Jarosław Kuchta

Projekt ModelGen

data pierwszego opracowania: 27.10.2023

data ostatniej modyfikacji: 31.12.2023

# Wprowadzenie

ModelGen to narzędzie do generowania kodu modelu dokumentu na podstawie bibliotek Microsoft.Office.Interop.Word oraz DocumentFormat.OpenXml. Celem projektu jest ułatwienie operowania na dokumencie programu Microsoft Word w sposób obiektowy i przy tworzeniu aplikacji opartych na tym dokumencie.

## Możliwości przetwarzania dokumentów Worda

Aktualnie deweloperzy mają kilka możliwości pisania własnych aplikacji do przetwarzania dokumentów programu Word.

1. Napisanie dodatku do programu Word w języku Visual Basic w wersji VBA (Visual Basic for Applications). Taki dodatek jest uruchamiany w środowisku Office z aplikacji Worda. Deweloper wykorzystuje interfejs automatyzacji COM (Component Object Model) tej aplikacji. Interfejs ten oferuje większość funkcjonalności, które są dostępne dla użytkownika z menu programu Word. Aplikacja Word umożliwia użytkownikowi rejestrowanie swoich działań w tzw. „makrach”, które są zapisywane jako procedury w języku Visual Basic. Użytkownik może edytować kod tych procedur i uruchamiać takie makra wielokrotnie. W ten sposób może usprawnić swoją pracę łącząc wiele operacji w sekwencje i powtarzając te działania w pętli. Działanie dodatku jest ściśle zespolone z działaniem aplikacji, tak że każda operacja wykonana w dodatku może być natychmiast odwzorowana w widoku dokumentu w aplikacji Word. Takie zespolenie ma też wadę polegającą na tym, że zawieszenie się (zapętlenie) wykonania kodu w dodatku zawiesza też działanie aplikacji Word, a wyjście z tego zawieszenia wymaga zrestartowania całej aplikacji Word. Mechanizm automatyzacji COM jest bardzo starym i mało wydajnym mechanizmem, a interfejs automatyzacji COM został zdefiniowany już bardzo dawno i nie odzwierciedla nowszych możliwości aplikacji Word. Tym, co utrudnia przeglądanie całego dokumentu w kodzie VBA jest brak możliwości przeglądania tekstu po fragmentach akapitów mających takie samo formatowanie.
2. Napisanie dodatku do programu Word w języku C# z wykorzystaniem mechanizmu automatyzacji aplikacji Word przez zestaw narzędzi o nazwie VSTO (Visual Studio Tools for Office). Narzędzia te obejmują zbiór bibliotek zapewniający wywołania funkcjonalności aplikacji Office udostępnianych przez interfejsy i klasy zdefiniowane w bibliotece Microsoft.Office.­Interop.Word z poziomu języka C#, a także zbiór dodatków do programu Visual Studio umożliwiających edycję dokumentów z poziomu środowiska IDE. Dodatki VSTO mają podobne możliwości i ograniczenia co dodatki VBA, ale użycie języka C# powoduje, że kod dodatku jest wykonywany w środowisku zarządzanym (.NET Framework) poza środowiskiem Office. Wprawdzie wymaga to w dalszym ciągu współdziałania z uruchomioną aplikacją Worda, ale jest dużo bardziej niezawodne. Poza tym deweloper ma do dyspozycji pełny model obiektowy .NET, dzięki czemu może pisać dodatki VSTO dużo bardziej złożone niż proste dodatki VBA, a nawet tworzyć „pełnokrwiste” aplikacje .NET współistniejące z programem Word.
3. Pisanie dodatków JavaScript do programu Word wykorzystujących modele obiektowe Word JavaScript API oraz Common (Office) API. Te modele zostały stworzone w oparciu o model automatyzacji COM i implementują większość jego koncepcji (chociaż nie wszystkie). W porównaniu do dodatków VBA język JavaScript oferuje bardziej efektywne struktury danych i kodu. Wykonanie skryptów JavaScript jest asynchroniczne, a więc zawieszenie się dodatku nie spowoduje zawieszenia się całej aplikacji. Z drugiej strony koncepcja zastosowania skryptów JS w aplikacji Word jest stosunkowo nowa, a jej implementacja jest niepozbawiona błędów i luk.
4. Tworzenie programów C# przetwarzających dokumenty Worda zapisane w plikach DOCX. To rozwiązanie jest zupełnie inne od poprzednich, gdyż nie wymaga współdziałania z uruchomioną aplikacją Worda. Jest zupełnie niezależne od instalacji Microsoft Office i może być też stosowane w innych niż Windows systemach operacyjnych (np. w Linuksie). Przetwarzane są dokumenty zapisane na dysku lub przesłane przez sieć. Program może modyfikować takie pliki lub zapisywać ich modyfikacje w innych plikach. Po przetworzeniu dokumentu w zewnętrznej aplikacji użytkownik może znowu otworzyć ten dokument w aplikacji Worda i dalej go edytować. Każdy plik DOCX to w rzeczywistości pakiet ZIP plików w formacie XML. Pakiet zawiera strukturę dokumentu głównego, definicje stylów, list numerowanych etc. Deweloper wykorzystuje tu bibliotekę DocumentFormat.OpenXml, która definiuje „silnie typowane” elementy Xml. Oznacza to, że każdy element XML zapisany w pliku XML spakowanym w DOCX jest odczytywany jako obiekt pewnej klasy C#, która określa jego strukturę wewnętrzną (przechowywane atrybuty XML i zawierane inne elementy OpenXml). Takie rozwiązanie udostępnia deweloperowi całą wewnętrzną strukturę dokumentu (akapity, formatowane fragmenty tekstu, poprawki redakcyjne etc.) ale z drugiej strony nie udostępnia funkcjonalności realizowanych normalnie przez aplikację Word (np. wyszukiwanie i zamianę tekstu). Te operacje deweloper musi zaimplementować samodzielnie. Niektóre takie operacje zostały napisane i udostępnione na GitHubie przez dewelopera posługującego się nickiem EricWhiteDev w postaci biblioteki OpenXmlPowerTools.

