Rozdział 26. OpenXML (.docx)

Klasa `Statystyka` używana w poniższym tekście jest dołączona na końcu dokumentu.

Nowsze wersje pakietu Microsoft Office stosują do zapisu dokumentów, arkuszy i prezentacji otwarty format OpenXML (pliki z rozszerzeniami *.docx*, *.xlsx* i *.ptpx*). Warto wobec tego rozważyć go jako format eksportu np. raportów, faktur itp. zamiast zwykle stosowanego w takich przypadkach zamkniętego formatu PDF. Zyskujemy dzięki temu możliwość edycji wyeksportowanego dokumentu, choć możemy również zabezpieczyć go przez zmianami hasłem (tego ostatniego nie będę tu jednak opisywał). W tym rozdziale przedstawię sposób tworzenia pliku dokumentu dla edytora tekstu Word w formacie OpenXML, w którym umieszczę prosty akapit złożony z kilku linii, dodam rysunek, a na końcu wstawię tabelę. W tym dokumencie umieścimy wyniki obliczeń statystycznych — wykorzystamy w tym celu projekt *Statystyka*, który rozwijaliśmy w rozdziałach 7., 14., 15. i 20. Związek nowej części projektu z wcześniejszą będzie jednak bardzo luźny. Potrzebne jest nam po prostu jakieś źródło przykładowego tekstu i liczb, które będą umieszczone w tabeli. Metody generujące dokument *.docx* z łatwością można będzie przenieść do innych projektów.

Pakiet NuGet

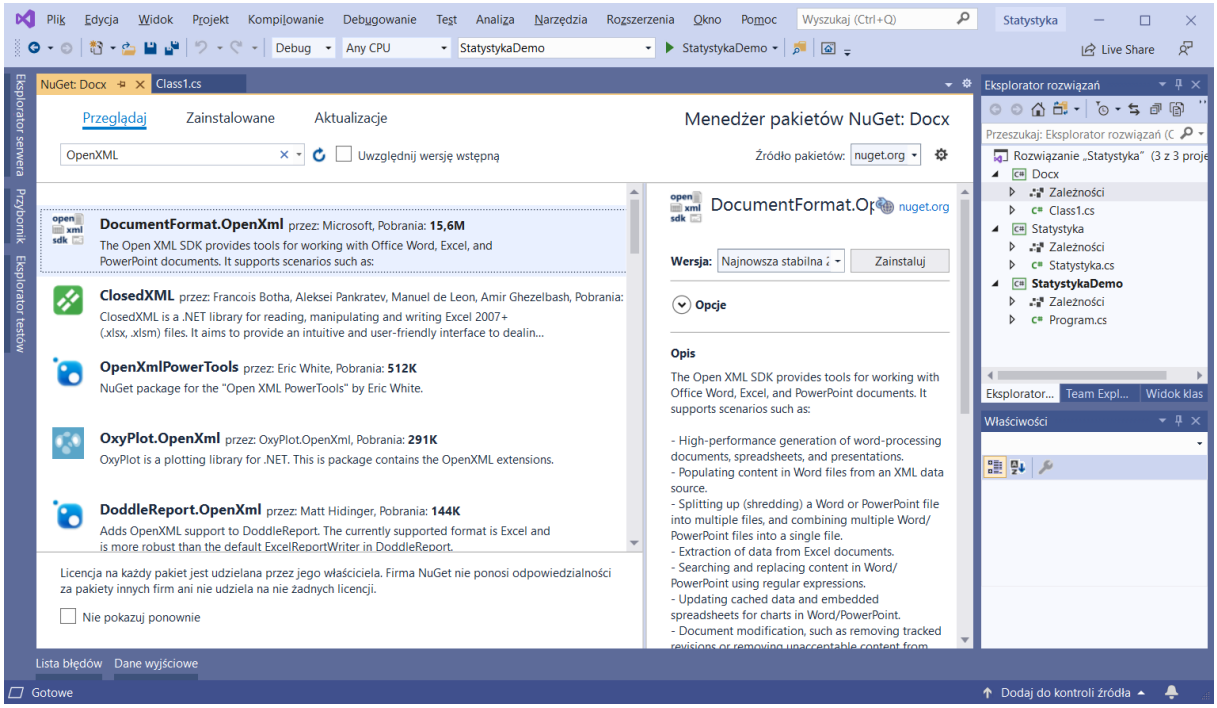
Obsługa formatu OpenXML wymaga wykorzystania pakietu NuGet, więc będzie to też świetna okazja, żeby przedstawić ten sposób dzielenia się kodem przez społeczność programistów .NET. Mechanizm NuGet nie przewiduje opłat, ale możliwe jest ustalenie warunków licencji. Witryna głównego repozytorium znajduje się pod adresem *http://nuget.org*[1]. Umożliwia przeglądanie i pobieranie pakietów, choć te czynności łatwiej wykonać, używając klienta wbudowanego w Visual Studio.

Proponuję, aby kod związany z obsługą formatu OpenXML umieścić w osobnej bibliotece. Dodajmy wobec tego do rozwiązania *Statystyka* nowy projekt biblioteki .NET Standard o nazwie *Docx*. W tym celu z menu kontekstowego w podoknie *Eksplorator rozwiązań* rozwiniętego dla całego rozwiązania wybieramy polecenie *Dodaj*, *Nowy projekt...*. W oknie *Dodawanie nowego projektu* zaznaczamy szablon *Biblioteka klas (.NET Standard)*, nadajemy projektowi nazwę *Docx* i tworzymy go.

Do tego projektu należy dodać pakiet NuGet o nazwie *DocumentFormat.OpenXml*. Aby to zrobić, w oknie *Eksplorator rozwiązań* zaznaczamy pozycję *Zależności* i z menu kontekstowego otwieranego prawym klawiszem myszy wybieramy polecenie *Zarządzaj pakietami NuGet...*. Nie pojawi się okno dialogowe, a zamiast tego utworzona zostanie nowa zakładka w głównej części okna Visual Studio. Klikamy *Przeglądaj*, aby zobaczyć najpopularniejsze pakiety z serwisu NuGet. Następnie w polu tekstowym w górnej części zakładki wpisujemy „OpenXML”. Zobaczymy pasujące do tego hasła pakiety, wśród których pierwszy to najprawdopodobniej *DocumentFormat.OpenXml* — to jest pakiet przygotowany przez Microsoft (rysunek 26.1). Ma ogromną przewagę pobrań (w marcu 2020 było to aż 15,6 miliona). Należy go zaznaczyć i kliknąć przycisk *Zainstaluj* widoczny w prawej części okna. Pojawi się wówczas okno dialogowe żądające potwierdzenia chęci instalacji pakietu. Po chwili pakiet zostanie pobrany i zainstalowany. Instalacja odbywa się na poziomie projektu. To oznacza, że w ewentualnych kolejnych projektach korzystających z OpenXML, nawet w tym

[1] Możliwe jest też lokalne przechowywanie pakietów NuGet (lokalne repozytorium Visual Studio), a nawet własny hosting pakietów.

samym rozwiązaniu, będzie trzeba zainstalować go ponownie, jednak jeżeli projekt jest w tym samym rozwiązaniu, pobrane pliki pakietu będą współdzielone. Możemy zamknąć zakładkę *NuGet: Docx*.



