LABORATORIUM INTELIGENTNYCH MASZYN I SYSTEMÓW
POLITECHNIKA WARSZAWSKA
WYDZIAŁ ELEKTRYCZNY

FOTO MOZAIKA

Jakub Zdziebłowski mayday.member@interia.pl

Semestr VII, informatyka/inżynieria komputerowa

30.01.2009

Spis treści

Wprowadzenie

Założenia

Celem projektu było stworzenie aplikacji generującej mozaikę składającą się ze zdjęć (zwanych segmentami) odtwarzających w jak najdokładniejszy sposób obraz bazowy. Obrazy wykorzystywane jako kafelki tworzą kolekcję, która jest dobierana w trakcie procesu dopasowywania do fragmentu obrazu bazowego na podstawie podobieństwa barw. Pobocznym elementem projektu było zapoznanie się z różnymi elementami języka C# w wersji dla .Net Framework 3.5 – w szczególności wykorzystania elementów do tworzenia graficznych interfejsów użytkownika (biblioteka GDI+), wielowątkowości i obsługi zdarzeń.

Możliwe rozwiązania

Problem segmentacji

Pierwszym problemem przy tworzeniu aplikacji jest odpowiednie podzielenie obrazu źródłowego na segmenty i wyznaczenie parametrów tych segmentów. Parametrami opisującymi segment mogą być składowe kolorów w segmencie lub parametry jasności, nasycenia i odcienia. Ważny jest również sposób wyznaczania tych parametrów – możemy wyznaczać je na podstawie wartości uśrednionej w segmencie lub na bazie wartości dominującej.

Zagadnienia związane z doborem segmentów

Kolejnym zagadnieniem rozpatrywanym w ramach projektu jest dobór odpowiedniego sposobu wyszukiwania elementów pasujących. Dopasowanie możemy wykonywać na podstawie odchyłki średniej parametrów bazy lub na podstawie podstawie ilości poprawek w obecnym obrazie w stosunku do ostatniego najlepiej dobranego obrazu. W ramach doboru segmentu możemy zwracać również uwagę na rozmiary segmentu. Obraz generowany może być zbudowany z segmentów kwadratowych (aspekt 1:1) lub o aspekcie zbliżonym do obrazu bazowego (portret, kwadrat, krajobraz). W pierwszym przypadku liczbę segmentów ustawić powinniśmy zgodnie z aspektem obrazu bazowego, w drugim powinna być ona równa w pionie i w poziomie. Dobór ilości segmentów wpływa na dokładność odwzorowania bazy – im więcej, tym łatwiej stworzyć w miarę dokładny obraz, przy ‚mniejszej liczbie segmentów ważne jest dobre dopasowanie kolorystyczne poszczególnych elementów. Jednym z problemów jest też sąsiedztwo elementów – jeżeli kilka segmentów obok siebie będzie miało zbliżoną indeksację, może się zdarzyć, że dopasowany do nich zostanie ten sam obraz.

Zagadnienia związane z segmentami.

W trakcie wczytywania kolekcji parametry segmentów mogą być liczone „w locie” lub mogą być one przechowywane na dysku i być wczytywane podczas ładowania kolekcji.

Wybór rozwiązania

Zadanie nie posiadało jednego z góry narzuconego rozwiązania, a ilość podproblemów była również dosyć duża. Algorytm, który znajduje się w wersji końcowej jest wypadkową różnych metod, które przyjąłem w czasie projektowania aplikacji. Niektóre z nich powodowane były ograniczeniami czasowymi (zarówno na projekt, jak i na testy). Dużym problemem przy implementacji tego zadania było testowanie i ocena jakościowa otrzymanych wyników. Proces tworzenia foto-mozaiki jest procesem długim, zajmującym kilka minut (w wersji pierwotnej, nie zoptymalizowanej pod kątem efektywnego wykorzystania struktur danych czas ten sięgał nawet godzin), dodatkowo, nie dało się przeprowadzić testów jednostkowych na tworzonych obrazach, gdyż zasadniczym kryterium działania aplikacji są walory estetyczne.

Wybór języka implementacji

Projekt nie wymagał stosowania żadnych specjalistycznych bibliotek, więc nie było ograniczeń związanych z wybraniem języka, w którym mógł zostać on stworzony. W gronie rozpatrywanych przeze mnie języków pojawili się czterej kandydaci:

Java – niewątpliwą zaletą Javy jest jej otwarta architektura niosąca ze sobą mnogość darmowych elementów do wykorzystania, wieloplatformowość oraz fakt, że w poprzednich projektach uczelnianych tworzyłem projekty głównie w tej technologii. Przeciwko przemawiały powolność wykonywania kodu związana z architekturą maszyny wirtualnej oraz nienajlepiej zaprojektowany pakiet Swing, przez co tworzenie interfejsu graficznego mogłoby się niepotrzebnie skomplikować.

C# - dobra alternatywa dla Javy, oferuje podobną funkcjonalność w pakietach systemowych, a dodatkowo środowisko uruchomieniowe można przyśpieszyć przez wykonywanie instrukcji przy dostępie do pamięci niezarządzanej (unsafe). Dodatkowo biblioteka GDI+ pozwala na uzyskanie bardzo dobrych interfejsów użytkownika w łatwy sposób.

C++ - bardzo szybki, ale programista musi sam kontrolować cały proces przydzielania i zwalniania pamięci. Kolejnym utrudnieniem jest nie tak rozbudowana biblioteka standardowa.

Delphi – równie szybki jak C++ (kod kompilowany do kodu maszynowego) i pozwalający na łatwe projektowanie interfejsu (duża ilość komponentów VCL), jednak niewygodny w pisaniu ze względu na archaiczną składnię. Kolejnym minusem jest brak zarządzania pamięcią.

Założenia projektu

Aplikacja tworzona w języku C# (.Net Framework 2.0) z wykorzystaniem środowiska programistycznego Microsoft Visual Studio 2008. Projekt składa się z biblioteki dynamicznej DLL zajmującej się implementacją algorytmów związanych z segmentacją obrazów, doborem segmentów do bazy, indeksowaniem plików segmentów oraz z aplikacji typu stand-alone będąca studiem do tworzenia mozaik – główne okno jest formularzem MDI pozwalającym na otwieranie wielu obrazów, okna obrazów są luźno wzorowane na oknach znanych z popularnych programów graficznych takich jak Adobe Photoshop czy Corel Photo-Paint (interesowało mnie głównie podobieństwo odtwarzaniu powiększenia obrazu).

Realizacja projektu

Szczegółowy opis realizacji

Segmentacja obrazów

Pierwszym zadaniem w projekcie było stworzenie algorytmu segmentującego obrazy. Była to baza, od której można było wyprowadzić następne elementy aplikacji. Obraz możemy podzielić na dowolną liczbę segmentów nie przekraczającą jego wielkości. Segment jest prostokątnym wycinkiem obrazu. Algorytm segmentacji opiera się o ideę zbioru rozmytego (nie implementowałem logiki rozmytej, po prostu skorzystałem z samej idei takich zbiorów). Z każdego segmentu tworzona jest kolekcja elementów, które są rozkładem trójkątnym w przedziale [0,255] trzech zmiennych niezależnych (reprezentujących składowe RGB pikseli). Szerokość rozkładu jest kontrolowana przez parametr zwany przeze mnie rozmyciem przyjmujący wartości [0,127]. Indeksacja polega na przeiterowaniu po zbiorze pikseli w segmencie, a następnie:

Dodaniu nowego elementu do kolekcji, jeżeli takiego elementu nie było (z wagą 1 – czubek rozkładu) oraz do pozostałych elementów w kolekcji, z odpowiednią wagą wynikającą z rozkładu trójkątnego.

Jeżeli element istniał wcześniej, to dodawany jest do istniejącego elementu z wagą 1 i do pozostałych elementów z wagą odpowiednio wynikającą z rozkładu trójkątnego.

Następnie dla każdego elementu wyliczamy ilość wystąpień czyli sumę 0-255 z ilość wystąpień*waga wystąpienia. Indeksem segmentu jest element o największej wartości dla wszystkich 3 składowych.

Oprócz samej implementacji algorytmu należało napisać klasy zarządzające indeksowaniem pojedynczego pliku oraz całej kolekcji, a także klasę parsera, służącą do odczytywania pliku.

Budowanie mozaiki

Każdy segment dobierany jest do fragmentu bazy, która jest tablicą 2x2 zindeksowanych segmentów. Największym problemem przy budowie tego algorytmu było przekształcenie przestrzeni 12 wymiarowej (2x2x3 – ilość segmentów w badanym fragmencie x ilość parametrów w segmencie) na jednowymiarowy parametr określający jakość doboru obrazu. Przystosowanie w pierwszym kroku inicjowane jest wartościami wykraczającymi poza wartości w obrazie. Następnie dla każdego obrazu obliczana jest błąd przystosowania dla poszczególnych parametrów. Następnie błąd obecnego przystosowania porównywany jest z najlepszym znalezionym przystosowaniem – dla każdego porównanego segmentu porównywany jest każdy parametr – jeżeli się on poprawia, przypisywana

jest mu wartość +1, jeżeli się pogarsza, wartość -1. Jeżeli suma zmian jest większa od 2 (parametr dobrany eksperymentalnie), dopasowanie segmentu jest dodatnie, w przeciwnym wypadku, jest ono ujemne. Jeżeli suma dopasowań w obszarze jest większa od 0, to obraz uważany jest za lepsze dopasowanie.

Dopasowane obrazy zapisywane są w tablicy z nazwami plików. W kolejnym kroku tworzona jest pusta bitmapa, na której malowane są poszczególne segmenty. W pierwotnej wersji dla każdego segmentu tworzona była osobna miniatura obrazu, który miał być wklejony, jednak ponieważ przeskalowywanie obrazu jest procesem stosunkowo długim, to system ten został obecnie zmieniony na wydajniejszy. W każdym kroku pobierana jest nazwa kolejnego pliku wstawianego na segment, tworzona jest jego miniatura i wklejana w odpowiednie miejsca, a nazwa pliku w tablicy zmieniana jest na wartość null.

Opis implementacyjny

namespace GraphicProcessor

BudowniczySegmentowanyObraz

Klasa buildera tworząca instancje klasy segmentowany obraz

Properties:

Obraz – ustawia obraz do segmentowania

SegmentyWysokosc, **SegmentySzerokosc** – ilość segmentów w pionie i w poziomie

Rozmycie – poziom rozmycia

Metody publiczne:

Cancel() - ustawia flagę odwołania przygotowania obiektu segmentowania

SegmentowanyObraz **Segmentuj**() – tworzy instancję klasy segmentowany obraz.

static FileIndexer

Segmentuje wybrany obraz

metody publiczne:

SegmentowanyObraz **Indeksuj**(Image obraz, int ilSegmSzer, int ilSegmWys, int ilSegmSzerSmall, int ilSegmWysSmall, byte rozmycie) – segmentuje obraz.

TworzPlikKolekcji(string sciezka, int ilSegmSzer, int ilSegmWys, int ilSegmSzerSmall, int ilSegmWysSmall, byte rozmycie) – segmentuje obraz i zapisuje informacje o nim na dysku.

