|  |  |
| --- | --- |
| Symbol projektu:  DocxDocument. ModelGen | |
| Nazwa projektu:  **Generator modelu obiektowego dokumentu** | |
| Tytuł dokumentu: | **Opis techniczny** |
| Nr wersji: | **1.1** |
| Data utworzenia: | **30.05.2023** |
| Data ostatniej aktualizacji: | **03.06.2024** |
| Osoba odpowiedzialna: | **Jarosław Kuchta (JK)** |
| Autorzy: | **Jarosław Kuchta (JK)** |

Historia dokumentu

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Wersja | Opis modyfikacji | Autor | Data otwarcia | Data zamknięcia |
| 1.0 | Pierwsza wersja dokumentu | JK | 30.05.2023 | 27.10.2023 |
| 1.1 | Wersja z interfejsami | JK | 2.06.2024 |  |

# Wprowadzenie

ModelGen to narzędzie do generowania obiektowego modelu dokumentu na podstawie dostępnych API oferowanych przez firmę Microsoft. Celem projektu jest ułatwienie operowania na dokumentach programu Microsoft Word w sposób obiektowy przy tworzeniu aplikacji opartych na tych dokumentach.

Znanych jest kilka modeli obiektowych dokumentu Microsoft Word:

* model automatyzacji programu Office (zwany modelem VBA),
* model Visual Studio Tools for Office (w skrócie VSTO),
* model Office Scripts,
* model OpenXml.

***Model VBA***jest najstarszym z ww. modeli. Został udostępniony już w pierwszych wersjach programu dla bardziej zaawansowanych użytkowników, którzy mogą zautomatyzować swoją pracę pisząc tzw. *makra*, czyli krótkie procedury w języku Visual Basic for Applications (w skrócie VBA). Program Word ma funkcjonalność pozwalającą użytkownikowi rejestrować wykonywane czynności i zapisywać je w formie makra. Użytkownik, który zapozna się z modelem VBA, może korygować makra tak, że powstają średnio zaawansowane programy z warunkami i iteracjami. Użytkownik może takie makro przypisać do skrótu klawiszowego lub do pozycji menu (obecnie wstążki) i wywoływać wielokrotnie. Uruchomienie makro następuje w programie Visual Basic dołączonym do Microsoft Office. Metody obiektowe zawarte w makrach, powodują wywołania interfejsu automatyzacji programu Word, który zachowuje się tak, jakby to użytkownik wydawał serie komend. To rozwiązanie ma kilka znaczących wad.

Po pierwsze wywołanie każdego makra jest synchroniczne – powoduje odcięcie użytkownika od interfejsu graficznego programu Word. Jeśli wykonanie makra przynosi nieoczekiwane skutki, to przerwanie jego działania jest możliwe tylko przez systemowe zamknięcie programu Word.

Po drugie sterowanie programem Word z programu Visual Basic jest mało efektywne. Visual Basic for Applications jest językiem przestarzałym. Interpreter tego języka został napisany ponad trzydzieści lat temu, w 1993 r. Język nie zawiera wielu udogodnień współczesnych języków obiektowych i nie da się w nim stworzyć bardziej zaawansowanych programów. Również mechanizm automatyzacji OLE/COM jest mało wydajny jak na dzisiejsze potrzeby.

Trzecią wadą jest ograniczenie modelu VBA do poziomu akapitu. Jeśli programista chce działać na tekście zawartym wewnątrz akapitu, np. na fragmentach o jednolitym formacie, to model VBA się do tego nie nadaje. Model VBA definiuje wprawdzie kolekcje zdań, słów i znaków, ale praktyczne ich wykorzystanie jest bardzo nieefektywne.

Wreszcie problemem, który spowodował częściowe wycofanie się firmy Microsoft z mechanizmu makr jest kwestia bezpieczeństwa. W pierwszej dekadzie XXI wieku pojawiła się fala ataków na użytkowników Microsoft Office za pośrednictwem zainfekowanych dokumentów Worda i Excela przesyłanych pocztą elektroniczną. W atakach była wykorzystywana możliwość automatycznego uruchamiania makr przy otwieraniu dokumentów. Reakcją firmy Microsoft było domyślne zablokowanie możliwości uruchamiania makr w ogóle. Aby odblokować tę możliwość użytkownik musi skorzystać z ukrytego menu dewelopera i jawnie zgodzić się uruchamianie.

