Министерство цифрового развития, связи и массовых коммуникаций Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования «Сибирский государственный университет телекоммуникаций и информатики» (СибГУТИ)

Практическая работа

«Приложение Универсальный калькулятор.

Выполнил: студент 4 курса ИВТ,

гр. ИП-813

Лесковец Д.М.

Проверил: ассистент кафедры ПМиК

Агалаков А.А.

Новосибирск, 2022 г

Цель

Целями данного практикума является формирование практических навыков:

-проектирования программ в технологии «объектно-ориентированного программирования»;

-реализации объектно-ориентированных проектов с помощью классов С++;

-использования библиотеки визуальных компонентов для построения интерфейса.

Результатом выполнения предлагаемых в практикуме лабораторных работ

станет приложение под Windows «Калькулятор универсальный»

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.

Спецификация

1.Калькулятор должен обеспечить вычисление выражений в одном из режимов:

• р-ичные числа,

• комплексные числа,

• рациональными дроби.

2. Остальные требования: работа с памятью, работа с буфером обмена, прецеденты использования те же, что и для калькуляторов р-ичных чисел, простых дробей и комплексных чисел [1].

Рекомендации к выполнению

Реализуйте универсальный калькулятор, используя результаты предыдущих лабораторных работ [1]. Разработку выполните в следующем порядке:

1. Разработайте иерархию классов TANumber (число), в основу которой положите абстрактный класс TANumber, от него должны будут наследовать действительные числа TPNumber(р-ичные числа с основанием 2..16), простые дроби TFrac и комплексные числа TComp; в классе TANumber объявите общие для всех приведённых типов чисел операции, как абстрактные методы;

2. Разработайте абстрактный тип данных TMemory (память), которая могла бы хранить и обрабатывать действительные числа TPNumber(ричные числа с основанием 2..16), простые дроби TFrac и комплексные 5 числа TComp; для этого тип хранимого числа определите как TANumber;

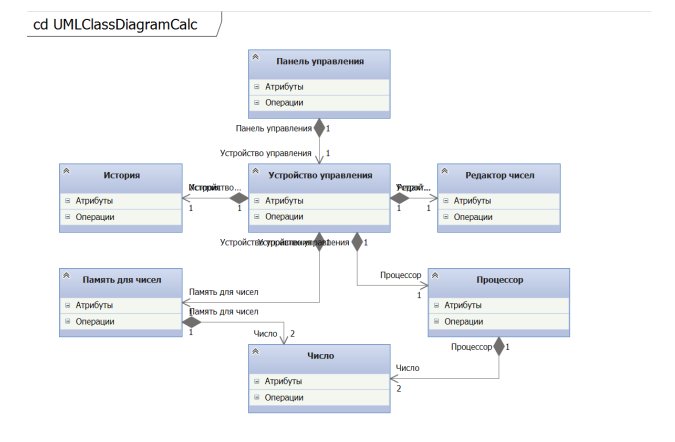
3. Разработайте абстрактный тип данных TProc (процессор), который мог бы обрабатывать действительные числа TPNumber(р-ичные числа с основанием 2..16), простые дроби TFrac и комплексные числа TComp; для этого тип обрабатываемого числа определите как TANumber;

4. Разработайте иерархию классов TAEditor (редактор), в основу которой положите абстрактный класс TAEditor, от него должны будут наследовать редакторы действительных чисел TREditor(р-ичные числа с основанием 2..16), простых дробей TFEditor и комплексных чисел TCEditor;

5. Разработайте класс TCtrl (управление калькулятором);

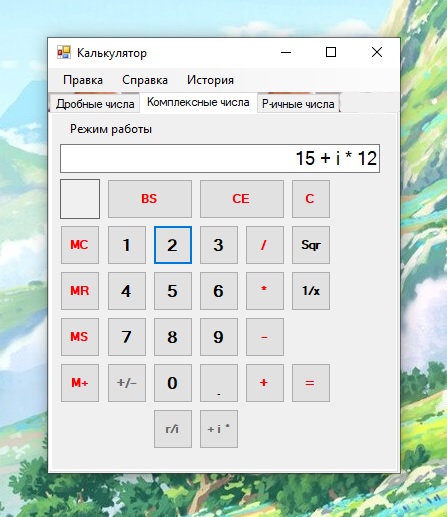
6. Разработайте TClcPnl (класс интерфейс калькулятора).

