FaceAppWithGPT2

FaceAppWithGPT2

Dependencies

- DlibDotNet v19.21.0.20220724
- Emgu.CV v4.9.0.5494
- Emgu.CV.Bitmap v4.9.0.5494
- Emgu.CV.runtime.windows v4.9.0.5494
- Xabe.FFmpeg v5.2.6

/FaceAppWithGPT2/Program.cs

```
using Emgu.CV;
using ImageProcessingLibrary.Helpers;
using ImageProcessingLibrary.PictureSizeAdaptation;

namespace FaceAppWithGPT2
{
    internal class Program
    {

            static void Main(string[] args)
            {
                if (args.Length < 3)
                {
                    Console.WriteLine("Usage: FaceAppWithGPT2 <inputDirectory> <outputDirect
                    Console.WriteLine("dimensionType: 'width' or 'height' (only required if
                    return;
                }

                string inputDirectory = args[0];
                string outputDirectory = args[1];
                string resizeOption = args[2];
                string dimensionType = args.Length > 3 ? args[3].ToLower() : string.Empty;

                try
                {
                    // Validate directories
                    DirectoryHelper.ValidateDirectory(inputDirectory);
                    if (!Directory.Exists(outputDirectory))
                    {
                        Directory.CreateDirectory(outputDirectory);
                    }

                    // Validate resize option at the beginning
            if (resizeOption.EndsWith("%"))
            {
                if (!int.TryParse(resizeOption.TrimEnd('%'), out int percentage) ||
                {
                    throw new ArgumentException("Invalid percentage value. It must b
                }
            }
            else if (int.TryParse(resizeOption, out int fixedSize))
            {
                if (fixedSize <= 0)
                {
                    throw new ArgumentException("Invalid size value. Width or height
                }
                if (string.IsNullOrEmpty(dimensionType) || (dimensionType != "width"
                {
                    throw new ArgumentException("Dimension type must be specified as
                }
            }
            else
            {
                throw new ArgumentException("Invalid resize option. Provide a percen
            }

            // Get image files from the input directory
            var imageFiles = DirectoryHelper.GetImageFiles(inputDirectory);

            // Instantiate the ImageResizer
            var imageResizer = new ImageResizer();

            // Resize each image and save it to the output directory
            foreach (var imagePath in imageFiles)
            {
                string outputPath = Path.Combine(outputDirectory, Path.GetFileName(i
                imageResizer.ResizeImage(imagePath, outputPath, resizeOption, dimens
                Console.WriteLine($"Resized image saved to: {outputPath}");
            }

            Console.WriteLine("Image resizing completed successfully.");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
    }
    }
}
```

ImageProcessingLibrary

Dependencies

- DlibDotNet v19.21.0.20220724
- Emgu.CV v4.9.0.5494
- Emgu.CV.Bitmap v4.9.0.5494
- Emgu.CV.runtime.windows v4.9.0.5494
- Xabe.FFmpeg v5.2.6

/ImageProcessingLibrary/Exceptions/ImageProcessingException.cs

```
using System;

namespace ImageProcessingLibrary.Exceptions
{
    public class ImageProcessingException : Exception
    {
        public ImageProcessingException(string message) : base(message) { }

        public ImageProcessingException(string message, Exception innerException) : base(mes
    }
}
```

/ImageProcessingLibrary/Helpers/DirectoryHelper.cs

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

namespace ImageProcessingLibrary.Helpers
{
    public static class DirectoryHelper
    {
        /// <summary>
        /// Validates if the given directory path exists. If it doesn't exist, throws a Dire
        /// </summary>
        /// <param name="directoryPath">The path of the directory to validate.</param>
        public static void ValidateDirectory(string directoryPath)
        {

            if (directoryPath == null)
                throw new ArgumentNullException(nameof(directoryPath), "Directory path canno
            if (string.IsNullOrWhiteSpace(directoryPath))
```

```
                throw new ArgumentException("Directory path cannot be empty.", nameof(direct

            if (!Directory.Exists(directoryPath))
                throw new DirectoryNotFoundException($"Directory '{directoryPath}' not founc
        }

        /// <summary>
        /// Gets all image files (JPG, PNG) from the specified directory.
        /// </summary>
        /// <param name="directoryPath">The path of the directory to search for image files
        /// <returns>A list of file paths for the images found in the directory.</returns>
        public static List<string> GetImageFiles(string directoryPath)
        {
            ValidateDirectory(directoryPath);

