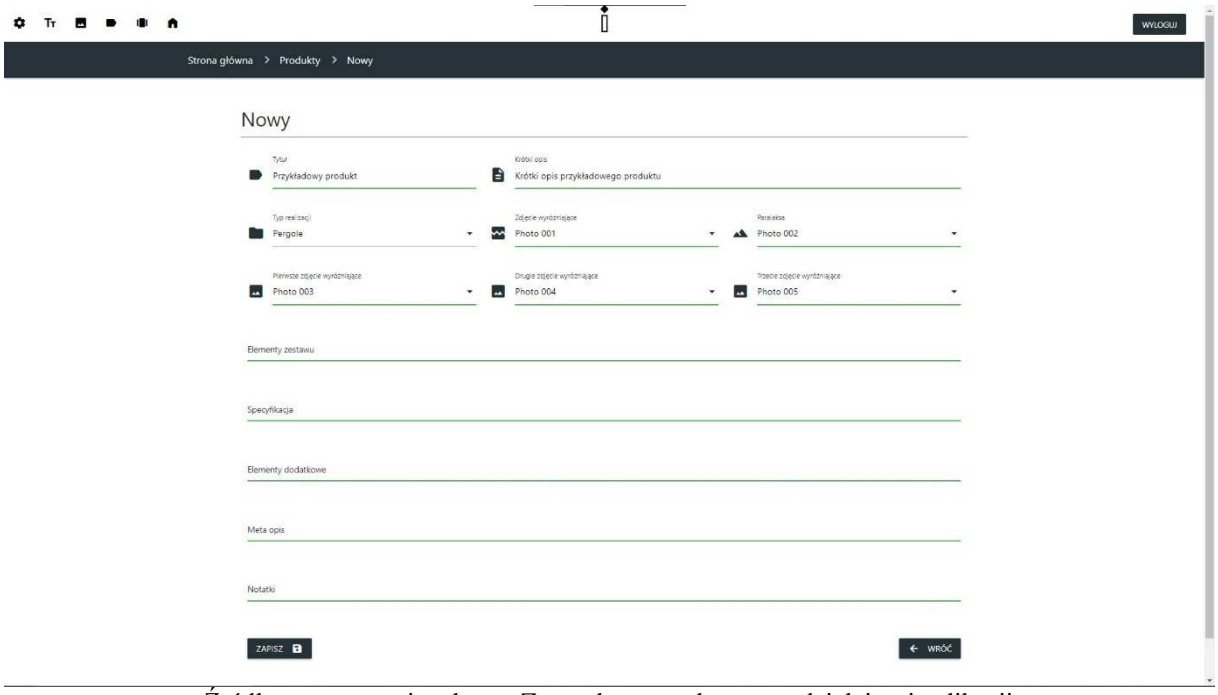
6.19 Procedura dodania nowego produktu 1



Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

2. Wypełnij przynajmniej wymagane pola i wybierz zdjęcia.

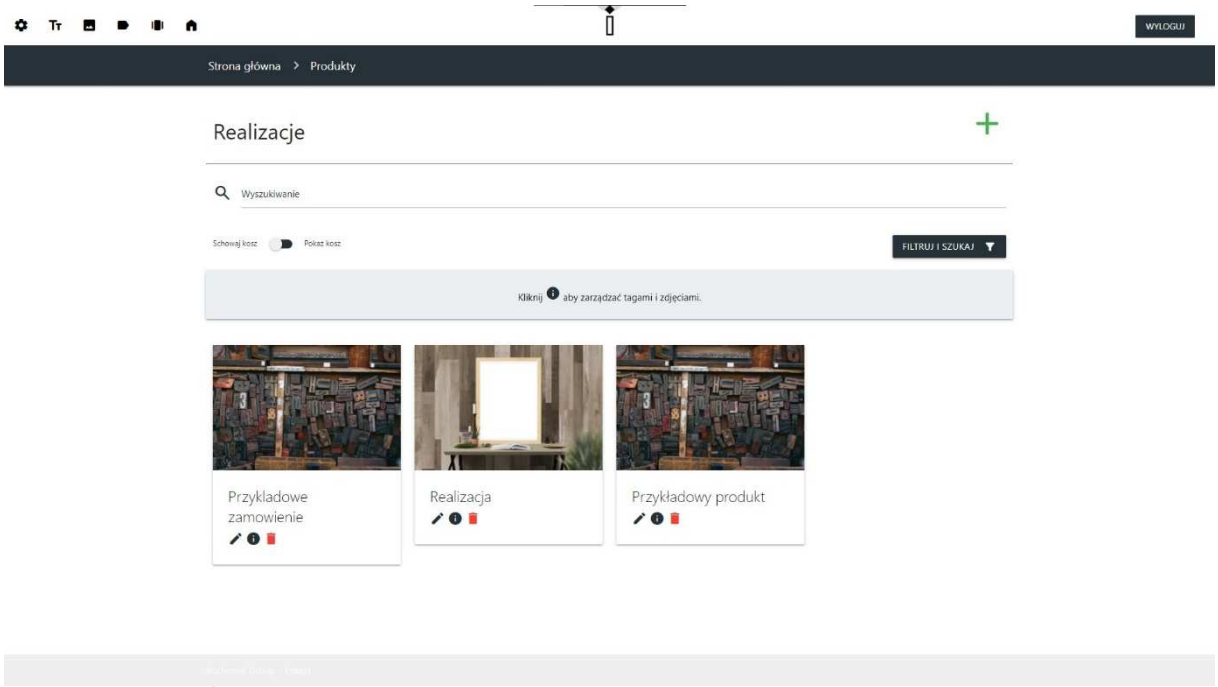
6.20 Procedura dodania nowego produktu 2



Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

3. Kliknij przycisk Zapisz widoczny na dole strony.

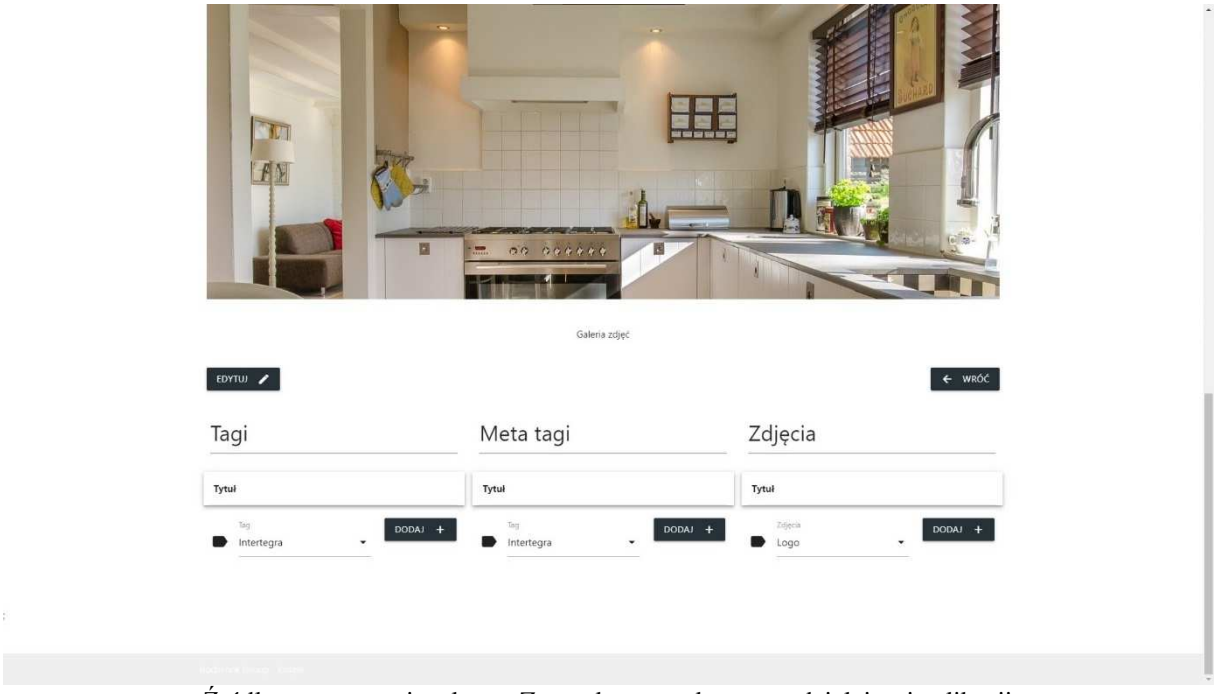
6.21 Procedura dodania nowego produktu 3



Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

4. Kliknij ikonę „i” przy nowo dodanym produkcie. Zjedź na sam dół widoku.

6.22 Procedura dodania nowego produktu 4

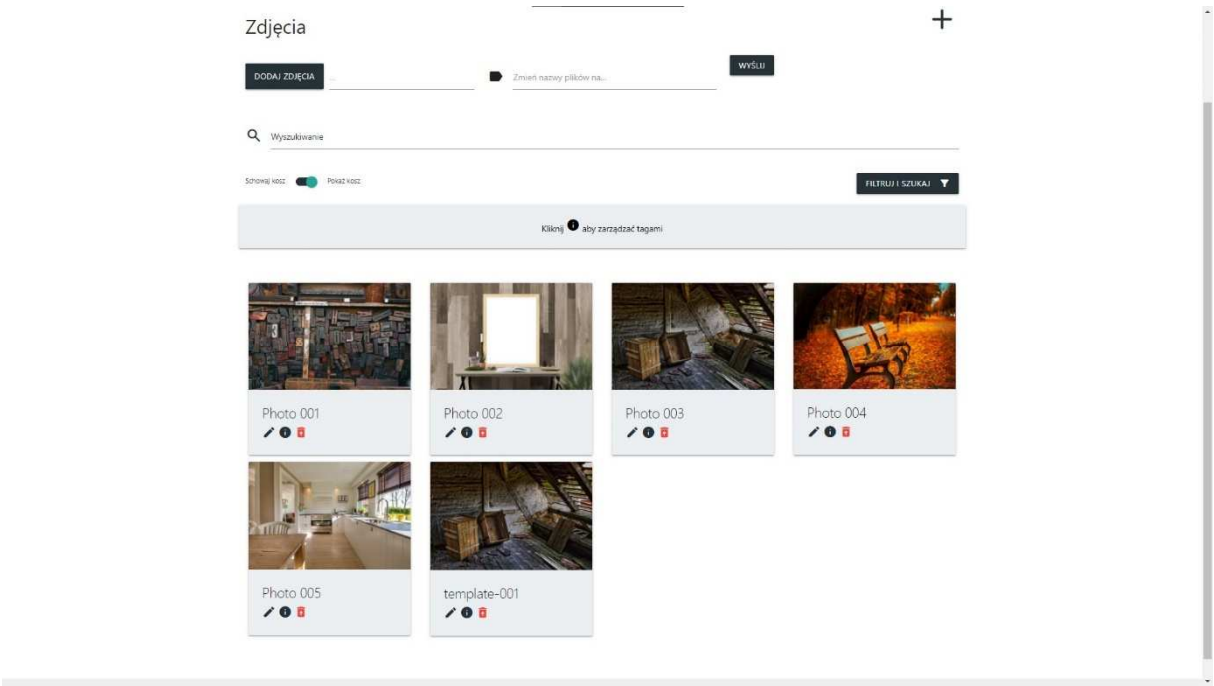


Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

5. Dodawaj tagi, meta tagi i zdjęcia produktu wybierając odpowiednie pozycje z listy i klikając odpowiednie przyciski.

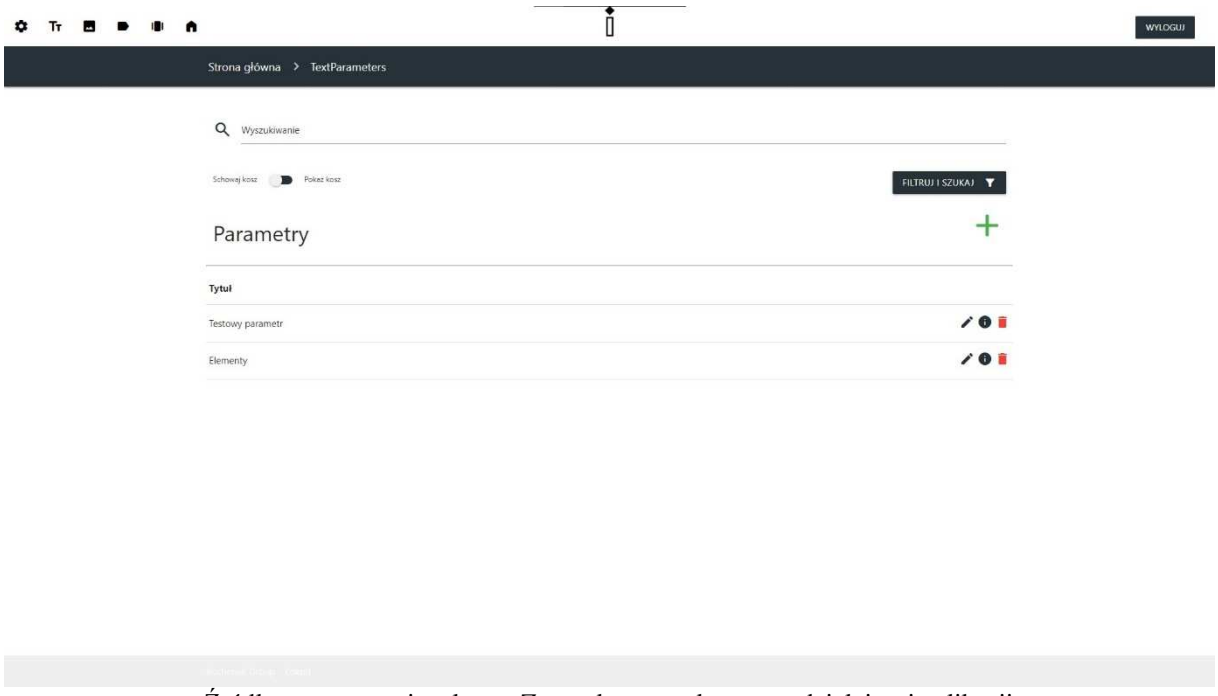
6.7. Wyróżnione widoki

6.23 Aplikacja internetowa widok - kosza ze zdjęciami



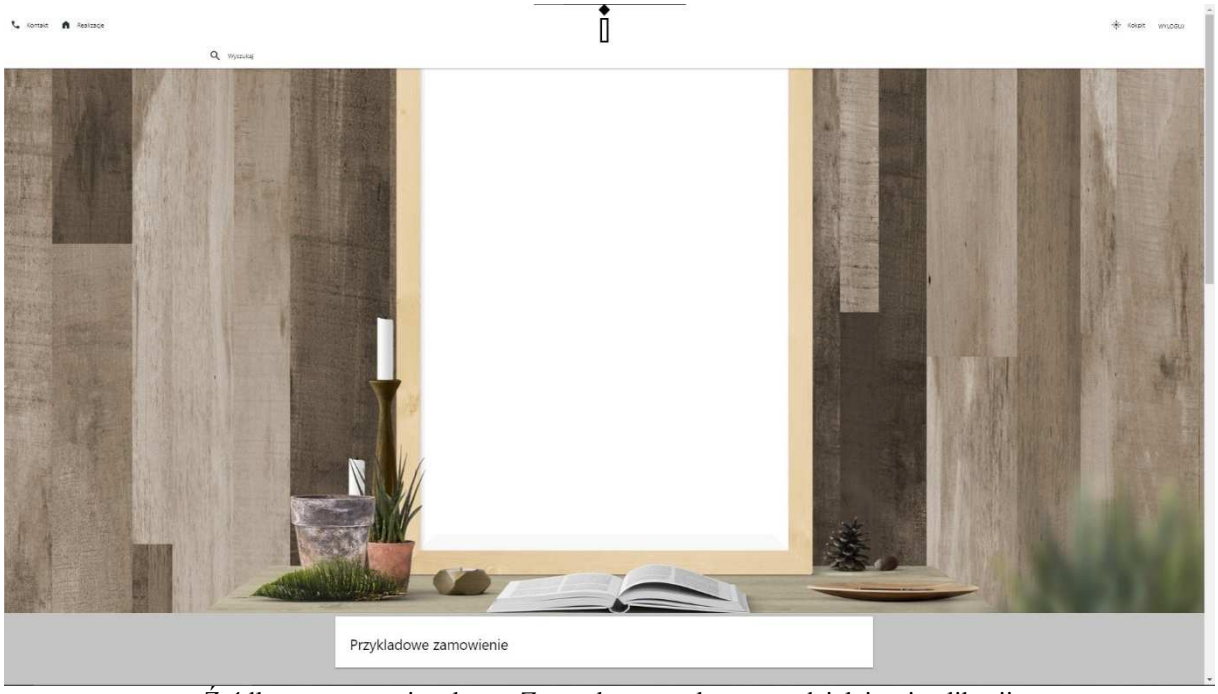
Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