Rysunek 26.1. Przeglądanie pakietów NuGet w Visual Studio

Formatowania

W utworzonym wcześniej projekcie biblioteki *Docx* zmieńmy nazwę pliku *Class1.cs* na *PomocnikDocx.cs* i pozwólmy także na zmianę nazwy klasy. W tej klasie zdefiniujemy kilka metod odpowiedzialnych za tworzenie akapitów w dokumencie Worda. Zaczniemy od akapitu zawierającego tekst. Jednak zanim się za to zabierzemy, dodajmy najpierw do projektu biblioteki *Docx* jeszcze jeden plik klasy, o nazwie *FormatowaniaDocx.cs*, w którym umieścimy formaty wykorzystywane przez planowane metody: kolor normalny i wyróżniony, domyślną czcionkę, styl tekstu i styl dla tytułu oraz ustawienia akapitu, w którym zwiększony jest odstęp między liniami (interlinia). Wszystkie te formatowania umieściłem w postaci własności w klasie `FormatowaniaDocx` widocznej na listingu 26.1.

Listing 26.1. Zebrane w jednej klasie formatowania, jakich będziemy używać w dokumencie

```
namespace Docx
{
    using DocumentFormat.OpenXml;
    using DocumentFormat.OpenXml.Wordprocessing;

    public static class FormatowaniaDocx
    {
        private const int rozmiarCzcionkiNormalny = 21;
        private const int rozmiarCzcionkiTytułu = 48;
        private const string kształtCzcionki = "Arial";

        //własności muszą zwracać za każdym razem nowe obiekty,
        //bo nie można używać obiektów, które już są częścią drzewa
        public static Color KolorWyróżniony
```

```
{
    get
    {
        return new Color() { Val = "203A8F" };
    }
}

public static Color KolorNormalny
{
    get
    {
        return new Color() { Val = "black" };
    }
}

public static RunFonts CzcionkaNormalna
{
    get
    {
        return new RunFonts()
        {
            Ascii = kształtCzcionki,
            HighAnsi = kształtCzcionki
        };
    }
}

public static RunProperties StylUstępuNormalny
{
    get
    {
        return new RunProperties()
        {
            RunFonts = CzcionkaNormalna,
            FontSize = new FontSize()
            {
                Val = new StringValue(rozmiarCzcionkiNormalny.ToString())
            },
            Color = KolorNormalny
        };
    }
}

public static RunProperties StylUstępuTytułu
{
    get
```

```
                {
                    return new RunProperties()
                    {
                        Bold = new Bold()
                        {
                            Val = OnOffValue.FromBoolean(true)
                        },
                        FontSize = new FontSize()
                        {
                            Val = new StringValue(rozmiarCzcionkiTytułu.ToString())
                        },
                        RunFonts = CzcionkaNormalna,
                        Color = KolorWyróżniony
                    };
                }
            }

            public static ParagraphProperties UstawieniaAkapituZwiększonaInterlinia
            {
                get
                {
                    ParagraphProperties paragraphProperties = new ParagraphProperties();
                    //interlinia 1,5 dla czcionki 12
                    paragraphProperties.Append(
                        new SpacingBetweenLines() { Line = "360" });
                    return paragraphProperties;
                }
            }
        }
    }
```

Formatowania zostały udostępnione jako własności, które przy każdym odczycie tworzą nowy obiekt typu `Color`, `RunProperties` lub `ParagraphProperties`. Wydaje się, że lepszym rozwiązaniem byłoby zapisanie tych obiektów w polach i udostępnianie przez własności już istniejących egzemplarzy. Oszczędzilibyśmy w ten sposób pamięć, szczególnie że wszystkie trzy wymienione wyżej typy to klasy, więc przekazywanie referencji do istniejących obiektów zamiast ciągłego tworzenia nowych wydaje się dobrym pomysłem. Niestety, struktura dokumentu OpenXML, a w niej mieszczą się także ustawienia, nie pozwala na powtórzenia. Każde wystąpienie formatu musi być osobnym unikalnym obiektem. Stąd taka postać powyższych własności.

Skoro wspomniana została struktura dokumentów w plikach *.docx*, warto ją pokrótce omówić. Dokument posiada tzw. główną część (ang. *main part*), która z kolei zawiera ciało dokumentu (ang. *document body*). Dopiero w ciele znajdują się akapity (ang. *paragraphs*). Akapity nie są jednak jednolitym tekstem — różne ich fragmenty mogą być formatowane niezależnie. Takie fragmenty po angielsku nazywane są *run*. W nazwach powyższych własności używam na ich określenie polskiej nazwy „ustęp”, choć nie jest ona jednoznaczna.

Tekst

Wróćmy teraz do klasy `PomocnikDocx` i umieśćmy w niej metodę tworzącą akapit składający się z kilku linii. Tablica łańcuchów zawierających te linie jest pierwszym argumentem tej metody, a oprócz niej przesyłane są

rozmiar i kolor czcionki. Kod pliku `PomocnikDocx` z niezbędnymi deklaracjami użycia przestrzeni nazw oraz dostępu do statycznych metod klasy `FormatowaniaDocx` widoczny jest na listingu 26.2.

Listing 26.2. Metoda tworząca akapit z tekstem

```
using System;

using DocumentFormat.OpenXml;
using DocumentFormat.OpenXml.Wordprocessing;

namespace Docx
{
    using static FormatowaniaDocx;

    public class PomocnikDocx
    {
        private static Paragraph twórzAkapit(string[] linie, int rozmiarCzcionki,
                                             Color kolorCzcionki)
        {
            Paragraph akapit = new Paragraph();
            akapit.Append(UstawieniaAkapituZwiększonaInterlinia);
            Run ustęp = akapit.AppendChild(new Run());
            RunProperties stylAkapitu = StylUstępuNormalny;
            stylAkapitu.FontSize.Val = new StringValue(rozmiarCzcionki.ToString());
            stylAkapitu.Color = kolorCzcionki;
            ustęp.Append(stylAkapitu);
            foreach (string line in linie)
            {
                ustęp.AppendChild(new Text(line));
                ustęp.AppendChild(new Break());
            }
            return akapit;
        }
    }
}
```

W pierwszej linii metody `twórzAkapit` tworzony jest obiekt akapitu, tj. obiekt typu `Paragraph` zdefiniowanego w przestrzeni nazw `DocumentFormat.OpenXml.Wordprocessing`[2]. Następnie powstaje obiekt ustępu (klasa `Run`), który formatuję, używając przesłanych przez argumenty wielkości i koloru czcionki. Do ustępu dodaję poszczególne linie z tablicy łańcuchów (obiekty typu `Text`), które przedzielam znakami nowej linii (obiekty `Break`). Metoda zwraca akapit z dodanymi liniami.

Tworzenie dokumentu

Do klasy `PomocnikDocx` dodajmy teraz metodę, która będzie tworzyła dokument Worda. Metoda ta nie będzie jednak zapisywać dokumentu do pliku, tylko umieszczać go w strumieniu. Dzięki temu dokument ten będziemy mogli zarówno zapisać do pliku, jak i przesłać lub wykorzystać w inny sposób. Metoda `wyślijDokumentDoStrumienia` widoczna jest na listingu 26.3. Przyjmuje obiekt typu

[2] Klasy o tej samej nazwie znajdują się także w innych przestrzeniach nazw pobranego pakietu NuGet.

`ParametryStatystyczneZHistogramem` z biblioteki *Statystyka*, z którego odczytuje łańcuch zwracany przez metodę `ToString`, który następnie dzieli na linie i umieszcza w dokumencie. Aby typ `ParametryStatystyczneZHistogramem` był widoczny, konieczne jest dodanie do projektu *Docx* odwołania do projektu *Statystyka*.