7. Выполните тестирование приложения «универсальный калькулятор». Диаграмма классов UML для калькулятора представлена на рисунке.

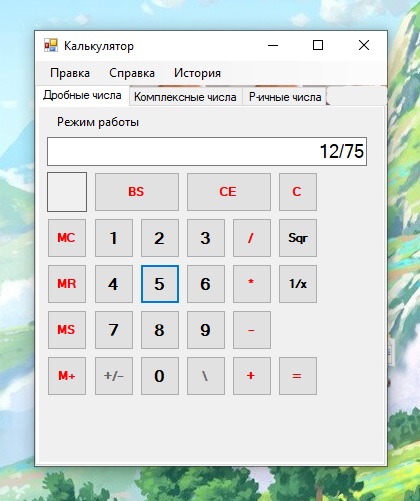


Реализация

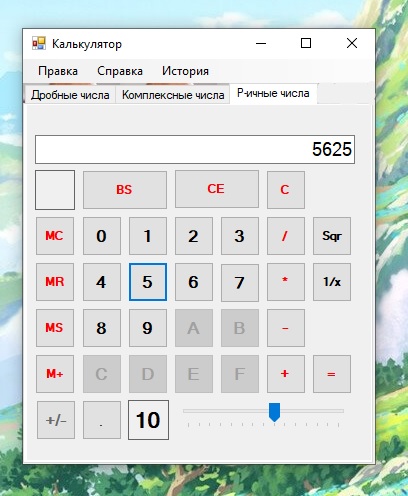
Комплексные числа:



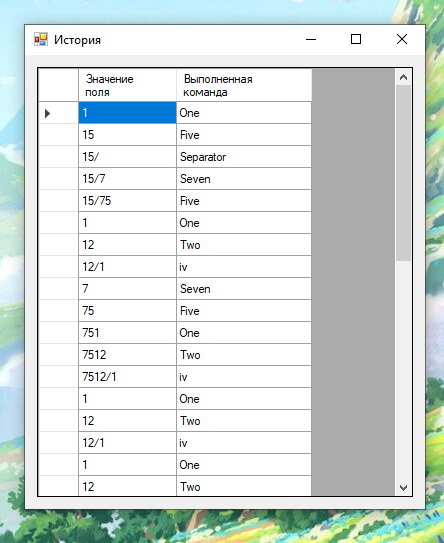
Дробные числа:



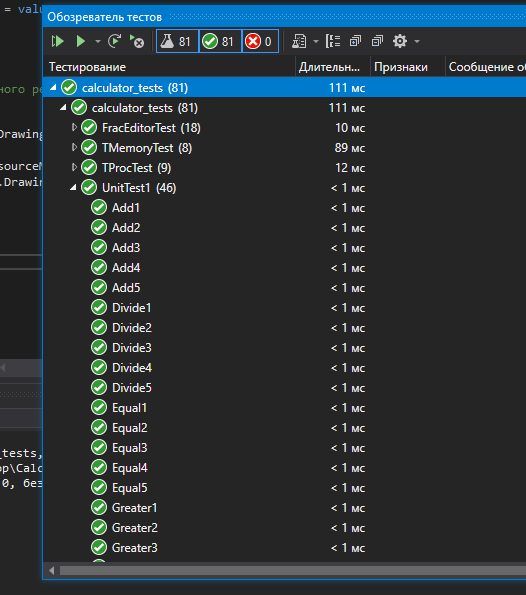
Р-ичные числа:



Окно Истории:



Результат работы тестов



Код программы

ADT\_Control.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace calculator

{

public class ADT\_Control<T, TEditor>

where T : ANumber, new()

where TEditor : AEditor, new()

{

public enum ADT\_Control\_State { cStart, cEditing, FunDone, cValDone, cExpDone, cOpDone, cOpChange, cError }

ADT\_Control\_State calcState;

TEditor editor;

ADT\_Proc<T> proc;

TMemory<T> memory;

public THistory history = new THistory();

public ADT\_Control\_State CurState

{

get

{

return calcState;

}

set

{

calcState = value;

}

}

public ADT\_Proc<T> Proc

{

get

{

return proc;

}

set

{

proc = value;

}

}

public TMemory<T> Memory

{

get

{

return memory;

}

set

{

memory = value;

}

}

public TEditor Edit

{

get

{

return editor;

}

set

{

editor = value;

}

}

public ADT\_Control()

{

Edit = new TEditor();

Proc = new ADT\_Proc<T>();

Memory = new TMemory<T>();

CurState = ADT\_Control\_State.cStart;

}

public string Reset()

{

Edit.Clear();

Proc.ResetProc();

Memory.Clear();

CurState = ADT\_Control\_State.cStart;

return Edit.ToString();

}

public string ExecCommandEditor(AEditor.Command command)

{

string toReturn;

if (CurState == ADT\_Control\_State.cExpDone)

{

Proc.ResetProc();

CurState = ADT\_Control\_State.cStart;

}

if (CurState != ADT\_Control\_State.cStart)

CurState = ADT\_Control\_State.cEditing;

toReturn = Edit.Edit(command);

T tmp = new T();

if (tmp is TPNumber)

{

dynamic a = tmp;

dynamic b = Edit;

a.Notation = b.Notation;

tmp = a;

}

tmp.SetString(toReturn);

proc.Right\_operand = tmp;

history.AddRecord(toReturn, command.ToString());

return toReturn;

}

public string ExecOperation(ADT\_Proc<T>.Operations operation)

{

if (operation == ADT\_Proc<T>.Operations.None)

return Edit.Number;

string toReturn;

try

{

switch (CurState)

{

case ADT\_Control\_State.cStart:

Proc.Left\_Result\_operand = Proc.Right\_operand;

Proc.Operation = operation;

CurState = ADT\_Control\_State.cOpDone;

Edit.Clear();

break;

case ADT\_Control\_State.cEditing:

Proc.DoOperation();

Proc.Operation = operation;

Edit.Clear();

CurState = ADT\_Control\_State.cOpDone;

break;

case ADT\_Control\_State.FunDone:

if (Proc.Operation == 0)

Proc.Left\_Result\_operand = Proc.Right\_operand;

else

Proc.DoOperation();

Proc.Operation = operation;

Edit.Clear();

CurState = ADT\_Control\_State.cOpChange;

Proc.Right\_operand = Proc.Left\_Result\_operand;

break;

case ADT\_Control\_State.cOpDone:

CurState = ADT\_Control\_State.cOpChange;

Edit.Clear();

break;

case ADT\_Control\_State.cValDone:

break;

case ADT\_Control\_State.cExpDone:

Proc.Operation = operation;

Proc.Right\_operand = Proc.Left\_Result\_operand;

CurState = ADT\_Control\_State.cOpChange;

Edit.Clear();

break;

case ADT\_Control\_State.cOpChange:

Proc.Operation = operation;

Edit.Clear();

break;

case ADT\_Control\_State.cError:

Proc.ResetProc();

return "ERR";

}

toReturn = Proc.Left\_Result\_operand.ToString();

}

catch

{

Reset();

return "ERROR";

}

history.AddRecord(toReturn, operation.ToString());

return toReturn;

}

public string ExecFunction(ADT\_Proc<T>.Functions function)

{

string toReturn;

try

{

if (CurState == ADT\_Control\_State.cExpDone)