6.24 Aplikacja internetowa - widok parametrów



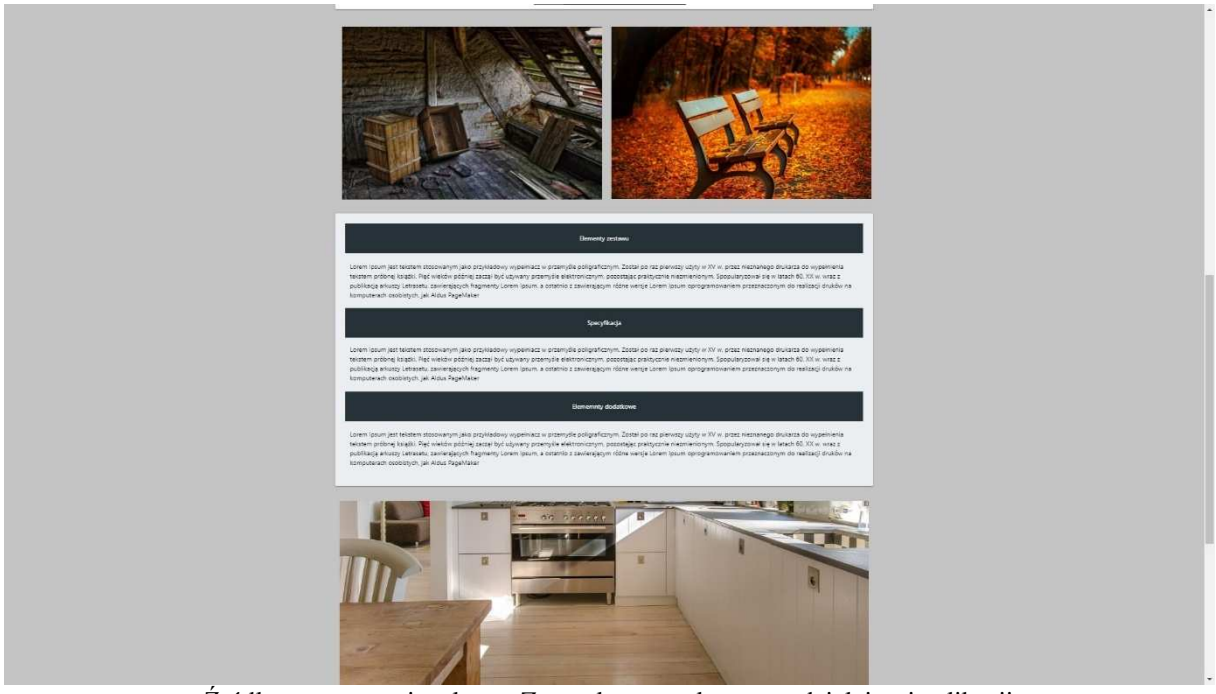
Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

6.25 Aplikacja internetowa - widok produktu 1



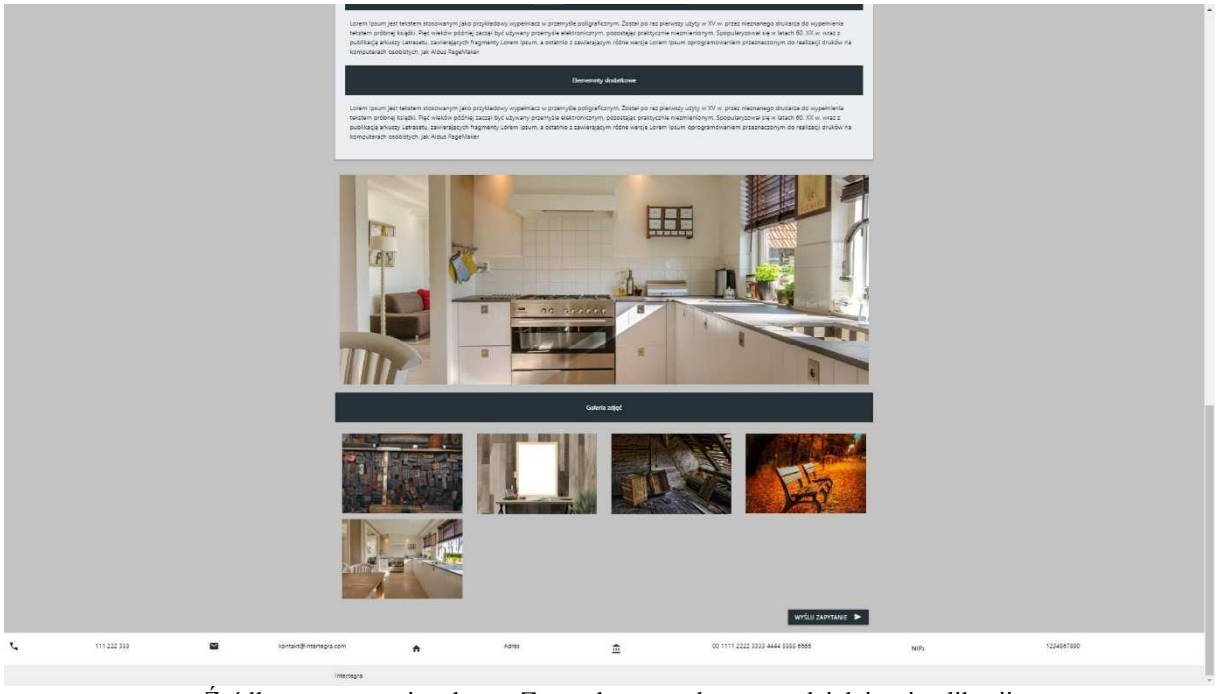
Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

6.26 Aplikacja internetowa - widok produktu 2



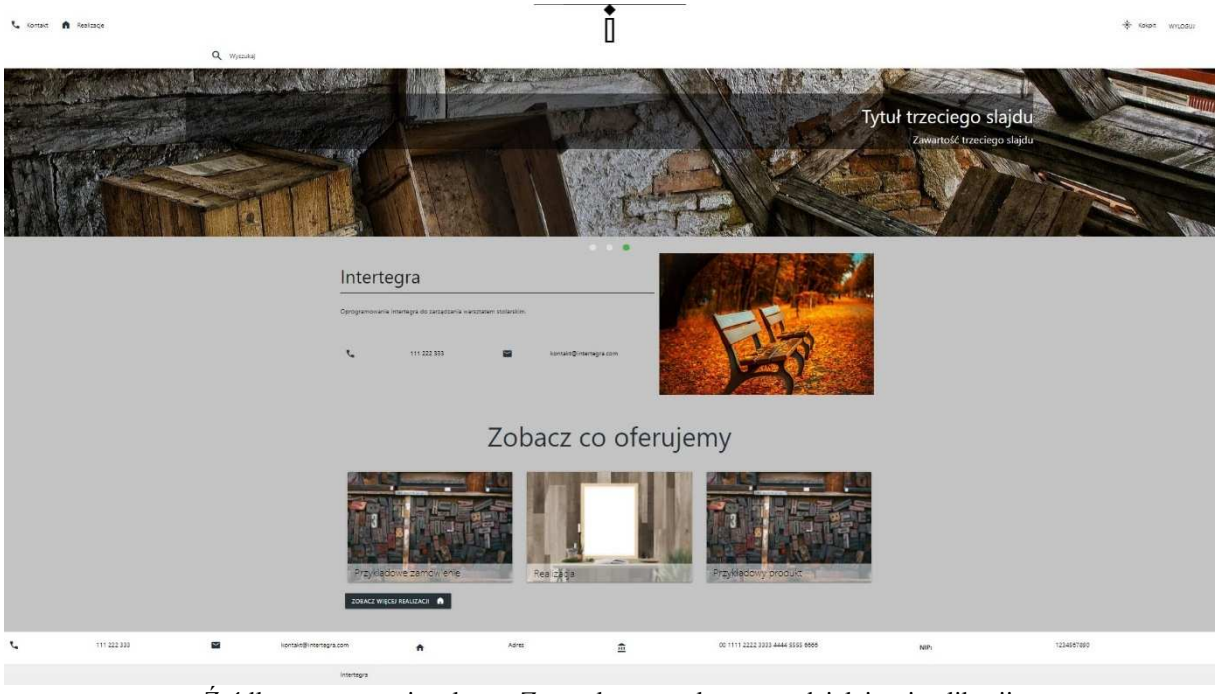
Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

6.27 Aplikacja internetowa - widok produktu 3



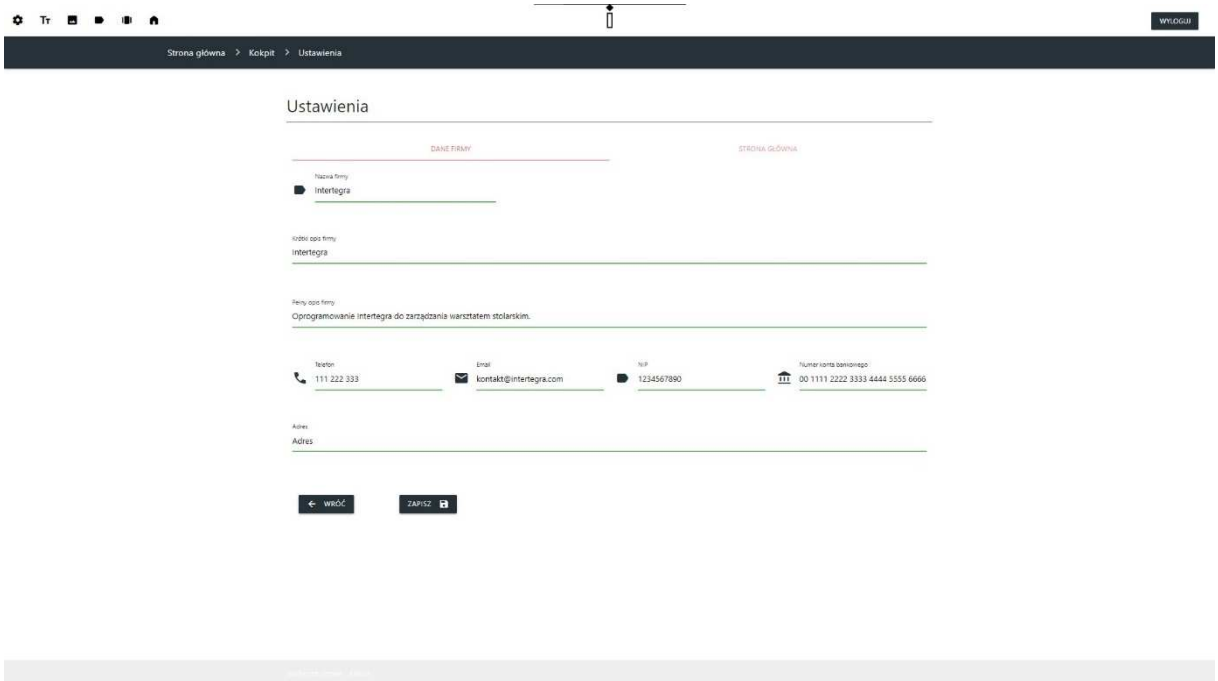
Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

6.28 Aplikacja internetowa - widok strony głównej



Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

6.29 Aplikacja internetowa - widok ustawień



Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

Rozdział 7. Projekt Solucji Intertegra

7.1. Wstęp

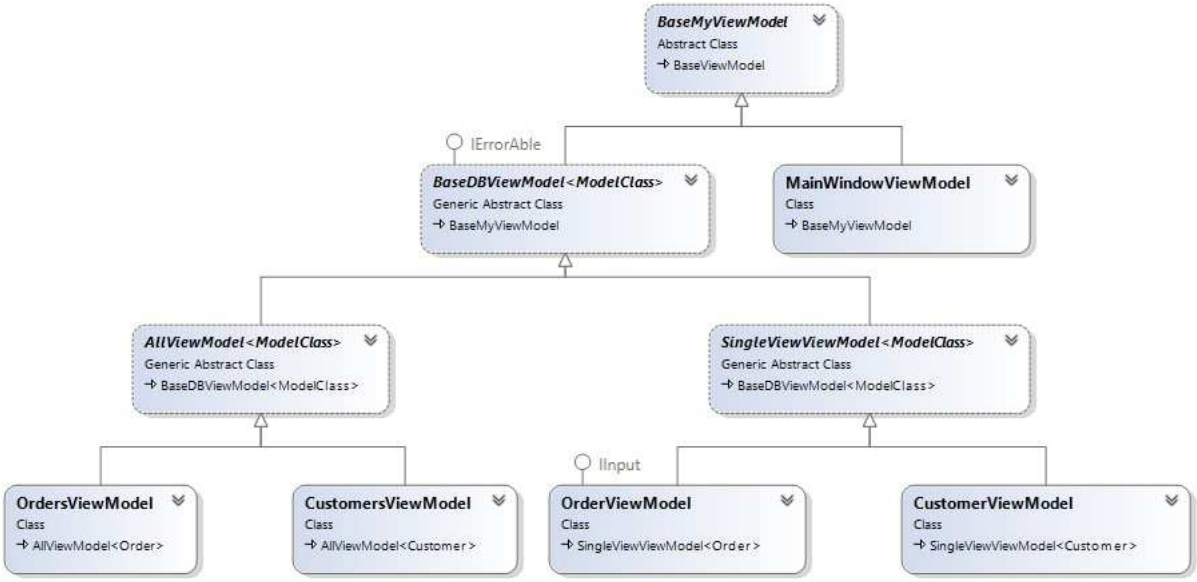
Aplikacja okienkowa przeznaczona jest do zainstalowania na komputerach firmowych. Pozbawiona jest mechanizmu logowania. Strukturę zamówienia można zobaczyć w rozdziale 4. Podobnie jak w poprzednim rozdziale pierwsze trzy podrozdziały będę prezentowały kod aplikacji, natomiast następne pokażą ją od strony użytkowej pokazując zrzuty ekranu.

7.2. Warstwa Model

Całą rolę tej warstwy przejął projekt BusinessLogic. Został on omówiony w rozdziale 5.

7.3. Warstwa ViewModel

7.1 Aplikacja okienkowa - diagram ViewModeli

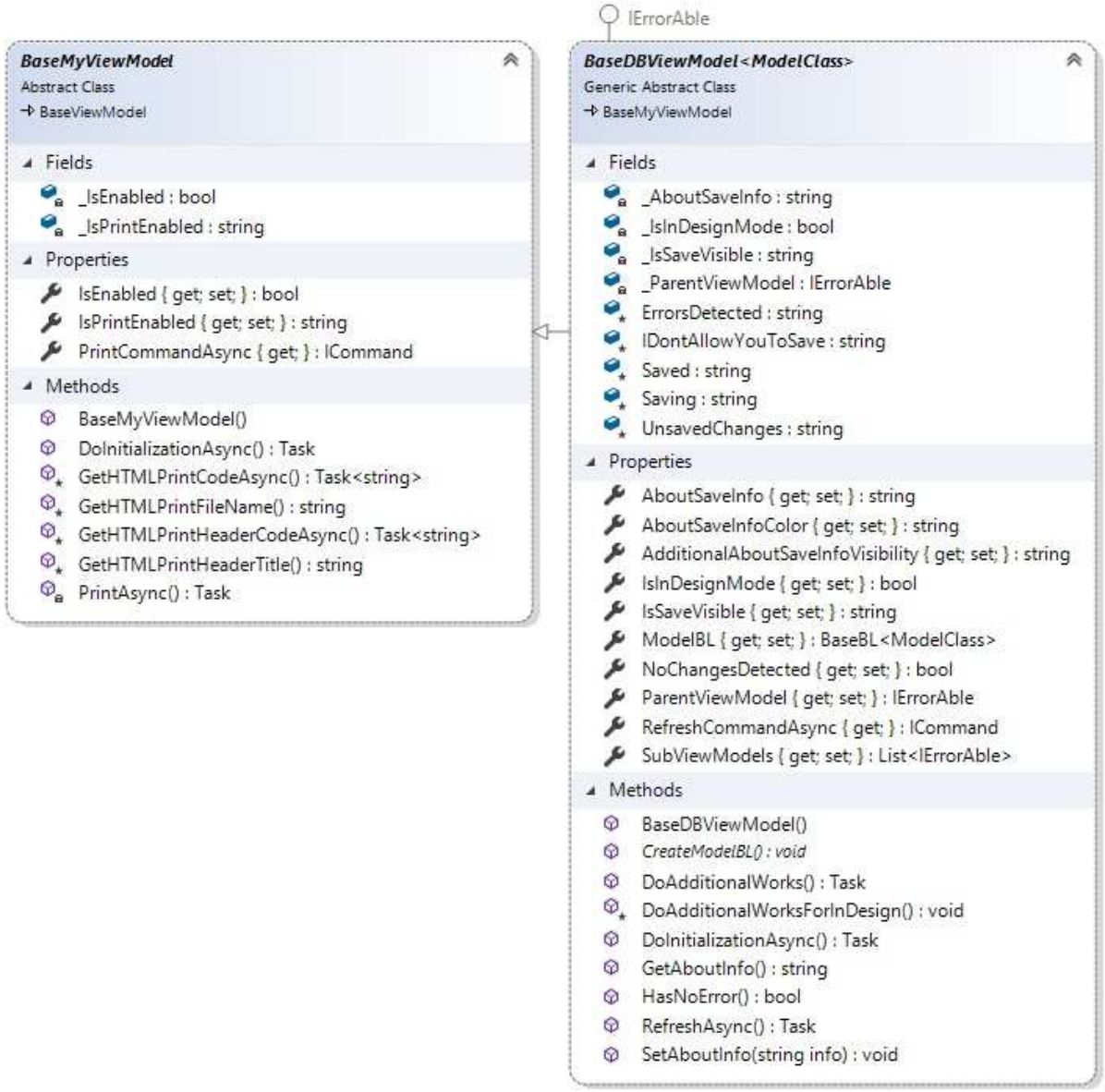


Źródło: opracowanie własne. Diagram wygenerowany przy użyciu Class designer Visual Studio

Klasy ViewModel można podzielić na dwie grupy, te operujące na pojedynczym *Modelu* lub na ich kolekcji. Na powyższym pokazano tylko po dwa reprezentatywne przykłady, dla każdego Modelu, z wyjątkiem tych użytych w aplikacji internetowej, istnieją takie klasy.