***Model Visual Studio For Office (VSTO)*** stanowi przeniesienie modelu VBA do świata .NET. Został opracowany w 2003 r. jako framework udostępniający mechanizm automatyzacji COM aplikacji Microsoft Office (w tym programu Word) dla programistów C#. Następnie (w 2008 r.) został rozszerzony do *Visual Studio For Applications (VSTA)* aby umożliwić dostęp do mechanizmów COM innych aplikacji. VSTO wykorzystuje praktycznie tę samą funkcjonalność aplikacji Office co model VBA, ale program pisany w C# może być znacznie bardziej złożony (w porównaniu do makr VBA).

***Mechanizm automatyzacji COM*** (Component Object Model) jest równie stary jak VBA (pierwsze opracowanie w 1993 r.). Mechanizm ten umożliwia sterowanie jednej aplikacji w systemie Windows przez inną aplikację. Zostało to wykorzystane np. w pakiecie aplikacji Microsoft Office do umieszczania arkuszy kalkulacyjnych programu Excel w dokumentach programu Word. Najpierw mechanizm ten był nazywany OLE (Object Linking and Embedding) i był ograniczony tylko do wstawiania zawartości z jednej aplikacji do drugiej lub do dołączania zawartości zapisanej w innym pliku. Po rozszerzeniu na inne możliwości został nazwany COM.

W mechanizmie COM aplikacja sterowana udostępnia swoje API przez zapisy w rejestrze systemu Windows. Funkcje API są rejestrowane jako interfejsy obiektowe obsługiwane przez komponenty aplikacji (zazwyczaj biblioteki DLL). Te interfejsy mogą być odczytywane przez aplikację sterującą, która może je wywoływać tak samo, jak inne funkcje w języku obiektowym (początkowo był wykorzystywany głównie język C++). Wywołanie funkcji jest przekazywane przez system operacyjny do komponentu obsługującego (handlera) wraz z parametrami. Handler może być napisany i skompilowany w innym języku niż aplikacja sterująca. Mówi się, że interfejs COM jest interfejsem typu ABI (Application Binary Interface), bo żądania i dane przekazywane są w postaci binarnej. Po wykonaniu żądania system zwraca do aplikacji sterującej sygnał potwierdzenia wykonania z ewentualnym wynikiem (również w postaci binarnej).

W 2000  r. firma Microsoft rozbudowała mechanizm COM o możliwości wywoływania komponentów aplikacji przez sieć internetową i nazwała ten nowy mechanizm COM+. Po wprowadzeniu frameworka .NET w 2002 r firma Microsoft zapewniła możliwość korzystania z mechanizmu COM i COM+ za pośrednictwem tzw. COM-interop assemblies. Te biblioteki zawierają wrapery (wrappers), czyli funkcje opakowujące interfejsy COM/COM+ w języku C#. Wywołania tych funkcji pochodzą ze *środowiska zarządzanego* (manager environment), a obsługiwane są w środowisku natywnym (native environment). Oba środowiska różnią się zwłaszcza co do czasu życia obiektów i niektórych typów danych, co powoduje znaczące problemy we współpracy aplikacji sterujących (pisanych w C#) i sterowanych (pisanych w C++). Aplikacja pisana w C# nie może mieć pewności, że aplikacja Office utrzymuje przy życiu obiekty zwracane na żądanie, dlatego nie powinna bazować na instancjach obiektów a na ich identyfikatorach. Praktycznie dla każdego żądania przesyłanego przez COM powinna na nowo pozyskiwać wykorzystywane obiekty.