IndexDirectory

Segmentuje kolekcję obrazów

Properties:

ilPlikow – zwraca ilość plików do indeksowania

metody publiczne:

Indeksuj(string path, , int ilSegmSzer, int ilSegmWys, int ilSegmSzerSmall, int ilSegmWysSmall, byte rozmycie) – indeksuje wybrany katalog ze zdjęciami

Zdarzenia:

FileIdexed – wołane w momencie zaindeksowania pliku

IndexingStarted – wołane w momencie rozpoczęcia indeksowania

MosaiqueBuilder

Builder tworzący obraz z mozaikami

Properties:

Baza – obraz, z którego tworzona jest mozaika

IsOnCancellation – ustawia odwołanie tworzenia mozaiki

MapaPlikow – tablica 2-wymiarowa z nazwami plików, które stworzą segmenty

Metody publiczne:

BuildMosaique2() – dopasowuje segmenty, które przechowywane są w MapaPlikow

Zdarzenia:

StateChanged – wołany przy dopasowaniu segmentu

ImageGenerated – wołano po zakończeniu dopasowywania

MsqReader

Parser czytający plik z informacjami o segmentach w pliku graficznym

Metody publiczne:

ReadAllSegments(string file) – czyta wszystkie segmenty w pliku

ReadSmallSegments(string file) – czyta tylko informacje o segmentacji 2x2

MsqWriter

Klasa tworząca plik z informacjami o obrazie

Metody publiczne:

SegmentowanyObraz **WriteSegments**(PixelData[,] data, string destPath) – zapisuje informacje o obrazie

SegmentowanyObraz **WriteSegments**(PixelData[,] data, PixelData[,] smallData, string destPath) – zapisuje informacje razem z segmentami 2x2.

static Obszar
Klasa statyczna wyszukująca kolor segmentu

Internal static PixelData **ZnajdzKolorObszaru**(Bitmap bmp, byte rozmycie)

PixelData
Klasa przechowująca informacje o segmencie

Properties:

R,**G**,**B** – informacje o kolorze

PixelValueDictionary
Klasa wykorzystywana w trakcie indeksowania obszaru, przechowuje kolekcję zbiorów rozmytych

Properties:

PixelData **SredniaBarwaObszaru** – zwraca informacje o wyliczonym obszarze

Metody:

DodajPixel(byte r, byte g, byte b) – dodaje informację o kolejnym pikselu, dodaje nowe elementy do kolekcji i uzupełnia informacje w istniejących

SegmentowanyObraz
Klasa przechowująca informacje o obrazie – segmentach, nazwie pliku i nazwie pliku z informacjami (.msq)

Properties:

PixelData[,] **TablicaObszarow** – tablica z indeksami w poszczególnych obszarach

String **FileName** – nazwa pliku

String **MsqFileName** – nazwa pliku z informacjami

ZbiorRozmyty
Klasa przechowująca instancję zbioru rozmytego.

Properties:

R,G,B – informacje o wartości wierzchołków dla zmiennych czerwonej, niebieskiej i zielonej

WartoscR, WartoscG, WartoscB, srednia – zwraca informacje o wartości zbioru dla poszczególnych składowych i wartości średniej z nich

Metody:

DodajPunkt(byte R, byte G, byte B) – dodaje informacje do wartości zbioru

Namespace GPTest

GPTest zawiera elementy interfejsu graficznego

AboutBox1

Okienko informacji o aplikacji.

DirectoryIndexing

Forma zawierająca kontrolkę (FolderIndexingUserControl) do indeksowania plików.

FolderIndexingUserControl

kontrolka służąca do indeksowania plików. Indeksowanie odbywa się w osobnym wątku, dzięki czemu, użytkownik nie ma wrażenia zamrożenia okienka podczas wykonywania. W kontrolce użyte jest okienko do otwierania katalogu oraz kontra oka z miniaturką obecnie przetwarzanego obrazu.

OknoGenerowania2

Okno generowania służy do podawania parametrów dla wygenerowanego pliku: ilości segmentów w pionie i w poziomie, ich wielkości oraz wyboru katalogów, z których kafelki mają być wykorzystane przegenerowaniu mozaiki. Podobnie jak przy indeksowaniu, proces generowania uruchamiany jest w osobnym wątku, aby nie zamrozić GUI. Dodatkowo, zrównolegleniu poddana została operacja indeksowania obrazu bazowego i zbierania segmentów (choć testy na 2 rdzeniowej maszynie nie wykazały przyspieszenia kodu).

OknoObrazu2b

Okno, w którym prezentowany jest plik graficzny. Grafikę można przybliżać za pomocą kliknięcia lewym przyciskiem myszy na niej i oddalać prawym przyciskiem.

PropertiesWindow

Okno ustawień aplikacji – można w nim ustawiać ścieżkę do katalogu kolekcji i katalogu, w którym domyślnie zapisywane są wygenerowane obrazy.

StudioWindow

Główne okno aplikacji, zaprojektowane jako kontener MDI, można w nim otwierać wiele plików graficznych. Zawiera menu, w którym możemy otwierać pliki, wywoływać okno generowania mozaik, okno indeksowania segmentów, ustawiać wewnątrz niego okna.

Opis uruchomieniowy

i) Pierwszym krokiem jest utworzenie katalogu do przechowywania segmentów. Katalog w obecnej wersji należy stworzyć własnoręcznie.

ii) Uruchamiamy aplikację i wybieramy Plik→Ustawienia… W oknie ustawień ustawiamy ścieżkę do kolekcji na katalog, który przed chwilą utworzyliśmy

iii) Z menu Kolekcja wybieramy Indeksuj. W oknie indeksowania wybieramy katalog, w którym mamy obrazki, następnie wpisujemy nazwę katalogu, w jakim mają być one przechowywane w kolekcji, po czym klikamy Indeksuj…

iv) Z menu plik wybieramy Otwórz… W oknie otwierania wybieramy plik graficzny, z którego chcemy utworzyć mozaikę, po czym klikamy przycisk Otwórz.

v) Z menu plik wybieramy przycisk Generuj… Otworzy się okno generowania.

vi) Wybieramy, który zestaw kafelków chcemy użyć, ilość kafelków i ich rozmiar, po czym klikamy generuj.

vii) Po wygenerowaniu obrazu możemy zapisać klikając na Zapisz w menu Plik.

Podsumowanie i wnioski

Projekt dotyczył generowania foto mozaik ze zdjęcia bazowego i kolekcji obrazów – segmentów.

Projekt zorganizowany został z wykorzystaniem platformy .Net Framework 2.0 w środowisku Microsoft Visual Studio 2008.

Sukcesem projektu jest osiągnięcie zadowalających obrazów testowych w całkiem krótkim czasie. Kolejnym sukcesem jest zapoznanie się z wieloma metodami wytwarzania oprogramowania w języku C# - wykorzystanie komponentów GUI, korzystanie z klasy BackgroundWorker do implementacji wielowątkowości, tworzenie zdarzeń i obiektów delegatów.

Główne cele zostały zrealizowane, jednak zabrakło czasu na pewne ściśle kosmetyczne sprawy, jak stworzenie dobrze działającego okna do wyświetlania plików graficznych, który można by wykorzystać w innych aplikacjach.

Możliwości rozbudowy

Podstawowym problemem przy dalszym rozwoju tej aplikacji pozostaje szybkość działania – możliwe jest zrównoleglenie większej ilości procesów, jak choćby stworzenie puli wątków indeksujących kilka segmentów w tym samym czasie, tak samo generowanie obrazu za pomocą kilku wątków. Poprawy wymaga również okno pokazywania obrazu – należałoby wykorzystać szybsze narzędzia, jak dostęp do kodu niezarządzanego wyświetlanej bitmapy. Kolejną zmianą mogłaby być zmiana interfejsu użytkownika na interfejs pisany w WPF.

Bibliografia

- Perry, S. C., Core C# i .Net, Pearson Education Inc, 2006, Helion. Rozdziały:

  5. Przetwarzanie tekstu i plikowe operacje wejścia-wyjścia w języku C#

  6. Budowanie aplikacji Windows Forms

  7. Kontrolki formularzy Windows Forms

  8. Elementy graficzne biblioteki GDI+ w technologii .Net

  13. Programowanie asynchroniczne i wielowątkowość

- Sivakumar, N., Events and event handling In C#, http://www.codeproject.com/KB/cs/csevents01.aspx

- Zielwak, M., Przetwarzanie obrazów, http://www.centrumxp.pl/dotNet/1506,1,Przetwarzanie_obraz%C3%B3w.aspx

- http://msdn.microsoft.com

Dodatek

BudowniczySegmentowanyObraz.cs

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Drawing.Imaging;

namespace GraphicProcessor
{
    class BudowniczySegmentowanyObraz
    {
        #region Pola
        private Image image;
        private int segmentyWysokosc;
        private int segmentySzerokosc;
        private byte rozmycie;
        private Boolean canceling;
        #endregion

        #region Properties
        public Image Obraz
        {
            set
            {
                image = value;
            }
        }
        public int SegmentyWysokosc
        {
            set
            {
                segmentyWysokosc = value;
            }
        }
        public int SegmentySzerokosc
        {
            set
            {
                segmentySzerokosc = value;
```

```
            }
        }
        public byte Rozmycie
        {
            set
            {
                rozmycie = value;
            }
        }
        public void Cancel()
        {
            canceling = true;
        }
        #endregion

        #region Metody publiczne
        public SegmentowanyObraz Segmentuj()
        {
            PixelData[,] koloryObszarow = new PixelData[segmentySzerokosc, segmentyWysokosc];
            if (image.PixelFormat != PixelFormat.Format24bppRgb)
            {
                Image bmpTmp = new Bitmap(image.Width, image.Height,
PixelFormat.Format24bppRgb);
                Graphics g = Graphics.FromImage(bmpTmp);
                g.DrawImage(image, image.Width, image.Height);
                g.Dispose();
                image = bmpTmp;
            }
            int wysObszaru = (int)decimal.Round((decimal)(image.Height / segmentyWysokosc),
0);
            int szerObszaru = (int)decimal.Round((decimal)(image.Width / segmentySzerokosc),
0);
            if (wysObszaru == 0)
            {
                wysObszaru++;
            }
            if (szerObszaru == 0)
            {
                szerObszaru++;
            }
            Bitmap resized = new Bitmap(segmentySzerokosc * szerObszaru, segmentyWysokosc *
wysObszaru);
            Graphics gr = Graphics.FromImage(resized);
            Rectangle dest = new Rectangle(0, 0, resized.Width, resized.Height);
            Rectangle src = new Rectangle(0, 0, image.Width, image.Height);
            gr.DrawImage(image, dest, src, GraphicsUnit.Pixel);
            gr.Dispose();
            image = resized;


            for (int x = 0; x < segmentySzerokosc; x++)
            {

                for (int y = 0; y < segmentyWysokosc; y++)
                {
                    if (!canceling)
                    {
                        int szer = 0;
                        if (x == segmentySzerokosc - 1)
                        {
                            szer = image.Width - x * szerObszaru;
                        }
                        else
                        {
                            szer = szerObszaru;
                        }
                        int wys = 0;
                        if (y == segmentyWysokosc - 1)
                        {
                            wys = image.Height - y * wysObszaru;
                        }
                        else
                        {
                            wys = wysObszaru;
                        }
```

```
                        //DateTime time1 = DateTime.Now;
                        //nowa wersja -LockBits
                        Rectangle r = new Rectangle(x * szerObszaru, y * wysObszaru, szer,
wys);
                        BitmapData data = resized.LockBits(r, ImageLockMode.ReadOnly,
PixelFormat.Format24bppRgb);
                        koloryObszarow[x, y] = Obszar.ZnajdzKolorObszaru(data, rozmycie);
                        resized.UnlockBits(data);
                        //stara wersja z resize'em
                        //Rectangle r = new Rectangle(x * szerObszaru, y * wysObszaru, szer,
wys);
                        //Bitmap tmp = new Bitmap(szerObszaru, wysObszaru);
                        //Graphics gTmp = Graphics.FromImage(tmp);
                        //gTmp.DrawImage(resized, new Rectangle(0, 0, szerObszaru,
wysObszaru), r, GraphicsUnit.Pixel);
                        //gTmp.Dispose();
                        //koloryObszarow[x, y] = Obszar.ZnajdzKolorObszaru(tmp, rozmycie);
                        //Console.WriteLine("Segment " + (x * segmentyWysokosc + y + 1) + "/"
+ (segmentySzerokosc * segmentyWysokosc));

                        //TimeSpan time = DateTime.Now.Subtract(time1);
                        //Console.WriteLine(": " + time.ToString());
                    }
                }
            }
            resized.Dispose();
            if (canceling)
            {
                canceling = false;
                return null;
            }
            return new SegmentowanyObraz(koloryObszarow);
        }
        #endregion
    }
}
```

FileIndexer

