Polsko-Japońska Wyższa Szkoła Technik Komputerowych

INŻYNIERIA OPROGRAMOWANIA

Inżynieria Oprogramowania i Baz Danych

Piotr Skołysz 4171
Piotr Burczyński 2649

VisMode - Wizualne modelowanie aplikacji biznesowych

Praca magisterska
Napisana pod kierunkiem
Dr. inż. Mariusza Trzaski

Warszawa, Czerwiec 2009

Streszczenie

Celem naszej pracy jest opracowanie zasad dotyczących wizualnego modelowania aplikacji biznesowych bez pisania kodu źródłowego. W tym celu przygotowaliśmy prototyp takiego oprogramowania. Mamy tu na myśli taki program, za pomocą którego przy niewielkiej znajomości baz danych i znikomej wiedzy programistycznej, będzie można stworzyć aplikacje biznesową (np. systemu wypożyczania książek). Od użytkownika końcowego będzie wymaga głównie wiedza na temat struktury firmy.

Istotą pracy jest zidentyfikowanie czy istnieje możliwość stworzenia takiego programu, który nie będzie wymagał od użytkownika pisania kodu programistycznego, a zarazem będzie na tyle funkcjonalny by pozwolić na swobodę w tworzeniu aplikacji.

1. Wstęp

1.1. Języki programowania.

Wydarzenie z 1623 roku, gdy Wilhelm Schickard, profesor matematyki i astronomii z Uniwersytetu w Tybindze, w liście do astronoma Johanna Keplera załącza rysunek zegara, który może wykonać 4 operacje arytmetyczne i wyciągnąć pierwiastek kwadratowy – można uznać za początek informatyki. Kiedy jednak powstał pierwszy język programowania ? Wiele źródeł jak Internet czy encyklopedie podają, iż pierwszym dużym, automatycznym komputerem cyfrowym był Harvard Mark 1, zaprojektowany w Ameryce przez zespół Howarda H. Aikena w latach 1939 – 1944. Jednak po zakończeniu II Wojny Światowej odkryto, że to komputer Z3 z 1941 roku skonstruowany przez niemieckiego konstruktora Konrada Zuse był pierwszym komputerem sterowanym przez program. W 1940 roku Konrad Zuse przedstawił wizję, według której komputery miały uwolnić ludzi od wykonywania głupich obliczeń. 1 kwietnia tego samego roku w Berlinie założył firmę o nazwie Zuse Apparatebau by budować komputery. Między latami 1942 a 1945 firma Zuse'a skonstruowała komputer Z4, od 1945 był on produkowany z modułem Planfertigungsteil (moduł przygotowania planu), który służył do tworzenia w prosty sposób taśm perforowanych zawierające instrukcje dla komputera. Dzięki temu modułowi powstał pierwszy asembler – język programowania niskiego poziomu, który uwolnił informatyków od zapisywania treści programu przy pomocy 0 i 1 i pozwolił w sposób zrozumiały dla człowieka na wprowadzanie oraz odczyt rozkazów i adresów. W latach 1954 – 1957 pracownik IBM John Backus wraz z zespołem stworzył pierwszy kompilator Fortranu (ang. FORmula TRANslator). Kompilator ten był pierwszym w historii kompilatorem języka wysokiego poziomu. To znacznie ułatwiło rozumienie kodu, gdyż jego większość stanowią tak naprawdę słowa np. w języku angielskim. Zwiększyło to tym samym poziom abstrakcji i dystans do sprzętowych niuansów. Przez następne lata dziedzina informatyki prężnie się rozwijała, aż w 1967 roku przedstawiono język przeznaczony do programowania symulacji komputerowych – można przyjąć, iż był to pierwszy język posiadający cechy obiektowości. Dało to początek rozwoju takich języków jak C++, Java, C#, Eiffel, Pyton, jak i wielu innych. Pojawiły się wielkie możliwości w programowaniu.

1.2. „Kryzys oprogramowania”, a inżynieria oprogramowania.

W latach pięćdziesiątych i na początku lat sześćdziesiątych tworzono wyłącznie małe programy. Wynikało to głównie z niewielkich możliwości ówczesnych komputerów, jak i z braku zapotrzebowania na złożone oprogramowanie. Tworzono programy głownie dla celów naukowych, wymagania były więc dość dobrze, często formalnie zdefiniowane. Co więcej, oprogramowanie było z reguły tworzone przez przyszłych użytkowników dla własnych potrzeb lub w ścisłej współpracy z użytkownikiem.

Sytuacja zmieniła się w drugiej połowie lat sześćdziesiątych. Rozwój sprzętu komputerowego oraz języków oprogramowania umożliwił tworzenie znacznie bardziej złożonych systemów. Pojawili się także pierwsi programiści, tj. ludzie zawodowo zajmujący się wytwarzaniem oprogramowania. W tym okresie uświadomiono sobie również przydatność komputerów w nowych zastosowaniach, między innymi w zarządzaniu. Podjęto szereg prób budowy złożonych systemów informatycznych, których realizacja wymagała współpracy wielu osób. Wiele z tych przedsięwzięć nie zostało nigdy ukończonych, pozostałe znacznie przekroczyły założony czas i budżet. Stało się jasne, że rozwój techniki budowy oprogramowania nie nadąża za rozwojem sprzętu. To właśnie zjawisko nazwano „kryzysem oprogramowania”.

Zjawisko to trwa zresztą do dziś. Ilustracją mogą być wyniki ankiety przeprowadzonej przez van Genuchtena (1991). W latach dziewięćdziesiątych 90% poważnych firm programistycznych w USA uważało, że często zdarzają się im opóźnienia w realizacji przedsięwzięć programistycznych. Nasze obserwacje potwierdzają, że sytuacja w Polsce nie jest z pewnością pod tym względem lepsza. Oprogramowanie jest prawdopodobnie jedynym produktem technicznym, w którym błędy są powszechnie akceptowane.

Od połowy lat sześćdziesiątych do połowy lat osiemdziesiątych nie nastąpił praktycznie żaden wzrost wydajności programistów. Należy dodać, że są to dane dotyczące wyłącznie poważnych firm programistycznych, nie biorą one więc pod uwagę wpływu tzw. „rewolucji mikrokomputerowej”. Trudno byłoby chyba znaleźć inną dziedzinę techniki, w której w tym okresie nastąpił wyraźny wzrost efektywności.

Jakie są przyczyny „kryzysu oprogramowania” ? Liczne konferencje i dyskusje na ten temat, które odbywały się w połowie lat sześćdziesiątych, zaowocowały wskazaniem następujących jego przyczyn:

- ✓ Duża złożoność systemów informatycznych.
- ✓ Niepowtarzalność poszczególnych przedsięwzięć.
- ✓ Nieprzejrzystość procesu budowy oprogramowania, tj. fakt trudności w ocenie stopnia zaawansowania prac. Niewątpliwie najgorszym sposobem oceny postępów jest pytanie się programistów o ocenę stopnia zaawansowania.
- ✓ Pozorna łatwość wytwarzania i dokonywania poprawek w oprogramowaniu. Narzędzia pozwalające tworzyć nawet całkiem duże programy są stosunkowo tanie. Każdy, kto korzystając z takich narzędzi w ciągu jednego dnia napisał, uruchomił i przetestował program liczący 100 linii kodu, może sądzić, że w ciągu 10 dni opracuje program liczący 1000 linii, w ciągu 100 dni program liczący 10.000 linii, a dziesięć takich osób w ciągu stu dni opracuje program liczący 100.000 linii. Tę przyczynę „kryzysu oprogramowania" ciekawie analizuje Baber (1989)

Rozmaite propozycje wyjścia z „kryzysu oprogramowania" zaowocowały powstaniem nowego działu informatyki – inżynierii oprogramowania.

1.3. Cel Projektu

Ówczesne języki programowania, a zwłaszcza narzędzia coraz bardziej ograniczają potrzebę pisania kodu. Doskonałym przykładem takiego narzędzia jest np. Visual Studio, w którym w bardzo prosty sposób możemy zaprojektować GUI własnego programu nie pisząc przy tym linijki kodu. Takie narzędzia nie tylko przyspieszają czas pisanych programów, lecz również pozwalają się skupić na samej istocie programu, a nie programowaniu jego wyglądu.

Dzięki takim narzędziom zaczęły powstawać coraz bardziej wydajne i potrzebne programy, które w dniu dzisiejszym są nieodzowną częścią większych przedsiębiorstw. Wyobraźmy sobie jakąkolwiek dużą firmę działającą bez programu do obiegu dokumentów, bądź nawet zwykłej bazy danych, w której przechowywane są dane kontrahentów. W dniu dzisiejszym wydaje się to niemożliwe. Brak takiego produktu wiąże się ze złą organizacją pracy, czyli stratą czasu, a co za tym idzie – stratą pieniędzy.

Podążając dalej tym tropem pragnęliśmy stworzyć prototyp programu, który do minimum ograniczy pisanie kodu, bądź je całkowicie wyeliminuje. W prosty i szybki sposób

będzie można stworzyć aplikację przykładowo do wystawiania faktur powiązaną z magazynem. Użytkownikiem końcowy będzie mógł być praktycznie każdy pracownik danej firmy, niekoniecznie programista. Wystarczyłaby do tego nawet niewielka wiedza na temat baz danych oraz struktury działania firmy, a nie wiedza programistyczna. Chcemy udowodnić iż przyszłością programowania może być „programowanie wizualne" – bez pisania kodu programistycznego.

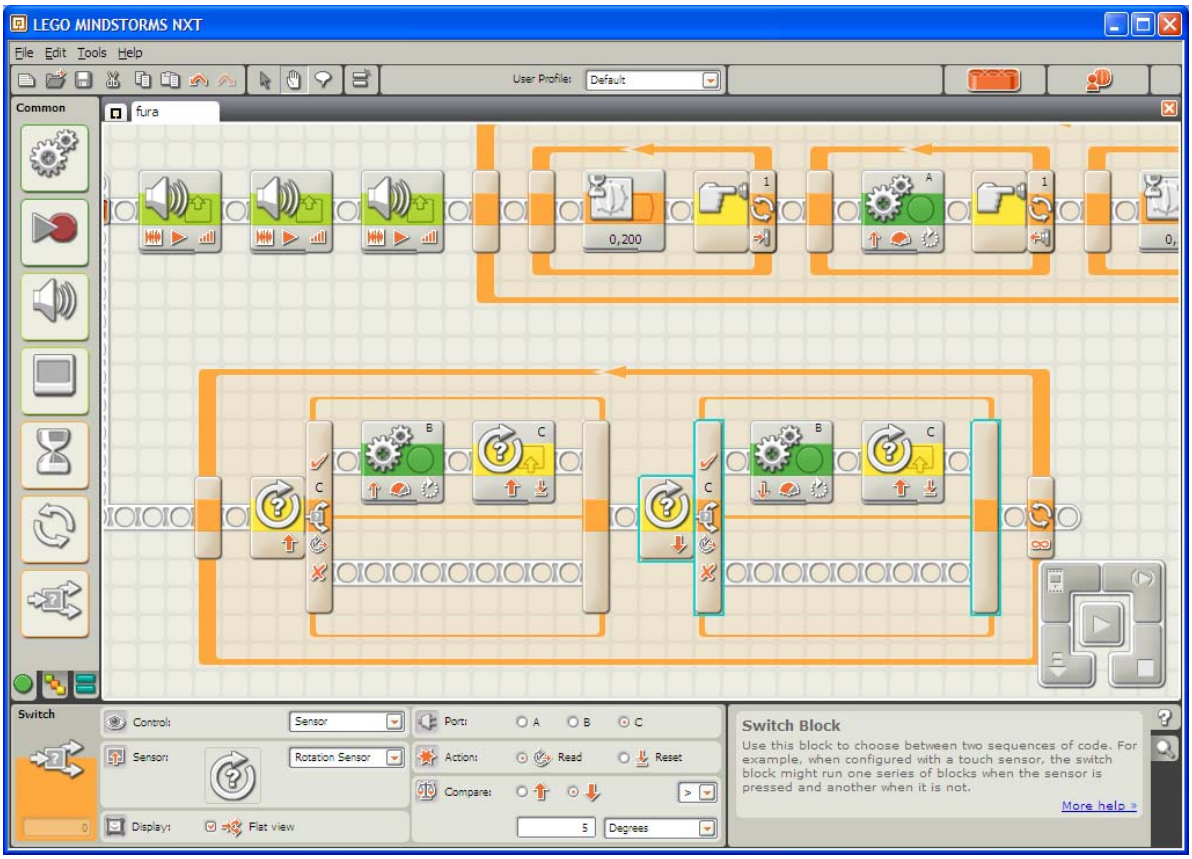
2. Koncepcja

Każdy programista rozpoczynając pracę nad jakąkolwiek aplikacją zaczyna od przemyślenia funkcji, które program powinien spełniać. Co więcej, musi wziąć również pod uwagę względy estetyczne, funkcjonalne oraz ergonomiczne. Na tym etapie powinien umieć przewidzieć wszystkie problemy, z którymi może się spotkać w późniejszym programowaniu, zachowaniu użytkownika, czy też bezpieczeństwie aplikacji. Jest to kluczowy etap pracy decydujący o pomyślności danego projektu.

2.1. Koncepcja pierwotna

Naszym pomysłem było stworzenie programu, który pozwalałby na generowanie w pełni funkcjonalnych aplikacji na potrzeby małych i średnich przedsiębiorstw. Główne założenie było takie, aby użytkownik nie musiał posiadać wiedzy na temat żadnego języka programowania. Jedyną wiedzą jaką powinien posiadać to wiedza o strukturze i sposobie działania firmy, w której pracuje. Dzięki technice "przeciągnij i upuść" (Drag&Drop) mógłby w prosty sposób tworzyć aplikacje dla swojego przedsiębiorstwa. Z drugiej strony program powinien być na tyle funkcjonalny, aby pozwał na stworzenie dokładnie tego wszystkiego co można by stworzyć za pomocą pisania kodu programistycznego. Naszą inspirację zaczerpnęliśmy z istniejącego programu przeznaczonego do sterowania maszynami zbudowanymi z klocków LEGO – Mindstorms.

Nasza pierwotna koncepcja polegała na tym, aby użytkownik tworzenie aplikacji rozpoczął od zaprojektowania formularzy wraz z ich elementami (kontrolkami). Wówczas przyjęliśmy analogiczne rozwiązania jak w przypadku narzędzia Visual Studio. Kolejnym krokiem miałoby być ustalenie zachowywania się kontrolek na danym formularzu wykorzystując nasz graficzny edytor kodu. Jego głównym założeniem było wizualne projektowanie logiki posługując się symbolami. Każdemu elementowi logicznemu (np. if, case, for bądź zmienne) odpowiadałaby stosowna ikona. Cały proces polegałby na przeciąganiu z paska narzędzi elementów na okno projektu, gdzie zaprezentowanoby w postaci symbolu ich funkcjonalność. Korzystając z okna właściwości można by wybrać opcje lub ustawienia, które spełniałyby nasze wymagania. W taki sam sposób mielibyśmy stworzyć interakcje miedzy poszczególnym formularzami. Rysunek 2.1 przedstawia przykład rozwiązana w programie LEGO Mindstorms.



Rysunek 2.1. Okno projektu w programie LEGO Mindstorms.

Po pełnym przeanalizowaniu naszego pomysłu doszliśmy do wniosku, iż budowa prototypu aplikacji tego typu mija się z celem, ponieważ:

- ✓ nie jesteśmy w stanie przewidzieć wszystkich zależność miedzy poszczególnymi funkcjami
- ✓ stworzony schemat graficzny mógłby być zbyt rozbudowany by go łatwo zrozumieć
- ✓ mogłaby pojawić się trudność w odnalezieniu elementów graficznych, które odpowiadałyby za wykonywanie danego zadania, jak i w wykrywaniu błędów
- ✓ potrzebne byłyby duże wymagania sprzętowe do obsługi rozbudowanych diagramów

2.2. Koncepcja ostateczna

W finalnej wersji naszej pracy nie chcieliśmy zasadniczo odbiegać od pierwotnie przyjętych założeń. Niemniej wraz z rozwojem projektu koncepcyjnego musieliśmy poddać go weryfikacji. Stwierdziliśmy, że stworzona przez nas aplikacja powinna być bardziej "szyta na miarę" – czyli przygotowana dla konkretnych rozwiązań. W związku z tym skoncentrowaliśmy się na stworzeniu takiego programu, który będzie odpowiadał nowo przyjętym standardom.

Głównym przeznaczeniem tego programu jest tworzenie aplikacji dla małych i średnich przedsiębiorstw. Większość tego rodzaju firm korzysta z aplikacji do przetrzymywania danych. Są to głównie pliki programu Excel, bądź proste bazy danych.

Idąc tym tropem skupiliśmy się na stworzeniu takiego rozwiązania, które w prosty sposób pozwoli opracować narzędzie do przetwarzania oraz zapamiętywania danych. Technika Drag&Drop miałaby ułatwić proces budowy, tak samo jak automatyzacja procesów pośrednich czyli na przykład automatyczne generowanie formularzy do edycji danych.

2.3. Podsumowanie

Polityką firm i przedsiębiorstw jest redukcja kosztów przy zachowaniu standardów produkcji. Z reguły akceptacja nowych kosztów wynika z przyjętej strategii firmy mającej na celu z jednej strony zintensyfikować produkcję a z drugiej systematycznie ją unowocześniać. W konsekwencji zainwestowane środki zwracają się po określonym czasie. Tym kosztem jest na ogół zakup oprogramowania przyspieszającego pracę lub zatrudnienie wykwalifikowanej osoby, która przygotuje nam takie rozwiązanie.

Naszym pomysłem stało się przygotowanie takiego programu, który w szybki i łatwy sposób pozwoli niewykwalifikowanej osobie opracować takie rozwiązanie. Z racji tego, iż docelowymi odbiorcami mają być małe i średnie przedsiębiorstwa, skupiliśmy swoją uwagę na ich zapotrzebowaniach na oprogramowanie.

Kierując się zebranymi informacjami i inspiracją programu Lego Mindstorms przyjęliśmy następującą koncepcję. Program musi pozwalać w szybki i sprawny sposób stworzyć aplikacje do zarządzania danymi. Jego prostota i ergonomia ma wykreować aplikację bez napisania linii kodu programistycznego. Oczywiście wiąże się to z utratą większości funkcjonalności, wiec daliśmy możliwość bardziej zaawansowanym

użytkownikom na ingerowanie w kod aplikacji, eksport do Visual Studio, tworzenie własnych wtyczek do programu projektowego

3. Stan sztuki

W dzisiejszych czasach wymaga się zwiększonych rezultatów przy mniejszym inwestowaniu. Wymaga się pracy, która może być mierzona przy pomocy ulepszonej efektywności, czasu i kosztów ogólnych jej wykonania. Spowodowało to pojawienie się na rynku wielu rozwiązań typu RAD (Rapid Application Development). Większość z nich niestety to produkty komercyjne. Służą głównie zarządzaniu, przechowywaniu i prezentacji danych. Wykorzystują one najpopularniejsze bazy danych, takie jak: MS SQL, Oracle czy MySQL. W tym rozdziale przedstawiamy przykładowe aplikacje.

3.1. Form Suite 4 .Net

Jest to narządzie pozwalające na tworzenie graficznych interfejsów pod platformę Windows jak i przeglądarek internetowych. Pozwala na budowę formularzy platformy .NET jak i Adobe Flash. Aplikacja potrafi konwertować obiekty Windows Forms na ASP.NET oraz automatycznie kompletować dane w komponentach Windows i WebLoader. Aplikacja posiada komponenty, które dostarczają wygodne metody do łatwego pozyskiwania danych. Używając komponentów formularzy bazodanowych, można je dynamicznie ładować i używać w obydwu środowiskach Windows i Web. Komponenty LoaderForm i LoaderControl mogą ładować i wyświetlać formularze oraz kontrolki, które były wcześniej stworzone przez użytkownika za pomocą aplikacji form.suite4.net. Funkcjonalność rozwiązań Web'owych jest osiągnięta przy wykorzystaniu technologii Adobe Flash. To pozwala na przeglądanie formularzy przy użyciu dowolnej przeglądarki internetowej i systemu operacyjnego, który ma zainstalowany Flash Player w wersji 7.0 lub wyższej.

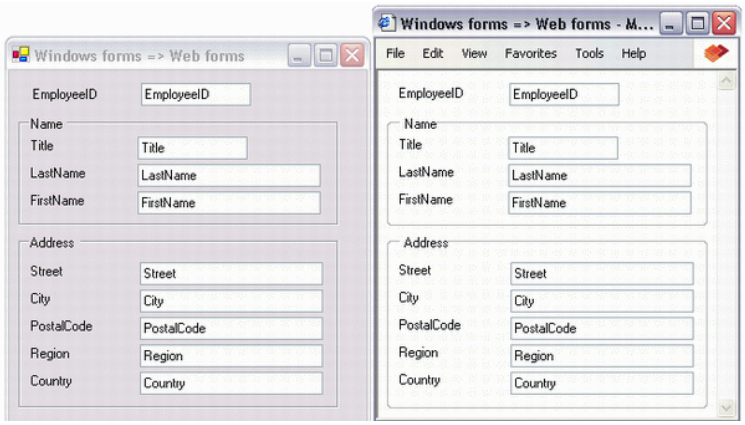
3.1.2 Projektowanie formularzy

Jedną z głównych cech form.suite4.net jest elastyczna architektura, która pozwala modyfikować formularze bez powtórnej kompilacji. Dzięki temu uzyskujemy narzędzia programistyczne, które umożliwiają dynamiczne ładowanie elementów projektu tylko poprzez czytanie składników pliku projektowego. W momencie wprowadzenia ostatecznych zmian w projekcie wystarczy umieścić zaktualizowany plik na serwerze. Każdym następnym razem zaktualizowane elementy zostaną wyświetlone w sposób w jaki użytkownik zdefiniował je w

aplikacji. Baza projektu zachowuje wszystkie kontrolki, obrazy i informacje o layout'cie przynależnym do konkretnego projektu. Wszystkie zmiany są zachowywane w jednym dokumencie co pozwala na ich szybkie i bezproblemowe wdrożenie. Poniżej kilka cech tej aplikacji.

- **Konwertowanie Windows Forms do ASP.NET Web Forms**

Aplikacja umożliwia importowanie projektów Visual Studio napisanych w Visual Basic lub C# i konwertowanie zawartych w nich formularzy do ASP.NET. Importowanie projektów Form Suite 4 .Net przedstawione jest na rysunku 3.1:



Rysunek 3.1. Importowanie projektów Form Suite 4 .Net

Konwerter ASP.NET wspiera następujące kontrolki Windows Forms:

- ✓ Button
- ✓ CheckBox
- ✓ ComboBox
- ✓ GroupBox
- ✓ Label
- ✓ LinkLabel
- ✓ Panel
- ✓ PictureBox
- ✓ RadioButton
- ✓ TabControl i TabPages
- ✓ TextBox

Jeżeli używamy innych kontrolek w swoim formularzu aplikacja pomaga zamienić każdą z nich w kontrolkę wspieraną przez form.suite4.net. W momencie kiedy pliki zostaną zapisane

we wskazanym folderze można je importować do istniejącego projektu ASP.NET jak również stworzyć nowy projekt używając Internet Information Services (IIS) lub Visual Studio.NET. Można je wówczas kompilować, przeglądać oraz wprowadzać potrzebne zmiany do rozszerzeń aspx i ascx.

- **Databinding z Webloader, LoaderControl i LoaderForm**

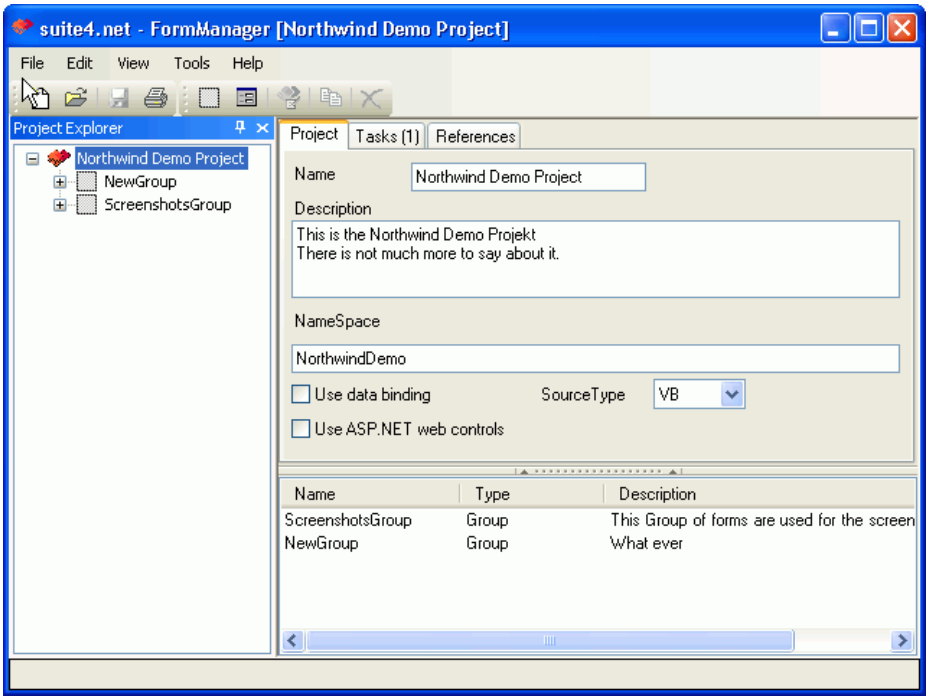
„DataBinding to potężny mechanizm wprowadzony w .NET framework, który umożliwia łączenie elementów bezpośrednio z danymi z bazy danych. Aplikacja pomaga przy łączeniu właściwości obiektów biznesowych z kontrolkami TextBox, NumericUpDown i innymi. [16]" Automatyczny mechanizm databinding jest wspierany przez Windows i komponenty WebLoader. Implementacja wymaga napisania nie więcej niż jednej linii kodu. W oknach LoaderControl i LoaderForm mamy zaimplementowane dwie techniki:

- ✓ Ręczne przypisanie wartości właściwości obiektów biznesowych do kontrolek i przypisanie ich z powrotem obiektom biznesowym.
- ✓ Użycie wbudowanych w Windows forms właściwości databinding.

Można powiązać obiekty biznesowe w stworzonej przez użytkownika aplikacji z formularzem projektu form.suit4.net. Kiedy wygeneruje się kod dla formularza, form.suite4.net zawiera deklaracje kodu oparte na wybranej technice databinding.

- **FormManager – detale projektu**

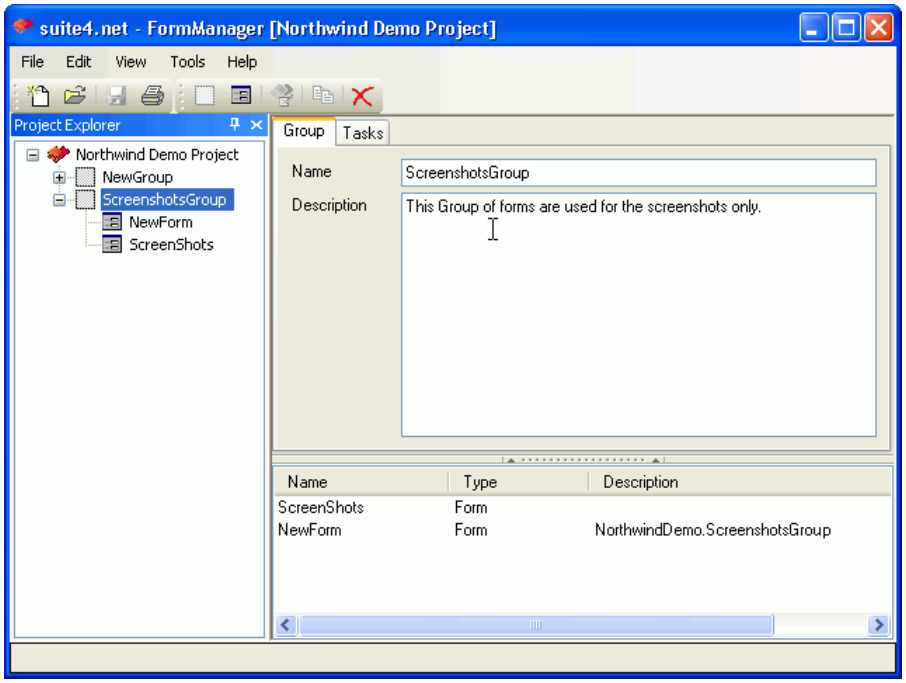
Form Manager jest głównym mechanizmem do zarządzania projektem. Można tutaj ustawić globalne opcje projektu, zarządzać jego elementami tworząc logiczne grupy, dodawać formularze do tych grup i kontrolki do formularzy, jak i przypisywać klasy biznesowe do formularzy. Szczegółowe informacje elementów z 'Project Explorer' wyświetlane są z prawej strony (rys. 3.2).



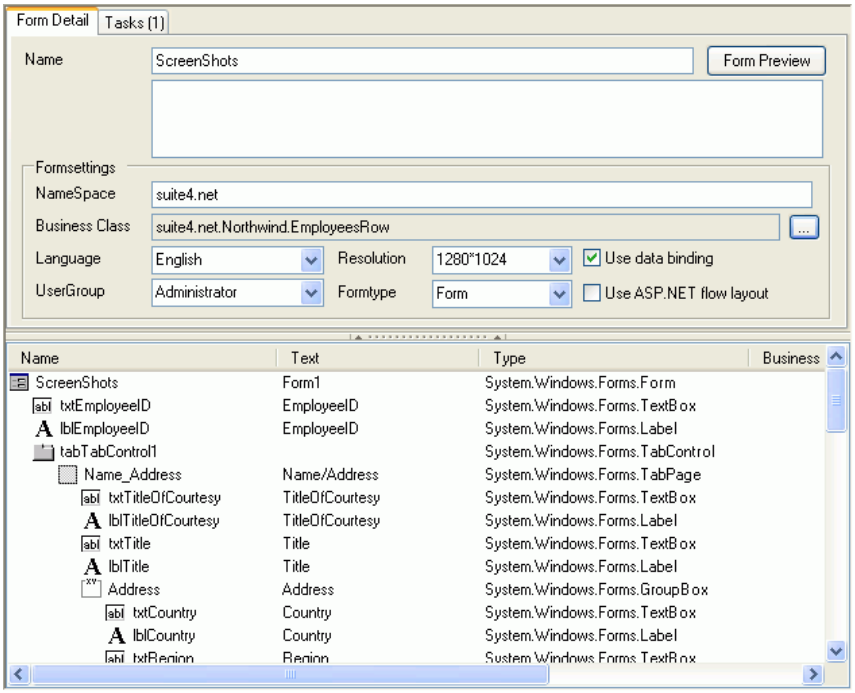
Rysunek 3.2. Okno FormManager – szczegółowe informacje projektu

Główne cechy FormManagera to :

- ✓ Umożliwia generowanie layout'ów dla Windows i Web Forms.
- ✓ Zarządza całym projektem. Można szybko przeglądać referencje, kontrolki, zadania i inne detale projektu, jego grup i formularzy.
- ✓ Można tworzyć bazowe formularze po których inne formularze będą dziedziczyły kontrolki i pozostałe właściwości.
- ✓ Project Explorer daje możliwość przeglądania wszystkich formularzy w projekcie.
- ✓ Form Detail (bardziej uszczegółowiony Design Explorer) umożliwia kopiowanie kontrolek oraz pojemników do innych formularzy przy pomocy 'przeciągnij i upuść'(Drag and Drop).
- ✓ Można załączać zadania i/lub opisy projektu, jego grup i formularzy. Pewne zadania są dodawane do systemu automatycznie.
- ✓ Pozwala importować i eksportować formularze z i do innych projektów oraz Visual Studio.
- ✓ W momencie eksportowania mamy możliwość zapisania pliku źródłowego w C# lub VB.
- ✓ W Project Explorer można kopiować i przenosić formularze do innych grup przy pomoc 'Drag and Drop'.

