Arthur Lamidel

La vache est dans le pré

Le projet n'a pas, selon moi, présenté de difficultés particulières, les objets à définir était assez clairs.
Le projet était néanmois intéressant à réaliser car il m'a ammené, au travers de la recherche d'optimisation, à travailler avec les énumérateurs et générateurs; chose à laquelle je n'avais pas vraiment été ammené à utiliser.

L'axe d'amélioration que je pense devoir le plus pousser serait les commentaires dans le code; je ne suis pas encore sûr des parties à commenter, et comment.

À noter que les décorateurs `[ExcludeFromCodeCoverage]` n'ont aucune valeur, et ne sont présent que pour indiquer à mon outil de couverture de tests les méthodes à ignorer.

Program.cs

```csharp
using System.Diagnostics.CodeAnalysis;

using Vache.Models;


namespace Vache;

public static class Program
{
    public const double TOLERANCE = 1e-2;

    [ExcludeFromCodeCoverage]
    public static void Main(string[] args)
    {
        // Ask the user to define the field
        Polygon2 field = Polygon2.FromInput();

        // Display all the infos about said field
        Console.WriteLine($"The area of the field is: {field.Area}");
        Console.WriteLine($"It's center of gravity is at: {field.CenterOfGravity}");
        Console.WriteLine(field.IsPointInside(field.CenterOfGravity)
                              ? "The cow is still in the field!"
                              : "The cow is outside of the field...");
    }
}
```

Models/Point2.cs

```csharp
using System.Diagnostics.CodeAnalysis;
using System.Diagnostics.Contracts;

using Vache.Utils;


namespace Vache.Models;

public class Point2
{
    #region Properties
    /// <summary>
    /// The coordinate on the X axis
    /// </summary>
    public double X { get; }

    /// <summary>
    /// The coordinate on the Y axis
    /// </summary>
    public double Y { get; }
    #endregion

    #region Constructor
    public Point2(double x, double y)
    {
        X = x;
        Y = y;
    }
    #endregion

    #region Methods
    /// <summary>
    /// Translate a point along a vector
    /// </summary>
    /// <param name="vec">The vector to translate along</param>
    /// <returns>A new point at the corresponding coordinates</returns>
    [Pure]
    public Point2 Translate(Vector2 vec)
        => new(X + vec.X, Y + vec.Y);

    /// <summary>
    /// Parse a string into a point object
    /// </summary>
    /// <param name="input">The string to parse</param>
    /// <returns>The parsed point</returns>
    /// <exception cref="ArgumentException">If the input string is not parseable</exception>
    public static Point2 Parse(string input)
    {
        // Check whether the input string match the format
        if (Consts.PointRe.IsMatch(input) is false)
            throw new ArgumentException("Invalid format", nameof(input));

        // Extract the numerical values and put them into an array
        double[] components = Consts.NumberRe.Matches(input)
                                    .Select(match => double.Parse(match.Value))
                                    .ToArray();

        // Since the 1st regex ensures only 2 values, we can address them directly
        return new Point2(components[0], components[1]);
    }

    /// <summary>
    /// Try to parse a string into a point object
    /// </summary>
    /// <param name="input">The string to parse</param>
    /// <param name="output">The parsed point</param>
    /// <returns>Whether the parsing succeeded</returns>
    public static bool TryParse(string input, [NotNullWhen(true)] out Point2? output)
    {
        try
        {
            output = Parse(input);
            return true;
        }
        catch (ArgumentException)
        {
            output = null;

            return false;
        }
    }

    public override string ToString()
        => $"({X}, {Y})";
    #endregion

    #region Operators
    public static bool operator ==(Point2? fst, Point2? snd)
        => fst is null == snd is null || fst?.Equals(snd) is true;

    public static bool operator !=(Point2? fst, Point2? snd)
        => fst is null != snd is null || fst?.Equals(snd) is false;

    public override bool Equals(object? obj)
    {
        if (obj is Point2 other)
            return X - other.X is < 1e-10 and > -1e-10
                && Y - other.Y is < 1e-10 and > -1e-10;

        return false;
    }

    public override int GetHashCode()
        => HashCode.Combine(X, Y);
    #endregion
}
```