```
using System;
using System.IO;
using System.Drawing;
using System.Drawing.Imaging;

namespace GraphicProcessor
{
    public static class FileIndexer
    {
        public static SegmentowanyObraz Indeksuj(Image obraz, int ilSegmSzer, int ilSegmWys,
byte rozmycie)
        {
            if (obraz == null)
            {
                throw new ArgumentNullException("Parametr obraz musi byc rozny od null");
            }
            else
            {
                SegmentowanyObraz segObraz = null;
                if (ilSegmSzer > obraz.Width || ilSegmWys > obraz.Height)
                {
                    throw new ResolutionTooSmallException("Ilosc segmentow wieksza od
rozdzielczosci pliku.", null);
                }
                else
                {
                    BudowniczySegmentowanyObraz builder = new BudowniczySegmentowanyObraz();
                    builder.Obraz = obraz;
                    builder.SegmentySzerokosc = ilSegmSzer;
                    builder.SegmentyWysokosc = ilSegmWys;
                    builder.Rozmycie = rozmycie;
                    segObraz = builder.Segmentuj();
                }
                return segObraz;
            }
        }
```

```
        public static void TworzPlikKolekcji(string sciezka, int ilSegmSzer, int ilSegmWys,
int ilSegmSzerSmall, int ilSegmWysSmall, byte rozmycie)
        {
            if (String.IsNullOrEmpty(sciezka))
            {
                throw new ArgumentNullException("Pusty parametr pliku.");
            }
            else if (!File.Exists(sciezka))
            {
                throw new FileNotFoundException("Plik nie istnieje");
            }
            else
            {
                FileInfo fInfo = new FileInfo(sciezka);
                Bitmap bmp = new Bitmap(sciezka);

                string indexFilePath = fInfo.FullName + ".msq";
                if (ilSegmSzer >= bmp.Width || ilSegmWys >= bmp.Height)
                {
                    throw new ResolutionTooSmallException("Ilosc segmentow wieksza od
rozdzielczosci pliku.", sciezka);
                }
                else
                {
                    if (bmp.Width > ilSegmSzer || bmp.Height > ilSegmWys)
                    {
                        int newWys = 0;
                        int newSzer = 0;
                        if (bmp.Width > bmp.Height)
                        {
                            newSzer = Resources.maxRozmiarObrazu;
                            newWys = bmp.Height * newSzer / bmp.Width;
                        }
                        else
                        {
                            newWys = Resources.maxRozmiarObrazu;
                            newSzer = bmp.Width * newWys / bmp.Height;
                        }
                        Bitmap tmp = new Bitmap(newSzer, newWys, PixelFormat.Format24bppRgb);
                        Graphics g = Graphics.FromImage(tmp);
                        g.DrawImage(bmp, new Rectangle(0, 0, tmp.Width, tmp.Height), new
Rectangle(0, 0, bmp.Width, bmp.Height), GraphicsUnit.Pixel);
                        g.Dispose();
                        bmp = tmp;
                    }
                    BudowniczySegmentowanyObraz builder = new BudowniczySegmentowanyObraz();
                    builder.Obraz = bmp;
                    builder.SegmentyWysokosc = ilSegmWys;
                    builder.SegmentySzerokosc = ilSegmSzer;
                    builder.Rozmycie = rozmycie;
                    SegmentowanyObraz sO = builder.Segmentuj();
                    PixelData[,] segmenty = sO.TablicaObszarow;
                    builder.SegmentyWysokosc = ilSegmWysSmall;
                    builder.SegmentySzerokosc = ilSegmSzerSmall;
                    builder.Obraz = bmp;
                    sO = builder.Segmentuj();
                    PixelData[,] segmentySmall = sO.TablicaObszarow;
                    bmp.Dispose();
                    MsqWriter.WriteSegments(segmenty, segmentySmall, indexFilePath);
                }
            }
        }

        public static void TworzPlikKolekcji(string sciezka, int ilSegmSzer, int ilSegmWys,
byte rozmycie)
        {
            if (String.IsNullOrEmpty(sciezka))
            {
                throw new ArgumentNullException("Pusty parametr pliku.");
            }
            else if (!File.Exists(sciezka))
            {
                throw new FileNotFoundException("Plik nie istnieje");
            }
```

```
            else
            {
                FileInfo fInfo = new FileInfo(sciezka);
                Bitmap bmp = new Bitmap(sciezka);

                string indexFilePath = fInfo.FullName + ".msq";
                if (ilSegmSzer >= bmp.Width || ilSegmWys >= bmp.Height)
                {
                    throw new ResolutionTooSmallException("Ilosc segmentow wieksza od
rozdzielczosci pliku.", sciezka);
                }
                else
                {
                    BudowniczySegmentowanyObraz builder = new BudowniczySegmentowanyObraz();
                    builder.Obraz = bmp;
                    builder.SegmentyWysokosc = ilSegmWys;
                    builder.SegmentySzerokosc = ilSegmSzer;
                    builder.Rozmycie = rozmycie;
                    SegmentowanyObraz sO = builder.Segmentuj();
                    bmp.Dispose();
                    MsqWriter.WriteSegments(sO.TablicaObszarow, indexFilePath);
                }
            }
        }
    }
}
```

IndexDirectory

```
using System;
using System.IO;

namespace GraphicProcessor
{
    public class IndexDirectory
    {
        public IndexDirectory(string path)
        {
            this.path = path;
            ilPlikow = 0;
            ObliczIlPlikow(path, ref ilPlikow);
        }

        private string path;

        private int ilPlikow;
        public int IlPlików
        {
            get
            {
                return ilPlikow;
            }
        }

        /// <summary>
        /// oblicza ilośc plikow do raportowania procesu
        /// </summary>
        /// <param name="path">obecnie badany katolog</param>
        /// <param name="licznik">licznik plikow</param>
        private void ObliczIlPlikow(string path, ref int licznik)
        {
            foreach (string d in Directory.GetDirectories(path))
            {
                ObliczIlPlikow(d, ref licznik);
            }
            licznik += Directory.GetFiles(path).Length;
        }

        public void Indeksuj(int ilSegmSzer, int ilSegmWys, int ilSegmSzerSmall, int
ilSegmWysSmall, byte rozmycie, ref bool cancelFlag)
        {
            Indeksuj(path, ilSegmSzer, ilSegmWys, ilSegmSzerSmall, ilSegmWysSmall, rozmycie,
ref cancelFlag);
        }
```

```
        private void Indeksuj(string path, int ilSegmSzer, int ilSegmWys, int
ilSegmSzerSmall, int ilSegmWysSmall, byte rozmycie, ref bool cancelFlag)
        {
            if (String.IsNullOrEmpty(path))
            {
                throw new ArgumentNullException("Podano pusty parametr.");
            }
            else if (!Directory.Exists(path))
            {
                throw new ArgumentException("Lokalizacja nie istnieje.");
            }
            else
            {
                foreach (string locPath in Directory.GetDirectories(path))
                {
                    if(!cancelFlag)
                    Indeksuj(locPath, ilSegmSzer, ilSegmWys, ilSegmSzerSmall, ilSegmWysSmall,
rozmycie, ref cancelFlag);
                }
                foreach (string locPath in Directory.GetFiles(path))
                {
                    if (!cancelFlag)
                    {
                        if (!File.Exists(locPath + ".msq"))
                        {
                            OnIndexingStarted(locPath);
                            FileIndexer.TworzPlikKolekcji(locPath, ilSegmSzer, ilSegmWys,
ilSegmSzerSmall, ilSegmWysSmall, rozmycie);
                            OnFileIndexed((1 / (float)ilPlikow * 100));
                        }
                    }
                }
            }
        }
        public delegate void IndxingStartedHandler(string file);

        public event IndxingStartedHandler IndexingStarted;

        public delegate void IndexHandler(float percent);

        public event IndexHandler FileIdexed;

        protected void OnIndexingStarted(string file)
        {
            if (IndexingStarted != null)
            {
                IndexingStarted(file);
            }
        }

        protected void OnFileIndexed(float percent)
        {
            if (FileIdexed != null)
            {
                FileIdexed(percent);
            }
        }
    }
}
```

MosaiqueBuilder

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

namespace GraphicProcessor
{
    /// <summary>
    /// Klasa budująca mozaikę ze zdjęcia
    /// </summary>
    public class MosaiqueBuilder
    {
        #region constructor
        private MosaiqueBuilder()
```

```
        {
            listaKafelkow = new List<SegmentowanyObraz>();
            mapaPlikow = null;
            isOnCancellation = false;
        }

        //MosaiqueBuilder - singleton implementation
        #region Singleton
        private static MosaiqueBuilder instance;
        public static MosaiqueBuilder Instance
        {
            get
            {
                if (instance == null)
                {
                    instance = new MosaiqueBuilder();
                }
                return instance;
            }
        }
        #endregion
        #endregion

        public void addSegment(SegmentowanyObraz obraz)
        {
            listaKafelkow.Add(obraz);
        }

        /// <summary>
        /// dodaje nowe kafelki do dopasowywanej kolekcji
        /// </summary>
        /// <param name="segmentedPictures">kolekcja kafelków</param>
        public void addSegments(IEnumerable<SegmentowanyObraz> segmentedPictures)
        {
            listaKafelkow.AddRange(segmentedPictures);
        }

        #region Properties

        public SegmentowanyObraz baza;
        /// <summary>
        /// Obrazek, z którego tworzona jest mozaika
        /// </summary>
        public SegmentowanyObraz Baza
        {
            get
            {
                return baza;
            }
            set
            {
                baza = value;
                int x = (baza.TablicaObszarow.GetLength(0) / 2) * 2;
                int y = (baza.TablicaObszarow.GetLength(1) / 2) * 2;
                mapaPlikow = new String[x / ROZMIAR, y / ROZMIAR];
            }
        }

        private static bool isOnCancellation;
        public bool IsOnCancellation
        {
            get
            {
                return isOnCancellation;
            }
            set
            {
                isOnCancellation = value;
            }
        }

        private List<SegmentowanyObraz> listaKafelkow;

        private String[,] mapaPlikow;
        public String[,] MapaPlikow
```

```
        {
            get
            {
                return mapaPlikow;
            }
        }
        #endregion

        private const int ROZMIAR = 2;

        public void BuildMosaique2()
        {
            if (baza == null)
            {
                throw new ArgumentNullException("Baza");
            }
            PixelData[,] segmentyBazy = baza.TablicaObszarow;
            String plikDopasowania = null;
            int iMax = (segmentyBazy.GetLength(0) / 2) * 2;
            int jMax = (segmentyBazy.GetLength(1) / 2) * 2;
            for (int i = 0; i < iMax; i += ROZMIAR)
            {
                for (int j = 0; j < jMax; j += ROZMIAR)
                {
                    if (!isOnCancellation)
                    {
                        PixelData[,] subbaza = getSubsegment(i, j, segmentyBazy);

                        int[, ,] najmniejszyBlad = { { { 256, 256, 256 }, { 256, 256, 256 }
}, { { 256, 256, 256 }, { 256, 256, 256 } } };

                        int[, ,] obecnyBlad;
                        foreach (SegmentowanyObraz kafelek in listaKafelkow)
                        {
                            PixelData[,] segmentyKafla = kafelek.TablicaObszarow;
                            obecnyBlad = LiczBlad(subbaza, segmentyKafla);
                            int roznica = LiczRozniceBledu(najmniejszyBlad, obecnyBlad);
                            if (roznica > 0)
                            {
                                najmniejszyBlad = obecnyBlad;
                                plikDopasowania = kafelek.FileName;
                            }
                            //Console.Write("|");
                        }
                        mapaPlikow[i / ROZMIAR, j / ROZMIAR] = plikDopasowania;
                        //Console.WriteLine("segment:" + i + "," + j);
                        OnStateChanged(i * (segmentyBazy.GetLength(0)) + j,
(segmentyBazy.GetLength(0)) * (segmentyBazy.GetLength(1)));
                    }
                }
            }
        }

        private int LiczRozniceBledu(int[, ,] najmniejszyBlad, int[, ,] obecnyBlad)
        {
            int suma = 0;
            for (int i = 0; i < najmniejszyBlad.GetLength(0); i++)
            {
                for (int j = 0; j < najmniejszyBlad.GetLength(1); j++)
                {
                    int sumaLokalna = 0;
                    for (int k = 0; k < najmniejszyBlad.GetLength(2); k++)
                    {
                        int roznicaBledu = najmniejszyBlad[i, j, k] - obecnyBlad[i, j, k];
                        if (roznicaBledu>0)
                        {
                            sumaLokalna++;
                        }
                        else if (roznicaBledu<0)
                        {
                            sumaLokalna--;
                        }
                    }
                    if (sumaLokalna > 2)
                        suma++;
```

```
                    else if (sumaLokalna < 0)
                        suma--;
                }
            }
            return suma;
        }

        private int[, ,] LiczBlad(PixelData[,] subbaza, PixelData[,] segmentyKafla)
        {
            int[, ,] ret = new int[subbaza.GetLength(0), subbaza.GetLength(1), 3];
            for (int i = 0; i < ret.GetLength(0); i++)
            {
                for (int j = 0; j < ret.GetLength(1); j++)
                {
                    ret[i, j, 0] = Math.Abs(subbaza[i, j].R - segmentyKafla[i, j].R);
                    ret[i, j, 1] = Math.Abs(subbaza[i, j].G - segmentyKafla[i, j].G);
                    ret[i, j, 2] = Math.Abs(subbaza[i, j].B - segmentyKafla[i, j].B);
                }
            }
            return ret;
        }



        private PixelData[,] getSubsegment(int x, int y, PixelData[,] baza)
        {

            PixelData[,] ret = new PixelData[ROZMIAR, ROZMIAR];
            for (int i = 0; i < ROZMIAR; i++)
            {
                for (int j = 0; j < ROZMIAR; j++)
                {
                    ret[i, j] = baza[x + i, y + j];
                }
            }
            return ret;
        }

        private double Dopasuj(PixelData[,] segmentyBazy, PixelData[,] segmentyKafla, float[,
,] wagi)
        {
            double srednia;
            double sredniaTmp = 0;
            int xMax = segmentyKafla.GetLength(0);
            int yMax = segmentyKafla.GetLength(1);
            bool[,] aResult = new bool[xMax, yMax];

            for (int x = 0; x < xMax; x++)
            {
                for (int y = 0; y < yMax; y++)
                {
                    float dopR = (float)(segmentyBazy[x, y].R - segmentyKafla[x, y].R) *
wagi[x, y, 0] / 255;
                    float dopG = (float)(segmentyBazy[x, y].G - segmentyKafla[x, y].G) *
wagi[x, y, 1] / 255;
                    float dopB = (float)(segmentyBazy[x, y].B - segmentyKafla[x, y].B) *
wagi[x, y, 2] / 255;
                    float dop = (Math.Abs(dopR) + Math.Abs(dopG) + Math.Abs(dopB));
                    //float dopNasycenia = Math.Abs(dopR + dopG + dopB) / 3;
                    sredniaTmp += dop;
                }
            }
            return sredniaTmp;
        }

        #region Events

        public delegate void StateChangedHandler(double currentState);
        public event StateChangedHandler StateChanged;
        public void OnStateChanged(int current, int length)
        {
            if (StateChanged != null)
            {
                double percent = (double)current / (double)length;
                Console.WriteLine(percent);
```

```
                StateChanged(percent * 100);
            }
        }

        public delegate void ImageGeneratedHandler();
        public event ImageGeneratedHandler ImageGenerated;
        public void OnImageGenerated()
        {
            if (StateChanged != null)
            {
                ImageGenerated();
            }
        }
        #endregion
    }
}
```