Programy korzystające z VSTO mogą działać synchronicznie, a przy wykorzystaniu wielowątkowości frameworka .NET – również asynchronicznie. Mogą być zintegrowane z aplikacją Office (tu: Word) przez dołączenie do interfejsu graficznego tej aplikacji. Najczęściej swój własny interfejs pokazują w dodatkowym panelu bocznym aplikacji. Niektóre (jak np. tłumacz) są zintegrowane z pakietem Office, a inne – rozpowszechniane jako dodatki COM. To że dodatki VSTO wykonywane są w oddzielonym od Office środowisku zapewnia większą niezależność i przy awarii dodatku niekoniecznie trzeba zamykać całą aplikację Office. Problemem jednak pozostaje kwestia wydajności całego mechanizmu.

***Model Office Scripts*** jest najnowszym mechanizmem automatyzacji. Został opracowany w oparciu modelu VBA, ale do zamiast Visual Basic do sterowania wykorzystuje język JavaScript. Makra są teraz nazywane skryptami. Skrypty są wykonywane asynchronicznie w środowisku Node.js. Sam model Office Scripts wprowadza niewielkie zmiany do modelu VBA.

***Model OpenXml*** jest zdecydowanie różny od poprzednich. Wszystkie powyższe modele odnoszą się do działającej aplikacji Office. Aplikacja udostępnia swoje wewnętrzne obiekty, które realizują funkcje opisane w interfejsie API. W odróżnieniu od tego model OpenXml działa na dokumentach zapisanych w plikach w standardzie ECMA-376 lub ISO/IEC 29500 (w wersji Office od 2007). Umożliwia wczytanie dokumentów z dysku, przetworzenie ich w programie napisanym w C# i zapisanie ponowne.

Model OpenXml różni się od poprzednich również podejściem do klas obiektowych. W modelu VBA i pochodnych od niego klasy obiektowe reprezentują pojęcia, którymi posługuje się użytkownik aplikacji Office. W przypadku programu Word są to akapity, tabele, fragmenty dokumentu zwane zakresami (w tym zakres wybrany). Są zdania, słowa Są też okna, kolekcja otwartych dokumentów, dokument aktywny. Program sterujący może reagować na zdarzenia występujące w aplikacji Office, np. na zmiany edycyjne dokumentu. W odróżnieniu od tego model OpenXml operuje na elementach dokumentu zapisanych w formacie XML. Rozpoznaje strukturę XML i każdemu elementowi przypisuje odpowiednią klasę. W modelu OpenXml brakuje pojęć związanych ze zdarzeniami, ale za to są klasy reprezentujące ciągi tekstu o jednolitym formatowaniu (czego z kolei brakuje w modelu VBA i jego pochodnych).

|  |  |  |  |
| --- | --- | --- | --- |
| Pojęcia | Model VBA, VSTO | Model JavaScript | Model OpenXml |
| Aplikacja | tak | tak | nie |
| Kolekcja okien | tak | tak | nie |
| Kolekcja dokumentów | tak | tak | nie |
| Dokument aktywny | tak | tak | nie |
| Metadane dokumentu (autor, tytuł) | tak | tak | tak |
| Ciało dokumentu | tak | tak | tak |
| Rozdziały | nie | nie | nie |
| Sekcje | tak | tak | nie |
| Nagłówki i stopki | tak | tak | tak |
| Przypisy dolne i końcowe | tak | tak |  |

Oczekiwanym wynikiem działania generatora jest model obiektowy dokumentu, którego elementy (klasy) reprezentują pojęcia „logiczne”, ale instancje tych klas są tworzone z klas elementów OpenXml.

Przykład

Pojęciem logicznym organizacji dokumentu jest *rozdział* (chapter). W bibliotece OpenXml nie ma elementu odpowiadającego rozdziałowi. Można rozpoznać strukturę rozdziałów po nagłówkach, czyli akapitach, które są przydzielone do stylów Heading1, Heading2 itd.

# Struktura dokumentu OpenXml

Dokument w standardzie OpenXml jest pakietem (plikiem ZIP) składającym się z wielu plików XML. Pod względem organizacyjnym pakiet OpenXml składa się z komponentów (zwanych w standardzie ECMA częściami). Wyróżnia się takie części jak MainDocumentPart, Styles, Fonts, Endnotes, Footnotes, HeadingX. Wszystkie te części są zapisywane jako struktury XML z pojedynczym elementem głównym (root). Części powiązane są między sobą relacjami. Relacje też są zapisywane w plikach XML (o rozszerzeniu nazwy „.rels”).