Models/Vector2.cs

```
using System.Diagnostics.CodeAnalysis;

using Vache.Utils;


namespace Vache.Models;

public class Vector2
{
    #region Variables
    private double? _norm;
    #endregion

    #region Properties
    /// <summary>
    /// The X component of the vector
    /// </summary>
    public double X { get; }

    /// <summary>
    /// The Y component of the vector
    /// </summary>
    public double Y { get; }

    /// <summary>
    /// The norm of the vector
    /// </summary>
    public double Norm
        => _norm ??= GetNorm();
    #endregion

    #region Constructors
    public Vector2(double x, double y)
    {
        X = x;
```

```
            Y = y;
        }

        public Vector2(Point2 start, Point2 end)
        {
            X = end.X - start.X;
            Y = end.Y - start.Y;
        }
        #endregion

        #region Methods
        /// <summary>
        /// Calculate the scalar product between this and another vector
        /// </summary>
        /// <param name="other">The other vector</param>
        /// <returns>The scalar product of the two vectors</returns>
        public double Scalar(Vector2 other)
            => X * other.X + Y * other.Y;

        /// <summary>
        /// Calculate the determinant between this and another vector
        /// </summary>
        /// <param name="other">The other vector</param>
        /// <returns>The determinant of the two vectors</returns>
        public double Determinant(Vector2 other)
            => X * other.Y - other.X * Y;

        /// <summary>
        /// Calculate the norm of the vector
        /// </summary>
        /// <returns>The norm of the vector</returns>
        private double GetNorm()
            => Math.Sqrt(X * X + Y * Y);

        /// <summary>
        /// Parse a string into a vector object
        /// </summary>
        /// <param name="input">The string to parse</param>
        /// <returns>The parsed vector</returns>
        /// <exception cref="ArgumentException">If the input string is not parseable</exception>
        public static Vector2 Parse(string input)
        {
            // Check whether the input string match the format
            if (Consts.VectorRe.IsMatch(input) is false)
                throw new ArgumentException("Invalid format", nameof(input));

            // Extract the numerical values and put them into an array
            double[] components = Consts.NumberRe.Matches(input)
                                        .Select(match => double.Parse(match.Value))
                                        .ToArray();

            // Since the 1st regex ensures only 2 values, we can address them directly
            return new Vector2(components[0], components[1]);
        }

        /// <summary>
        /// Try to parse a string into a vector object
        /// </summary>
        /// <param name="input">The string to parse</param>
        /// <param name="output">The parsed vector</param>
        /// <returns>Whether the parsing succeeded</returns>
        public static bool TryParse(string input, [NotNullWhen(true)] out Vector2? output)
        {
            try
            {
                output = Parse(input);

                return true;
            }
            catch (ArgumentException)
            {
                output = null;

                return false;
            }
        }

        public override string ToString()
            => $"{{{X}, {Y}}}";
        #endregion
```

```
    #region Operators
    public static Vector2 operator +(Vector2 fst, Vector2 snd)
        => new(fst.X + snd.X, fst.Y + snd.Y);

    public static Vector2 operator -(Vector2 fst, Vector2 snd)
        => new(fst.X - snd.X, fst.Y - snd.Y);

    public static Vector2 operator *(Vector2 vec, double k)
        => new(vec.X * k, vec.Y * k);

    public static Vector2 operator /(Vector2 vec, double k)
        => new(vec.X / k, vec.Y / k);

    public static Vector2 operator -(Vector2 vec)
        => new(-vec.X, -vec.Y);

    public static bool operator ==(Vector2? fst, Vector2? snd)
        => fst is null == snd is null || fst?.Equals(snd) is true;

    public static bool operator !=(Vector2? fst, Vector2? snd)
        => fst is null != snd is null || fst?.Equals(snd) is false;

    public override bool Equals(object? obj)
    {
        if (obj is Vector2 other)
            return X - other.X is < 1e-10 and > -1e-10
                && Y - other.Y is < 1e-10 and > -1e-10;

        return false;
    }

    public override int GetHashCode()
        => HashCode.Combine(X, Y);
    #endregion
}
```

