Models/Point2.cs

```cs
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));
```

```
        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
}
```