Elementy i atrybuty XML czytane przez wyspecjalizowany czytnik klasy OpenXmlReader tworzy instancje klas zdefiniowanych w OpenXml.

Podstawową (abstrakcyjną) klasą elementów OpenXml jest:

* OpenXmlElement

Na tej podstawie zdefiniowano inne podstawowe klasy abstrakcyjne:

* TypedOpenXmlElement:
* OpenXmlCompositeElement:
  + OpenXmlPartRootElement,
  + TypedOpenXmlCompositeElement,
  + OpenXmlUnknownElement,
* OpenXmlLeafElement:
  + OpenXmlLeafTextElement,
  + TypedOpenXmlLeafElement,
  + TypedOpenXmlLeafTextElement,
* OpenXmlMiscNode.

Większość klas wyspecjalizowanych OpenXml jest pochodna od klas:

* TypedOpenXmlCompositeElement,
* TypedOpenXmlLeafElement,
* TypedOpenXmlLeafTextElement.

Z kolei atrybuty XML służą do tworzenia klas wartości:

* OpenXmlSimpleType:
  + OpenXmlSimpleType<T>:
    - EnumValue<T>,
    - OpenXmlComparableSimpleValue<T>:
      * BooleanValue,
      * ByteValue,
      * DateTimeValue,
      * DecimalValue,
      * DoubleValue,
      * Int16Value,
      * Int32Value,
      * Int64Value,
      * IntegerValue,
      * OnOffValue,
      * SByteValue,
      * SingleValue,
      * TrueFalseBlankValue,
      * TrueFalseValue,
      * UInt16Value,
      * UInt32Value,
      * UInt64Value.
  + OpenXmlComparableSimpleReference<T>:
    - OpenXmlComparableSimpleReference<String>
      * StringValue:
        + HexBinaryValue,
        + Base64BinaryValue,
  + ListValue<T>.

# Proces przetwarzania

Proces przetwarzania modelu składa się z sześciu kroków:

1. Skanowanie typów – typy publiczne z biblioteki źródłowej DocumentFormat.OpenXml są przeglądane i rejestrowane rekurencyjnie metodą refleksji typów począwszy od podanego typu głównego. Dla każdej klasy przeglądane i rejestrowane są typy właściwości publicznych. Dodatkowo dołączane są typy odczytywane ze schematu typów biblioteki, gdzie mogą występować typy elementów składowych nieuwzględnione w deklaracji klas biblioteki. Rejestrowane są również typy wyliczeniowe i typy strukturalne.
2. Uzupełnienie dokumentacji – ponieważ nie wszystkie typy źródłowe mają odpowiednie opisy, więc w tej fazie dodawane są opisy z uprzednio przygotowanego pliku XML.
3. Zmiana nazw typów – zarejestrowane typy mają zmieniane nazwy wynikowe tak, aby dopasować nazwy typów generowanych do modelu wynikowego.
4. Konwertowanie typów – ustalane jest odwzorowanie części typów na typy podstawowe – typy systemowe C# lub predefiniowane typy bazowe modelu wynikowego
5. Końcowe dopasowanie – typy przeznaczone do generowania są sprawdzane pod względem unikatowości nazw w przestrzeniach nazw. Sprawdzana jest dostępność typów danych właściwości klas. Dokonywane są niezbędne korekty.
6. Generowanie typów wynikowych – generowany jest kod odpowiednich typów (interfejsów, klas, struktur, typów wyliczeniowych) umieszczany w plikach o odpowiednich nazwach w katalogach tworzonych w ścieżce plików wynikowych na podstawie przestrzeni nazw.

W ostatnim etapie mogą być wykorzystywane różne typy generatorów, które dają różne wyniki. Aktualnie istnieją trzy typy generatorów:

* ModelGenerator – który produkuje klasy danych modelu. Klasy mają właściwości, które przechowują informacje wczytane w formacie OpenXml.