Models/Polygon2.cs

```
using System.Collections;
using System.Diagnostics.CodeAnalysis;
using System.Text;

using Vache.Utils;


namespace Vache.Models;

public class Polygon2
{
    #region Variables
    private double? _area;
    private Point2? _cog;
    #endregion

    #region Properties
    private Point2[] Points { get; }

    public double Area
        => _area ??= GetArea();

    public Point2 CenterOfGravity
        => _cog ??= GetCenterOfGravity();
    #endregion

    #region Constructor
    public Polygon2(IEnumerable<Point2> points)
    {
        Point2[] array = points as Point2[] ?? points.ToArray();

        if (array.Length < 3)
            throw new ArgumentException("Should have at least 3 points", nameof(points));
```

```csharp
            if (array.Distinct().Count() < 3)
                throw new ArgumentException("Should have at least 3 distinct points", nameof(points));

            Points = array;
        }
        #endregion

        #region Methods
        /// <summary>
        /// Determines if a specific point is within the bounds of the polygon
        /// </summary>
        /// <param name="point">The point to check against</param>
        /// <returns>Whether the point is inside or not</returns>
        public bool IsPointInside(Point2 point)
        {
            double output = 0;

            // For each adjacent pair of points fst, snd
            foreach ((Point2 fst, Point2 snd) in Points.Pairs(true))
            {
                // Get their vectors to the point being tested
                Vector2 vector1 = new(fst, point),
                        vector2 = new(snd, point);

                // Calculate the formula and aggregate it into output
                output += Math.Acos(vector1.Scalar(vector2)
                                  / (vector1.Norm * vector2.Norm))
                        * Math.Sign(vector1.Determinant(vector2));
            }

            return output is > Program.TOLERANCE or < -Program.TOLERANCE;
        }

        /// <summary>
        /// Calculate the area of the polygon
        /// </summary>
        /// <returns>The area of the polygon</returns>
        private double GetArea()
        {
            double output = 0;

            // For each adjacent pair of points fst, snd
            foreach ((Point2 fst, Point2 snd) in Points.Pairs(true))
            {
                // Calculate the formula and aggregate it into output
                output += fst.X * snd.Y
                        - fst.Y * snd.X;
            }

            return output / 2d;
        }

        /// <summary>
        /// Calculate the center of gravity of the polygon
        /// </summary>
        /// <returns>The center of gravity of the polygon</returns>
        private Point2 GetCenterOfGravity()
        {
            double outputX = 0;
            double outputY = 0;

            // For each adjacent pair of points fst, snd
            foreach ((Point2 fst, Point2 snd) in Points.Pairs(true))
            {
                // Calculate the formula and aggregate it into the outputs
                outputX += (fst.X + snd.X)
                         * (fst.X * snd.Y - fst.Y * snd.X);
                outputY += (fst.Y + snd.Y)
                         * (fst.X * snd.Y - fst.Y * snd.X);
            }

            double area = Area;

            return new Point2(outputX / (6d * area),
                              outputY / (6d * area));
        }

        /// <summary>
        /// Parse a string into a polygon object
        /// </summary>
        /// <param name="input">The string to parse</param>
        /// <returns>The parsed polygon</returns>
        /// <exception cref="ArgumentException">If the input string is not parseable</exception>
        public static Polygon2 Parse(string input)
        {
            // Check whether the input string match the format
            if (Consts.PolygonRe.IsMatch(input) is false)
                throw new ArgumentException("Invalid format", nameof(input));

            // Extract the numerical values and put them into an enumerable
            IEnumerable<double> matches = Consts.NumberRe.Matches(input)
                                                .Select(match => double.Parse(match.Value));

            // Use an explicit enumerator, allowing more freedom in addressing values
            using IEnumerator<double> matchEnumerator = matches.GetEnumerator();

            List<Point2> points = new();
            // Get two values per iteration
            // Safe since the 1st regex ensures an even number of values
            while (matchEnumerator.MoveNext())
            {
                double x = matchEnumerator.Current;
                matchEnumerator.MoveNext();
                double y = matchEnumerator.Current;

                points.Add(new Point2(x, y));
            }

            return new Polygon2(points);
        }

        /// <summary>
        /// Try to parse a string into a polygon object
        /// </summary>
        /// <param name="input">The string to parse</param>
        /// <param name="output">The parsed polygon</param>
        /// <returns>Whether the parsing succeeded</returns>
        public static bool TryParse(string input, [NotNullWhen(true)] out Polygon2? output)
        {
            try
            {
                output = Parse(input);

                return true;
            }
            catch (ArgumentException)
            {
                output = null;

                return false;
            }
        }

        /// <summary>
        /// Create a field object from user input
        /// </summary>
        /// <returns>The newly created field</returns>
        [ExcludeFromCodeCoverage]
        public static Polygon2 FromInput()
        {
            int nPosts;
            // Ask for the number of points until a valid answer is given
            do
                Console.Write("Enter the number of points: ");
            while (int.TryParse(Console.ReadLine(), out nPosts) is false);

            var posts = new Point2[nPosts];
            for (var i = 0; i < nPosts; i++)
            {
                double postX, postY;

                // Ask for each coordinate until a valid answer is given
                do
                    Console.Write($"Enter point {i + 1}'s x coordinate: ");
                while (double.TryParse(Console.ReadLine(), out postX) is false);

                do
                    Console.Write($"Enter point {i + 1}'s y coordinate: ");
                while (double.TryParse(Console.ReadLine(), out postY) is false);
                posts[i] = new Point2(postX, postY);
                Console.WriteLine();
            }

            return new Polygon2(posts);
        }

        public override string ToString()
        {
            IEnumerator   enumerator = Points.GetEnumerator();
            StringBuilder output     = new();

            enumerator.MoveNext();
            output.Append(enumerator.Current);

            while (enumerator.MoveNext())
                output.Append($", {enumerator.Current}");

            return output.ToString();
        }
        #endregion

        #region Operators
        public static bool operator ==(Polygon2? fst, Polygon2? snd)
            => (fst is null && snd is null) || fst?.Equals(snd) is true;

        public static bool operator !=(Polygon2? fst, Polygon2? snd)
            => fst is null != snd is null || fst?.Equals(snd) is false;

        public override bool Equals(object? obj)
        {
            if (obj is Polygon2 other)
                return Points.Except(other.Points)
                             .Any() is false;

            return false;
        }

        public override int GetHashCode()
            => Points.Aggregate(0, (curr, point) => curr * 17 + point.GetHashCode());
        #endregion
    }
```

