Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и
информатики»
(СибГУТИ)

Кафедра прикладной математики и кибернетики

Современные Технологии Программирования 2

Лабораторная работа

«Универсальный калькулятор»

Выполнил:
Студент IV курса ИВТ,
группы ИП-713
Михеев Никита Алексеевич

Работу проверил:
Ассистент кафедры ПМиК
Агалаков Антон Александрович

Новосибирск 2020 г.

Оглавление

1. Задание

1. Разработайте Универсальный калькулятор с интерфейсом в стиле Windows, который позволил бы вычислять выражения с р-ичными числами, простыми дробями, комплексными числами.

2. Калькулятор необходимо снабдить системой справочной.

3. Для установки калькулятора необходимо создать инсталлятор

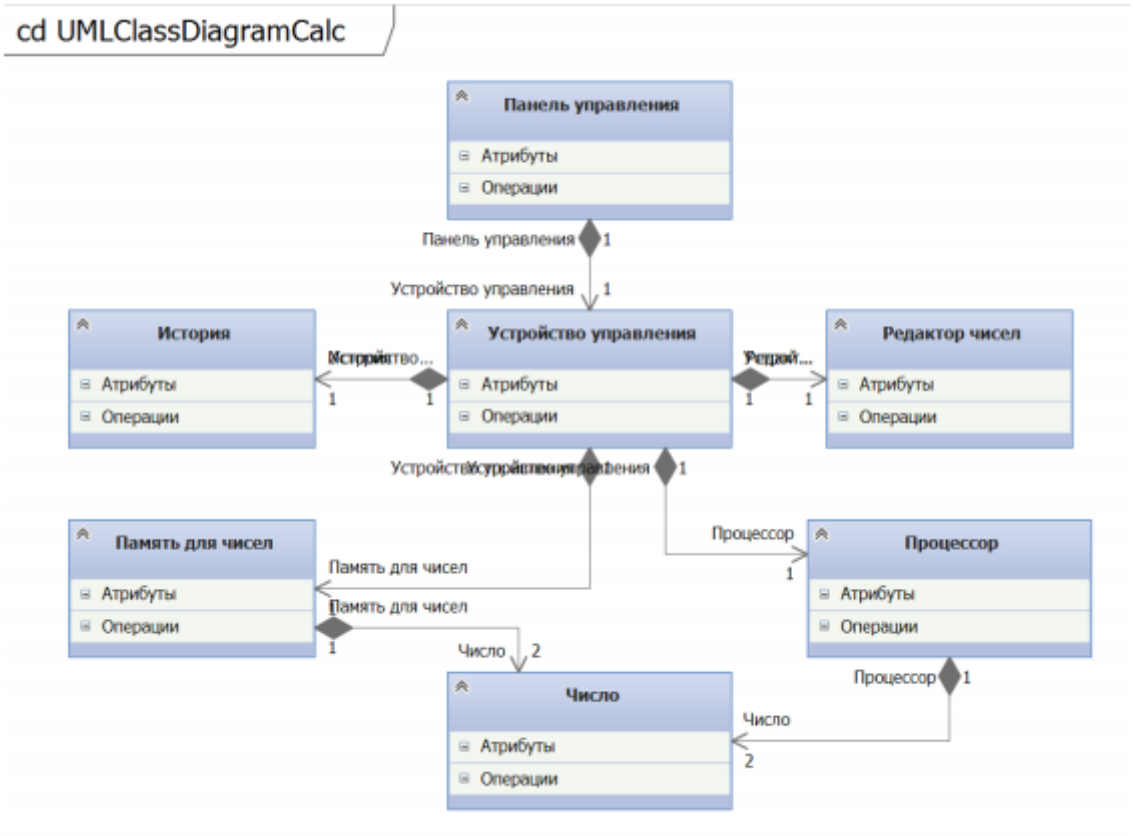


Рис.1 – диаграмма классов UML для калькулятора.

2. Листинг

TFrac.cs:

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

namespace Calculator {
    public sealed class TFrac : ANumber {
        public TNumber Numerator;
        public TNumber Denominator;

        #region Current Class Things
        static void Swap<T>(ref T lhs, ref T rhs) {
            T temp;
```

```
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
public static long GCD(long a, long b) {
    a = Math.Abs(a);
    b = Math.Abs(b);
    while (b > 0) {
        a %= b;
        Swap(ref a, ref b);
    }
    return a;
}
#endregion

#region Constructor
public TFrac() {
    Numerator = new TNumber(0);
    Denominator = new TNumber(1);
}
public TFrac(TNumber a, TNumber b) {
    try {
        if (a < 0 && b < 0) {
            a *= -1;
            b *= -1;
        }
        else if (b < 0 && a > 0) {
            b *= -1;
            a *= -1;
        }
        else if (a == 0 && b == 0 || b == 0 || a == 0 && b == 1) {
            Numerator = new TNumber(0);
            Denominator = new TNumber(1);
            return;
        }
        Numerator = new TNumber(a);
        Denominator = new TNumber(b);
        long gcdResult = GCD((long)a.Number, (long)b.Number);
        if (gcdResult > 1) {
            Numerator /= gcdResult;
            Denominator /= gcdResult;
        }
    } catch {
        throw new OverflowException();
    }
}
public TFrac(int a, int b) {
    if (a < 0 && b < 0) {
        a *= -1;
        b *= -1;
```

```
    }
    else if (b < 0 && a > 0) {
        b *= -1;
        a *= -1;
    }
    else if (a == 0 && b == 0 || b == 0 || a == 0 && b == 1) {
        Numerator = new TNumber(0);
        Denominator = new TNumber(1);
        return;
    }
    Numerator = new TNumber(a);
    Denominator = new TNumber(b);
    long gcdResult = GCD(a, b);
    if (gcdResult > 1) {
        Numerator /= gcdResult;
        Denominator /= gcdResult;
    }
}
public TFrac(string fraction) {
    Regex FracRegex = new Regex(@"^-?(\d+)/(\d+)$");
    Regex NumberRegex = new Regex(@"^-?\d+/?$");
    if (FracRegex.IsMatch(fraction)) {
        List<string> FracParts = fraction.Split('/').ToList();
        Numerator = new TNumber(FracParts[0]);
        Denominator = new TNumber(FracParts[1]);
        if (Denominator.IsZero()) {
            Numerator = new TNumber(0);
            Denominator = new TNumber(1);
            return;
        }
        long gcdResult = GCD((long)Numerator.Number, (long)Denominator.Number);
        if (gcdResult > 1) {
            Numerator /= gcdResult;
            Denominator /= gcdResult;
        }
        return;
    }
    else if (NumberRegex.IsMatch(fraction)) {
        Numerator = new TNumber(fraction);
        Denominator = new TNumber(1);
        return;
    }
    else {
        Numerator = new TNumber(0);
        Denominator = new TNumber(1);
        return;
    }
}
public TFrac(TFrac anotherFrac) {
    Numerator = anotherFrac.Numerator;
```

```
            Denominator = anotherFrac.Denominator;
        }
        #endregion

        #region Override operators
        public static TFrac operator +(TFrac a, TFrac b) {
            TFrac temp;
            temp = new TFrac(a.Numerator * b.Denominator + a.Denominator * b.Numerator, a.Denominator * b.
Denominator);
            return temp;
        }
        public static TFrac operator *(TFrac a, TFrac b) {
            TFrac temp;
            temp = new TFrac(a.Numerator * b.Numerator, a.Denominator * b.Denominator);
            return temp;
        }
        public static TFrac operator -(TFrac a, TFrac b) {
            TFrac temp;
            temp = new TFrac(a.Numerator * b.Denominator - a.Denominator * b.Numerator, a.Denominator * b.
Denominator);
            return temp;
        }
        public static TFrac operator /(TFrac a, TFrac b) {
            if (b.IsZero())
                throw new Exception();
            TFrac temp;
            temp = new TFrac(a.Numerator * b.Denominator, a.Denominator * b.Numerator);
            return temp;
        }
        public static TFrac operator -(TFrac a) {
            return new TFrac(-a.Numerator, a.Denominator);
        }
        public static bool operator ==(TFrac a, TFrac b) {
            return a.Numerator == b.Numerator && a.Denominator == b.Denominator;
        }
        public static bool operator !=(TFrac a, TFrac b) {
            return a.Numerator != b.Numerator && a.Denominator != b.Denominator;
        }
        public static bool operator >(TFrac a, TFrac b) {
            return (a.Numerator / a.Denominator) > (b.Numerator / b.Denominator);
        }
        public static bool operator <(TFrac a, TFrac b) {
            return (a.Numerator / a.Denominator) < (b.Numerator / b.Denominator);
        }
        #endregion

        #region Abstract Override
        public override ANumber Add(ANumber a) {
            TFrac temp;
```