Klasy bazowe

7.2 Aplikacja okienkowa – bazowe klasy ViewModel



Źródło: opracowanie własne. Diagram wygenerowany przy użyciu Class designer Visual Studio

Klasa *BaseMyViewModel* zawiera pole *IsEnabled*, które informuje czy interfejs jest włączony, czyli czy użytkownik może wykonywać na nim interakcje. Podczas ładowania musi on być wyłączony. Metoda *DoInitializationAsync* przeznaczona jest do wywołania, już po tym jak widok zostanie stworzony i wyświetlony użytkownikowi. Zostało tutaj również zainicjalizowane drukowanie. Pole *SubViewModels* w klasie BaseDBViewModel zawiera kolekcję podległych ViewModeli. Na przykład ViewModel Zamówienia będzie tutaj posiadał między innymi ViewModel Kosztorysu. *DoAdditionalWorks* jest metodą wirtualną, która przeznaczona jest do nadpisania w finalnej klasie ViewModel i jest wywoływana na samym końcu procesu inicjalizacji. Natomiast *DoAdditionalWorksForInDesign* jest metodą pomocnicza wykorzystywaną przy tworzeniu aplikacji, nieistotną w wersji produkcyjnej.

7.3 Aplikacja okienkowa - klasa BaseMyViewModel drukowanie 1

```
 99          private async Task PrintAsync()
100          {
101              IsPrintEnabled = false.ToString();
102              string printCode = "<html>\n<head>\n"
103                                  + "<link rel=\"stylesheet\" href=\"https://
                         cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/css/
                         materialize.min.css\">\n"
104                                  + "<meta name=\"viewport\" content=
                         \"width=device-width, initial-scale= 1.0\" />\n</head>
                         \n<body>\n"
105                                  + "<div style=\"margin: 64px\">\n\n"
106                                  + await GetHTMLPrintHeaderCodeAsync()
107                                  + "\n\n"
108                                  + await GetHTMLPrintCodeAsync()
109                                  + "\n\n</div></body></html>";
110              Debug.WriteLine("Drukowanie:\n" + printCode);
111
112              HtmlToPdf htmlToPdf = new HtmlToPdf();
113              PdfDocument pdfDocument = htmlToPdf.ConvertHtmlString(printCode);
114              string filePath = @"drukowanie\" + GetHTMLPrintFileName() +
                   ".pdf";
115              pdfDocument.Save(filePath);
116
117
118              IsPrintEnabled = true.ToString();
119
120              Process.Start("explorer", AppDomain.CurrentDomain.BaseDirectory +
                   filePath);
121          }
122
123          protected virtual async Task<string> GetHTMLPrintCodeAsync() => await
               Task.FromResult("");
124
125          protected virtual async Task<string> GetHTMLPrintHeaderCodeAsync() =>
               await Task.FromResult("<div style='margin: 64px;'>\n"
126                                  + "<div class='row'>\n"
127                                  + "<div class='col s10'>\n"
128                                  + "<div class='row grey-text bold'>"
129                                  + (await new TextParameterBL().GetAllModelsFromDB
    ().FindAsync(1)).Content
130                                  + "</div>\n"
131                                  + "<div class='row'>"
132                                  + "<h3>"
133                                  + GetHTMLPrintHeaderTitle()
134                                  + "</h3>"
135                                  + "</div>\n"
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

7.4 Aplikacja okienkowa - klasa BaseMyViewModel drukowanie 2

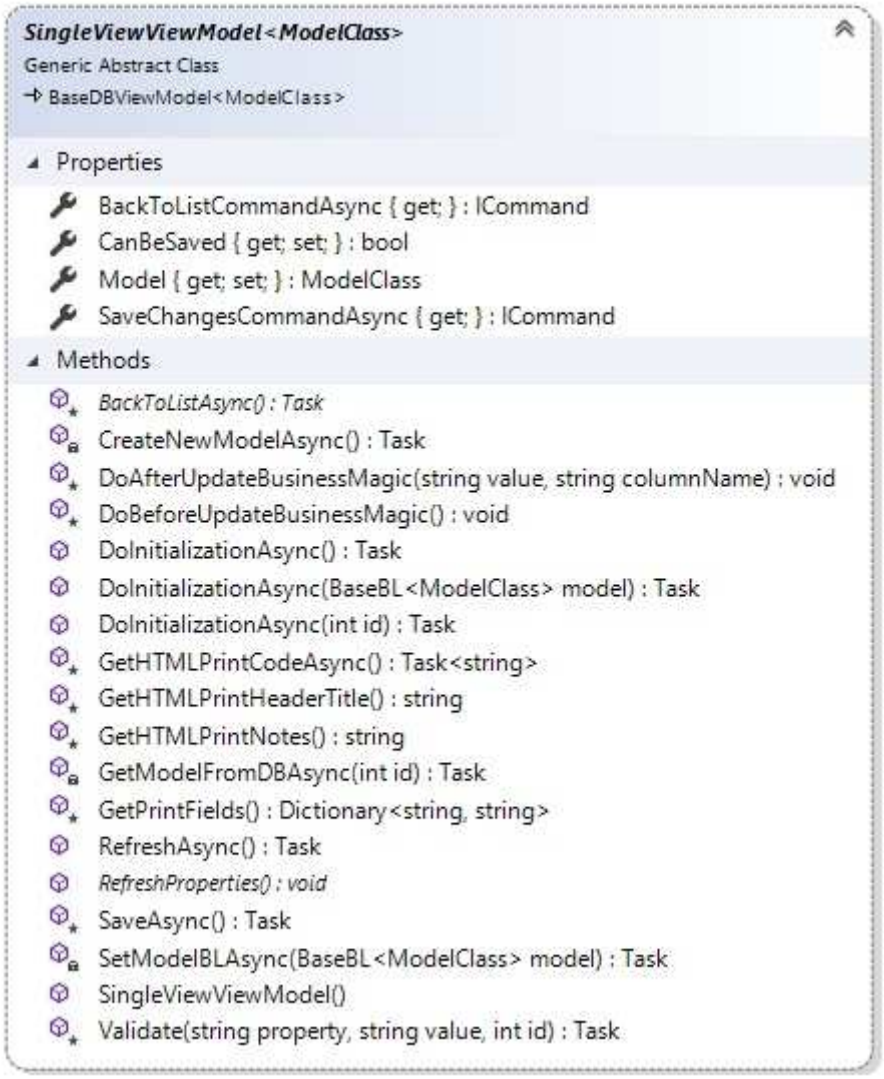
```
136                                                                    
                        + "</div>\n"
137                                                                    
                        + "<div class='col s2 right'><img height='128px'
    src='https://intertegra.bochenekgroup.pl/images/logo.svg'></div>\n"
138                                                                    
                        + "</div>\n"
139                                                                    
                        + "</div>\n"
140                                                                    
                        + "<hr />");
141
142         protected virtual string GetHTMLPrintHeaderTitle() => DisplayName;
143
144         protected virtual string GetHTMLPrintFileName() => DisplayName;
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Do drukowania zostało wykorzystane rozszerzenie *Select.HtmlToPdf.NetCore*[8]. W powyższych funkcjach tworzony jest kod HTML strony, która zostaniew wydrukowana. Podobnie jak w aplikacji internetowej zostało tutaj wykorzystane Material Design. Metoda *GetHTMLPrintCodeAsync* zwraca ciało dokumentu. Tutaj jest to pusty tekst, lepiej zostanie to wypełnione w klasach dziedziczących.

[8] Rozszerzenie autorstwa SelectPdf pobrane z NuGet Manager w Visual Studio. Strona rozszerzenia: https://www.nuget.org/packages/Select.HtmlToPdf.NetCore/20.2.0?_src=template

7.5 Aplikacja okienkowa - SingleViewModel



Źródło: opracowanie własne. Diagram wygenerowany przy użyciu Class designer Visual Studio

Klasa *SingleViewViewModel* jest klasą bazową dla ViewModeli operujących na jednym Modelu. Klasa abstrakcyjna BackToListAsync ma za zadanie powrócić do listy wszystkich Modeli, ale czasami może to być powrót do klasy dziedziącej po *SingleViewModel*. Na przykład wywołanie tej metody przez użytkownika w widoku elementów kosztorysu, spowoduje powrót do zamówienia, z racji tego, że kosztorys jest częścią widoku zamówienia. ViewModel dziedziczący po tej klasie może zostać zainicjalizowany na trzy sposoby. *CreateNewModelAsync* spowoduje stworzenie nowego Modelu przy użyciu metody o podobnej nazwie z logiki biznesowej. Metoda *GetModelFromDBAsync* pobierze istniejący z bazy danych według podanego Id. Natomiast *SetModelBLAsync* przypisze całą logikę biznesową, a więc także Model, do którego dostęp odbywa się właśnie przez logikę biznesową. Wszystkie te metody wywoływane są odpowiednimi metodami *DiInitializationAsync()*. Kod tych metod prezentuje się następująco.

7.6 Aplikacja okienkowa - klasa SingleViewModel inicjalizacja

```
78          private async Task CreateNewModelAsync()
79          {
80              await ModelBL.CreateNewModelAsync();
81              AboutSaveInfo = UnsavedChanges;
82              if (ParentViewModel != null)
83                  ParentViewModel.SetAboutInfo(UnsavedChanges);
84              foreach (var item in GetType().GetProperties().Where(item =>
                  item.Name.Contains("ErrorInfo")))
85              {
86                  PropertyInfo propertyInfo = GetType().GetProperty
                      (item.Name.Replace("ErrorInfo", ""));
87                  if (propertyInfo != null)
88                      await Validate(item.Name.Replace("ErrorInfo", ""),
                          propertyInfo.GetValue(this)?.ToString(),
                          ModelBL.GetModelID());
89              }
90              RefreshProperties();
91              await DoAdditionalWorks();
92              IsEnabled = true;
93          }
94
95          private async Task GetModelFromDBAsync(int id)
96          {
97              ModelBL.Model = await ModelBL.GetAllModelsFromDB().FindAsync(id);
98              foreach (var item in GetType().GetProperties().Where(item =>
                  item.Name.Contains("ErrorInfo")))
99              {
100                 PropertyInfo propertyInfo = GetType().GetProperty
                      (item.Name.Replace("ErrorInfo", ""));
101                 if (propertyInfo != null)
102                     await Validate(item.Name.Replace("ErrorInfo", ""),
                          propertyInfo.GetValue(this)?.ToString(),
                          ModelBL.GetModelID());
103             }
104             RefreshProperties();
105             await DoAdditionalWorks();
106             IsEnabled = true;
107         }
108
109         private async Task SetModelBLAsync(BaseBL<ModelClass> model)
110         {
111             ModelBL = model;
112             foreach (var item in GetType().GetProperties().Where(item =>
                  item.Name.Contains("ErrorInfo")))
113             {
114                 PropertyInfo propertyInfo = GetType().GetProperty
                      (item.Name.Replace("ErrorInfo", ""));
115                 if (propertyInfo != null)
116                     await Validate(item.Name.Replace("ErrorInfo", ""),
                          propertyInfo.GetValue(this)?.ToString(),
                          ModelBL.GetModelID());
117             }
118             RefreshProperties();
119             await DoAdditionalWorks();
120             IsEnabled = true;
121         }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Uaktualniona tutaj została wspominana przy okazji klasy *BasyMyViewModel* metoda *GetHTMLPrintCodeAsync*.

7.7 Aplikacja okienkowa - klasa SingleViewModel drukowanie

```
200         protected override Task<string> GetHTMLPrintCodeAsync()
201         {
202             string code = "<div class='row' style='display: flex; flex-wrap:
                wrap; '>\n";
203             foreach (var item in GetPrintFields())
204             {
205                 code += "\t<div class='col s3' style='margin: 0px 0px 24px
                    0px;'>\n" +
206                     "\t\t<div class='grey-text' style='margin: 3px'><b>" +
                        item.Key.ToUpper() + "</b></div>\n" +
207                     "\t\t<div>" + (item.Value ?? ViewResources.None) + "</div>
                        \n" +
208                     "\t</div>\n";
209             }
210             code += "</div>\n" +
211                 "<div class='row'><div class='col s12'>" + GetHTMLPrintNotes()
                      + "</div></div>";
212             return Task.FromResult(code);
213         }
214
215         protected virtual string GetHTMLPrintNotes() => (string)typeof
              (ModelClass).GetProperty("Notes").GetValue(Model);
216
217         protected virtual Dictionary<string, string> GetPrintFields() => new
              Dictionary<string, string>();
218
219         protected override string GetHTMLPrintHeaderTitle()
220         {
221             PropertyInfo propertyInfo = typeof(ModelClass).GetProperty
                  ("Title");
222             if (propertyInfo == null)
223                 return base.GetHTMLPrintHeaderTitle();
224             else
225                 return base.GetHTMLPrintHeaderTitle() + " " +
                      propertyInfo.GetValue(Model);
226         }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Wykorzystuje się tutaj abstrakcyjną metodę *GetPrintFields*, która zwraca słownik (ang. dictionary) zawierający nazwę i wartość pola. Nie jest to metoda abstrakcyjna, ponieważ nie wszystkie ViewModele będą mogły być drukowane. Jest więc ona wirtualna i zaimplementowana „na pusto". Zwrócona kolekcja może wyglądać tak: Imię i Adam, Nazwisko i Kowalski, Miejscowość i Warszawa itd. Następnie generowany jest kod, który umieszcza te pola w trzy elementowych wierszach. Dodatkowo na dole wyświetlane są notatki. Tutaj ponownie wykorzystany jest fakt, że każda tabela bazy danych musi zawierać kolumnę *Notes*. Uaktualniony został tutaj także tytuł nagłówka. Jeśli w Modelu odnaleziono pole o

nazwie *Title*, czyli jego tytuł, jest on tam dopisywany. Przykładowa metoda zwracająca pola do drukowania wygląda następująco.