Utils/Consts.md

```
using System.Text.RegularExpressions;


namespace Vache.Utils;

public static class Consts
{
    #region Regex
    /// <summary>
    /// Describes any valid IEEE floating point number
    /// </summary>
    private const string NUM_PATTERN = @"[-+]?(\d+\.?|\d*\.\d+)(e[+-]?\d+)?";

    /// <summary>
    /// Regex matching any valid IEEE floating point number
    /// </summary>
    public static readonly Regex NumberRe = new(NUM_PATTERN);

    /// <summary>
    /// Regex matching the pattern "(a, b)"
    /// </summary>
    public static readonly Regex PointRe = new($@"\({NUM_PATTERN}, ?{NUM_PATTERN}\)");

    /// <summary>
    /// Regex matching the pattern "{a, b}"
    /// </summary>
    public static readonly Regex VectorRe = new($@"\{{{NUM_PATTERN}, ?{NUM_PATTERN}\}}");

    /// <summary>
    /// Regex matching the pattern "(a, b), (c, d), ..." repeating at least thrice
    /// </summary>
    public static readonly Regex PolygonRe = new($@"^(?:\({NUM_PATTERN}, ?{NUM_PATTERN}\)(?:, ?|$))
{{3,}}$");
    #endregion
}
```

Utils/EnumerableExtension.md

```
using System.Diagnostics.Contracts;


namespace Vache.Utils;

public static class EnumerableExtension
{
    /// <summary>
    /// Return adjacent pairs of values from the input
    /// </summary>
    /// <param name="input">The enumerable to enumerate over</param>
    /// <param name="cycle">Whether the last element should be paired with the first</param>
    /// <returns>An enumerable of pairs of values</returns>
    [Pure]
    public static IEnumerable<(T, T)> Pairs<T>(this IEnumerable<T> input, bool cycle = false)
    {
        using IEnumerator<T> enumerator = input.GetEnumerator();

        // If the enumerator is empty, stop
        if (enumerator.MoveNext() is false)
            yield break;

        // Store the element from the previous iteration
        T last = enumerator.Current;
        while (enumerator.MoveNext())
        {
            // Get the current element
            T curr = enumerator.Current;

            // Yield the previous element and this one
            yield return (last, curr);

            // The current element is now the previous
            last = curr;
        }

        // If we are not cycling the enumerable, stop
        if (!cycle)
            yield break;

        // Reset to the 1st element
        enumerator.Reset();
        enumerator.MoveNext();

        // Return the very last element and the first one
        yield return (last, enumerator.Current);
    }
}
```

Tests/PointTest.md

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

using Vache.Models;


namespace Vache.Tests;

[TestClass]
public class PointTest
{
    [TestMethod]
    public void PointTranslate1()
    {
        Point2  pnt = new(1, 2);
        Vector2 vec = new(2, 3);
        Point2  res = new(3, 5);

        Assert.AreEqual(res, pnt.Translate(vec));
    }

    [TestMethod]
    public void PointTranslate2()
    {
        Point2  pnt = new(4.5, -2);
        Vector2 vec = new(-3, 0);
        Point2  res = new(1.5, -2);

        Assert.AreEqual(res, pnt.Translate(vec));
    }

    [TestMethod]
    public void PointParseSuccess()
    {
        Assert.IsTrue(Point2.TryParse("(1, 2)", out Point2? point));
        Assert.IsNotNull(point);
    }

    [TestMethod]
    public void PointParseFail1()
    {
        Assert.IsFalse(Point2.TryParse("(1, 2", out Point2? point));
        Assert.IsNull(point);
    }

    [TestMethod]
    public void PointParseFail2()
    {
        Assert.IsFalse(Point2.TryParse("(1 2)", out Point2? point));
        Assert.IsNull(point);
    }

    [TestMethod]
    public void PointParseFail3()
    {
        Assert.IsFalse(Point2.TryParse("(A, 2)", out Point2? point));
        Assert.IsNull(point);
    }