# Konstrukcja oprogramowania

W bibliotece ModelGen zdefiniowano dwie główne klasy kreatorów kodu, które analizują typy zdefiniowane w bibliotece źródłowej i generują kod klas wynikowych do odpowiedniej biblioteki docelowej. Są to

* ModelCreator – kreator klas (i innych typów) modelu,
* ConverterCreator – kreator klas konwertujących klasy modelu na elementy formatu OpenXml.

## Konwerter bazowy

Oba kreatory są zdefiniowane na bazie abstrakcyjnej klasy BaseConverter, która zawiera metodę RunOn(Type) organizującą proces analizy i generowania klas modelu oraz metody implementujące kolejne kroki tego procesu. Są to (w kolejności wywoływania):

* 1. ScanType(Type) – skanująca i rejestrująca typy
  2. RenameTypes – zmieniająca nazwy typów,
  3. AddTypeConversions – ustalająca konwersje niektórych typów na typy proste i typy bazowe.
  4. CheckTypeUsage – sprawdzająca użycie typów i zaznaczająca te typy, które będą generowane,
  5. ValidateTypes – sprawdzająca poprawność typów do generowania i dokonująca korekty w razie potrzeby,
  6. GenerateCode – abstrakcyjna metoda generująca kod. Ta metoda jest implementowana inaczej w każdej z klas potomnych.

Każda z tych metod zwraca wynik typu TimeSpan reprezentująca czas wykonania procedury.

Pomiędzy kolejnymi krokami procesu mogą być wywoływane metody monitorujące proces. Sterują tym opcjonalne parametry metody RunOn:

* monitorDisplaySelector (typu wyliczeniowego MDS),
* displayOptions (typu DisplayOptions).

Flagi MDS wybierają krok, po którym wyświetlane są wyniki analizy:

* ScannedNamespaces – po pierwszym kroku wyświetlane jest podsumowanie zeskanowanych przestrzeni nazw,
* ScannedTypes – po pierwszym kroku wyświetlane są zeskanowane typy (alternatywnie do ScannedNamespaces),
* TypeRenames – po drugim kroku wyświetlane są relacje zmiany nazwy,
* TypeConversions – po trzecim kroku wyświetlane są relacje konwersji typów,
* TypeUsage – po czwartym kroku wyświetlane są szczegóły przestrzeni nazw przeznaczonych do generowania.
* ValidatedTypes – po piątym kroku wyświetlane są szczegóły sprawdzonych i skorygowanych przestrzeni nazw.

Parametr typu DisplayOptions jest wykorzystywany przy wyświetlaniu przestrzeni nazw (p. *Monitorowanie procesu*).

## Monitorowanie procesu

Do monitorowania procesu jest wykorzystywana klasa ModelDisplay. Klasa ta jest statyczna, co umożliwia wykorzystywanie jej metod we wszystkich pozostałych klasach i wyświetlanie wyników na wspólnym wyjściu.

Klasa działa w oparciu o Writer (typu IndentedTextWriter), który domyślnie jest ustawiony na konsolę.

Główne metody klasy ModelDisplay są następujące:

* SetOutput – ustaw wyjście na podany TextWriter,
* WriteLine(string) – wypisz linię tekstu na wyjściu.
* WriteLine() – wypis pustą linię na wyjściu.
* WriteSameLine(string) – wróć na początek linii i wypisz tekst w tej samej linii (poprzednio wypisany tekst jest wymazywany),
* ShowNamespaceSummary(OTS) – wypisz podsumowanie przestrzeni nazw modelu, parametr typu OTS umożliwia wybór kategorii przestrzeni nazw (p. poniżej),
* ShowNamespaceDetails(DisplayOptions) – wypisz szczegóły (typy) wszystkich przestrzeni nazw.
* ShowNamespaceDetails(string, DisplayOptions) – wypisz szczegóły (typy) określonej przestrzeni nazw,
* ShowTypes(string, DisplayOptions) – wypisz wszystkie typy określonej przestrzeni nazw,
* ShowTypeInfo(TypeInfo, DisplayOptions) – wypisz informacje o określonym typie,
* ShowGenericParamsConstraints(TypeInfo, DisplayOptions) – wypisz informacje o parametrach generycznych zadeklarowanych w określonym typie,
* ShowImplementedInterfaces(TypeInfo, DisplayOptions) – wypisz informacje o interfejsach implementowanych w określonym typie,
* ShowElementsTypes(TypeInfo, DisplayOptions) – wypisz informacje o typach elementów, które mogą się pojawić w określonym typie,
* ShowOutgoingRelationships(TypeInfo, DisplayOptions) – wypisz informacje o relacjach wychodzących z typu (dla których typ jest źródłem),
* ShowIncomingRelationships(TypeInfo, DisplayOptions) – wypisz informacje o relacjach wchodzących do typu (dla których typ jest celem),
* ShowEnumValues(TypeInfo, DisplayOptions) – wypisz informacje o wartościach typu wyliczeniowego,
* ShowProperties(TypeInfo, DisplayOptions) – wypisz informacje o właściwościach klasy,
* ShowTypeConversions() – wypisz wszystkie konwersje typów,
* ShowTypeConversion(TypeOnfo) – wypisz konwersję danego typu,
* ShowTypeRenames() – wypisz zmiany nazw typów,

Parametr typu DisplayOptions umożliwia podanie opcji wyboru kategorii przestrzeni nazw oraz wyboru informacji do wyświetlania o typach. Klasa DisplayOptions udostępnia następujące ustawienia:

* NamespaceTypeSelector: NTS – umożliwia wybór przestrzeni oryginalnych, docelowych lub systemowych poprzez parametr typu wyliczeniowego NTS. Typ ten ma następujące opcje:
* Any – żaden filtr wyboru nie jest stosowany do przestrzeni nazw.
* Origin – wybierane są przestrzenie nazw rozpoczynające się od „DocumentFormat”,
* Target – wybierane są przestrzenie nazw rozpoczynające się od „DocumentModel”.
* System – wybierane są przestrzenie nazw rozpoczynające się od „System”.
* Namespaces: string[] – filtr przestrzeni nazw do wyświetlenia. Może zawierać przestrzenie nazw lub ich wzorce (z wieloznacznymi gwiazdkami)
* TypeKindSelector: TKS – umożliwia wybór rodzajów typów do wyświetlenia poprzez parametr typu wyliczeniowego TKS. Typ ten ma następujące opcje:
* Any – żaden filtr wyboru nie jest stosowany do typów.
* Enum – wybierane są typy wyliczeniowe,
* Struct – umożliwia wybór docelowych przestrzeni nazw rozpoczynających się od „DocumentModel”.
* System – umożliwia wybór systemowych przestrzeni nazw rozpoczynających się od „System”.
* Typenames: string[] – filtr nazw typów do wyświetlenia. Może zawierać nazwy typów lub ich wzorce (z wieloznacznymi gwiazdkami)
* TypeDataSelector: TDS – określa, które informacje są wyświetlane dla typów. Możliwe opcje wyboru to:
  + AcceptedTypesOnly – wyświetlane są tylko typy akceptowane do generowania kodu.
  + OriginalNames – wyświetlane są oryginalne nazwy typów,
  + BaseTypes – wyświetlane są typy bazowe klas,
  + ImplementedInterfaces – wyświetlane są interfejsy implementowane przez klasy,
  + ElementTypes – wyświetlane są informacje o typach elementów, które mogą się pojawić w danym typie.
  + GenericParamsConstraints – wyświetlane są zastrzeżenia dotyczące parametrów typów uogólnionych,
  + OutgoingRelationships – wyświetlane są relacje wychodzące z danego typu do innych typów (te, dla których dany typ jest źródłem relacji),
  + IncomingRelationships – wyświetlane są relacje wchodzące do danego typu z innych typów (te, dla których dany typ jest celem relacji),
  + SelectedSemantics – wyświetlane są relacje o wybranych semantykach,
  + ExcludedSemantics – nie wyświetlane są relacje o pewnych semantykach,
  + EnumValues – wyświetlane są wartości typów wyliczeniowych,
  + Properties – wyświetlane są właściwości typów,
  + HideUnnacceptedProperties – niezaakceptowane właściwości są ukrywane,
  + HideUnnacceptedTypeDetails – inne niezaakceptowane szczegóły typów są ukrywane.