```
            temp = new TFrac(Numerator * (a as TFrac).Denominator + Denominator * (a as TFrac).Numerator,
Denominator * (a as TFrac).Denominator);
            return temp;
        }
        public override ANumber Mul(ANumber a) {
            TFrac temp;
            temp = new TFrac((a as TFrac).Numerator * Numerator, (a as TFrac).Denominator * Denominator);
            return temp;
        }
        public override ANumber Div(ANumber a) {
            TFrac temp;
            temp = new TFrac((a as TFrac).Numerator * Denominator, (a as TFrac).Denominator * Numerator);
            return temp;
        }
        public override ANumber Sub(ANumber a) {
            TFrac temp;
            temp = new TFrac((a as TFrac).Numerator * Denominator - (a as TFrac).Denominator * Numerator,
(a as TFrac).Denominator * Denominator);
            return temp;
        }
        public override object Square() {
            TFrac temp;
            temp = new TFrac((TNumber)Numerator.Square(), (TNumber)Denominator.Square());
            return temp;
        }
        public override object Reverse() {
            return new TFrac(Denominator, Numerator);
        }
        public override bool IsZero() {
            return Numerator.IsZero();
        }
        public override void SetString(string str) {
            TFrac TempFrac = new TFrac(str);
            Numerator = TempFrac.Numerator;
            Denominator = TempFrac.Denominator;
        }
        #endregion

        public override string ToString() {
            return Numerator.ToString() + "/" + Denominator.ToString();
        }
        public override bool Equals(object obj) {
            var frac = obj as TFrac;
            return frac != null &&
                   Numerator == frac.Numerator &&
                   Denominator == frac.Denominator;
        }
    }
}
```

TPNumber.cs:

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

namespace Calculator {
    public sealed class TPNumber : ANumber {
        public static class Conver_10_p {
            public static string Do(double n, int p, int c) {
                if (p < 2 || p > 16)
                    throw new IndexOutOfRangeException();
                if (c < 0 || c > 10)
                    throw new IndexOutOfRangeException();

                string LeftSideString;
                string RightSideString;
                long LeftSide = 0;
                double RightSide = 0f;

                try {
                    LeftSide = (long)n;
                    RightSide = n - LeftSide;
                    if (RightSide < 0)
                        RightSide *= -1;

                    LeftSideString = int_to_P(LeftSide, p);
                    RightSideString = flt_to_P(RightSide, p, c);
                }
                catch {
                    throw new OverflowException();
                }
                return LeftSideString + (RightSideString == String.Empty ? "" : ".") + RightSideString;
            }
            public static char int_to_Char(long d) {
                if (d < 0 || d > 15)
                    throw new IndexOutOfRangeException();

                string SymbolArray = "0123456789ABCDEF";
                return SymbolArray.ElementAt((int)d);
            }
            public static string int_to_P(long n, long p) {
                if (p < 2 || p > 16)
                    throw new IndexOutOfRangeException();
                if (n == 0)
                    return "0";
                bool HaveMinus = false;
                if (n < 0) {
                    HaveMinus = true;
                    n *= -1;
                }
```

```
        string PNumber = string.Empty;

        while (n > 0) {
            PNumber += int_to_Char(n % p);
            n /= p;
        }

        if (HaveMinus)
            PNumber += "-";

        char[] TempArray = PNumber.ToCharArray();
        Array.Reverse(TempArray);
        return new string(TempArray);
    }
    public static string flt_to_P(double n, int p, int c) {
        if (p < 2 || p > 16)
            throw new IndexOutOfRangeException();
        if (c < 0 || c > 10)
            throw new IndexOutOfRangeException();

        string PNumber = string.Empty;
        for (int i = 0; i < c; ++i) {
            PNumber += int_to_Char((int)(n * p));
            n = n * p - (int)(n * p);
        }
        PNumber = PNumber.TrimEnd('0');
        return PNumber;
    }
}
public static class Conver_p_10 {
    private static int char_To_num(char ch) {
        string AllVariants = "0123456789ABCDEF";
        if (!AllVariants.Contains(ch))
            throw new IndexOutOfRangeException();
        return AllVariants.IndexOf(ch);
    }
    private static double convert(string P_num, int P, double weight) {
        if (weight % P != 0)
            throw new Exception();

        long Degree = (long)Math.Ceiling(Math.Log(weight, P)) - 1;
        double Result = 0.0f;

        for (int i = 0; i < P_num.Length; ++i, --Degree)
            Result += char_To_num(P_num.ElementAt(i)) * Math.Pow(P, Degree);

        return Result;
    }
    public static double dval(string P_num, int P) {
        if (P < 2 || P > 16)
```

```
                    throw new IndexOutOfRangeException();
                bool HaveMinus = false;
                if (P_num.First() == '-') {
                    HaveMinus = true;
                    P_num = P_num.Remove(0, 1);
                }
                foreach (char ch in P_num) {
                    if (ch == '.')
                        continue;
                    if (char_To_num(ch) > P)
                        throw new Exception();
                }

                double Number = 0.0f;
                Regex LeftRight = new Regex("^[0-9A-F]+\\.[0-9A-F]+$");
                Regex Right = new Regex("^0\\.[0-9A-F]+$");
                Regex Left = new Regex("^[0-9A-F]+.?$");
                if (LeftRight.IsMatch(P_num)) {
                    Number = convert(P_num.Remove(P_num.IndexOf('.'), 1), P, Math.Pow(P, P_num.IndexOf('.'
)));
                }
                else if (Left.IsMatch(P_num)) {
                    if (P_num.Last() == '.')
                        P_num = P_num.Remove(P_num.Length - 1);
                    Number = convert(P_num, P, Math.Pow(P, P_num.Length));
                }
                else if (Right.IsMatch(P_num)) {
                    Number = convert(P_num.Remove(P_num.IndexOf('.'), 1), P, 0);
                }
                else throw new Exception();

                return HaveMinus ? -Number : Number;
            }
        }

        public TNumber Number;
        public TNumber Notation;
        public TNumber Precision;

        public TPNumber() {
            Number = new TNumber();
            Notation = new TNumber(10);
            Precision = new TNumber(5);
        }
        public TPNumber(TNumber num, TNumber not, TNumber pre) {
            if (not < 2 || not > 16 || pre < 0 || pre > 10) {
                Number = new TNumber();
                Notation = new TNumber(10);
                Precision = new TNumber(5);
            }
```

```
    else {
        Number = new TNumber(num);
        Notation = new TNumber(not);
        Precision = new TNumber(pre);
    }
}
public TPNumber(TNumber num, int not, int pre) {
    if (not < 2 || not > 16 || pre < 0 || pre > 10) {
        Number = new TNumber();
        Notation = new TNumber(10);
        Precision = new TNumber(5);
    }
    else {
        Number = new TNumber(num);
        Notation = new TNumber(not);
        Precision = new TNumber(pre);
    }
}
public TPNumber(double num, int not, int pre) {
    if (not < 2 || not > 16 || pre < 0 || pre > 10) {
        Number = new TNumber();
        Notation = new TNumber(10);
        Precision = new TNumber(5);
    }
    else {
        Number = new TNumber(num);
        Notation = new TNumber(not);
        Precision = new TNumber(pre);
    }
}
public TPNumber(TPNumber anotherTPNumber) {
    Number = anotherTPNumber.Number;
    Notation = anotherTPNumber.Notation;
    Precision = anotherTPNumber.Precision;
}
public TPNumber(string str, TNumber not, TNumber pre) {
    Notation = not;
    Precision = pre;
    try {
        Number = new TNumber(Conver_p_10.dval(str, Convert.ToInt32(not.Number)));
    }
    catch {
        throw new System.OverflowException();
    }
}
public TPNumber(string str, int not, int pre) {
    try {
        Number = new TNumber(Conver_p_10.dval(str, not));
        Notation = new TNumber(not);
        Precision = new TNumber(pre);
```

```
    }
    catch {
        throw new System.OverflowException();
    }
}

public static TPNumber operator +(TPNumber a, TPNumber b) {
    return new TPNumber(a.Number + b.Number, a.Notation, a.Precision);
}
public static TPNumber operator *(TPNumber a, TPNumber b) {
    return new TPNumber(a.Number * b.Number, a.Notation, b.Notation);
}
public static TPNumber operator -(TPNumber a, TPNumber b) {
    return new TPNumber(a.Number - b.Number, a.Notation, a.Precision);
}
public static TPNumber operator /(TPNumber a, TPNumber b) {
    return new TPNumber(a.Number / b.Number, a.Notation, a.Precision);
}
public static TPNumber operator -(TPNumber a) {
    return new TPNumber(-a.Number, a.Notation, a.Precision);
}
public static bool operator ==(TPNumber a, TPNumber b) {
    return a.Number == b.Number;
}
public static bool operator !=(TPNumber a, TPNumber b) {
    return a.Number != b.Number;
}
public static bool operator >(TPNumber a, TPNumber b) {
    return a.Number > b.Number;
}
public static bool operator <(TPNumber a, TPNumber b) {
    return a.Number < b.Number;
}
public override ANumber Add(ANumber a) {
    return new TPNumber((a as TPNumber).Number + Number, Notation, Precision);
}
public override ANumber Mul(ANumber a) {
    return new TPNumber((a as TPNumber).Number * Number, Notation, Precision);
}
public override ANumber Div(ANumber a) {
    return new TPNumber((a as TPNumber).Number / Number, Notation, Precision);
}
public override ANumber Sub(ANumber a) {
    return new TPNumber((a as TPNumber).Number - Number, Notation, Precision);
}
public override object Square() {
    return new TPNumber((TNumber)Number.Square(), Notation, Precision);
}
public override object Reverse() {
    return new TPNumber((TNumber)Number.Reverse(), Notation, Precision);
```

```
        }
        public override bool IsZero() {
            return Number.IsZero();
        }
        public override void SetString(string str) {
            Number = new TNumber(Conver_p_10.dval(str, Convert.ToInt32(Notation.Number)));
        }

        public override string ToString() {
            string str;
            try {
                str = Conver_10_p.Do(Number.Number, Convert.ToInt32(Notation.Number), Convert.ToInt32(Prec
ision.Number));

            }
            catch {
                throw new System.OverflowException();
            }
            return str;
        }
        public override bool Equals(object obj) {
            var number = obj as TPNumber;
            return number != null &&
                   EqualityComparer<TNumber>.Default.Equals(Number, number.Number) &&
                   EqualityComparer<TNumber>.Default.Equals(Notation, number.Notation) &&
                   EqualityComparer<TNumber>.Default.Equals(Precision, number.Precision);
        }
    }
}
```