    [TestMethod]
    public void PointString1()
    {
        Point2 pnt = new(1, 2);

        Assert.AreEqual("(1, 2)", pnt.ToString());
    }

    [TestMethod]
    public void PointString2()
    {
        Point2 pnt = new(-1, 2.5);

        Assert.AreEqual("(-1, 2.5)", pnt.ToString());
    }

    [TestMethod]
    public void PointEqual1()
    {
        Point2 pnt1 = new(1, 2),
               pnt2 = new(1, 2);

        Assert.IsTrue(pnt1 == pnt2);
    }

    [TestMethod]
    public void PointEqual2()
    {
        Point2 pnt1 = new(1, 2),
               pnt2 = new(-1, 2);

        Assert.IsTrue(pnt1 != pnt2);
    }

    [TestMethod]
    public void PointEqual3()
    {
        Point2? vec1 = new(1, 2),
                vec2 = null;

        Assert.IsFalse(vec1 == vec2);
    }

    [TestMethod]
    public void PointEqual4()
    {
        Point2? vec1 = new(1, 2),
                vec2 = null;

        Assert.IsTrue(vec1 != vec2);
    }

    [TestMethod]
    public void PointEqual5()
    {
        Point2? vec1 = null,
                vec2 = null;

        Assert.IsTrue(vec1 == vec2);
    }

    [TestMethod]
    public void PointEqual6()
    {
        Point2? vec1 = null,
                vec2 = null;

        Assert.IsFalse(vec1 != vec2);
    }

    [TestMethod]
    public void PointEqual7()
    {
        Point2  pnt = new(1, 2);
        Vector2 vec = new(1, 2);

        // ReSharper disable once SuspiciousTypeConversion.Global
        Assert.IsFalse(pnt.Equals(vec));
    }

    [TestMethod]
    public void PointHash()
    {
        Point2 pnt = new(1, 2),
               res = new(1, 2);

        Assert.IsTrue(pnt.GetHashCode() == res.GetHashCode());
    }
}
```

Tests/VectorTest

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

using Vache.Models;

namespace Vache.Tests;

[TestClass]
public class VectorTest
{
    [TestMethod]
    public void VectorParseSuccess()
    {
        Assert.IsTrue(Vector2.TryParse("{1, -1}", out Vector2? vec));
        Assert.IsNotNull(vec);

        Assert.AreEqual(1, vec.X);
        Assert.AreEqual(-1, vec.Y);
    }

    [TestMethod]
    public void VectorParseFail1()
    {
        Assert.IsFalse(Vector2.TryParse("{1, -1", out Vector2? vec));
        Assert.IsNull(vec);
    }

    [TestMethod]
    public void VectorParseFail2()
    {
        Assert.IsFalse(Vector2.TryParse("{1 -1}", out Vector2? vec));
        Assert.IsNull(vec);
    }

    [TestMethod]
    public void VectorParseFail3()
    {
        Assert.IsFalse(Vector2.TryParse("{A, -1}", out Vector2? vec));
        Assert.IsNull(vec);
    }

    [TestMethod]
    public void VectorString1()
    {
        Vector2 vec = new(1, 2);

        Assert.AreEqual("{1, 2}", vec.ToString());
    }

    [TestMethod]
    public void VectorString2()
    {
        Vector2 vec = new(-1, 2.5);

        Assert.AreEqual("{-1, 2.5}", vec.ToString());
    }

    [TestMethod]
    public void VectorAdd1()
    {
        Vector2 vec1 = new(1, 1),
                vec2 = new(2, 3),
                res  = new(3, 4);

        Assert.AreEqual(res, vec1 + vec2);
    }

    [TestMethod]
    public void VectorAdd2()
    {
        Vector2 vec1 = new(0, 5),
                vec2 = new(-3, -7),
                res  = new(-3, -2);

        Assert.AreEqual(res, vec1 + vec2);
    }

    [TestMethod]
    public void VectorSubtract1()
    {
        Vector2 vec1 = new(2, 3),
                vec2 = new(1, 1),
                res  = new(1, 2);

        Assert.AreEqual(res, vec1 - vec2);
    }

    [TestMethod]
    public void VectorSubtract2()
    {
        Vector2 vec1 = new(-3, 2),
                vec2 = new(-6, 4),
                res  = new(3, -2);

        Assert.AreEqual(res, vec1 - vec2);
    }

    [TestMethod]
    public void VectorNegate()
    {
        Vector2 vec = new(-3, 2),
                res = new(3, -2);

        Assert.AreEqual(res, -vec);
    }

    [TestMethod]
    public void VectorMultiply1()
    {
        Vector2 vec = new(1, 2),
                res = new(2, 4);

        Assert.AreEqual(res, vec * 2);
    }

    [TestMethod]
    public void VectorMultiply2()
    {
        Vector2 vec = new(1, -2),
                res = new(-3, 6);

        Assert.AreEqual(res, vec * -3);
    }

    [TestMethod]
    public void VectorDivide1()
    {
        Vector2 vec = new(1, 2),
                res = new(0.5, 1);

        Assert.AreEqual(res, vec / 2);
    }

    [TestMethod]
    public void VectorDivide2()
    {
        Vector2 vec = new(1, -2),
                res = new(-0.25, 0.5);

        Assert.AreEqual(res, vec / -4);
    }

    [TestMethod]
    public void VectorEqual1()
    {
        Vector2 vec1 = new(1, 2),
                vec2 = new(1, 2);

        Assert.IsTrue(vec1 == vec2);
    }

    [TestMethod]
    public void VectorEqual2()
    {
        Vector2 vec1 = new(1, 2),
                vec2 = new(-1, 2);

        Assert.IsTrue(vec1 != vec2);
    }

    [TestMethod]
    public void VectorEqual3()
    {
        Vector2? vec1 = new(1, 2),
                 vec2 = null;

        Assert.IsFalse(vec1 == vec2);
    }

    [TestMethod]
    public void VectorEqual4()
    {
        Vector2? vec1 = new(1, 2),
                 vec2 = null;

        Assert.IsTrue(vec1 != vec2);
    }

    [TestMethod]
    public void VectorEqual5()
    {
        Vector2? vec1 = null,
                 vec2 = null;

        Assert.IsTrue(vec1 == vec2);
    }

    [TestMethod]
    public void VectorEqual6()
    {
        Vector2? vec1 = null,
                 vec2 = null;

        Assert.IsFalse(vec1 != vec2);
    }

    [TestMethod]
    public void VectorEqual7()
    {
        Vector2 vec = new(1, 2);
        Point2  pnt = new(1, 2);

        // ReSharper disable once SuspiciousTypeConversion.Global
        Assert.IsFalse(vec.Equals(pnt));
    }

    [TestMethod]
    public void VectorHash()
    {
        Vector2 vec = new(1, 2),
                res = new(1, 2);

        Assert.IsTrue(vec.GetHashCode() == res.GetHashCode());
    }
}
```