            // Define allowed image extensions
            string[] allowedExtensions = { ".jpg", ".jpeg", ".png" };

            // Get all files with allowed extensions
            var imageFiles = new List<string>();
            foreach (var file in Directory.GetFiles(directoryPath))
            {
                if (Array.Exists(allowedExtensions, ext => ext.Equals(Path.GetExtension(file
                {
                    imageFiles.Add(file);
                }
            }

            return imageFiles;
        }
    }
}
```

/ImageProcessingLibrary/Helpers/ImageHelper.cs

```
using Emgu.CV;
using Emgu.CV.CvEnum;
using Emgu.CV.Structure;
using DlibDotNet;
using DlibDotNet.Extensions;
using System.Drawing;
using System.Collections.Generic;
using System;
using ImageProcessingLibrary.Exceptions;
using ImageProcessingLibrary.Logging;
```

```
namespace ImageProcessingLibrary.Helpers
{
    public static class AlignmentHelper
    {
        /// <summary>
        /// Detects facial landmarks for a given image using a shape predictor.
        /// </summary>
        /// <param name="image">The input image as a Bitmap.</param>
        /// <param name="shapePredictor">The Dlib shape predictor model to use for detectio
        /// <returns>A list of detected facial landmarks as PointFs.</returns>
        public static List<PointF> DetectFacialLandmarks(Bitmap image, ShapePredictor shapeP
        {
            if (image == null)
                throw new ArgumentNullException(nameof(image), "Input image cannot be null."

            if (shapePredictor == null)
                throw new ArgumentNullException(nameof(shapePredictor), "Shape predictor can

            using (var dlibImage = image.ToArray2D<RgbPixel>())
            using (var detector = Dlib.GetFrontalFaceDetector())
            {
                var faces = detector.Operator(dlibImage);
                if (faces.Length == 0)
                    throw new ImageProcessingException("No face detected in the image.");

                var face = faces[0]; // Assuming only one face for simplicity
                var landmarks = shapePredictor.Detect(dlibImage, face);

                var points = new List<PointF>();
                for (uint i = 0; i < landmarks.Parts; i++)
                {
                    points.Add(new PointF((float)landmarks.GetPart(i).X, (float)landmarks.Ge
                }

                return points;
            }
        }

        /// <summary>
        /// Computes the affine transformation matrix required to align facial landmarks to
        /// </summary>
        /// <param name="sourcePoints">The current facial landmarks as a list of PointF.</pa
        /// <param name="destinationPoints">The desired facial points for alignment.</param>
        /// <returns>The affine transformation matrix as a Mat.</returns>
        public static Mat ComputeAffineTransform(List<PointF> sourcePoints, List<PointF> des
        {
            if (sourcePoints == null || sourcePoints.Count != 3)
                throw new ArgumentException("Source points must contain exactly 3 points.",

            if (destinationPoints == null || destinationPoints.Count != 3)
                throw new ArgumentException("Destination points must contain exactly 3 point

            return CvInvoke.GetAffineTransform(sourcePoints.ToArray(), destinationPoints.To
        }

        /// <summary>
        /// Applies an affine transformation to an image to align it based on a given trans
        /// </summary>
        /// <param name="image">The input image as a Mat.</param>
        /// <param name="transformationMatrix">The affine transformation matrix.</param>
        /// <param name="outputSize">The desired size of the output image.</param>
        /// <returns>The aligned image as a Mat.</returns>
        public static Mat ApplyAffineTransformation(Mat image, Mat transformationMatrix, Si
        {
            if (image == null || image.IsEmpty)
                throw new ArgumentNullException(nameof(image), "Input image cannot be null c

            if (transformationMatrix == null || transformationMatrix.IsEmpty)
                throw new ArgumentNullException(nameof(transformationMatrix), "Transformatic

            var alignedImage = new Mat();
            CvInvoke.WarpAffine(image, alignedImage, transformationMatrix, outputSize, Inter

            return alignedImage;
        }
    }
}
```

/ImageProcessingLibrary/Interfaces/IFaceAligner.cs

```
using Emgu.CV;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ImageProcessingLibrary.Interfaces
{
    internal interface IFaceAligner
    {
        void AlignFaces(string inputPath, string outputPath);
        Mat AlignFace(Mat image);
    }
}
```

/ImageProcessingLibrary/Interfaces/IImageResizer.cs