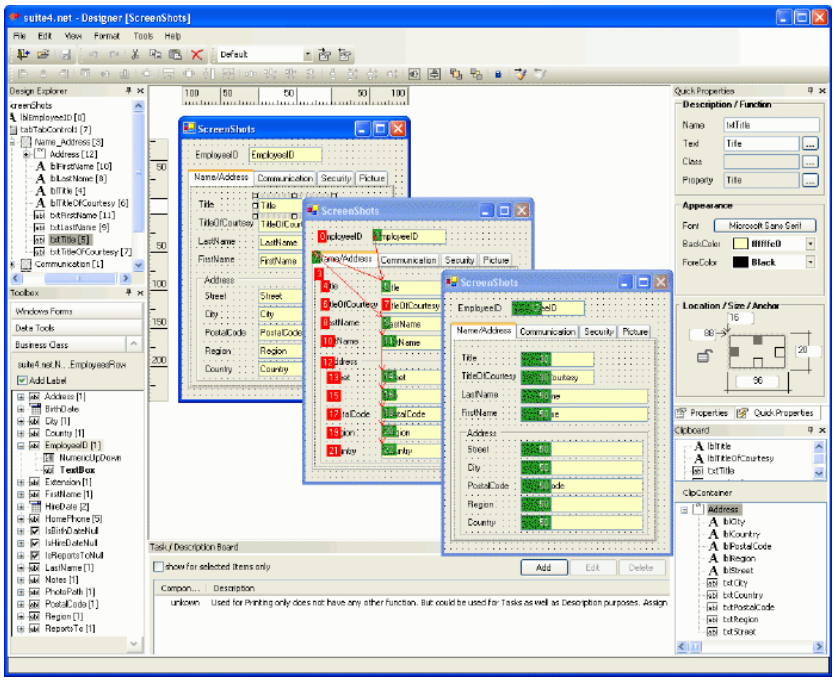


Rysunek 3.3. Okno FormManager – szczegółowe informacje grupy



Rysunek 3.4. Okno FormManager – szczegółowe informacje grupy

- **FormDesigner**



Rysunek 3.5. Wygląd okna FormDesigner

Główne cechy FormDesigner to:

- ✓ Współpracuje z klasami biznesowymi (Databinding) tak samo jak z kontrolkami czy pojemnikami niezależnie od standardu.
- ✓ Design Explorer umożliwia szybki przegląd kontrolek użytych w formularzu.
- ✓ Clipcontainer jest polepszonym schowkiem pozwalającym na kopiowanie kontrolek i pojemników przy pomocy ‘Drag and Drop’.
- ✓ Task/Description Board pozwala nadawać kontrolkom, pojemnikom i formularzom zadania oraz opisy.
- ✓ QuickProperties umożliwia szybkie i skuteczne edytowanie powszechnie używanych właściwości kontrolek, pojemników i formularzy.
- ✓ Tabindex Editor wyświetla graficznie sekwencje zakładek w formularzu i pozwala na ich zmianę automatycznie lub ręcznie przy pomocy ‘Drag and Drop’.
- ✓ Generowanie i formatowanie kodu dla własnych potrzeb jest możliwe przy pomocy ustawienia opcji takich jak data binding , inicjalizacja czy po prostu przeglądanie kodu źródłowego.
- ✓ Przy pomocy Limit Editor można szybko przeglądać i zmieniać wielkości wejściowe dla użytkownika (dla takich kontrolek jak TextBox i NumericUpDown).

- ✓ Pozwala używać arkuszy stylów do zarządzania wyglądem.

3.2. NConstruct

NConstruct jest szybkim narzędziem i środowiskiem deweloperskim. „Jego celem jest zredukowanie czasu rozwoju najczęściej używanych aplikacji przez przedsiębiorstwa, opierając się na przetrzymywaniu i prezentacji danych. [15]" Zasadniczo odbywa się to w dwóch wymiarach. Pierwszy przedstawia NConstruct Builder, kreator umożliwiający tworzenie aplikacji typu Klient – Serwer do zarządzania i prezentowania danych z bazy w bardzo krótkim czasie. Tak wygenerowane aplikacje tworzą drugi wymiar NConstruct, czyli NConstruct System. Składa się on z wzajemnie powiązanych aplikacji serwera i klienta, co daje rozszerzony zestaw automatycznie wspomaganych funkcji jak ochrona, zabezpieczanie, prezentacja i edytowanie danych. Przestrzegając zasad deweloperskich NConstruct, programista może korzystać z tych funkcji na swoich własnych obiektach i może całkowicie się skupić na implementacji logiki biznesowej. NConstruct System staje się punktem integracyjnym różnych już rozszerzonych funkcji, które mogą zostać dodane do niego jako moduły. Poniżej opisujemy najważniejsze z nich:

- **Input i Output**

NConstruct Builder może być uważane za urządzenie, które bierze relacyjną bazę danych jako dane wejściowe i tworzy kod źródłowy do aplikacji, który zarządza danymi. Następnie kompiluje ten kod i tworzy ostatecznie w pełni funkcjonalną aplikację. Wszystko co zostaje stworzone w NConstruct Builder może być modyfikowane przez deweloperów i dopasowywane do ich potrzeb.

- **Aplikacja klient – serwer**

Dane wyjściowe stworzone przez NConstruct Builder składają się z aplikacji klienta i serwera. To umożliwia instalację różnych klientów dla różnych użytkowników podłączonych do jednego serwera. Używając szybkiego połączenia internetowego oraz otwarcia odpowiednich portów po stronie serwera, klient może w rzeczywistości być umieszczony w odległej lokalizacji na całym świecie. Komunikacja pomiędzy serwerem i klientem jest zabezpieczona przez specjalny proces identyfikacji na żądanie. To oznacza, że każde żądanie jest wyposażone w zakodowaną identyfikację klienta, co uniemożliwia pozostałym

komunikowanie się z serwerem. Dodatkowo „komunikacja pomiędzy klientem i serwerem przez internet może być całkowicie zabezpieczona na przykład przy użyciu specjalnego ‘tunneling’ jak prywatne sieci wirtualne (Virtual Private Networks) [15]”.

- **Dostęp do danych**

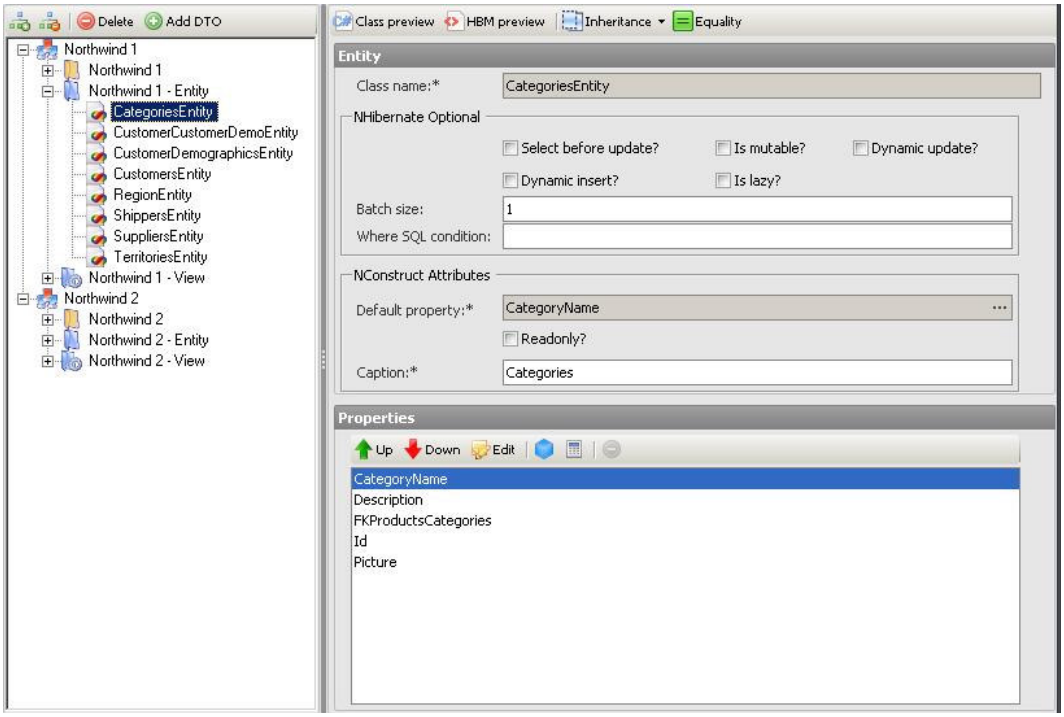
NConstruct ma dostęp do danych w bazie poprzez mapowanie obiektów przedstawiających rejestry w tabelach bazodanowych, używając NHibernate jako oprogramowania mapującego. Nie tylko używa NHibernate, ale również przygotowuje do tego całe pliki mapujące, w zależności od tabeli znajduje konkretną bazę danych i decyzje podjęte przez dewelopera podczas definiowania parametrów kodu generującego. Obecnie wpierane bazy danych to: Microsoft AQL Server, Microsoft AQL Server 2005 i Oracle 10g.

- **Dodatki NConstruct**

Poza integracją mapowania danych NConstruct oferuje także implementację interfejsu graficznego oraz mechanizmów bezpieczeństwa. Wszystkie dane z tabel umieszczonych w bazie danych są wyświetlane w „Gridach”. Dodawać można specjalne filtry do przenoszenia danych z serwera do klienta, eliminuje to niekończące oczekiwanie na nie w przypadku wielkich tabeli. Obiekty przedstawiające zmapowane dane z bazy są zaopatrzone w specjalne atrybuty umożliwiające ich modyfikację na specjalnym formularzu edytującym NConstruct. Automatycznie wspierane zarządzanie umożliwia administratorowi aplikacji udzielanie użytkownikom praw dostępu do aplikacji i poszczególnych jej danych. Dostęp może być udzielony do przeglądania, edytowania i usuwania danych. Ponieważ różni użytkownicy mają dostęp do tych samych danych NConstruct wprowadza mechanizmy blokujące.

- **Encje**

Są to zasadniczo obiekty mapowane z baz danych. Zwykle przedstawiają konkretne dane z tabeli bądź widoku. Są mocno spokrewnione z NHibernate, który rozpoznaje ich właściwości poprzez konkretne pliki mapujące (pliki hbm) generowane przez NConstruct.
Rysunek 3.6 przedstawia okno ustawień encji.

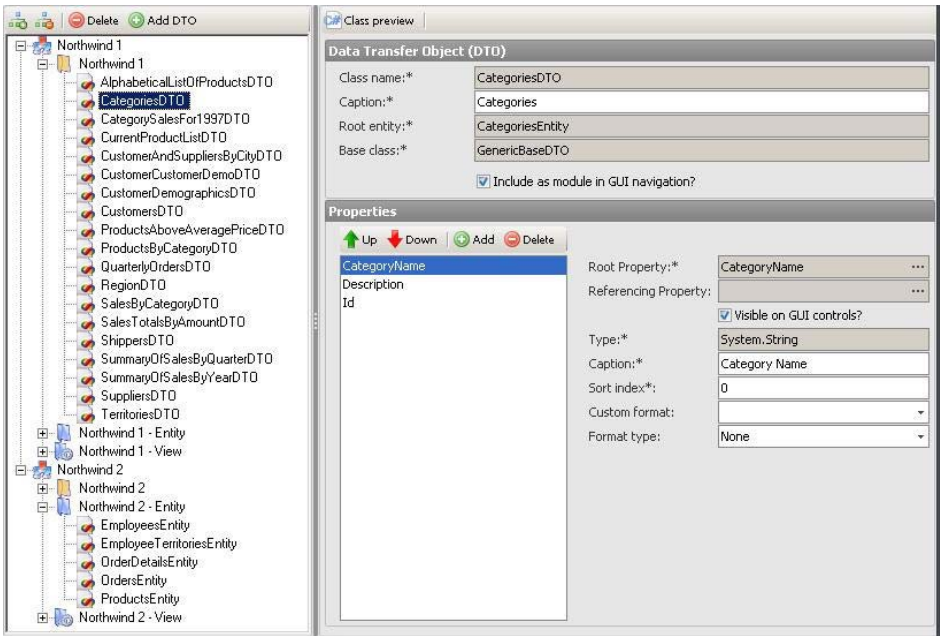


Rysunek 3.6. NConstruct – okno ustawień encji.

NHibernate przetrzymuje je w bazie danych, a za pomocą funkcji load i save odczytuje lub je zapisuje. Obiekty te są generowanie podczas procesu generowania kodu poprzez NConstruct Builder'a. Programiści w późniejszej fazie mogą rozszerzać ich funkcjonalność. Aby jednak zachować ich podstawowe funkcje niektóre z ich właściwości muszą pozostać niezmienione. Istotną sprawą jest by pamiętać, że obiekty te mogą być generowane jedynie dla tabeli i widoków posiadających klucz główny.

- **Data Transfer Objects**

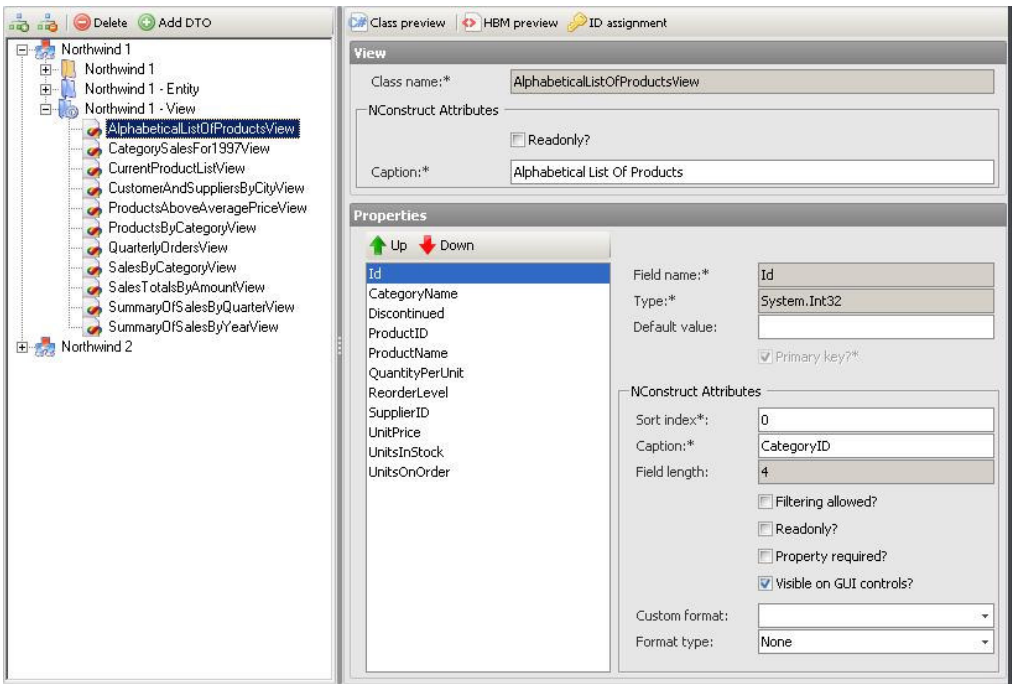
W celu zredukowania liczby danych przesyłanych przez sieć, obiekty bazodanowe (Data Transfer Objects - DTO) są przypisywane do encji i zawierają tylko te cechy, które są potrzebne do ich wyświetlania. Każda encja może mieć przypisanych kilka DTO.
Okno ustawień DTO przedstawia rysunek 3.7.



Rysunek 3.7. NConstruct – okno ustawień DTO

- **Widoki**

Widoki są obsługiwane bardzo podobnie jak tabele (rys. 3.8). W rzeczywistości encje bazujące na widokach nie różnią się od tych bazowanych na tabelach. Tak jak w przypadku tabel, encja może być stworzona tylko na widoku, który posiada klucz główny. Jeśli go nie posiada, encja taka nie będzie zdefiniowana podczas procesu generowania kodu.



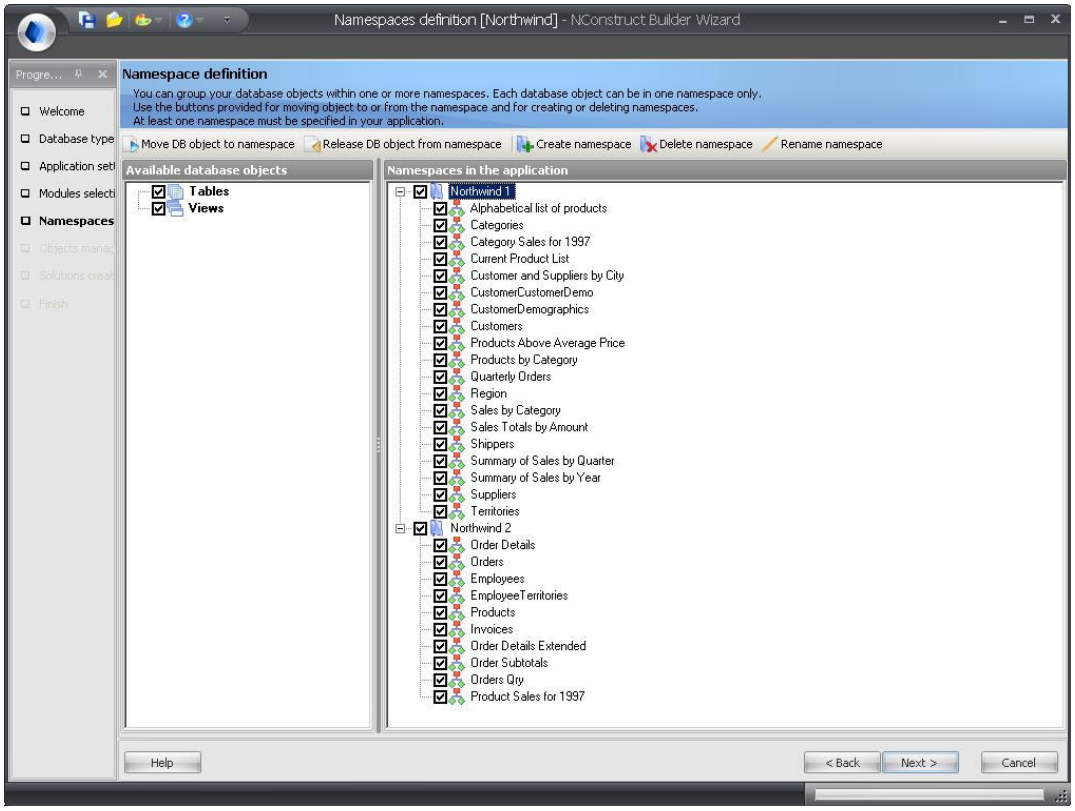
Rysunek 3.8. NConstruct – okno ustawień widoku (View)

- **Atrybuty**

Właściwości encji oraz DTO mogą być zaopatrzone w atrybuty, które posiadają specjalne znaczenie w środowisku NConstruct. Atrybuty te zawierają informacje, które mogą być wykorzystywane do wspierania automatycznego zarządzania danymi przy pomocy generycznych gridów i generycznych formularzy. Atrybuty encji i DTO są definiowane przez programistów w narzędziu NConstruct Builder'a i mogą być później zmieniane.

- **Przestrzenie nazw (Namespaces)**

Encje i DTO mogą być podzielone na kilka przestrzeni. Usprawnia to zarządzania kodem w przypadku dużej liczby tabeli. Podział na przestrzenie nazw może być również dokonany podczas procesu generowania kodu. Zwykle encje oraz (ich) DTO są grupowane w zależności od funkcji jakie pełnią w aplikacji (rys. 3.9).



Rysunek 3.9. NConstruct – przestrzenie nazw (namesapces)

- **Obiekty systemowe**

Obiektami systemowymi w NConstruct jest wszystko to do czego może być zastosowany system zabezpieczeń i blokad. Są dwa typy obiektów systemowych, którym udziela się pozwolenia na czytanie, pisanie i usuwanie oraz jeden typ, któremu udziela się pozwolenie

wykonywania (execute). Pamiętajmy, że pojedyncza jednostka, reprezentująca konkretny rekord bazy danych nie jest obiektem systemowy. Obiekt systemowym to raczej klasa przedstawiająca konkretne encje niż konkretne dane z bazy, więc gdy udziela się pozwolenia pewnym encjom w rzeczywistości udziela się go odpowiadającym im tabelom w bazie danych.

.

- **Table systemowe NConstruct**

Jak już zostało wspomniane NConstruct wspiera zarządzanie zabezpieczeniami oraz blokadami. Zabezpieczenia określają dostęp użytkownika do aplikacji i indywidualnych danych, podczas gdy blokowanie uniemożliwia utratę danych podczas gdy zarządza nimi kilku użytkowników. NConstruct przechowuje informacje o aktualnych ustawieniach bezpieczeństwa i aktualnych blokowaniach w tak zwanych tabelach systemowych (rys. 3.10), które są wygenerowane przez NConstruct Builder'a. Te tabele to:

Nazwa tabeli	Opis
ClientSesions	zawiera informacje o aktualnych sesjach klienta
SystemObjectsLocks	zawiera informacje o aktualnych obiektach
SystemObjectsLocksHistory	zawiera informacje o przekazanych blokowniach obiektów
SystemObjects	zawiera informacje o jednostkach i DTO. NConstruct Server jest zaprojektowany by robić update tej tabeli zawsze gdy rozpoczyna pracę
SystemPermissions	zawiera zgody – które z zadań ma mieć dostęp do obiektu i w jaki sposób
SystemRoles	zawiera definicje zadań, do których są przyporządkowani użytkownicy
SystemUsers	zawiera listę użytkowników mających dostęp do aplikacji
SystemUsersInRoles	definiuje, który użytkownik ma jakie zadanie by dać im dostęp do obiektu w zależności od ich zadań
UserSettings	zawiera informacje o konkretnych ustawieniach użytkowników

Rysunek 3.10. Tabele systemowe NConstruct

- **NConstruct Builder**

NConstruct Builder jak twierdzą jego twórcy [15] jest „esencją technologii NConstruct.” Odpowiedzialny jest za generowanie wszystkich klasy potrzebnych do prezentowania danych z bazy (encje i DTOs), jak również potrzebny dla NHibernate do mapowania danych z bazy do obiektów i odwrotnie. NConstruct Builder został zaprojektowany jako kreator. Każda podstrona zbiera konkretne informacje od programisty przed przejściem do następnej.

- **NConstruct Server**

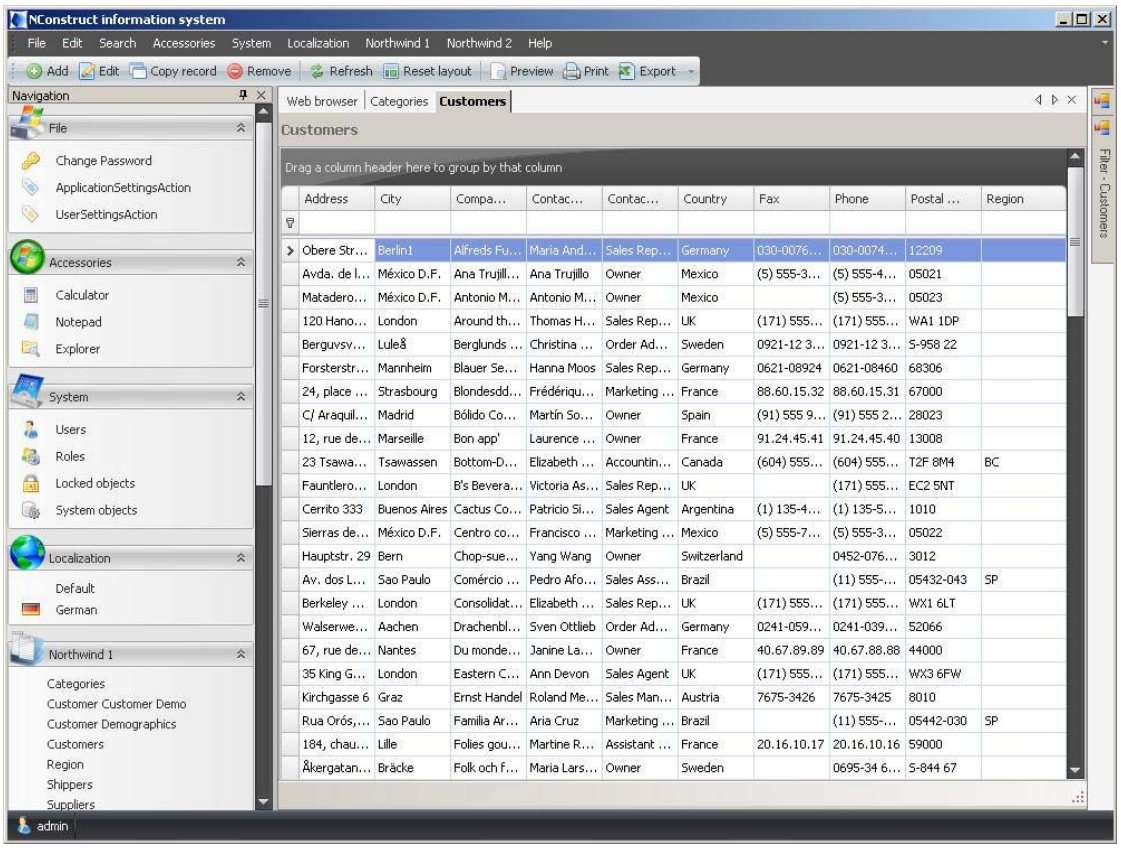
NConstruct Server pełni rolę serwera w architekturze klient-serwer. Umożliwia on podłączenie się kilku klientom jednocześnie jak i zapewnia system bezpieczeństwa oraz blokowania. Jest to również środowisko, w którym powinno się implementować większość logiki biznesowej, ponieważ dostęp do danych jest dużo szybszy na serwerze niż na kliencie. NConstruct Server definiuje dwie warstwy po stronie serwera:
- data layer
- services layer

Data layer składa się z bibliotek, które zawierają tylko klasy, których wyłączną funkcją jest przetrzymywanie danych. NConstruct rozpoznaje trzy typy klas danych – encje, widoki i DTO. Są zawsze przynajmniej dwie biblioteki Data layer. Pierwsza to biblioteka *NConstruct.Server.Data*, zawierająca klasy danych w odniesieniu do tabeli systemowych. Druga to biblioteka <ApplicationName>.Server.Data, zawierająca wszystkie klasy danych generowane z bazy danych użytkownika przez NConstruct Builder. Można zawsze dodać swoją własną bibliotekę, zawierającą własne klasy danych.

Services layer składa się z bibliotek, przetrzymujących logikę biznesową. Są tu również przynajmniej dwie biblioteki. Pierwsza to biblioteka *NConstruct.Server.Services*, zawierająca logikę biznesową NConstruct. Jest ona odpowiedzialna za zarządzanie mechanizmami bezpieczeństwa i blokowania oraz za dostęp i zarządzanie danymi w podległych bazach danych. Druga biblioteka to *<ApplicationName>.Server.Services*, która jest generowana przez NConstruct Builder. Jest to pusta biblioteka przygotowana dla użytkownika do implementacji logiki biznesowej ze strony serwera.

- **NConstruct Client**

NConstruct Client wspiera dwa typy kontrolek po stronie serwera – kontrolki Microsoft i Developer Express 6.3. Kontrolki Microsoft są wspierane przez wartości domyślne, podczas gdy kontrolki Developer Express muszą być dodatkowo wspierane przez dewelopera używając systemu NConstruct. Główny formularz NConstruct Klient przedstawiono na rysunku 3.11.



Rysunek 3.11.Główny formularz - NConstruct Client

- **NConstruc Web Client**

NConstruct Web Client nieznacznie różni się od Nconstruct Windows Client . Najważniejszą różnicą jest fakt, że Nconstruct Web Client nie wspiera właściwości administracyjnych. „Jest częścią rozwiązania Nconstruct Client i jest właściwie web serwerem, który odwołuje się do logiki serwera bardziej bezpośrednio niż na odległość. [15]”

3.3. Podsumowanie

Zasadniczą różnicą pomiędzy przedstawionymi tu aplikacjami, a naszym rozwiązaniem jest złożoność. Wszystkie powyżej opisane aplikacje są bardzo rozbudowane i posiadają bardzo dużą ilość różnych funkcji. Naszym głównym celem była prostota obsługi. Staraliśmy się, aby interfejs był w pewien sposób intuicyjny i było można zapoznać się z obsługą aplikacji w jak najkrótszym czasie.

Sprowadza się to do kilku prostych kroków jak np. tworzenie obiektów wraz z jego właściwościami, ustalenie powiązań miedzy nimi (obiektami), generowanie formularzy i ich ewentualna obróbka poprzez ustawienie znajdujących się na nich kontrolek.

Prostota obsługi zazwyczaj wiąże się z pewnym ograniczeniem funkcjonalności. Nasza aplikacja nie posiada aż tak wielu funkcji co wyżej przedstawione, udostępniliśmy jednak funkcję eksportu projektu do Visual Studio co przyczynia się do możliwości jej dalszej rozbudowy o dowolną funkcjonalność.