W metodzie `wyślijDokumentDoStrumienia` powstaje obiekt typu `WordprocessingDocument`, który reprezentuje cały dokument edytora Word. Do niego dodawana jest część główna, a do niej ciało. Do ciała dodawany jest akapit tworzony zdefiniowaną wcześniej metodą `twórzAkapit`. Dokument należy na końcu zamknąć metodą `Close`. To ważne, bo nie zrobi tego metoda `Dispose` wywoływana przy wyjściu z konstrukcji `using(){}`.[3]

Listing 26.3. Tworzenie dokumentu i przesyłanie go do strumienia

```
using System;
using System.Globalization;
using System.IO;


using DocumentFormat.OpenXml;
using DocumentFormat.OpenXml.Packaging;
using DocumentFormat.OpenXml.Wordprocessing;


using Statystyka;


namespace Docx
{
    using static FormatowaniaDocx;


    public static class PomocnikDocx
    {
        private static Paragraph twórzAkapit(string[] linie, int rozmiarCzcionki,
                                             Color kolorCzcionki)
        {
            ...
        }



        private static void wyślijDokumentDoStrumienia(
            ParametryStatystyczneZHistogramem parametryStatystyczne,
            Stream stream, IFormatProvider formatProvider)
        {
            //tworzenie dokumentu Worda
            using (WordprocessingDocument dokumentWorda = WordprocessingDocument.Create(
                stream, WordprocessingDocumentType.Document, true))
            {
                MainDocumentPart głównaCzęśćDokumentu =
                    dokumentWorda.AddMainDocumentPart();
                głównaCzęśćDokumentu.Document = new Document();
```

[3] Jeżeli chcielibyśmy użyć `using` jako modyfikatora (por. rozdział 6.) w deklaracji zmiennej lokalnej, musielibyśmy zmienić wersję specyfikacji .NET Standard z domyślnej dla Visual Studio 2019 wersji 2.0 na 2.1.

```
                    Body ciałoDokumentu = new Body();
                    głównaCzęśćDokumentu.Document.AppendChild(ciałoDokumentu);

                    //akapit z tekstem
                    string[] linie = parametryStatystyczne.ToString().Split('\n');
                    Paragraph akapit = twórzAkapit(linie, 25, KolorNormalny);
                    ciałoDokumentu.AppendChild(akapit);

                    dokumentWorda.Close(); //zamknięcie dokumentu
                }
                stream.Flush();
            }

            public static void EksportujDoPlikuDocx(
                this ParametryStatystyczneZHistogramem parametryStatystyczne,
                string ścieżkaPliku, IFormatProvider formatProvider = null)
            {
                if (formatProvider == null) formatProvider = new CultureInfo("pl-PL");
                if (File.Exists(ścieżkaPliku)) File.Delete(ścieżkaPliku);
                using (FileStream strumieńPliku = new FileStream(ścieżkaPliku,
                                                                 FileMode.CreateNew))
                {
                    wyślijDokumentDoStrumienia(parametryStatystyczne,
                                               strumieńPliku, formatProvider);
                    strumieńPliku.Close();
                }
            }
        }
    }
```

Na listingu 26.3 widoczna jest także metoda `EksportujDoPlikuDocx`. To pierwsza metoda publiczna w klasie `PomocnikDocx`. Jest metodą rozszerzającą dla typu `ParametryStatystyczneZHistogramem` i zapisuje dokument Word do pliku. W tym celu tworzy obiekt typu `FileStream` i przesyła go do metody `wyślijDokumentDoStrumienia`, która zapisuje w nim dokument.

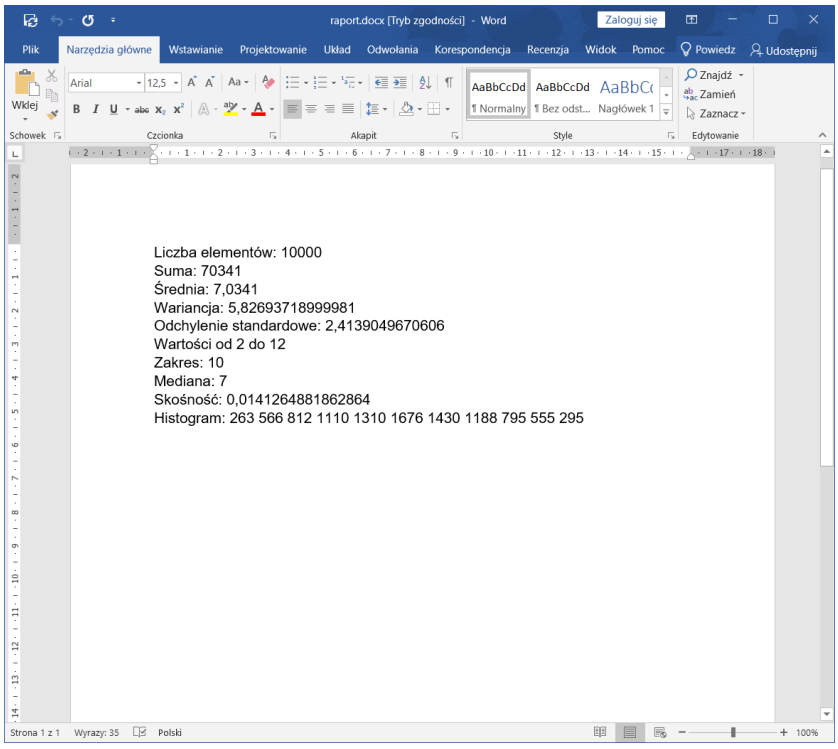
Możemy już sprawdzić, czy uda się nam utworzyć dokument Worda z poziomu aplikacji, wykorzystując powyższe metody. Aby to zrobić, dodajmy do projektu aplikacji *StatystykaDemo* odwołanie do biblioteki *Docx*, a następnie w pliku *Program.cs* zadeklarujmy użycie przestrzeni nazw *Docx* i zmodyfikujmy metodę `Main` zgodnie ze wzorem z listingu 26.4. Po uruchomieniu programu w katalogu, w którym jest skompilowany plik aplikacji, znajdziemy także plik *raport.docx* ze zbiorem parametrów statystycznych (rysunek 26.2).

Listing 26.4. Test tworzenia dokumentu Word z raportem z obliczeń statystycznych

```
using System;
using Docx;

namespace StatystykaDemo
{
    using static Statystyka.Statystyka;
```

Materiał z książki Jacka Matulewskiego „C#. Lekcje programowania. Praktyczna nauka programowania dla platform .NET i .NET Core” dla studentów kursu .NET 2023/2024. Proszę o nieudostępnianie osobom spoza kursu.

```
        class Program
        {
            static int[] zbiórSumOczekZDwóchKostek(int liczbaRzutówKostką = 100) ...


            static void Main(string[] args)
            {
                int[] wartości = zbiórSumOczekZDwóchKostek(10000);
                double[] tablica = Array.ConvertAll<int, double>(wartości, i => (double)i);

                Statystyka.ParametryStatystyczneZHistogramem parametryStatystyczne =
                    new Statystyka.ParametryStatystyczneZHistogramem(tablica, 11);
                Console.WriteLine(parametryStatystyczne.ToString());

                parametryStatystyczne.EksportujDoPlikuDocx("raport.docx");
            }
        }
    }
```



Rysunek 26.2. Dokument utworzony w aplikacji otwarty w edytorze Word z pakietu Microsoft Office

Uwaga! Przed kolejnymi uruchomieniami rozwijanej aplikacji należy zamknąć edytor Word lub inny edytor, w którym otworzyliśmy utworzony dokument. Edytory wywłaszczają dostęp do dokumentu, co uniemożliwi aplikacji jego zmianę, a to doprowadzi do wyjątku.