7.8 Aplikacja okienkowa - AnnotationViewModel drukowanie

```
229         protected override Dictionary<string, string> GetPrintFields() => new
              Dictionary<string, string>
230         {
231             { DisplayNames.WakeTimeDisplayName, Model.WakeTime.ToString() },
232             { DisplayNames.DeadlineDisplayName, Model.Deadline.ToString() },
233             { DisplayNames.IsDoneDisplayName, Model.IsDone ?
                ViewResources.Yes : ViewResources.No }
234         };
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

W tej klasie została zrealizowana walidacja danych, która korzysta z logiki biznesowej. Każde pole na widoku po edycji jest sprawdzane. Poniżej pokazano przykładowe pole z klasy *OrderViewModel* oraz metody walidujące z klasy *SingleViewViewModel*.

7.9 Aplikacja okienkowa - klasa OrderViewModel przykładowe pole

```
674         public decimal Deposit
675         {
676             get
677             {
678                 return (Model?.Deposit).GetValueOrDefault();
679             }
680             set
681             {
682                 if (value != Model.Deposit)
683                 {
684                     DoBeforeUpdateBusinessMagic();
685                     Model.Deposit = value;
686                     DoAfterUpdateBusinessMagic(value.ToString(), nameof
                        (Deposit));
687                     OnPropertyChanged(() => Deposit);
688                 }
689             }
690         }
691         private string _DepositErrorInfo;
692
693         public string DepositErrorInfo
694         {
695             get
696             {
697                 return _DepositErrorInfo;
698             }
699             set
700             {
701                 if (value != _DepositErrorInfo)
702                 {
703                     _DepositErrorInfo = value;
704                     OnPropertyChanged(() => DepositErrorInfo);
705                 }
706             }
707         }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

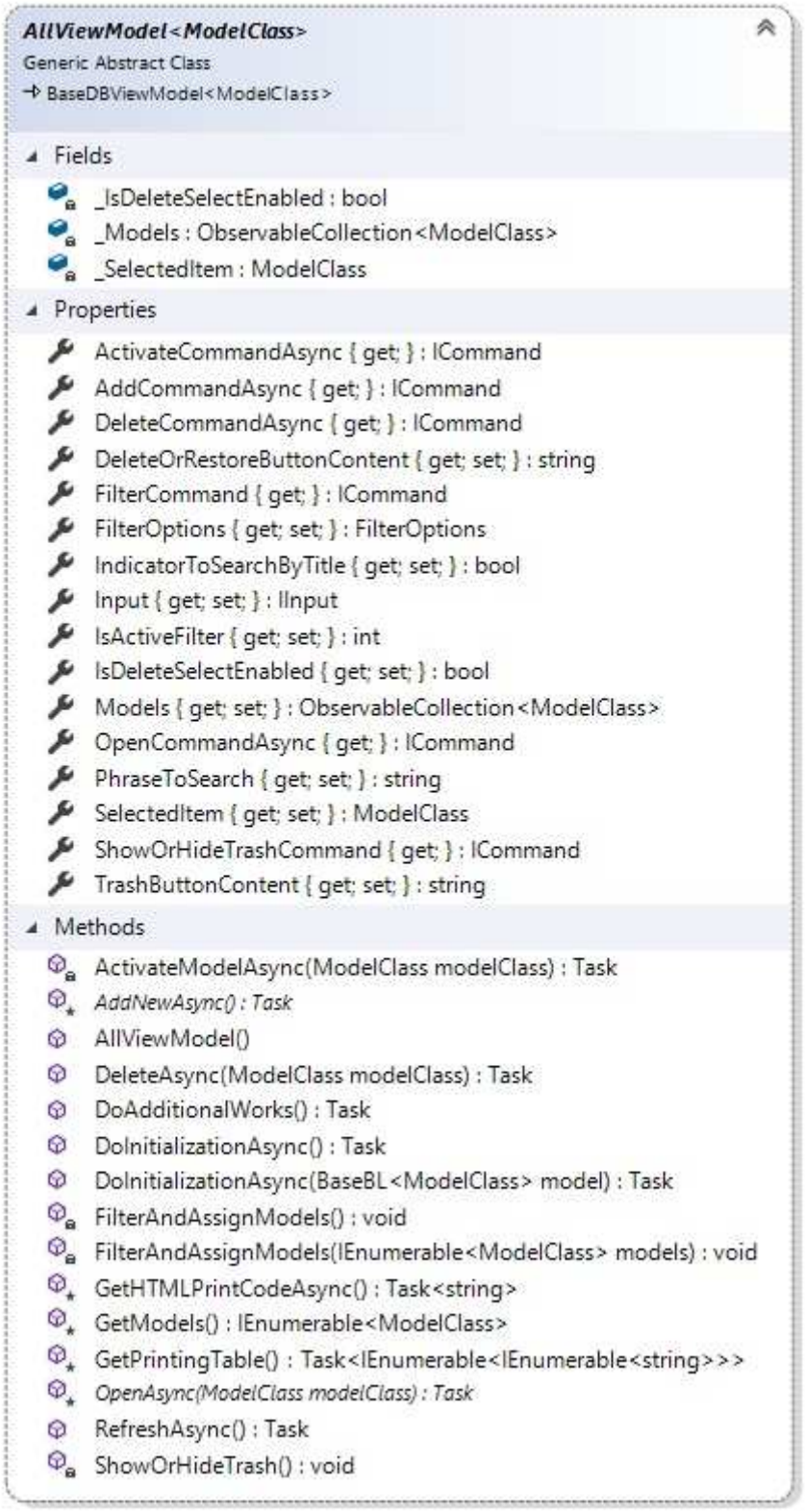
7.10 Aplikacja okienkowa - klasa SingleViewModel walidacja

```
237         protected virtual async Task Validate(string property, string value,
              int id)
238         {
239             PropertyInfo propertyInfo = GetType().GetProperty(property +
                "ErrorInfo");
240             if (propertyInfo != null)
241             {
242                 propertyInfo.SetValue(this, "Sprawdzam");
243                 OnPropertyChanged(() => CanBeSaved);
244                 if (value == null)
245                     value = "";
246                 value = await ModelBL.CheckAndGetErrorAsync(property, value,
                    id);
247                 propertyInfo.SetValue(this, value);
248             }
249             OnPropertyChanged(() => CanBeSaved);
250         }
251
252         protected void DoBeforeUpdateBusinessMagic()        {        }
253
254         protected void DoAfterUpdateBusinessMagic(string value, string
              columnName)
255         {
256             AboutSaveInfo = UnsavedChanges;
257             if (ParentViewModel != null)
258                 ParentViewModel.SetAboutInfo(UnsavedChanges);
259             Task.Run(async () => { await Validate(columnName, value,
                ModelBL.GetModelID()); });
260         }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Jak można zaobserwować pole Deposit odwołuje się do Modelu przez logikę biznesową. Przed uaktualnieniem wartości wywoływana jest metoda *DoBeforeUpdateBusinessMagic*, która w obecnej wersji aplikacji jest pusta, jednak została ona zaimplementowana, gdyby w przyszłości była potrzeba wykonania jakichś akcji. Po aktualizacji wartości wywoływana jest metoda *DoAfterUpdateBusinessMagic*, która najpierw wysyła informację do nadrzędnego ViewModelu (w ViewModelu Kosztorysu jest to Zamówienie, ponieważ Kosztorys jest częścią widoku Zamówienia) o wykonanych zmianach. Następnie w nowym wątku odbywa się sama walidacja. Wykorzystywane jest tutaj pole z dopiskiem *ErrorInfo*, dlatego zawsze podczas tworzenia pól dla widoku trzeba pamiętać aby takie pole dla komunikatu błędu stworzyć.

7.11 Aplikacja okienkowa - AllViewModel



Źródło: opracowanie własne. Diagram wygenerowany przy użyciu Class designer Visual Studio

Klasy dziedziczące po *AllViewModel* opreują na kolekcji Modeli przechowywanych w polu *Models*. Realizowane jest tutaj usuwanie i przywracanie Modeli z bazy danych. Tak jak w klasie opisywanej wcześniej, tutaj również zostało uaktualnione drukowanie.

7.12 Aplikacja okienkowa - AllViewModel drukowanie

```
300         protected virtual async Task<IEnumerable<IEnumerable<string>>>
              GetPrintingTable()
301         { return await Task.FromResult(new List<List<string>>()); }
302
303         protected override async Task<string> GetHTMLPrintCodeAsync()
304         {
305             IEnumerable<IEnumerable<string>> table = await GetPrintingTable();
306             if (table.IsNullOrEmpty())
307                 return "";
308             string code = "<div class='row'>\n<table>\n<thead>\n<tr>\n";
309             foreach (string item in table.First())
310             {
311                 code += "<th>" + item + "</th>\n";
312             }
313             code += "</tr>\n";
314             code += "</thead>\n";
315             if (table.Count() > 1)
316             {
317                 code += "<tbody>\n";
318                 foreach (IEnumerable<string> row in table.Skip(1))
319                 {
320                     code += "<tr>\n";
321                     foreach (string item in row)
322                     {
323                         code += "<td>" + item + "</td>";
324                     }
325                     code += "\n</tr>\n";
326                 }
327                 code += "</tbody>\n";
328             }
329             code += "</table></div>";
330             return code;
331         }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Na dołączonej grafice widzimy nową metodę wirtualną *GetPrintingTableAsync()* wypełnioną ”na pusto”. Ma ona za zadanie zwrócić kolekcję kolekcji, gdzie pierwsza kolekcja będzie zawierała zawartość nagłówka tabeli, a każda kolejna wartości pojedynczego wiersza. Na podstawie tych kolekcji tworzony jest kod tabeli[9]. Przykładowa metoda zwracająca kolekcje do drukowania wygląda następująco.

[9] Kod tabeli stworzono na podstawie oficjalnej dokumentacji Material Design: https://materializecss.com/table.html

7.13 Aplikacja okienkowa - AnnotationsViewModel drukowanie

```
101         protected override async Task<IEnumerable<IEnumerable<string>>>
              GetPrintingTable()
102         {
103             List<IEnumerable<string>> table = new List<IEnumerable<string>>
104             {
105                 new string[]
106                 {
107                     DisplayNames.TitleDisplayName,
108                     DisplayNames.WakeTimeDisplayName,
109                     DisplayNames.DeadlineDisplayName,
110                     DisplayNames.IsDoneDisplayName,
111                     DisplayNames.OrderDisplayName
112                 }
113             };
114
115             table.AddRange(Models.Select(item => new string[]
116             {
117                 item.Title,
118                 item.WakeTime.ToString(),
119                 item.Deadline.ToString(),
120                 item.IsDone ? ViewResources.Yes : ViewResources.No,
121                 item.Order.Title
122             }));
123             return await Task.FromResult(table);
124         }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

W klasie *AllViewModel* zrealizowano także filtrowanie.

7.14 Aplikacja okienkowa - AllViewModel filtrowanie

```
215         private void FilterAndAssignModels() => Models = new
              ObservableCollection<ModelClass>(ModelBL.FilterModels(GetModels()));
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Wykorzystywana jest tutaj logika biznesowa. Podobnie jak w aplikacji internetowej potrzeba najpierw wypełnić wszystkie filtry, dlatego kontrolki na widok za nie odpowiedzialne są odpowiednio zbindowane.

7.15 Aplikacja okienkowa - OrdersViewModel filtr

```
36          public int HasCustomerFilter
37          {
38              get => (((OrderBL)ModelBL)?.HasCustomerFilter).GetValueOrDefault();
39              set
40              {
41                  if (value != ((OrderBL)ModelBL).HasCustomerFilter)
42                  {
43                      ((OrderBL)ModelBL).HasCustomerFilter = value;
44                      OnPropertyChanged(() => HasCustomerFilter);
45                  }
46              }
47          }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Zmiana interfejsu użytkownika

Powyżej przedstawiono Bazowe klasy ViewModel dla aplikacji okienkowej. Każda z klas, które je implementowała przeznaczona jest do wykorzystania w interfejsie użytkownika. Wspomniano już, że metody *InitializeAsync* służą do zainicjalizowania widoku. Jest to wypełnienie listy lub pobranie z bazy danych konkretnego Modelu. Dzieje się to dopiero po tym jak użytkownik zobaczy nowy interfejs. Na czas tej operacji interfejs jest dezaktywowany poprzez pole *IsEnabled*. Klasą bezpośrednio odpowiedzialną za tworzenie nowych interfejsów użytkownika, wydobywanie z nich ViewModel (podczas tworzenia widoku automatycznie przypisywany jest do niego odpowiedni ViewModel) oraz ich inicjalizację jest *UserControlCreator*. Jest to statyczna klasa zawierająca długą listę metod, każda odpowiedzialna za stworzenie widoku, zainicjowanie go i zwrócenia ViewModel. Dla każdego Modelu zostało stworzonych pięć metod tworzących widok:

- z listą wszystkich Modeli,
- bez inicjalizacji,
- z nowym Modelem,
- z Modelem pobranym z bazy danych,
- z przypisana logiką biznesową (a więc także Modelem).

Warto tutaj wspomnieć, że trzy ostatnie metody korzystają najpierw z metody bez inicjalizacji, by później ViewModel poddać jedynie odpowiedniej inicjalizacji.

Wykorzystywane jest tutaj rozszerzenie MvvmLightLibs [10], która umożliwia przesyłanie wiadomości pomiędzy obiektami poszczególnych klas. Przykład możemy zobaczyć poniżej.

7.16 Aplikacja okienkowa - UserControlCreator na podstawie Annotation

```
701         public static async Task<AnnotationsViewModel>
              CreateAllAnnotationsViewModelAsync()
702         {
703             var View = new AnnotationsView();
704             Messenger.Default.Send(new MainWindowContentMessage(View));
705             var dataContext = (AnnotationsViewModel)View.DataContext;
706             await dataContext.DoInitializationAsync();
707             return dataContext;
708         }
709
710         public static AnnotationViewModel CreateAnnotationViewModel()
711         {
712             var View = new AnnotationView();
713             Messenger.Default.Send(new MainWindowContentMessage(View));
714             AnnotationViewModel dataContext = (AnnotationViewModel)
                  View.DataContext;
715             return dataContext;
716         }
717
718         public static async Task<AnnotationViewModel>
              CreateNewAnnotationViewModelAsync()
719         {
720             var ViewModel = CreateAnnotationViewModel();
721             await ViewModel.DoInitializationAsync();
722             return ViewModel;
723         }
724
725         public static async Task<AnnotationViewModel>
              CreateGetAnnotationViewModelAsync(int id)
726         {
727             var ViewModel = CreateAnnotationViewModel();
728             await ViewModel.DoInitializationAsync(id);
729             return ViewModel;
730         }
731
732         public static async Task<AnnotationViewModel>
              CreateSetAnnotationViewModelAsync(AnnotationBL modelBL)
733         {
734             var ViewModel = CreateAnnotationViewModel();
735             await ViewModel.DoInitializationAsync(modelBL);
736             return ViewModel;
737         }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio\

[10] Rozszerzenie autorstwa Laurent Bugnion (GalaSoft) pobrane z NuGet Manager w Visual Studio. Strona rozszerzenia: https://www.nuget.org/packages/MvvmLightLibs/5.4.1.1?_src=template

W linii 713 widzimy wykorzystanie wyżej wspominanego rozszerzenia. W specjalnej wiadomości wysyłany jest nowo stworzony widok, który następnie odbierany jest w ViewModelu okna głównego.

7.17 Aplikacja okienkowa - MainWindowViewModel - zmiana zawartości

```
66          public MainWindowViewModel() : base()
67          {
68              Start();
69              Messenger.Default.Register<MainWindowContentMessage>(this, item =>
                  { SetContent(item.UserControl); });
70          }
71
72          private void SetContent(UserControl userControl)
73          {
74              Content = userControl;
75          }
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Aby wiadomość została poprawnie odebrana należało ją zarejestrować w konstruktorze. Następnie po jej odebraniu wywoływana jest metoda *SetContent*, która przypisuje widok do pola *Content*, do którego zbindowana jest zawartość okna.

7.4. Warstwa View

Podczas tworzenia widoków zostało wykorzystanie rozszerzenie Infragistics.Themes.MetroDark.Wpf[11].

Okno główne

[11] Rozszerzenie autorstwa brianlagunas pobrane z NuGet Manager w Visual Studio. Strona rozszerzenia: https://www.nuget.org/packages/Infragistics.Themes.MetroDark.Wpf/1.0.0?_src=template

7.18 Aplikacja okienkowa - Strona główna

```
                OpenWorkersCommandAsync, IsAsync=True}"/>
37              </MenuItem>
38              <MenuItem Header="{x:Static BL:DisplayNames.MaterialsDisplayName}">
39                  <MenuItem Foreground="Black" Header="{x:Static
                      BL:DisplayNames.MaterialsDisplayName}" Command="{Binding
                      OpenMaterialsCommandAsync, IsAsync=True}"/>
40                  <MenuItem Foreground="Black" Header="{x:Static
                      BL:DisplayNames.DeliveriesDisplayName}" Command="{Binding
                      OpenDeliveriesCommandAsync, IsAsync=True}"/>
41                  <Separator/>
42                  <MenuItem Foreground="Black" Header="{x:Static
                      BL:DisplayNames.SuppliersDisplayName}" Command="{Binding
                      OpenSuppliersCommandAsync, IsAsync=True}"/>
43              </MenuItem>
44              <MenuItem Header="{x:Static
                  BL:DisplayNames.AnnotationsDisplayName}" Command="{Binding
                  OpenAnnotationsCommandAsync, IsAsync=True}"/>
45          </Menu>
46          <ScrollViewer Grid.Row="1" Content="{Binding Content}" Margin="16 32"/>
47          <StackPanel Orientation="Horizontal" Grid.Row="2">
48              <Label Content="{Binding CompanyName}"/>
49          </StackPanel>
50      </Grid>
51  </Window>
52
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