Dużym uproszczeniem dla użytkownika jest możliwość wykorzystania bazy jako plik w formacie XML. Dla większej ilości danych to rozwiązanie nie jest jednak najlepsze, aczkolwiek do takich zastosowań, do których ta konkretna aplikacja została przeznaczona, w zupełności wystarczy. Dzięki zastosowaniu takiego rozwiązania użytkownik nie musi posiadać nawet wiedzy o istnieniu czegoś takiego jak baza danych. Podsumowując moglibyśmy się pokusić o stwierdzenie, iż nauczenie się obsługi przedstawionych tu aplikacji nie jest w cale łatwiejsze niż nauczenie się programowania w takich językach jak Java czy C#. Co prawda mamy w nich zapewnione automatycznie generowanie formularzy lecz dzięki takim narzędziom jak Visual Studio czy JBuilder projektowanie interfejsów graficznych nie stwarza większych problemów nawet średnio zaawansowanym użytkownikom. Pod tym względem nasza aplikacja znacząco się wyróżnia.

4. Użyte technologie

4.1. Język C# i platforma .NET

Wprowadzenie platformy .NET w lipcu 2000 roku przez Microsoft było większą częścią drugiego sukcesu – zaprezentowania języka C# 2.0**.** Jest to w pełni dojrzały język programowania, zbudowany na doświadczeniach ostatnich 30 lat. C# jest obiektowy, nowoczesny, a zarazem prosty i bezpieczny. Został przystosowany zarówno do tworzenia programów na platformę .NET jak i do tworzenia aplikacji internetowych. Tak jak geny podobieństwa przekazywane miedzy pokoleniami, tak i tutaj możemy dostrzec wpływ innych języków jak Java, C++ czy Visual Basic. Wykorzystano również doświadczenia związane z wcześniejszą wersją tego języka.

4.1.1. Platforma .NET

Platforma .NET to nowe API (ang. Application Programming Interface) – środowisko udostępniające nowy interfejs tworzenia aplikacji. Jest to nowy zbiór funkcji i nowe narzędzia do tworzenia aplikacji ułatwiających korzystanie z serwisów WWW oraz klasycznych API dla systemów operacyjnych z rodziny Windows. Platforma .NET udostępnia również technologie korporacji Microsoft z późnych lat dziewięćdziesiątych. Przykładowo jest to obsługa komponentów COM+, formatu XML, obiektowości, nowych protokołów przesyłania danych, jak SOAP (ang. Simple Object Access Protocol), WSDL (ang. Web Service Definition Language) i UDDI (ang. Universal Description, Discovery and Integration), a także koncentracja na zastosowaniach internetowych. Wszystkie te technologie zostały zintegrowane w architekturze DNA (ang. Distributed interNet Applictaions).

Na rozwój platformy .NET i związanych z nią technologii firma Microsoft poświęciła bardzo wiele środków. Skutki tego zaangażowania robią duże wrażenie, a zakres zastosowań platformy .NET jest olbrzymi. Są to trzy grupy produktów:

- Grupa języków programowania, w skład której wchodzą między innymi C# i VB, zestaw narzędzi programistycznych na czele z Visual Studio .NET, bogata biblioteka klas umożliwiająca tworzenie serwisów oraz aplikacji internetowych i okienkowych, a

- Dwie generacje serwerów .NET Enterprise – tych obecnych już na rynku, a także tych, które dopiero się pojawią.
- Nowe urządzenie przystosowane do platformy .NET, między innymi telefony komórkowe oraz konsole do gier.

4.1.2. .NET Framework

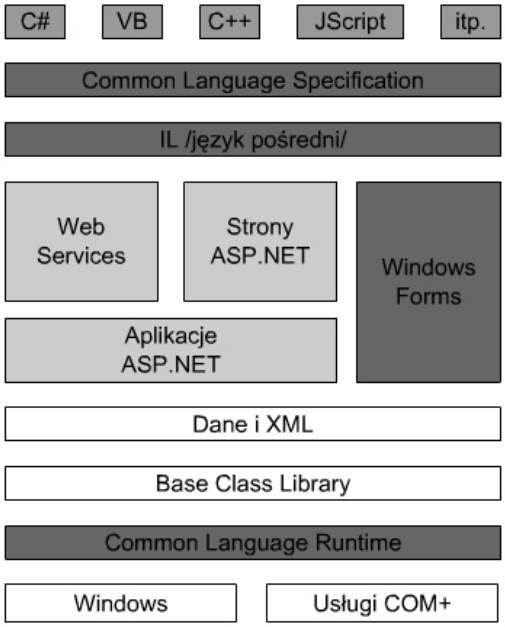
„Platforma .NET sprzyja nie tylko niezależności języków, ale również ich integracji. Oznacza to, że można dziedziczyć po klasach, przechwytywać wyjątki i korzystać z polimorfizmu używając w jednej aplikacji różnych języków. .NET Framework umożliwia to dzięki specyfikacji zwanej wspólnym systemem typów (ang. Common Type System – CTS), z którą muszą być zgodne wszystkie komponenty platformy .NET.” Na przykład wszystko w .NET jest obiektem danej klasy, dziedziczącej po głównej klasie bazowej o nazwie System.Object. CTS obsługuje ogólne pojęcia klas, interfejsów i delegatów, a te ostatnie obsługują wywołania zwrotne. [3]”

.NET zawiera również wspólną specyfikację języka (ang. Common Language Specification – CLS), przekazującą zestaw zasadniczych reguł potrzebnych do integracji języków. CLS określa podstawowe wymagania, które powinien spełniać język z rodziny .NET. Kompilatory zgodne z CLS tworzą obiekty, które mogą pracować ze sobą. Wszystkie języki zgodne z CLS mogą używać dowolnych elementów biblioteki klas platformy .NET (ang. Framework Class Libary – FCL).

.NET Framework funkcjonuje ponad poziomem systemu operacyjnego, którym może być dowolny system z rodziny Windows. Dzięki architekturze środowiska CLR może to być także jakikolwiek system z rodziny UNIX lub wiele innych. Składa się z wielu elementów jak na przykład:

- ✓ Pięć oficjalnych języków – C#, VB, Visual C++, Visual J# i JScript.NET.
- ✓ Środowisko CLR – obiektowa platforma, na której działają programy napisane w wyżej wspomnianych językach.
- ✓ Liczne powiązane biblioteki klas, zwane FCL.

Rysunek 4.1 przedstawia podział .NET Framework.



Rysunek 4.1. Architektura .NET Framework

4.1.3. IL i specyfikacja CLS

Na rysunku 4.1 można zauważyć, że języki programowania znajdują się powyżej wspólnej specyfikacji języków CLS i języka pośredniego MSIL (ang. Microsoft Intermediate Language), często skracanego do IL. Te dwa składniki należy rozpatrywać łącznie ze wspólnym środowiskiem uruchomieniowym CLR.

Projektując .NET, Microsoft podjął radykalną decyzję – sposób kompilacji bądź interpretacji programów oraz generowania programów nie będzie się różnił dla poszczególnych języków programowania. Kompilacja kodu w języku C# ma miejsce w trakcie budowy projektu. Kod IL jest zapisywany w postaci pliku na dysku twardym. W trakcie uruchomiania programu zachodzi ponowna kompilacja kodu IL. Tym razem jest to tak zwana kompilacja JIT (ang. Just In Time), w wyniku której powstaje kod maszynowy wykonywany następnie przez procesor maszyny. Standardowo kompilatory JIT uruchamiane są na żądanie. W momencie wywołania metody kompilator JIT analizuje kod IL oraz tworzy bardzo wydajny kod maszynowy. W momencie gdy aplikacja jest uruchomiona, kompilacja zachodzi tylko wtedy, kiedy jest to potrzebne, natomiast po kompilacji kod znajduje się w pamięci podręcznej, dzięki czemu możliwe jest jego szybkie wykorzystanie. Aplikacja działa

tym szybciej im dłużej jest uruchomiona, ponieważ możliwe jest wykorzystanie większej ilości skompilowanego kodu.

Tradycyjne podejście zakłada, że każdy kompilowany język programowania ma własny binarny kod pośredni, własne typy danych oraz może być obiektowy lub nie. Stosowane są różne mechanizmy przekazywania parametrów do podprogramów. Parametry mogą być odkładane na stos w innej kolejności i kto inny może te parametry usuwać. W rezultacie współpraca komponentów pisanych w różnych językach programowania jest dość trudna. Z takiego powodu powstało wiele systemów pełniących rolę neutralnego pośrednika, czego przykładem jest COM. Stosowanie takiego pośrednika rodzi kolejne problemy. Jego użycie zwiększa złożoność całego systemu, a ponadto pośrednik realizuje tylko zbiór funkcji wspólnych dla wszystkich języków.

Każdy język działający na platformie .NET Framework nie jest kompilowany na własny język binarny, ale na kod bajtowy zwany IL. Nie ma więc znaczenia czy kompilowany jest język w VB, w C++ czy też w C# - w wyniku kompilacji zawsze generowany jest taki sam kod pośredni.

IL różni się od innych języków pośrednich, tym że zawiera konstrukcje charakterystyczne dla obiektowych języków programowania, takie jak dziedziczenie i polimorfizm. Ponadto zawiera konstrukcje charakterystyczne dla typowych języków programowania, jaką jest na przykład obsługa wyjątków. W związku z tym można łatwo przystosować obiektowy język programowania do pracy w .NET (o ile uwzględni się kilka ograniczeń, takich jak pojedyncze dziedziczenie), a do języków, w których nie ma konstrukcji obiektowych, można dodać obsługę takich konstrukcji. Przykładem może być wersja Visual Basic dla .NET nazwana Visual Basic .NET.

Obiektowość IL to bardzo ważna cecha. W .NET wszystko jest obiektowe, gdyż wszystkie języki programowania korzystają ze wspólnej warstwy obiektowej – IL. W rezultacie języki te są zintegrowane. Czasami jednak język programowania musi być dostosowany do modelu wymaganego przez .NET.

Stosowanie języka pośredniego ma wiele zalet. Jedną z nich jest to, że kod uzyskany z kompilacji programów napisanych w różnych językach programowania ma taką samą postać. Łatwo więc łączyć kody pisane w różnych językach, gdyż są one zgodne na poziomie kodu bajtowego.

Języki będą w pełni zgodne, jeżeli będzie ustalony wspólny zbiór konstrukcji i typów danych oraz będą stosowane konwencje ich stosowania. Ten wspólny zbiór został zdefiniowany w CLS. Przestrzeganie zaleceń CLS pozwala na pełną integrację wszystkich

języków programowania dostępnych na platformie .NET. CLS został za darmo udostępniony twórcom kompilatorów. „Wiele firm tworzących kompilatory i wielu projektantów języków programowania poinformowało już, że opracują wersję swoich produktów dla platformy .NET. Tak więc można się spodziewać, że będą dostępne takie języki programowania jak na przykład: COBOL, Fortran, Python, i Perl. [4]”

4.1.4. Środowisko CLR

Najważniejszym elementem .NET Framework jest środowisko CLR, w którym tworzone są programy. W jego skład wchodzi maszyna wirtualna, pod wieloma względami przypominająca maszynę wirtualną Java. Ogólnie rzecz biorąc, CLR tworzy obiekty, przydziela im pamięć, sprawdza bezpieczeństwo, wykonuje zadania i odzyskuje pamięć. Wspólny system typów jest również częścią CLR.

Nad poziomem środowiska CLR na rysunku 4.1 widzimy zestaw klas bazowych platformy, dalej dodatkową warstwę klas do obsługi danych i formatu XML i na samej górze rodziny klasy Web Services, Web Forms oraz Windows Forms. Wszystkie te klasy razem tworzą FCL. Udostępnia ona obiektowy interfejs API dla każdej możliwej operacji, którą obsługuje platforma .NET. Składa się ona z ponad 4000 klas FCL co usprawnia szybkie programowanie aplikacji na komputery domowe, aplikacji typu klient – serwer oraz innych programów i serwisów internetowych.

Najniższy poziom FCL czyli zestaw klas bazowych, przypomina zbiór klas języka Java. Obsługują one operacje wejścia i wyjścia, manipulację ciągami znaków, zarządzanie bezpieczeństwem, komunikację internetową, zarządzanie wątkami, manipulację tekstem, odzwierciedlanie, operacje związane z kolekcjami, a także różne inne funkcje.

Następny poziom stanowi warstwa klas rozszerzających klasy bazowe, które umożliwiają zarządzanie i manipulację danymi w formacie XML. Klasy do obsługi danych dają możliwość stabilnego zarządzania danymi przechowywanymi przez bazy danych. Należą do nich między innymi klasy strukturalnego języka zapytań (ang. Structured Query Language – SQL), które pozwalają na przetwarzanie danych za pomocą standardowego interfejsu SQL. Platforma .NET obsługuje również wiele innych klas umożliwiających przetwarzanie danych w formacie XML, na przykład ich przeszukiwanie i translacje.

Klasy będące rozszerzeniem klas bazowych platformy stanowią wraz z klasami do obsługi danych i XML warstwę przeznaczoną do budowy aplikacji za pomocą trzech różnych technologii – Web Services, Web Forms, Windows Forms. Web Services zawiera liczne

klasy, które umożliwiają tworzenie prostych komponentów, działających nawet w obliczu zabezpieczeń i oprogramowania typu NAT. Ponieważ usługi Web Services wykorzystują do komunikacji protokoły HTTP i SOAP, komponenty te umożliwiają obsługę standardu Plug and Play w internecie.

Windows Forms i Web Forms umożliwiają nam użycie technologii RAD przy tworzeniu aplikacji internetowych i okienkowych na zasadzie przeciągnięcia kontrolki na formę, klikajnięcia jej dwukrotnie i wpisania kodu wykonującego się w odpowiedzi na dane zdarzenie.

4.1.5. Biblioteki klas bazowych

Każdy język programowania i system operacyjny mają biblioteki i funkcje, z których korzystają programiści. Są to na przykład: biblioteka standardowa języka C (ang. C Runtime Libary), Windows API, Standar Tample Libary języka C++, czy też biblioteki MFC i ATL firmy Microsoft.

Niestety, wszystkie te biblioteki są zależne od języka programowania lub od systemu, dla którego zostały stworzone (bądź jednego i drugiego) i nie mają części wspólnych zawierających choćby najprostsze typy danych czy operacje. „Każdy, kto tworzył programy dla środowiska COM, wie, ile trzeba się natrudzić aby przekształcić kolekcję z kodu pisanego w Visual Basic do C++ .Wymaga to zastosowania SAFEARRAY i interfejsu IEnum. [4]”

.NET oprócz tego, że posiada własną bibliotekę klas – Base Class Libary (posiadającą identyczne rozwiązania jak w przypadku zwykłych bibliotek klas), charakteryzuje się dwoma wyróżniającymi je funkcjami:

- ✓ jest to biblioteka klas dla IL, więc można z niej korzystać w każdym języku programowania, który da się skompilować na IL,
- ✓ jest to obiektowa biblioteka klas, więc zestaw funkcji dostępny jest poprzez klasy umieszczone w hierarchiczne uporządkowanych przestrzeniach nazw.

.NET zawiera również wiele innych składników:

- ✓ Definicje typów podstawowych, takich jak Int32. Typy te odwzorowane są na konkretne typy danego języka programowania.

- ✓ Wspólne klasy kolekcji, jak: tablica, lista, tablica mieszająca, wyliczenia, kolekcja, stos.
- ✓ Klasy definiujące wyjątki. Wszystkie języki platformy .NET mogą korzystać z obsługi wyjątków, gdyż jest to część BCL. Tak więc można zgłosić wyjątek w metodzie napisanej w C# i obsłużyć go w metodzie napisanej w VB.
- ✓ Klasy wejścia – wyjścia dla konsoli, plików i strumieni.
- ✓ Klasy umożliwiające programowanie sieciowe, w tym obsługę gniazd.
- ✓ Klasy dostępu do baz danych, w tym między innymi, klasy do obsługi ADO i SQL-a.
- ✓ Klasy do obsługi grafiki, w tym grafiki dwuwymiarowej, przetwarzania obrazów i drukowania.
- ✓ Klasy do tworzenia graficznego interfejsu użytkownika.
- ✓ Klasy do obsługi serializacji.
- ✓ Klasy do implementacji i zapewnienia przestrzegania zasad bezpieczeństwa.
- ✓ Klasy do tworzenia systemów rozproszonych korzystających z sieci WWW.
- ✓ Klasa do obsługi XML-a.
- ✓ Klasy do obsługi wyjątków, zegarów i innych funkcji systemu operacyjnego.

CLR to obiektowy i niezależny od języka programowania następca Windows API. Daje dostęp do bogatego zbioru usług wykorzystanych przez współczesne aplikacje, takich jak korzystanie z sieci WWW, wymiana danych i graficzny interfejs użytkownika.

4.1.6. Jak działa .NET

Kompilatory w środowisku .NET generują pliki EXE i DLL, ale zawartość tych plików jest inna niż do tej pory. Oprócz kodu IL, powstałego z kompilacji kodu źródłowego, zawierają także metadane (ang. metadata). Są to dane opisujące klasy i och możliwości, oddzielone od kodu i klas. Fakt, że metadane, w przeciwieństwie do zmiennych i metod, nie są częścią klasy jest bardzo istotny.

Do czego możemy wykorzystać metadane? Systemy posługujące się komponentami podczas wykonania korzystają z informacji o komponentach, które nie powinny być częścią kodu. Jednym z przykładów są informacje o zabezpieczeniach. Zakładamy, że aplikacja posługuje się komponentem, z którego mogą skorzystać tylko niektórzy użytkownicy. Z reguły lista upoważnionych użytkowników, bądź grup użytkowników może się zmieniać w czasie. Oczywista wiadomością jest, że w czasie wykonania powinno się sprawdzić czy dany

użytkownik, który domaga się dostępu do komponentu znajduje się na tej liście. W jaki sposób można to osiągnąć ? W momencie kiedy informacje o zabezpieczeniach byłyby częścią kodu, aplikacja mająca na celu ustalenie legalności żądania użytkownika musiałaby stworzyć egzemplarz danej klasy, a następnie przesłać zapytanie, co często może być kłopotliwe. Dodatkowo, w sytuacji gdy dojdzie do zmiany listy upoważnionych użytkowników powinno się zmienić kod i w efekcie skompilować go.

Z reguły uważa się, że najlepszym rozwiązaniem jest rozdzielenie takiej informacji od kodu obiektu. W konsekwencji tego rodzaju informacje mogłyby zostać wykorzystane przez narzędzia systemowe czy innego rodzaju aplikacje. W ramach technologii COM stosuj się dwa mechanizmy przechowywania metadanych, gdzie każdy z nich obsługuje różne informacje o komponentach. Jednym z przykładów jest rejestr systemu Windows, który przechowuje informacje indentyfikacyjne i konfiguracyjne. Dzięki tym informacjom COM jest w stanie ustalić, gdzie jest komponent oraz jak stworzyć egzemplarz dla niego. Kolejny mechanizmem jest biblioteka typów składająca się z informacji o wewnętrznej strukturze komponentów, jak na przykład informacje o metodach, zdarzeniach i jego atrybutach.

W wyżej omówionych przykładach metadane są przechowywane w różnych miejscach niż sam komponent, co może stanowić problem. Lista wrogich scenariuszy jest potencjalnie długa – przykładowo biblioteka typów może zniknąć, komponent może zostać skojarzony z biblioteką typów innego komponentu, komponent może nie być zarejestrowany (w Rejestrze nie będzie jego metadanych), dane w rejestrze mogą zostać zniszczone, etc. W .NET rozwiązano to w ten sposób, że metadane przechowywane są wraz z komponentami w jednym pliku. Dzięki takiej metodzie przenoszenie komponentów na inne komputery jest o wiele mniej podatne na błędy konfiguracyjne.

Metadane są wykorzystywane przez CLR do wielu różnych celów, między innymi do:

- ✓ lokalizowania i ładowania klas,
- ✓ rozmieszczania obiektów,
- ✓ odczytywania informacji o metodach i właściwościach klas,
- ✓ egzekwowania zasad bezpieczeństwa,
- ✓ odczytywania informacji o tym, czy i w jaki sposób klasa uczestniczy w transakcjach.

4.1.7. Windows Forms

Tworzenie graficznego interfejsu użytkownika w VB już od wielu lat polega na wybieraniu kontrolek z przybornika, przeciąganiu i upuszczaniu ich na formularz. Kolejną czynnością jest ustawianie właściwości kontrolek i napisanie kodu obsługującego zdarzenie związane z formularzami i kontrolkami. Biblioteka Windows Forms pozwala tworzyć interfejs w ten sam sposób. Ponieważ jest ona częścią .NET Framework, można z niej korzystać w każdym języku programowania platformy .NET.

Biblioteka Windows Forms to kompletny zbiór funkcji o możliwościach związanych z VB. Pozwalają one tworzyć formularze, umieszczać kontrolki na formularzach, ustawiać właściwości kontrolek i definiować interakcje między kontrolkami a formularzami. Można tworzyć aplikacje i okienka dialogowe SDI (ang. Single Document Interface) i MDI (ang. Multiple Document Interface). Windows Forms zawiera dość bogaty zbiór kontrolek, między innymi: listy wyboru, kalendarz, pola tekstowe RTF (ang. Rich Text Format).

4.1.8. Język C#

Język C# jest niezwykle prosty – zawiera jedynie 80 słów kluczowych i kilkanaście wbudowanych typów danych. Jest jednak niezwykle wydajnym narzędziem do implementacji współczesnych technik programistycznych. Język C# obsługuje strukturalne, oparte na komponentach, i obiektowe programowanie, czego zresztą można oczekiwać po współczesnym języku zbudowanym na doświadczeniach z C++ oraz z językiem Java. W wersji 2.0 C# dodano najważniejsze brakujące elementy, jak typy ogólne i anonimowe metody.

Język C# został stworzony przez mały zespół kierowany przez dwóch uznanych inżynierów Microsoftu – Andersa Hejlsberga i Scotta Wilamutha. Hejlsberg jest też znany z jako twórca Turbo Pascala, popularnego języka programowania, a także jako kierownik zespołu tworzącego Borland Delhi, jednego z pierwszych udanych zintegrowanych środowisk programowania aplikacji typu klient – serwer.

Istotą każdego obiektowego języka programowania jest obsługa tworzenia i używania klas. Klasy pozwalają definiować nowe typy, co umożliwia rozszerzenie języka w celu lepszego dopasowania go do rozwiązywanego problemu. Język C# zawiera słowa kluczowe służące do deklarowania nowych klas, ich metod i właściwości, a także do obsługi hermetyzacji, dziedziczenia i polimorfizmu – trzech podstawowych właściwości programowania obiektowego.

W języku C# wszystkie elementy klasy znajdują się w jednej deklaracji. Definicje klas języka C# nie wymagają odrębnych plików nagłówkowych ani plików języka interfejsu (ang. Interface Definition Language – IDL). Jezyk C# obsługuje ponadto nowy styl dokumentacji z wykorzystaniem XML, co ułatwia tworzenie dokumentacji programów zarówno w formie drukowanej, jak i dostępnych na komputerze.

Język C# obsługuje także interfejsy, które umożliwiają zawarcie z klasą kontraktu dotyczącego usług wymaganych przez danych interfejs. W języku C# klasa może dziedziczyć tylko po jednej klasie bazowej, ale może obsługiwać wiele interfejsów. Kiedy klasa w języku C# obsługuje interfejs, gwarantuje udostępnianie wymaganych przez niego usług.

W języku C# występują również struktury, które różnią się nieco od ich odpowiednika z C++. W C# struktury to proste i ograniczone typy, które stawiają mniejsze wymagania systemowi operacyjnemu i systemowi pamięci od zwykłych klas. Struktury nie mogą dziedziczyć po klasach ani stanowić klasy bazowej, mogą jednak obsługiwać interfejs.

Język C# charakteryzuje się pełną obsługa delegatów, umożliwiających pośrednie wywołanie metod. Podobne mechanizmy znajdują się również w innych językach, na przykład wskaźniki na funkcje składowe w C++, jednak delegaty to referencje bezpieczne ze względu na typ, które umożliwiają hermetyzację metod poprzez specyficzne sygnatury i zwracane typy.

C# udostępnia różne mechanizmy związane z komponentami, jak właściwości, zdarzenia, i konstruktory deklaratywne (na przykład atrybuty). Do usługi programowania z wykorzystaniem komponentów służą metadane zapisywane w kodzie klasy. Metadane opisują klasę wraz z jej metodami i właściwościami, a także wymagane zabezpieczenia i inne cechy (na przykład, czy klasa może być serializowana). Kod zawiera wszystkie informacje niezbędne do jego wykonania, a skompilowana klasa jest samowystarczalną jednostką. "Dlatego środowisko, które potrafi odczytać metadane klasy i jej kod, nie musi korzystać z innych informacji by jej użyć. Korzystając z języka C# i środowiska CLR można dodać do klasy własne metadane tworząc atrybuty. Podobnie można odczytać metadane klasy używając typów środowiska CLR obsługujących odzwierciedlanie. [3]”

Po skompilowaniu kodu powstaje podzespół. Jest to zbiór plików widoczny dla programisty jako biblioteka DLL lub plik wykonywalny (z rozszerzeniem .EXE). W przypadku platformy .NET podzespół to podstawowa jednostka wielokrotnego wykorzystania kodu, kontroli wersji, bezpieczeństwa i rozwoju. Środowisko CLR udostępnia liczne klasy do manipulacji podzespołami. Warto wspomnieć, że język C# obsługuje również:

- ✓ bezpośredni dostęp do pamięci przez wskaźniki podobne do tych z języka C++
- ✓ słowa kluczowe oznaczające takie operacje jako niebezpieczne,
- ✓ wskazówki dla mechanizmów przywracania nieużytków, powstrzymujące je przed przedwczesnym usuwaniem pamięci, na którą wskazuje wskaźnik.

4.2. Język XML

O XML-u (eXtensible Markup Language - rozszerzalny język znaczników) stało się bardzo głośno od czasu pojawienia się relacyjnych baz danych. Można by pomyśleć iż jest to istne panaceum na wszystkie bolączki, a każdy szanujący się produkt powinien obsługiwać XML-a.

Dziś słowo XML to słowo – wytrych, tak jak klika lat temu obiektowość. Prawdą jest również, że XML to rewolucja w wyszukiwaniu i wymianie danych. Jest to po prostu język służący do opisu danych. „Ma strukturę drzewiastą, co oznacza że składa się z głównego elementu(korzenia) który zawiera kolejne elementy(gałęzie), które mogą zawierać kolejne elementy(mniejsze gałęzie), i/lub wartości(liście). Każdy element może posiadać także atrybuty(tak jak drzewo może mieć różne kolory czy rozmiary liści).[9]”

4.2.1. Podstawowe składniki dokumentu XML

Podstawowymi składnikami dokumentu XML są:

- ✓ elementy,
- ✓ atrybuty, umieszczane w elementach, jako informacje uzupełniające.

Elementy mogą być:

- ✓ nie puste – posiadają treść,
- ✓ puste – bez treści.

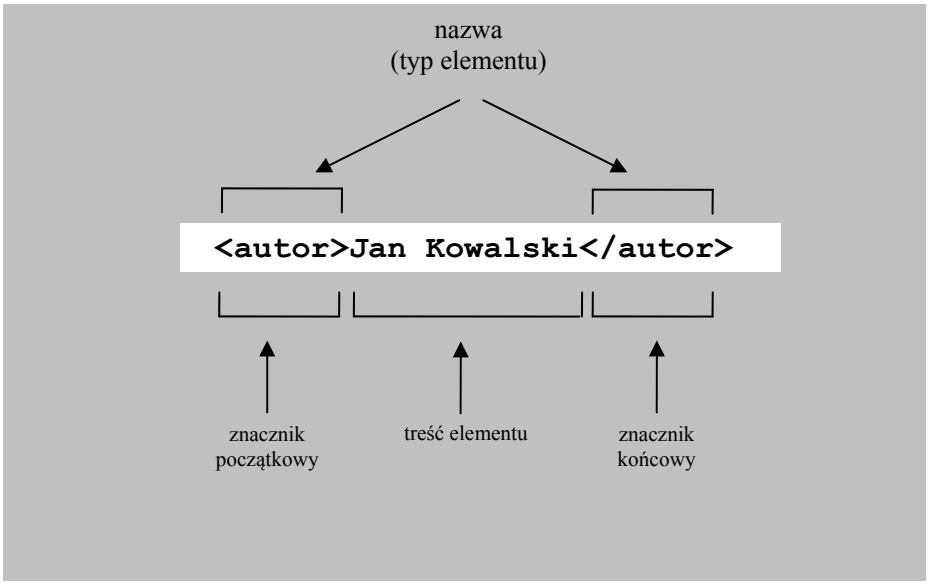
W elemencie XML można umieszczać:

- ✓ deklaracje,
- ✓ jednostki,
- ✓ instrukcje przetwarzania.

Omówmy szerzej wyżej wymienione składniki dokumentów XML.

Elementy

Głównym tworzywem dokumentu XML jest element. Posiada on zazwyczaj jakąś treść, która poprzedzona jest znacznikiem początkowym i zakończona znacznikiem końcowym. Przedstawiono to na rysunku 4.2. Obydwa znaczniki mieszczą nazwę elementu w znakach „<" i „>", przy czym w znaczniku końcowym nazwa poprzedzona jest znakiem „/".



Rysunek 4.2 Składowe elementu XML

W momencie, gdy element posiada dowolną treść, to w języku XML musi wystąpić zarówno znacznik początkowy jak i końcowy. Brak któregoś z nich oznaczać będzie, że konkretny element, a więc i cały dokument, nie są poprawne. W nazwach elementów rozpoznawane są duże i małe litery. Przy tworzeniu dokumentu XML, należy pamiętać, by nazwa w znaczniku początkowym była identyczna jak w znaczniku końcowym. Element o nazwie *autor* jest inny niż element o nazwie *Autor*, *AUTOR* lub *aUtor*. Poniższy element jest niepoprawny, ponieważ nazwa w znaczniku końcowym jest inna niż w znaczniku początkowym:

```
<autor>Jan Kowlaski</AUTOR>
```

Listing 4.1. XML – niepoprawny znacznik

Poprawne są natomiast takie elementy:

```
<autor>Bolesław Prus</autor>
<autor>Jan Kochanowski</autor>
<AUTOR>Henryk Sienkiewicz</AUTOR>
```

Listing 4.2 XML – poprawne znaczniki

Pierwsze dwa są takiego samego rodzaju, trzeci zaś jest inny. Inna jest w tym przypadku nazwa, a więc zazwyczaj inne jest również znaczenie.