```
using Emgu.CV;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ImageProcessingLibrary.Interfaces
{


        public interface IImageResizer
        {
            /// <summary>
            /// Resizes the image while maintaining the aspect ratio, based on a given fixe
            /// </summary>
            /// <param name="inputPath">The path of the input image.</param>
            /// <param name="outputPath">The path where the resized image will be saved.</p
            /// <param name="resizeOption">The resize option, either a fixed size or percen
            /// <param name="dimensionType">Indicates whether the fixed size is for width (
            void ResizeImage(string inputPath, string outputPath, string resizeOption, stri

            /// <summary>
            /// Resizes the image while maintaining the aspect ratio, based on a given fixe
            /// </summary>
            /// <param name="image">The input image as a Mat object.</param>
            /// <param name="fixedSize">The fixed size for either width or height.</param>
            /// <param name="isWidth">Indicates whether the fixed size is for width (true)
            Mat ResizeImageKeepingAspectRatio(Mat image, int fixedSize, bool isWidth);

            /// <summary>
            /// Resizes the image by a given percentage, maintaining the original aspect ra
            /// </summary>
            /// <param name="image">The input image as a Mat object.</param>
            /// <param name="percentage">The percentage by which the image should be resize
            Mat ResizeImageByPercentage(Mat image, int percentage);
        }
}
```

/ImageProcessingLibrary/Logging/Logger.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ImageProcessingLibrary.Logging
{
    public static class Logger
    {
        public static void LogInfo(string message)
        {
            Console.WriteLine($"[INFO] {DateTime.Now}: {message}");
        }

        public static void LogError(string message)
        {
            Console.WriteLine($"[ERROR] {DateTime.Now}: {message}");
        }

        public static void LogWarning(string message)
        {
            Console.WriteLine($"[WARNING] {DateTime.Now}: {message}");
        }
    }
}
```

/ImageProcessingLibrary/PictureAlignment/FaceAligner.cs

```
using DlibDotNet;
using DlibDotNet.Extensions;
using Emgu.CV;
using ImageProcessingLibrary.Exceptions;
using ImageProcessingLibrary.Helpers;
using ImageProcessingLibrary.Interfaces;
using System.Drawing;
using Logger = ImageProcessingLibrary.Logging.Logger;

namespace ImageProcessingLibrary.PictureAlignment
{
    internal class FaceAligner : IFaceAligner
    {
        private readonly ShapePredictor _shapePredictor;
```

```
public FaceAligner()
{
    // Load the pretrained shape predictor model from Dlib
    try
    {
        _shapePredictor = ShapePredictor.Deserialize("shape_predictor_68_face_landma
    }
    catch (Exception ex)
    {
        Logger.LogError("Failed to load shape predictor model.");
        throw new ImageProcessingException("Failed to load shape predictor model.",
    }
}

public void AlignFaces(string inputPath, string outputPath)
{
    try
    {
        Logger.LogInfo($"Starting face alignment for image: {inputPath}");

        if (!File.Exists(inputPath))
        {
            throw new FileNotFoundException($"Input file not found: {inputPath}");
        }

        using (var image = CvInvoke.Imread(inputPath))
        {
            if (image == null || image.IsEmpty)
            {
                throw new ImageProcessingException($"Failed to load image: {inputPat
            }

            using (var alignedImage = AlignFace(image))
            {
                CvInvoke.Imwrite(outputPath, alignedImage);
            }
        }

        Logger.LogInfo($"Successfully aligned face for image: {inputPath} -> {output
    }
    catch (FileNotFoundException ex)
    {
        Logger.LogError(ex.Message);
    }
    catch (ImageProcessingException ex)
    {
        Logger.LogError($"Image processing error: {ex.Message}");
    }
    catch (Exception ex)
    {
        Logger.LogError($"Unexpected error aligning image {inputPath}: {ex.Message}"
    }
}

public Mat AlignFace(Mat image)
{
    try
    {
        // Convert Emgu.CV Mat to Bitmap to use with Dlib
        using (var bitmap = image.ToBitmap())
        {
            // Detect facial landmarks using the helper method
            var landmarks = AlignmentHelper.DetectFacialLandmarks(bitmap, _shapePred

            // Define the desired facial points for alignment
            var desiredPoints = new[]
            {
                new PointF(30.0f, 30.0f), // Left eye
                new PointF(70.0f, 30.0f), // Right eye
                new PointF(50.0f, 70.0f)  // Mouth center
            };

            // Compute affine transformation using the helper method
            var transformation = AlignmentHelper.ComputeAffineTransform(landmarks, n

            // Apply affine transformation using the helper method
            var alignedImage = AlignmentHelper.ApplyAffineTransformation(image, tran

            return alignedImage;
        }
    }
    catch (ArgumentException ex)
    {
        Logger.LogError($"Invalid argument: {ex.Message}");
        throw new ImageProcessingException("Error during face alignment due to inval
    }
    catch (ImageProcessingException ex)
    {
        Logger.LogError($"Image processing error: {ex.Message}");
        throw;
    }
    catch (Exception ex)
```

```
            {
                Logger.LogError($"Unexpected error during face alignment: {ex.Message}");
                throw new ImageProcessingException("Error during face alignment.", ex);
            }
        }
    }
}
```