7.19 Aplikacja okienkowa kod 1

```
<Window
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:Intertegra.View"
        xmlns:BL="clr-namespace:BusinessLogic;assembly=BusinessLogic"
        xmlns:Intertegra="clr-namespace:Intertegra" xmlns:SingleViews="clr-
          namespace:Intertegra.View.SingleViews"
          x:Class="Intertegra.View.MainWindow"
        mc:Ignorable="d"
        Title="Intertegra" WindowState="Maximized"
        Background="DimGray"
        Foreground="White"
        d:DesignWidth="1080" Icon="/Resources/logo.png">
    <Window.Resources>
        <ResourceDictionary Source="MainResources.xaml"/>
    </Window.Resources>
    <Window.DataContext>
        <Intertegra:MainWindowViewModel/>
    </Window.DataContext>
    <Grid IsEnabled="{Binding IsEnabled}" Background="{StaticResource
      AccordionArrowExpandedBackgroundBrush}">
        <Grid.RowDefinitions>
            <RowDefinition Height="auto"/>
            <RowDefinition/>
            <RowDefinition Height="auto"/>
        </Grid.RowDefinitions>
        <Menu Background="Transparent" Foreground="{StaticResource Brush03}">
            <MenuItem Header="{x:Static BL:DisplayNames.PhotosDisplayName}"
              Command="{Binding OpenPhotosCommandAsync, IsAsync=True}"/>
            <MenuItem Header="{x:Static BL:DisplayNames.CustomersDisplayName}"
              Command="{Binding OpenCustomersCommandAsync, IsAsync=True}"/>
            <MenuItem Header="{x:Static BL:DisplayNames.OrdersDisplayName}">
                <MenuItem Foreground="Black" Header="{x:Static
                  BL:DisplayNames.OrdersDisplayName}" Command="{Binding
                  OpenOrdersCommandAsync, IsAsync=True}"/>
                <Separator />
                <MenuItem Foreground="Black" Header="{x:Static
                  BL:DisplayNames.RoofTypesDisplayName}" Command="{Binding
                  OpenRoofTypesCommandAsync, IsAsync=True}"/>
                <MenuItem Foreground="Black" Header="{x:Static
                  BL:DisplayNames.ConstructionTypesDisplayName}"
                  Command="{Binding OpenConstructionTypesCommandAsync,
                  IsAsync=True}"/>
                <MenuItem Foreground="Black" Header="{x:Static
                  BL:DisplayNames.AttachmentMethodsDisplayName}"
                  Command="{Binding OpenAttachmentMethodsCommandAsync,
                  IsAsync=True}"/>
                <MenuItem Foreground="Black" Header="{x:Static
                  BL:DisplayNames.SectionsOfStructuralElementsDisplayName}"
                  Command="{Binding
                  OpenSectionsOfStructuralElementsCommandAsync, IsAsync=True}"/
                  >
                <MenuItem Foreground="Black" Header="{x:Static
                  BL:DisplayNames.WorkersDisplayName}" Command="{Binding
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

7.20 Aplikacja okienkowa kod 2

```
                OpenWorkersCommandAsync, IsAsync=True}"/>
37              </MenuItem>
38              <MenuItem Header="{x:Static BL:DisplayNames.MaterialsDisplayName}">
39                  <MenuItem Foreground="Black" Header="{x:Static
                      BL:DisplayNames.MaterialsDisplayName}" Command="{Binding
                      OpenMaterialsCommandAsync, IsAsync=True}"/>
40                  <MenuItem Foreground="Black" Header="{x:Static
                      BL:DisplayNames.DeliveriesDisplayName}" Command="{Binding
                      OpenDeliveriesCommandAsync, IsAsync=True}"/>
41                  <Separator/>
42                  <MenuItem Foreground="Black" Header="{x:Static
                      BL:DisplayNames.SuppliersDisplayName}" Command="{Binding
                      OpenSuppliersCommandAsync, IsAsync=True}"/>
43              </MenuItem>
44              <MenuItem Header="{x:Static
                  BL:DisplayNames.AnnotationsDisplayName}" Command="{Binding
                  OpenAnnotationsCommandAsync, IsAsync=True}"/>
45          </Menu>
46          <ScrollViewer Grid.Row="1" Content="{Binding Content}" Margin="16 32"/>
47          <StackPanel Orientation="Horizontal" Grid.Row="2">
48              <Label Content="{Binding CompanyName}"/>
49          </StackPanel>
50      </Grid>
51  </Window>
52
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

W liniach 17-19 widoczne jest przypisanie do widoku DataContext, czyli jednej z klas ViewModel. W liniach 14-16 przypisuje się do widoki Resources, plik ze stylami widoku. Jako pierwsze element zawartości widoku widzimy pasek menu z przyciskami do wywoływania okien. Następnie do widoku przewijanych (ScrollViewer) zbindowano pole Content, które jest główną częścią widoku. W stopce umieszczono nazwę firmy.

Przykładowy widok pojedynczego Modelu

7.21 Aplikacja okienkowa - widok Annotation

```
17      <StackPanel>
18          <Button Content="Otwórz zamówienie" Command="{Binding BachToOrderAsync,
               IsAsync=True}" HorizontalAlignment="Left"/>
19          <ItemsControl Style="{StaticResource StandardFields}">
20              <ItemsControl Style="{StaticResource StandardLabelAndField}"
                  ToolTip="{x:Static BL:DisplayNames.TitleDisplayName}">
21                  <Label Content="{x:Static BL:DisplayNames.TitleDisplayName}"/>
22                  <TextBox Text="{Binding Title}" Style="{StaticResource
                      TextBoxStandard}"/>
23                  <Label Content="{Binding TitleErrorInfo}" ToolTip="{Binding
                      TitleErrorInfo}" Foreground="Red"/>
24              </ItemsControl>
25              <ItemsControl Style="{StaticResource StandardLabelAndField}"
                  ToolTip="{x:Static BL:DisplayNames.WakeTimeDisplayName}">
26                  <Label Content="{x:Static
                      BL:DisplayNames.WakeTimeDisplayName}"/>
27                  <DatePicker SelectedDate="{Binding WakeTime}"/>
28                  <Label Content="{Binding WakeTimeErrorInfo}" ToolTip="{Binding
                      WakeTimeErrorInfo}" Foreground="Red"/>
29              </ItemsControl>
30              <ItemsControl Style="{StaticResource StandardLabelAndField}"
                  ToolTip="{x:Static BL:DisplayNames.DeadlineDisplayName}">
31                  <Label Content="{x:Static
                      BL:DisplayNames.DeadlineDisplayName}"/>
32                  <DatePicker SelectedDate="{Binding Deadline}"/>
33                  <Label Content="{Binding DeadlineErrorInfo}" ToolTip="{Binding
                      DeadlineErrorInfo}" Foreground="Red"/>
34              </ItemsControl>
35              <ItemsControl Style="{StaticResource StandardLabelAndField}"
                  ToolTip="{x:Static BL:DisplayNames.IsDoneDisplayName}">
36                  <Label Content="{x:Static BL:DisplayNames.IsDoneDisplayName}"/>
37                  <CheckBox IsChecked="{Binding IsDone}"/>
38                  <Label Content="{Binding IsDoneErrorInfo}" ToolTip="{Binding
                      IsDoneErrorInfo}" Foreground="Red"/>
39              </ItemsControl>
40          </ItemsControl>
41          <ItemsControl Style="{StaticResource LargeFields}">
42              <ItemsControl Style="{StaticResource LargeLabelAndField}"
                  ToolTip="{x:Static BL:DisplayNames.NotesDisplayName}">
43                  <Label Content="{x:Static BL:DisplayNames.NotesDisplayName}"/>
44                  <TextBox Text="{Binding Notes}" Style="{StaticResource
                      TextBoxLarge}"/>
45                  <Label Content="{Binding NotesErrorInfo}" ToolTip="{Binding
                      NotesErrorInfo}" Foreground="Red"/>
46              </ItemsControl>
47          </ItemsControl>
48      </StackPanel>
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

Do tworzenia widoków pojedynczego Modelu wykorzystano kontrolkę *ItemsControl* z podpiętym stylem *StandardFields*. Układa on wszystkie elementy w czteroelementowej siatce. Każde pole również zapisane jest w *ItemsControl*, które z kolei używa stylu, który umieszcza elementy jeden pod drugim. Zamiast *StandardFields* wykorzystuje się także *LargeFields*, który

umieszcza elementy jeden pod drugim. Podobnie jak w aplikacji internetowej do wyświetlania nazw pól wykorzystuje się pliki Resources. Pod kontrolką, do której zbindowana jest wartość pola umieszcza się pole tekstowe z zawartością błędu. Wszystkie takie widoki dziedziczą po *SingleView*, który umieszcza na widoku przyciski do zapisu, druku oraz odświeżenia. I powrotu do listy.

Przykładowy widok wszystkich Modeli

7.22 Aplikacja okienkowa - widok Annotations

```
18      <StackPanel>
19          <ItemsControl Style="{StaticResource StandardFields}">
20              <ItemsControl Style="{StaticResource StandardLabelAndField}">
21                  <Label Content="{x:Static BL:DisplayNames.IsDoneDisplayName}"/>
22                  <ComboBox ItemsSource="{Binding FilterOptions.Options}"
                      SelectedValue="{Binding IsDoneFilter}"
                      DisplayMemberPath="Value" SelectedValuePath="Key"/>
23              </ItemsControl>
24              <ItemsControl Style="{StaticResource StandardLabelAndField}">
25                  <Label Content="{x:Static
                      BLV:ViewResources.DoesWakeTimePassed}"/>
26                  <ComboBox ItemsSource="{Binding FilterOptions.Options}"
                      SelectedValue="{Binding DoesWakeTimePassedFilter}"
                      DisplayMemberPath="Value" SelectedValuePath="Key"/>
27              </ItemsControl>
28              <ItemsControl Style="{StaticResource StandardLabelAndField}">
29                  <Label Content="{x:Static
                      BLV:ViewResources.DoesDeadlinePassed}"/>
30                  <ComboBox ItemsSource="{Binding FilterOptions.Options}"
                      SelectedValue="{Binding DoesDeadlinePassedFilter}"
                      DisplayMemberPath="Value" SelectedValuePath="Key"/>
31              </ItemsControl>
32          </ItemsControl>
33          <DataGrid ItemsSource="{Binding Models}" Style="{StaticResource
              StandardAllViewDataGrid}">
34              <DataGrid.Columns>
35                  <DataGridTextColumn Binding="{Binding Title}">
36                      <DataGridTextColumn.Header>
37                          <TextBlock Text="{x:Static
                          BL:DisplayNames.TitleDisplayName}"/>
38                      </DataGridTextColumn.Header>
39                  </DataGridTextColumn>
40                  <DataGridTextColumn Binding="{Binding WakeTime}">
41                      <DataGridTextColumn.Header>
42                          <TextBlock Text="{x:Static
                          BL:DisplayNames.WakeTimeDisplayName}"/>
43                      </DataGridTextColumn.Header>
44                  </DataGridTextColumn>
45                  <DataGridTextColumn Binding="{Binding Deadline}" Width="*">
46                      <DataGridTextColumn.Header>
47                          <TextBlock Text="{x:Static
                          BL:DisplayNames.DeadlineDisplayName}"/>
48                      </DataGridTextColumn.Header>
49                  </DataGridTextColumn>
50              </DataGrid.Columns>
51          </DataGrid>
52      </StackPanel>
```

Źródło: opracowanie własne. Wydrukowano z Visual Studio

W liniach 19-32 umieszczono dodatkowe filtry na widoku. Natomiast później w kontrolce *DataGrid (tabela)* widoczna jest sama lista modeli, tutaj w nagłówkach tabeli wykorzystywane są pliki Resources. Każdy z takich widoków dziedziczy po *AllView*, który umieszcza pole wyszukiwanego tekstu, oraz przyciski do pokazania kosza, filtrowania i wyszukania, odświeżenia i drukowania.

Zablokowanie możliwości zmiany okna bez zapisu.

W trakcie tworzenia aplikacji pojawiła się potrzeba zablokowania pewnych zmian interfejsu póki nie nastąpi zapis. Na przykład póki nowe zamówienie nie zostanie zapisane, nie można dodać do jego kosztorysu nowego elementu. Wykonane to zostało w następujący sposób.

7.23 Aplikacja okienkowa - zapisz by aktywować