MsqReader

```
using System;
using System.IO;

namespace GraphicProcessor
{
    public class MsqReader
    {
        public static SegmentowanyObraz ReadAllSegments(string file)
        {
            if (String.IsNullOrEmpty(file))
            {
                throw new ArgumentNullException("Nie podano nazwy pliku.");
            }
            if (!File.Exists(file))
            {
                throw new FileNotFoundException("Plik nie istnieje", file);
            }
            FileStream fs = null;
            PixelData[,] obszary = null;
            try
            {
                fs = new FileStream(file, FileMode.Open, FileAccess.Read);
                fs.Position = 0;
                byte[] buffer = new byte[4];
                fs.Read(buffer, 0, 4);
                int szer = BitConverter.ToInt32(buffer, 0);
                fs.Read(buffer, 0, 4);
                int wys = BitConverter.ToInt32(buffer, 0);
                obszary = new PixelData[szer, wys];
                for (int i = 0; i < szer; i++)
                {
                    for (int j = 0; j < wys; j++)
                    {
                        int b = fs.ReadByte();
                        int g = fs.ReadByte();
                        int r = fs.ReadByte();
                        PixelData pD = new PixelData();
                        pD.R = Convert.ToByte(r);
                        pD.G = Convert.ToByte(g);
                        pD.B = Convert.ToByte(b);
                        obszary[i, j] = pD;
                    }
                }
            }
            catch (Exception)
            {
                throw new Exception("Could not read file: " + file);
            }
            finally
            {
                if (fs != null)
                {
                    fs.Close();
                    fs.Dispose();
                }
            }
```

```
            SegmentowanyObraz sO = new SegmentowanyObraz(obszary);
            sO.MsqFileName = file;
            return sO;
        }
        public static SegmentowanyObraz ReadSmallSegments(string file)
        {
            if (String.IsNullOrEmpty(file))
            {
                throw new ArgumentNullException("Nie podano nazwy pliku.");
            }
            if (!File.Exists(file))
            {
                throw new FileNotFoundException("Plik nie istnieje", file);
            }
            FileStream fs = null;
            PixelData[,] obszary = null;
            try
            {
                fs = new FileStream(file, FileMode.Open, FileAccess.Read);
                byte[] buffer = new byte[4];
                fs.Read(buffer, 0,4);
                int szer = BitConverter.ToInt32(buffer, 0);
                fs.Read(buffer, 0, 4);
                int wys = BitConverter.ToInt32(buffer, 0);
                fs.Position += szer * wys * 3;
                fs.Read(buffer, 0, 4);
                szer = BitConverter.ToInt32(buffer, 0);
                fs.Read(buffer, 0, 4);
                wys = BitConverter.ToInt32(buffer, 0);
                obszary = new PixelData[szer, wys];
                for (int i = 0; i < szer; i++)
                {
                    for (int j = 0; j < wys; j++)
                    {
                        int b = fs.ReadByte();
                        int g = fs.ReadByte();
                        int r = fs.ReadByte();
                        PixelData pD = new PixelData();
                        pD.R = Convert.ToByte(r);
                        pD.G = Convert.ToByte(g);
                        pD.B = Convert.ToByte(b);
                        obszary[i, j] = pD;
                    }
                }
            }
            catch (Exception)
            {
                throw new Exception("Could not read file: " + file);
            }
            finally
            {
                if (fs != null)
                {
                    fs.Close();
                    fs.Dispose();
                }
            }
            SegmentowanyObraz sO = new SegmentowanyObraz(obszary);
            sO.MsqFileName = file;
            return sO;
        }
    }
}
```

MsqWriter

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;

namespace GraphicProcessor
{
    public static class MsqWriter
    {
```

```
        public static void WriteSegments(PixelData[,] data, string destPath)
        {
            if (data == null || String.IsNullOrEmpty(destPath))
            {
                throw new ArgumentNullException("Nie podano argumentu.");
            }
            FileStream fs = null;
            if (!File.Exists(destPath))
            {
                fs = new FileStream(destPath, FileMode.CreateNew, FileAccess.Write);
                BufferedStream bs = new BufferedStream(fs);
                byte[] byteSzer = BitConverter.GetBytes(data.GetUpperBound(0) + 1);
                byte[] byteWys = BitConverter.GetBytes(data.GetUpperBound(1) + 1);
                bs.Write(byteSzer, 0, 4);
                bs.Write(byteWys, 0, 4);
                foreach (PixelData pD in data)
                {
                    bs.WriteByte(pD.R);
                    bs.WriteByte(pD.G);
                    bs.WriteByte(pD.B);
                }
                bs.Close();
                bs.Dispose();
                fs.Dispose();
            }
        }

        public static void WriteSegments(PixelData[,] data, PixelData[,] smallData, string
destPath)
        {
            if (data == null || String.IsNullOrEmpty(destPath))
            {
                throw new ArgumentNullException("Nie podano argumentu.");
            }
            FileStream fs = null;
            if (!File.Exists(destPath))
            {
                fs = new FileStream(destPath, FileMode.CreateNew, FileAccess.Write);
                BufferedStream bs = new BufferedStream(fs);
                byte[] byteSzer = BitConverter.GetBytes(data.GetUpperBound(0)+1);
                byte[] byteWys = BitConverter.GetBytes(data.GetUpperBound(1)+1);
                bs.Write(byteSzer, 0, 4);
                bs.Write(byteWys, 0, 4);
                foreach (PixelData pD in data)
                {
                    bs.WriteByte(pD.R);
                    bs.WriteByte(pD.G);
                    bs.WriteByte(pD.B);
                }
                byteSzer = BitConverter.GetBytes(smallData.GetUpperBound(0)+1);
                byteWys = BitConverter.GetBytes(smallData.GetUpperBound(0)+1);
                bs.Write(byteSzer, 0, 4);
                bs.Write(byteWys, 0, 4);
                foreach (PixelData pD in smallData)
                {
                    bs.WriteByte(pD.R);
                    bs.WriteByte(pD.G);
                    bs.WriteByte(pD.B);
                }
                bs.Close();
                bs.Dispose();
                fs.Dispose();
            }
        }
    }
}
```

Obszar

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Drawing.Imaging;
```

```
namespace GraphicProcessor
{
    internal static class Obszar
    {
        internal static PixelData ZnajdzKolorObszaru(BitmapData bmp, byte rozmycie)
        {
            PixelValueDictionary slownikZbiorow = new PixelValueDictionary(rozmycie);

            unsafe
            {
                byte* wskObszaru = (byte*)bmp.Scan0;
                int wskPrzesuniecia = 0;
                int offset = bmp.Stride - bmp.Width * 3;
                for (int y = 0; y < bmp.Height; y++)
                {
                    for (int x = 0; x < bmp.Width; x++)
                    {
                        byte b = wskObszaru[wskPrzesuniecia++];
                        byte g = wskObszaru[wskPrzesuniecia++];
                        byte r = wskObszaru[wskPrzesuniecia++];
                        //wskPrzesuniecia++;
                        slownikZbiorow.DodajPixel(r, g, b);
                    }
                    wskPrzesuniecia += offset;
                }

            }
            return slownikZbiorow.SredniaBarwaObszaru;
        }
        internal static PixelData ZnajdzKolorObszaru(Bitmap bmp, byte rozmycie)
        {
            PixelValueDictionary slownikZbiorow = new PixelValueDictionary(rozmycie);

            unsafe
            {
                for (int y = 0; y < bmp.Height; y++)
                {
                    for (int x = 0; x < bmp.Width; x++)
                    {
                        Color c = bmp.GetPixel(x, y);
                        byte b = c.R;
                        byte g = c.G;
                        byte r = c.B;
                        //wskPrzesuniecia++;
                        slownikZbiorow.DodajPixel(r, g, b);
                    }
                    //wskPrzesuniecia += offset;
                }
            }
            return slownikZbiorow.SredniaBarwaObszaru;
        }
    }
}
```

PixelData

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Drawing.Imaging;

namespace GraphicProcessor
{
    internal static class Obszar
    {
        internal static PixelData ZnajdzKolorObszaru(BitmapData bmp, byte rozmycie)
        {
            PixelValueDictionary slownikZbiorow = new PixelValueDictionary(rozmycie);

            unsafe
            {
                byte* wskObszaru = (byte*)bmp.Scan0;
```

```
                int wskPrzesuniecia = 0;
                int offset = bmp.Stride - bmp.Width * 3;
                for (int y = 0; y < bmp.Height; y++)
                {
                    for (int x = 0; x < bmp.Width; x++)
                    {
                        byte b = wskObszaru[wskPrzesuniecia++];
                        byte g = wskObszaru[wskPrzesuniecia++];
                        byte r = wskObszaru[wskPrzesuniecia++];
                        //wskPrzesuniecia++;
                        slownikZbiorow.DodajPixel(r, g, b);
                    }
                    wskPrzesuniecia += offset;
                }

