Jarosław Kuchta

Projekt DocxDocument

data utworzenia: 16.12.2023

data ostatniej modyfikacji: 16.12.2023

# Wprowadzenie

Celem projektu jest ułatwienie operowania na dokumencie programu Microsoft Word w sposób obiektowy przy tworzeniu aplikacji opartych na tym dokumencie.

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 niezależne aplikacje .NET luźno współdziałające 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 „mocno 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 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 wybranych scenariuszy użycia, takich jak:

* 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.

Powyższe problemy ujawniły potrzebę stworzenia pełnego modelu obiektowego dokumentu, który ułatwiłby przetwarzanie obiektowe elementów dokumentu, w tym definiowanie klas, które aktualnie nie istnieją w modelu dokument Docx.

Przykładem zastosowania takiego modelu może być aplikacja do edycji dokumentacji projektów informatycznych. Istnieje wiele aplikacji tego typu wykorzystujących różne sposoby i formaty zapisu. Nie chodzi tu o tylko takie aplikacje jak Doxygen, które służą do generowania dokumentów PDF czy HTML z komentarzy zawartych w kodzie źródłowym. Chodzi także o aplikacje do tworzenia dokumentów bardziej opisowych, służących zrozumieniu idei projektu, wymagań projektowych, czy też projektu architektury. Tu zasadniczo dostępne są proste aplikacje do edycji tekstu formatowanego (np. w formacie Wiki lub Markdown) albo złożone systemy do modelowania graficznego (np. Visual Paradigm, Enterprise Architect). Brakuje aplikacji łączących możliwości tych wszystkich aplikacji i systemów.

Taka aplikacja pobierałaby dane z systemu deweloperskiego (np. typu Visual Studio albo Idea) i prezentowałaby je w dokumencie procesora tekstu (takim jak Word), który zawierałby opisy, specyfikacje i diagramy. 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 powinien zawierać „logiczne” elementy projektowe oraz umożliwiać ich prezentowanie w treści „wizualnej” dokumentu poprzez ich powiązanie z takimi elementami „wizualnymi” jak tekst formatowany, tabela czy diagram.

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.

Na szczęście 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

Ideą projektu DocxDocument 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 wspomagania 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.