Przedstawione powyżej rozwiązania mają swoje wady i zalety. Zaletą pierwszych trzech podejść jest możliwość wykorzystania funkcjonalności aplikacji Worda do przetwarzania dokumentów. Zaletą ostatniego rozwiązania jest dostęp do wszystkich szczegółów treści dokumentu a także dużo większa szybkość przetwarzania dokumentów. Wspólną wadą każdego z nich jest brak pełnego modelu obiektowego dokumentu Worda. Pełnego, tzn. opisującego zarówno pełną wewnętrzną strukturę dokumentu (z dokładnością do jednolicie sformatowanego fragmentu tekstu, jak „Run” z biblioteki DocumentFormat.OpenXml), jak i złożone operacje na tej strukturze (jak np. wyszukiwanie i zamiana tekstu). Najbardziej zbliżone do pożądanego ideału jest ostatnie rozwiązanie, z odczytywaniem dokumentu DOCX przez bibliotekę DocumentFormat.OpenXml i z popularnymi operacjami zdefiniowanymi w bibliotece OpenXmlPowerTools. Problemem utrudniającym wykorzystanie tego rozwiązania jest ograniczenie zbioru funkcjonalności dostępnych w OpenXmlPowerTools do kilku scenariuszy użycia:

* Dzielenie plików DOCX/PPTX na wiele plików.
* Łączenie wielu plików DOCX/PPTX w jeden plik.
* Wypełnianie treści w plikach szablonów DOCX danymi z XML.
* Wysokiej jakości konwersja DOCX do HTML/CSS i na odwrót.
* Wyszukiwanie i zastępowanie treści w DOCX/PPTX za pomocą wyrażeń regularnych.
* Zarządzanie śledzonymi wersjami, w tym wykrywanie śledzonych wersji i akceptowanie śledzonych wersji.
* Aktualizacja wykresów w plikach DOCX/PPTX, w tym aktualizacja danych z pamięci podręcznej, a także osadzonego XLSX.
* Pobieranie metryk z plików DOCX, w tym hierarchii używanych stylów, używanych języków i używanych czcionek.
* Pisanie plików XLSX przy użyciu znacznie prostszego kodu niż bezpośrednie pisanie znaczników, w tym podejście strumieniowe, które umożliwia pisanie plików XLSX z milionami wierszy.
* Wyodrębnianie danych (wraz z formatowaniem) z arkuszy kalkulacyjnych.