/ImageProcessingLibrary/PictureSizeAdaptation/ImageResizer.cs

```
using Emgu.CV;
using Emgu.CV.CvEnum;
using ImageProcessingLibrary.Interfaces;
using ImageProcessingLibrary.Logging;
using ImageProcessingLibrary.Exceptions;
using System;
using System.IO;

namespace ImageProcessingLibrary.PictureSizeAdaptation
{

        public class ImageResizer : IImageResizer
        {
            public void ResizeImage(string inputPath, string outputPath, string resizeOptio
            {
                try
                {
                    // Log the start of the resize process
                    Logger.LogInfo($"Starting resizing for image: {inputPath}");

                    // Validate input paths
                    if (!File.Exists(inputPath))
                    {
                        throw new FileNotFoundException($"Input file not found: {inputPath}'
                    }

                    using (var image = CvInvoke.Imread(inputPath))
                    {
                        if (image == null || image.IsEmpty)
                        {
                            throw new ImageProcessingException($"Failed to load image: {inpu
                        }

                        if (resizeOption.EndsWith("%"))
                        {
                            int percentage = int.Parse(resizeOption.TrimEnd('%'));
                using (var resizedImage = ResizeImageByPercentage(image, percent
                {
                    CvInvoke.Imwrite(outputPath, resizedImage);
                }
            }
            else if (int.TryParse(resizeOption, out int fixedSize))
            {
                using (var resizedImage = dimensionType == "width"
                    ? ResizeImageKeepingAspectRatio(image, fixedSize, isWidth: t
                    : ResizeImageKeepingAspectRatio(image, fixedSize, isWidth: f
                {
                    CvInvoke.Imwrite(outputPath, resizedImage);
                }
            }
        }

        // Log the completion of the resize process
        Logger.LogInfo($"Successfully resized image: {inputPath} -> {outputPath}
    }
    catch (FileNotFoundException ex)
    {
        Logger.LogError($"File not found: {ex.Message}");
    }
    catch (ArgumentException ex)
    {
        Logger.LogError($"Invalid argument: {ex.Message}");
    }
    catch (ImageProcessingException ex)
    {
        Logger.LogError($"Image processing error: {ex.Message}");
    }
    catch (Exception ex)
    {
        Logger.LogError($"Unexpected error resizing image {inputPath}: {ex.Messa
    }
}

public Mat ResizeImageKeepingAspectRatio(Mat image, int fixedSize, bool isWidth)
{
    try
    {
        int newWidth, newHeight;

        if (isWidth)
        {
            newWidth = fixedSize;
```

```
                    newHeight = (int)(image.Height * ((double)fixedSize / image.Width));
                }
                else
                {
                    newHeight = fixedSize;
                    newWidth = (int)(image.Width * ((double)fixedSize / image.Height));
                }

                var resizedImage = new Mat();
                CvInvoke.Resize(image, resizedImage, new System.Drawing.Size(newWidth, n

                return resizedImage;
            }
            catch (Exception ex)
            {
                throw new ImageProcessingException("Error while resizing the image while
            }
        }

        public Mat ResizeImageByPercentage(Mat image, int percentage)
        {
            try
            {
                int newWidth = (int)(image.Width * (percentage / 100.0));
                int newHeight = (int)(image.Height * (percentage / 100.0));

                var resizedImage = new Mat();
                CvInvoke.Resize(image, resizedImage, new System.Drawing.Size(newWidth, n

                return resizedImage;
            }
            catch (Exception ex)
            {
                throw new ImageProcessingException("Error while resizing the image by pe
            }
        }
    }
}
```