            }
            return slownikZbiorow.SredniaBarwaObszaru;
        }
        internal static PixelData ZnajdzKolorObszaru(Bitmap bmp, byte rozmycie)
        {
            PixelValueDictionary slownikZbiorow = new PixelValueDictionary(rozmycie);

            unsafe
            {
                for (int y = 0; y < bmp.Height; y++)
                {
                    for (int x = 0; x < bmp.Width; x++)
                    {
                        Color c = bmp.GetPixel(x, y);
                        byte b = c.R;
                        byte g = c.G;
                        byte r = c.B;
                        //wskPrzesuniecia++;
                        slownikZbiorow.DodajPixel(r, g, b);
                    }
                    //wskPrzesuniecia += offset;
                }
            }
            return slownikZbiorow.SredniaBarwaObszaru;
        }
    }
}
```

PixelValueDictionary

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

namespace GraphicProcessor
{
    internal class PixelValueDictionary
    {
        private SortedDictionary<String, ZbiorRozmyty> slownik;
        private byte rozmycie;

        internal PixelValueDictionary(byte rozmycie)
        {
            this.slownik = new SortedDictionary<String, ZbiorRozmyty>();
            this.rozmycie = rozmycie;
        }

        internal void DodajPixel(byte r, byte g, byte b)
        {
            string key = String.Format(Resources.PixelHexFormat, r)
                + String.Format(Resources.PixelHexFormat, g)
                + String.Format(Resources.PixelHexFormat, b);

            ZbiorRozmyty z = null;
            if (slownik.TryGetValue(key, out z))
            {
                z.DodajPunkt(r, g, b);
            }
```

```
            else
            {
                z = new ZbiorRozmyty(r, g, b, rozmycie);
                slownik.Add(key, z);
            }

        }

        internal PixelData SredniaBarwaObszaru
        {
            get
            {
                SortedDictionary<string, ZbiorRozmyty>.ValueCollection slownikV =
slownik.Values;
                List<ZbiorRozmyty> lista = new List<ZbiorRozmyty>();
                foreach (ZbiorRozmyty zR in slownikV)
                {
                    lista.Add(zR);
                }

                lista.Sort();
                LinkedList<ZbiorRozmyty> lL = new LinkedList<ZbiorRozmyty>(lista);
                //pobieramy najwiekszy element.
                ZbiorRozmyty z = lL.First.Value;
                PixelData pD = new PixelData();
                pD.R = z.R;
                pD.G = z.G;
                pD.B = z.B;
                return pD;
            }
        }
    }
}
```

Resources

```
namespace GraphicProcessor
{
    internal static class Resources
    {
        internal const string PixelHexFormat = "{0,2:X}";
        internal const string ZlyRozmiarTablicy = "Podany indeks nie pasuje do rozmiaru
tablicy.";
        internal const int maxRozmiarObrazu = 200;
    }
}
```

SegmentowanyObraz

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Drawing.Imaging;

namespace GraphicProcessor
{
    public class SegmentowanyObraz
    {
        #region konstruktory
        public SegmentowanyObraz(PixelData[,] segmenty)
        {
            koloryObszarow = segmenty;
        }
        #endregion
        public PixelData WezKolorObszaru(int x, int y)
        {
            if (koloryObszarow == null)
            {
            }
            int szerTablicy = koloryObszarow.GetUpperBound(0);
            int wysTablicy = koloryObszarow.GetUpperBound(1);
            if (x < 0 || y < 0 || x > szerTablicy || y > wysTablicy)
```

```
            {
                throw new ArgumentOutOfRangeException(Resources.ZlyRozmiarTablicy);
            }
            return koloryObszarow[x, y];
        }

        public PixelData[,] TablicaObszarow
        {
            get
            {
                return (PixelData[,])koloryObszarow.Clone();
            }
        }

        public String FileName
        {
            get
            {
                return fileName;
            }
            set
            {
                fileName = value;
                msqFileName = value + ".msq";
            }
        }

        public String MsqFileName
        {
            get
            {
                return msqFileName;
            }
            set
            {
                msqFileName = value;
                FileName = value.Replace(".msq", "");
                FileName = FileName.Replace(".MSQ", "");
            }
        }

        private PixelData[,] koloryObszarow;
        private String fileName;
        private String msqFileName;
    }
}
```

ZbiorRozmyty

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

namespace GraphicProcessor
{
    internal class ZbiorRozmyty : IComparable
    {
        internal ZbiorRozmyty(byte R, byte G, byte B, byte rozmycie)
        {
            wierzcholek = new PixelData();
            wierzcholek.R = R;
            wierzcholek.G = G;
            wierzcholek.B = B;
            if (rozmycie > RozmycieMax)
            {
                throw new ArgumentException("Za duza wartosc przedzialu rozmycia.(>128)",
"rozmycie");
            }
            else
            {
                this.rozmycie = rozmycie;
            }

            minR = wierzcholek.R - rozmycie;
            maxR = wierzcholek.R + rozmycie;
```

```
            minG = wierzcholek.G - rozmycie;
            maxG = wierzcholek.G + rozmycie;
            minB = wierzcholek.B - rozmycie;
            maxB = wierzcholek.B + rozmycie;

            wartoscR = 1;
            wartoscG = 1;
            wartoscB = 1;
        }

        internal void DodajPunkt(byte R, byte G, byte B)
        {
            int minR = wierzcholek.R - rozmycie;
            int maxR = wierzcholek.R + rozmycie;
            int minG = wierzcholek.G - rozmycie;
            int maxG = wierzcholek.G + rozmycie;
            int minB = wierzcholek.B - rozmycie;
            int maxB = wierzcholek.B + rozmycie;

            if (R < minR || R > maxR)
            {
                wartoscR += 0;
            }
            else
            {
                if (R <= this.R)
                {
                    wartoscR += (R - minR) / (this.R - minR);
                }
                else
                {
                    wartoscR += (R - maxR) / (this.R - maxR);
                }
            }
            if (B < minB || B > maxB)
            {
                wartoscB += 0;
            }
            else
            {
                if (B <= this.B)
                {
                    wartoscB += (B - minB) / (this.B - minB);
                }
                else
                {
                    wartoscB += (B - maxB) / (this.B - maxB);
                }
            }
            if (G < minG || G > maxG)
            {
                wartoscG += 0;
            }
            else
            {
                if (G <= this.G)
                {
                    wartoscG += (G - minG) / (this.G - minG);
                }
                else
                {
                    wartoscG += (G - maxR) / (this.G - maxG);
                }
            }
        }

        #region Wlaciwosci

        public byte R
        {
            get { return wierzcholek.R; }
        }

        public byte G
        {
```

```
            get { return wierzcholek.G; }
        }

        public byte B
        {
            get { return wierzcholek.B; }
        }

        public double WartoscR
        {
            get { return wartoscR; }
        }

        public double WartoscG
        {
            get { return wartoscG; }
        }

        public double WartoscB
        {
            get { return wartoscB; }
        }

        public double Srednia
        {
            get
            {
                srednia = (wartoscR + wartoscG + wartoscB) / 3;
                return srednia;
            }
        }
        #endregion

        #region Pola
        private PixelData wierzcholek;

        private int minR;
        private int maxR;
        private int minG;
        private int maxG;
        private int minB;
        private int maxB;

        private double wartoscR;
        private double wartoscG;
        private double wartoscB;
        private double srednia;

        private byte rozmycie;
        #endregion

        #region Stałe
        private const byte RozmycieMax = Byte.MaxValue / 2;
        private const string compareExceptionText = "Obiekt nie jest klasy ZbiorRozmyty.";
        #endregion

        #region IComparable Members

        public int CompareTo(object obj)
        {
            ZbiorRozmyty z = obj as ZbiorRozmyty;
            if (z == null)
            {
                throw new ArgumentException(compareExceptionText);
            }
            else
            {
                double roznica = this.Srednia - z.Srednia;
                int ret = 0;
                if (roznica <= Double.Epsilon || roznica >= -Double.Epsilon)
                {
                    ret = 0;
                }
                else if (roznica < -Double.Epsilon)
                {
```

```
                    ret = -1;
                }
                else
                {
                    ret = 1;
                }
                return ret;
            }
        }

        #endregion
    }
}
```

FolderIndexingUserContrtol

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Drawing.Imaging;
using System.Data;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.IO;
using GraphicProcessor;

namespace GPTest
{
    public partial class FolderIndexingUserControl : UserControl
    {
        public FolderIndexingUserControl()
        {
            progress = 0;
            InitializeComponent();
            Control.CheckForIllegalCrossThreadCalls = false;
            cancel = false;
        }

        public void SetDestinationPath(string path)
        {
            collectionLocation = path;
        }

        private void textBox1_TextChanged(object sender, EventArgs e)
        {
            if (!string.IsNullOrEmpty(textBox2.Text))
            {
                indexButton.Enabled = true;
            }
        }

        private void button1_Click(object sender, EventArgs e)
        {

            if (folderBrowserDialog1.ShowDialog() == DialogResult.OK)
            {
                textBox1.Text = folderBrowserDialog1.SelectedPath;
            }

        }

        private void textBox2_TextChanged(object sender, EventArgs e)
        {
            if (string.IsNullOrEmpty(textBox2.Text))
            {
                indexButton.Enabled = false;
            }
            else
                if (!string.IsNullOrEmpty(textBox1.Text))
                {
                    indexButton.Enabled = true;
                }
        }
```

```
        private void button2_Click(object sender, EventArgs e)
        {
            if (this.backgroundWorkerIndexing.IsBusy)
            {
                cancel = true;
                //this.backgroundWorkerIndexing.CancelAsync();
            }
            Form parentAsForm = Parent as Form;
            parentAsForm.Close();
        }

        private void indexButton_Click(object sender, EventArgs e)
        {
            cancel = false;
            char separator = Path.DirectorySeparatorChar;
            string finalPath = collectionLocation + separator + textBox2.Text;
            if (Directory.Exists(finalPath))
            {
                if (MessageBox.Show("Wybrany folder istnieje. Wszystkie pliki zostaną z niego
usunięte.",
                    "Folder docelowy istnieje", MessageBoxButtons.OKCancel) ==
DialogResult.Cancel)
                {
                    return;
                }
            }
            this.backgroundWorkerIndexing.RunWorkerAsync();
        }

        #region Pola

        private string collectionLocation;
        private float progress;
        private bool cancel;
        #endregion

        const int ROZMIAR = 2;

        private void backgroundWorkerIndexing_DoWork(object sender, DoWorkEventArgs e)
        {
            progress = 0;
            char separator = Path.DirectorySeparatorChar;
            textBox2.Enabled = false;
            textBox1.Enabled = false;
            button1.Enabled = false;
            numericUpDown1.Enabled = false;
            this.Cursor = Cursors.WaitCursor;
            foreach (Control c in this.Controls)
            {
                c.Cursor = Cursors.WaitCursor;
            }
            button2.Cursor = Cursors.Default;
            indexButton.Enabled = false;
            groupBox4.Text = "kopiowanie...";
            CopyToPath(textBox1.Text, collectionLocation + separator + textBox2.Text);
            string destPath = collectionLocation + separator + textBox2.Text;
            IndexDirectory iD = new IndexDirectory(destPath);
            progressBar1.Value = 0;
            progressBar1.Maximum = iD.IlPlików;
            progressBar1.Step = 1;
            iD.FileIdexed += new IndexDirectory.IndexHandler(CommitProgress);
            iD.IndexingStarted += new IndexDirectory.IndxingStartedHandler(ShowPreview);
            iD.Indeksuj((int)numericUpDown2.Value, (int)numericUpDown2.Value, ROZMIAR,
ROZMIAR, (byte)numericUpDown1.Value, ref cancel);
        }

        private void CopyToPath(string SrcPath, string DestPath)
        {
            char separator = Path.DirectorySeparatorChar;
            Directory.CreateDirectory(DestPath);
            foreach (string subDir in Directory.GetDirectories(SrcPath))
            {
                string dest = DestPath + separator +
subDir.Substring(subDir.LastIndexOf(separator));
```