Jednak głównym problemem jest fakt, że model obiektowy DocumentFormat.OpenXml reprezentuje model „fizyczny” dokumentu, a nie model „logiczny”. To znaczy, że pojęcia (klasy i właściwości) w nim wyrażone reprezentują elementy i atrybuty XML, nie zaś obiekty składowe dokumentu. Nie ma np. pojęcia „zakładki” (Bookmark), jest tylko znacznik początku i końca zakładki (BookmarkStart i BookmarkEnd). Nie ma pojęcia „sekcji” (Section), są tylko właściwości sekcji (SectionProperties) zapisywane w wybranych akapitach.

## Potrzeba pełnego modelu obiektowego

Powyższe problemy ujawniły potrzebę stworzenia pełnego modelu obiektowego dokumentu, który ułatwiłby przetwarzanie obiektowe elementów dokumentu, w tym zdefiniowanie elementów „wyższego poziomu”, jak np. tabela specyfikacyjna (czyli tabela o ściśle określonej zawartości) pożądana dla tworzenia dokumentacji technicznej projektów inżynierskich. Taka dokumentacja musi być redagowana w sposób usystematyzowany i kontrolowany. Musi odzwierciedlać rzeczywistą strukturę „logiczną” projektu, ale z drugiej strony też umożliwiać elastyczną prezentację tej struktury. Tak więc dokument Worda powinien zawierać „logiczne” elementy projektowe (co jest możliwe przy wykorzystaniu części dokumentu zwanej CustomXmlPart) oraz umożliwiać ich prezentowanie w treści „wizualnej” dokumentu poprzez ich powiązanie z takimi elementami „wizualnymi” jak tabela. Wiersze takiej specjalnie oznaczonej („specyfikacyjnej”) tabeli odwzorowywałyby właściwości elementów projektowych, a szablon dokumentu definiowałby formatowanie i dozwoloną zawartość takich tabel. Oprócz tego oczywiście dokument umożliwiałby umieszczanie dowolnego tekstu i rysunków.

Taka koncepcja wykorzystania edytora Word do redagowania dokumentacji projektowej wymaga połączenia ze sobą przynajmniej dwóch przedstawionych powyżej rozwiązań technicznych. Z jednej strony wymagane jest współdziałanie specjalnie napisanej aplikacji projektowej z uruchomioną aplikacją Worda, w tym przechwytywanie zdarzeń występujących w aplikacji Worda (np. przejście autora do redagowania tabeli specyfikacyjnej). Jest to możliwe przy wykorzystaniu automatyzacji VSTO. Z drugiej strony wymagany jest dostęp do wewnętrznej struktury dokumentu, co jest możliwe jedynie poprzez DocumentFormat.OpenXml.

## Pobieranie dokumentu z aplikacji

Niekonieczne jest zapisywanie całego dokumentu Worda w pliku DOCX. Możliwe jest pozyskanie fragmentu dokumentów interfejsie automatyzacji poprzez tzw. „FlatOpc”. Jest to tekst w formacie XML zawierający „spłaszczoną” strukturę OpenXml, czyli nie zbiór plików XML spakowany formatem ZIP, a te same dane umieszczone w pojedynczym drzewie elementów OpenXml. Dodatek VSTO może pobrać fragment dokumentu w formacie FlatOpc i przetworzyć na OpenXml lub pobrać cały dokument w formacie OpenXml. Tak pobrany dokument może przesłać do zewnętrznej aplikacji, która potrafi czytać i zapisywać format OpenXml. Po modyfikacji dokumentu dodatek VSTO powinien zastąpić przesłaną część dokumentu zmodyfikowaną częścią pliku OpenXml. Proces ilustruje rys. 1.