TComplex.cs:

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

namespace Calculator {
    public sealed class TComplex : ANumber {
        public TNumber Real;
        public TNumber Imaginary;
        const string Separator = " + i * ";
        const int OverflowStringLimit = 15;
        public double Abs() {
            return Math.Sqrt(Real.Number * Real.Number + Imaginary.Number * Imaginary.Number);
        }
        public double GetRad() {
            if (Real > 0)
                return Math.Atan((Imaginary / Real).Number);
            else if (Real == 0 && Imaginary > 0)
                return Math.PI / 2;
            else if (Real < 0 && Imaginary.Number >= 0)
```

```
                return Math.Atan((Imaginary / Real).Number + Math.PI);
            else if (Real < 0 && Imaginary.Number < 0)
                return Math.Atan((Imaginary / Real).Number - Math.PI);
            else if (Real == 0 && Imaginary < 0)
                return -Math.PI / 2;
            return 0;
        }
        public double GetDegree() {
            return GetRad() * 180 / Math.PI;
        }
        public TComplex Pwr(int n) {
            return new TComplex(Math.Pow(Abs(), n) * Math.Cos(n * GetRad()), Math.Pow(Abs(), n) * Math.Sin
(n * GetRad()));
        }
        public TComplex Root(int n, int i) {
            if (i >= n || i < 0 || n < 0)
                return new TComplex();
            return new TComplex(Math.Pow(Abs(), 1.0 / n) * Math.Cos((GetDegree() + 2 * Math.PI * i) / n),
Math.Pow(Abs(), 1.0 / n) * Math.Sin((GetDegree() + 2 * Math.PI * i) / n));
        }

        public TComplex() {
            Real = new TNumber(0);
            Imaginary = new TNumber(0);
        }
        public TComplex(double anReal, double anImaginary) {
            Real = new TNumber(anReal);
            Imaginary = new TNumber(anImaginary);
        }
        public TComplex(int anReal, int anImaginary) {
            Real = new TNumber(anReal);
            Imaginary = new TNumber(anImaginary);
        }
        public TComplex(TNumber anReal, TNumber anImaginary) {
            Real = anReal;
            Imaginary = anImaginary;
        }
        public TComplex(TComplex anotherComplex) {
            Real = anotherComplex.Real;
            Imaginary = anotherComplex.Imaginary;
        }
        public TComplex(string str) {
            Regex FullNumber = new Regex(@"^-?(\d+.?\d*)\s+\+\s+i\s+\*\s+-?(\d+.?\d*)$");
            Regex LeftPart = new Regex(@"^-?(\d+.?\d*)(\s+\+\s+i\s+\*\s+)?$");
            if (FullNumber.IsMatch(str)) {
                List<string> Parts = str.Split(new string[] { Separator }, StringSplitOptions.None).ToList
();
                Real = new TNumber(Parts[0]);
                Imaginary = new TNumber(Parts[1]);
            }
```

```
            else if (LeftPart.IsMatch(str)) {
                if (str.Contains(Separator))
                    str = str.Replace(Separator, string.Empty);
                Real = new TNumber(str);
                Imaginary = new TNumber();
            }
            else {
                Real = new TNumber(0);
                Imaginary = new TNumber(0);
            }
        }

        public static TComplex operator +(TComplex a, TComplex b) {
            TComplex toReturn = new TComplex(a.Real + b.Real, a.Imaginary + b.Imaginary);
            if (toReturn.Real.ToString().Length > OverflowStringLimit)
                throw new OverflowException();
            else if (toReturn.Imaginary.ToString().Length > OverflowStringLimit)
                throw new OverflowException();
            return toReturn;
        }
        public static TComplex operator *(TComplex a, TComplex b) {
            TComplex toReturn = new TComplex(a.Real * b.Real - a.Imaginary - b.Imaginary, a.Real * b.Imagi
nary + b.Imaginary * a.Real);
            if (toReturn.Real.ToString().Length > OverflowStringLimit)
                throw new OverflowException();
            else if (toReturn.Imaginary.ToString().Length > OverflowStringLimit)
                throw new OverflowException();
            return toReturn;
        }
        public static TComplex operator -(TComplex a, TComplex b) {
            TComplex toReturn = new TComplex(a.Real - b.Real, a.Imaginary - b.Imaginary);
            if (toReturn.Real.ToString().Length > OverflowStringLimit)
                throw new OverflowException();
            else if (toReturn.Imaginary.ToString().Length > OverflowStringLimit)
                throw new OverflowException();
            return toReturn;
        }
        public static TComplex operator /(TComplex a, TComplex b) {
            TComplex toReturn = new TComplex((a.Real * b.Real + a.Imaginary * b.Imaginary) / (b.Real * b.R
eal + b.Imaginary + b.Imaginary), (b.Real * a.Imaginary - a.Real * b.Imaginary) / (b.Real * b.Real + b.Ima
ginary * b.Imaginary));
            if (toReturn.Real.ToString().Length > OverflowStringLimit)
                throw new OverflowException();
            else if (toReturn.Imaginary.ToString().Length > OverflowStringLimit)
                throw new OverflowException();
            return toReturn;
        }
        public static TComplex operator -(TComplex a) {
            return new TComplex(-a.Real, a.Imaginary);
        }
```

```
        public static bool operator ==(TComplex a, TComplex b) {
            return (a.Real == b.Real && a.Imaginary == b.Imaginary);
        }
        public static bool operator !=(TComplex a, TComplex b) {
            return (a.Real != b.Real || a.Imaginary != b.Imaginary);
        }

        public override ANumber Add(ANumber a) {
            TComplex toReturn = new TComplex(Real + (a as TComplex).Real, Imaginary + (a as TComplex).Imag
inary);
            if (toReturn.Real.ToString().Length > OverflowStringLimit)
                throw new OverflowException();
            else if (toReturn.Imaginary.ToString().Length > OverflowStringLimit)
                throw new OverflowException();
            return toReturn;
        }
        public override ANumber Mul(ANumber a) {
            TComplex toReturn = new TComplex(Real * (a as TComplex).Real - Imaginary - (a as TComplex).Ima
ginary, Real * (a as TComplex).Imaginary + (a as TComplex).Imaginary * Real);
            if (toReturn.Real.ToString().Length > OverflowStringLimit)
                throw new OverflowException();
            else if (toReturn.Imaginary.ToString().Length > OverflowStringLimit)
                throw new OverflowException();
            return toReturn;
        }
        public override ANumber Div(ANumber a) {
            TComplex toReturn = new TComplex((Real * (a as TComplex).Real + Imaginary * (a as TComplex).Im
aginary) / ((a as TComplex).Real * (a as TComplex).Real + (a as TComplex).Imaginary + (a as TComplex).Imag
inary), ((a as TComplex).Real * Imaginary - Real * (a as TComplex).Imaginary) / ((a as TComplex).Real * (a
 as TComplex).Real + (a as TComplex).Imaginary * (a as TComplex).Imaginary));
            if (toReturn.Real.ToString().Length > OverflowStringLimit)
                throw new OverflowException();
            else if (toReturn.Imaginary.ToString().Length > OverflowStringLimit)
                throw new OverflowException();
            return toReturn;
        }
        public override ANumber Sub(ANumber a) {
            TComplex toReturn = new TComplex(Real - (a as TComplex).Real, Imaginary - (a as TComplex).Imag
inary);
            if (toReturn.Real.ToString().Length > OverflowStringLimit)
                throw new OverflowException();
            else if (toReturn.Imaginary.ToString().Length > OverflowStringLimit)
                throw new OverflowException();
            return toReturn;
        }
        public override object Square() {
            TComplex toReturn = new TComplex(Real * Real - Imaginary * Imaginary, Real * Imaginary + Real
* Imaginary);
            if (toReturn.Real.ToString().Length > OverflowStringLimit)
                throw new OverflowException();
```