FaceMorphingLibrary

Dependencies

- DlibDotNet v19.21.0.20220724
- Emgu.CV v4.9.0.5494
- Emgu.CV.Bitmap v4.9.0.5494

- Emgu.CV.runtime.windows v4.9.0.5494
- Xabe.FFmpeg v5.2.6

/FaceMorphingLibrary/Class1.cs

```
namespace FaceMorphingLibrary
{
    public class Class1
    {

    }
}
```

VideoGenerationLibrary

Dependencies

- DlibDotNet v19.21.0.20220724
- Emgu.CV v4.9.0.5494
- Emgu.CV.Bitmap v4.9.0.5494
- Emgu.CV.runtime.windows v4.9.0.5494
- Xabe.FFmpeg v5.2.6

/VideoGenerationLibrary/Class1.cs

```
namespace VideoGenerationLibrary
{
    public class Class1
    {

    }
}
```

ImageProcessingLibrary.Tests

Dependencies

- coverlet.collector v6.0.0
- Emgu.CV.Bitmap v4.9.0.5494
- Emgu.CV.runtime.windows v4.9.0.5494
- Microsoft.NET.Test.Sdk v17.8.0
- NUnit v3.14.0
- NUnit.Analyzers v3.9.0
- NUnit3TestAdapter v4.5.0

/ImageProcessingLibrary.Tests/DirectoryHelperTests.cs

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using ImageProcessingLibrary.Helpers;
using NUnit.Framework;

namespace ImageProcessingLibrary.Tests
{
    [TestFixture]
    public class DirectoryHelperTests
    {
        [Test]
        public void ValidateDirectory_ShouldThrowArgumentNullException_WhenPathIsNull()
        {
            // Act & Assert
            Assert.Throws<ArgumentNullException>(() => DirectoryHelper.ValidateDirectory(nul
        }

        [Test]
        public void ValidateDirectory_ShouldThrowArgumentException_WhenPathIsEmpty()
        {
            // Act & Assert
            Assert.Throws<ArgumentException>(() => DirectoryHelper.ValidateDirectory(""));
        }

        [Test]
        public void ValidateDirectory_ShouldThrowDirectoryNotFoundException_WhenDirectoryDoe
        {
            // Arrange
            string nonExistentDirectory = "C:\\NonExistentDirectory";

            // Act & Assert
            Assert.Throws<DirectoryNotFoundException>(() => DirectoryHelper.ValidateDirecto
        }

        [Test]
        public void ValidateDirectory_ShouldNotThrowException_WhenDirectoryExists()
        {
            // Arrange
            string existingDirectory = Path.GetTempPath();
            // Act & Assert
            Assert.DoesNotThrow(() => DirectoryHelper.ValidateDirectory(existingDirectory))
        }

        [Test]
        public void GetImageFiles_ShouldReturnEmptyList_WhenNoImagesArePresent()
        {
            // Arrange
            string tempDirectory = Path.Combine(Path.GetTempPath(), "EmptyDirectory");
            Directory.CreateDirectory(tempDirectory);

            try
            {
                // Act
                List<string> imageFiles = DirectoryHelper.GetImageFiles(tempDirectory);

                // Assert
                Assert.AreEqual(0, imageFiles.Count);
            }
            finally
            {
                // Cleanup
                Directory.Delete(tempDirectory);
            }
        }

        [Test]
        public void GetImageFiles_ShouldReturnImageFiles_WhenImagesArePresent()
        {
            // Arrange
            string tempDirectory = Path.Combine(Path.GetTempPath(), "ImageDirectory");
            Directory.CreateDirectory(tempDirectory);

            string imagePath1 = Path.Combine(tempDirectory, "image1.jpg");
            string imagePath2 = Path.Combine(tempDirectory, "image2.png");
            File.Create(imagePath1).Dispose();
            File.Create(imagePath2).Dispose();

            try
            {
                // Act
                List<string> imageFiles = DirectoryHelper.GetImageFiles(tempDirectory);

                // Assert
                Assert.AreEqual(2, imageFiles.Count);
                Assert.Contains(imagePath1, imageFiles);
```