Aplikacja Worda

Docx

Dodatek VSTO

Interfejs automatyzacji



Aplikacja zewnętrzna

OpenXml



Docx

Rys. . Przetwarzanie dokumentów Worda za pomocą zewnętrznej aplikacji

## Idea projektu ModelGen

Ideą projektu ModelGen jest stworzenie „logicznego” modelu obiektowego zgodnego z interfejsem automatyzacji Word w oparciu o „fizyczny” model obiektowy OpenXml. Wówczas deweloper tworzący aplikację zewnętrzną będzie mógł się posługiwać się klasami ułatwiającymi przetwarzanie logicznej struktury dokumentu DOCX w formacie OpenXml.

Podstawą definicji nowego modelu obiektowego dokumentu jest model „fizyczny” OpenXml. Na nim nabudowany jest model „logiczny” wzorowany na interfejsie VSTO InteropWord. Ponieważ zarówno model OpenXml, jak i InteropWord zawierają ogromną liczbę typów (OpenXml - 5105, InteropWord - 867), więc potrzebne jest narzędzie do automatycznego generowania nowego modelu na podstawie tych dwóch.

## Oczekiwany wynik

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

Na przykład główną klasą OpenXml odczytywaną z pakietu DOCX jest WordprocessingDocument. Klasa ta udostępnia bezpośrednio obiekt PackageProperties, a pośrednio ExtendedFilePropertiesPart.Properties i CustomFilePropertiesPart.Properties. Dwie pierwsze klasy definiują właściwości wbudowane, a trzecia – właściwości definiowane przez użytkownika. Załóżmy, że główną klasą nowego modelu jest DocumentModel.Document. Ta klasa powinna mieć właściwość BuiltInProperties udostępniającą obiekt tej klasy, który będzie podawał i przyjmował właściwości z PackageProperties oraz ExtendedFile­PropertiesPart.Properties, a także właściwość klasy CustomProperties, która będzie udostępniać elementy klasy CustomProperty reprezentujące elementy zapisane w CustomFilePropertiesPart.Properties. Ilustruje to rys. 2

docProps\core.xml

docProps\app.xml

docProps\custom.xml

Document.docx

PackageProperties

ExtendedFilePropertiesPart.

Properties

WordprocessingDocument

CustomFilePropertiesPart.

Properties

BuiltItProperties

DocumentModel.Document

CustomProperties

Rys. . Koncepcja odwzorowania klas modelu dokumentu

Klasa BuiltInProperties z przestrzeni nazw DocumentModel reprezentuje klasę PackageProperties z przestrzeni System.IO.Packaging oraz Properties z przestrzeni DocumentFormat.OpenXml.ExtendedProperties. Ta pierwsza zawiera właściwości związane z pakietem ZIP, a druga – z dokumentem aplikacji Office.

## Podejście etapowe

Drogą do uzyskania oczekiwanego wyniku jest podejście składające się z trzech etapów:

* W pierwszym etapie trzeba określić ogólną strukturę modelu. Powinna być ona niezależna od implementacji, a jednocześnie elastyczna dla zapewnienia możliwości dopasowania się do nowych wersji aplikacji Worda jak również rozszerzania modelu o elementy nieistniejące w samym dokumencie. Taka struktura może być stworzona w oparciu o pewien zbiór interfejsów obiektowych uporządkowanych w formie drzewa.
* W drugim etapie stworzyć implementację interfejsów opracowanych w pierwszym etapie. Implementacja będzie oparta o strukturę XML dokumentu DOCX.
* Trzeci etap będzie obejmował opracowanie mechanizmu rozszerzania modelu, chociaż zagadnienia z tym związane powinny już być uwzględniane w poprzednich etapach.

## Koncepcja interfejsów modelu

Dobrą podstawą dla opracowania struktury interfejsów obiektowych modelu dokumentu jest model VSTO wyrażany przez bibliotekę Microsoft.Office.Interop.Word.[[1]](#footnote-1)

Wszystkie interfejsy w tej bibliotece są zorganizowane w formę drzewa, którego korzeniem jest interfejs Application reprezentujący aplikację Worda. Interfejs ten oferuje możliwości otwierania dokumentów, z których każdy jest reprezentowany przez interfejs Document. Wszystkie otwarte dokumenty są dostępne przez kolekcję Documents będącą właściwością interfejsu Application.