```
            else if (toReturn.Imaginary.ToString().Length > OverflowStringLimit)
                throw new OverflowException();
            return toReturn;
        }
        public override object Reverse() {
            TComplex toReturn = new TComplex(Real / (Real * Real + Imaginary * Imaginary), -
(Imaginary / (Real * Real + Imaginary * Imaginary)));
            if (toReturn.Real.ToString().Length > OverflowStringLimit)
                throw new OverflowException();
            else if (toReturn.Imaginary.ToString().Length > OverflowStringLimit)
                throw new OverflowException();
            return toReturn;
        }
        public override bool IsZero() {
            return Real.IsZero() && Imaginary.IsZero();
        }
        public override void SetString(string str) {
            TComplex temp = new TComplex(str);
            Real = temp.Real;
            Imaginary = temp.Imaginary;
        }

        public override string ToString() {
            return Real.ToString() + Separator + Imaginary.ToString();
        }
        public override bool Equals(object obj) {
            var complex = obj as TComplex;
            return complex != null &&
                   EqualityComparer<TNumber>.Default.Equals(Real, complex.Real) &&
                   EqualityComparer<TNumber>.Default.Equals(Imaginary, complex.Imaginary);
        }
    }
}
```

TFracEditor.cs:

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

namespace Calculator {
    public sealed class FracEditor : AEditor {
        private string number;
        public override string Number {
            get => number;
            set {
                number = new TFrac(value).ToString();
            }
        }

        const string ZeroFraction = "0/";
        const string Separator = "/";
```

```
const int LeftSideOnlyLimit = 14;
const int BothSideLimit = 22;

public FracEditor() {
    number = "0";
}
public FracEditor(int a, int b) {
    number = new TFrac(a, b).ToString();
}
public FracEditor(string str) {
    number = new TFrac(str).ToString();
}

public override bool IsZero() {
    return number.StartsWith(ZeroFraction) || number == "0";
}
public override string ToogleSign() {
    if (number.ElementAt(0) == '-')
        number = number.Remove(0, 1);
    else
        number = "-" + number;
    return number;
}
public override string AddNumber(int a) {
    if (!HaveSeparator() && number.Length > LeftSideOnlyLimit)
        return number;
    else if (number.Length > BothSideLimit)
        return number;
    if (a < 0 || a > 9)
        return number;
    if (a == 0)
        AddZero();
    else if (number == "0" || number == "-0")
        number = number.First() == '-' ? "-" + a.ToString() : a.ToString();
    else
        number += a.ToString();
    return number;
}
public override bool Equals(object obj) {
    return obj is FracEditor editor &&
           number == editor.number;
}
public override string AddZero() {
    if (HaveSeparator() && number.Last().ToString() == Separator)
        return number;
    if (number == "0" || number == "0/")
        return number;
    number += "0";
    return number;
}
```

```
public override string RemoveSymbol() {
    if (number.Length == 1)
        number = "0";
    else if (number.Length == 2 && number.First() == '-')
        number = "-0";
    else
        number = number.Remove(number.Length - 1);
    return number;
}
public override string Clear() {
    number = "0";
    return number;
}
public override string Edit(Enum com) {
    switch (com) {
        case Command.cZero:
            AddZero();
            break;
        case Command.cOne:
            AddNumber(1);
            break;
        case Command.cTwo:
            AddNumber(2);
            break;
        case Command.cThree:
            AddNumber(3);
            break;
        case Command.cFour:
            AddNumber(4);
            break;
        case Command.cFive:
            AddNumber(5);
            break;
        case Command.cSix:
            AddNumber(6);
            break;
        case Command.cSeven:
            AddNumber(7);
            break;
        case Command.cEight:
            AddNumber(8);
            break;
        case Command.cNine:
            AddNumber(9);
            break;
        case Command.cSign:
            ToogleSign();
            break;
        case Command.cSeparator:
            AddSeparator();
```

```
                break;
            case Command.cBS:
                RemoveSymbol();
                break;
            case Command.CE:
                Clear();
                break;
            default:
                break;
        }
        return Number;
    }
    public override string AddSeparator() {
        if (!number.Contains(Separator))
            number += Separator;
        return number;
    }
    public override bool HaveSeparator() {
        return number.Contains(Separator);
    }
    public override string ToString() {
        return Number;
    }
  }
}
```

TPnumberEditor.cs:

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

namespace Calculator {
    public sealed class PNumberEditor : AEditor {
        private string number;
        public override string Number {
            get {
                return number;
            }
            set {
                number = new TPNumber(value, Notation, Precision).ToString();
            }
        }

        public TNumber Notation;
        public TNumber Precision;
        const int LeftSideOnlyLimit = 12;
        const int BothSideLimit = 22;

        Regex ZeroPNumber = new Regex("^-?(0+|.?0+|0+.(0+)?)$");
        const string Separator = ".";
```

```
        public PNumberEditor() {
            number = "0";
            Notation = new TNumber(10);
            Precision = new TNumber(5);
        }
        public PNumberEditor(string str, TNumber not, TNumber pre) {
            if (not < 2 || not > 16 || pre < 0 || pre > 10) {
                number = "0";
                Notation = new TNumber(10);
                Precision = new TNumber(5);
            }
            else {
                Notation = not;
                Precision = pre;
                number = new TPNumber(str, Notation, Precision).ToString();
            }
        }
        public PNumberEditor(double num, TNumber not, TNumber pre) {
            if (not < 2 || not > 16 || pre < 0 || pre > 10) {
                number = "0";
                Notation = new TNumber(10);
                Precision = new TNumber(5);
            }
            else {
                Notation = not;
                Precision = pre;
                number = new TPNumber(num, Convert.ToInt32(Notation.Number), Convert.ToInt32(Precision.Num
ber)).ToString(); ;
            }
        }
        public PNumberEditor(double num, int not, int pre) {
            if (not < 2 || not > 16 || pre < 0 || pre > 10) {
                number = "0";
                Notation = new TNumber(10);
                Precision = new TNumber(5);
            }
            else {
                Notation = new TNumber(not);
                Precision = new TNumber(pre);
                number = TPNumber.Conver_10_p.Do(num, not, pre);
            }
        }
        public PNumberEditor(string str) {
            Notation = new TNumber(10);
            Precision = new TNumber(5);
            number = new TPNumber(str, Notation, Precision).ToString();
        }
        public override bool IsZero() {
            return ZeroPNumber.IsMatch(number);
        }
```

```
        public override string ToogleSign() {
            if (number.ElementAt(0) == '-')
                number = number.Remove(0, 1);
            else
                number = "-" + number;
            return number;
        }
        public override string AddNumber(int num) {
            if (!HaveSeparator() && number.Length > LeftSideOnlyLimit)
                return number;
            else if (number.Length > BothSideLimit)
                return number;
            if (num < 0 || num >= Notation.Number)
                return number;
            if (num == 0)
                AddZero();
            else if (number == "0" || number == "-0")
                number = number.First() == '-' ? "-
" + TPNumber.Conver_10_p.int_to_Char(num).ToString() : TPNumber.Conver_10_p.int_to_Char(num).ToString();
            else
                number += TPNumber.Conver_10_p.int_to_Char(num).ToString();
            return number;
        }
        public override bool Equals(object obj) {
            var editor = obj as PNumberEditor;
            return editor != null &&
                   number == editor.number &&
                   EqualityComparer<TNumber>.Default.Equals(Notation, editor.Notation) &&
                   EqualityComparer<TNumber>.Default.Equals(Precision, editor.Precision) &&
                   Number == editor.Number &&
                   EqualityComparer<Regex>.Default.Equals(ZeroPNumber, editor.ZeroPNumber);
        }

        public override string RemoveSymbol() {
            if (number.Length == 1)
                number = "0";
            else if (number.Length == 2 && number.First() == '-')
                number = "-0";
            else
                number = number.Remove(number.Length - 1);
            return number;
        }
        public override string Clear() {
            number = "0";
            return number;
        }
        public override string Edit(Enum com) {
            switch (com) {
                case Command.cZero:
                    AddZero();
```

```
    break;
case Command.cOne:
    AddNumber(1);
    break;
case Command.cTwo:
    AddNumber(2);
    break;
case Command.cThree:
    AddNumber(3);
    break;
case Command.cFour:
    AddNumber(4);
    break;
case Command.cFive:
    AddNumber(5);
    break;
case Command.cSix:
    AddNumber(6);
    break;
case Command.cSeven:
    AddNumber(7);
    break;
case Command.cEight:
    AddNumber(8);
    break;
case Command.cNine:
    AddNumber(9);
    break;
case Command.cA:
    AddNumber(10);
    break;
case Command.cB:
    AddNumber(11);
    break;
case Command.cC:
    AddNumber(12);
    break;
case Command.cD:
    AddNumber(13);
    break;
case Command.cE:
    AddNumber(14);
    break;
case Command.cF:
    AddNumber(15);
    break;
case Command.cSign:
    ToogleSign();
    break;
case Command.cSeparator:
```