W ramach małego eksperymentu zmodyfikujmy w metodzie `twórzAkapit` parametr akapitu, dodając polecenie

```
akapit.Append(UstawieniaAkapituZwiększonaInterlinia);
```

zmieniające odstęp między liniami (to powinna być druga linia metody). Instrukcja ta pokazuje, że nie ma wygodnych typów zbierających możliwe ustawienia akapitu, ustępu itp. Do dodania różnych formatowań do

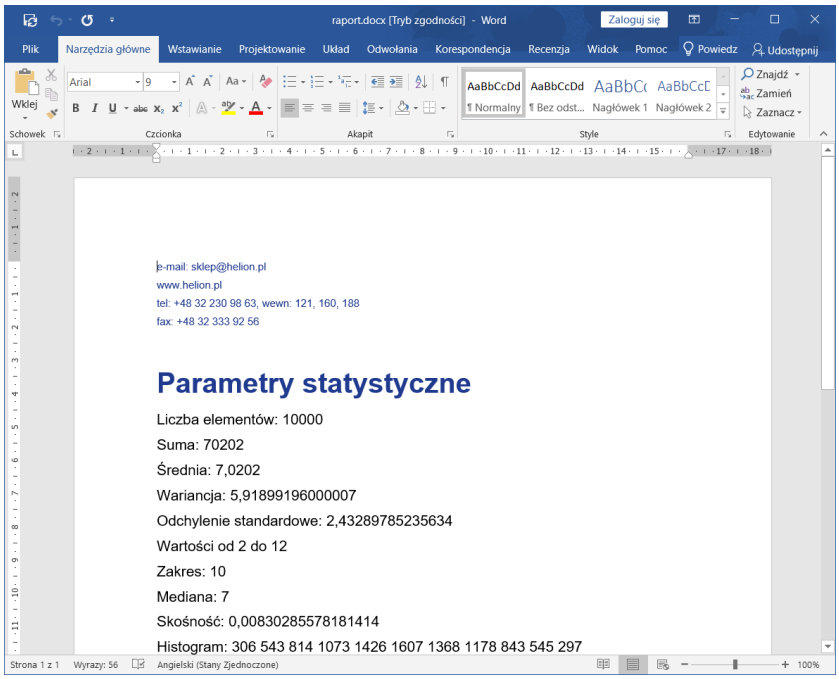
akapitu używamy metody `Append`, w przypadku której system *IntelliSense* nie wspomoże nas, podpowiadając możliwe wartości.

Dodamy do dokumentu jeszcze dwa akapity. Pierwszy utworzy nagłówek z danymi kontaktowymi. Do jego utworzenia użyjemy istniejącej już metody `twórzAkapit`. Zastosujemy jednak mniejszy rozmiar czcionki i zmieniony kolor. Natomiast do utworzenia akapitu tytułu użyjemy nowej metody, która przyjmuje tylko jeden argument — łańcuch tytułu — i formatuje go z użyciem stylów wcześniej zdefiniowanych w klasie `FormatowanieDocx`. Zmiany widoczne są na listingu 26.5, a efekt uruchomienia aplikacji na rysunku 26.3.

Listing 26.5. Tworzenie akapitu składającego się z jednej linii tekstu z powiększoną czcionką

```
using ...

namespace Docx
{
    using static FormatowaniaDocx;

    public static class PomocnikDocx
    {
        private static Paragraph twórzAkapit(string[] linie, int rozmiarCzcionki,
                                             Color kolorCzcionki) ...

        private static Paragraph twórzAkapitTytułu(string tytuł) //tytuł
        {
            Paragraph akapitTytułu = new Paragraph();
            Run ustępTytułu = akapitTytułu.AppendChild(new Run());
            ustępTytułu.Append(StylUstępuTytułu);
            ustępTytułu.AppendChild(new Text(tytuł));
            return akapitTytułu;
        }

        private static void wyślijDokumentDoStrumienia(
            ParametryStatystyczneZHistogramem parametryStatystyczne,
            Stream stream, IFormatProvider formatProvider)
        {
            //tworzenie dokumentu Worda
            using (WordprocessingDocument dokumentWorda = WordprocessingDocument.Create(
                stream, WordprocessingDocumentType.Document, true))
            {
                MainDocumentPart głównaCzęśćDokumentu =
                    dokumentWorda.AddMainDocumentPart();
                głównaCzęśćDokumentu.Document = new Document();

                Body ciałoDokumentu = new Body();
                głównaCzęśćDokumentu.Document.AppendChild(ciałoDokumentu);

                //nagłówek
                string[] linieNagłówka = new string[]
                {
                    "e-mail: sklep@helion.pl",
```

```
                    "www.helion.pl",
                    "tel: +48 32 230 98 63, wewn: 121, 160, 188",
                    "fax: +48 32 333 92 56"
                };
                Paragraph akapitNagłówka =
                    twórzAkapit(linieNagłówka, 18, KolorWyróżniony);
                ciałoDokumentu.AppendChild(akapitNagłówka);

                //tytuł
                ciałoDokumentu.AppendChild(twórzAkapitTytułu("Parametry statystyczne"));

                //akapit z tekstem
                string[] linie = parametryStatystyczne.ToString().Split('\n');
                Paragraph akapit = twórzAkapit(linie, 25, KolorNormalny);
                ciałoDokumentu.AppendChild(akapit);

                dokumentWorda.Close(); //zamknięcie dokumentu
            }

            stream.Flush();
        }

        public static void ExportujDoDocx(
            this ParametryStatystyczneZHistogramem parametryStatystyczne,
            string ścieżkaPliku, IFormatProvider formatProvider = null) ...
    }
}
```

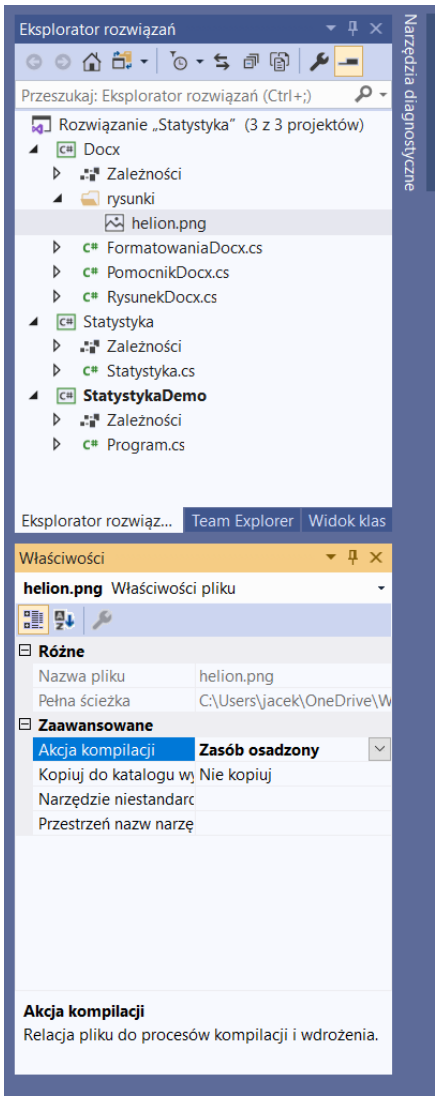


Rysunek 26.3. Dokument uzupełniony o nagłówek z adresem i tytuł, czyli dwa kolejne akapity

Rysunek

Przygotujmy teraz metodę, która będzie tworzyć akapit zawierający rysunek. Rysunek może zostać pobrany z zasobów aplikacji lub z biblioteki, ale również wczytany z pliku.

