Jarosław Kuchta

Model Gen

Opis techniczny

data opracowania: 30.05.2023

# Wprowadzenie

ModelGen to narzędzie do generowania kodu modelu dokumentu na podstawie biblioteki DocumentFormat.OpenXml (zwanej dalej *biblioteką źródłową*). Generuje kod klas (typów) elementów modelu w katalogu DocumentModel (zwanego dalej *biblioteką docelową modelu*) oraz kod konwerterów klas modelu z/do formatu OpenXML w katalogu DocumentModel.OpenXml (zwanego dalej *biblioteką docelową konwerterów*).

Narzędzie występuje w postaci rozwiązania C# z dwoma projektami:

* biblioteką ModelGen realizującą proces generowania kodu,
* programem uruchomieniowym ModelGenRun.

## 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.

## Proces generowania kodu

Proces generowania kodu, zarówno typów modelu, jak i konwerterów typów, składa się z sześciu kroków:

1. Skanowanie typów – typy publiczne z biblioteki źródłowej 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. Zmiana nazw typów – zarejestrowane typy mają zmieniane nazwy wynikowe tak, aby dopasować nazwy typów generowanych do konkretnego kreatora.
3. Konwertowanie typów – ustalane jest odwzorowanie części typów na typy podstawowe – typy systemowe C# lub typy bazowe modelu zdefiniowane uprzednio.
4. Sprawdzanie użycia typów – tylko typy wykorzystywane w innych typach będą generowane. Niektóre typy zostaną pominięte ze względu na konwersję innych typów na typy podstawowe.
5. Walidowanie typów i przestrzeni nazw – typy przeznaczone do generowania są sprawdzane pod względem unikatowości nazw w przestrzeniach nazw. Dokonywane są niezbędne korekty.
6. Generowanie typów wynikowych – konkretny kreator generuje 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.

Pierwsze pięć kroków jest wspólnych dla obu kreatorów, ostatni jest realizowany w każdym kreatorze osobno.

# 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.