```
                AddSeparator();
                break;
            case Command.cBS:
                RemoveSymbol();
                break;
            case Command.CE:
                Clear();
                break;
            default:
                break;
        }
        return Number;
    }
    public override string AddSeparator() {
        if (!number.Contains(Separator))
            number += Separator;
        return number;
    }
    public override bool HaveSeparator() {
        return number.Contains(Separator);
    }
    public override string AddZero() {
        if (HaveSeparator() && number.Last().ToString() == Separator)
            return number;
        if (number == "0" || number == "0.")
            return number;
        number += "0";
        return number;
    }
    public override string ToString() {
        return number;
    }
  }
}
```

TComplexEdiror.cs:

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

namespace Calculator {
    public sealed class ComplexEditor : AEditor {
        private string number;
        public override string Number {
            get => number;
            set {
                number = new TComplex(value).ToString();
            }
        }
        const int OverflowStringLimit = 15;
```

```
        Regex ZeroComplex = new Regex(@"^-?(0+.?0*)(\s*\+\s*i\s*\*\s*-?(0+.?0*)|(\s*\+\s*i\s*\*\s*-
?))?$");
        const string Separator = " + i * ";
        public ComplexEditor() {
            number = "0";
        }
        public ComplexEditor(int a, int b) {
            number = new TComplex(a, b).ToString();
        }
        public ComplexEditor(string str) {
            number = new TComplex(str).ToString();
        }
        public override bool IsZero() {
            return ZeroComplex.IsMatch(number);
        }
        public override string ToogleSign() {
            if (HaveSeparator()) {
                List<string> Parts = new List<string>();
                Parts = Number.Split(new string[] { Separator }, StringSplitOptions.None).ToList();
                if (Parts[0].First() == '-')
                    Parts[0] = Parts[0].Remove(0, 1);
                else
                    Parts[0] = '-' + Parts[0];
                if (Parts[1].First() == '-')
                    Parts[1] = Parts[1].Remove(0, 1);
                else
                    Parts[1] = '-' + Parts[1];
                number = Parts[0] + Separator + Parts[1];
                return number;
            }
            if (number.First() == '-')
                number = number.Remove(0, 1);
            else
                number = '-' + number;
            return number;
        }
        public override string AddNumber(int a) {
            if (a < 0 || a > 9)
                return number;
            if (a == 0)
                AddZero();
            else if (number == "0" || number == "-0")
                number = number.First() == '-' ? "-" + a.ToString() : a.ToString();
            else if (number.EndsWith(" 0") || number.EndsWith(" -0"))
                number = number.Remove(number.Length - 1) + a.ToString();
            else number += a.ToString();
            return number;
        }

        public override string AddZero() {
```

```
            if (number == "0" || number == "-0" || number.EndsWith(" 0") || number.EndsWith(" -
0") || number.EndsWith(Separator))
                return number;
            number += "0";
            return number;
        }
        public override string RemoveSymbol() {
            if (number.Length == 1)
                number = "0";
            else if (number.Length == 2 && Number.First() == '-')
                number = "-0";
            else if (HaveSeparator() && Number.ElementAt(Number.Length - 2) == ' ')
                number = number.Remove(number.IndexOf(Separator));
            else
                number = number.Remove(number.Length - 1);
            return number;
        }
        public override string Clear() {
            number = "0";
            return Number;
        }
        public override string Edit(Enum com) {
            switch (com) {
                case Command.cZero:
                    AddZero();
                    break;
                case Command.cOne:
                    AddNumber(1);
                    break;
                case Command.cTwo:
                    AddNumber(2);
                    break;
                case Command.cThree:
                    AddNumber(3);
                    break;
                case Command.cFour:
                    AddNumber(4);
                    break;
                case Command.cFive:
                    AddNumber(5);
                    break;
                case Command.cSix:
                    AddNumber(6);
                    break;
                case Command.cSeven:
                    AddNumber(7);
                    break;
                case Command.cEight:
                    AddNumber(8);
                    break;
```

```
                case Command.cNine:
                    AddNumber(9);
                    break;
                case Command.cSign:
                    ToogleSign();
                    break;
                case Command.cSeparator:
                    AddNumberSeparator();
                    break;
                case Command.cBS:
                    RemoveSymbol();
                    break;
                case Command.CE:
                    Clear();
                    break;
                case Command.cNumbSeparator:
                    AddSeparator();
                    break;
                default:
                    break;
            }
            return Number;
        }
        public override string AddSeparator() {
            if (!HaveSeparator())
                Number = string.Concat(Number, Separator, "0");
            return Number;
        }
        public override bool HaveSeparator() {
            return Number.Contains(Separator);
        }
        public override string ToString() {
            return Number;
        }

        public string AddNumberSeparator() {
            if (!HaveSeparator() && !number.Contains("."))
                number += ".";
            else if (HaveSeparator()) {
                List<string> Parts = new List<string>();
                Parts = Number.Split(new string[] { Separator }, StringSplitOptions.None).ToList();
                if (!Parts[1].Contains("."))
                    number += ".";
            }
            return number;
        }
    }
}
```