```
                Assert.Contains(imagePath2, imageFiles);
            }
            finally
            {
                // Cleanup
                Directory.Delete(tempDirectory, true);
            }
        }
    }
}
```

/ImageProcessingLibrary.Tests/ImageResizerTests.cs

```
using System;
using System.Collections.Generic;
using System.Drawing.Imaging;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using ImageProcessingLibrary.PictureSizeAdaptation;
using NUnit.Framework;

namespace ImageProcessingLibrary.Tests
{
    [TestFixture]
    public class ImageResizerTests
    {
        [Test]
        public void ResizeImageKeepingAspectRatio_ShouldResizeBasedOnWidth_WhenWidthIsProvic
        {
            // Arrange
            var imageResizer = new ImageResizer();
            string tempDirectory = Path.GetTempPath();
            string inputPath = Path.Combine(tempDirectory, "input.jpg");
            string outputPath = Path.Combine(tempDirectory, "output.jpg");

            // Create a valid dummy image file
            using (Bitmap bitmap = new Bitmap(200, 100))
            {
                using (Graphics g = Graphics.FromImage(bitmap))
                {
                    g.Clear(Color.White);
                    g.DrawRectangle(Pens.Black, 10, 10, 180, 80);
                }
            bitmap.Save(inputPath, ImageFormat.Jpeg);
        }

        try
        {
            // Act
            imageResizer.ResizeImage(inputPath, outputPath, "100", "width");

            // Assert
            Assert.IsTrue(File.Exists(outputPath));
            using (var outputImage = Image.FromFile(outputPath))
            {
                Assert.AreEqual(100, outputImage.Width);
                Assert.AreEqual(50, outputImage.Height); // Aspect ratio maintained
            }
        }
        finally
        {
            // Cleanup
            File.Delete(inputPath);
            File.Delete(outputPath);
        }
    }

    [Test]
    public void ResizeImageKeepingAspectRatio_ShouldResizeBasedOnHeight_WhenHeightIsProv
    {
        // Arrange
        var imageResizer = new ImageResizer();
        string tempDirectory = Path.GetTempPath();
        string inputPath = Path.Combine(tempDirectory, "input.jpg");
        string outputPath = Path.Combine(tempDirectory, "output.jpg");

        // Create a valid dummy image file
        using (Bitmap bitmap = new Bitmap(200, 100))
        {
            using (Graphics g = Graphics.FromImage(bitmap))
            {
                g.Clear(Color.White);
                g.DrawRectangle(Pens.Black, 10, 10, 180, 80);
            }
            bitmap.Save(inputPath, ImageFormat.Jpeg);
        }

        try
        {
```

```
        // Act
        imageResizer.ResizeImage(inputPath, outputPath, "50", "height");

        // Assert
        Assert.IsTrue(File.Exists(outputPath));
        using (var outputImage = Image.FromFile(outputPath))
        {
            Assert.AreEqual(100, outputImage.Width); // Aspect ratio maintained
            Assert.AreEqual(50, outputImage.Height);
        }
    }
    finally
    {
        // Cleanup
        File.Delete(inputPath);
        File.Delete(outputPath);
    }
}

[Test]
public void ResizeImageByPercentage_ShouldResizeImageCorrectly_WhenPercentageIsProvi
{
    // Arrange
    var imageResizer = new ImageResizer();
    string tempDirectory = Path.GetTempPath();
    string inputPath = Path.Combine(tempDirectory, "input.jpg");
    string outputPath = Path.Combine(tempDirectory, "output.jpg");

    // Create a valid dummy image file
    using (Bitmap bitmap = new Bitmap(200, 100))
    {
        using (Graphics g = Graphics.FromImage(bitmap))
        {
            g.Clear(Color.White);
            g.DrawRectangle(Pens.Black, 10, 10, 180, 80);
        }
        bitmap.Save(inputPath, ImageFormat.Jpeg);
    }

    try
    {
        // Act
        imageResizer.ResizeImage(inputPath, outputPath, "50%", "");

        // Assert
        Assert.IsTrue(File.Exists(outputPath));
```

```
                using (var outputImage = Image.FromFile(outputPath))
                {
                    Assert.AreEqual(100, outputImage.Width); // 50% of original width
                    Assert.AreEqual(50, outputImage.Height);  // 50% of original height
                }
            }
            finally
            {
                // Cleanup
                File.Delete(inputPath);
                File.Delete(outputPath);
            }
        }
    }
}
```

Sonstige Dateien

Dependencies

- No dependencies found