Również stosowanie polskich znaków diakrytycznych wymaga odpowiedniego zadeklarowania i ścisłego respektowania kodowania znaków we wszystkich dokumentach XML, a także dokumentach związanych, na przykład arkuszach stylów. Wiąże się z tym niestety kilka problemów. Przykładem może być stosowanie dokumentów XML w kontaktach międzynarodowych bądź wykorzystanie ich w połączeniu z bazami danych, które mogą mieć różne standardy kodowania. Z racji tego nie powinno się stosować polskich znaków w nazwach elementów przy dużych projektach. Dozwolone jest by nazwy znaczników zawierały cyfry, myślniki (dotyczy to także: półpauzy, pauzy oraz innych łączników) i kropki, ważne tylko by się od nich nie zaczynały. Mogą zaczynać się od podkreślenia i dwukropka. Poprawne są poniższe przykłady:

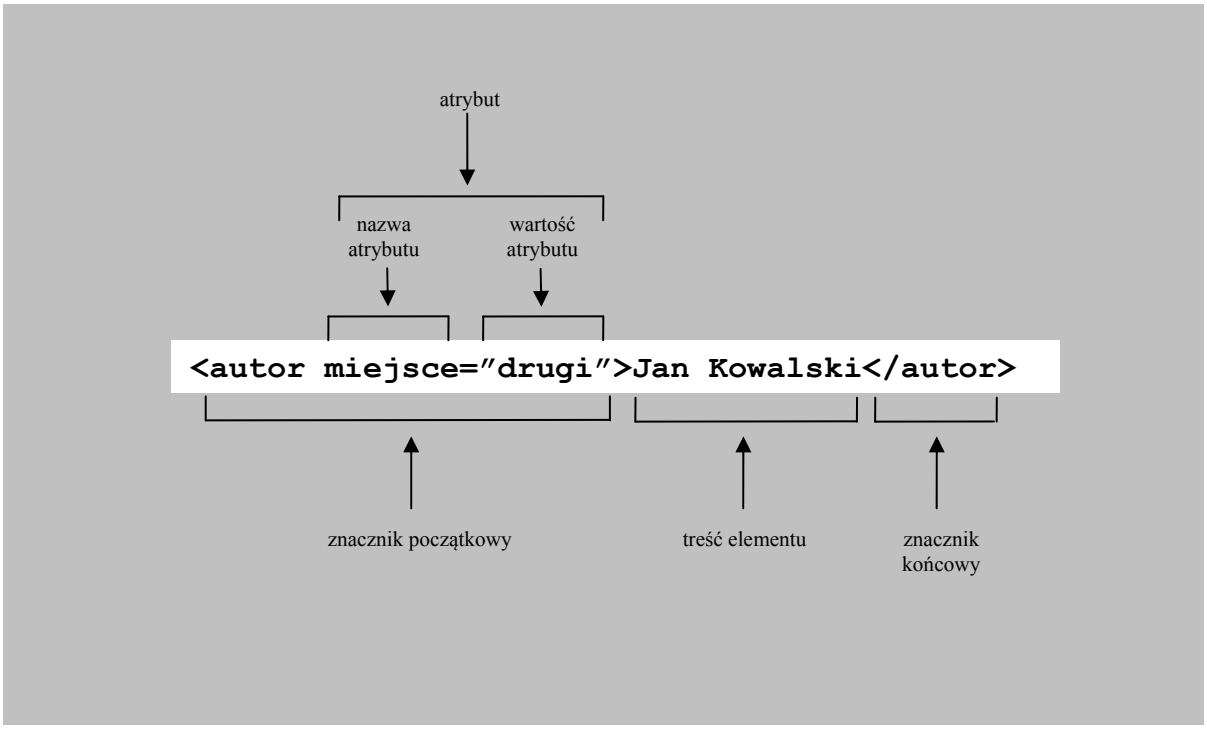
```
<Nazwa.1-2_3:a> Treść </ Nazwa.1-2_3:a>
<_Nazwa.1-2_3:a> Treść </_Nazwa.1-2_3:a>
<:Nazwa.1-2_3:a> Treść </:Nazwa.1-2_3:a>
```

Listing 4.3. XML – znaczniki ze znakami specjalnymi

Nazwa elementu niesie ze sobą bardzo ważną informację. Dzięki niej można określić znaczenie treści zawartej między znacznikami. Z tego powodu należy stosować takie nazwy aby były zrozumiałe. Nie powinno stosować się skrótów, lepiej użyć nazwy na przykład *faktura* zamiast *fktr*, z tego względu, iż odbiorcami dokumentów XML są zazwyczaj zwykli użytkownicy, a nie programiści. Nie ma obawy przed stosowaniem długich nazw, ponieważ język XML nie ma w tej kwestii żadnych ograniczeń.

Atrybuty

Tworząc dokument XML autor może przekazywać informacje nie tylko po przez treść i nazwę elementu. Dodatkowe informacje można umieszczać w postaci atrybutów w znaczniku początkowym elementów. Przedstawia to rysunek 4.3. Atrybuty zawsze posiadają nazwę oraz wartość, poprzedzoną znakiem równości.

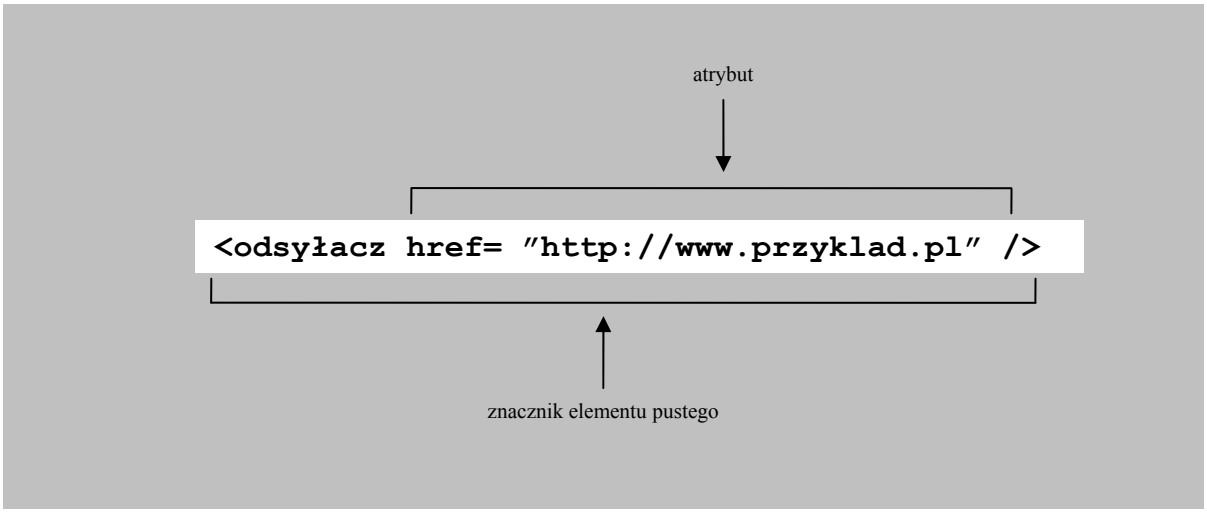


Rysunek 4.3 Element zawierający atrybut

Atrybuty występują tylko w znaczniku początkowym lub elementu pustego. Element może zawierać więcej niż jeden atrybut, pod warunkiem, że posiadają one różne nazwy. Podobnie jak w nazwach elementów, w nazwach atrybutów rozróżnia się małe i wielkie litery. Stosowanie cudzysłowu lub apostrofu dla wartości atrybutu jest wymogiem i pozwala na przechowywanie dowolnego ciągu znaków. Ewentualne liczby traktowane są jako ciągi cyfr, zaś sama kolejność atrybutów jest dowolna.

Elementy puste

Język XML pozwala również stosować elementy puste, czyli takie które nie zawierają treści. W takim przypadku zamiast znacznika początkowego i końcowego, można użyć innego pojedynczego znacznika – znacznika elementu pustego. Składa się na niego nazwa elementu, po której występuję znak „/”, tak jak przedstawia rysunek 4.4.



Rysunek 4.4. Element pusty zawierający atrybut

Można oczywiście zastosować obydwa znaczniki i nie umieścić między nimi żadnej treści, jak na poniższym przykładzie.

```
<faktura_usunieta/>
<faktura_usunieta></faktura_usunieta>
```

Listing 4.4. XML – pusty znacznik

Elementy puste wykorzystuje się zazwyczaj wtedy, gdy zawierają one atrybuty. Można je stosować także wtedy, gdy potrzebne jest przekazanie dodatkowych informacji do programów przetwarzających dokument XML.

4.2.2. Hierarchia elementów.

„Elementy mogą przechowywać tekst (pomiędzy znacznikiem początkowym i końcowym), mogą być puste, ale mogą także zawierać inne elementy — podelementy — dla których są wtedy elementami nadrzędnymi [6]” , np.:

```
<autorzy>
     <wspolautor>Jan Kowalski</wspolautor>
     <wspolautor>Tomasz Nowak</wspolautor>
<autorzy>
```

Listing 4.5. XML – podelementy

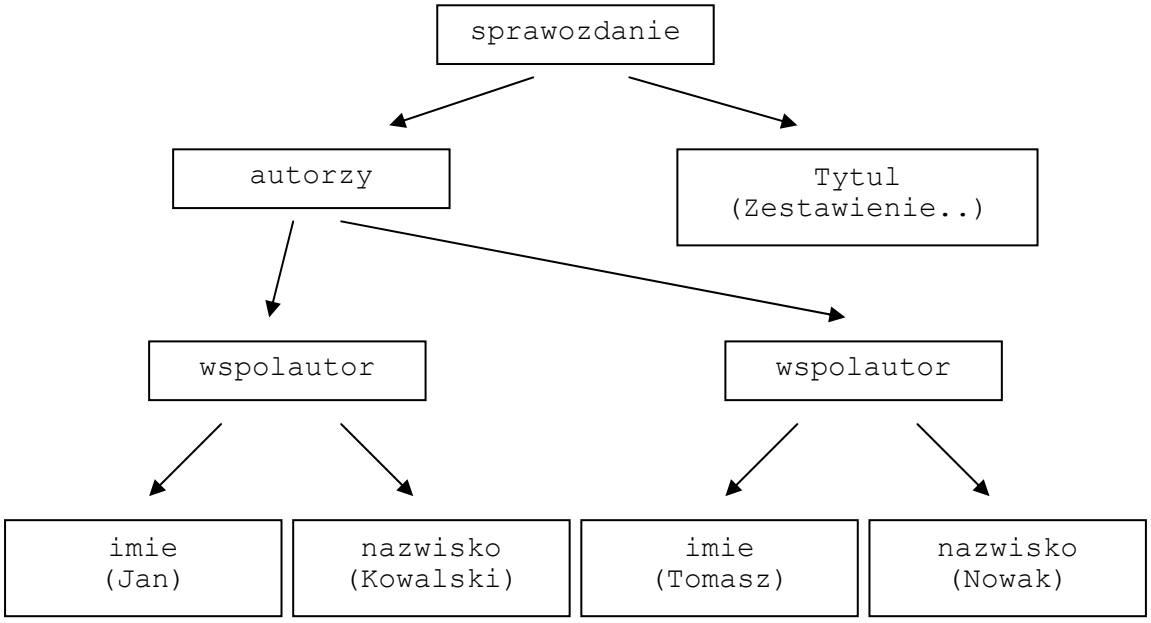
Element *autorzy* zawiera w sobie dwa inne elementy (podelementy) – *wspolautor* – przy czym oba są tego samego rodzaju. Zagnieżdżenie może być bardziej skomplikowane:

```
<sprawozdanie>
     <autorzy liczba_autorow=”2”>
          <wspolautor>
               <imie>Jan</imie>
               <nazwisko>Kowlaski</nazwisko>
          </wspolautor>
          <wspolautor>
               <imie>Tomasz</imie>
               <nazwisko>Nowak</nazwisko>
          </wspolautor>
     </autorzy>

     <tytul> Zestawienie zyskow</tytul>
</sprawozdanie>
```

Listing 4.6. XML – zagnieżdżone podelemeny

Element *sprawozdanie* składa się z podelementu *autorzy* (który również posiada podelementy). Po nim następuje podelement *tytul*. Kolejność podelementów ma duże znaczenie, więc zamiana kolejności podelementu *autorzy* z *tytuł* da w efekcie całkiem inny dokument XML.

Zagnieżdżając jedne elementy w drugich, zawieramy w dokumencie informacje o zależnościach występujących pomiędzy tym, co zostało zawarte w podelelmencie, a samym elementem, a także o zależnościach pomiędzy podelementami. Relacje te to hierarchia elementów, co przedstawia rysunek 4.5.



Rysunek 4.5. Hierarchia elementów

W związku z powyższym język XML idealnie nadaje się do przechowywania informacji, w których maja miejsce relacje podległości (nadrzędny – podrzędny), na przykład: ogół – szczegół, całość – część, lista – element listy. Trudniej jest zbudować hierarchię dla relacji o podobnym poziomie ogólności, jak na przykład klient – towar. Relacja pomiędzy klientem i towarem wyraża się zwykle poprzez zakup lub sprzedaż. Dane o dostawcach lub towarach powielałyby się. Lepszym rozwiązaniem jest zastosowanie w takim przypadku relacyjnej bazy danych.

Stosowanie elementów o takiej samej nazwie w różnych miejscach dokumentu, na różnych poziomach jest dozwolone, warunkiem jest tylko by element nie był korzeniem (elementem głównym). Element *nazwa* umieszczony jest w różnych kontekstach (nazwa firmy, nazwa dostawcy, nazwa towaru, nazwa magazynu, nazwa jednostek miary) i posiada różne rodzaje danych – tekstowe, numeryczne, pojedyncze słowa.

```
<dane_firmowe>
      <nazwa>Producent Jogurtów „Alka”</nazwa>

      <dostawa>
            <odbiorca>
                  <nazwa>Hurtownia Spożywcza „Lalka”</nazwa>
                  <adres>ul. Nieznana 23, Warszawa</adres>
            </odbiorca>
            <towar>
                  <nazwa>Jogurt naturalny</nazwa>
                  <magazyn>
                        <nazwa>5</nazwa>
                  </magazyn>
                  <opakowanie>
                        <nazwa>paleta</nazwa>
                        <jednostka>szt.</jednostka>
                  </opakowanie>
            </towar>
      </dostawa
</dane_firmowe>
```

Listing 4.7. XML – stosowanie takich samych nazw elementów

4.2.3. Rodzaje elementów

W przypadku, gdy element zawiera w sobie jakieś dane (tekst lub podelementy), jest nazywany pojemnikiem (ang. container element). Pojemnik może mieć następujące rodzaje zawartości:

- ✓ zawartość elementową – gdy zawiera wyłącznie inne elementy:

```
<autor>
      <opis>Współautor sprawozdania</opis>
      <imie>Jan</imie>
      <nazwisko>Kowalski</nazwisko>
<autor>
```

Listing 4.8. XML – rodzaje elementów – zawartość elementowa

- ✓ zawartość mieszaną – gdy zawiera tekst oraz inne elementy:

```
<autor>
      Współautor sprawozdania
      <imie>Jan</imie>
      <nazwisko>Kowalski</nazwisko>
<autor>
```

Listing 4.9. XML – rodzaje elementów – zawartość mieszana

- ✓ zawartość tekstową – gdy zawiera tylko tekst:

```
<autor> Współautor sprawozdania: Jan Kowalski</autor>
```

Listing 4.10. XML – rodzaje elementów – zawartość tekstowa

Gdy mamy zawartość mieszaną elementy mogą być wplecione w tekst:

```
<sprawozdanie>
Autorami sprawozdania są: <autor> Jan Kowalski</autor> oraz
<autor> Tomasz Nowak</autor>. Innych autorów nie ma.
</sprawozdanie>
```

Listing 4.11. XML – zawartość mieszana wpleciona w tekst

Zawartość mieszana nie jest jednakże najlepszym rozwiązaniem. Może wywołać ona zamieszanie. Można ją używać dla tekstów, w których chcemy jedynie wydzielić pewne szczególne informacje.

4.2.4. Deklaracja XML

Jak już wiadomo, podstawowym składnikiem dokumentu XML są elementy. To one zawierają podstawową informację, którą chcemy zawrzeć w dokumencie. Jednak aby dokument był rozpoznawany jak dokument XML, należy na jego starcie zamieścić deklarację XML, w której należy:

- ✓ określić wersję języka XML użytego w dokumencie – parametr *version*,
- ✓ określić rodzaj kodowania znaków – parametr *encoding*,
- ✓ podać informacje o pomięciu (lub nie) przetwarzania zewnętrznych definicji typu dokumentu oraz zewnętrznych jednostek – parametr *standalone*.

Deklaracja XML zaczyna się znakami *<?xml* i kończy *?>*. Poniżej znajduje się przykład deklaracji XML:

```
<?xml version=”1.0” encoding=”UTF-8” standalone=”yes” ?>
```

Listing 4.12. XML – deklaracja

Parametr *version* jest obowiązkowy i na dzień dzisiejszy powinien mieć wartość 1.0, gdyż tylko taka wersja obecnie istnieje. Pozostałe parametry są opcjonalne więc nie muszą występować. XML narzuca nam też pewne ograniczenia w kwestii kodowania. Mamy możliwość stosowania tylko jednego rodzaju w dokumencie. W przypadku stosowania języka XML w połączeniu z bazami danych, w których często stosuje się różne rodzaje kodowania, może to być pewnym utrudnieniem.

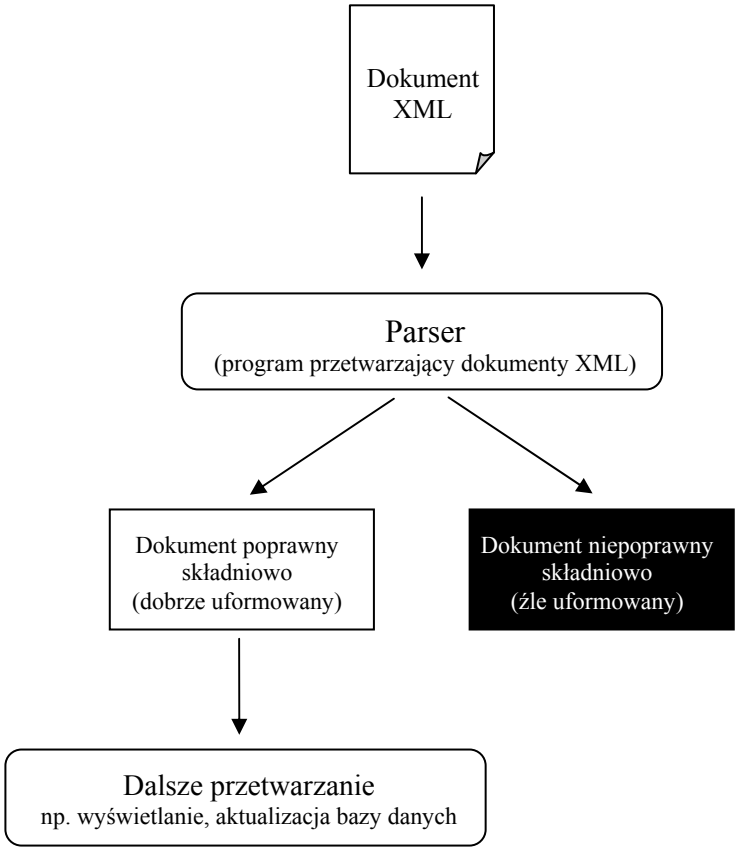
4.2.5. Przetwarzanie dokumentów XML

„Dokumenty XML mogą być czytane przez użytkowników w zwykłych edytorach tekstowych, jednak najczęściej są one przetwarzane przez programy, zwane parserami. [6]” Takie programy najpierw powinny sprawdzić poprawność składniową dokumentu XML. Sposób przetwarzania dokumentów XML przedstawiony został na rysunku 4.6.

Następnie mogą one wykorzystać informację zawartą w dokumencie w konkretnym celu:

- ✓ wyświetlić ją użytkownikowi w określonym formacie,

- ✓ zmienić postać informacji, np. przekształcić dokument XML na stronę w języku HTML,
- ✓ wygenerować nowy dokument XML i przekazać go innemu programowi,
- ✓ przetworzyć informację do postaci zgodnej z określonym formatem bazy danych i zaktualizować bazę danych.



Rysunek 4.6. Przetwarzanie dokumentu XML

Do dalszego przetwarzania XML zwykle niezbędne jest posiadanie dodatkowych informacji, np. arkuszy stylów, transformacji, definicji typu dokumentu, itp.

4.2.6. Inne składniki języka XML

Dokument XML może zawierać, oprócz zwykłych elementów, także instrukcje dla programu przetwarzającego – parsera, takie jak:

- ✓ deklaracje (w tym komentarze i bloki tekstu),

- ✓ instrukcje przetwarzania,
- ✓ jednostki.

- **Deklaracje**

Deklaracje umieszczone są w tak zwanych znacznikach deklaracji. Nieznacznie różnią się od zwykłych znaczników. Otoczone są znakami „<!” oraz „>”. Na początku każdej deklaracji powinna znajdować się nazwa niewątpliwie identyfikująca daną deklarację:

```
<!NAZWA_DEKLARACJI …>
```

Listing 4.13. XML – budowa deklaracji

Znaczniki deklaracji wykorzystywane są również do grupowania kilku różnych deklaracji zamkniętych w zbiorze przy użyciu znaków „[” oraz „]”. Niektórych rodzajów deklaracji nie wolno grupować:

```
<!zbior_deklaracji[
     <!deklaracja1>
     <!deklaracja2>
]>
```

Listing 4.14. XML – budowa zbioru deklaracji

Standard XML 1.0 określa następujące deklaracje:

- ✓ deklaracje podstawowe (umieszczane w dokumentach XML),
- ✓ deklaracje wykorzystywane w definicjach typu dokumentu DTD.

Deklaracje podstawowe to:

- ✓ komentarze,
- ✓ sekcje CDATA.

Komentarze

Komentarze można umieszczać w dowolnym miejscu w dokumencie XML. Zawierają one zazwyczaj informację w języku naturalnym zrozumiałą jedynie dla człowieka, dlatego pomijane są przez programy przetwarzające dokument – parsery. Komentarz otoczony jest znakami „<!--” oraz „-->”. Poniżej przykład komentarza:

```
<autor> Jan Kowalski</autor>  <!-- autor sprawozdania -->
```

Listing 4.15. XML – komentarz

Sekcje CDATA

Nie wszystkie znaki (symbole) mogą być użyte w treści elementu, gdyż mają istotne znaczenie dla przetwarzania dokumentu. W przypadku, gdy potrzeba użyć znaków „<”, „>” czy „&”, zamiast używania kodów odpowiadających tym znakom (<, >, &) można wykorzystać tak zwany blok tekstu, czyli sekcję CDATA. Wszystkie znaki ujęte w tej deklaracji będą wówczas traktowane jako zwykły tekst. Sekcja ta ma postać:

```
<![CDATA[tekst]]>
```

Listing 4.16. XML – sekcja CDATA

Rozważmy przykład, w którym chcemy aby została wyświetlona informacja z pliku XML – „Naciśnij <<< Enter >>> aby pójść dalej”. Poprawny dokument XML wyglądałby następująco:

```
<system>
  <komunikat>
  Naciśnij &lt; &lt; &lt; Enter &gt; &gt; &gt; aby pójść dalej
  </komunikat>
</system>
```

Listing 4.17. XML – rozwiązanie bez sekcji CDADTA

Zamiast symbolu „<” oraz „>”zastosowano odpowiednio < i >. Można również użyć tutaj sekcji CDATA. Dokument wyglądałby następująco:

```
<system>
  <komunikat>
  <![CDATA[Naciśnij <<< Enter >>> aby pójść dalej]]>
  </komunikat>
</system>
```

Listing 4.18. XML – rozwiązanie z sekcją CDADTA

Tekst zawarty w sekcji CDATA będzie zawsze traktowany przez parser jako ciąg znaków.

- **Instrukcje przetwarzania**

Instrukcje przetwarzania zawierają informacje konieczne dla programów przetwarzających dokument XML. Instrukcje umieszczane są w bloku ograniczonym znakami „<?" i „?>". Zawierają one cel oraz instrukcję do wykonania. Celem jest słowo kluczowe identyfikujące aplikację, której przeznaczona jest instrukcja do wykonania. Jeżeli program przetwarzający dokument rozpozna cel , zadana instrukcja zostanie przez niego wykonana:

```
<!-- instrukcje dla robotów internetowych: zaindeksować i nie
przechodzić dalej po odsyłaczach -->
<?robots index='yes' follow='no'?>
<!-- instrukcje dla edytorów tekstowych -->
<?DTPSystem DO:page-break?>     <!--Dla programu DTP: wstawić
stronę-->
<?EdytorTXT {New Page}?> <!-- Edytor tekstowy ma wstawić stronę -
->
```

Listing 4.19. XML – instrukcje przetwarzania

Jeżeli cel nie jest znany aplikacji przetwarzającej dokument, wtedy cała instrukcja zostanie przez nią pominięta. Instrukcje umożliwiają umieszczanie w dokumencie XML fragmentów innych języków, zwłaszcza programowania. Pewnym ograniczaniem jest jednak fakt, iż dwuznak „?>" nie może zostać zawarty w instrukcji do wykonania.

„Instrukcje przetwarzania mogą być:

- ✓ standardowe, zdefiniowane w którymś ze standardów związanym z językiem XML, np.: `<?xml…?>`, `<?xml-stylesheet…?>` - wykorzystywane w języku XSLT
- ✓ niestandardowe - niezwiązane ze standardami języka XML, czyli utworzone dla specjalistycznych potrzeb, np. dla skryptów PHP, konkretnego edytora tekstów czy dla potrzeb przetwarzania w procesie wymiany danych między firmą X a firmą Y. [6]"

- **Jednostki**

Język XML umożliwia fizyczną separację części przechowywanego dokumentu. Przykładowo, gdyby dokument XML opisywał książkę, to jej rozdziały można by umieścić w

osobnych plikach. Umożliwia to łatwiejszą kontrolę, prostszą edycję czy nawet krótszy czas ładowania pliku przez Internet. Każda taka osobna część jest nazywana jednostką bądź encją. Każda z nich swoją nazwę, przez którą jest identyfikowana. Jednostka jest definiowana słowem kluczowym *ENTITY*, które wchodzi w skład deklaracji jednostki, a ta z kolei może być wystawiona jedynie w deklaracji typów dokumentów. Wykorzystanie jednostki w dokumencie polega na umieszczeniu w żądanym miejscu odsyłacza do danej jednostki. Odesłanie składa się z nazwy jednostki poprzedzonej znakiem „&" oraz zakończonej znakiem średnika. W dokumencie może wystąpić wiele odesłań do tej samej jednostki. Najczęściej stosuje się jednostki, gdy:

- ✓ ta sama informacja używana jest wielokrotnie,
- ✓ informacja jest częścią dużego dokumentu i z powodów praktycznych podzielona jest na łatwiejsze w zarządzaniu części,
- ✓ informacja zawiera dane nie przetwarzane przez dokument XML, lecz przez inną aplikację.

Najprostszą formą jednostki jest jednostka wewnętrzna tekstowa, zwana także ogólną jednostką wewnętrzną. Jej deklaracja znajduje się w przetwarzanym dokumencie XML – zawiera wyłącznie tekst. Stosuje się ją w przypadku częstego powtarzania fragmentu tekstu. Następująca deklaracja :

```
<!ENTITY XML „eXtesible Markup Language”>
```

Listing 4.20. XML – jednostka wewnętrzna tekstowa

oznacza, że nazwie XML przyporządkowana tekst „eXtesible Markup Language", zaś każde odwołanie o tej nazwie (czyli *&XML;*) będzie zastępowane przez ten tekst. Przykładowe odwołanie:

```
<opis>To jest tekst o XML czyli o &XML;. Angielski termin „&XML;”
oznacza rozszerzalny język znaczników</opis>
```

Listing 4.21. XML – zastosowanie jednostki wewnętrznej

jest równoznaczne z:

```
<opis>To jest tekst o XML czyli o eXtesible Markup Language.
Angielski termin „eXtesible Markup Language” oznacza rozszerzalny
język znaczników</opis>
```

Listing 4.22. XML – wynik stosowania jednostki tekstowej wewnętrznej

Kolejnym typem jednostek jest jednostka zewnętrzna tekstowa, zwana także ogólną jednostką zewnętrzną. Zdefiniowanie jednostki zewnętrznej różni się od definicji jednostki wewnętrznej. W tym przypadku konieczne jest podanie, z jakiego pliku należy zaczerpnąć informację o treści danej jednostki. Deklaracja:

```
<!ENTITY XML SYSTEM „http://www.informacje.pl/XML skrot.xml”>
```

Listing 4.23. XML – jednostka zewnętrzna tekstowa

oznacza, że w pliku http://www.informacje.pl/XML_skrot.xml znajduje się angielska nazwa języka XML. Odwołanie do ogólnej jednostki zewnętrznej jest takie samo jak do wewnętrznej:

```
<opis>To jest tekst o XML czyli o &XML;. Angielski termin „&XML;”
oznacza rozszerzalny język znaczników</opis>
```

Listing 4.24. XML – zastosowanie jednostki zewnętrznej

4.2.7. Schematy XML

Dane zapisywane są w języku XML, natomiast sposób prezentacji określany jest za pomocą języka XSL – pozwala on przekształcać dane z dokumentu XML na bazie szablonu. Korzystając z tego narzędzia można dane przegrupowywać i wybierać. Można je także przygotować do wizualnego przeglądania. Arkusze stylów XSL mogą być osobnymi plikami, które za pomocą deklaracji zawartej na początku dokumentu przypisuje im się dokumenty z danymi. Oczywiście jednego arkusza stylów można używać do przekształceń wielu plików z danymi tworzącymi tę samą strukturę.

XSD jest językiem definiowania schematów dokumentów XML. Dzięki schematom XSD możemy przekazać razem z danymi ich typ, strukturę, swego rodzaju instrukcję obsługi dla systemów otrzymujących dane XML.

Dzięki schematom XML można weryfikować dane oraz strukturę dokumentu. Weryfikacja poprawności dokumentu odbywa się między innymi po przez sprawdzenie:

- ✓ czy znacznik końcowy elementu głównego jest ostatnim ze znaczników elementów,
- ✓ czy liczba znaczników początkowych jest równa liczbie znaczników końcowych.

Dane dokumentu również należy poddać analizie poprawności. Można zauważyć, że dane mogą być zawarte w różnych strukturach (bez lub z atrybutami). Powstaje więc problem w rozpoznawania formatu. W takim przypadku można zdefiniować schemat XML, który będzie używać aplikacje – w ten sposób zostanie zweryfikowana struktura. Schemat może zawierać ograniczenia dziedzinowe danych (przykładowo czy wartość liczby ma być dodatnia, itp.).