```
                CopyToPath(subDir, dest);
            }
            foreach (string f in Directory.GetFiles(SrcPath))
            {
                string path = DestPath + separator + f.Substring(f.LastIndexOf(separator));
                bool overwrite = true;
                if (isGraphicalFile(f))
                {
                    File.Copy(f, path, overwrite);
                }
            }
        }

        private Boolean isGraphicalFile(String file)
        {
            bool ret = false;
            FileInfo fileInfo = new FileInfo(file);
            if (System.Text.RegularExpressions.Regex.IsMatch(fileInfo.Extension,
@"\.(jpg|jpeg|gif|bmp|png)",
                System.Text.RegularExpressions.RegexOptions.IgnoreCase))
            {
                ret = true;
            }
            return ret;
        }

        private void CommitProgress(float percent)
        {
            progress += percent;
            this.backgroundWorkerIndexing.ReportProgress(Convert.ToInt16(progress));
        }

        private void ShowPreview(string file)
        {
            this.groupBox4.Text = ".." + file.Substring(collectionLocation.Length);
            try
            {
                this.pictureBox1.Image = Image.FromFile(file);
            }
            catch (OutOfMemoryException)
            {
                this.pictureBox1.Image = null;
            }
        }

        private void backgroundWorkerIndexing_ProgressChanged(object sender,
ProgressChangedEventArgs e)
        {
            this.progressBar1.PerformStep();
        }

        private void backgroundWorkerIndexing_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)
        {

            this.progressBar1.Value = this.progressBar1.Maximum;
            this.Cursor = Cursors.Arrow;
            foreach (Control c in this.Controls)
            {
                c.Cursor = Cursors.Default;
            }
            MessageBox.Show("Zakończono indeksowanie plików.", "Indeksowanie plików",
MessageBoxButtons.OK, MessageBoxIcon.Asterisk, MessageBoxDefaultButton.Button1);
            pictureBox1.Image = null;
            textBox1.Text = null;
            textBox2.Text = null;
            textBox2.Enabled = true;
            this.groupBox4.Text = "";
            this.progressBar1.Value = 0;
            button1.Enabled = true;
            numericUpDown1.Enabled = true;
            ;
        }
    }
```

```
}
```

AboutBox1

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Linq;
using System.Reflection;
using System.Windows.Forms;

namespace GPTest
{
    internal partial class AboutBox1 : Form
    {
        private static AboutBox1 instance = new AboutBox1();

        internal static AboutBox1 GetInstance()
        {
            return instance;
        }

        private AboutBox1()
        {
            InitializeComponent();
            this.Text = String.Format("About {0} {0}", AssemblyTitle);
            this.labelProductName.Text = AssemblyProduct;
            this.labelVersion.Text = String.Format("Version {0} {0}", AssemblyVersion);
            this.labelCopyright.Text = AssemblyCopyright;
            this.labelCompanyName.Text = AssemblyCompany;
            this.textBoxDescription.Text = AssemblyDescription;
        }

        #region Assembly Attribute Accessors

        public string AssemblyTitle
        {
            get
            {
                object[] attributes =
Assembly.GetExecutingAssembly().GetCustomAttributes(typeof(AssemblyTitleAttribute), false);
                if (attributes.Length > 0)
                {
                    AssemblyTitleAttribute titleAttribute =
(AssemblyTitleAttribute)attributes[0];
                    if (titleAttribute.Title != "")
                    {
                        return titleAttribute.Title;
                    }
                }
                return
System.IO.Path.GetFileNameWithoutExtension(Assembly.GetExecutingAssembly().CodeBase);
            }
        }

        public string AssemblyVersion
        {
            get
            {
                return Assembly.GetExecutingAssembly().GetName().Version.ToString();
            }
        }

        public string AssemblyDescription
        {
            get
            {
                object[] attributes =
Assembly.GetExecutingAssembly().GetCustomAttributes(typeof(AssemblyDescriptionAttribute),
false);
                if (attributes.Length == 0)
                {
```

```
                    return "";
                }
                return ((AssemblyDescriptionAttribute)attributes[0]).Description;
            }
        }

        public string AssemblyProduct
        {
            get
            {
                object[] attributes =
Assembly.GetExecutingAssembly().GetCustomAttributes(typeof(AssemblyProductAttribute), false);
                if (attributes.Length == 0)
                {
                    return "";
                }
                return ((AssemblyProductAttribute)attributes[0]).Product;
            }
        }

        public string AssemblyCopyright
        {
            get
            {
                object[] attributes =
Assembly.GetExecutingAssembly().GetCustomAttributes(typeof(AssemblyCopyrightAttribute),
false);
                if (attributes.Length == 0)
                {
                    return "";
                }
                return ((AssemblyCopyrightAttribute)attributes[0]).Copyright;
            }
        }

        public string AssemblyCompany
        {
            get
            {
                object[] attributes =
Assembly.GetExecutingAssembly().GetCustomAttributes(typeof(AssemblyCompanyAttribute), false);
                if (attributes.Length == 0)
                {
                    return "";
                }
                return ((AssemblyCompanyAttribute)attributes[0]).Company;
            }
        }
        #endregion
    }
}
```

DirectoryIndexing

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace GPTest
{
    public partial class DirectoryIndexing : Form
    {
        public static DirectoryIndexing getInstance()
        {
            if (instance == null)
            {
                instance = new DirectoryIndexing();
```

```
            }
            return instance;
        }

        private static DirectoryIndexing instance;

        private DirectoryIndexing()
        {
            InitializeComponent();