Form1.cs:

```
using System;
```

```
using System.Linq;
using System.Windows.Forms;

namespace Calculator {
    public partial class Form1 : Form {
        TCtrl<TFrac, FracEditor> fracController;
        TCtrl<TPNumber, PNumberEditor> pNumberController;
        TCtrl<TComplex, ComplexEditor> complexController;
        const string TAG_FRAC = "FRAC_";
        const string TAG_COMPLEX = "COMPLEX_";
        const string TAG_PNUMBER = "PNUMBER_";
        const string OPERATIONS = "+-/*";
        bool PNumberMode = true;
        bool FracMode = true;
        bool ComplexMode = true;
        enum ComplexFunctions {
            Pwr, Root, Abs, Dgr, Rad
        }

        private string NumberBeatifier(string Tag, string str) {
            if (str == "ERROR")
                return str;
            string ToReturn = str;
            switch (Tag) {
                case TAG_PNUMBER:
                    break;
                case TAG_FRAC:
                    if (FracMode == true)
                        ToReturn = str;
                    else if (new TFrac(str).Denominator == 1)
                        ToReturn = new TFrac(str).Numerator.ToString();
                    break;
                case TAG_COMPLEX:
                    if (ComplexMode == true)
                        ToReturn = str;
                    else if (new TComplex(str).Imaginary == 0)
                        ToReturn = new TComplex(str).Real.ToString();
                    break;
            }
            return ToReturn;
        }
        private static AEditor.Command CharToEditorCommand(char ch) {
            AEditor.Command command = AEditor.Command.cNone;
            switch (ch) {
                case '0':
                    command = AEditor.Command.cZero;
                    break;
                case '1':
                    command = AEditor.Command.cOne;
                    break;
```

```
    case '2':
        command = AEditor.Command.cTwo;
        break;
    case '3':
        command = AEditor.Command.cThree;
        break;
    case '4':
        command = AEditor.Command.cFour;
        break;
    case '5':
        command = AEditor.Command.cFive;
        break;
    case '6':
        command = AEditor.Command.cSix;
        break;
    case '7':
        command = AEditor.Command.cSeven;
        break;
    case '8':
        command = AEditor.Command.cEight;
        break;
    case '9':
        command = AEditor.Command.cNine;
        break;
    case 'A':
        command = AEditor.Command.cA;
        break;
    case 'B':
        command = AEditor.Command.cB;
        break;
    case 'C':
        command = AEditor.Command.cC;
        break;
    case 'D':
        command = AEditor.Command.cD;
        break;
    case 'E':
        command = AEditor.Command.cE;
        break;
    case 'F':
        command = AEditor.Command.cF;
        break;
    case '.':
        command = AEditor.Command.cSeparator;
        break;
    case '-':
        command = AEditor.Command.cSign;
        break;
}
return command;
```

```
}
private static TProc<T>.Oper CharToOperationsCommand<T>(char ch) where T : ANumber, new() {
    TProc<T>.Oper command = TProc<T>.Oper.None;
    switch (ch) {
        case '+':
            command = TProc<T>.Oper.Add;
            break;
        case '-':
            command = TProc<T>.Oper.Sub;
            break;
        case '*':
            command = TProc<T>.Oper.Mul;
            break;
        case '/':
            command = TProc<T>.Oper.Div;
            break;
    }
    return command;
}
private static AEditor.Command KeyCodeToEditorCommand(Keys ch) {
    AEditor.Command command = AEditor.Command.cNone;
    switch (ch) {
        case Keys.Back:
            command = AEditor.Command.cBS;
            break;
        case Keys.Delete:
        case Keys.Escape:
            command = AEditor.Command.CE;
            break;
    }
    return command;
}

public Form1() {
    fracController = new TCtrl<TFrac, FracEditor>();
    pNumberController = new TCtrl<TPNumber, PNumberEditor>();
    complexController = new TCtrl<TComplex, ComplexEditor>();
    InitializeComponent();
    Size = new System.Drawing.Size(310, 382);
}

private void Button_Number_Edit(object sender, EventArgs e) {
    Button button = (Button)sender;
    string FullTag = button.Tag.ToString();
    if (FullTag.StartsWith(TAG_FRAC)) {
        Enum.TryParse(FullTag.Replace(TAG_FRAC, string.Empty), out AEditor.Command ParsedEnum);
        tB_Frac.Text = fracController.ExecCommandEditor(ParsedEnum);
    }
    else if (FullTag.StartsWith(TAG_COMPLEX)) {
        Enum.TryParse(FullTag.Replace(TAG_COMPLEX, string.Empty), out AEditor.Command ParsedEnum);
```

```
            tB_Complex.Text = complexController.ExecCommandEditor(ParsedEnum);
        }
        else if (FullTag.StartsWith(TAG_PNUMBER)) {
            pNumberController.Edit.Notation = new TNumber(trackBar_PNumber.Value);
            Enum.TryParse(FullTag.Replace(TAG_PNUMBER, string.Empty), out AEditor.Command ParsedEnum);
            tB_PNumber.Text = pNumberController.ExecCommandEditor(ParsedEnum);
        }
    }

    private void Button_Number_Operation(object sender, EventArgs e) {
        Button button = (Button)sender;
        string FullTag = button.Tag.ToString();
        if (FullTag.StartsWith(TAG_FRAC)) {
            string Command = FullTag.Replace(TAG_FRAC, string.Empty);
            Enum.TryParse(Command, out TProc<TFrac>.Oper ParsedEnum);
            tB_Frac.Text = NumberBeatifier(TAG_FRAC, fracController.ExecOperation(ParsedEnum));
        }
        else if (FullTag.StartsWith(TAG_COMPLEX)) {
            string Command = FullTag.Replace(TAG_COMPLEX, string.Empty);
            Enum.TryParse(Command, out TProc<TComplex>.Oper ParsedEnum);
            tB_Complex.Text = NumberBeatifier(TAG_COMPLEX, complexController.ExecOperation(ParsedEnum)
);
        }
        else if (FullTag.StartsWith(TAG_PNUMBER)) {
            string Command = FullTag.Replace(TAG_PNUMBER, string.Empty);
            Enum.TryParse(Command, out TProc<TPNumber>.Oper ParsedEnum);
            tB_PNumber.Text = pNumberController.ExecOperation(ParsedEnum);
        }
    }

    private void Button_Number_Function(object sender, EventArgs e) {
        Button button = (Button)sender;
        string FullTag = button.Tag.ToString();
        if (FullTag.StartsWith(TAG_FRAC)) {
            string Command = FullTag.Replace(TAG_FRAC, string.Empty);
            Enum.TryParse(Command, out TProc<TFrac>.Func ParsedEnum);
            tB_Frac.Text = NumberBeatifier(TAG_FRAC, fracController.ExecFunction(ParsedEnum));
        }
        else if (FullTag.StartsWith(TAG_COMPLEX)) {
            string Command = FullTag.Replace(TAG_COMPLEX, string.Empty);
            Enum.TryParse(Command, out TProc<TComplex>.Func ParsedEnum);
            tB_Complex.Text = NumberBeatifier(TAG_COMPLEX, complexController.ExecFunction(ParsedEnum))
;
        }
        else if (FullTag.StartsWith(TAG_PNUMBER)) {
            string Command = FullTag.Replace(TAG_PNUMBER, string.Empty);
            Enum.TryParse(Command, out TProc<TPNumber>.Func ParsedEnum);
            tB_PNumber.Text = pNumberController.ExecFunction(ParsedEnum);
        }
    }
```

```
private void Button_Reset(object sender, EventArgs e) {
    Button button = (Button)sender;
    string FullTag = button.Tag.ToString();
    if (FullTag.StartsWith(TAG_FRAC)) {
        tB_Frac.Text = fracController.Reset();
        label_Frac_Memory.Text = string.Empty;
    }
    else if (FullTag.StartsWith(TAG_COMPLEX)) {
        tB_Complex.Text = complexController.Reset();
        label_Complex_Memory.Text = string.Empty;
    }
    else if (FullTag.StartsWith(TAG_PNUMBER)) {
        tB_PNumber.Text = pNumberController.Reset();
        label_PNumber_Memory.Text = string.Empty;
    }
}

private void Button_FinishEval(object sender, EventArgs e) {
    Button button = (Button)sender;
    string FullTag = button.Tag.ToString();
    if (FullTag.StartsWith(TAG_FRAC)) {
        tB_Frac.Text = NumberBeatifier(TAG_FRAC, fracController.Calculate());
    }
    else if (FullTag.StartsWith(TAG_COMPLEX)) {
        tB_Complex.Text = NumberBeatifier(TAG_COMPLEX, complexController.Calculate());
    }
    else if (FullTag.StartsWith(TAG_PNUMBER)) {
        tB_PNumber.Text = pNumberController.Calculate();
    }
}

private void Button_Memory(object sender, EventArgs e) {
    Button button = (Button)sender;
    string FullTag = button.Tag.ToString();
    if (FullTag.StartsWith(TAG_FRAC)) {
        string Command = FullTag.Replace(TAG_FRAC, string.Empty);
        Enum.TryParse(Command, out TMemory<TFrac>.Commands ParsedEnum);
        dynamic exec = fracController.ExecCommandMemory(ParsedEnum, tB_Frac.Text);
        if (ParsedEnum == TMemory<TFrac>.Commands.Copy)
            tB_Frac.Text = exec.Item1.ToString();
        label_Frac_Memory.Text = exec.Item2 == TMemory<TFrac>.NumStates.ON ? "M" : string.Empty;
    }
    else if (FullTag.StartsWith(TAG_COMPLEX)) {
        string Command = FullTag.Replace(TAG_COMPLEX, string.Empty);
        Enum.TryParse(Command, out TMemory<TComplex>.Commands ParsedEnum);
        dynamic exec = complexController.ExecCommandMemory(ParsedEnum, tB_Complex.Text);
        if (ParsedEnum == TMemory<TComplex>.Commands.Copy)
            tB_Complex.Text = exec.Item1.ToString();
```

```
                label_Complex_Memory.Text = exec.Item2 == TMemory<TComplex>.NumStates.ON ? "M" : string.Em
pty;
            }
            else if (FullTag.StartsWith(TAG_PNUMBER)) {
                string Command = FullTag.Replace(TAG_PNUMBER, string.Empty);
                Enum.TryParse(Command, out TMemory<TPNumber>.Commands ParsedEnum);
                dynamic exec = pNumberController.ExecCommandMemory(ParsedEnum, tB_PNumber.Text);
                if (ParsedEnum == TMemory<TPNumber>.Commands.Copy)
                    tB_PNumber.Text = exec.Item1.ToString();
                label_PNumber_Memory.Text = exec.Item2 == TMemory<TPNumber>.NumStates.ON ? "M" : string.Em
pty;
            }
        }

        private void СправкаToolStripMenuItem_Click(object sender, EventArgs e) {
            MessageBox.Show("Выполнил:\nМихеев Н.А.\nГруппа: ИП-
713.", "Универсальный калькулятор", MessageBoxButtons.OK, MessageBoxIcon.Information);
        }

        private void TrackBar_PNumber_ValueChanged(object sender, EventArgs e) {
            label_PNumber_P.Text = trackBar_PNumber.Value.ToString();
            pNumberController.Edit.Notation = new TNumber(trackBar_PNumber.Value);
            tB_PNumber.Text = pNumberController.Reset();
            label_PNumber_Memory.Text = string.Empty;
            string AllowedEndings = "0123456789ABCDEF";
            foreach (Control i in tabPage_PNumber.Controls.OfType<Button>()) {
                if (AllowedEndings.Contains(i.Name.ToString().Last()) && i.Name.ToString().Substring(i.Nam
e.ToString().Length - 2, 1) == "_") {
                    int j = AllowedEndings.IndexOf(i.Name.ToString().Last());
                    if (j < trackBar_PNumber.Value) {
                        i.Enabled = true;
                    }
                    if ((j >= trackBar_PNumber.Value) && (j <= 15)) {
                        i.Enabled = false;
                    }
                }
            }
            pNumberController.Proc.Lop_Res.Notation = new TNumber(trackBar_PNumber.Value);
            pNumberController.Proc.Rop.Notation = new TNumber(trackBar_PNumber.Value);
        }

        private void tabControl_SelectedIndexChanged(object sender, EventArgs e) {
            switch (tabControl.SelectedIndex) {
                case 0:
                    Size = new System.Drawing.Size(310, 382);
                    break;
                case 1:
                    Size = new System.Drawing.Size(355, 382);
                    break;
                case 2:
```

```
                    Size = new System.Drawing.Size(355, 433);
                    break;
                default:
                    break;
            }
        }

        private void Form1_KeyPress(object sender, KeyPressEventArgs e) {
            switch (tabControl.SelectedIndex) {
                case 0: {
                    if ((e.KeyChar >= '0' && e.KeyChar <= '9') || (e.KeyChar >= 'A' && e.KeyChar <= 'F') |
| (e.KeyChar == '.' && PNumberMode))
                        tB_PNumber.Text = pNumberController.ExecCommandEditor(CharToEditorCommand(e.KeyCha
r));
                    else if (OPERATIONS.Contains(e.KeyChar))
                        tB_PNumber.Text = NumberBeatifier(TAG_PNUMBER, pNumberController.ExecOperation(Cha
rToOperationsCommand<TPNumber>(e.KeyChar)));
                    break;
                }
                case 1: {
                    if ((e.KeyChar >= '0' && e.KeyChar <= '9') || e.KeyChar == '.')
                        tB_Frac.Text = fracController.ExecCommandEditor(CharToEditorCommand(e.KeyChar));
                    else if (OPERATIONS.Contains(e.KeyChar))
                        tB_Frac.Text = NumberBeatifier(TAG_FRAC, fracController.ExecOperation(CharToOperat
ionsCommand<TFrac>(e.KeyChar)));
                    break;
                }
                case 2: {
                    if ((e.KeyChar >= '0' && e.KeyChar <= '9') || e.KeyChar == '.')
                        tB_Complex.Text = complexController.ExecCommandEditor(CharToEditorCommand(e.KeyCha
r));
                    else if (OPERATIONS.Contains(e.KeyChar))
                        tB_Complex.Text = NumberBeatifier(TAG_COMPLEX, complexController.ExecOperation(Cha
rToOperationsCommand<TComplex>(e.KeyChar)));
                    break;
                }
                default:
                    break;
            }
        }

        private void Form1_KeyDown(object sender, KeyEventArgs e) {
            switch (tabControl.SelectedIndex) {
                case 0: {
                    if (e.KeyCode == Keys.Enter)
                        b_PNumber_Eval.PerformClick();
                    else {
                        AEditor.Command comm = KeyCodeToEditorCommand(e.KeyCode);
                        if (comm != AEditor.Command.cNone)
                            tB_PNumber.Text = pNumberController.ExecCommandEditor(comm);
```

```
            }
            break;
        }
        case 1: {
            if (e.KeyCode == Keys.Enter)
                b_Frac_Eval.PerformClick();
            else {
                AEditor.Command comm = KeyCodeToEditorCommand(e.KeyCode);
                if (comm != AEditor.Command.cNone)
                    tB_Frac.Text = pNumberController.ExecCommandEditor(comm);
            }
            break;
        }
        case 2: {
            if (e.KeyCode == Keys.Enter)
                b_Complex_Eval.PerformClick();
            else {
                AEditor.Command comm = KeyCodeToEditorCommand(e.KeyCode);
                if (comm != AEditor.Command.cNone)
                    tB_Complex.Text = pNumberController.ExecCommandEditor(comm);
            }
            break;
        }
        default:
            break;
    }
}

private void дробьFracTSMI_Click(object sender, EventArgs e) {
    дробьFracTSMI.Checked = true;
    числоFracTSMI.Checked = false;
    FracMode = true;
}

private void числоFracTSMI_Click(object sender, EventArgs e) {
    дробьFracTSMI.Checked = false;
    числоFracTSMI.Checked = true;
    FracMode = false;
}

private void комплексноеComplexTSMI_Click(object sender, EventArgs e) {
    комплексноеComplexTSMI.Checked = true;
    действительноеComplexTSMI.Checked = false;
    ComplexMode = true;
}

private void действительноеComplexTSMI_Click(object sender, EventArgs e) {
    комплексноеComplexTSMI.Checked = false;
    действительноеComplexTSMI.Checked = true;
    ComplexMode = false;
```

```
        }
    }
}
```