Schematy XML wprowadzają możliwość definiowania dwóch rodzajów typów zawartości :

- ✓ prostych
- ✓ złożonych

Typy proste

Są to zbiory wartości atomowych (nie zawierające zagnieżdżonych elementów), czyli wszystkie typy wbudowane (np. tekst, wartość logiczna. liczba), jak również utworzone na ich fundamencie. Poniższy przykład przedstawia deklarację typu prostego:

```
<xs:element name="Nazwa" type="xs:string" />
```

Listing 4.25. XML – deklaracja typu prostego

Schematy XML oferują ponad 40 wbudowanych typów danych. Te najczęściej używane to:

- ✓ string – ciąg znaków
- ✓ boolean – wartość logiczna (PRAWDA/FAŁSZ)
- ✓ integer – liczba całkowita z przedziału <–126789; 126789>
- ✓ float – liczba rzeczywista
- ✓ dateTime – data i czas
- ✓ date – data

- ✓ time – czas
- ✓ gYear – rok (gregoriański)

Typy złożone

Typy złożone natomiast zbudowane są z elementów podrzędnych. Dopuszczalne są różne sposoby grupowania tych elementów:

- ✓ grupowanie (xsd:group)
- ✓ ustalenie ściślej kolejności – sekwencja (xsd:sequence)
- ✓ zezwolenie na wybór z kilku możliwości – alternatywa (xsd:choice)
- ✓ określenie zbioru elementów, nie narzucające kolejności (xsd:all)

W celu określenia liczby wystąpień poszczególnych elementów lub całych grup używa się atrybutów:

- ✓ minOccurs – minimalna ilość wystąpień
- ✓ maxOccurs – maksymalna liczba wystąpień

przyjmujące wartości całkowite lub wartość „unbound" oznaczającą nieograniczona liczbę wystąpień. Jeśli atrybuty te nie są podane, to przyjmowana jest ich domyślna wartość, czyli własne typy danych można bardzo łatwo utworzyć na podstawie tzw. aspektów (*ang. facets*). Najistotniejsze to:

- ✓ minInclusive i maxInclusive – określające minimalną i maksymalną wartość liczbową włączając w to wartości brzegowe
- ✓ minExclusive i maxExclusive – określające minimalną i maksymalną wartość liczbową wyłączając wartości brzegowe
- ✓ pattern – określające wyrażenie regularne
- ✓ enumeration – ograniczające typ jedynie do wyliczonych wartości
- ✓ list – umożliwiające tworzenie list wartości typu prostego
- ✓ length – wymagana długość tekstu lub listy
- ✓ minLength i maxLength – określające minimalną i maksymalną długość tekstu lub listy

Poniżej znajduje się dokument XML:

```
<?xml version="1.0" encoding="utf-8" ?>
<sprawozdania>
  <sprawozdanie ID=”1” Tyutl=”Roczne 1997” Autor=”Nowak”>
  <sprawozdanie ID=”2” Tyutl=”Roczne 1998” Autor=”Kowalski”>
  <sprawozdanie ID=”3” Tyutl=”Roczne 1999” Autor=”Elegancki”>
  <sprawozdanie ID=”4” Tyutl=”Roczne 2000” Autor=”Kosicelny”>
</sprawozdania>
```

Listing 4.26. XML – dokument XML

oraz jego schemat:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="sprawozdania">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="sprawozdanie">
          <xs:complexType>
            <xs:attribute name="ID" type="xs:unsignedByte"
use="required" />
            <xs:attribute name="Tytul" type="xs:string"
use="required" />
            <xs:attribute name="Autor" type="xs:string"
use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing 4.27. XML – schemat XML

XSD zawiera zbiór predefiniowanych typów oraz standardowy język opisów własnych typów złożonych. Dokument XSD korzysta z tych typów do opisania i ograniczenia zawartości komunikatu XML.

5. Narzędzia

5.1. Microsoft Visual Studio 2005

Produkt firmy Microsoft – Visual Studio 2005 jest to zbiór narzędzi programistycznych pozwalający na tworzenie samodzielnych aplikacji – zarówno internetowych, okienkowych czy też na urządzenia przenośne. W wersji 2005 zostało wprowadzonych bardzo dużo nowych udogodnień w porównaniu z wcześniejszym już bogatym zestawem narzędzi z poprzednich wersji. Głównym celem nowinek jest zwiększenie produktywności programistów, ułatwienie wykonywania żmudnych zadań i włączenie całego zespołu projektowego w tworzenie oprogramowania. W kolejnych podrozdziałach opiszemy niektóre przydatne narzędzia tego produktu.

5.1.1. Projektowanie, pisanie i przeglądanie kodu

Każdy programista ma za zadanie napisać kod jak najbardziej czytelny, który można by zrozumieć, pielęgnować czy rozszerzać. Niestety programiści najczęściej pracują pod dużą presją czasu – muszą oddać jak najszybciej wersje produkcyjną programu, bądź nie mają wystarczająco dużo czasu by przejść odpowiednie szkolenia. Jest to główną przyczyną sytuacji, w której wiele kodów nie spełnia standardu. Takie warunki powodują, iż nie da się usunąć nadmiernej złożoności kodu, można ją co najwyżej zwiększyć. Tu z pomocą przychodzi Visual Studio 2005. Może udostępniać wskazówki i „szkolenia" bezpośrednio w edytorze. Wykonuje za programistę rutynowe i żmudne operację i to w dodatku bezbłędnie. Daje to więcej czasu na przemyślenie logiki biznesowej i utworzenie produktu o wyższej jakości w krótszym czasie.

Planowanie i projektowanie kodu

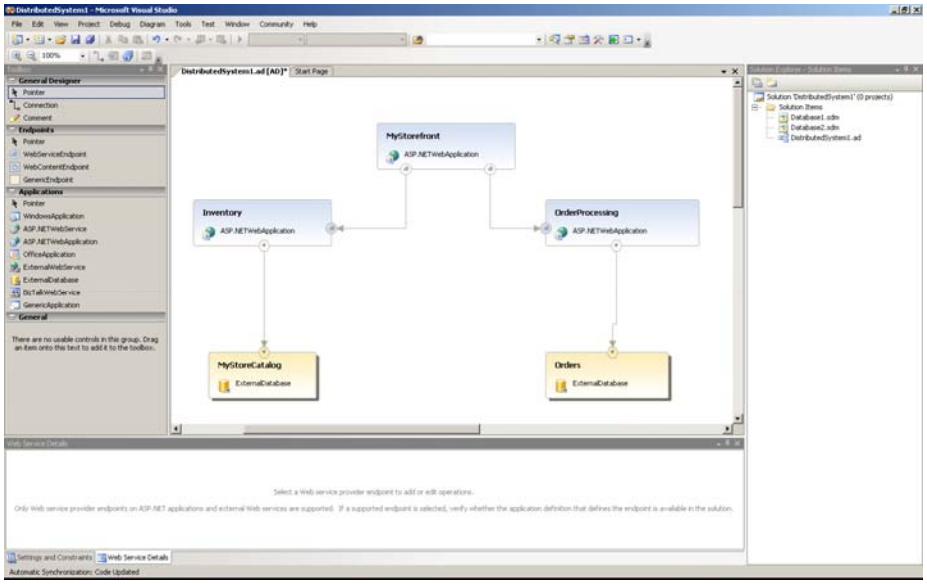
Wizualizacja kodu oraz solidna architektura nie tylko ułatwia proces programowania, ale również sprzyja tworzeniu lepszego oprogramowania. Zazwyczaj dokumenty z architekturą projektu, jak na przykład Visio (produkt Microsoftu), tworzone są na początku pracy, w wyniku czego stają się nieaktualne już w trakcie pisania kodu. Prawie nigdy nie ma czasu na aktualizacje diagramów i można tylko „obserwować ich szybką degradację do dokumentów z „oryginalnym projektem" czy „wizją systemu" [1]". Problem ten miał zostać rozwiązany dzięki synchronizacji między narzędziami do modelowania a projektem

programistycznym. Powodowało to jednak kolejne trudności, gdyż „programiści wolą oglądać kod w oknie z kodem lub środowisku IDE, a nie w odrębnym narzędziu. [1]”

Rozwiązanie tej ciężkiej sytuacji architektów i programistów umożliwiło Visual Studio 2005. Zanim rozpocznie się tworzenie kodu można stworzyć pełen projekt aplikacji. Narzędzia do modelowania są zintegrowane ze środowiskiem IDE, dzięki czemu zawsze zachodzi synchronizacja miedzy modelem i kodem. Okno Code Designer przykładowo wyświetla graficzny obraz kodu. Jakiekolwiek wprowadzone zmiany w tym oknie natychmiast są widoczne w kodzie i na odwrót. Poniżej pokazany spis przedstawia opis graficznych okien projektowych w Visual Studio 2005.

- **Application Designer**

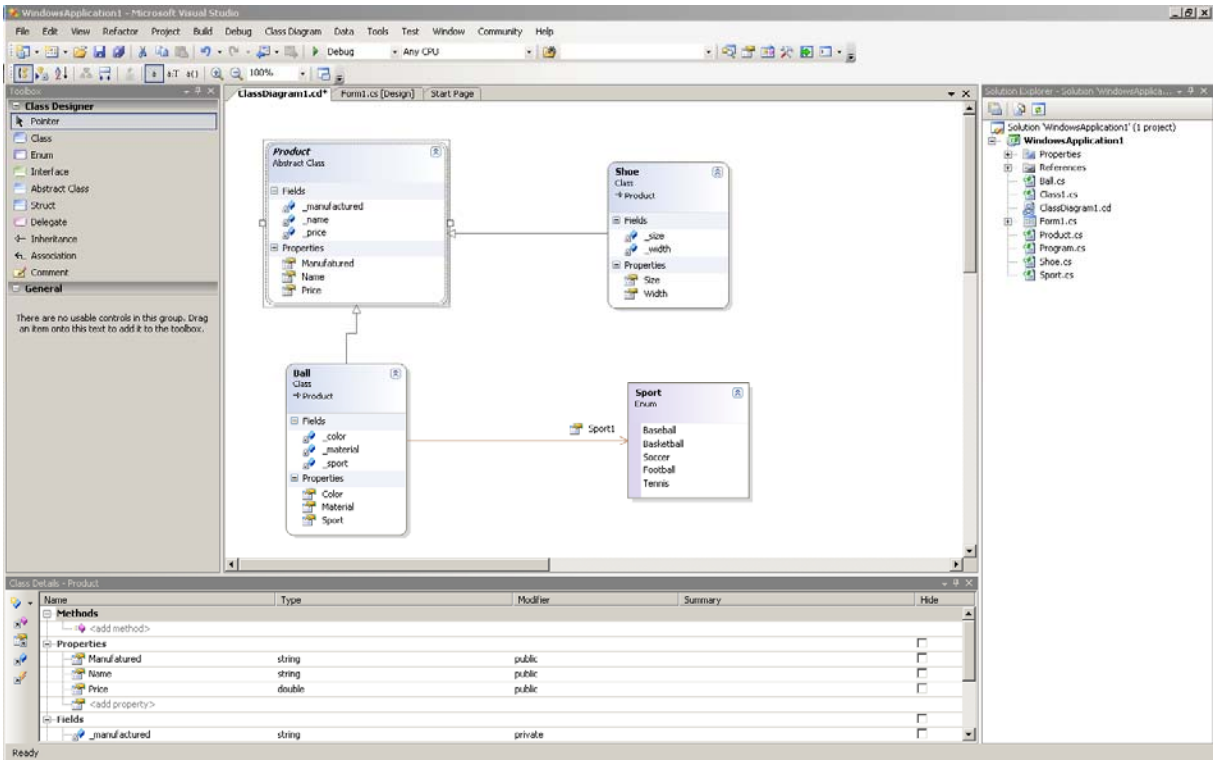
Okno to umożliwia architektom kreację modelów definicji systemu – SDM (ang. System Definition Model), które określają jakie „aplikacje” (w znaczeniu: witryna internetowa, usługa sieciowa, baza danych, etc.) zostały połączone ze sobą w danym rozwiązaniu. Rozważmy przykład witryny internetowej, która komunikuje się z licznymi usługami sieciowymi, te natomiast komunikują się z bazą bądź kolejką komunikatów. Application Designer umożliwia tworzenie takiego typu modeli. Architekt nie tylko może określić, które aplikacje komunikują się ze sobą, ale także zdefiniować ograniczenia w relacjach między nimi. Rysunek 5.1 przedstawia taki przykład.



Rysunek 5.1. Sklep internetowy widoczny w oknie Application Designer

- **Class Designer**

Okno to umozliwia nam tworzenie i modyfikowanie obiektów.za pomocą narzędzi graficznych umożliwiających zarówno definiowanie klas i relacji między nimi, dodawanie właściwości i metod do tych klas, jak i modyfikację poszczególnych elementów w obrębie właściwości i metod. Przy pomocy Class Designer możemy również poddać kod refaktoryzacji. Przykładowo zmiana nazwy jakiejś konkretnej metody w tym oknie powoduje natychmiastową aktualizację nazwy w innych miejscach, między innymi w miejscu wywołania. Kolejną ważną zaletą jest opcja szybkiego przeglądania kodu. Przeciągając do okna projektowego parę kluczowych klas modelu oraz pozwalając narzędziu określić relacje między nimi zachodzące w kodzie możemy, przy pomocy graficznych referencji, przetestować działanie aplikacji. Rysunek 5.2 przedstawia taki mechanizm.

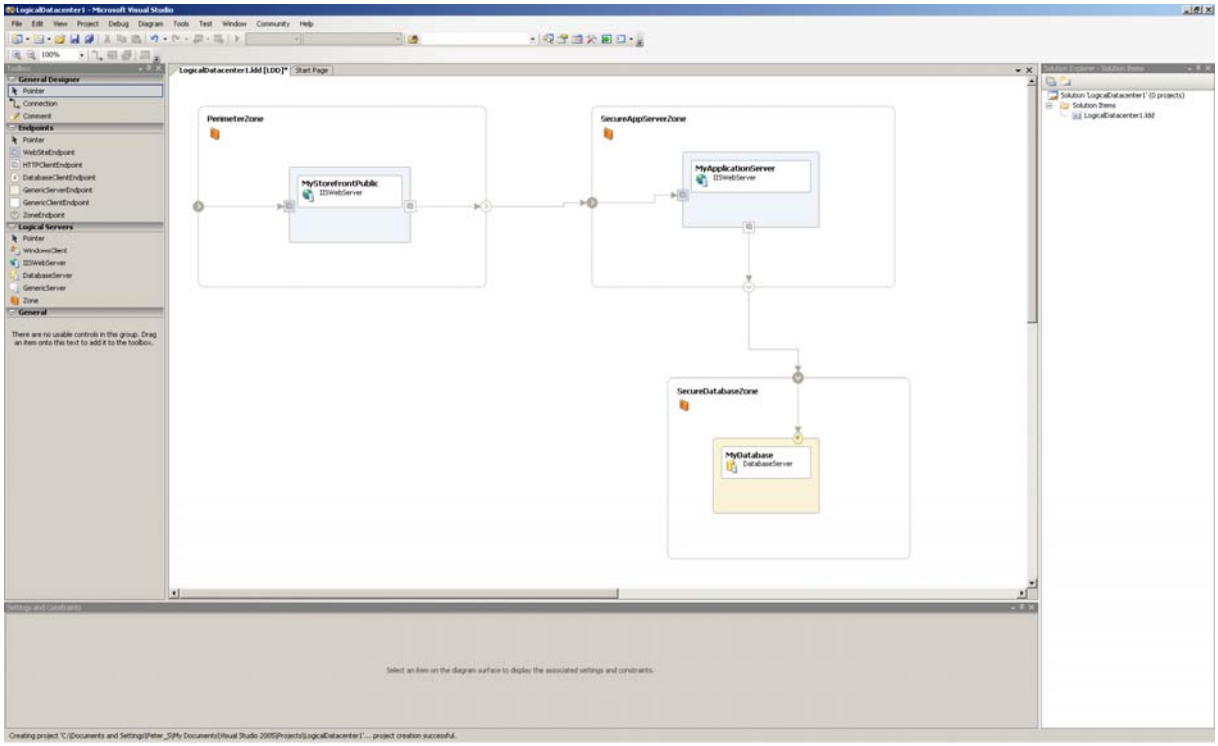


Rysunek 5.2. Ilustracja obiektów i relacji między nimi w oknie Class Diagram

- **Logical Datacenter Designer**

Logical Datacenter Designer umożliwia usprawnianie komunikacji pomiędzy zespołami, które odpowiadają za infrastrukturę i programowanie. Istnieje tu możliwość określania granic wyznaczających strefy w obrębie centrum danych, które mogą przekraczać te granice, a także serwerów znajdujących się w poszczególnych strefach. Możemy również „narzucić

ograniczenia na elementy znajdujące się w modelu logicznym, włączając w to wersje oprogramowania działające na maszynach, typy danych, które można przesyłać do danych komputerów, zapytania kierowane do aplikacji, informacje w globalnej pamięci podręcznej podzespołów, zarządzanie sesjami i tak dalej. Te modele można podpisać i kontrolować ich wersje (ponieważ rzadko się zmieniają), a projekt aplikacji można sprawdzić względem modelu logicznego. [1]" Na rysunku 5.3 przedstawiono logiczną reprezentację serwerów w przykładowym centrum danych.



Rysunek 5.3. Logiczna reprezentacja serwerów w przykładowym centrum danych

- **Deployment Designer**

Dzięki Deployment Designer możemy testować wdrażanie na podstawie logicznego modelu centrum danych. Mechanizm ten sprawdza możliwość wdrożenia rozwiązania. Pozwala również na przeciągniecie aplikacji na odpowiedni serwer i sprawdzenie możliwości wdrożenia przy pomocy kompilatora wdrożeń. Przykładowo mechanizm ten nie pozwala na umieszczenie aplikacji sieciowej na serwerze bazodanowym, który nie umożliwia przesyłania danych internetowych. W trakcie kompilacji może zweryfikować, czy na docelowym serwerze istnieje oprogramowanie, które umożliwiałoby start aplikacji. Wszystkie błędy są pojawiają się jako spis zadań , jak błędy w kodzie. Wszystko to pozwala wykryć problemów

wdrożeniowych, zanim kod zostanie umieszczony na serwerach produkcyjnych, kiedy błędy te mogłyby wyjść stosunkowo kosztowne.

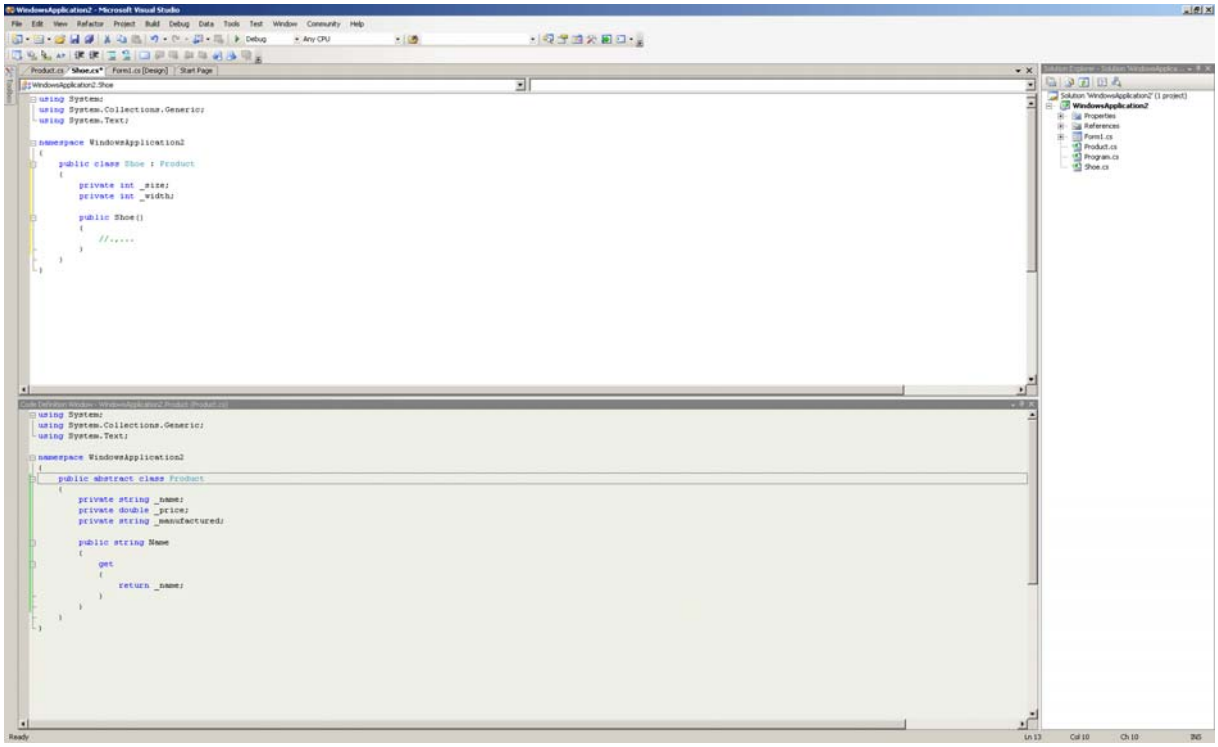
Podglądanie kodu i nawigacja po nim

Raz na jakiś czas każdy programista spotyka się z bazą nieznanego kodu w celu naniesienia poprawek, bądź rozbudowania go. Zazwyczaj trzeba przejrzeć tysiące linijek kodu, wykonać setki przejść, aby wykryć różne zależności i zrozumieć istniejącą złożoność. Mechanizmy takie jak Class Designer mogą stać się tu bardzo pomocne. Visual Studio 2005 ma kolejne nowe właściwości, które mogą być rzydatne przy tego typu zadaniach. Lista tych funkcji została opisana poniżej.

- **Code Definition Window**

Przeglądając kod, programiści niejednokrotnie natrafiają na typ, do którego potrzebują uzupełniających informacji. Potrzebują na przykład zobaczyć kod, na którym bazuje typ, by dowiedzieć się jak został zaimplementowany. Zazwyczaj zmuszeni byli do przeszukiwania plików z zawierającym je kodem lub do użycia opcji Go to Definition, dostępnej po kliknięciu prawego klawisza myszy. Visual Studio 2005 prezentuje nam nowość – okno z definicjami kodu. Okno to powoduje rozbicie edytora na dwie części. Jedna to aktywna część z edytowanym aktualnie kodem a druga z kodem przeznaczonym tylko do odczytu. Rozważmy przykład gdzie mamy dwie klasy. Pierwsza o nazwie „Shoe" dziedziczy po abstrakcyjnej klasie bazowej „Product". Dzięki Code Definition Window mamy możliwość podglądu klasy bazowej bez przełączania się między plikami, tak jak prezentuje to rysunek 5.4.

Okno to nie służy jedynie do przeglądania kodu tylko do odczytu. Umożliwia również zaznaczanie tekstu by kopiować go bezpośrednio do kodu, markowanie punktów przerwań lub tworzenie zakładek. W każdej chwili można przejść do trybu edycji kodu wyświetlanego w oknie z definicjami kodu wybierając Edit Definition.



Rysunek 5.4. Code Definition Window – okno z definicjami kodu.

- **Class View – wyszukiwarka**

Okno Class View w tej wersji Visual Studio „dorobiło się" wyszukiwarki. Jest to bardzo przydatne narzędzie pozwalające, jak sama nazwa wskazuje, w bardzo szybki sposób znaleźć klasę wśród tysięcy wierszy kodu i setek klas. Tak jak większość wyszukiwarek szuka klasy pasujące do danego ciągu znaków.

- **Find All Reference**

Opcja Find All Reference jest kolejną pomocą do nawigowania po kodzie. Dostępna jest poprzez kliknięcie prawego klawisza myszy na danym typie. Do znalezienia wszystkich plików i lokalizacji, gdzie używany jest dany typ, nie używa analizy łańcucha znaków lecz kompilatora.

5.1.2. Edycja i diagnozowanie kodu

Kiedy programista rozpoczyna swą prace zupełnie od początku ma nieograniczone możliwości. Nie wszyscy są zadowoleni z takiego punktu wyjścia, a inni wręcz przeciwnie – cenią możliwość stworzenia czegoś nowego. Pisanie kodu od podstaw coraz częściej schodzi na drugi plan zadań programistycznych. Dzisiejszymi zadaniami programistów jest raczej

modyfikacja, rozszerzanie i usprawnianie istniejącego kodu, niż konstruowanie nowych rozwiązań. Podczas pracy wielokrotnie muszą przeszukać kod, w celu zlokalizowania usterek i miejsc wymagających naniesienia poprawek. Również w tej dziedzinie Microsoft przyszedł z pomocą do programistów. Najistotniejsze właściwości zostały opisane poniżej.

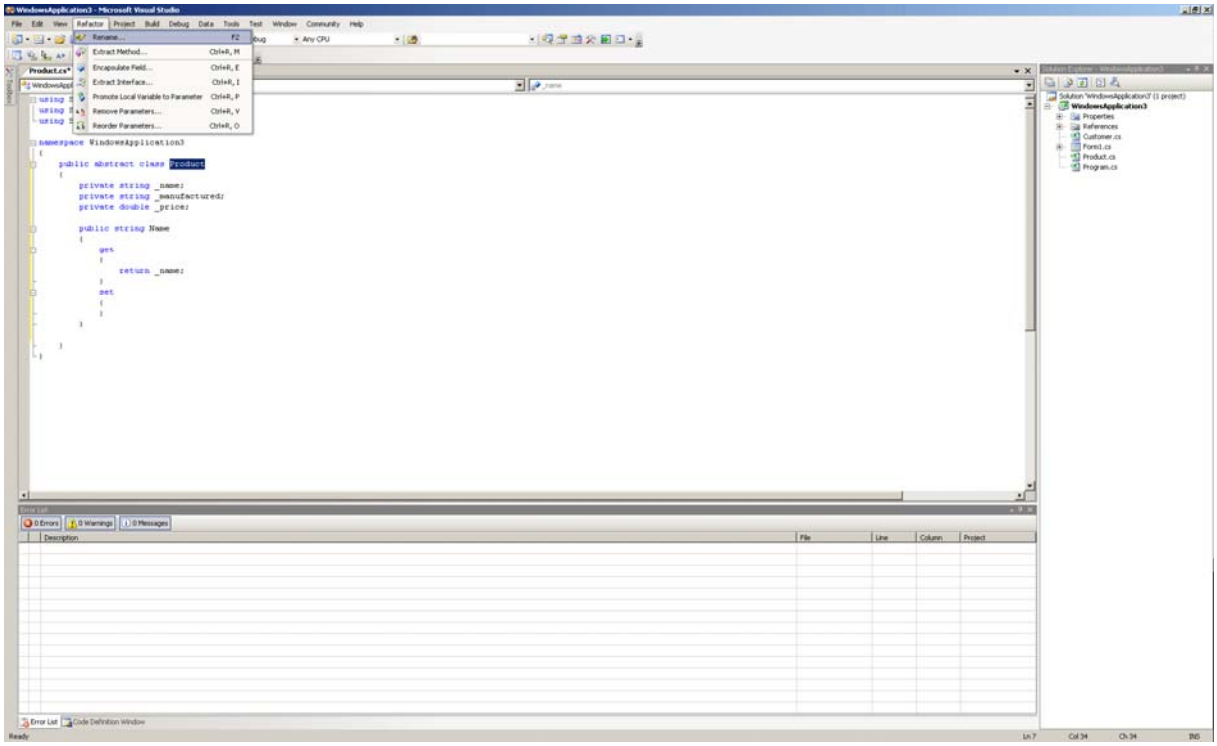
- **Refaktoryzacja**

Refaktoryzacja jest to proces wprowadzania zmian przez programistów w projekcie, które nie mają zbytnio wpływu na jego funkcjonalność. Poprawia on czytelność kodu, zwiększa możliwość ponownego wykorzystania rozwiązania lub wyklucza bezcelowe powtarzanie podobnych fragmentów kodu. „Największy problem dotyczący refaktoryzacji polega na wprowadzaniu zmian w stosunkowo stabilnej bazie kodu. Znaczne modyfikacje są dopuszczalne w początkowych fazach rozwoju programu, jednak ostatnia rzeczą, jakiej pragną członkowie zespołu w obliczu zbliżania się daty udostępnienia wersji produkcyjnej, jest wprowadzanie zmian, które mogą przyczynić się powstania nowych błędów. [1]” Ten dylemat pomaga rozwiązać edytor języka C# w Visual Studio. Udostępniono tutaj liczne opcje związane z refaktoryzacją. Przy pomocy różnych mechanizmów możemy przykładowo zmienić nazwę metody i być pewnym, iż zostaną zmienione wszystkie przypadki jej użycia. Innym przykładem może być sytuacja modyfikacji klasy, której programista chce użyć jako modelu nowej klasy. Dobrze by było aby te dwie klasy miały wspólny interfejs – możliwe to jest dzięki mechanizmom do refaktoryzacji. Pobieram po porostu istniejący interfejs, a następnie implementujemy go w nowej klasie. Silnik refaktoryzacji wykorzystuje kompilator do poprawnej i kompletnej modyfikacji kodu. Nawet komentarze są przeszukiwane w celu dokonania odpowiednich modyfikacji. Poniżej lista możliwości refaktoryzacji:

- ✓ **Rename** – zmiana nazw właściwości, metod, pól, zmiennych, etc.
- ✓ **Extract Method** – tworzenie nowych metod przy użyciu istniejących (wybranych) wierszy kodu danej metody.
- ✓ **Promote Local to Parametr** – użycie lokalnej zmiennej metody i przekształcenie jej na parametr. Narzędzie to również aktualizuje przypadki wywołania danej metody.
- ✓ **Reorder Paramteres** – zmiana kolejności parametrów w sygnaturze danej metody.
- ✓ **Remove Paramteres** – usunięcie danego parametru z metody. Właściwość ta również aktualizuje przypadki wywołania metody, skąd usuwa wartości przekazywane jako parametr.

- ✓ **Encapsulate Field** – szybkie generowanie właściwości na podstawie danego pola.
- ✓ **Extract Interface** – używa istniejącej klasy lub struktury i generuje odpowiadający im interfejs, który można zaimplementować w innych klasach.

Najczęściej stosowany sposób wywołania poleceń refaktoryzacji to korzystanie z menu *Refactor*. Dodawane ono do IDE, kiedy okno z kodem języka C# jest otwarte i aktywne. Rysunek 5.5. przedstawia działanie tego menu.



Rysunek 5.5. Działanie menu Refactor.