Zacznijmy od dodania do projektu biblioteki *Docx* rysunku w formacie PNG. Przygotujmy dla niego folder *rysunki*. Plik rysunku możemy po prostu przeciągnąć do podokna *Eksplorator rozwiązań* i upuścić go w folderze *rysunki*. Możemy go również dodać, wybierając z menu kontekstowego folderu polecenie *Dodaj*, *Istniejący element...*. Dla ustalenia uwagi przyjmijmy, że plik ten nazywa się *helion.png*. Po dodaniu rysunku należy go zaznaczyć i w oknie własności (jeżeli go nie widać, należy nacisnąć klawisz *F4*) zmienić ustawienie *Akcja kompilacji* na *Zasób osadzony*. Dzięki temu rysunek zostanie dodany do pliku skompilowanej biblioteki.



Rysunek 26.4. Plik helion.png w podfolderze rysunki i jego własności

Strumień do rysunku umieszczonego w zasobach można uzyskać metodą `Assembly.GetManifestResourceStream`. Metodę tę wykorzystuje widoczna na listingu 26.6 metoda `pobierzRysunekZZasobów`, która pobiera łańcuch identyfikujący zasób (w naszym przypadku będzie to łańcuch *Docx.rysunki.helion.png*). W przypadku pliku odczytywanego z dysku należałoby użyć metody `pobierzRysunekZPliku`, również widocznej na listingu 26.6. W tym drugim przypadku należy zmienić ustawienie pliku *Kopiuj do katalogu wyjściowego* na *Zawsze kopiuj*, aby plik *helion.png* był kopiowany do katalogu, w którym jest umieszczana skompilowana biblioteka, a dokładniej do jego podkatalogu *rysunki*. Wówczas jego ścieżka względna to *rysunki/helion.png*. Obie metody umieściłem w klasie `PomocnikDocx`.

Listing 26.6. Pobieranie strumienia do plików w zasobach i pliku na dysku

```
private static Stream pobierzRysunekZZasobów(string nazwaZasobu)
```

```
    {
        Assembly assembly = typeof(PomocnikDocx).GetTypeInfo().Assembly;
        Stream strumień = assembly.GetManifestResourceStream(nazwaZasobu);
        return strumień;
    }

    private static Stream pobierzRysunekZPliku(string ścieżkaPliku)
    {
        Stream strumień = new FileStream(ścieżkaPliku, FileMode.Open);
        return strumień;
    }

    private static Drawing twórzRysunekWDokumencie(WordprocessingDocument dokumentWorda,
                                                   Stream strumień)
    {
        MainDocumentPart częśćGłówna = dokumentWorda.MainDocumentPart;
        ImagePart częśćRysunku = częśćGłówna.AddImagePart(ImagePartType.Png);
        częśćRysunku.FeedData(strumień);
        return RysunekDocx.CreateDrawingElement(dokumentWorda,
                                                częśćGłówna.GetIdOfPart(częśćRysunku),
                                                3, 1);
    }
```

Na listingu 26.6 widoczna jest także metoda `twórzRysunekWDokumencie`, w której do głównej części dokumentu dodajemy część rysunku (ang. *image part*). Nie chodzi oczywiście o dodanie fragmentu rysunku, a o dodanie części w strukturze dokumentu, która będzie zawierać rysunek. Do tej części wczytywany jest rysunek ze strumienia, co robimy, wywołując metodę `ImagePart.FeedData`. Dopiero tak osadzonego w dokumencie rysunku można użyć w akapitach. Najpierw musimy jednak utworzyć z niego element typu `Drawing`. Do tego użyjemy gotowej metody, która jest szczegółowo opisana w dokumentacji Microsoft na stronie *https://docs.microsoft.com/en-us/office/open-xml/how-to-insert-a-picture-into-a-word-processing-document*. Widoczna na listingu 26.7 metoda `CreateDrawingElement` zawiera kod skopiowany z tej strony. Metodę tę umieściłem w osobnym pliku *RysunekDocx.cs* dołączonym do biblioteki *Docx* i będę traktował jak czarną skrzynkę. Metoda ta przyjmuje jako argument obiekt reprezentujący dokument Worda oraz identyfikator części rysunku i rozmiary — a raczej proporcje — elementu `Drawing`, jaki ma powstać.

Listing 26.7. Kod skopiowany z dokumentacji Microsoft tworzący element Drawing z rysunku dołączonego do dokumentu

```
using DocumentFormat.OpenXml;
using DocumentFormat.OpenXml.Packaging;
using DocumentFormat.OpenXml.Wordprocessing;
using A = DocumentFormat.OpenXml.Drawing;
using DW = DocumentFormat.OpenXml.Drawing.Wordprocessing;
using PIC = DocumentFormat.OpenXml.Drawing.Pictures;

namespace Docx
{
    public static class RysunekDocx
    {
        public static Drawing CreateDrawingElement(this WordprocessingDocument wordDoc,
                                                   string relationshipId,
```

```
                                                float widthCoeff, float heightCoeff)
        {
            // Define the reference of the image.
            Drawing element =
                 new Drawing(
                     new DW.Inline(
                         //new DW.Extent() { Cx = 990000L, Cy = 792000L },
                         new DW.Extent()
                         {
                             Cx = (long)(990000L * widthCoeff),
                             Cy = (long)(792000L * heightCoeff)
                         },
                         new DW.EffectExtent()
                         {
                             LeftEdge = 0L,
                             TopEdge = 0L,
                             RightEdge = 0L,
                             BottomEdge = 0L
                         },
                         new DW.DocProperties()
                         {
                             Id = (UInt32Value)1U,
                             Name = "Rysunek"
                         },
                         new DW.NonVisualGraphicFrameDrawingProperties(
                             new A.GraphicFrameLocks() { NoChangeAspect = true }),
                         new A.Graphic(
                             new A.GraphicData(
                                 new PIC.Picture(
                                     new PIC.NonVisualPictureProperties(
                                         new PIC.NonVisualDrawingProperties()
                                         {
                                             Id = (UInt32Value)0U,
                                             Name = "Rysunek"
                                         },
                                         new PIC.NonVisualPictureDrawingProperties()),
                                     new PIC.BlipFill(
                                         new A.Blip(
                                             new A.BlipExtensionList(
                                                 new A.BlipExtension()
                                                 {
                                                     Uri =
                                                    "{28A0092B-C50C-407E-A947-70E740481C1C}"
                                                 })
                                         )
                                         {
```

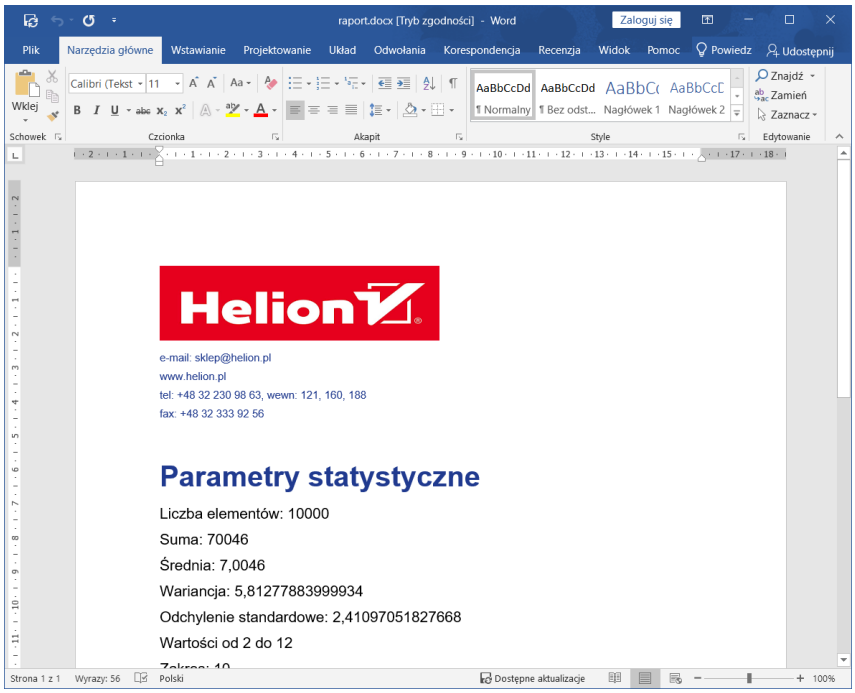
```
                                                Embed = relationshipId,
                                                CompressionState =
                                                A.BlipCompressionValues.Print
                                            },
                                            new A.Stretch(
                                                new A.FillRectangle())),
                                        new PIC.ShapeProperties(
                                            new A.Transform2D(
                                                new A.Offset() { X = 0L, Y = 0L },
                                                new A.Extents() { Cx = (long)(990000L *
widthCoeff), Cy = (long)(792000L * heightCoeff) }),
                                            new A.PresetGeometry(
                                                new A.AdjustValueList()
                                            )
                                            { Preset = A.ShapeTypeValues.Rectangle }))
                                )
                                { Uri =
"http://schemas.openxmlformats.org/drawingml/2006/picture" })
                        )
                        {
                            DistanceFromTop = (UInt32Value)0U,
                            DistanceFromBottom = (UInt32Value)0U,
                            DistanceFromLeft = (UInt32Value)0U,
                            DistanceFromRight = (UInt32Value)0U,
                            EditId = "50D07946"
                        });


                return element;
            }
        }
    }
```