TCtrl.cs:

```
namespace Calculator {
    public sealed class TCtrl<T, Editor>
        where T : ANumber, new()
        where Editor : AEditor, new() {
        public enum TCtrlState {
            cStart, cEditing, FunDone, cOperDone, cExpDone, cOpChange, cError
        }

        Editor edit;
        TProc<T> proc;
        TMemory<T> memory;
        TCtrlState curState;

        public TCtrlState CurState { get => curState; set => curState = value; }
        public TProc<T> Proc { get => proc; set => proc = value; }
        public TMemory<T> Memory { get => memory; set => memory = value; }
        public Editor Edit { get => edit; set => edit = value; }

        public TCtrl() {
            Edit = new Editor();
            Proc = new TProc<T>();
            Memory = new TMemory<T>();
            curState = TCtrlState.cStart;
        }

        public string ExecCommandEditor(AEditor.Command command) {
            string ToReturn;
            if (CurState == TCtrlState.cExpDone) {
                Proc.Reset();
                CurState = TCtrlState.cStart;
            }
            if (CurState != TCtrlState.cStart)
                CurState = TCtrlState.cEditing;
            ToReturn = Edit.Edit(command);
            T TempObj = new T();
            if (TempObj is TPNumber) {
                dynamic a = TempObj;
                dynamic b = Edit;
                a.Notation = new TNumber(b.Notation);
                TempObj = a;
            }
            TempObj.SetString(ToReturn);
            Proc.Rop = TempObj;
            return ToReturn;
        }
```

```
public string ExecOperation(TProc<T>.Oper oper) {
    if (oper == TProc<T>.Oper.None)
        return Edit.Number;
    string ToReturn;
    try {
        switch (CurState) {
            case TCtrlState.cStart:
                Proc.Lop_Res = Proc.Rop;
                Proc.Operation = oper;
                CurState = TCtrlState.cOperDone;
                Edit.Clear();
                break;
            case TCtrlState.cEditing:
                Proc.DoOper();
                Proc.Operation = oper;
                Edit.Clear();
                CurState = TCtrlState.cOperDone;
                break;
            case TCtrlState.FunDone:
                if (Proc.Operation == TProc<T>.Oper.None)
                    Proc.Lop_Res = Proc.Rop;
                else
                    Proc.DoOper();
                Proc.Operation = oper;
                Edit.Clear();
                CurState = TCtrlState.cOpChange;
                break;
            case TCtrlState.cOperDone:
                CurState = TCtrlState.cOpChange;
                Edit.Clear();
                break;
            case TCtrlState.cExpDone:
                Proc.Operation = oper;
                Proc.Rop = Proc.Lop_Res;
                CurState = TCtrlState.cOpChange;
                Edit.Clear();
                break;
            case TCtrlState.cError:
                Proc.Reset();
                return "ERR";
            case TCtrlState.cOpChange:
                Proc.Operation = oper;
                Edit.Clear();
                break;
            default:
                break;
        }
        ToReturn = Proc.Lop_Res.ToString();
    }
    catch {
```

```
        Reset();
        return "ERROR";
    }
    return ToReturn;
}

public string ExecFunction(TProc<T>.Func func) {
    string ToReturn;
    try {
        if (CurState == TCtrlState.cExpDone) {
            Proc.Rop = Proc.Lop_Res;
            Proc.Operation = TProc<T>.Oper.None;
        }
        Proc.DoFunc(func);
        CurState = TCtrlState.FunDone;
        ToReturn = Proc.Rop.ToString();
    }
    catch {
        Reset();
        return "ERROR";
    }
    return ToReturn;
}

public string Calculate() {
    string ToReturn;
    try {
        if (CurState == TCtrlState.cStart)
            Proc.Lop_Res = Proc.Rop;
        Proc.DoOper();
        CurState = TCtrlState.cExpDone;
        ToReturn = Proc.Lop_Res.ToString();
    }
    catch {
        Reset();
        return "ERROR";
    }
    return ToReturn;
}

public string Reset() {
    Edit.Clear();
    Proc.Reset();
    Memory.Clear();
    curState = TCtrlState.cStart;
    return Edit.ToString();
}

public (T, TMemory<T>.NumStates) ExecCommandMemory(TMemory<T>.Commands command, string str) {
    T TempObj = new T();
```

```
            TempObj.SetString(str);
            (T, TMemory<T>.NumStates) obj = (null, TMemory<T>.NumStates.OFF);
            try {
                obj = Memory.Edit(command, TempObj);
            }
            catch {
                Reset();
                return obj;
            }
            if (command == TMemory<T>.Commands.Copy) {
                Edit.Number = obj.Item1.ToString();
                Proc.Rop = obj.Item1;
            }
            return obj;
        }
    }
}
```