this.folderIndexingUserControl1.SetDestinationPath(Properties.Settings.Default.CollectionLocation);
        }
    }
}
```

OknoGenerowania2

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using GraphicProcessor;
using System.Threading;

namespace GPTest.Windows_Forms
{
    public partial class OknoGenerowania2 : Form
    {
        public OknoGenerowania2()
        {
            InitializeComponent();
            collectionLocation = Properties.Settings.Default.CollectionLocation;

        }

        #region Pola

        /// <summary>
        /// lokalizacja kolekcji obrazów
        /// </summary>
        private string collectionLocation;
        /// <summary>
        /// obiekt budujący mozaiki
        /// </summary>
        private readonly MosaiqueBuilder builder = MosaiqueBuilder.Instance;
        /// <summary>
        /// obraz - bitmapa
        /// </summary>
        private Image bazaObraz;
        /// <summary>
        /// obrazek bazowy
        /// </summary>
        private String baza;
        /// <summary>
        /// posegmentowany obraz bazowy
        /// </summary>
        SegmentowanyObraz bazaSegmenty;
        /// <summary>
        /// przechowuje proporcje obrazu (wykorzystywane do obliczenia wielkosci kafelka)
        /// </summary>
        private double imageAspect = 1;
        /// <summary>
        /// przechowuje obecna wartosc il*UpDown
        /// </summary>
        private decimal pozUpDown = 100;
        private decimal pionUpDown = 100;
        /// <summary>
        /// przechowuje obecna wartosc wielk*UpDown
        /// </summary>
        private decimal wielkPoz = 10;
```

```
        private decimal wielkPion = 10;
        object checkBaseSender = null;
        private Image wygenerowanyObraz;
        #endregion

        #region Properties

        public String Baza
        {
            get
            {
                return baza;
            }
            set
            {
                baza = value;
            }
        }

        public Image BazaImage
        {
            get
            {
                return bazaObraz;
            }
            set
            {
                bazaObraz = value;
            }
        }

        public int MaxIlSegmPion
        {
            get
            {
                return (int)ilPionUpDown.Value;
            }
            set
            {
                pionUpDown = value;
                ilPionUpDown.Maximum = value;
                ilPionUpDown.Value = ilPionUpDown.Maximum;
            }
        }

        public int MaxIlSegmPoz
        {
            get
            {
                return (int)ilPozUpDown.Value;
            }
            set
            {
                pozUpDown = value;
                ilPozUpDown.Maximum = value;
                ilPozUpDown.Value = ilPozUpDown.Maximum;
            }
        }

        public int MaxWielkSegmPion
        {
            get
            {
                return (int)wielkPionUpDown.Value;
            }
            set
            {
                wielkPionUpDown.Maximum = value;
                wielkPionUpDown.Value = wielkPionUpDown.Maximum;
            }
        }

        public int MaxWielkSegmPoz
        {
            get
```

```
            {
                return (int)wielkPozUpDown.Value;
            }
            set
            {
                wielkPozUpDown.Maximum = value;
                wielkPozUpDown.Value = wielkPozUpDown.Maximum;
            }
        }

        public double ImageAspect
        {
            get
            {
                return imageAspect;
            }
            set
            {
                if (value <= 0)
                {
                    throw new ArgumentOutOfRangeException("Aspect obrazu musi byc wiekszy od
0.");
                }
                imageAspect = value;
                if (imageAspect > 1)
                {
                    wielkPionUpDown.Value = wielkPionUpDown.Minimum;
                    wielkPozUpDown.Value = (decimal)Math.Round((double)wielkPionUpDown.Value
* imageAspect);
                }
                else
                {
                    wielkPozUpDown.Value = wielkPozUpDown.Minimum;
                    wielkPionUpDown.Value = (decimal)Math.Round((double)wielkPozUpDown.Value
/ imageAspect);
                }
            }
        }

        #endregion

        /// <summary>
        /// dodaje element do drzewa kafelków
        /// </summary>
        /// <param name="path">badana ścieżka</param>
        /// <param name="parentNode">element, do którego dodawane są elementy</param>
        private void AddNode(string path, TreeNode parentNode)
        {
            TreeNode newNode;
            if (parentNode != null)
            {
                System.IO.DirectoryInfo current = new System.IO.DirectoryInfo(path);
                newNode = new TreeNode(current.Name);
                parentNode.Nodes.Add(newNode);
            }
            else
            {
                newNode = treeView1.Nodes[0];
            }
            foreach (string dir in System.IO.Directory.GetDirectories(path))
            {
                AddNode(dir, newNode);
            }
        }

        private void CollectFiles(object collectFilesArguments)
        {
            if (collectFilesArguments == null)
            {
                throw new ArgumentNullException("Argument nie może byc null");
            }

            CollectFilesAgruments args = collectFilesArguments as CollectFilesAgruments;
            if (args == null)
            {
```

```
                throw new ArgumentException("Wymagany obiekt typu CollectFilesAgruments");
            }
            String directory = args.Collection;
            ICollection<SegmentowanyObraz> kolekcja = args.Kolekcja;
            TreeNode node = args.Node;

            foreach (string file in System.IO.Directory.GetFiles(directory))
            {
                if (file.EndsWith(".msq") || file.EndsWith(".MSQ"))
                {
                    try
                    {
                        SegmentowanyObraz obraz = MsqReader.ReadSmallSegments(file);
                        kolekcja.Add(obraz);
                        //Console.WriteLine("Dodano plik: " + file);
                    }
                    catch (Exception e)
                    {
                        MessageBox.Show(this, e.Message, "Error", MessageBoxButtons.OK,
MessageBoxIcon.Error);
                    }
                }
            }
            foreach (string dir in System.IO.Directory.GetDirectories(directory))
            {
                System.IO.DirectoryInfo dI = new System.IO.DirectoryInfo(dir);
                TreeNode currentNode = null;
                foreach (TreeNode n in node.Nodes)
                {
                    if (n.Text.Equals(dI.Name))
                    {
                        currentNode = n;
                        break;
                    }
                }
                if (currentNode != null && currentNode.Checked)
                {
                    args.Collection = dir;
                    args.Node = currentNode;
                    CollectFiles(args);
                }
            }
        }

        #region Form Events
        private void OknoGenerowania2_Shown(object sender, EventArgs e)
        {
            if (string.IsNullOrEmpty(collectionLocation))
            {
                MessageBox.Show(this, "Kolekcja kafelków nie została jeszcze utworzona.",
                    "Brak kolekcji kafelków", MessageBoxButtons.OK, MessageBoxIcon.Error);
                this.Close();
            }
            else if (!System.IO.Directory.Exists(collectionLocation))
            {
                MessageBox.Show(this, "Kolekcja kafelków nie istnieje.",
                    "Brak kolekcji kafelków", MessageBoxButtons.OK, MessageBoxIcon.Error);
                this.Close();
            }
            else
            {
                treeView1.Nodes.Clear();
                System.IO.DirectoryInfo collection = new
System.IO.DirectoryInfo(collectionLocation);
                treeView1.Nodes.Add(new TreeNode(collection.Name));
                AddNode(collectionLocation, null);
            }
        }

        /// <summary>
        /// obsługa kliknięcia Zamknij
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void button2_Click(object sender, EventArgs e)
```

```
        {
            this.Close();
        }

        private void wielkPionUpDown_ValueChanged(object sender, EventArgs e)
        {
            decimal newVal = (decimal)Math.Round((double)wielkPionUpDown.Value *
imageAspect);
            if (newVal > wielkPozUpDown.Maximum)
            {
                wielkPozUpDown.Value = wielkPozUpDown.Maximum;
                wielkPionUpDown.Value = (decimal)Math.Round((double)wielkPozUpDown.Value /
imageAspect);
            }
            else if (newVal < wielkPozUpDown.Minimum)
            {
                wielkPozUpDown.Value = wielkPozUpDown.Minimum;
                wielkPionUpDown.Value = (decimal)Math.Round((double)wielkPozUpDown.Value /
imageAspect);
            }
            else
            {
                wielkPozUpDown.Value = newVal;
            }
            UpdateInfoLabel();
        }

        private void wielkPozUpDown_ValueChanged(object sender, EventArgs e)
        {
            decimal newVal = (decimal)Math.Round((double)wielkPozUpDown.Value / imageAspect);
            if (newVal > wielkPionUpDown.Maximum)
            {
                wielkPionUpDown.Value = wielkPionUpDown.Maximum;
                wielkPozUpDown.Value = (decimal)Math.Round((double)wielkPionUpDown.Value *
imageAspect);
            }
            else if (newVal < wielkPionUpDown.Minimum)
            {
                wielkPozUpDown.Value = wielkPozUpDown.Minimum;
                wielkPozUpDown.Value = (decimal)Math.Round((double)wielkPionUpDown.Value *
imageAspect);
            }
            else
            {
                wielkPionUpDown.Value = newVal;
            }
            UpdateInfoLabel();
        }

        private void ilPionUpDown_ValueChanged(object sender, EventArgs e)
        {
            ilUpDownZachowanie(ref pionUpDown, ilPionUpDown);
            UpdateInfoLabel();
        }

        private void ilPozUpDown_ValueChanged(object sender, EventArgs e)
        {
            ilUpDownZachowanie(ref pozUpDown, ilPozUpDown);
            UpdateInfoLabel();
        }

        private void ilUpDownZachowanie(ref decimal privateValue, NumericUpDown control)
        {
            if (privateValue > control.Value)
            {
                control.Value = Math.Round(privateValue / 2);
            }
            else if (privateValue < control.Value)
            {
                decimal newVal = Math.Round(privateValue * 2);
                if (newVal > control.Maximum)
                {
                    control.Value = control.Maximum;
                }
                else
```

```
                {
                    control.Value = newVal;
                }
            }
            privateValue = control.Value;
        }

        /// <summary>
        /// Zmienia informacje o rozmiarze pliku
        /// </summary>
        private void UpdateInfoLabel()
        {
            //uncompressed file size in kBytes
            Decimal fileSize = (ilPionUpDown.Value * ilPozUpDown.Value *
wielkPionUpDown.Value * wielkPozUpDown.Value) / 1000000;
            this.label8.Text = String.Format("rozmiar pliku: {0}x{1}, {2:#.00} MB",
ilPozUpDown.Value * wielkPozUpDown.Value,
                ilPionUpDown.Value * wielkPionUpDown.Value, fileSize);
        }

        /// <summary>
        /// obsługa kliknięcia Generuj
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void button1_Click(object sender, EventArgs e)
        {
            progressBar1.Visible = true;
            progressBar1.Style = ProgressBarStyle.Marquee;
            progressBar1.MarqueeAnimationSpeed = 100;
            BackgroundWorker worker = new BackgroundWorker();
            worker.DoWork += new DoWorkEventHandler(worker_DoWork);
            worker.RunWorkerCompleted += new
RunWorkerCompletedEventHandler(worker_RunWorkerCompleted);
            worker.RunWorkerAsync();
            //progressBar1.MarqueeAnimationSpeed = 0;
        }

        void worker_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
        {
            progressBar1.MarqueeAnimationSpeed = 0;
            progressBar1.Style = ProgressBarStyle.Blocks;
            OnImageCreated(wygenerowanyObraz);
        }

        void worker_DoWork(object sender, DoWorkEventArgs e)
        {
            infoLabel.Text = "indeksowanie segmentów";
            //indeksowanie i tworzenie kolekcji
            CollectFilesAgruments cFAgrs = new CollectFilesAgruments();
            cFAgrs.Collection = Properties.Settings.Default.CollectionLocation;
            cFAgrs.Kolekcja = new List<SegmentowanyObraz>();
            cFAgrs.Node = treeView1.Nodes[0];
            //DateTime start = DateTime.Now;
            Thread collectThread = new Thread(new
ParameterizedThreadStart(this.CollectFiles));
            collectThread.Start(cFAgrs);
            SegmentowanyObraz obraz = FileIndexer.Indeksuj(bazaObraz, (int)ilPozUpDown.Value
* 2, (int)ilPionUpDown.Value * 2, Convert.ToByte(64));
            builder.Baza = obraz;
            collectThread.Join();
            //TimeSpan czas = DateTime.Now.Subtract(start);
            //Console.WriteLine("Czas wykonania: " + czas.Minutes + ":" + czas.Seconds + "."
+ czas.Milliseconds);

            infoLabel.Text = "dopasowywanie obrazów";
            //tworzenie mozaiki
            builder.addSegments(cFAgrs.Kolekcja);
            builder.BuildMosaique2();

            infoLabel.Text = "generowanie obrazu";
            //generowanie obrazu
            Image newImage = generujObraz();
            //backgroundWorker2.ReportProgress(100);
            infoLabel.Text.Remove(0);
```

```
            wygenerowanyObraz = newImage;
        }

        private Bitmap ResizeBitmap(Bitmap b, int nWidth, int nHeight)
        {
            Bitmap result = new Bitmap(nWidth, nHeight);
            using (Graphics g = Graphics.FromImage((Image)result))
                g.DrawImage(b, 0, 0, nWidth, nHeight);
            return result;
        }

        private Image generujObraz()
        {
            int obecnyProgress = 0;
            String[,] mapa = builder.MapaPlikow;
            Image newImage = new Bitmap((int)(ilPozUpDown.Value * wielkPozUpDown.Value),
(int)(ilPionUpDown.Value * wielkPionUpDown.Value));
            int segmentSzer = (int)wielkPozUpDown.Value;
            int segmentWys = (int)wielkPionUpDown.Value;
            Graphics g = Graphics.FromImage(newImage);
            for (int i = 0; i < mapa.GetLength(0); i++)
            {
                for (int j = 0; j < mapa.GetLength(1); j++)
                {
                    if (mapa[i, j] != null)
                    {
                        String obecnyPlik = mapa[i, j];
                        Bitmap segm = new Bitmap(mapa[i, j]);
                        segm = ResizeBitmap(segm, segmentSzer, segmentWys);
                        for (int k = i; k < mapa.GetLength(0); k++)
                        {
                            for (int l = 0; l < mapa.GetLength(1); l++)
                            {
                                if (mapa[k, l] != null && mapa[k, l].Equals(obecnyPlik))
                                {
                                    g.DrawImage(segm, k * segmentSzer, l * segmentWys);
                                    mapa[k, l] = null;
                                }
                            }
                        }
                        segm.Dispose();
                        //g.DrawImage(segm,
                        //    new Rectangle(i * segmentSzer, j * segmentWys, segmentSzer,
segmentWys),
                        //    new Rectangle(0, 0, segm.Width, segm.Height),
GraphicsUnit.Pixel);
                        //double progress = ((double)i * (mapa.GetUpperBound(0) + 1) + j) /
((mapa.GetUpperBound(0) + 1) * (mapa.GetUpperBound(1) + 1)) * 100;
                        //int percProgress = (int)progress;
                        //if (percProgress > obecnyProgress)
                        //{
                        //    obecnyProgress = percProgress;
                        //    //backgroundWorker2.ReportProgress(percProgress);
                        //}
                        //Console.WriteLine("Wykonano:" + i + "," + j);
                    }
                }
            }
            g.Dispose();
            return newImage;
        }

        private void treeView1_AfterCheck(object sender, TreeViewEventArgs e)
        {
            if (e.Node.Checked == true)
            {
                if (checkBaseSender == null)
                {
                    checkBaseSender = sender;
                    if (e.Node.Parent != null)
                    {
                        e.Node.Parent.Checked = true;
                    }
                    foreach (TreeNode t in e.Node.Nodes)
                    {
```

```
                        t.Checked = true;
                    }
                    checkBaseSender = null;
                }
            }
            else
            {
                if (e.Node.Parent != null)
                {
                    TreeNode parent = e.Node.Parent;
                    bool hasCheckedNodes = false;
                    foreach (TreeNode t in parent.Nodes)
                    {
                        if (t.Checked)
                        {
                            hasCheckedNodes = true;
                            break;
                        }
                    }
                    if (!hasCheckedNodes)
                    {
                        parent.Checked = false;
                    }
                }
            }
        }
        #endregion

        #region Event ImageCreated
        public delegate void ImageCreatedHandler(Image img);
        public event ImageCreatedHandler ImageCreated;
        public void OnImageCreated(Image img)
        {
            if (ImageCreated != null)
            {
                ImageCreated(img);
            }
        }
        #endregion
    }

    internal class CollectFilesAgruments
    {
        internal String Collection
        {
            get;
            set;
        }
        internal TreeNode Node
        {
            get;
            set;
        }
        internal ICollection<SegmentowanyObraz> Kolekcja
        {
            get;
            set;
        }
    }
}
```