- **Narzędzia do wizualizacji**

Częstym przypadkiem, w aplikacjach sterowanych danymi jest zdiagnozowanie problemu. Podgląd danych w oknie debugera jest oczywiście niemożliwy. Programista jest skazany na przekopanie się przez szereg małoistotnych opcji lub przejść do okna SQL. Dzięki kolejnej właściwości Visual Studio 2005 – narzędziom do wizualizacji – może łatwo przejrzeć w tabeli zawartość zbioru danych bezpośrednio w środowisku IDE w trybie diagnozowania. Mechanizmy te udostępniają znaczącą, graficzną reprezentację zmiennych w trybie diagnozowania. Wywołuje się je z wysokości okna *DataTips* oraz okien *Watch*, *Autos* i

Locals. Dostępne są narzędzia do wizualizacji dla danych oraz formatów XML i HTML. Istnieje także pełna obsługa tworzenia własnych narzędzi tego typu i instalowania ich w środowisku IDE.

- **Dodatkowe narzędzia diagnostyczne**

Visual Studio posiada całą gamę rozszerzeń w zakresie diagnozowania. Poniższa lista opisuje krótko jeszcze kilka przydatnych, które nie zostały opisane.

- ✓ **IntelliSense w oknie Watch**

  Łatwiej dodawać elementy do okna *Watch*, wpisując nazwę zmiennej i przechodzić bezpośrednio w oknie po modelu obiektowym za pomocą *IntellSense*.

- ✓ **Zdalne diagnozowanie**

  W dużym stopniu została poprawiona usługa zdalnego diagnozowania. Wystarczy umieścić jeden plik na zdalnej maszynie, by umożliwić zdalne diagnozowanie. Dołożono także starań, aby ten mechanizm był w pełni bezpieczny.

- ✓ **DataTips**

  Już wcześniej istniała możliwość wyświetlania danych przy pomocy umieszczenia kursora myszy nad zmienną w edytorze w trybie diagnozowania. Niestety wyświetlany był tylko jeden wiersz tekstu. „*DataTips* to właściwość umożliwiająca wyświetlanie większej ilości danych i zagłębiania się w nie (podobnie jak w oknie *QuickWatch*) bezpośrednio w wyniku umieszczenia kursora myszy nad elementem. [1]”

- ✓ **Punkty śledzenia**

  Punkty przerywania (ang. break point) to nowy i usprawniony mechanizm, który za równo pozwala lepiej zarządzać nimi, jak i tworzyć własne. Punkt śledzenia to punkt przerwania, który umożliwia wykonanie niestandardowych operacji w momencie, kiedy wykonanie kodu dojdzie do tego punktu. Niestandardowe operacje polegają na wyświetleniu w oknie *Output* kominkatów, lub uruchomienie makr środowiska Visual Studio.

5.1.3. Połączenie z danymi

W platformie .NET 2.0 pojawiło się kilka rozszerzeń związanych z nawiązywaniem połączenia z danymi w postaci kontrolek i łatwiejszych w użyciu narzędzi bardziej ze sobą powiązanymi. Jest parę nowości związanymi z różnymi przestrzeniami nazw (System.Data, System.Data.SQLClient i tak dalej). Zdecydowanie jednak większość rozszerzeń dotyczy

sposobu działania Visual Studio i nowych kontrolek w zakresie wiązania danych. Poniższa lista opisuje niektóre istotne z nich.

- **Wiązanie danych bez konieczności używania kodu**

Bez względu od tego, czy programista tworzy nową aplikację bazująca na formularzach sieciowych, która łączy się bezpośrednio z bazą danych, czy trójwarstwowy program dostarczający dane do kontrolek sieciowych poprzez obiekty biznesowe i usługi sieciowe, Visual Studio pozawala zautomatyzować cały ten proces. Środowisko rozpoznaje różne źródła danych, włączając w to obiekty biznesowe, listy, bazy danych, (SQL, Oracle i inne), dane XML i usługi sieciowe.

- **Formularze Windows**

W formularzach Windows programiści mogą dołączyć do aplikacji źródło danych, a następnie używać jego tabel oraz relacji między nimi – przeciągając potrzebne elementy na formularz. Visual studio definiuje, które dane powiązane są z konkretnym kontrolkami (np. dane dotyczące daty z kalendarzem) i w jaki sposób te dane są wyświetlane na formularzu w relacji nadrzędne – podrzędne. Naturalnie programista ma możliwość przesłonięcia tych informacji.

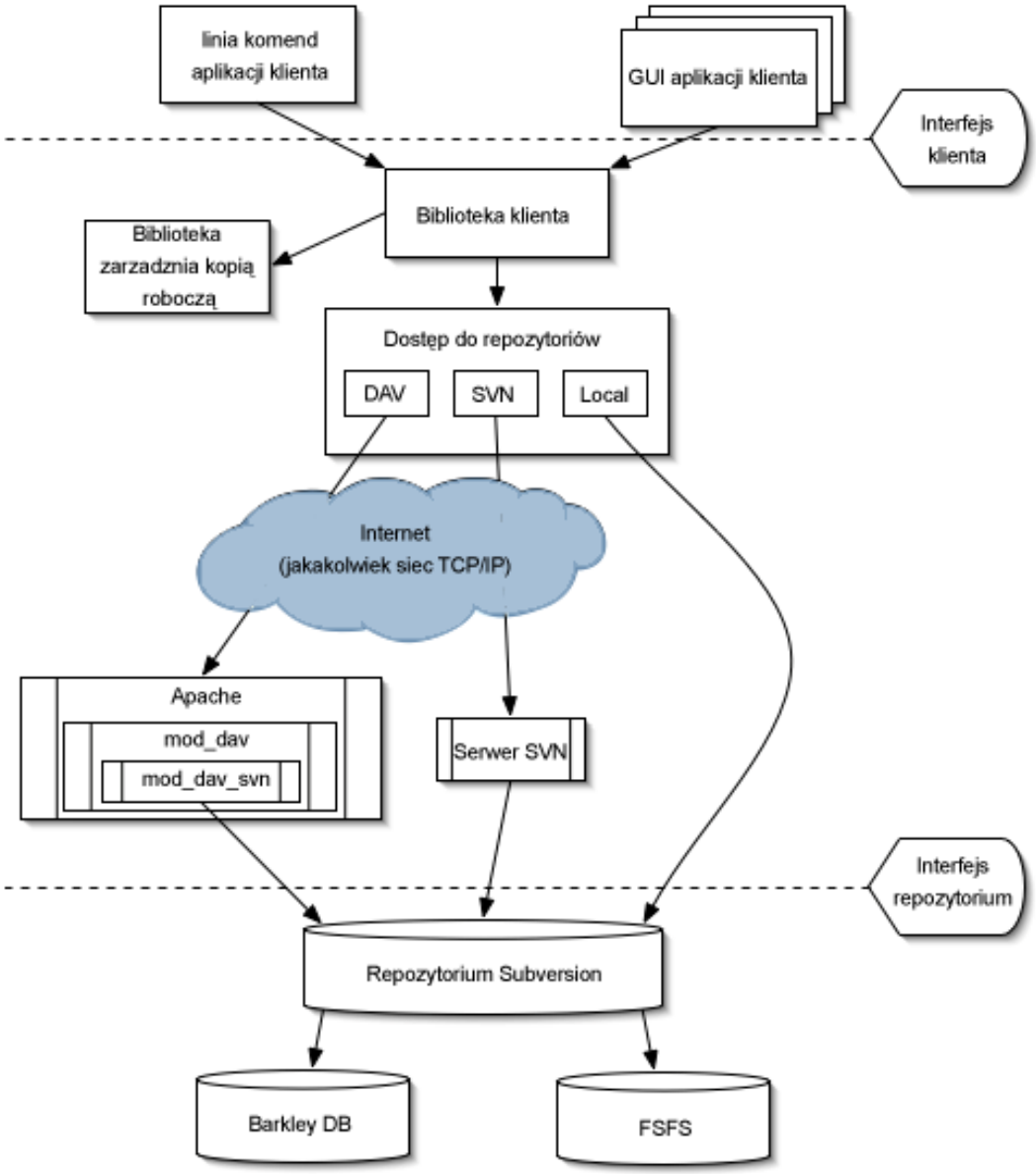
5.2. Subversion – kontrola kodu źródłowego

Cel stosowania kontroli kodu źródłowego jest stosunkowo prosty – w czasie tworzenie oprogramowania potrzebne jest centralne miejsce do przechowywania danych oraz kontrola dostępu do plików, które składają się na podstawowe elementy projektu. Inaczej mówiąc, system kontroli kodu źródłowego służy do centralnego zarządzania nie tylko plikami z kodem źródłowym, ale także innymi wieloma jednostkami bazującymi na plikach utworzonych w czasie tworzenia projektu. Te elementy mogą obejmować dokumenty wymagające spełnienia określonych warunków, schematy sieci czy plany testów. Zadania systemu kontroli kodu źródłowego można rozbić na poniższe zagadnienia:

- ✓ Bezpieczne i niezawodne centralne przechowywanie plików.
- ✓ Umożliwianie łączenia grup wersji plików w celu utworzenia „wydania”.

- ✓ Umożliwianie wielu użytkownikom jednoczesnego korzystania z tego samego pliku. Służą do tego mechanizmy przesyłania zmian (ang. check – in), zajmowania plików do edycji (ang. check– out) i scalania.
- ✓ Śledzenia zmian w pliku, ich autorów, a także czasu i przyczyn wprowadzenia zmian.

Subversion jest darmowym oprogramowaniem i obsługuje wszystkie te zadania oraz inne. Działa również w sieć, dzięki czemu może być wykorzystywany przez użytkowników na różnych komputerach. Na pewnym poziomie zdolności różnych osób do modyfikowania i zarządzania tymi samymi danymi z dowolnej lokalizacji – sprzyja współpracy. Prace mogą posuwać się szybciej bez potrzeby wprowadzania wszystkich zmian. Ponieważ są one wersjonowane, nie trzeba obawiać się utarty jakości, jeżeli ktoś się pomyli podczas wprowadzania danych, możemy je zawsze cofnąć. Niektóre systemy kontroli wersji są systemami do zarządzania konfiguracją oprogramowania. Są one wyspecjalizowane do zarządzania kodem źródłowym oraz mają wiele funkcji które są specyficzne dla rozwoju oprogramowania – znają składnie języków programowania czy dostarczają narzędzia wspomagające budowę aplikacji. Jednak Subversion nie jest jednym z takich systemów. Jest to ogólny system do zarządzania dowolną kolekcją plików czy folderów. Dla jednych mogą to być plik z kodem źródłowym dla innych czymkolwiek od listy zakupów do zbioru materiałów muzycznych.



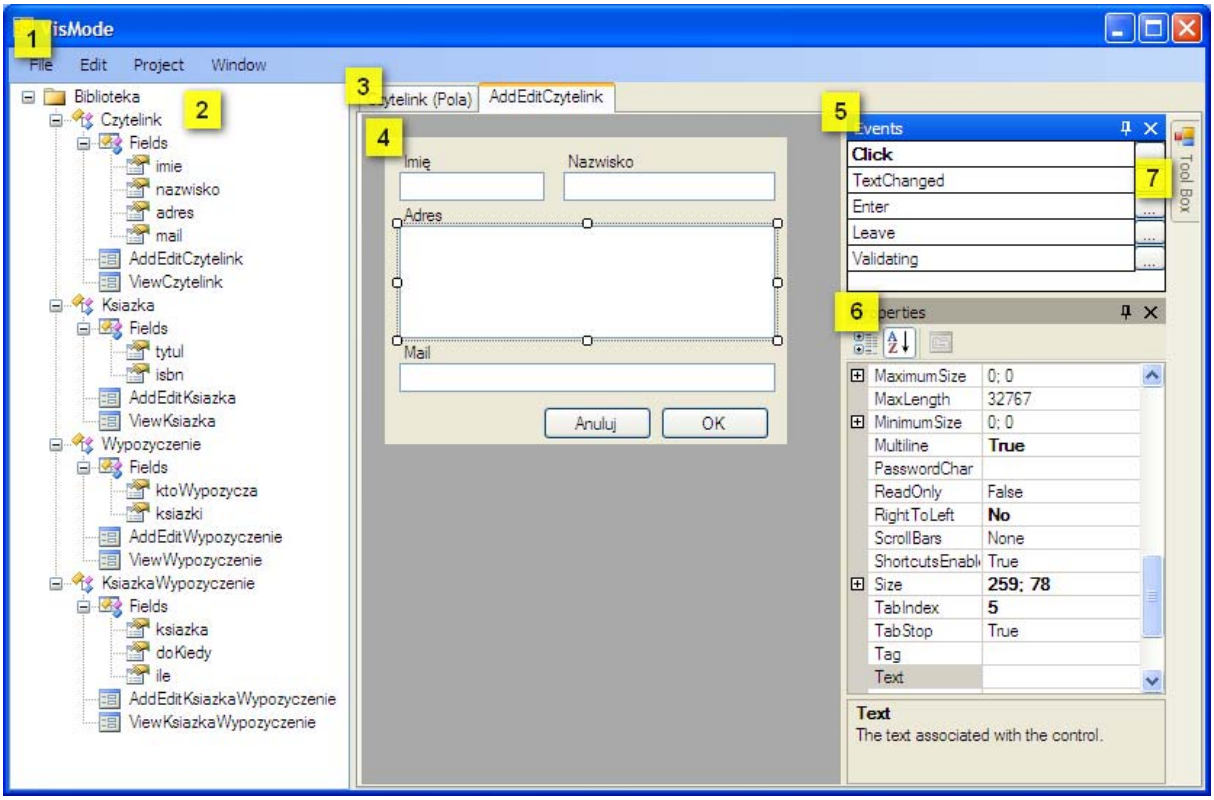
Rysunek 5.6. Architektura Subversion

6. Prototyp rozwiązania

Rozdział ten poświęcony jest najistotniejszym cechom prototypu rozwiązania jakie zostało przygotowane. Zawarte w nim są wszystkie pomysły jakie zastosowano, podział poszczególnych elementów programu, jego budowa, jak również przykład zastosowania.

6.1. Interfejs użytkownika

Główne okno aplikacji zostało przedstawione na rys. 6.1. Jest ono podzielone na szereg kontrolek, które zostaną dokładniej omówione w dalszej części pracy.



Rysunek 6.1. Główne okno aplikacji VisMode.

Opis:

1 – Menu programu

2 – Drzewo projektu

3 – Pasek zakładek

4 – Okna robocze

5 – Okno Events

6 – Okno Properties

7 – Okno Tool Box

6.1.1. Menu programu

Menu to służy, jak w większości programów, do standardowych operacji, takich jak zapis czy otwieranie wcześniej przygotowanych projektów. Posiada też funkcje takie jak eksport utworzonego schematu aplikacji do narzędzia Visual Studio 2005, czy tworzenie i dodawanie własnych wtyczek.

6.1.2. Drzewo projektu

Element ten został stworzony w celu łatwiejszego zarządzania danymi oraz strukturą programu. Możliwości, które nam daje to okno to przede wszystkim dodawanie nowych pól dla danej kategorii, jak i formularzy. Formularze do przeglądania i edycji są dodawane automatycznie dla każdej głównej kategorii. Jednym kliknięciem myszy generujemy sobie dzięki nim gotowe okno z naniesionym już kontrolkami. Wszystkie atrybuty dzięki technice Drag&Drop można przeciągać na okno projektu formularza. Dzięki zastosowaniu układu drzewa mamy szybki i czytelny dostęp do tego czego poszukujemy.

6.1.3. Pasek zakładek

Skonstruowany został w celu zwiększenia ergonomii. Dzięki zakładkom przełączamy się pomiędzy oknami edycji formularza czy też właściwości atrybutów. Ergonomia wynika tutaj z faktu, iż uniknięto konieczności otwierania wielu osobnych okien. Daje nam to bardziej czytelne i przejrzyste narzędzie. Jest to bardzo często stosowane rozwiązanie w wielu aplikacjach.

6.1.4. Okna robocze

Mamy dwa zasadnicze okna robocze. Jedno służy do projektowania formularzy – rozkładu kontrolek lub ich edytowania, drugie do edycji właściwości pól.

- **Okno projektu formularza**

W tym miejscu projektuje się wygląd formularza. Mamy możliwość pełnej edycji wszystkich elementów zawartych na karcie, jak i korzystając z narzędzia Tool Box wolno nam dodawać kolejne. Tak jak wcześniej wspomniano istnieje opcja przeciągania atrybutów z okna drzewa

projektu, a gotowy projekt rozwiązania, np. do edycji, może być wygenerowany automatycznie.

- **Okno edycji właściwości pól**

Okno to daje możliwość usuwania lub dodawania nowych pól. Jednak jego głównym zadaniem jest określenie najważniejszych cech atrybutów. Właściwości te to przede wszystkim nazwa, nazwa wyświetlana, typ danego pola czy tez jego walidacja. W oknie walidacji posługując się językiem C# możemy ustalić swoje wymagania.

6.1.5. Okno Events

Jak sama nazwa wskazuje okno to służy do obsługi zdarzeń. Określamy dzięki temu narzędziu zachowania dla poszczególnych kontrolek, jak i dla całego programu. Ze względu na używanie języka C# przy określaniu konkretnego zdarzenia, mamy pełen wpływ na to co się dzieje. Samo już okno edycji kodu ułatwia nam pracę licznymi podpowiedziami.

6.1.6 Okno Properties

Jest to analogiczne okno właściwości do tego, które możemy znaleźć w Visual Studio. Służy nam ono dokładnie do tych samych zadań, czyli zarówno do edycji wyglądu jak i cech kontrolek lub obiektów. Ilość właściwości na jaki ma wpływ to narzędzie jest zbyt duża, by opisać każda z osobna. Każdy kto choć raz miał styczność z programem Microsoftu powinien dobrze kojarzyć ten mechanizm. Dodatkową opcją jaka została stworzona jest „selection type", dostępna jedynie dla zbioru obiektów. Przy jej pomocy można stworzyć nowy obiekt lub wybrać z listy już istniejących. Przyjmuje ona następujące wartości: new – tworzenie nowej instancji obiektu, select – wybór już istniejącego elementu z kolekcji,, custom – samodzielna obsługa zdarzeń kontrolki.

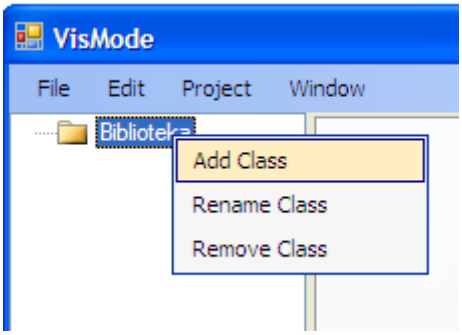
6.1.7.Okno Tool Box

Jest to lista kontrolek, które można przeciągać na okno projektu formularza. Znajdują się na niej te najpotrzebniejsze i najczęściej używane: przycisk, pole tekstowe, etykieta, pole wyboru, pole rozwijane oraz VisGrid –zmodyfikowany komponent DataGridView.

6.2. Przykładowe rozwiązanie

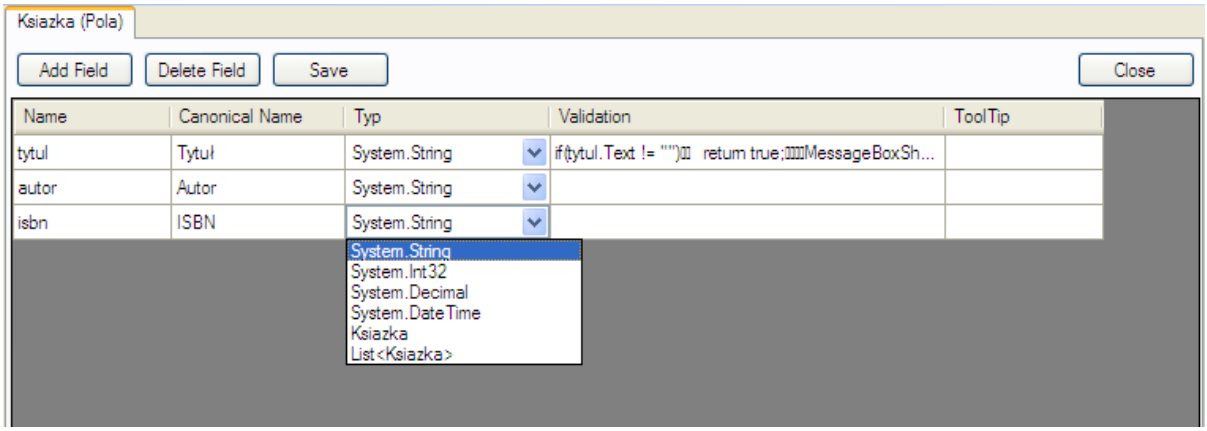
W celu przedstawienia działania naszego prototypu, przedstawimy przykładowe rozwiązanie dla systemu biblioteki. Przejdziemy przez cały proces budowy aplikacji, począwszy od stworzenia nowego projektu poprzez generowanie formularzy, aż do eksportu do środowiska Visual Studio 2005 i kompilacji.

Pierwszą czynnością jest utworzenie nowego projektu. Po uruchomieniu programu z menu File wybieramy New, w oknie które się pojawi wpisujemy nazwę projektu "Biblioteka", dodatkowo możemy w skrócie go opisać. Po zatwierdzeniu (przycisk Zakończ) zostaje wyświetlone główne okno projektu. W drzewie (po lewej stronie) pojawiła się jego nazwa. Klikając ją prawym przyciskiem myszy rozwijamy menu kontekstowe z którego wybieramy opcję "Add Class". Wpisujemy nazwę nowej klasy, w naszym przypadku będzie to "Ksiazka", pamiętajmy, że nazwy klas nie pomogą zaczynać się od cyfr oraz nie powinny zawierać znaków specjalnych oraz polskich liter. Przedstawione jest to na rysunku 6.2.

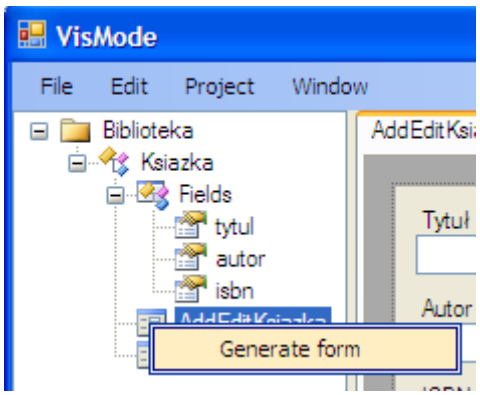


Rysunek 6.2. VisMode – tworzenie nowej klasy.

Kolejną czynnością jest dodanie atrybutów do klasy, podwójnym kliknięciem na podelement Fileds, "(double click to add)" otwieramy edytor pól, i dodajemy je tak jak zostało to przedstawione na poniższym rysunku.

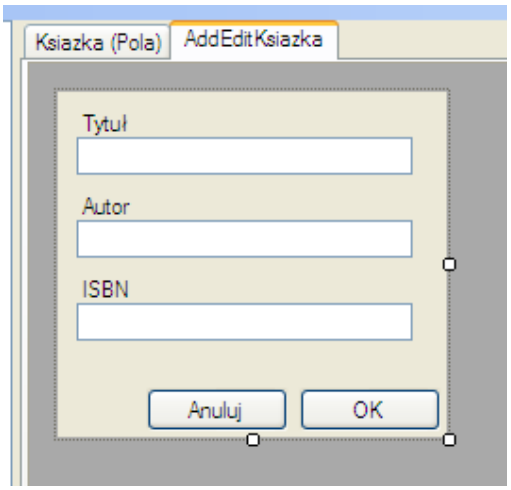


Rysunek 6.3. VisMode – dodawanie atrybutów do klasy.



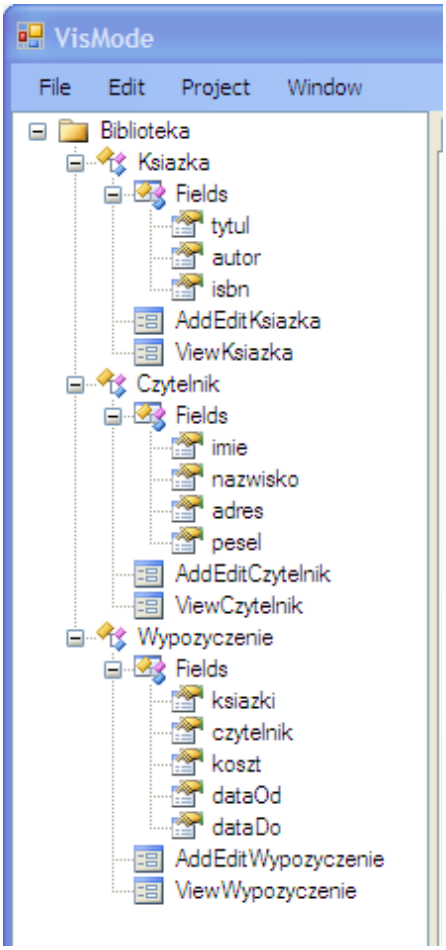
Rysunek 6.4. VisMode – generowanie formularza.

Po ustaleniu jakie informacje ma przechowywać obiekt "Ksiazka", możemy przejść do części projektowania formularza. Jak widać, podczas dodawania nowej klasy, wraz z pojawieniem się jej zostały dodane również dwa formularze. Jeden służy do dodawania i edycji obiektów klasy natomiast drugi do przeglądania kolekcji już istniejących wystąpień. Każdy z nich możemy zaprojektować oddzielnie poprzez przeciąganie kontrolek z ToolBox'a i drzewa projektu na dany formularz. My jednak skorzystamy z automatycznego generatora, jest to sposób szybszy i prostszy. Generowanie formularzy polega na otwarciu go do edycji i wybraniu "Generate form" z menu kontekstowego, tak jak zostało to przestawiona na rysunku 6.4. a rezultat wygenerowanego formularza możemy obejrzeć poniżej:



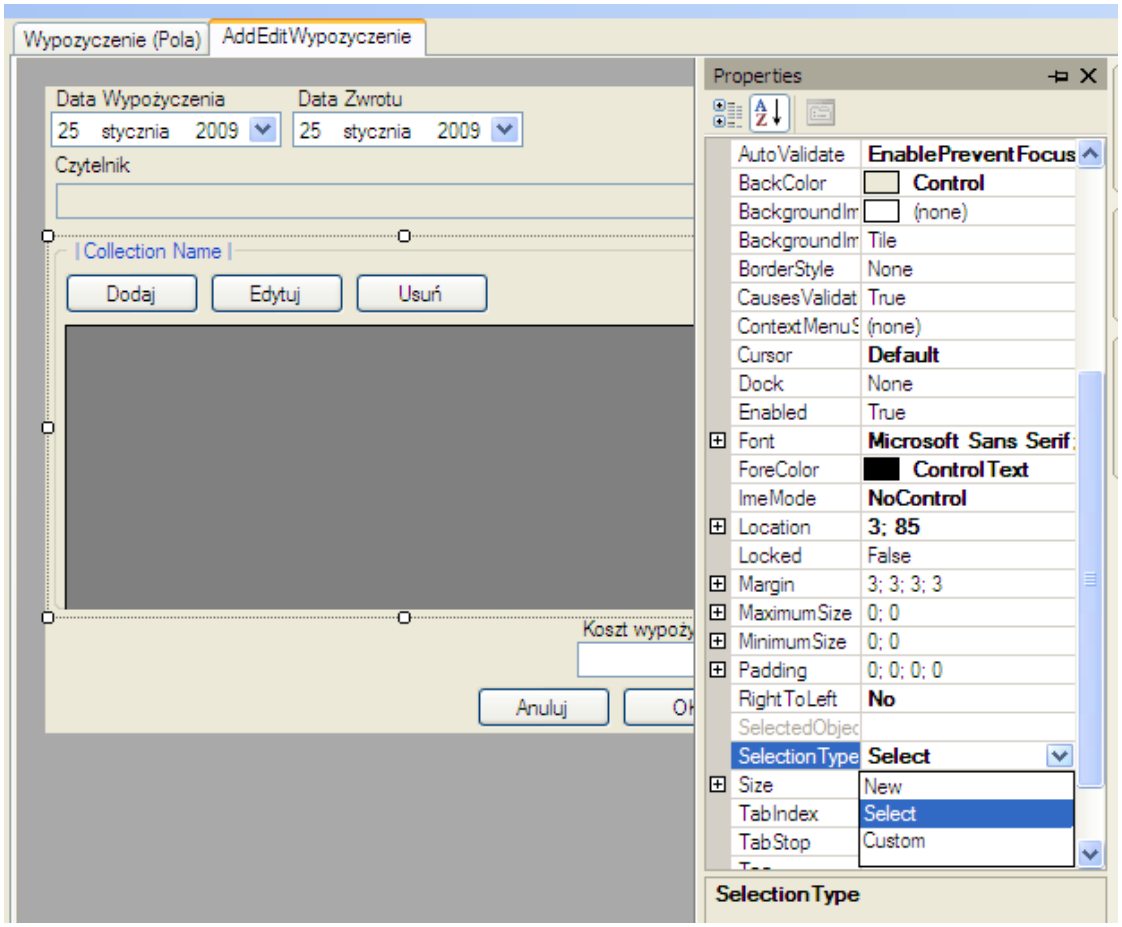
Rysunek 6.5. VisMode – formularz Ksiazka.

Analogicznie jak w powyższym przykładzie tworzymy pozostałe klasy wraz z atrybutami i generujemy do nich formularze. Struktura całego projektu została przedstawiona na rysunku 6.6.



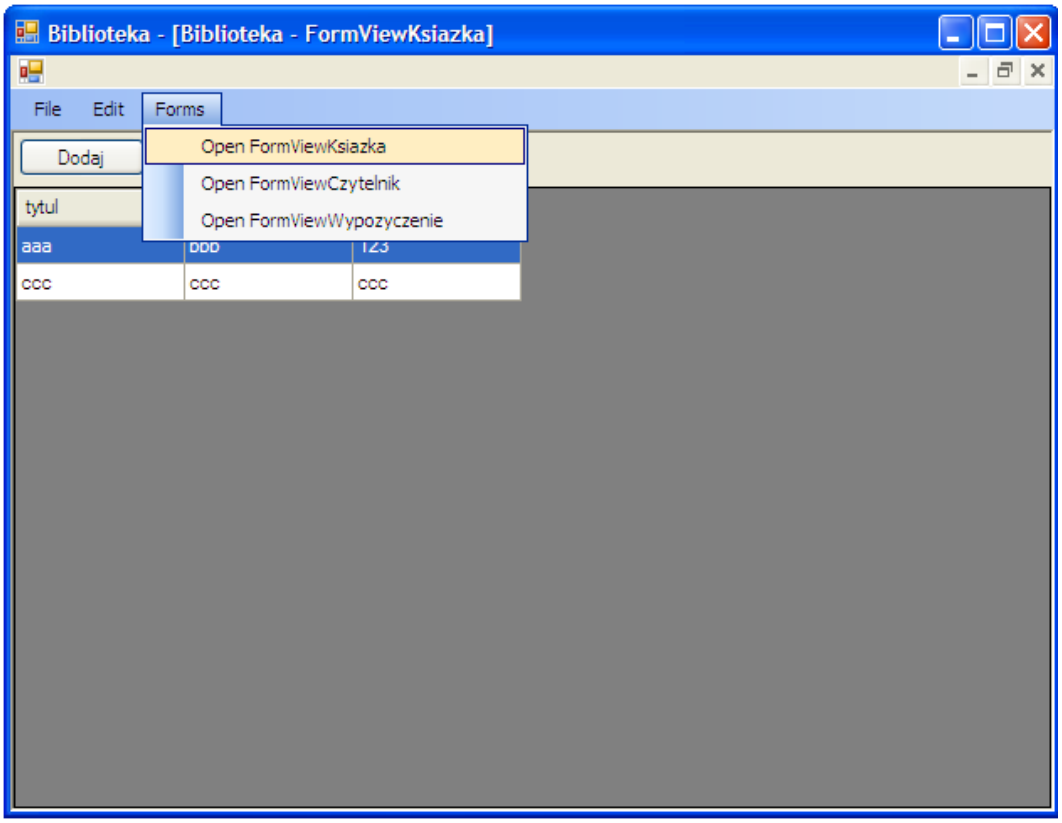
Rysunek 6.6. VisMode – struktura projektu.