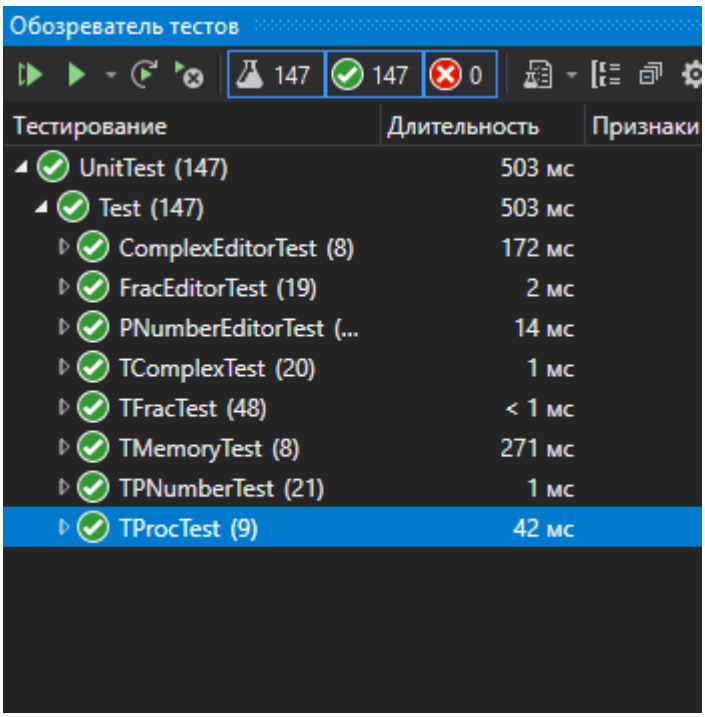
TMemory.cs:

```
namespace Calculator {
    public sealed class TMemory<T> where T : ANumber, new() {
        public enum NumStates {
            OFF, ON
        }

        public enum Commands {
            Store, Add, Clear, Copy
        }

        T fNumber;
        NumStates fState;
        public T FNumber {
            get { fState = NumStates.ON; return fNumber; }
            set { fNumber = value; fState = NumStates.ON; }
        }
        public NumStates FState {
            get => fState;
            set => fState = value;
        }

        public TMemory() {
            FNumber = new T();
            FState = NumStates.OFF;
        }

        public TMemory(T number) {
            FNumber = number;
            FState = NumStates.OFF;
        }

        public T Add(T number) {
```

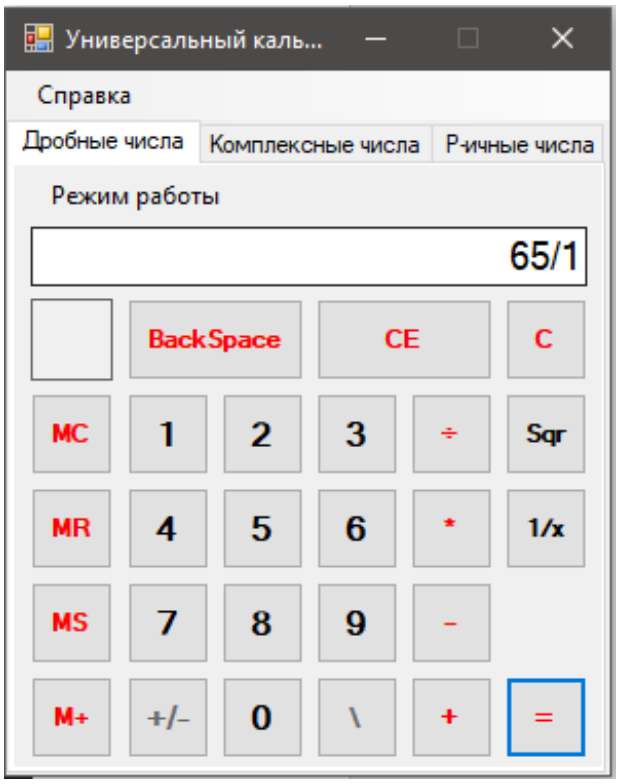
```
            FState = NumStates.ON;
            dynamic a = fNumber;
            dynamic b = number;
            fNumber = (T)(a + b);
            return fNumber;
        }

        public void Clear() {
            fNumber = new T();
            FState = NumStates.OFF;
        }

        public (T, NumStates) Edit(Commands command, T newNumber) {
            switch (command) {
                case Commands.Store:
                    FState = NumStates.ON;
                    fNumber = newNumber;
                    break;
                case Commands.Add:
                    FState = NumStates.ON;
                    dynamic a = fNumber;
                    dynamic b = newNumber;
                    fNumber = (T)(a + b);
                    break;
                case Commands.Clear:
                    Clear();
                    break;
            }
            return (fNumber, fState);
        }
    }
}
```

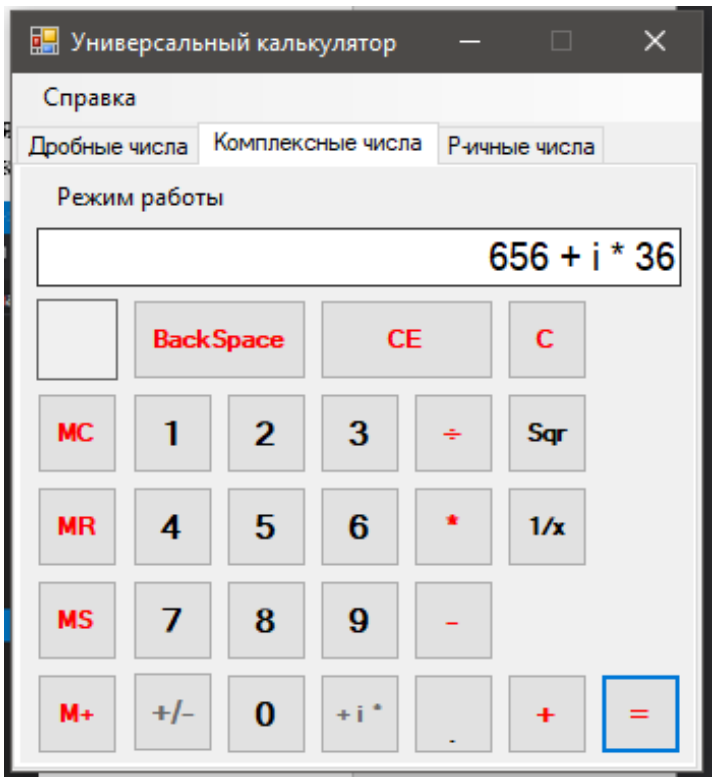
3. Результаты тестирования

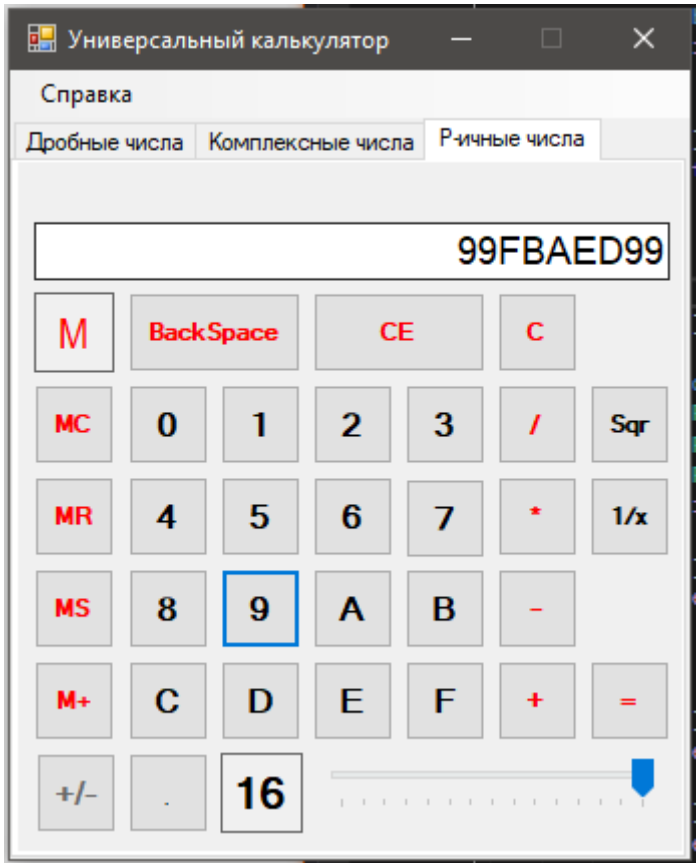
Сначала были успешно выполнены все юнит-тесты классов:



Затем программа была проверена с интерфейсом:




Универсальный калькулятор
Справка
Дробные числа
Комплексные числа
Р-ичные числа
Режим работы
656 + i * 36
BackSpace
CE
C
MC
1
2
3
÷
Sqr
MR
4
5
6
*
1/x
MS
7
8
9
-
M+
+/-
0
+ i *
.
+
=


Универсальный калькулятор
Справка
Дробные числа
Комплексные числа
Р-ичные числа
99FBAED99
M
BackSpace
CE
C
MC
0
1
2
3
/
Sqr
MR
4
5
6
7
*
1/x
MS
8
9
A
B
-
M+
C
D
E
F
+
=
+/-
.
16