Domyślnym ustawieniem jest BaseTypes|Properties.

* ListLimit: int – narzuca limit elementów wyświetlanych na liście:
  + ImplementedInterfaces,
  + ElementsTypes,
  + OutgoingRelationships,
  + IncomingRelationships,
  + EnumValues,
  + Properties.

Domyślnym ustawieniem jest 10.

* SemanticsFilter: string[] – opcjonalna lista semantyk stosowana jako filtr przy opcjach TDS:
  + OutgoingRelationships –filtr na semantyki relacji wychodzących,
  + IncomingRelationships – filtr na semantyki relacji wchodzących,
  + SelectedSemantics – jako wybrane semantyki,
  + ExcludedSemantics – jako wykluczone semantyki,

Opcje TDS SelectedSemantics i ExcludedSemantics mogą być stosowane alternatywnie (albo-albo), dlatego jest tylko jedno pole na filtr semantyk (wybieranych albo wykluczanych).

## Informacje o typach i przestrzeniach nazw

Przy skanowaniu typów z biblioteki źródłowej następuje ich rejestracja w statycznej klasie TypeManager w polu:

* KnownTypes: Dictionary<Type, TypeInfo>

Jednocześnie następuje rejestracja ich przestrzeni nazw w polu klasy TypeManager:

* KnownNamespaces: Dictionary<string, TypeDictionary>

Dla każdego typu tworzona jest instancja klasy danych TypeInfo zawierająca informacje wykorzystywane w procesie analizy i generowania kodu. Ponadto dla typów klas, struktur i interfejsów wykorzystywane są instancje klasy PropInfo, a dla typów wyliczeniowych – instancje klasy EnumInfo.

Informacje o typach, właściwościach i wartościach wyliczeniowych są odczytywane z biblioteki źródłowej przez mechanizm refleksji typów (System.Reflection). Tam są zdefiniowane analogiczne klasy (TypeInfo, PropertyInfo, FieldInfo), ale zakres reprezentowanych przez nie danych jest niewystarczający do procesu przetwarzania, stąd wynikła konieczność zdefiniowania własnych klas w module kreatora.

Część wspólnych danych z klas TypeInfo, PropInfo i EnumInfo jest reprezentowana przez wspólną klasę podstawową ModelInfo, która implementuje interfejs IOwnedElement, reprezentujący element posiadający obiekt właścicielski (Owner). Zdefiniowanie osobnego interfejsu było konieczne ze względu na to, że właściciela muszą mieć też inne dane przetwarzane w procesie. Oprócz klasy ModelInfo interfejs IOwnedElement jest implementowany przez klasy:

* OwnedCollection<T>,
* CustomAttribNamedArgument,
* CustomAttribTypedArgument.

### IOwnedElement

Interfejs IOwnedElement definiuje tylko jedną właściwość oznaczającą właściciela danego elementu:

* Owner: object {opt}.

Jej interpretacja zależy od klasy implementującej interfejs.

### OwnedCollection<T>

Klasa OwnedCollection<T> jest kolekcją obiektów implementujących IOwnedElement. Sama też implementuje IOwnedElement. Jeśli ma ustawioną właściwość Owner, to przy dodawaniu elementu do kolekcji ustawia właściciela elementu na ten sam obiekt.