```
191             <Label Content="{x:Static
                  BL:DisplayNames.SaveToEnableDisplayName}"
                  Visibility="{Binding AdditionalAboutSaveInfoVisibility}"
                  Foreground="{StaticResource SaveToEnableBrush}"/>
192             <allviews:WorkRegistersView DataContext="{Binding
                  WorkRegistersViewModel}" IsEnabled="{Binding
                  NoChangesDetected}"/>
```

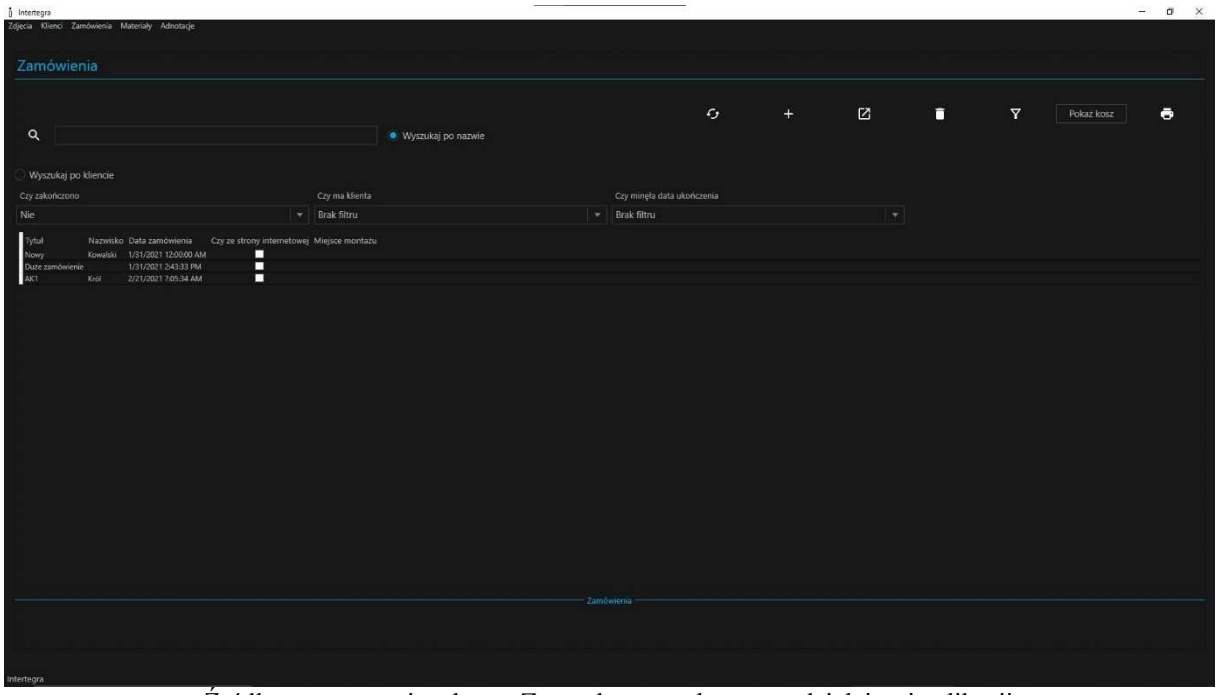
Źródło: opracowanie własne. Wydrukowano z Visual Studio

Wykorzystane tutaj zostało pole *NoChangesDetected*, które aktualizowane jest po każdej zmianie na widoku oraz przy zapisie.

7.5. Procedura dodania nowego zamówienia

1. Otwórz zamówienia i kliknij znak +.

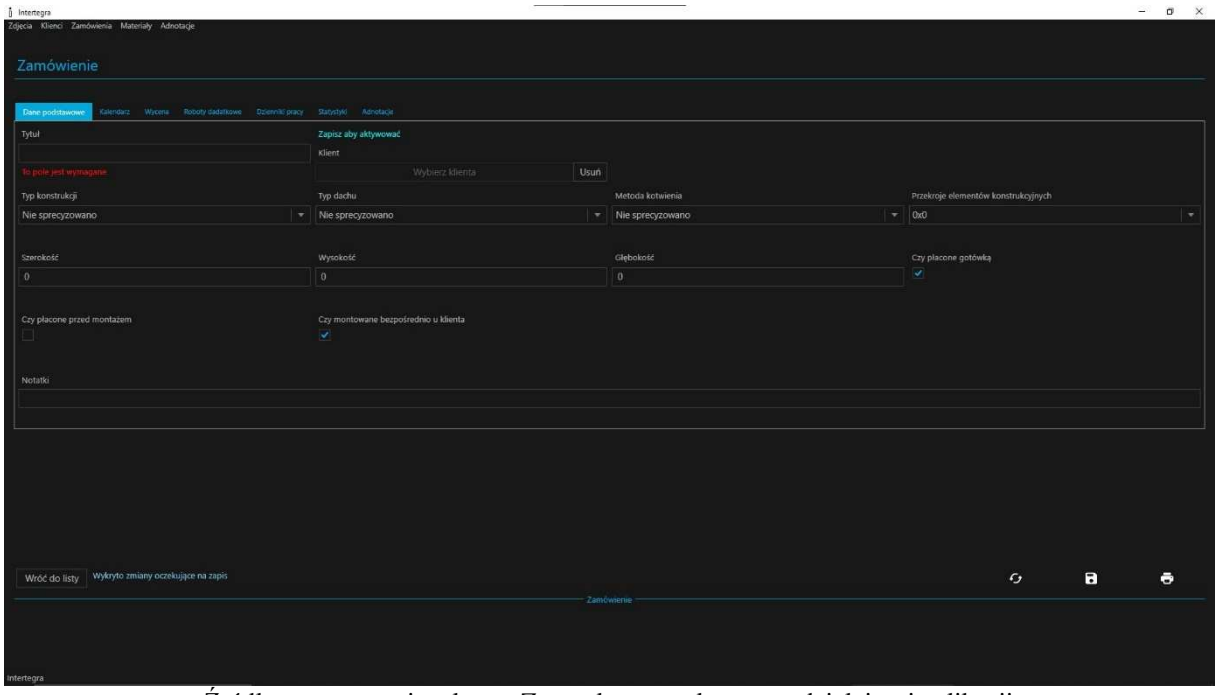
7.24 Procedura dodania nowego zamówienia 1



Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

2. Wpisz tytuł zamówienia.

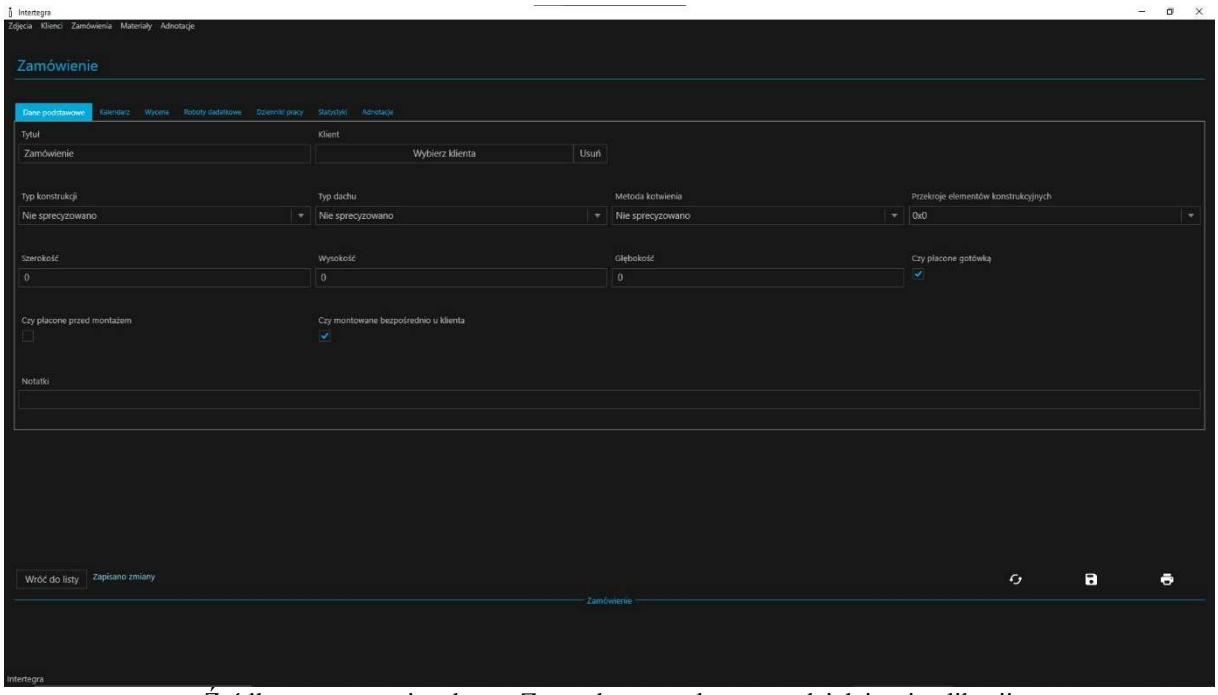
7.25 Procedura dodania nowego zamówienia 2



Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

3. Zapisz zmiany klikając ikonę dyskietki na dole.

7.26 Procedura dodania nowego zamówienia 3

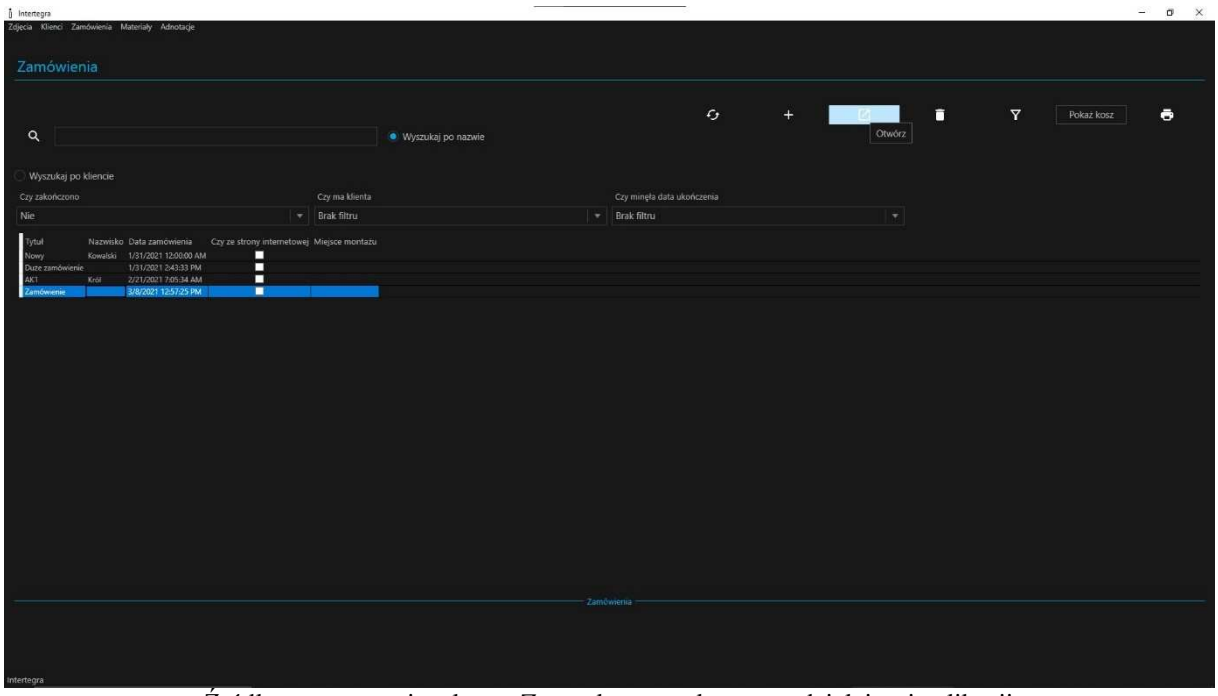


Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

7.6. Procedura dodania klienta do zamówienia

1. Otwórz zamówienia, kliknij zamówienie, do którego chcesz dodać klienta i otwórz je odpowiednim przyciskiem.

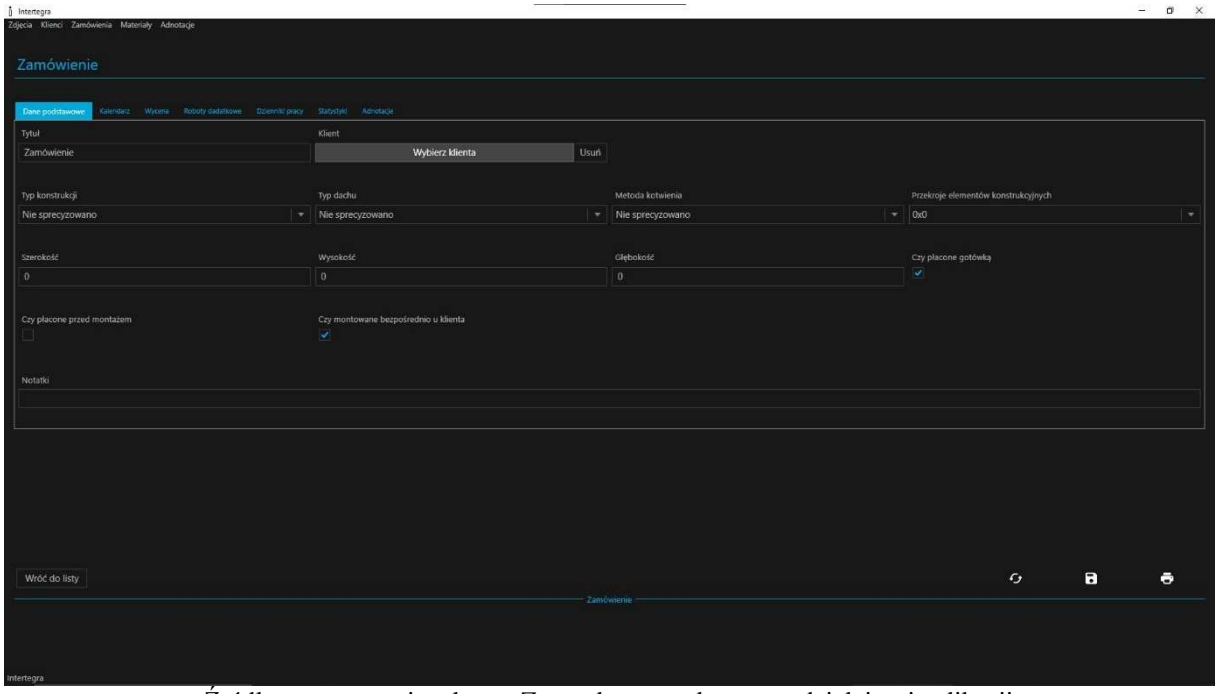
7.27 Procedura dodania klienta do zamówienia 1



Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

2. Kliknij przycisk „Wybierz klienta”.

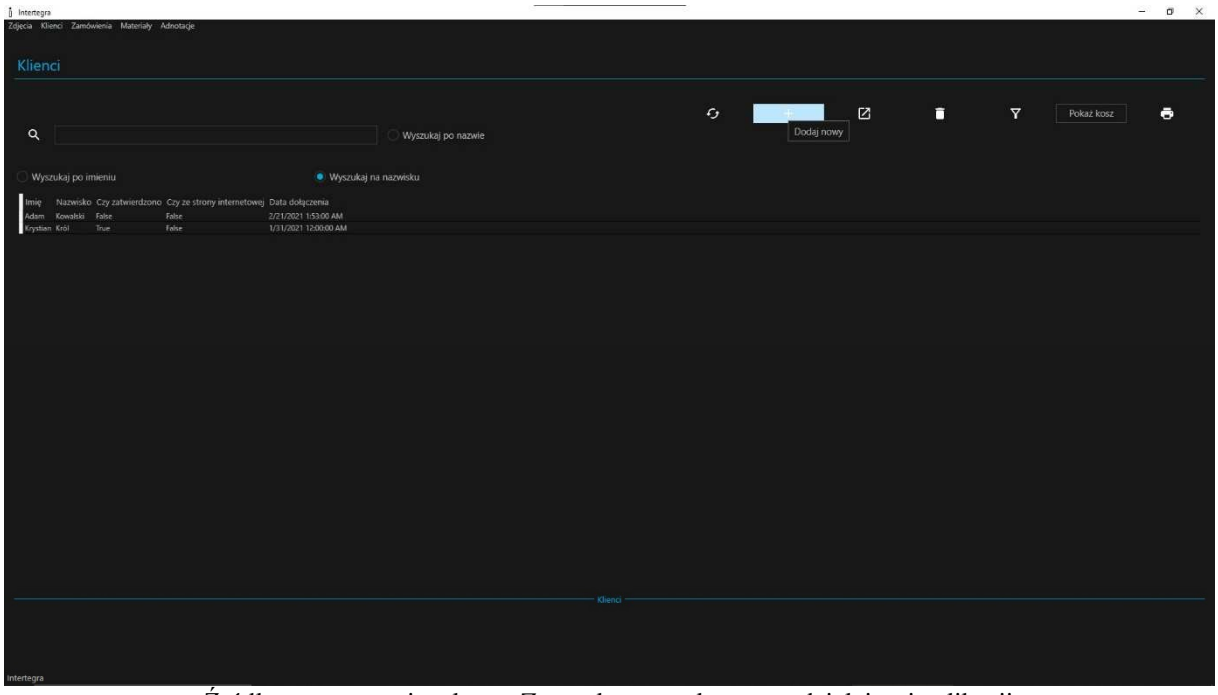
7.28 Procedura dodania klienta do zamówienia 2



Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

3. Kliknij przycisk „Dodaj nowy” (możesz też „otworzyć” wybranego klienta).

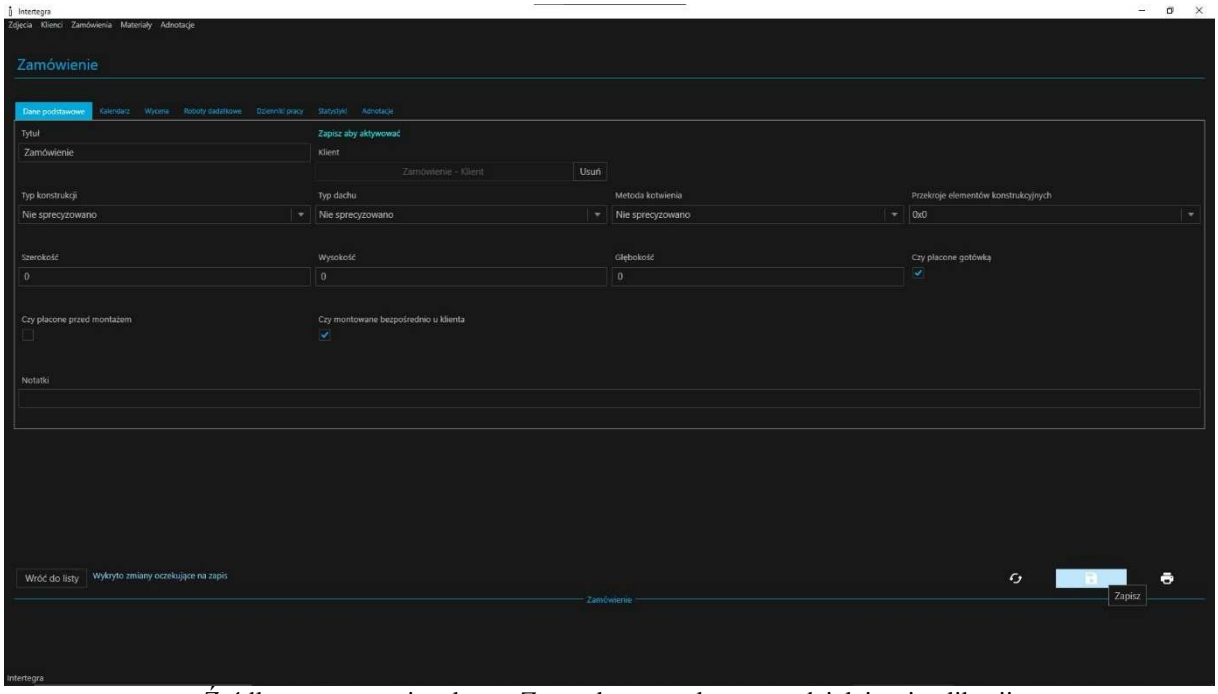
7.29 Procedura dodania klienta do zamówienia 3



Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

4. Zapisz zmiany. Następnie jeszcze raz kliknij przycisk klienta, wcześniej widniał tam napis „Dodaj klienta”.

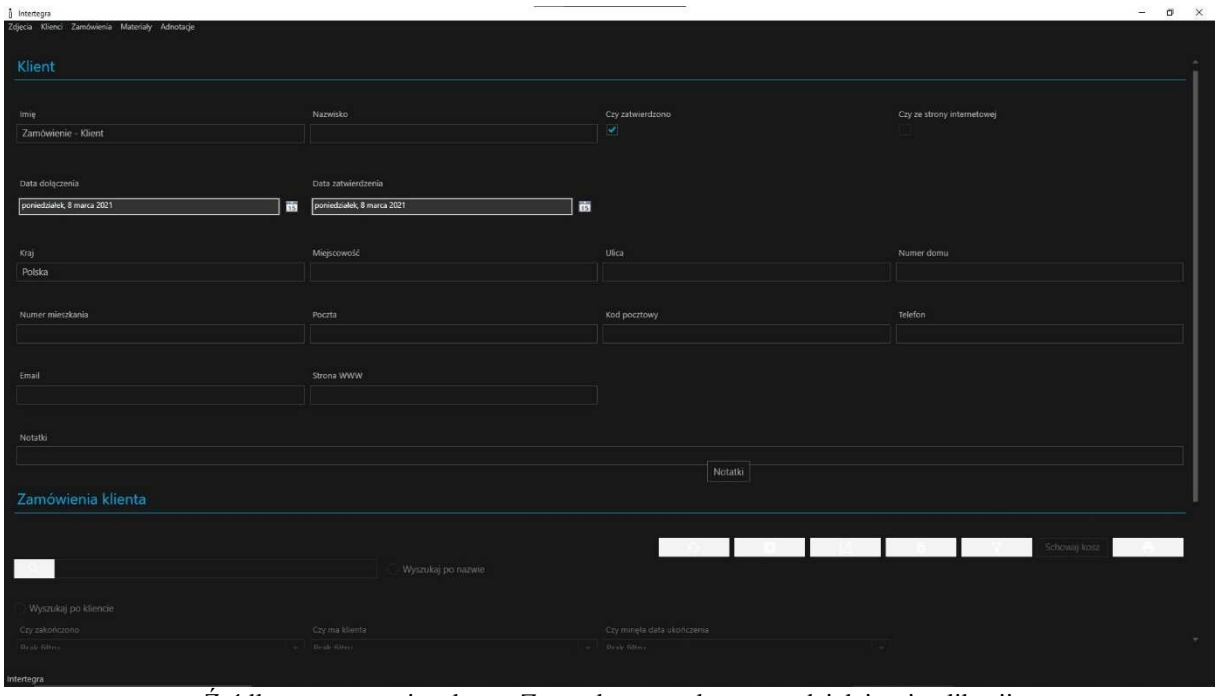
7.30 Procedura dodania klienta do zamówienia 4



Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

5. Edytuj dane klienta.

7.31 Procedura dodania klienta do zamówienia 5

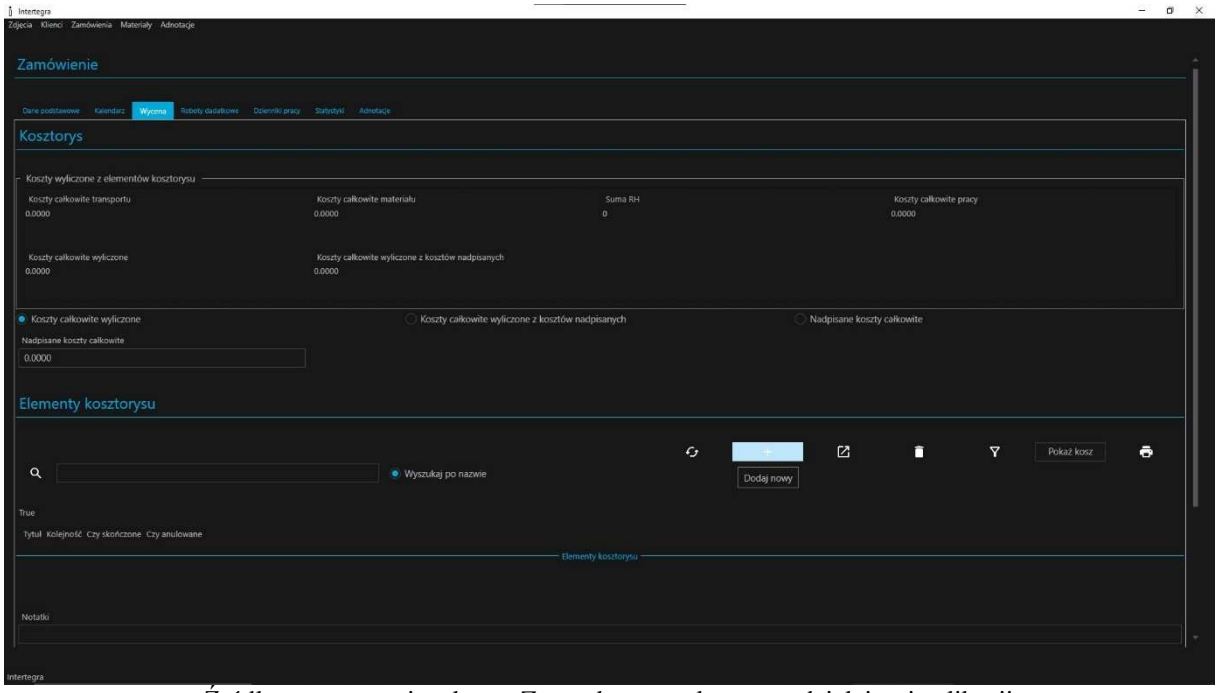


Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

7.7. Procedura edycji kosztorysu

1. Otwórz zamówienia. Na górze otwórz zakładkę „Wycena”. W strefie „Elementy kosztorysu” kliknij „Dodaj nowy”.

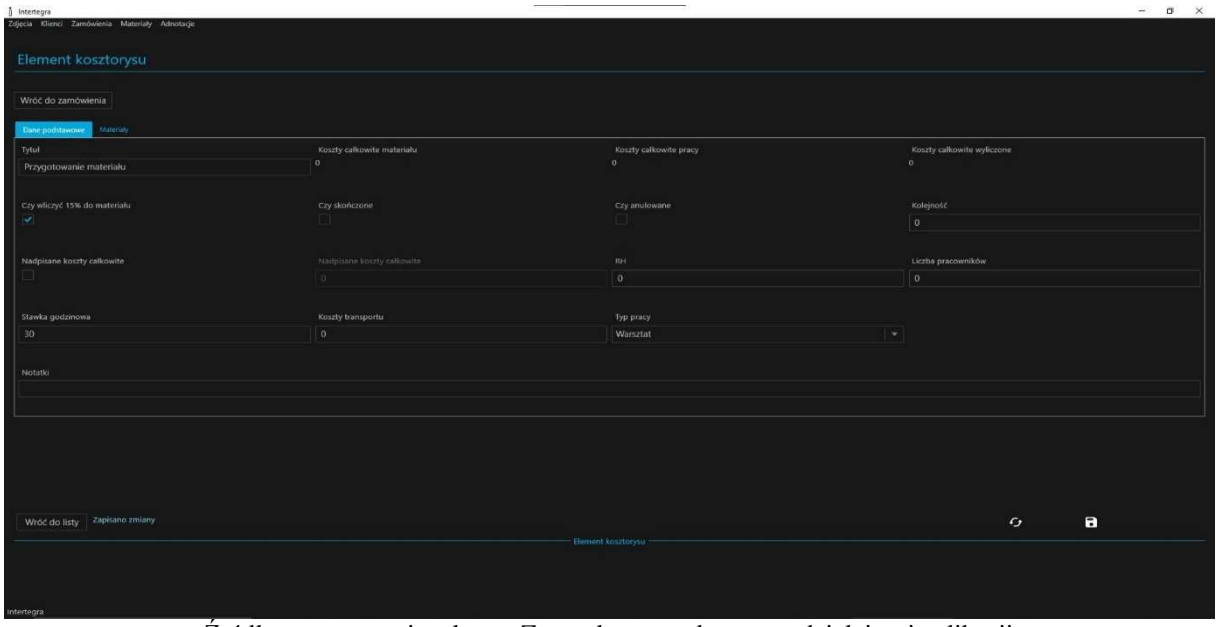
7.32 Procedura edycji kosztorysu 1



Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

2. Uzupełnij tytuł i zapisz zmiany.

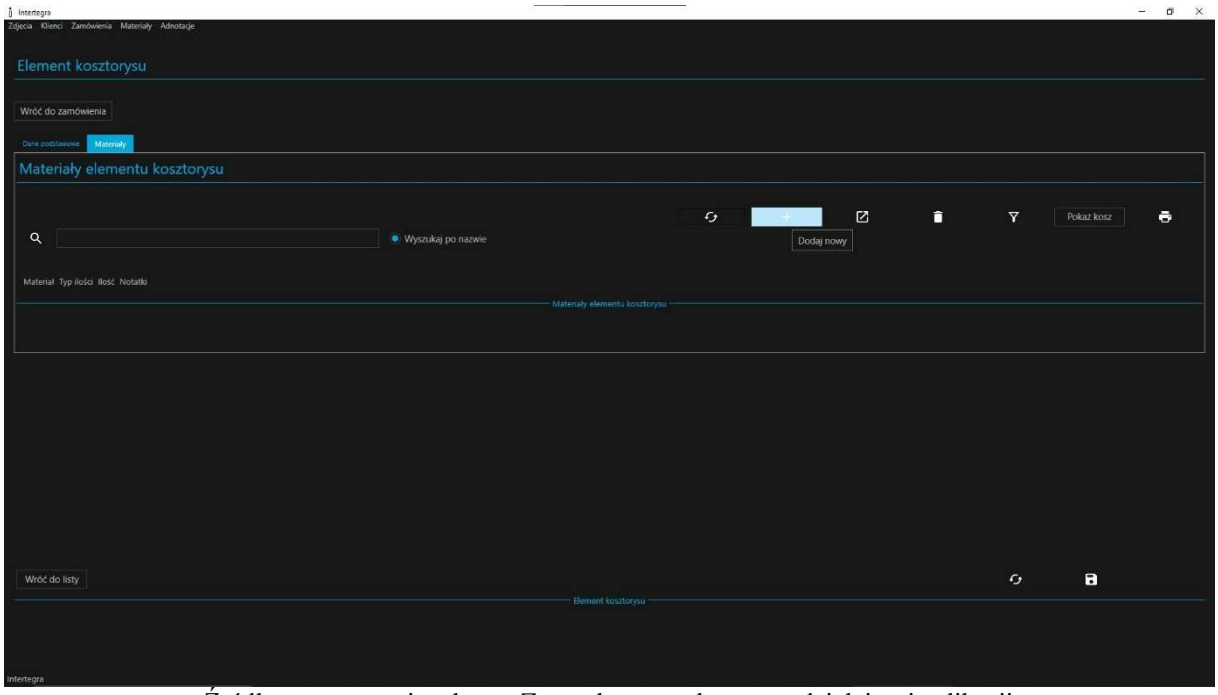
7.33 Procedura edycji kosztorysu 2



Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

3. Na górze przejdź do zakładki „Materiały” i kliknij „Dodaj nowy”.

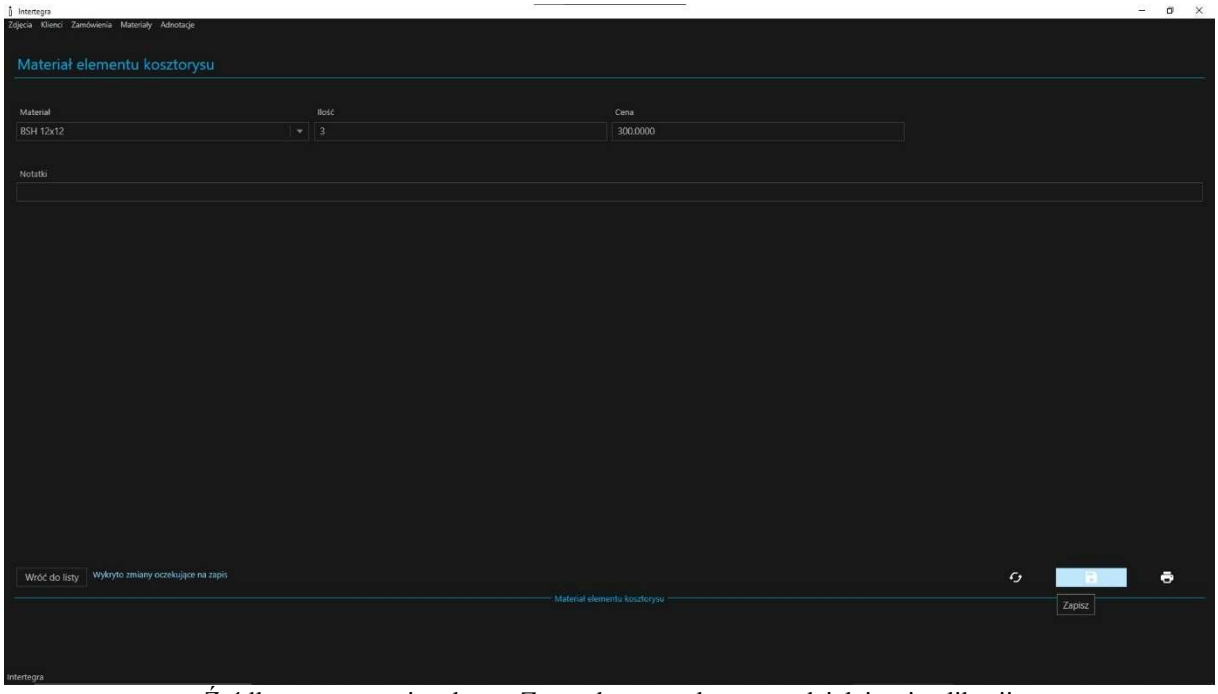
7.34 Procedura edycji kosztorysu 3



Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