Aby na początku tworzonego dokumentu dodać rysunek z logo (rysunek 26.5), potrzebujemy jeszcze jednej metody, tworzącej akapit zawierający rysunek. To będzie jednak bardzo proste — w jednej linii utworzymy obiekt `Paragraph` i dodamy do niego obiekt `Run`, który będzie zawierać obiekt `Drawing`. Robi to widoczna na listingu 26.8 przeciążona metoda `twórzAkapit`, która przyjmuje obiekt `Drawing` przez argument. Na tym samym listingu widoczne są również zmiany w kodzie metody `wyślijDokumentDoStrumienia`.

Listing 26.8. Umieszczamy rysunek na początku dokumentu

```
private static Paragraph twórzAkapit(Drawing rysunek)
{
    return new Paragraph(new Run(rysunek));
}


private static void wyślijDokumentDoStrumienia(
    ParametryStatystyczneZHistogramem parametryStatystyczne,
    Stream stream, IFormatProvider formatProvider)
{
    //tworzenie dokumentu Worda
```

```
        using (WordprocessingDocument dokumentWorda = WordprocessingDocument.Create(
            stream, WordprocessingDocumentType.Document, true))
        {
            MainDocumentPart głównaCzęśćDokumentu = dokumentWorda.AddMainDocumentPart();
            głównaCzęśćDokumentu.Document = new Document();

            Body ciałoDokumentu = new Body();
            głównaCzęśćDokumentu.Document.AppendChild(ciałoDokumentu);

            //rysunek
            Stream strumieńRysunku = pobierzRysunekZZasobów("Docx.rysunki.helion.png");
            //Stream strumieńRysunku = pobierzRysunekZPliku("rysunki/helion.png");
            Drawing rysunek = twórzRysunekWDokumencie(dokumentWorda, strumieńRysunku);
            Paragraph akapitObrazu = twórzAkapit(rysunek);
            ciałoDokumentu.AppendChild(akapitObrazu);

            //nagłówek
            ...

            dokumentWorda.Close(); //zamknięcie dokumentu
        }

        stream.Flush();
    }
```



Rysunek 26.5. Dokument z wstawionym rysunkiem

Tabela

Ostatnim elementem, który dodamy do dokumentu, będzie tabela zawierająca histogram w zbiorze wielokrotnie losowanych sum oczek na dwóch kostkach do gry. Zacznijmy od umieszczenia w klasie `PomocnikDocx` metody `twórzTabelę`, która przyjmuje dwuwymiarową tablicę łańcuchów z zawartościami komórek tabeli (listing 26.9). Opcjonalnie metoda ta będzie także jako drugi argument pobierać jednowymiarową tablicę z zawartością komórek nagłówka tabeli. Sporą część ciała nowej metody zajmuje kod odpowiedzialny za przygotowanie obiektu `własnościTabeli`, który zawiera ustawienia dotyczące grubości linii na brzegach tabeli i między jej komórkami. Następnie, jeżeli jednowymiarowa tablica `komórkiNagłówka` z tytułami kolumn nie jest równa `null`, tworzony jest wiersz nagłówka (obiekt typu `TableRow`), do którego dodawane są komórki (obiekty `TableCell`). Taki wiersz dodawany jest do tabeli. Aby dodać komórkę do wiersza, a wiersz do tabeli, wykorzystujemy metodę `Append`. Postępując dokładnie tak samo, na podstawie zawartości tablicy `komórki` tworzymy zbiór wierszy, które są także dodawane do tabeli. Na końcu utworzony obiekt tabeli zwracany jest przez wartość metody.

Listing 26.9. Metoda tworząca tabelę

```
private static Table twórzTabelę(string[,] komórki, string[] komórkiNagłówka = null)
{
    if (komórkiNagłówka != null && komórkiNagłówka.Length != komórki.GetLength(0))
        throw new Exception("Nieprawidłowa liczba elementów nagłówka");

    Table tabela = new Table(); //pusta tabela

    uint grubośćZewnętrznejLinii = 10;
    uint grubośćWewnętrznejLinii = 2;

    //ustawienia tabeli
    TableProperties własnościTabeli = new TableProperties(new TableBorders(
        new TopBorder()
        {
            Val = new EnumValue<BorderValues>(BorderValues.BasicThinLines),
            Size = grubośćZewnętrznejLinii
        },
        new BottomBorder()
        {
            Val = new EnumValue<BorderValues>(BorderValues.BasicThinLines),
            Size = grubośćZewnętrznejLinii
        },
        new LeftBorder()
        {
            Val = new EnumValue<BorderValues>(BorderValues.BasicThinLines),
            Size = grubośćZewnętrznejLinii
        },
        new RightBorder()
        {
            Val = new EnumValue<BorderValues>(BorderValues.BasicThinLines),
            Size = grubośćZewnętrznejLinii
        },
        new InsideHorizontalBorder()
```

```
            {
                Val = new EnumValue<BorderValues>(BorderValues.BasicThinLines),
                Size = grubośćWewnętrznejLinii
            },
            new InsideVerticalBorder()
            {
                Val = new EnumValue<BorderValues>(BorderValues.BasicThinLines),
                Size = grubośćWewnętrznejLinii
            }));
        tabela.AppendChild(własnościTabeli);

        if (komórkiNagłówka != null)
        {
            TableRow wierszNagłówka = new TableRow();
            for (int kolumna = 0; kolumna < komórkiNagłówka.Length; kolumna++)
            {
                TableCell komórkaTabeli = new TableCell();
                komórkaTabeli.Append(new TableCellProperties(new TableCellWidth()
                {
                    Type = TableWidthUnitValues.Dxa,
                    Width = "2400"
                }));

                Run ustępKomórkiTabeli = new Run(new Text(komórkiNagłówka[kolumna]));
                ustępKomórkiTabeli.RunProperties = new RunProperties()
                {
                    Bold = new Bold() { Val = OnOffValue.FromBoolean(true) },
                    Color = KolorWyróżniony
                };
                komórkaTabeli.Append(new Paragraph(ustępKomórkiTabeli));
                wierszNagłówka.Append(komórkaTabeli);
            }
            tabela.Append(wierszNagłówka);
        }

        for (int wiersz = 0; wiersz < komórki.GetLength(1); wiersz++)
        {
            TableRow wierszTabeli = new TableRow();
            for (int kolumna = 0; kolumna < komórki.GetLength(0); kolumna++)
            {
                TableCell komórkaTabeli = new TableCell();
                komórkaTabeli.Append(new TableCellProperties(new TableCellWidth()
                {
                    Type = TableWidthUnitValues.Dxa, Width = "2400" //szerokość komórki
                }));
                komórkaTabeli.Append(
```