### ModelElement

Klasa ModelElement grupuje właściwości wspólne dla elementów modelu. Te właściwości to:

* Owner: object {opt} – obiekt właścicielski elementu,
* Name: string – nazwa elementu (taka, jak to odczytano z biblioteki źródłowej),
* NewName: QualifiedName {opt} – nowa nazwa (wraz z ewentualną nową przestrzenią nazw) stosowana przy zmianie nazwy elementu.
* IsAccepted: bool – określa, czy element został zaakceptowany do dalszego przetwarzania (alternatywnie z IsRejected).
* IsRejected: bool – określa, czy element został odrzucony od dalszego przetwarzania (alternatywnie z IsAccepted).
* IsUsed: bool – określa, czy element jest wykorzystywany przez inne elementy,
* IsConverted: bool – określa, czy element jest konwertowany na inny element,
* IsConvertedTo: bool – określa, czy inny element jest konwertowany na ten element,
* Summary: string {opt} – krótki opis tekstowy elementu,
* Documentation: XElement {opt} – dokumentacja elementu (w formie Xml),
* CustomAttributes: OwnerCollection<CustomAttribInfo> - kolekcja atrybutów użytkownika przypisanych do elementu.

### CustomAttribInfo

Klasa CustomAttribInfo jest elementem modelu (ModelElement), który reprezentuje atrybut użytkownika (CustomAttribute z refleksji typów) przypisany do elementu. Jej właściwości to:

* AttributeType: TypeInfo – typ atrybutu,
* ConstructorArguments: OwnedCollection<CustomAttribTypedArgument> – kolekcja argumentów przekazanych przez konstruktor,
* NamedArguments: OwnedCollection<CustomAttribNamedArgument> – kolekcja argumentów przekazanych przez konstruktor.

### CustomAttribTypedArgument

Klasa CustomAttribTypedArgument reprezentuje argument atrybutu użytkownika przekazywany przez konstruktor atrybutu. Jej właściwości to:

* Owner: object {opt} – obiekt właścicielski argumentu, ponieważ argument NIE jest traktowany jako samodzielny element modelu, więc implementuje interfejs IOwnedElement przez tę właściwość.
* ArgumentTypeInfo: TypeInfo – informacja o typie argumentu,
* Value: object {opt} – wartość argumentu.

### CustomAttribNamedArgument

Klasa CustomAttribNamedArgument reprezentuje argument atrybutu użytkownika przekazywany przez nazwę atrybutu. Rozszerza klasę CustomAttribTypedArgument o nazwę:

* Name: string – nazwa argumentu.

Pozostałe właściwości bez zmian.

### TypeInfo

Klasa TypeInfo przechowuje informacje o typie odczytanym z biblioteki źródłowej oraz o typie przeznaczonym do generowania kodu w bibliotece docelowej.

* Acceptance: ARS – status akceptacji elementu do dalszego przetwarzania. Typ ARS jest typem wyliczeniowym o trzech możliwych wartościach:
  + None – status nieokreślony,
  + Accepted – element zaakceptowany do dalszego przetwarzania,
  + Rejected – element odrzucony od dalszego przetwarzania.

# Struktura rozwiązania

ModelGen występuje w postaci rozwiązania C# z czterema projektami:

* biblioteką ModelGen realizującą proces generowania kodu,
* konsolowym programem uruchomieniowym ModelGenRun,
* okienkowym programem uruchomieniowym ModelGenApp,
* biblioteką ModelConfig organizującą dane konfiguracyjne modelu.

## Program uruchomieniowy

Program ModelGenRun zawiera dwie procedury:

* GenerateModelTypes – do generowania typów modelu,
* GenerateTypeConverters – do generowania konwerterów typów.

Obie procedury mają podobną strukturę:

1. Tworzą podstawową ścieżkę plików wynikowych na podstawie lokalizacji rozwiązania.
2. Tworzą odpowiedni kreator dla klas wynikowych:
   * ModelCreator – kreator modelu,
   * ConverterCreator – kreator konwerterów klas modelu na elementy formatu OpenXml.
3. Uruchamiają kreator dla podanego typu z biblioteki źródłowej, który to typ będzie służył za typ główny (korzeń w drzewie analizy).

Aby wygenerować pełny model dokumentu tekstowego należy podać typ DocumentFormat.OpenXml.Packaging.WordprocessingDocument.

DocumentFormat.OpenXml (zwanej dalej *biblioteką źródłową*).