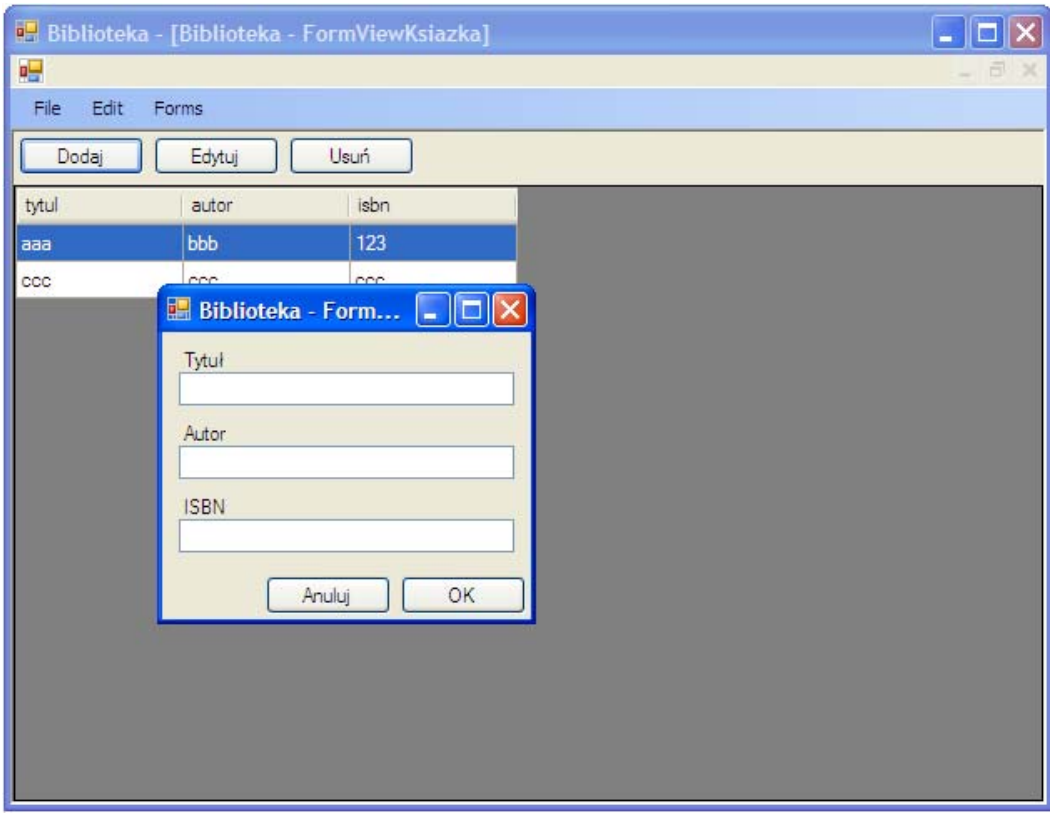
Ostatnią rzeczą jaką nam pozostaje przy tworzeniu tego projektu jest zmienienie właściwości "SelectionType" dwóch kontrolek znajdujących się na formularzu "AddEditWypozyczenie". Zmiany tej należy dokonać na kontrolkach "ksiazka" i "czytelnik" a wartość jaką należy ustawić to: Select. Zmiana tego parametru wiąże się ze zmianą zachowania kontrolki, "select" – oznacza wybór już istniejącego elementu z kolekcji (rys. 6.7).

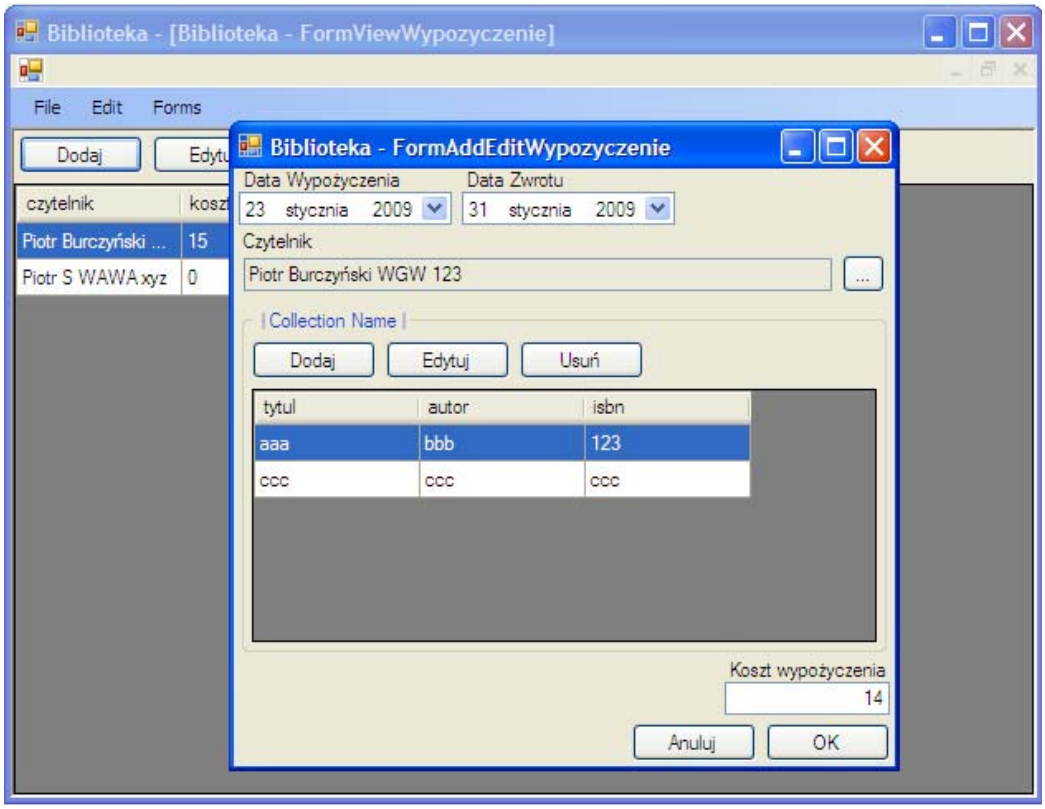


Rysunek 6.7. VisMode – zmiana właściwości SelectionType.

Teraz eksportujemy stworzony projekt do środowiska Visual Studio 2005, w którym go otwieramy, kompilujemy i uruchamiamy. Aplikacja, która została w ten sposób przygotowana, została przedstawiona na rysunku 6.8.


Biblioteka - [Biblioteka - FormViewKsiazka]
File
Edit
Forms
Open FormViewKsiazka
Open FormViewCzytelnik
Open FormViewWypozyczenie
Dodaj
tytul
aaa
ccc
ccc
ccc


Biblioteka - [Biblioteka - FormViewKsiazka]
File
Edit
Forms
Dodaj
Edytuj
Usuń
tytul
autor
isbn
aaa
bbb
123
ccc
Biblioteka - Form...
Tytuł
Autor
ISBN
Anuluj
OK



Rysunek 6.8. VisMode – aplikacja dla systemu Bilioteka.

6.3. Elementy aplikacji

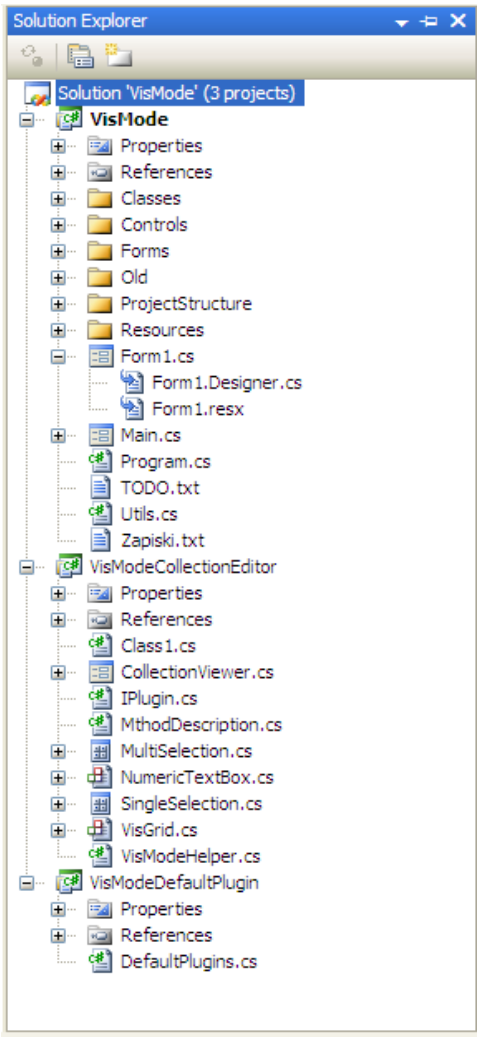
Aplikacja jest w całości napisana w języku C#, który jest częścią platformy Microsoft .NET Framework 2.0. Do stworzenia go wykorzystano narzędzie Microsoft Visual Studio 2005 oraz dwie zewnętrzne biblioteki:

- ✓ DesignSurfaceExt
- ✓ Net Docking Library for Windows Forms

Cały projekt (Solution) jest podzielony na trzy odrębne podprojekty (Projects) jak (rys. 6.9):

- ✓ VisMode (Windows Appliaction)
- ✓ VisModeCollectionEditor (Class Library)
- ✓ VisModeDefaultPlugin (Class Library)

Każdy z nich pełni odrębną rolę.



Rysunek 6.9. Podział projektu w Solution Explorer

6.3.1. VisMode

VisMode jest to główna aplikacja zawierająca podstawowe elementy programu, takie jak:

- ✓ główne okno programu,
- ✓ kontrolki do edycji formularzy projektu,
- ✓ kontrolki edycji pól klas projektu,
- ✓ generator kodu,
- ✓ walidator kodu,
- ✓ strukturę klas w, której program zapisuje zmiany dokonane przez użytkownika podczas tworzenia projektu.

Korzysta on także z zasobów dwóch pozostałych aplikacji bezpośrednio (VisModeCollectionEditor – kontrolki) i pośrednio (VisModeDefaultPlugin – funkcje rozszerzające możliwości aplikacji)

6.3.2. VisModeColectionEditor

VisModeColectionEditor to rozwiązanie biblioteki, która zawiera wspólne elementy zarówno dla aplikacji VisMode jak i dla aplikacji, która jest z niej generowana. Wspólnymi elementami dla obu aplikacji są przede wszystkim takie kontrolki jak:

- ✓ NumericTextBox
- ✓ VisGrid
- ✓ SingleSelection
- ✓ Multi selection

Kolejnym ważnym elementem tej biblioteki jest interfejs IPlugin. Interfejs ten może być wykorzystany do pisania wtyczek rozszerzających możliwości aplikacji o nowe metody, z których użytkownik korzysta podczas pisania kodu projektu.

6.3.3. VisModeDefaultPlugin

Ten niewielka aplikacja zawiera tylko jedną klasę DefaultPlugins implementującą interfejs IPlugin z apliakcji VisModeCollectionEditor. Została ona stworzona jako przykład biblioteki – wtyczki dostarczającej metody, które mogą być wykorzystane podczas pisania kodu projektu. Są to:

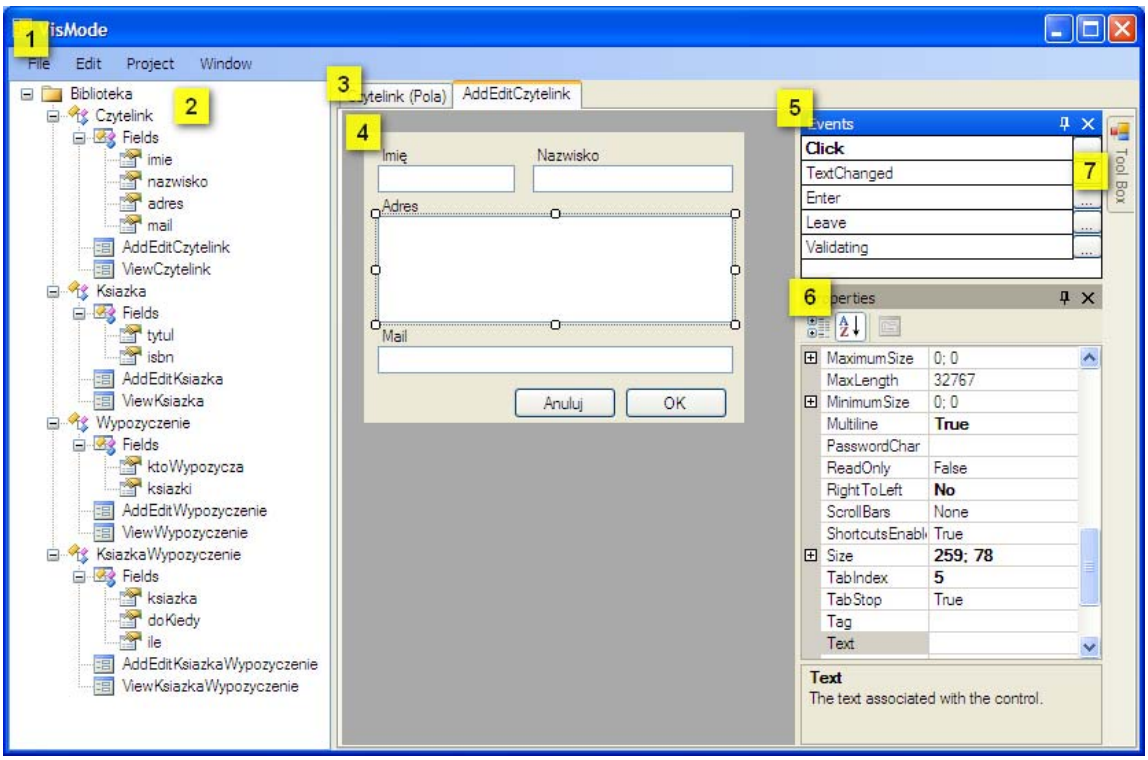
- ✓ `decimal avg(object obj, string fieldName)`
- ✓ `int count(IList obj)`
- ✓ `decimal sum(IList obj, string fieldName)`

Metody te zliczają średnią (avg) oraz sumują wartości (sum)z danego pola kolekcji . Liczą również ilość elementów w kolekcji (count).

6.4. Budowa prototypu

6.4.1 Główne okno aplikacji

Wiedząc już z jakich elementów składa się prototyp, skupy się na jego budowie. Opis należałoby rozpocząć od głównego okna aplikacji, które prezentowane było wcześniej, zostało to przypomniane na rysunku 6.10.



Rysunek 6.10. Główne okno aplikacji VisMode.

Opis:

1 – Menu programu

2 – Drzewo projektu

3 – Pasek zakładek

4 – Okna robocze

5 – Okno Events

6 – Okno Properties

7 – DockPanel for Suite for .Net 2.0

Zajmiemy się teraz opisem poszczególnych elementów oraz sposobem w jaki zostały wykonane.

Menu programu

Jest to standardowy komponent .NET, *MenuStrip* zwierający elementy *ToolStripMenuItem*, które w połączeniu ze sobą tworzą menu.

Drzewo projektu

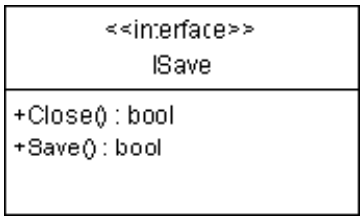
Jest to kolejny komponentem .NET – *TreeView*, wraz z jej elementem *TreeNode* tworzą strukturę drzewa, gdzie na poszczególnych węzłach w polu *Tag* zapisane są informacje na temat jaką rolę dany węzeł spełnia. Informacje te zapisywane są tylko w przypadku dwóch elementów: atrybut obiektu oraz formularz (View, AddEdit). Dane które są przetrzymywane w polu *Tag*, służą do identyfikowania jaką kontrolkę należy otworzyć, aby edytować dany element projektu. W przypadku węzła atrybut obiektu, możliwe jest jego przeciągnięcie i upuszczenie na kontrolkę *DesignSurfaceExt,* będącą częścią kontrolki *EditForm*. Skutkuje to dodaniem nowej kontrolki do projektowanego formularza, jej rodzaj zależy od typu jaki dany węzeł przechowuje w polu *Tag*. Dodatkowo dla każdego elementu drzewa wyświetlana jest krótka podpowiedź (*ToolTip*). W celu jej wyświetlenia wystarczy przesunąć kursor myszy na dany element i odczekać krótką chwilę. Okno podpowiedzi jest programowo przesuwane tak aby pokazywało się na końcu ogonka kursora, a nie jak jest domyślnie na jego początku co powoduje, że część kursora przysłania tekst podpowiedzi.

Pasek zakładek

Każda z kontrolek która jest umieszczana na zakładce (*EditFields*, *EditForms*) implementuje interfejs *ISave*. Dzięki niemu możemy poinformować daną kontrolkę, że chcemy ją zapisać lub zamknąć. Wyświetli on nam odpowiednie zapytanie do użytkownika o ewentualną akceptację zachowanych zmian. Mechanizm ten jest wykorzystywany w dwóch miejscach, w chwili zamknięcia zakładki (skrót Ctrl+W, lub z menu Window > Close Active Window) lub aplikacji. W obydwu przypadkach wywoływana jest metoda *Close()*. W zależności jaka wartość zostanie przez nią zwrócona, dana zakładka jest zamykana bądź nie. Poniższy fragment kodu przedstawia zastosowanie interfejsu *ISave*:

```
((Classes.ISave)projectCtrl.ActiveTab.Controls[0]).Close()
```

Listing 6.1. Fragment kodu ISave



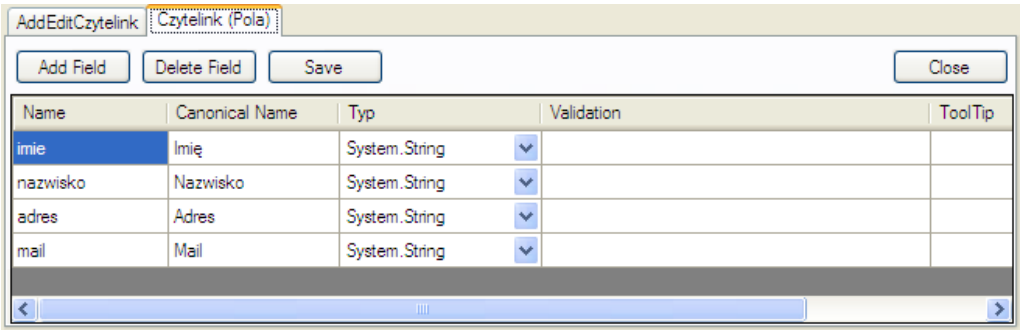
Rysunek 6.11. – Metody interfejsu ISave.

Okna robocze

Jak już wspomniano mamy dwa okna robocze – do projektowania formularzy i do edycji atrybutów. Poniżej omówimy ich działania od strony programistycznej.

- **Kontrolka *FieldsEditor* – edycja pól klas**

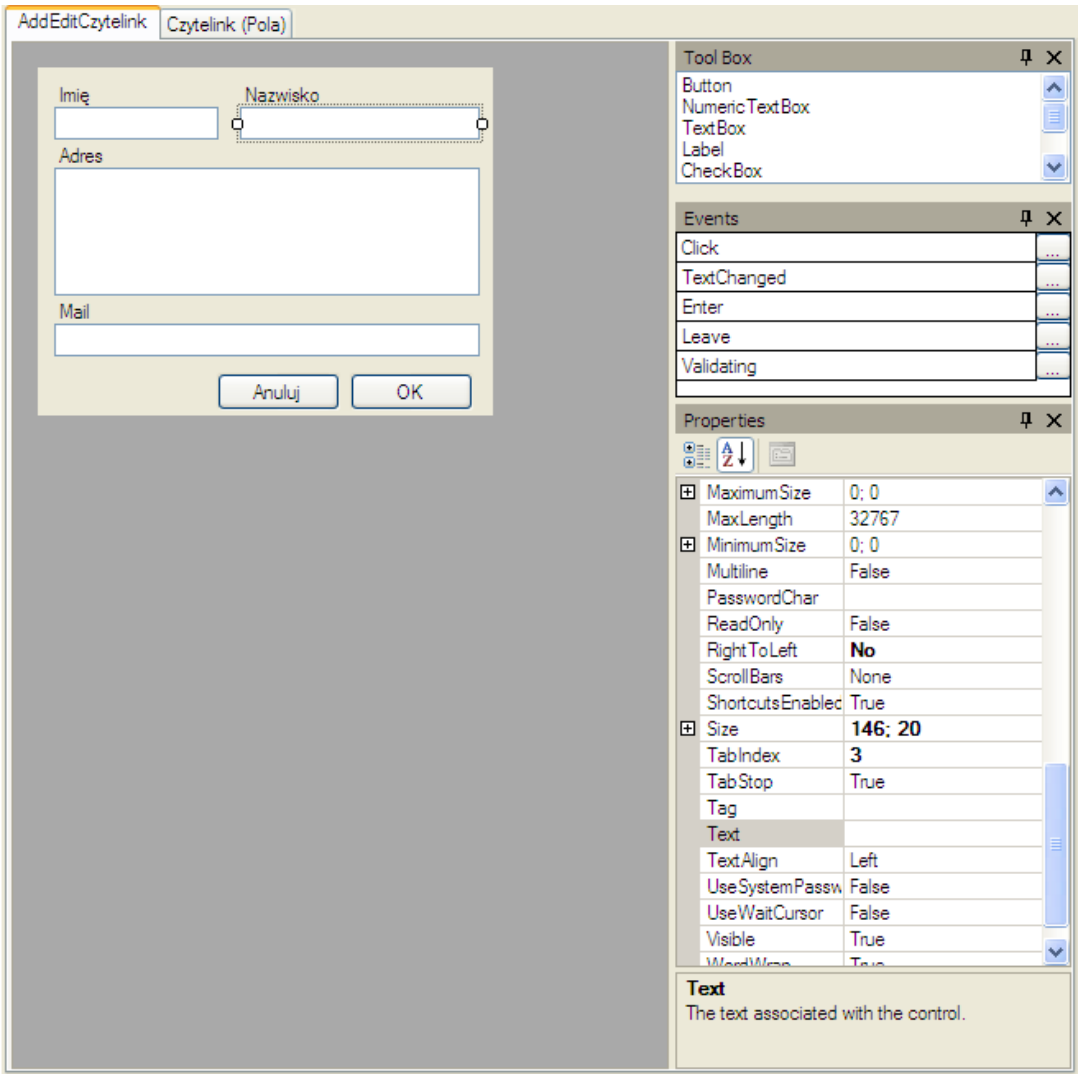
Rysunek 6.12 przedstawia ów kontrolkę.



Rysunek 6.12. – Kontrolka FieldEditor.

Do jej źródła (`public List<FieldInfo> DataSource`) podpinana jest kolekcja pól. W pierwszej kolejności zostaje ona sklonowana w taki sposób, aby edytując jej elementy nie modyfikować oryginału, ponieważ kolekcja jest przekazywana jako referencja. Następnie klon zostaje podpięty do kontrolki *DataGridView*, która to w wygodny sposób pozwala na jej edytowanie. W momencie wydania przez użytkownika polecenia zapisu, sklonowana kolekcja jest przepisywana do oryginalnego źródła.

- **Kontrolka *EditForm* – edycja formularzy**

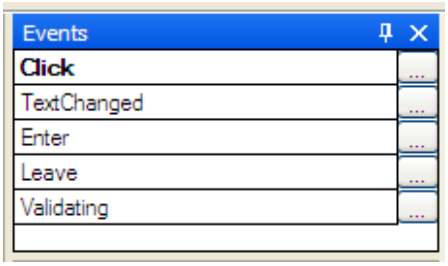


Rysunek 6.13. – Kontrolka EditForm.

Utworzenie tego elementu było najbardziej czaso- i pracochłonną czynnością. Wynikało to z potrzeby zbudowania kontrolki, która dałaby użytkownikowi łatwe i wygodne narzędzie do projektowania (tj. dodawania i rozmieszczania elementów) poszczególnych formularzy. Początkowo próbowaliśmy ową kontrolkę stworzyć sami, jednak mimo poprawnego działania, nie spełniała ona w pełni naszych oczekiwań. Stwierdziliśmy, iż dalszy jej rozwój pochłonąłby zbyt duże nakłady czasu. Z tej przyczyny zdecydowaliśmy się na wykorzystanie gotowego rozwiązania. Z pomocą przyszła biblioteka *DesignSurfaceExt*. Dzięki jej zastosowaniu otrzymaliśmy nie tylko gotowy mechanizm projektowania formularzy, ale również mechanizm cofania zmian (*undo/redo*), edycji (*cut/copy/paste/delete*) oraz wyrównywania elementów formularza do krawędzi (*SnapLines*). Biblioteka

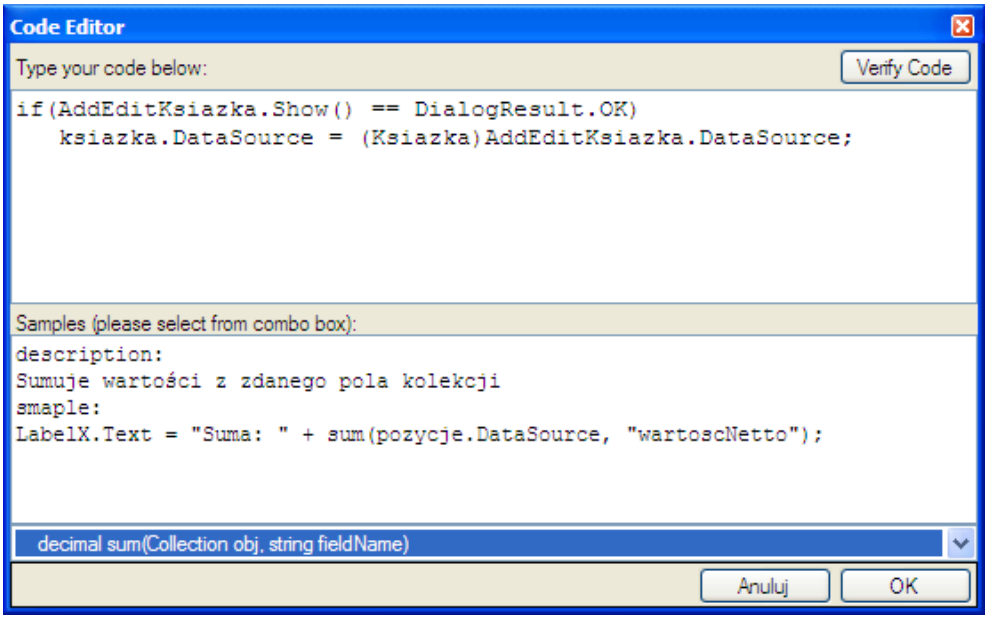
DesignSurfaceExt jest rozszerzeniem (dziedziczy) klasy *DesignSufce*, która natomiast jest częścią .NET Framework 2.0. Bezpośrednio nie wprowadza takich udogodnień jak porządkowanie kolejności przełączanie klawiszem TAB między kontrolkami (TabOrder), mechanizmu cofania i powtarzania zmian (UndoEngine), czy możliwości wyrównywania kontrolek względem siebie (Grid Alignment).

Okno Events



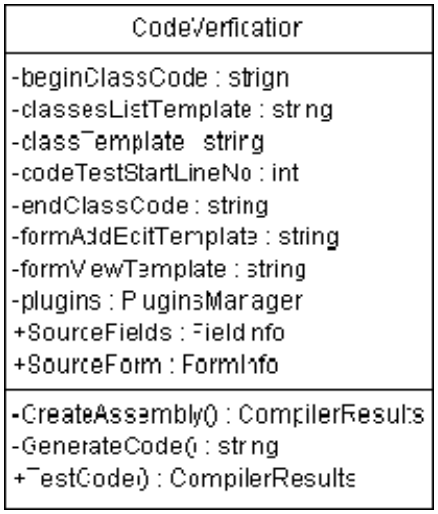
Rysunek 6.14. – Kontrolka EventGrid.

Lista na rysunku 6.14 przedstawia tylko wybrane zdarzenia kontrolki *EventGrid*. W celu zaprogramowania zdarzenia należy wybrać przycisk po prawej stronie od jego nazwy. Spowoduje to otwarcie formularz Code Editor (rys. 6.15).



Rysunek 6.15. – Okno formularza CodeEditor.

Jest on podzielony na dwie części. W górnej użytkownik wpisuje swój kod, w dolnej może obejrzeć przykłady kodu z wykorzystaniem konkretnych właściwości (properties) wybranej kontrolki oraz zastosowanie formuł. Kod wpisywany przez użytkownika jest dokładnie językiem programowania C# w wersji 2.0. Osoba korzystająca z *CodeEditor* powinna znać chociażby jego podstawy. Z pomocą przychodzi tu również walidator kodu, który sprawdza czy wpisany kod jest poprawnie interpretowany przez kompilator. Za weryfikację kodu odpowiedzialna jest klasa *CodeVerfication* (rys. 6.16). W chwili gdy użytkownik zażąda sprawdzenia kodu, tworzona jest nowa instancja klasy. Następnie do jej pola (ang. *filed*) *SourceForm* podpinana jest informacja o formularzu (*FormInfo*), w którym edytowany kod się znajduje. Teraz aby przetestować kod wywoływana jest funkcja *TestCode(string)*, a jej parametrem jest kod użytkownika.