4. Wybierz materiał, wpisz ilość i cenę i zapisz.

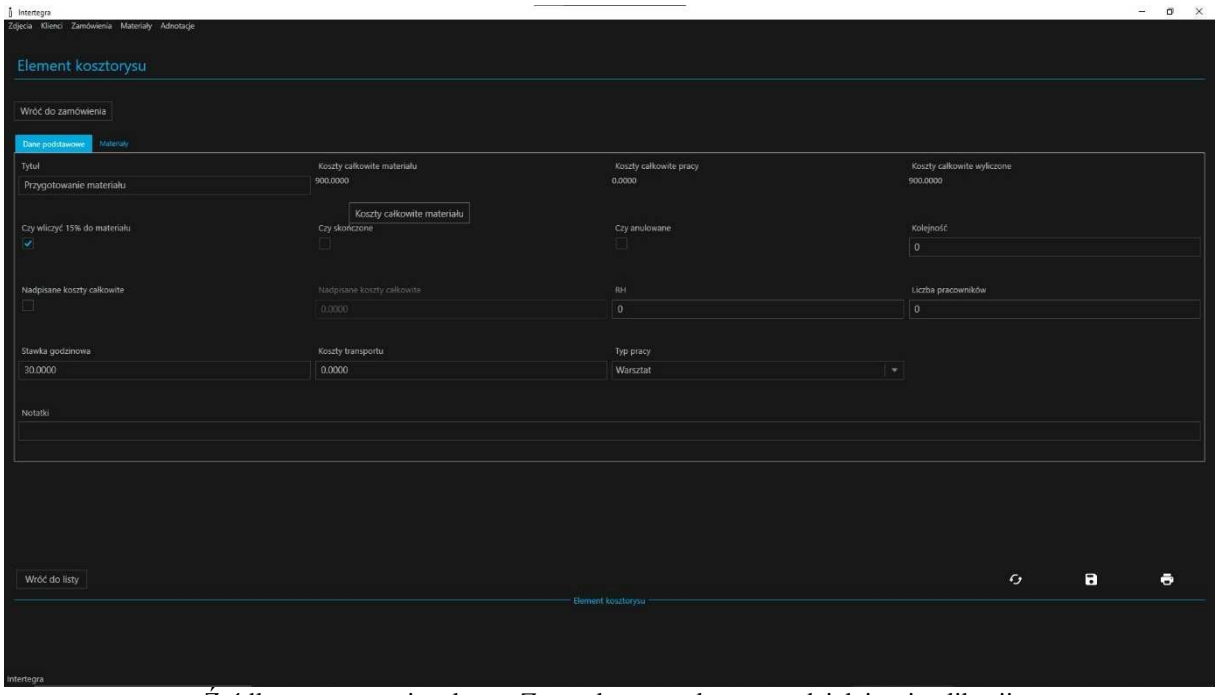
7.35 Procedura edycji kosztorysu 4



Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

5. Wróć do elementu kosztorysu klikając „Wróć do listy” i zaobserwuj zmiany w kosztach.

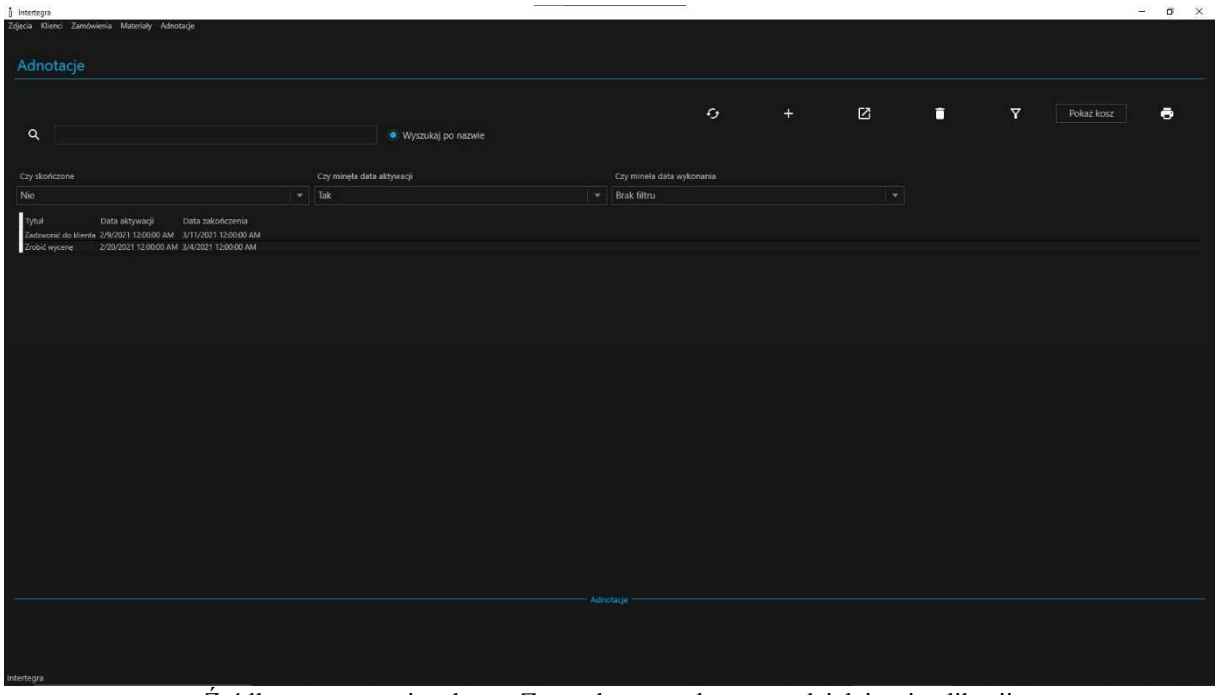
7.36 Procedura edycji kosztorysu 5



Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

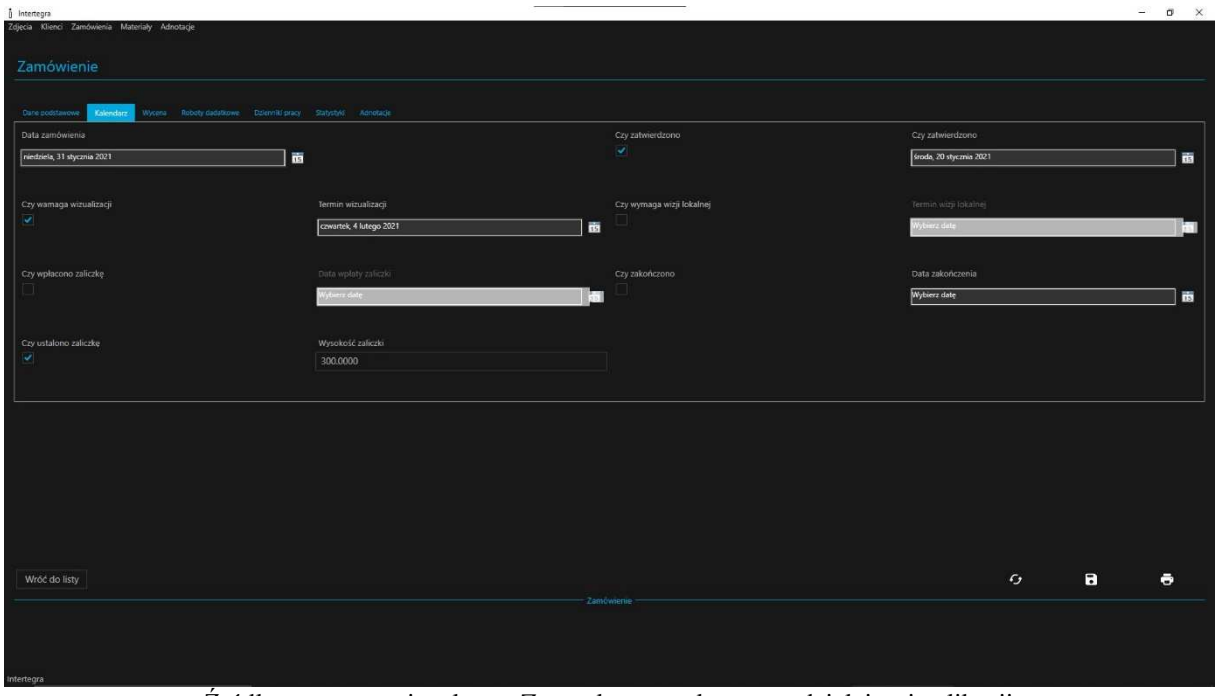
7.8. Wyróżnione widoki

7.37 Aplikacja okienkowa - widok adnotacji



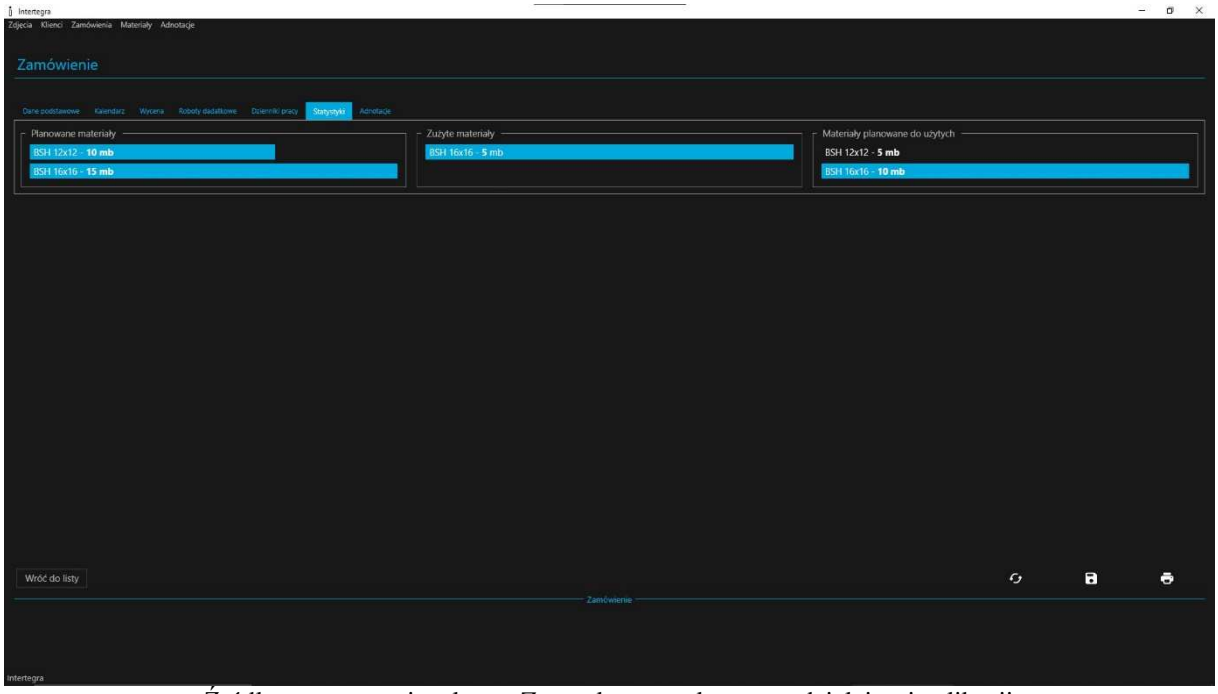
Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

7.38 Aplikacja okienkowa - widok kalendarza zamówienia



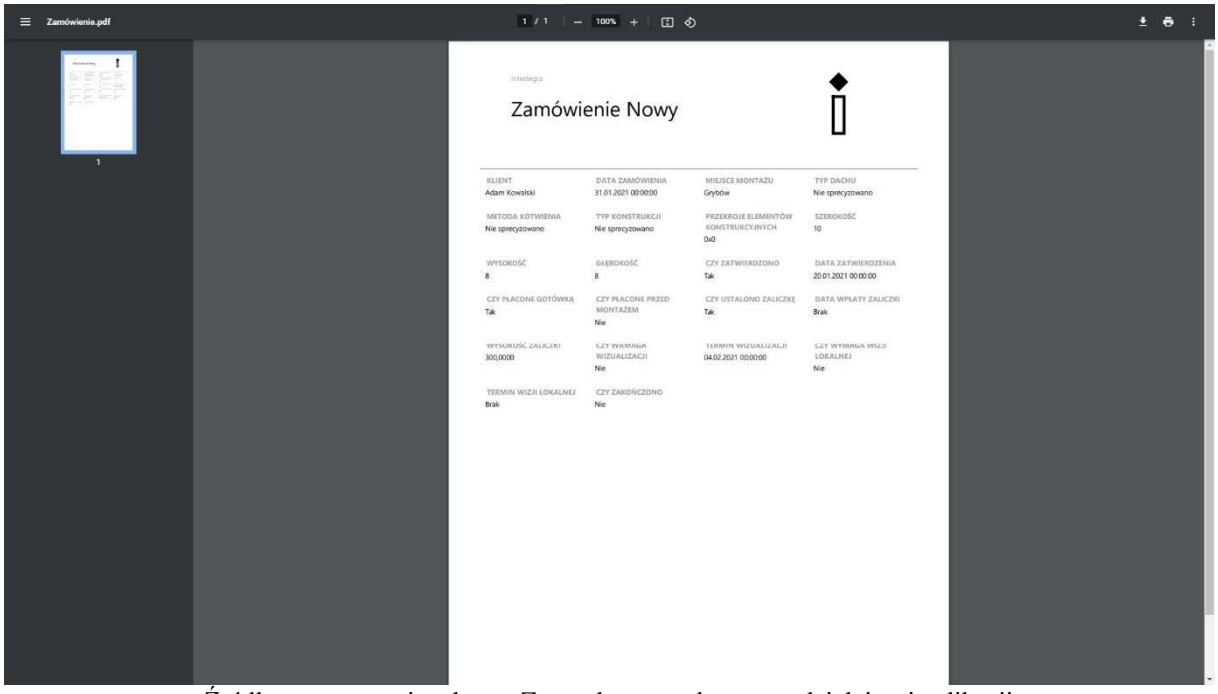
Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

7.39 Aplikacja okienkowa - widok statystyk zamówienia



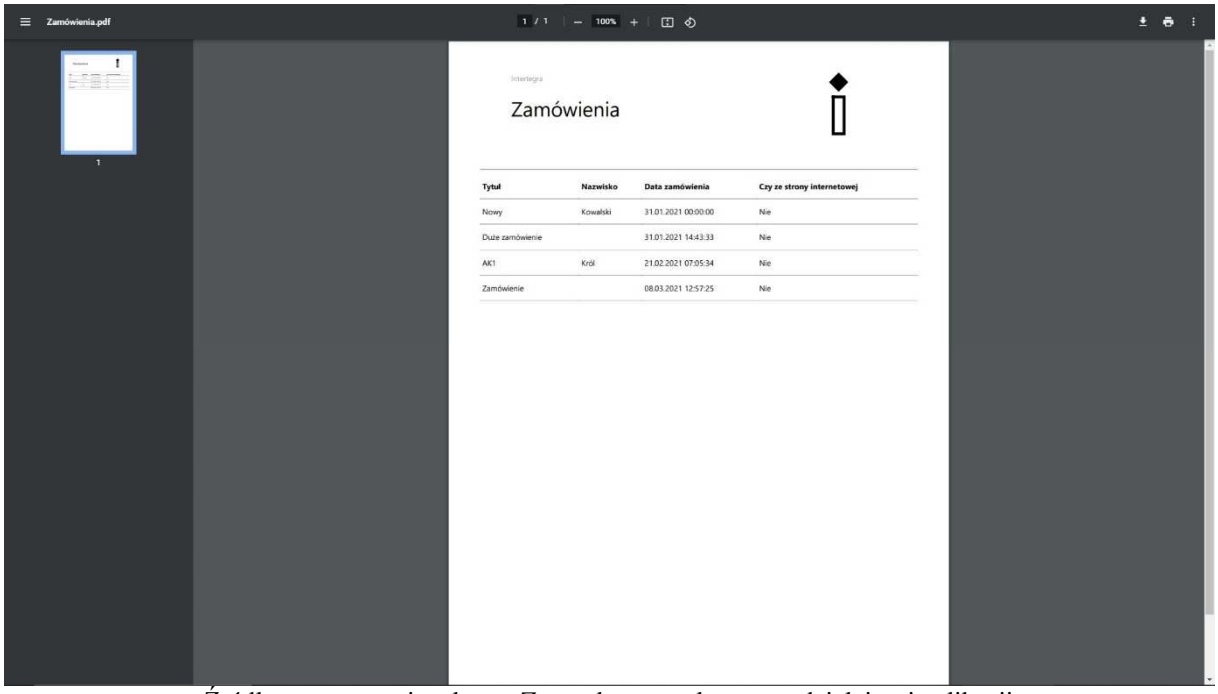
Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

7.40 Aplikacja okienkowa - widok wydrukowanego zamówienia



Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

7.41 Aplikacja okienkowa - widok wydrukowanej listy zamówień



Źródło: opracowanie własne. Zrzut ekranu wykonany z działającej aplikacji

Rozdział 8. Podsumowanie

Celem pracy było stworzenie systemu wspomagającego funkcjonowanie firmy stolarskiej. Zarówno w sferze prezentacji wykonywanych przez nią usług potencjalnym klientom jak i pomocy w zarządzaniu zamówieniami. Pierwszą funkcjonalność wypełniła aplikacja internetowa. Dzięki niej firma odbierana jest przez odwiedzających jako profesjonalna i rzetelna. To z kolei przekłada się większa ilość zapytań. Zdecydowaną zaletą jest tutaj prostota, które od początku była jednym z głównych założeń. Odwiedzający nie musi zapoznawać się z funkcjonowaniem strony, a całą uwagę skupić na produkcie. Aplikacja okienkowa w odpowiedni, dopasowany do sposobu funkcjonowania firmy, wspomaga zarządzanie zamówieniami. Prócz podstawowych danych o kliencie widok kalendarza pozwala usystematyzować i zaplanować pracę. Dużą rolę spełniają tutaj adnotacje, które można stworzyć z dużym wyprzedzeniem, a które nie pozwolą zapomnieć o najważniejszych spotkaniach lub pracach. Niecenioną pomocą jest system kosztorysów, który przyspiesza proces wyceny zamówienia. Dodatkowo dzienniki pracy pozwalają podsumować prace. Ocenić jak planowane koszty i zarobki mają się do realnie wykonanej pracy i poświęconego czasu.

Sporą trudnością podczas projektowania obu projektów, które z perspektywy kodu są ze sobą mocno powiązane, było wypracowanie systemu, który będzie łatwy w późniejszej modyfikacji. Obszerne komentarze w kodzie z pewnością ułatwią późniejsze prace. Chociaż praca nad abstrakcyjnością i rozbudowa klas bazowych przysporzyła bardzo wiele trudności i komplikacji to jednak sprawiła, że dodanie nowych funkcjonalności, które doskonale wpasują się w całość działającego systemu, jest już procesem łatwym i niewymagającym dużych ilości kodu.