Interfejs Document ma swoje właściwości, w tym kolekcje elementów podrzędnych, takich jak sekcje (Sections), akapity (Paragraphs), tabele (Tables) i inne. Podział na kolekcje następuje dalej, w elementach składowych. Prawie wszystkie elementy dokumentu są pogrupowane w kolekcje. Kolekcje są indeksowane liczbami całkowitymi (począwszy od 1), a czasami również nazwami elementów składowych (takich jak np. Style).

Elementy składowe dokumentu odnoszące się do treści tekstowej (jak i sam dokument) udostępniają właściwość Range, która reprezentuje ich zawartość. Interfejs Range sam też udostępnia kolekcje Sections, Paragraphs i Tables, ale sięga jeszcze głębiej w strukturę tekstu podając kolekcje zdań (Sentences), słów (Words) i znaków (Characters).

Ta struktura interfejsów może zostać odtworzona w modelu docelowym. Uzasadnienie jest jednak wprowadzenie kilku modyfikacji:

* Poza zakresem projektu ModelGen znajduje się interfejs Application. Dokument może być otwierany przez metodę Open interfejsu Document, a kolekcja otwartych dokumentów może być zorganizowana poza planowanym modelem. To, co interfejs Application może zapewnić, to pobieranie treści dokumentu lub jego fragmentu z aplikacji Worda w formacie FlatOpc. Interfejs taki mógłby również zapewnić reagowanie na zdarzenia z aplikacji Worda.
* Podobnie poza modelem dokumentu znajduje się interfejs Selection reprezentujący wybrany zakres dokumentu oraz wszystkie interfejsy związane z interfejsem użytkownika aplikacji Worda (Window, TaskPane etc.) oraz z jego interfejsem VBA (Word Basic for Applications).
* Interfejs Range musi być pomyślany w inny sposób. Obecnie ma on właściwości Start i End reprezentujące odpowiednio początkową i końcową pozycję znakową (jako liczby całkowite). Wchodzi to w konflikt z hierarchiczną strukturą elementów dokumentu. W takiej strukturze może obejmować całą zawartość elementu lub wybrany zbiór jego elementów składowych. Elementy składowe mogą być różnych klas, ale dla każdego elementu złożonego muszą być zdefiniowane dopuszczalne klasy elementów składowych. Dla przykładu elementami składowymi tabeli mogą być wiersze i kolumny, ale komórki stanowią już składowe wierszy i kolumn. Aby uzyskać dostęp do treści komórki trzeba się będzie odwołać do jej właściwości Range.
* Interfejsy Sentences, Words i Characters są mało przydatne. Algorytm wyodrębniania zdań z tekstu nie odróżnia kropek kończących zdania od kropek kończących skrótowce. Podobnie algorytm oddzielający słowa ma problemy ze „słowami” złożonymi, np. składającymi się z członów typu „niebiesko-zielony” albo sekwencjami alfanumerycznymi (np. 0x20AB, czy H2O). Z kolei kolekcja Characters podaje zakresy klasy Range obejmujące pojedyncze znaki, co jest mocno nieefektywne w przetwarzaniu tekstu.
* Brakuje w modelu VSTO interfejsu reprezentujące ciąg znaków o identycznym formatowaniu. W modelu DocumentFormat.OpenXml.Wordprocessing taki ciąg jest reprezentowany przez klasę Run będącą jedną z możliwych składowych akapitu.
* Większość kolekcji w interfejsie VSTO jest indeksowanych od 1. Jest to naleciałość z języka Visual Basic. Tymczasem większość współczesnych języków programowania obsługuje kolekcje indeksowane od 0 i takie też powinno być indeksowanie kolekcji elementów modelu docelowego.

## Koncepcje implementacji modelu dokumentu

Jako że przetwarzanie dokumentów Worda ma się opierać o format DOCX, więc trzeba zapewnić konwersję elementów XML zawartych w tym formacie na obiekty i klasy implementujące obiektowe interfejsy docelowego modelu dokumentu.