{

Proc.Right\_operand = Proc.Left\_Result\_operand;

Proc.Operation = ADT\_Proc<T>.Operations.None;

}

Proc.DoFunction(function);

CurState = ADT\_Control\_State.FunDone;

toReturn = Proc.Right\_operand.ToString();

}

catch

{

Reset();

return "ERROR";

}

history.AddRecord(toReturn, function.ToString());

return toReturn;

}

public string Calculate()

{

string ToReturn;

try

{

if (CurState == ADT\_Control\_State.cStart)

Proc.Left\_Result\_operand = Proc.Right\_operand;

Proc.DoOperation();

CurState = ADT\_Control\_State.cExpDone;

// Edit.SetEditor(Proc.Left\_Result\_operand);

ToReturn = Proc.Left\_Result\_operand.ToString();

}

catch

{

Reset();

return "ERROR";

}

return ToReturn;

}

public (T, bool) ExecCommandMemory(TMemory<T>.Commands command, string str)

{

T TempObj = new T();

TempObj.SetString(str);

(T, bool) obj = (null, false);

try

{

obj = Memory.Edit(command, TempObj);

}

catch

{

Reset();

return obj;

}

if (command == TMemory<T>.Commands.Copy)

{

Edit.Number = obj.Item1.ToString();

Proc.Right\_operand = obj.Item1;

}

return obj;

}

}

}

ADT\_Proc.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace calculator

{

public class ADT\_Proc<T> where T : ANumber, new()

{

public enum Operations

{

None, Add, Sub, Mul, Div

}

public enum Functions

{

Rev, Sqr

}

T left\_result\_operand;

T right\_operand;

Operations operation;

public T Left\_Result\_operand

{

get

{

return left\_result\_operand;

}

set

{

left\_result\_operand = value;

}

}

public T Right\_operand

{

get

{

return right\_operand;

}

set

{

right\_operand = value;

}

}

public Operations Operation

{

get

{

return operation;

}

set

{

operation = value;

}

}

public ADT\_Proc()

{

operation = Operations.None;

left\_result\_operand = new T();

right\_operand = new T();

}

public ADT\_Proc(T leftObj, T rightObj)

{

operation = Operations.None;

left\_result\_operand = leftObj;

right\_operand = rightObj;

}

public void ResetProc()

{

operation = Operations.None;

T newObj = new T();

left\_result\_operand = right\_operand = newObj;

}

public void DoOperation()

{

try

{

dynamic a = left\_result\_operand;

dynamic b = right\_operand;

switch (operation)

{

case Operations.Add:

left\_result\_operand = a.Add(b);

break;

case Operations.Sub:

left\_result\_operand = a.Sub(b);

break;

case Operations.Mul:

left\_result\_operand = a.Mul(b);

break;

case Operations.Div:

left\_result\_operand = a.Div(b);

break;

default:

left\_result\_operand = right\_operand;

break;

}

}

catch

{

throw new System.OverflowException();

}

}

public void DoFunction(Functions function)

{

dynamic a = right\_operand;

switch (function)

{

case Functions.Rev:

a = a.Reverse();

right\_operand = (T)a;

break;

case Functions.Sqr:

a = a.Square();

right\_operand = (T)a;

break;

default:

break;

}

}

}

}

AEditor.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace calculator

{

public abstract class AEditor

{

public enum Command

{

cZero, cOne, cTwo, cThree, cFour, cFive, cSix, cSeven, cEight, cNine,

cA, cB, cC, cD, cE, cF,

cSign, cSeparator, cNumbSeparator,

cBS, cCE,

cToggleComplexMode,

cNone

}

public abstract string Number

{

get;

set;

}

//public abstract bool SetEditor(ANumber number);

public abstract string AddNumber(int num);

public abstract string ToggleMinus();

public abstract string AddSeparator();

public abstract string AddZero();

public abstract bool IsZero();

public abstract string RemoveSymbol();

public abstract string Clear();

public abstract string Edit(Command command);

}

}

ANumber.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace calculator

{

public abstract class ANumber

{

public abstract void SetString(string str);

}

}

Program.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace calculator

{

static class Program

{

/// <summary>

/// Главная точка входа для приложения.