Rysunek 6.16. – Struktura klasy CodeVerification.

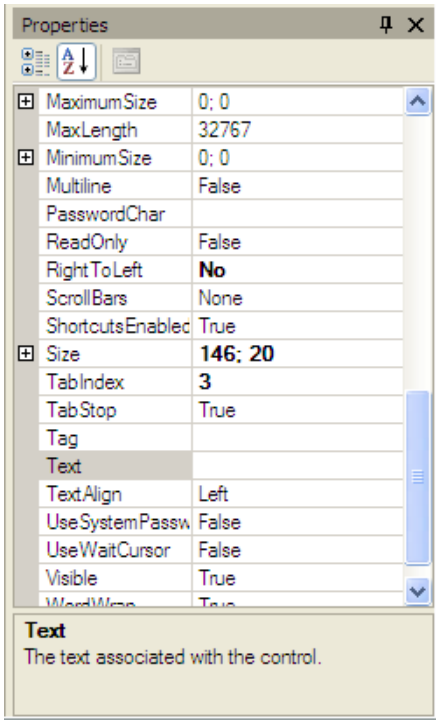
Na podstawie informacji z *FormInfo* dodatkowo generowany jest kod z informacjami o innych obiektach – kontrolkach, które dany formularz zawiera, oraz podstawowy (sama klasa bez ciała) kod o pozostałych formularzach zawartych w projekcie. Tak wygenerowany kod jest przekazywany do prywatnej funkcji *CreateAssembly(string),* która wykorzystuje wbudowaną funkcjonalność w .NET Framework do kompilacji kodu (CSharpCodeProvider). Poniższa linia kodu źródłowego przedstawia w jaki sposób otrzymujemy wynik o poprawności kompilacji.

```
CompilerResults results =

compiler.CompileAssemblyFromSource

(compilerParams,strRealSourceCode);
```

Listing 6.2. Fragment kodu przedstawiający wynik poprawności kompilacji

Klasa *CompilerResults* zawiera zestaw błędów jakie powstały podczas kompilacji. Informacje te to lokalizacja, czyli numer linii oraz kolumna, w której kompilator zauważył błąd wraz z opisem. Lokalizacja błędu jest dodatkowo poddawana odpowiedniemu przesunięciu, tak aby zwracana informacja była dokładnym wskazaniem błędu w kodzie użytkownika, a nie w wygenerowanym.

Okno Properties

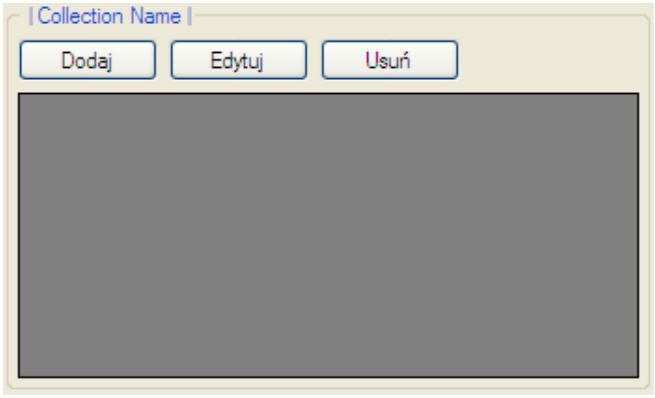


Rysunek 6.17. – Kontrolka properties.

Jest to standardowy obiekt .NET Framework. Rozbudowany jest dodatkowo o właściwość *SelctionType,* w skład której wchodzą dwie kontrolki projektu *VisModeCollectionEditor* –

SingleSelection i *MultiSelection*. Współdzielone są pomiędzy program VisMode oraz aplikacje z nich powstające.

Rysunek 6.18. – Kontrolka SingleSelection.



Rysunek 6.19. – Kontrolka MultiSelection.

W pierwszym przypadku potrzebne jest ustalenie przez użytkownika parametrów takich jak chociażby położenie i wielkość. W drugim przypadku są wykorzystywane do tego, do czego zostały tak naprawdę stworzone, czyli wyboru obiektu z kolekcji. Różnica pomiędzy jedną, a drugą kontrolką polega na tym, iż pierwsza pozwala wybrać tylko jeden obiekt, a druga pozwala na wybranie wielu elementów. Natomiast to, co mają ze sobą wspólnego to jedna z właściwość Selection Type, która może przyjąć jedną z trzech wartości: New, Select, Custom. Właściwość ta jest dość ważnym elementem, ponieważ decyduje o tym czy dany obiekt, który ma być wybrany, będzie pochodził z istniejącej już kolekcji (opcja Select), czy zostanie utworzona jego nowa instancja (opcja New) lub czy zdarzenie wyboru zostanie obsłużone przez użytkownika indywidualnie (opcja Custom).

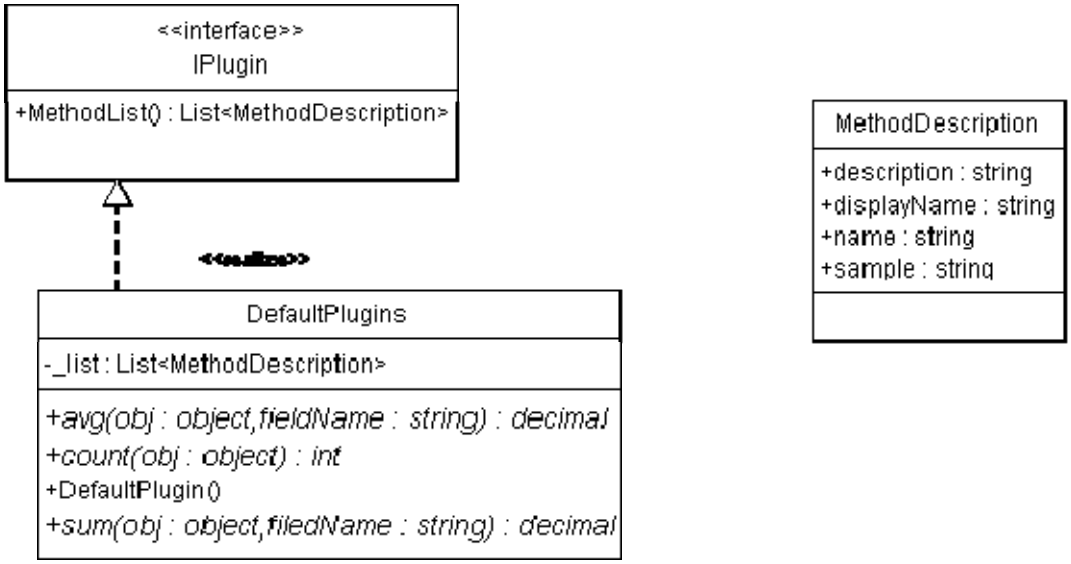
DockPanel Suite for .NET 2.0

DockPanel Suite jest darmową biblioteką dzięki której mogliśmy wprowadzić mechanizm ukrywających się okien, taki jak został zastosowany w Visual Studio. Biblioteka nie wnosi żadnej większej funkcjonalności do naszej aplikacji, jednak poprawia jej estetykę oraz ułatwia poruszanie się po większej ilości okien, które wyświetlane jednocześnie byłyby bardzo małe i nieczytelne.

6.4.2. Inne komponenty prototypu

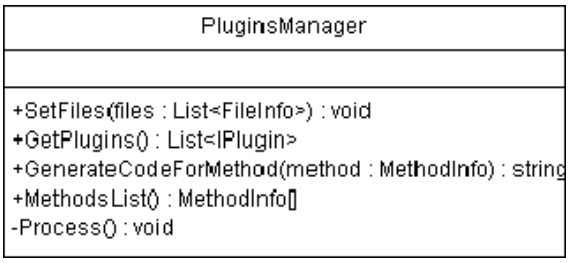
Wtyczki

„Wtyczka (ang. plug-in) to dodatek do istniejącego programu rozszerzający jego możliwości lub automatyzujący żmudne czynności. [12]” Możliwości jakie dają pluginy w naszej aplikacji to rozszerzanie jej o nowe formuły, czyli funkcje z których użytkownik może skorzystać podczas implementacji kodu (w oknie Code Editor). Do zarządzania wtyczkami powstała klasa *PluginsMenager*. Jest ona wykorzystywana zarówno podczas generowanie kodu jaki i jego walidacji. Mechanizm sam w sobie jest dość prosty. W celu przygotowania plugin'u, który będzie rozszerzał aplikacje o nowe formuły wystarczy stworzyć bibliotekę. Będzie ona implementowała nasz interfejs IPlugin oraz dostarczała zestaw funkcji, które będą statyczne oraz miały zasięg publiczny.



Rysunek 6.20. – Interfejs IPlugin oraz przykładowa wtyczka.

Jak widzimy na załączonym diagramie klas (rys. 6.20), musimy jeszcze zaimplementować metodę *MethodList*. Zwraca ona listę, która zawiera spis nazw metod, ich składnię wywołania oraz przykładowe użycie. Dzięki takiej liście jesteśmy w stanie w łatwy sposób zaprezentować użytkownikowi jakie funkcje ma do swojej dyspozycji oraz w jaki sposób może je wykorzystać. Jak już wcześniej było wspomniane do zarządzanie wtyczkami służy nam klasa *PluginsMenager*. Dzięki jej zastosowaniu jesteśmy w stanie szybko i łatwo określić lokalizację bibliotek – plugin'ów (rys. 6.21).

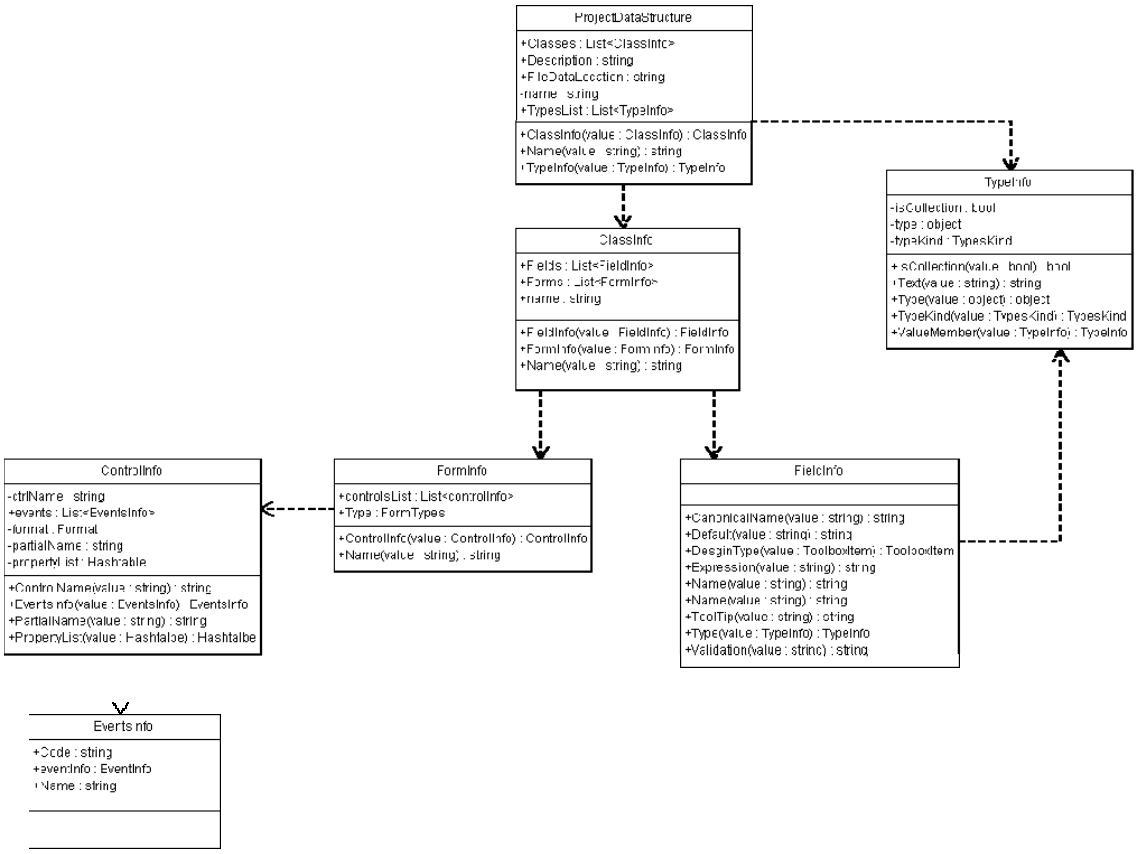


Rysunek 6.21. – Struktura klasy PluginsManager.

Struktura danych projektowych

Struktura ta została stworzona w celu przetrzymywania danych na temat projektu kreowanego przez użytkownika. Zawiera ona informacje na temat tworzonych obiektów, klas (ClassInfo), atrybutów klas (FieldInfo), formularzy AddEdit/View (FormInfo), kontrolek które użytkownik dodał na formularze (ControlInfo) oraz zdarzeń, które są oprogramowane do danej kontrolki (EventInfo). Dokładna struktura znajduje się na załączonym diagramie (rys. 6.22).

Do jej trwałego zapisu na dysku skorzystaliśmy z mechanizmu serializacji.

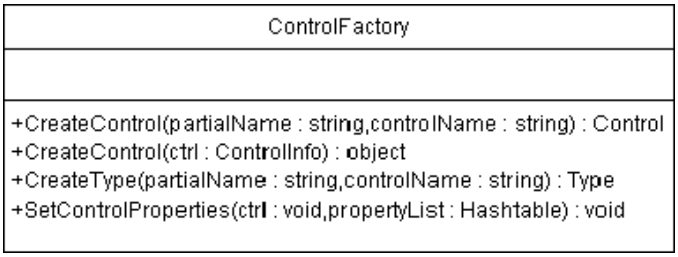


Rysunek 6.22. – Struktura danych projektowych.

Serializacja

„Serializacja – w programowaniu komputerów proces przekształcania obiektów, tj. instancji określonych klas, do postaci szeregowej, czyli w strumień bajtów, z zachowaniem aktualnego stanu obiektu. Serializowany obiekt może zostać utrwalony w pliku dyskowym, przesłany do innego procesu lub innego komputera poprzez sieć. Procesem odwrotnym do serializacji jest deserializacja. Proces ten polega na odczytaniu wcześniej zapisanego strumienia danych i odtworzeniu na tej podstawie obiektu klasy wraz z jego stanem bezpośrednio sprzed serializacji. [11]” W naszym przypadku jest to bardzo wygodny i szybki sposób na zapisanie dużej ilości danych, bez potrzeby tworzenia skomplikowanej struktury bazy czy jakiejkolwiek innej. Serializacja ma również swoje ograniczenia. Nie pozwala ona w bezpośredni sposób na zapisywanie informacji o takich obiektach jak formularz czy kontrolka. Zastosowano więc rozwiązanie, które zapisuje wszystkie serializowalne atrybuty obiektu oddzielnie, poprzez sprawdzanie czy dany atrybut jest serializowalny i dopisanie go do kolekcji, dokładnie *Hashtable*. Tak samo przy odtwarzaniu tak zapisanych danych trzeba wszystkie atrybuty

danej kontrolki przywrócić do pierwotnego stanu. Do tego zadania została stworzona klasa *ControlFactory*. Pobiera ona informacje o zapisanych atrybutach, zwraca natomiast kontrolkę z jej wszystkimi ustawionymi właściwościami (rys. 6.23).

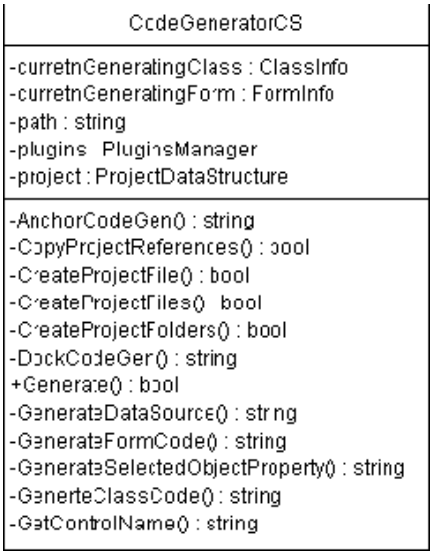


Rysunek 6.23. – Struktura klasy ContorlFactory.

6.5. Generowanie kodu i jego struktura

Generowanie kodu

Po zakończeniu procesu projektowania, eksportujemy projekt zgodny z Visual Studio 2005. Możemy go tam poddać dalszej obróbce lub po prostu skompilować i uruchomić. Za wygenerowanie kodu odpowiedzialna jest klasa *CodeGeneratorCS*. Oprócz generowania kodu jest odpowiedzialna również za stworzenie struktury katalogów, kopiowania bibliotek i dodawania do nich odpowiednich referencji. Odpowiedzialna jest również za stworzenie wszystkich plików z kodem źródłowym jak i sam plik projektu.



Rysunek 6.24. – Struktura klasy CodeGeneratorCS.

Proces generowania rozpoczyna się wybierając z menu File > Export > Visual Studio 2005. W tym momencie tworzona jest nowa instancja klasy *CodeGeneratorCS*. W konstruktorze przekazywana jest struktura projektu, a wywoływana metoda *Generate(string path)* pozwala wskazać miejsce docelowe dla tworzonego projektu.

```
CodeGeneratorCS cgen = new
CodeGeneratorCS(projectCtrl.DataSource);

cgen.Generate("D:\\VisModeProjekt\\");
```

Listing 6.3. Fragment kodu przedstawiający instancje klasy CodeeneratorCS

Kolejność zdarzeń wykonywanych w metodzie *Generate* jest następująca:

- ✓ tworzenie folderów projektu (*CreateProjectFolders()*)
- ✓ kopiowanie bibliotek (*CopyProjectReferences()*)
- ✓ tworzenie pliku projektu (Project.csproj) i na końcu (*CreateProjectFile()*)
- ✓ tworzenie wszystkich plików z kodem źródłowym (*CreateProjectFiles()*)

Metoda *CreateProjectFiles()* jest najważniejszym ogniwem. Jest odpowiedzialna za generowanie kodu:

- ✓ głównego formularza
- ✓ formularzy BaseForm, BaseAddEditForm, BaseViewForm
- ✓ klas
- ✓ formularzy stworzonych przez użytkownika

Kod głównego formularza jest już częściowo wygenerowany, dodajemy do niego tylko elementy menu umożliwiając otwieranie okien do przeglądania kolekcji obiektów (*View*) oraz podmieniając nazwę przestrzeni (*namespace*) na nazwę projektu. W formularzach BaseAddEditForm oraz BaseViewForm tak jak poprzednio podmieniana jest tylko nazwa namespace, cała reszta pozostaje bez zmian. Nie dotyczy to jednak formularza BaseForm, to w jego ciele znajdują się wszystkie instancje formularzy:

```
protected static FormAddEditCzytelnik AddEditCzytelink =
new FormAddEditCzytelnik();
protected static FormViewCzytelnik ViewCzytelink =
new FormViewCzytelnik();
protected static FormAddEditKsiazka AddEditKsiazka =
new FormAddEditKsiazka();
protected static FormViewKsiazka ViewKsiazka =
new FormViewKsiazka();
```

Listing 6.4. Fragment kodu klasy BaseFrom przedstawiający instancje formularzy

oraz obiektów:

```
protected static List<Czytelnik> Czytelink =
new List<Czytelnik>();
protected static List<Ksiazka> Ksiazka = new List<Ksiazka>();
```

Listing 6.4. Fragment kodu klasy BaseFrom przedstawiający instancje obiektów

Istotą uwagi jest jeszcze klasa (*VisModeDB*), służy jako obiekt do serializacji i deserializacji, czyli zapamiętywania wszystkich zmian jakie dokonał użytkownik przy pracy z wykreowaną aplikacją:

```
[Serializable]
public class VisModeDB
{
    public List<Czytelink> Czytelnik;
    public List<Ksiazka> Ksiazka;
}
```

Listing 6.5. Fragment kodu klasy VisModeDB przedstawiający serializację

do zapisywania i odczytywania danych generowane są metody:

```
public static void SaveData();

public static void LoadData();
```

Listing 6.6. Fragment kodu klasy BaseFrom przedstawiający zapis i odczyt danych

Ostatnim elementem jest generowanie odwołań do formuł, które dostarczane są przez mechanizm plugin'ów:

```
public System.Decimal

avg(System.Object obj, System.String fieldName)

{

Return

VisModeDefaultPlugin.DefaultPlugins.avg(obj, fieldName);

}

public System.Decimal

sum(System.Object obj, System.String fieldName)

{

return

VisModeDefaultPlugin.DefaultPlugins.sum(obj, fieldName);

}
```

Listing 6.7. Fragment kodu przedstawiający generowanie odwołań do formuł

Po wygenerowaniu bazowych formularzy generowany jest kod klas – obiektów:

```
[Serializable]
public class Ksiazka
{
private System.String _tytul;
public System.String tytul
{
get { return _tytul; }
set
{
_tytul = value;
}
}
private System.String _isbn;
public System.String isbn
{
get { return _isbn; }
set
{
_isbn = value;
}
}
(…)
}
```

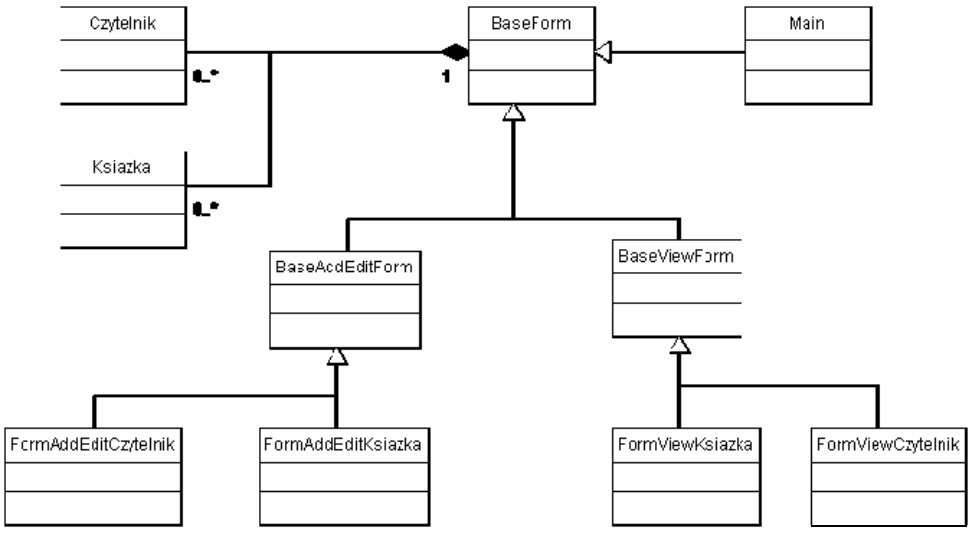
Listing 6.8. Fragment kodu klas obiektów

Na samym końcu generowany zostaje kod formularzy służących do dodawania i przeglądania elementów stworzonych przez wygenerowaną aplikację. Niestety kod tych formularzy jest zbyt obszerny, aby umieścić przykład w tej pracy. Wspomnieć tylko można, iż zawiera on

wszystkie kontrolki, które zostały utworzone przez użytkownika w czasie projektowania wraz z odpowiednio ustawionymi właściwościami oraz kod zdarzeń do ich obsługi.

Struktura wygenerowanego kodu

Struktura aplikacji, która została wygenerowana jest przedstawiona na diagramie klas (rys. 6.25). Zrozumienie tego schematu na pewno w dużej mierze ułatwi ewentualną dalszą pracę nad wygenerowaną aplikacją.



Rysunek 6.25. – Struktura wygenerowanego kodu.

Jak można zauważyć klasa BaseForm jest głównym elementem, po której dziedziczą wszystkie pozostałe formularze. Dzieje się tak, ponieważ zawiera ona definicje kolekcji obiektów (tj. *Czytelnik, Ksiazka*) oraz wszystkich formularzy jakie znajdują się w aplikacji. Rozwiązanie to zostało stworzone dla wygody użytkownika projektującego, tak aby nie musiał tworzyć za każdym razem nowej instancji formularza lub mógł w łatwy sposób odwołać się do kolekcji danego obiektu. Do przetrzymywania danych posłużył mechanizm serializacji (*XmlSerializer*). Wszystkie kolekcje przed zapisaniem są przepisywane do wewnętrznej klasy *VisModeDB*, a następnie zapisywane na dysku w postaci pliku XML. Zapisanie danych następuje po wywołaniu metody *SaveData()*. Jest ona automatycznie wywoływana podczas zamykania aplikacji. Analogicznie działa metoda *LoadData()*, która odczytuje wcześniej zapisane dane i jest wywoływana podczas uruchamiania aplikacji. BaseForm zawiera również metody, które ułatwiają komunikację z użytkownikiem

końcowym poprzez okna dialogowe – *MessageBoxShow(string text)* lub *MessageBoxYesNo(string text)*. Wyświetlają one komunikaty informacyjne lub pozwalają zadać pytanie, na które można odpowiedzieć twierdząco bądź przecząco. Ostatnim omówionym elementem tej klasy jest odwołanie się do formuł dołączanych jako wtyczki. Przykładowy kod został przedstawiony poniżej.

```
public System.Decimal
avg(System.Object obj, System.String fieldName)
{
return
VisModeDefaultPlugin.DefaultPlugins.avg(obj, fieldName);
}
public System.Decimal
sum(System.Object obj, System.String fieldName)
{
return
VisModeDefaultPlugin.DefaultPlugins.sum(obj, fieldName);
}
```

Listing 6.9. Fragment kodu przedstawiający wywołanie metod z wtyczek

Wszystkie funkcje, które pochodzą z wtyczek są przysłaniane lokalnymi funkcjami o takiej samej nazwie. Wprowadza to pewne ograniczenie, które nie pozwala na używanie takich samych nazw, jednak z drugiej strony jest dużym ułatwieniem, gdyż trzeba pamiętać, w której z bibliotek dana funkcja się znajduje. Odwołujemy się do niej jak do lokalnej metody. Formularz *Main* jest głównym oknem wygenerowanej aplikacji i jest rodzicem dla pozostałych okien, otwieranych jako dokumenty MID (multiple-document interface). Kolejnymi bazowymi formularzami są *BaseAddEditForm* oraz *BaseViewForm*. Zostały one wprowadzone do struktury w celu umożliwienia w miarę szybki sposób ujednolicania formularzy, które będą po nich dziedziczyć, poprzez ustalanie wspólnych dla nich

właściwości (chociażby takiej jak *ShowInTaskbar*). Dodatkowym wspólnym elementem jest metoda *Show()*, która jako swój wynik zwraca *DialogResult*. Przysłania on swoją bazową funkcjonalność, w której metoda nic nie zwraca (jest typu void). Wszystkie formularze dziedziczące po *BaseAddEditForm* są formularzami do edycji pojedynczego obiektu, takiego jak książka czy czytelnik. Posiadają mechanizm pozwalający na podpięcie obiektu w prosty sposób. Do tego celu wykorzystaliśmy właściwość *DataSource*. Odwrotnie sytuacja wygląda na formularzach dziedziczących po *BaseViewForm*. Wyświetlają kolekcję obiektów, z możliwością wybrania jednego z nich. Właściwość pozwalająca na wskazanie wybranego elementu nazywa się *SelectedObject*. Zastosowanie tych dwóch właściwości w kodzie zostało przedstawiona poniżej:

```
AddEditKsiazka.DataSource = (Ksiazka)VisGrid1.SelectedObject;
```

Listing 6.10. Fragment kodu przedstawiający przypisanie obiektu wybranego z listy do edytora obiektu

7. Zakończenie

Dla każdego programisty głównym celem jest stworzenie systemu w pełni efektywnego oraz realizującego wszystkie stawiane mu wymagania. Potrzeba do tego umiejętności łączenia wielu elementów w jedno finalne rozwiązanie. W momencie gdy wymaga się od nas zwiększonych rezultatów, ulepszonej efektywności oraz krótszego czasu pracy pojawienie się technologii RAD wydaje się być idealnym rozwiązaniem. Usprawnia ona pracę programistów udostępniając wiele gotowych komponentów przez co jest coraz bardziej powszechna. Jednak w momencie, gdy pragnie się stworzyć mniej trywialną aplikację rozwiązania typu RAD szybko ukazują swoje ograniczenia. Tworzy się pewne zagrożenie chociażby z racji ryzyka nierozważnego jej kształtowania.

Celem naszej pracy było stworzenie takiego prototypu, który nie będzie wymagał od użytkownika napisania linii kodu programistycznego zachowując przy tym pełną funkcjonalność. Podstawowym problemem okazało się być tworzenie projektów z niezliczoną ilością zmiennych i funkcji, które w większości niewielkich aplikacji są wykorzystane w minimalnej części. Różnica pomiędzy kodem napisanym przez programistę a używaniem aplikacji typu RAD jest taka, iż programista pisząc kod tworzy go ‘inteligentnie’ czyli zgodnie ze swoimi oczekiwaniami i koryguje go zgodnie z pojawiającymi się na bieżąco problemami.

W tym momencie śmiało można powiedzieć, że stworzona przez nas aplikacja jest efektywna dla mniej skomplikowanych aplikacji, a co za tym idzie małych i średnich przedsiębiorstw. Ogromna jej zaletą jest możliwość stworzenia projektu w bardzo szybkim czasie. Z drugiej strony dla bardziej rozbudowanych rozwiązań istnieje możliwość tworzenia własnego kodu, czy import do narzędzia Visual Studio.

Bibliografia

Książki:

1. „Microsoft Visual Studio 2005. Księga eksperta PL”, Lars Powers, Mike Sell; Microsoft Press 2007
2. „Inżynieria oprogramowania” Andrzej Jaszkiewicz; Helion 1997
3. „C# programowanie” Jasse Liberty; O’Reilly 2006
4. „Visual Studio .NET: .NET Framework – Czarna księga” Julian Templeman, David Vitter; Helion 2003
5. „Developing XML Solutions” Jake Strum; Microsoft Press 2000
6. „XML na poważnie” Przemysław Kazienko, Krzysztof Gwiazda; Helion 2002

Internet:

7. http://www.codeguru.pl/article-319.aspx
8. http://www.microsoft.com/poland/developer/produkty/SOURCESAFE.mspx
9. http://pl.wikibooks.org/wiki/XML/Wprowadzenie
10. http://www.w3.org/TR/xmlschema-0/
11. http://pl.wikipedia.org/wiki/Serializacja
12. http://pl.wikipedia.org/wiki/Wtyczka
13. http://svnbook.red-bean.com/en/1.5/svn.intro.whatis.html#svn.intro.architecture.dia-1
14. http://form-generator-wizard-for-net.qarchive.org/
15. http://www.nconstruct.com/Docs/NConstructDevelopersGuide.pdf
16. http://www.suite4.net/

Spis rysunków

Listingi