Tests/PolygonTest

```
    using Microsoft.VisualStudio.TestTools.UnitTesting;

    using Vache.Models;


    namespace Vache.Tests;

    [TestClass]
    public class PolygonTest
    {
        [TestMethod]
        public void PolygonFull1()
        {
            var polygon = new Polygon2(new Point2[] { new(-1, 1), new(-1, -1), new(1, -1), new(1, 1) });

            double area      = polygon.Area;
            Point2 cog       = polygon.CenterOfGravity;
            bool   inPolygon = polygon.IsPointInside(cog);

            Assert.AreEqual(4, area, Program.TOLERANCE);
            Assert.AreEqual(0, cog.X, Program.TOLERANCE);
            Assert.AreEqual(0, cog.Y, Program.TOLERANCE);
            Assert.IsTrue(inPolygon);
        }

        [TestMethod]
        public void PolygonFull2()
        {
            var polygon = new Polygon2(new Point2[] { new(-16.6, -20), new(-12, -18), new(-11, -16), new(-15,
    -15) });

            double area      = polygon.Area;
            Point2 cog       = polygon.CenterOfGravity;
            bool   inPolygon = polygon.IsPointInside(cog);

            Assert.AreEqual(14.4, area, Program.TOLERANCE);
            Assert.AreEqual(-13.95, cog.X, Program.TOLERANCE);
            Assert.AreEqual(-17.25, cog.Y, Program.TOLERANCE);
            Assert.IsTrue(inPolygon);
        }

        [TestMethod]
        public void PolygonFull3()
        {
            var polygon = new Polygon2(new Point2[] { new(-1, -1), new(2, 3), new(5, -1), new(2, 2) });

            double area      = polygon.Area;
            Point2 cog       = polygon.CenterOfGravity;
            bool   inPolygon = polygon.IsPointInside(cog);

            Assert.AreEqual(-3, area, Program.TOLERANCE);
            Assert.AreEqual(2, cog.X, Program.TOLERANCE);
            Assert.AreEqual(1.333, cog.Y, Program.TOLERANCE);
            Assert.IsFalse(inPolygon);
        }

        [TestMethod]
        public void PolygonFull4()
        {
            var polygon = new Polygon2(new Point2[] { new(-1, -1), new(-1, -2), new(2, -5), new(4, 1), new(2,
    -4) });

            double area      = polygon.Area;
            Point2 cog       = polygon.CenterOfGravity;
            bool   inPolygon = polygon.IsPointInside(cog);

            Assert.AreEqual(4, area, Program.TOLERANCE);
            Assert.AreEqual(1.04, cog.X, Program.TOLERANCE);
            Assert.AreEqual(-2.91, cog.Y, Program.TOLERANCE);
            Assert.IsFalse(inPolygon);
        }

        [TestMethod]
        public void PolygonZeroPoints()
        {
            Assert.ThrowsException<ArgumentException>(() => new Polygon2(Array.Empty<Point2>()));
        }

        [TestMethod]
        public void PolygonOnePoint()
        {
            Assert.ThrowsException<ArgumentException>(() => new Polygon2(new Point2[] { new(1, 1) }));
        }

        [TestMethod]
        public void PolygonTwoPoints()
        {
            Assert.ThrowsException<ArgumentException>(() => new Polygon2(new Point2[] { new(1, 1), new(2, 2)
    }));
        }
        [TestMethod]
        public void PolygonNonDistinctPoints()
        {
            Assert.ThrowsException<ArgumentException>(() => new Polygon2(new Point2[] { new(1, 1), new(1, 1),
    new(1, 1) }));
        }

        [TestMethod]
        public void PolygonParseSuccess()
        {
            Assert.IsTrue(Polygon2.TryParse("(-1, 1), (-1, -1), (1, -1), (1, 1)", out Polygon2? polygon));
            Assert.IsNotNull(polygon);

            double area      = polygon.Area;
            Point2 cog       = polygon.CenterOfGravity;
            bool   inPolygon = polygon.IsPointInside(cog);

            Assert.AreEqual(4, area, Program.TOLERANCE);