/// </summary>

[STAThread]

static void Main()

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

Application.Run(new Form1());

}

}

}

TComplex.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Text.RegularExpressions;

using System.Threading.Tasks;

namespace calculator

{

public class TComplex : ANumber

{

private double real;

private double imaginary;

const string Separator = " + i \* ";

public double Real

{

get

{

return real;

}

set

{

real = value;

}

}

public double Imaginary

{

get

{

return imaginary;

}

set

{

imaginary = value;

}

}

public TComplex()

{

real = 0;

imaginary = 0;

}

public TComplex(int a, int b)

{

real = a;

imaginary = b;

}

public TComplex(double a, double b)

{

real = a;

imaginary = b;

}

public TComplex(TComplex complex)

{

real = complex.real;

imaginary = complex.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 = Double.Parse(Parts[0]);

imaginary = Double.Parse(Parts[1]);

}

else if (LeftPart.IsMatch(str))

{

if (str.Contains(Separator))

str = str.Replace(Separator, string.Empty);

real = Double.Parse(str);

imaginary = 0;

}

else

{

real = 0;

imaginary = 0;

}

}

public TComplex Copy()

{

return (TComplex)this.MemberwiseClone();

}

public TComplex Add(TComplex b)

{

TComplex res = this.Copy();

res.real += b.real;

res.imaginary += b.imaginary;

return res;

}

public TComplex Sub(TComplex b)

{

TComplex res = this.Copy();

res.real -= b.real;

res.imaginary -= b.imaginary;

return res;

}

public TComplex Mul(TComplex b)

{

TComplex res = this.Copy();

res.real = this.real \* b.real - this.imaginary \* b.imaginary;

res.imaginary = this.real \* b.imaginary + this.imaginary \* b.real;

return res;

}

public TComplex Div(TComplex b)

{

TComplex res = this.Copy();

res.real = (this.real \* b.real + this.imaginary \* b.imaginary) / (b.real \* b.real + b.imaginary \* b.imaginary);

res.imaginary = (b.real \* this.imaginary - this.real \* b.imaginary) / (b.real \* b.real + b.imaginary \* b.imaginary);

return res;

}

public TComplex Square()

{

TComplex res = this.Copy();

res.real = this.real \* this.real - this.imaginary \* this.imaginary;

res.imaginary = this.real \* this.imaginary + this.real \* this.imaginary;

return res;

}

public TComplex Reverse()

{

TComplex res = this.Copy();

res.real = this.real / (this.real \* this.real + this.imaginary \* this.imaginary);

res.imaginary = -this.imaginary / (this.real \* this.real + this.imaginary \* this.imaginary);

return res;

}

public TComplex Minus()

{

TComplex res = this.Copy();

res.real = 0 - res.real;

res.imaginary = 0 - res.imaginary;

return res;

}

public double Abs()

{

return Math.Sqrt(this.real \* this.real + this.imaginary \* this.imaginary);

}

public double Rad()

{

if (this.real > 0)

return Math.Atan(this.imaginary / this.real);

if (this.real == 0 && this.imaginary > 0)

return (Math.PI / 2);

if (this.real < 0)

return (Math.Atan(this.imaginary / this.real) + Math.PI);

if (this.real == 0 && this.imaginary < 0)

return (-Math.PI / 2);

return 0;

}

public double Degree()

{

return Rad() \* 180 / Math.PI;

}

public TComplex Pow(int n)

{

TComplex res = this.Copy();

res.real = Math.Pow(Abs(), n) \* Math.Cos(n \* Rad());

res.imaginary = Math.Pow(Abs(), n) \* Math.Sin(n \* Rad());

return res;

}

public TComplex Sqrt(int powN, int rootI)