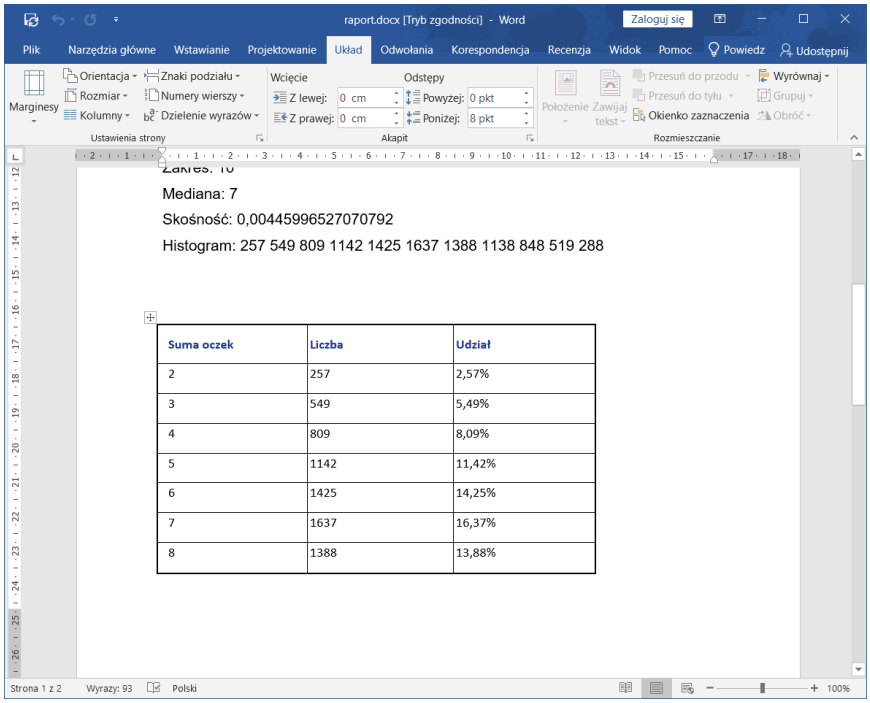
```
                    new Paragraph(
                        new Run(
                            new Text(komórki[kolumna, wiersz]))); //tekst w komórce
                wierszTabeli.Append(komórkaTabeli);
            }
            tabela.Append(wierszTabeli);
        }

        return tabela;
    }
```

Metodę `twórzTabelę` wykorzystamy, aby dodać do dokumentu tabelę z trzema kolumnami zawierającymi kolejno: możliwe wartości sumy oczek na dwóch kostkach (liczby od 2 do 12), liczbę wystąpień tych wartości oraz procent wystąpień w ogólnej liczbie rzutów kostkami. Informacje te umieszczam w tablicy o nazwie `komórki`, którą zapełniam w pętli `for`. Dodatkowo do metody `twórzTabelę` przesyłam trójelementową tablicę z tytułami kolumn. Pokazuje to listing 26.10. Efekt widoczny jest na rysunku 26.6.

Listing 26.10. Przygotowywanie danych dla tabeli i jej tworzenie

```
    private static void wyślijDokumentDoStrumienia(
        ParametryStatystyczneZHistogramem parametryStatystyczne,
        Stream stream, IFormatProvider formatProvider)
    {
        //tworzenie dokumentu Worda
        using (WordprocessingDocument dokumentWorda = WordprocessingDocument.Create(
            stream, WordprocessingDocumentType.Document, true))
        {
            MainDocumentPart głównaCzęśćDokumentu = dokumentWorda.AddMainDocumentPart();
            głównaCzęśćDokumentu.Document = new Document();

            Body ciałoDokumentu = new Body();
            głównaCzęśćDokumentu.Document.AppendChild(ciałoDokumentu);

            //rysunek
            ...

            //tabela
            string[,] komórki = new string[3, parametryStatystyczne.Histogram.Length];
            for (int i = 0; i < parametryStatystyczne.Histogram.Length; ++i)
            {
                komórki[0, i] = (i + 2).ToString(formatProvider);
                komórki[1, i] = parametryStatystyczne.Histogram[i].ToString(formatProvider);
                komórki[2, i] = $"{(parametryStatystyczne.Histogram[i] /
    parametryStatystyczne.Rozmiar):p2}";
            }
            Table tabela = twórzTabelę(komórki,
                                       new string[] { "Suma oczek", "Liczba", "Udział" });
            ciałoDokumentu.Append(tabela);

            dokumentWorda.Close(); //zamknięcie dokumentu
```

```
        }


        stream.Flush();
    }
```



Suma oczek	Liczba	Udział
2	257	2,57%
3	549	5,49%
4	809	8,09%
5	1142	11,42%
6	1425	14,25%
7	1637	16,37%
8	1388	13,88%

Rysunek 26.6. Tabela wstawiona do dokumentu

Strumień w pamięci

Wspomniałem wcześniej, że rozdzieliłem metodę przesyłającą dokument do strumienia od metody zapisującej zawartość tego strumienia do pliku, ponieważ sam strumień może zostać użyty w aplikacji także w inny sposób. Listing 26.11 zawiera przykład, w którym tworzę strumień przechowywany w pamięci (obiekt typu `MemoryStream`). Przekazuję go do metody `wyślijDokumentDoStrumienia`, która umieszcza w nim dokument.

Listing 26.11. Zapisanie dokumentu w pamięci

```
public static MemoryStream PobierzStrumieńDocx(
    this ParametryStatystyczneZHistogramem parametryStatystyczne,
    IFormatProvider formatProvider = null)
{
    if (formatProvider == null) formatProvider = new CultureInfo("pl-PL");
    MemoryStream strumieńWPamięci = new MemoryStream();
    wyślijDokumentDoStrumienia(parametryStatystyczne, strumieńWPamięci, formatProvider);
    //strumieńWPamięci.Close(); //nie należy zamykać strumienia, jeżeli ma być użyty
    return memoryStream;
}
```

Materiał z książki Jacka Matulewskiego „C#. Lekcje programowania. Praktyczna nauka programowania dla platform .NET i .NET Core” dla studentów kursu .NET 2023/2024. Proszę o nieudostępnianie osobom spoza kursu.

Statystyka

Trzeci przykład jest mniej typowy, bo nie korzysta z zasadniczych zalet paradygmatu programowania obiektowego. Jak pamiętamy, w rozdziale 7. zdefiniowaliśmy zbiór metod służących do obliczania podstawowych parametrów statystycznych danych zebranych w tablicy liczb `double`. Każda z tych metod działa w znacznym stopniu niezależnie, otrzymując wszystkie potrzebne dane przez głowę i zwracając obliczone parametry przez wartość. Czy wobec tego w tym przypadku ma sens definiowanie dla nich osobnej klasy? Tak. Możemy zdefiniować klasę statyczną, która zbiera wszystkie te metody w jednym miejscu, ułatwiając ich wywoływanie z różnych miejsc kodu, a dodatkowo ułatwiając ich przenoszenie do innych projektów, szczególnie jeżeli przygotujemy bibliotekę. Ponadto w następnym rozdziale pójdziemy o krok dalej i wykorzystamy tę klasę do definicji tzw. rozszerzeń (lub inaczej metod rozszerzających), które jeszcze bardziej ułatwią używanie tych metod.