            Assert.AreEqual(0, cog.X, Program.TOLERANCE);
            Assert.AreEqual(0, cog.Y, Program.TOLERANCE);
            Assert.IsTrue(inPolygon);
        }

        [TestMethod]
        public void PolygonParseFail1()
        {
            Assert.IsFalse(Polygon2.TryParse("(-1), (-1, -1), (1, -1), (1, 1)", out Polygon2? polygon));
            Assert.IsNull(polygon);
        }

        [TestMethod]
        public void PolygonParseFail2()
        {
            Assert.IsFalse(Polygon2.TryParse("(-1, 1), (-1 -1), (1, -1), (1, 1)", out Polygon2? polygon));
            Assert.IsNull(polygon);
        }

        [TestMethod]
        public void PolygonParseFail3()
        {
            Assert.IsFalse(Polygon2.TryParse("(-1, 1), (-1, -1), (b, -1), (1, 1)", out Polygon2? polygon));
            Assert.IsNull(polygon);
        }

        [TestMethod]
        public void PolygonParseFail4()
        {
            Assert.IsFalse(Polygon2.TryParse("(-1, 1), (-1, -1)", out Polygon2? polygon));
            Assert.IsNull(polygon);
        }

        [TestMethod]
        public void PolygonString1()
        {
            Polygon2 polygon = new(new Point2[] { new(1, 1), new(-1, 1), new(-1, -1) });

            Assert.AreEqual("(1, 1), (-1, 1), (-1, -1)", polygon.ToString());
        }

        [TestMethod]
        public void PolygonString2()
        {
            Polygon2 polygon = new(new Point2[] { new(0, -6), new(2.5, 8.4), new(-1.2, 99) });

            Assert.AreEqual("(0, -6), (2.5, 8.4), (-1.2, 99)", polygon.ToString());
        }

        [TestMethod]
        public void PolygonEquals1()
        {
            Polygon2 poly1 = new(new Point2[] { new(1, 1), new(-1, 1), new(-1, -1) }),
                     poly2 = new(new Point2[] { new(1, 1), new(-1, 1), new(-1, -1) });

            Assert.IsTrue(poly1 == poly2);
        }

        [TestMethod]
        public void PolygonEquals2()
        {
            Polygon2 poly1 = new(new Point2[] { new(1, 1), new(-1, 1), new(-1, -1) }),
                     poly2 = new(new Point2[] { new(1, 2), new(-1, 1), new(-1, -1) });

            Assert.IsTrue(poly1 != poly2);
        }

        [TestMethod]
        public void PolygonEquals3()
        {
            Polygon2? poly1 = new(new Point2[] { new(1, 1), new(-1, 1), new(-1, -1) }),
                      poly2 = null;

            Assert.IsFalse(poly1 == poly2);
        }

        [TestMethod]
        public void PolygonEquals4()
        {
            Polygon2? poly1 = new(new Point2[] { new(1, 1), new(-1, 1), new(-1, -1) }),
                      poly2 = null;

            Assert.IsTrue(poly1 != poly2);
        }

        [TestMethod]
        public void PolygonEquals5()
        {
            Polygon2? poly1 = null,
                      poly2 = null;

            Assert.IsTrue(poly1 == poly2);
        }

        [TestMethod]
        public void PolygonEquals6()
        {
            Polygon2? poly1 = null,
                      poly2 = null;

            Assert.IsFalse(poly1 != poly2);
        }

        [TestMethod]
        public void PolygonEquals7()
        {
            Polygon2 poly  = new(new Point2[] { new(1, 1), new(-1, 1), new(-1, -1) });
            Point2[] array = { new(1, 1), new(-1, 1), new(-1, -1) };

            Assert.IsFalse(poly.Equals(array));
        }

        [TestMethod]
        public void PolygonHash()
        {
            Polygon2 poly1 = new(new Point2[] { new(1, 1), new(-1, 1), new(-1, -1) }),
                     poly2 = new(new Point2[] { new(1, 1), new(-1, 1), new(-1, -1) });

            Assert.IsTrue(poly1.GetHashCode() == poly2.GetHashCode());
        }
    }
```