Ze względu na znaczące różnice w interfejsach niemożliwe jest prosta implementacja interfejsów docelowego modelu dokumentu w klasach DocumentFormat.OpenXml. Wielokrotnie może się zdarzyć, że do implementacji pojedynczego interfejsu potrzebny będzie dostęp do wielu klas DocumentFormat.OpenXml.

Możliwe są dwa podejścia do implementacji:

* Wczytywanie atrybutów i elementów składowych z elementów XML pliku DOCX do pól klas implementujących interfejs docelowy modelu dokumentu. Po ewentualnej modyfikacji pola te mogłyby zostać (na żądanie) zapisane jako atrybuty i elementy XML z powrotem w pliku DOCX.
* Przechowywanie otwartych obiektów klas z biblioteki DocumentFormat.OpenXml wewnątrz klas implementujących interfejs docelowy modelu dokumentu. Wówczas ewentualne modyfikacje mogłyby od razu być odzwierciedlone w strukturze XML dokumentu DOCX.

Oba rozwiązania zakładają użycie klas biblioteki DocumentFormat.OpenXml jako pośredniczących w czytaniu i zapisywaniu struktury XML pliku DOCX. Istnieje możliwość bezpośredniego czytania i pisania XML z użyciem biblioteki System.Xml.Linq, jednak wówczas trzeba rozróżniać między atrybutami XML i elementami XML oraz ich odpowiednie dekodowanie/kodowanie. Skorzystanie z ułatwień udostępnianych przez bibliotekę DocumentFormat.OpenXml jest prostsze w implementacji, gdyż atrybuty i elementy XML są już przetwarzane na wartości typów prostych i obiekty klas zdefiniowanych w języku C#. W przypadku wielu elementów modelu dokumentu wystarczy proste obudowanie (ang. enveloping) klas DocumentFormat.OpenXml i proste tłumaczenie wartości prostych za pomocą kilku wspólnych konwerterów.

Trzeba pamiętać, że wewnętrzna struktura całego pliku DOCX zawiera wiele plików XML połączonych w pakiet ZIP. Biblioteka DocumentFormat.OpenXml współdziała z biblioteką System.IO.Packaging, która obsługuje pakiety ZIP.

Zastosowanie pierwszego podejścia zakłada wczytywanie całego pliku DOCX. Po wczytaniu można plik DOCX zamknąć, a otworzyć ponownie do zapisu po dokonaniu wszystkich zmian. Trzeba jednak zauważyć, że o ile wczytanie struktury DocumentFormat.OpenXml dokumentu i jej zapisanie w pliku jest szybkie, o tyle translacja ze struktury DocumentFormat.OpenXml do klas implementujących docelowy model dokumentu może być czasochłonne.

Drugie podejście zakłada skorzystanie z klasy WordprocessingDocument z przestrzeni nazw DocumentFormat.OpenXml.Packaging. Obiekt tej klasy reprezentuje cały pakiet ZIP w formacie OpenXml i zawiera wiele części (reprezentowanych przez klasę OpenXmlPart i jej pochodne). Dopiero odwołanie się do poszczególnych części otwiera odpowiednie struktury XML i tworzy instancje klas składowych dokumentu zdefiniowane w przestrzeni nazw DocumentFormat.OpenXml.Wordprocessing. Dzięki temu struktura obiektowa modelu będzie tworzona w pamięci operacyjnej w miarę potrzeb aplikacji ją wykorzystującej.

## Koncepcja procesu przetwarzania

Ze względu na ogromną liczbę elementów modelu dokumentu wskazane jest napisanie aplikacji ułatwiającej generowanie modelu na bazie istniejących modeli: modelu VSTO i OpenXml. Z modelu VSTO trzeba wybrać te interfejsy, które zostaną przeniesione do docelowego modelu dokumentu. Po uzupełnieniu docelowej struktury interfejsów modelu trzeba wygenerować klasy je implementujące w oparciu o bibliotekę OpenXml.