Biblioteka. Interfejs IEnumerable<>

Aby trzeci przykład nie był tylko kopiowaniem gotowych metod do klasy statycznej, zmienimy także typ przyjmowanego przez nie argumentu z tablicy liczb `double` na `IEnumerable<double>`. To interfejs implementowany przez wszystkie kolekcje platformy .NET. Metody będą zatem mogły przyjmować różne typy kolekcji, a nie tylko tablice. To jednak utrudni nam zadanie, bo interfejs ten nie definiuje metody, która pozwala na odczytanie elementu o podanym indeksie[4]. To wyklucza w praktyce użycie pętli `for`. Zamiast tego mamy zadeklarowany w interfejsie iterator i możliwość użycia pętli `foreach`. Stąd np. w metodzie `Ekstrema` pojawi się bezpośrednie odwołanie do iteratora i jego metody `MoveNext` „napędzającej” pętlę `while` po wszystkich elementach kolekcji.

1. Stwórzmy nowy projekt i rozwiązanie korzystając z szablonu *Biblioteka klas (.NET Standard)* i nazwijmy go *Statystyka*. Tego projektu nie będzie można uruchomić, ale potem do rozwiązania dodamy projekt aplikacji konsolowej, w której będziemy testować tę bibliotekę.
2. W projekcie *Statystyka* zmieniamy nazwę pliku *Class1.cs* na *Statystyka.cs* (można to zrobić w podoknie *Eksplorator rozwiązań* korzystając z klawisza *F2*).
3. Kopiujemy do niego metody zdefiniowane w rozdziale 7. Listing 14.18 zawiera ich kod z wyróżnieniami w miejscach, w których zostały zmienione, aby uwzględnić że dane przekazywane są przez interfejs `IEnumerable<double>`, a nie w postaci tablicy `double[]`. Dodana została też nowa metoda `Liczba` ustalająca liczbę elementów w kolekcji (zamiennik metody `Count` z LINQ) oraz metoda `KopiujDoTablicy` przenosząca elementy z kolekcji do tablicy (odpowiednik metody `ToArray`).

Listing 14.18. Strasznie długi kod, który jest na szczęście tylko kopią kodu z rozdziału 7. z wyróżnionymi zmianami

```
using System;
using System.Collections.Generic;


namespace Statystyka
{
    public static class Statystyka
    {
        public static double Suma(IEnumerable<double> wartości)
        {
            double suma = 0;
            foreach (double wartość in wartości) suma += wartość;
            return suma;
        }
```

[4] Modyfikując te metody będę starał się nie korzystać z metod rozszerzających zdefiniowanych w ramach LINQ, skoro tworzymy jego niby-zastępnik.

```
        public static int Liczba(IEnumerable<double> wartości)
        {
            int liczba = 0;
            foreach (double wartość in wartości) liczba++;
            return liczba;
        }

        public static double Średnia(IEnumerable<double> wartości)
        {
            if (wartości == null)
                throw new ArgumentNullException("Przesłano obiekt pusty");
            int liczba = Liczba(wartości);
            if (liczba == 0) throw new ArgumentException("W tablicy nie ma elementów");
            return Suma(wartości) / liczba;
        }

        //tu nie korzystam z metody Suma, żeby uniknąć dwóch pętli zamiast jednej
        public static double Średnia2(IEnumerable<double> wartości)
        {
            if (wartości == null) throw new ArgumentNullException("Przesłano obiekt
pusty");
            int liczba = 0;
            double suma = 0;
            foreach (double wartość in wartości)
            {
                liczba++;
                suma += wartość;
            }
            if (liczba == 0) throw new ArgumentException("W tablicy nie ma elementów");
            return suma / liczba;
        }

        public static double Wariancja(IEnumerable<double> wartości)
        {
            double średnia = Średnia(wartości);
            double wariancja = 0;
            foreach (double wartość in wartości)
            {
                double odchylenie = wartość - średnia;
                wariancja += odchylenie * odchylenie;
            }
            return wariancja / Liczba(wartości);
        }

        public static double OdchylenieStandardowe(IEnumerable<double> wartości)
        {
```

```
        return Math.Sqrt(Wariancja(wartości));
    }

    public static void Ekstrema(IEnumerable<double> wartości,
                                out double minimum, out double maksimum)
    {
        if (wartości == null)
            throw new ArgumentNullException("Przesłano obiekt pusty");
        if (Liczba(wartości) == 0)
            throw new ArgumentException("W tablicy nie ma elementów");

        IEnumerator<double> iterator = wartości.GetEnumerator();
        iterator.MoveNext();
        double wartości0 = iterator.Current;
        minimum = wartości0;
        maksimum = wartości0;
        while (iterator.MoveNext())
        {
            double element = iterator.Current;
            if (element < minimum)
            {
                minimum = element;
            }
            if (element > maksimum)
            {
                maksimum = element;
            }
        }
    }

    public static double Zakres(IEnumerable<double> wartości)
    {
        double minimum, maksimum;
        Ekstrema(wartości, out minimum, out maksimum);
        return maksimum - minimum;
    }

    public static double[] KopiujDoTablicy(IEnumerable<double> wartości)
    {
        double[] tablica = new double[Liczba(wartości)];
        int i = 0;
        foreach (double wartość in wartości)
        {
            tablica[i] = wartość;
            i++;
        }
```

```
            return tablica;
        }

        public static double Mediana(IEnumerable<double> wartości)
        {
            if (wartości == null)
                throw new ArgumentNullException("Przesłano obiekt pusty");
            if (Liczba(wartości) == 0)
                throw new ArgumentException("W tablicy nie ma elementów");

            double[] _wartości = KopiujDoTablicy(wartości);
            Array.Sort(_wartości);

            if (_wartości.Length % 2 != 0) return _wartości[_wartości.Length / 2];
            else return (_wartości[_wartości.Length / 2 - 1] +
_wartości[_wartości.Length / 2]) / 2.0;
        }

        public static double Skośność(IEnumerable<double> wartości)
        {
            return (Średnia(wartości) - Mediana(wartości)) /
OdchylenieStandardowe(wartości);
        }

        public static int[] Histogram(IEnumerable<double> wartości,
                                      int liczbaPrzedziałów)
        {
            double rozmiarPrzedziału = Zakres(wartości) / liczbaPrzedziałów;
            if (rozmiarPrzedziału == 0) throw new Exception("Niepoprawne dane");

            double minimum, maksimum;
            Ekstrema(wartości, out minimum, out maksimum);

            int[] histogram = new int[liczbaPrzedziałów];
            foreach (double wartość in wartości)
            {
                int i = (int)((wartość - minimum) / rozmiarPrzedziału);
                if (i == liczbaPrzedziałów) i = liczbaPrzedziałów - 1; //skrajny punkt
wkładamy do ostatniego przedziału
                histogram[i]++;
            }
            return histogram;
        }
    }
}
```

Zwróćmy uwagę, że oprócz metody `Średnia`, zdefiniowana jest również metoda `Średnia2`, w której zamiast używać metod `Suma` i `Liczba`, co *de facto* oznacza wykonanie dwóch pętli, wykonujemy tylko jedną pętlę obliczając równocześnie liczbę elementów i sumę wartości.

Zwróćmy też uwagę na zmianę w metodzie znajdującej ekstrema. W odróżnieniu od metody z rozdziału 7. metoda ta zwraca wartości minimum i maksimum, a nie ich indeksy. Indeksy bowiem na niewiele by nam się zdały, skoro interfejs `IEnumerable<>` nie gwarantuje obecności operatora `[]`.