Tests/RegexTest

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

using Vache.Utils;


namespace Vache.Tests;

[TestClass]
public class RegexTest
{
    [TestMethod]
    public void NumberRegex1()
    {
        Assert.IsTrue(Consts.NumberRe.IsMatch("42"));
    }

    [TestMethod]
    public void NumberRegex2()
    {
        Assert.IsTrue(Consts.NumberRe.IsMatch("69.420"));
    }

    [TestMethod]
    public void NumberRegex3()
    {
        Assert.IsTrue(Consts.NumberRe.IsMatch(".666"));
    }

    [TestMethod]
    public void NumberRegex4()
    {
        Assert.IsTrue(Consts.NumberRe.IsMatch("0."));
    }

    [TestMethod]
    public void NumberRegex5()
    {
        Assert.IsTrue(Consts.NumberRe.IsMatch("1e1"));
    }

    [TestMethod]
    public void NumberRegex6()
    {
        Assert.IsTrue(Consts.NumberRe.IsMatch("-18"));
    }

    [TestMethod]
    public void NumberRegex7()
    {
        Assert.IsTrue(Consts.NumberRe.IsMatch("-1e-1"));
    }

    [TestMethod]
    public void NumberRegex8()
    {
        Assert.IsTrue(Consts.NumberRe.IsMatch("+4"));
    }

    [TestMethod]
    public void NumberRegex9()
    {
        Assert.IsFalse(Consts.NumberRe.IsMatch("A"));
    }

    [TestMethod]
    public void NumberRegex10()
    {
        Assert.IsFalse(Consts.NumberRe.IsMatch("."));
    }

    [TestMethod]
    public void NumberRegex11()
    {
        Assert.IsFalse(Consts.NumberRe.IsMatch("e"));
    }
}
```

Tests/PairsTest

```
    using Microsoft.VisualStudio.TestTools.UnitTesting;

    using Vache.Utils;
```

```
namespace Vache.Tests;

[TestClass]
public class PairsTest
{
    [TestMethod]
    public void PairsTest1()
    {
        int[]        testArray   = { 1, 2, 3, 4, 5 };
        (int, int)[] resultArray = { (1, 2), (2, 3), (3, 4), (4, 5), };

        (int, int)[] cycledArray = testArray.Pairs().ToArray();

        for (var i = 0; i < resultArray.Length; i++)
            Assert.AreEqual(resultArray[i], cycledArray[i]);
    }

    [TestMethod]
    public void PairsTest2()
    {
        int[] testArray =
        {
            7, 2, 1, 8, 1,
            2, 0,
        };
        (int, int)[] resultArray =
        {
            (7, 2), (2, 1), (1, 8), (8, 1), (1, 2),
            (2, 0), (0, 7),
        };

        (int, int)[] cycledArray = testArray.Pairs(true).ToArray();

        for (var i = 0; i < cycledArray.Length; i++)
            Assert.AreEqual(resultArray[i], cycledArray[i]);
    }

    [TestMethod]
    public void PairsTestEmpty()
    {
        int[]        testArray   = Array.Empty<int>();
        (int, int)[] resultArray = Array.Empty<(int, int)>();

        (int, int)[] cycledArray = testArray.Pairs().ToArray();

        Assert.AreEqual(resultArray, cycledArray);
    }

    [TestMethod]
    public void PairsTestCycleOne()
    {
        int[]        testArray   = { 0 };
        (int, int)[] resultArray = { (0, 0) };

        (int, int)[] cycledArray = testArray.Pairs(true).ToArray();

        for (var i = 0; i < cycledArray.Length; i++)
            Assert.AreEqual(resultArray[i], cycledArray[i]);
    }
}
```