W przyszłości system na pewno będzie podlegał pewnej ewolucji. Pomyśleć tutaj można o systemie notującym historię wpłat klienta, dodawanie zdjęć do zamówienia lub automatycznego tworzenia kosztorysu na podstawie projektu 3d wykonanego w zewnętrznym programie. Takie funkcjonalności będzie można z czasem dodać z łatwością, bez konieczności przebudowy całego projektu.

Aplikacja internetowa była początkowo testowana pod innym adresem witryny, niż ten używany przez firmę. Zostały dodane wymagane produkty i strona była przez pracowników firmy oraz osoby z nią związane sprawdzana w poszukiwaniu błędów. W tym momencie zrezygnowano, ze względów biznesowych, z opcji bezpośredniego złożenia zamówienia przez klienta na stronie produktu przez formularz. Jako pewne niedociągnięcie można tutaj wymienić

nieoptymalną galerię, pod której otwarciu wszystkie zdjęcia ładują się w pełnym rozmiarze. Ten mechanizm z całą pewnością będzie musiał zostać poprawiony. Po kilku tygodniach testu strona została umieszczona pod docelowym adresem WWW z użyciem samodzielnie wykupionej i skonfigurowanej maszyny wirtualnej.

Testowanie aplikacji okienkowej odbywało się przez równoczesne z dotychczasowym sposobem prowadzenie zamówień w tej właśnie aplikacji. Następnie porównywane przez właściciela firmy były obie te metody i ich efekty. Nieco wadliwy okazał się system walidacji i zapisu, okazało się, że czasami występuje potrzeba podwójnego kliknięcia przycisku zapisu, gdyż za pierwszym razem wyświetlany jest komunikat o błędzie, szczególnie przy zbyt szybkiej próbie zatwierdzenia zmian. Chociaż system kosztorysów okazał być tylko pomocniczy w procesie wyceny, bardzo dużą pomocą okazała się pozostała część systemu. Magazynowanie danych zamówienia oraz klientów i adnotacji, połączone z możliwością drukowania usystematyzowały oraz znacząco przyspieszyły i ułatwiły pracę w firmie.

Rozdział 9. Spis rysunków

Zastosowano numerację podwójną, pierwszy cyfra oznacza numer rozdziału.

Rozdział 10. Bibliografia

10.1. Wykaz literatury

1. Z. Fryźlewicz, E. Bukowska, D. Nikończuk, ASP.NET MVC 4. Programowanie aplikacji webowych, Gliwice 2013,
2. J. Matulewski, M. Grabek, M. Pakulski, D. Borycki, ASP.NET Web Forms. Kompletny przewodnik dla programistów interaktywnych aplikacji internetowych w Visual Studio, Gliwice 2014,
3. K. Żydzik, T. Rak, C# 6.0 i MVC 5. Tworzenie nowoczesnych portali internetowych, Gliwice 2015.

10.2. Wykaz stron internetowych

1. O Material design, https://materializecss.com/getting-started.html (data odczytu 01.03.2021),
2. O technologii .NET https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet (data odczytu 01.03.2021),
3. O ASP .NET https://dotnet.microsoft.com/apps/aspnet (data odczytu 01.03.2021),
4. O WPF https://docs.microsoft.com/en-us/visualstudio/designers/getting-started-with-wpf?view=vs-2019 (data odczytu 01.03.2021),
5. Język C# https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/ https://docs.microsoft.com/pl-pl/dotnet/csharp/whats-new/csharp-version-history (data odczytu 01.03.2021),
6. 6. O Entity Framework Core https://docs.microsoft.com/en-us/ef/ (data odczytu 01.03.2021).