OknoObrazu2b

```
using System;
using System.ComponentModel;
using System.Drawing;
using System.Drawing.Imaging;
using System.Text;
using System.Windows.Forms;

namespace GraphicProcessor
{
    public partial class OknoObrazu2b : Form
    {
        #region Constructors
```

```
        private OknoObrazu2b()
        {
            //ustawienia początkowe zoomu
            zoomIndex = 13;
            zoomWart = zoom[zoomIndex];

            InitializeComponent();

            pWidok = new Point(0, 0);
            sWidok = new Size(panel1.Width, panel1.Height);
            viewBuffer = new Bitmap(panel1.Width, panel1.Height);

        }

        public OknoObrazu2b(string file, bool isGenerated)
            : this(file)
        {
            this.isGenerated = isGenerated;
        }


        public OknoObrazu2b(string file)
            : this()
        {
            if (string.IsNullOrEmpty(file))
            {
                throw new ArgumentNullException("file nie moze byc typu null lub łańcuchem
pustym.");
            }
            if (!System.IO.File.Exists(file))
            {
                throw new ArgumentException("plik: " + file + " nie istnieje.");
            }
            try
            {
                Bitmap bmp = new Bitmap(file);
                this.obraz = bmp;
            }
            catch (System.IO.FileNotFoundException)
            {
                throw new ArgumentException("Nie można utworzyc obrazu: " + file);
            }
        }

        public OknoObrazu2b(Bitmap bmp, bool isGenerated)
            : this(bmp)
        {
            this.isGenerated = isGenerated;
        }

        public OknoObrazu2b(Bitmap obraz)
            : this()
        {
            if (obraz == null)
            {
                throw new ArgumentNullException("nieprawidłowy argument");
            }
            this.obraz = obraz;
        }
        #endregion

        #region Pola
        //nazwa pliku
        private String plik;
        //wczytany obraz
        private Bitmap obraz;
        //wyswietlana czesc
        private Bitmap viewBuffer;

        //obecna wartosc zoom
        private double zoomWart;
        //indeks w tablicy zoom
        private int zoomIndex;
        private Point pWidok;
        private Size sWidok;
```

```
        private bool isGenerated;
        #endregion

        #region Stałe
        private static readonly double[] zoom =
            new double[18] { 0.01, 0.02, 0.03, 0.04, 0.05, 0.0625,
                0.0833, 0.125, 0.1667, 0.25, 0.3333, 0.5, 0.6667,
                1,4, 8, 16, 32 }
            //{ 0.01, 0.02, 0.03, 0.04, 0.05, 0.0625,
            //0.0833, 0.125, 0.1667, 0.25, 0.3333, 0.5, 0.6667,
            //1, 2, 3, 4, 5, 6, 7, 8, 12, 16, 32 }
                ;
        #endregion

        #region Properties
        /// <summary>
        /// gets file name
        /// </summary>
        public String NazwaPliku
        {
            set
            {
                plik = value;
            }
            get
            {
                return plik;
            }
        }
        /// <summary>
        /// gets true if image was generated by Mosaique
        /// </summary>
        public bool IsGenerated
        {
            get
            {
                return isGenerated;
            }
        }
        /// <summary>
        /// gets Window image
        /// </summary>
        public Bitmap Obraz
        {
            get
            {
                return obraz;
            }
        }
        #endregion

        #region EventHandlers
        private void panel1_Paint(object sender, PaintEventArgs e)
        {
            if (viewBuffer == null)
            {
                ustawViewBuffer();
            }
            Graphics g = e.Graphics;
            g.DrawImage(viewBuffer, new Point(0, 0));
            g.Dispose();
            base.OnPaint(e);
        }

        private void OknoObrazu2_Load(object sender, EventArgs e)
        {
            setZoomLabel();
            ustawScrolle();
            ustawViewBuffer();
            this.panel1.Invalidate();
        }

        private void vScrollBar1_ValueChanged(object sender, EventArgs e)
        {
            pWidok.Y = vScrollBar1.Value;
```

```
            ustawViewBuffer();
            this.panel1.Invalidate();
        }

        private void hScrollBar1_ValueChanged(object sender, EventArgs e)
        {
            ScrollEventArgs args = e as ScrollEventArgs;
            pWidok.X = hScrollBar1.Value;
            ustawViewBuffer();
            this.panel1.Invalidate();
        }

        private void panel1_SizeChanged(object sender, EventArgs e)
        {
            ustawScrolle();
            viewBuffer.Dispose();
            viewBuffer = new Bitmap(panel1.Width, panel1.Height);
            ustawViewBuffer();
            panel1.Invalidate();
        }

        private void panel1_Click(object sender, EventArgs e)
        {
            MouseEventArgs args = e as MouseEventArgs;
            int kierunek = 0;
            if (args.Button.Equals(MouseButtons.Left))
            {
                kierunek = 1;
            }
            else if (args.Button.Equals(MouseButtons.Right))
            {
                kierunek = -1;
            }
            else
            {
                return;
            }
            int nowyIndeks = zoomIndex + kierunek;
            //jezeli poza zakresem zoomow
            if (nowyIndeks < 0 || nowyIndeks >= zoom.Length)
            {
                return;
            }
            //poprawna zmiana zoomu
            else
            {
                zoomIndex = nowyIndeks;
                zoomWart = zoom[zoomIndex];

                ustawScrolle();
                setZoomLabel();

                ustawViewBuffer();
                this.panel1.Invalidate();
            }
        }
        #endregion

        private void ustawScrolle()
        {
            vScrollBar1.Maximum = obraz.Height - panel1.Height;
            hScrollBar1.Maximum = obraz.Width - panel1.Width;
        }

        private void setZoomLabel()
        {
            double scaleText = zoomWart * 100;
            this.toolStripStatusLabel1.Text = scaleText.ToString() + "%";
        }

        private void ustawViewBuffer()
        {
            Size pobierane = new Size((int)(viewBuffer.Width / zoomWart),
(int)(viewBuffer.Height / zoomWart));
            Console.WriteLine(pobierane.Width + "x" + pobierane.Height);
```

```
            if (pWidok.X + pobierane.Width > obraz.Width)
            {
                pWidok.X = obraz.Width - pobierane.Width;
                if (pWidok.X < 0)
                {
                    pWidok.X = 0;
                }
            }
            if (pWidok.Y + pobierane.Height > obraz.Height)
            {
                pWidok.Y = obraz.Height - pobierane.Height;
                if (pWidok.Y < 0)
                {
                    pWidok.Y = 0;
                }
            }

            if (zoomWart > 1)
            {
                int globalX, globalY;
                for (int x = 0; x < pobierane.Width; x++)
                {
                    for (int y = 0; y < pobierane.Height; y++)
                    {
                        Color current = obraz.GetPixel(pWidok.X + x, pWidok.Y + y);
                        for (int i = 0; i < zoomWart; i++)
                        {
                            for (int j = 0; j < zoomWart; j++)
                            {
                                viewBuffer.SetPixel((int)(x * zoomWart + i), (int)(y *
zoomWart + j), current);
                                globalX = (int)(x * zoomWart + i);
                                globalY = (int)(y * zoomWart + j);
                            }
                        }
                    }
                }
            }
            else
            {
                Graphics g = Graphics.FromImage(viewBuffer);
                g.Clear(Color.FromKnownColor(KnownColor.AppWorkspace));
                g.DrawImage(obraz, new Rectangle(0, 0, viewBuffer.Width, viewBuffer.Height),
0, 0, obraz.Width, obraz.Height, GraphicsUnit.Pixel);
                g.Dispose();
            }
        }
    }
}
```

PropertiesWindow

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace GPTest
{
    public partial class PropertiesWindow : Form
    {
        public PropertiesWindow()
        {
            InitializeComponent();
            textBox1.Text = Properties.Settings.Default.CollectionLocation;
            textBox2.Text = Properties.Settings.Default.OutputLocation;
        }

        private void button1_Click(object sender, EventArgs e)
        {
            folderBrowserDialog1.Description = "Katalog kolekcji";
```

```
            folderBrowserDialog1.SelectedPath = textBox1.Text;
            if (folderBrowserDialog1.ShowDialog() != DialogResult.Cancel)
            {
                textBox2.Text = folderBrowserDialog1.SelectedPath;
            }
        }

        private void button2_Click(object sender, EventArgs e)
        {
            folderBrowserDialog1.Description = "Katalog mozaik";
            folderBrowserDialog1.SelectedPath = textBox2.Text;
            if (folderBrowserDialog1.ShowDialog() != DialogResult.Cancel)
            {
                textBox2.Text = folderBrowserDialog1.SelectedPath;
            }
        }

        private void button3_Click(object sender, EventArgs e)
        {
            Properties.Settings.Default.CollectionLocation = textBox1.Text;
            Properties.Settings.Default.OutputLocation = textBox2.Text;
            Properties.Settings.Default.Save();
            this.Visible = false;
        }
    }
}
```

StudioWindow

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using GraphicProcessor;
using GPTest.Windows_Forms;

namespace GPTest
{
    public partial class StudioWindow : Form
    {
        public StudioWindow()
        {
            InitializeComponent();
            disableFileConnectedMenuItems();
            oknoGenerowania.ImageCreated += new
OknoGenerowania2.ImageCreatedHandler(oknoGenerowania_ImageCreated);
            oknoGenerowania.Owner = this;
            saveFileDialog1.InitialDirectory = Properties.Settings.Default.OutputLocation;
            openFileDialog1.InitialDirectory =
Properties.Settings.Default.CollectionLocation;
        }

        void oknoGenerowania_ImageCreated(Image img)
        {
            OknoObrazu2b okno = new OknoObrazu2b(img as Bitmap, true);
            okno.MdiParent = this;
            okno.Show();
            this.ActivateMdiChild(okno);
            okno.BringToFront();
            okno.Focus();
            this.oknoGenerowania.Close();
        }

        #region FileConnectedMenuItems
        private void enableFileConnectedMenuItems()
```

```
        {
            zapiszMenuItem.Enabled = true;
            zapiszJakoMenuItem.Enabled = true;
            zamknijMenuItem.Enabled = true;
            generujMenuItem1.Enabled = true;
        }

        private void disableFileConnectedMenuItems()
        {
            zapiszMenuItem.Enabled = false;
            zapiszJakoMenuItem.Enabled = false;
            zamknijMenuItem.Enabled = false;
            generujMenuItem1.Enabled = false;
        }
        #endregion

        #region Pola
        private OtwieranieObrazuProgressWindow progressWindow = new
OtwieranieObrazuProgressWindow();
        private readonly DirectoryIndexing oknoIndeksowania =
DirectoryIndexing.getInstance();
        private readonly OknoGenerowania2 oknoGenerowania = new OknoGenerowania2();
        private readonly AboutBox1 about = AboutBox1.GetInstance();

        private Boolean projectOpened = false;
        #endregion

        #region MenuItemClicks
        private void indeksujToolStripMenuItem_Click(object sender, EventArgs e)
        {
            DirectoryIndexing.getInstance().ShowDialog();
        }

        private void aboutMenuItem_Click(object sender, EventArgs e)
        {

            about.ShowDialog(this);
        }

        private void OtworzMenuItem_Click(object sender, EventArgs e)
        {
            if (openFileDialog1.ShowDialog() == DialogResult.OK)
            {
                Form okno = new OknoObrazu2b(openFileDialog1.FileName, false);
                okno.FormClosed += new FormClosedEventHandler(okno_FormClosed);
                okno.MdiParent = this;
                okno.Show();
                enableFileConnectedMenuItems();
            }
        }
        private void zamknijMenuItem_Click(object sender, EventArgs e)
        {
            Form active = this.ActiveMdiChild;
            active.Close();
            active.Dispose();
        }

        private void plikMenuItem1_Click(object sender, EventArgs e)
        {
            OknoObrazu2b active = this.ActiveMdiChild as OknoObrazu2b;
            if (active == null)
            {
                zapiszJakoMenuItem.Enabled = false;
                zapiszMenuItem.Enabled = false;
                generujMenuItem1.Enabled = false;
            }
            else if (active != null && active.IsGenerated)
            {
                zapiszJakoMenuItem.Enabled = true;
                zapiszMenuItem.Enabled = true;
                generujMenuItem1.Enabled = false;
            }
            else
            {
                zapiszJakoMenuItem.Enabled = false;
```

```
                zapiszMenuItem.Enabled = false;
                generujMenuItem1.Enabled = true;
            }
        }

        private void generujMenuItem1_Click(object sender, EventArgs e)
        {
            OknoObrazu2b okno = this.ActiveMdiChild as OknoObrazu2b;
            oknoGenerowania.BazaImage = okno.Obraz;
            oknoGenerowania.Baza = okno.NazwaPliku;
            oknoGenerowania.ImageAspect = (double)okno.Obraz.Width /
(double)okno.Obraz.Height;
            oknoGenerowania.MaxIlSegmPoz = okno.Obraz.Width / 2;
            oknoGenerowania.MaxIlSegmPion = okno.Obraz.Height / 2;
            oknoGenerowania.ShowDialog();
        }

        private void ustawieniaToolStripMenuItem_Click(object sender, EventArgs e)
        {
            PropertiesWindow properties = new PropertiesWindow();
            properties.StartPosition = FormStartPosition.CenterParent;
            properties.Show();
        }

        private void zakonczMenuItem_Click(object sender, EventArgs e)
        {
            this.Close();
        }

        private void zapiszMenuItem_Click(object sender, EventArgs e)
        {
            if (saveFileDialog1.ShowDialog(this) != DialogResult.Cancel)
            {
                OknoObrazu2b okno = this.ActiveMdiChild as OknoObrazu2b;
                if (okno != null && okno.IsGenerated)
                {
                    Image img = okno.Obraz;
                    string path = saveFileDialog1.FileName;
                    okno.NazwaPliku = path;
                    switch (saveFileDialog1.FilterIndex)
                    {
                        case 1:
                            img.Save(path, System.Drawing.Imaging.ImageFormat.Jpeg);
                            break;
                        case 2:
                            img.Save(path, System.Drawing.Imaging.ImageFormat.Png);
                            break;
                        case 3:
                            img.Save(path, System.Drawing.Imaging.ImageFormat.Bmp);
                            break;
                    }
                    okno.Text = path;
                }
            }
        }
        #endregion

        void okno_FormClosed(object sender, FormClosedEventArgs e)
        {
            Form senderForm = sender as Form;
            senderForm.Dispose();
            if (this.MdiChildren == null || this.MdiChildren.Length == 0)
            {
                disableFileConnectedMenuItems();
            }
        }

        #region OknoMenu
        private void uporzadkujOknaToolStripMenuItem_Click(object sender, EventArgs e)
        {
            this.LayoutMdi(MdiLayout.ArrangeIcons);
        }

        private void oknaKaskadowoToolStripMenuItem_Click(object sender, EventArgs e)
        {
```

```
            this.LayoutMdi(MdiLayout.Cascade);
        }

        private void sasiadujacoWPionieToolStripMenuItem_Click(object sender, EventArgs e)
        {
            this.LayoutMdi(MdiLayout.TileVertical);
        }

        private void sasiadujacoWPoziomieToolStripMenuItem_Click(object sender, EventArgs e)
        {
            this.LayoutMdi(MdiLayout.TileHorizontal);
        }
        #endregion


    }
}
```