Proces przetwarzania obu modeli można zdefiniować na bazie wspólnego procesu, który zawierać będzie wspólne aktywności naszkicowane następująco:

* Wczytywanie typów – typy publiczne z biblioteki źródłowej (Microsoft. Office. Interop. Word lub DocumentFormat. OpenXml) będą wczytywane do programu ModelGen z uwzględnieniem przestrzeni nazw.
* Filtrowanie typów – typy (i przestrzenie nazw) będą oznaczane jako zaakceptowane lub odrzucone. Typy odrzucane nie będą dalej przetwarzane. Typy zaakceptowane po przetworzeniu będą umieszczane w modelu dokumentu. Znaczniki typów zaakceptowanych będą umożliwiać stopniowe generowanie docelowego modelu dokumentu.
* Wczytanie wewnętrznej struktury typów – jeden z typów trzeba wyznaczyć jako typ główny. W przypadku pierwszej z bibliotek będzie to interfejs Document, a drugiej – klasa WordprocessingDocument. Począwszy od tego typu głównego wczytywane będą właściwości publiczne typów zaakceptowanych. Umożliwi to odwoływanie się do typów z innych bibliotek.
* Określanie relacji między – relacje dziedziczenia, implementowania interfejsów oraz zawierania. W tym ostatnim przypadku może być potrzebna analiza schematów XML formatu OpenXml.
* Uzupełnianie dokumentacji – typy akceptowane i właściwości akceptowane mają wczytywane opisy bezpośrednio z biblioteki lub z uprzednio przygotowanego pliku XML.
* Przetworzenie struktury przestrzeni nazw – w przypadku biblioteki Microsoft.Office.Interop.Word jest jedna przestrzeń nazw, ale typy z tej przestrzeni odwołują się też do bibliotek Microsoft.Vbe.Interop oraz office. W przypadku biblioteki DocumentFormat.OpenXml jest wiele przestrzeni nazw związanych zarówno z osobnymi grupami funkcjonalności (np. z tworzeniem wykresów), jak i kolejnymi wersjami aplikacji Word.
* Przypisanie typów do właściwych przestrzeni nazw – można to zrobić za pomocą odpowiednich reguł nazw jak i relacji między typami.
* Agregacja pewnych stałych – w bibliotece Microsoft.Office.Interop.Word wykorzystuje się wiele stałych typu całkowitego, które powinny być zorganizowane w typy wyliczeniowe.
* Zmiana nazw pewnych typów – ponieważ typy o tych samych nazwach z kilku różnych źródłowych przestrzeni nazw mogą trafić do tej samej docelowej przestrzeni, więc trzeba im nadać różne nazwy docelowe. Również typom wyliczeniowym trzeba zmienić nazwy tak, aby pasowały do zaplanowanych w docelowym modelu dokumentu.
* Konwertowanie typów – ustalane jest odwzorowanie części typów na typy podstawowe – typy systemowe C# lub predefiniowane typy bazowe modelu wynikowego. Trzeba zaimplementować konwertery typów i zastosować je w projekcie ModelGen.
* Końcowe dopasowanie – typy zaakceptowane są reorganizowane tak, aby dopasować się do docelowego modelu dokumentu.
* Generowanie typów wynikowych –kod typów zaakceptowanych jest generowany i umieszczany w plikach o odpowiednich nazwach w katalogach tworzonych w ścieżce plików wynikowych na podstawie przestrzeni nazw.

Należy nastawić się na pracę iteracyjną, przy czym pliki wynikowe generowane w kodzie języka C# będą kompilowane w biblioteki wynikowe i będą wykorzystywane zwrotnie jako biblioteki wejściowe. Poglądowo koncepcję procesu przetwarzania zilustrowano na . Przyjęto na nim, że biblioteki wynikowe mają wspólne nazwy rozpoczynające się od DocumentModel.

Biblioteki źródłowe

* Microsoft. Office. Interop. Word
* DocumentFormat. OpenXml
* …

Biblioteki wynikowe

* DocumentModel
* …

ModelGen

Rys. . Iteracyjny proces przetwarzania

1. Szczegółowa analiza tego modelu została przedstawiona w dokumencie [..\Model dokumentu Interop Word.docx](../Model%20dokumentu%20Interop%20Word.docx) [↑](#footnote-ref-1)