{

if (powN == 0)

{

TComplex res0 = this.Copy();

res0.real = 1;

res0.imaginary = 0;

return res0;

}

if (rootI == 0)

return new TComplex();

TComplex new1 = Pow(powN);

TComplex res = this.Copy();

res.real = Math.Pow(new1.Abs(), 1 / (double)rootI) \* Math.Cos((new1.Rad() + 2 \* Math.PI \* rootI) / rootI);

res.imaginary = Math.Pow(new1.Abs(), 1 / (double)rootI) \* Math.Sin((new1.Rad() + 2 \* Math.PI \* rootI) / rootI);

return res;

}

public bool Equal(TComplex anClass)

{

return (this.real == anClass.real && this.imaginary == anClass.imaginary);

}

public bool NotEqual(TComplex anClass)

{

return (this.real != anClass.real || this.imaginary != anClass.imaginary);

}

public double GetRealNumber()

{

return this.real;

}

public double GetImaginaryNumber()

{

return this.imaginary;

}

public string GetRealString()

{

return this.real.ToString();

}

public string GetImaginaryString()

{

return this.imaginary.ToString();

}

public override void SetString(string str) {

TComplex temp = new TComplex(str);

Real = temp.Real;

Imaginary = temp.Imaginary;

}

public override string ToString()

{

return GetRealString() + " + i \* " + GetImaginaryString();

}

}

}

TComplexEditor.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text.RegularExpressions;

namespace calculator

{

public class TComplexEditor : AEditor

{

public enum PartToEdit

{

Real, Imag

};

string complex\_num;

PartToEdit mode;

Regex ZeroComplex = new Regex(@"^-?(0+.?0\*)(\s\*\+\s\*i\s\*\\*\s\*-?(0+.?0\*)|(\s\*\+\s\*i\s\*\\*\s\*-?))?$");

const string Separator = " + i \* ";

public override string Number

{

get

{

return complex\_num;

}

set

{

complex\_num = new TComplex(value).ToString();

}

}

public TComplexEditor()

{

complex\_num = "0";

mode = PartToEdit.Real;

}

public TComplexEditor(int a, int b)

{

complex\_num = new TComplex(a, b).ToString();

mode = PartToEdit.Real;

}

public TComplexEditor(string str)

{

complex\_num = new TComplex(str).ToString();

mode = PartToEdit.Real;

}

public override bool IsZero()

{

/\*string tmp = pNumber;

if (tmp[0] == '-')

tmp = tmp.Substring(1);

tmp = tmp.Replace('-', '+');

if (tmp == zero)

return true;

else

return false;\*/

return ZeroComplex.IsMatch(complex\_num);

}

public override string ToggleMinus()

{

if (complex\_num.Contains(Separator))

{

if (mode == PartToEdit.Real)

{

if (complex\_num[0] == '-')

complex\_num = complex\_num.Substring(1);

else

complex\_num = '-' + complex\_num;

}

else

{

complex\_num = complex\_num.Substring(0, complex\_num.IndexOf(Separator)) + Separator + "-" +

complex\_num.Substring(complex\_num.IndexOf(Separator) + Separator.Length);

}

return complex\_num;

}

if (mode == PartToEdit.Imag)

ToggleMode();

if (complex\_num[0] == '-')

complex\_num = complex\_num.Substring(1);

else

complex\_num = '-' + complex\_num;

return complex\_num;

}

public PartToEdit ToggleMode()

{

if (mode == PartToEdit.Real)

mode = PartToEdit.Imag;

else

mode = PartToEdit.Real;

return mode;

}

public override string AddNumber(int a)

{

if (a < 0 || a > 9)

return complex\_num;

if (a == 0)

AddZero();

string left = "", right = "";

if (complex\_num.Contains(Separator))

{

left = complex\_num.Substring(0, complex\_num.IndexOf(Separator));

right = complex\_num.Substring(complex\_num.IndexOf(Separator) + +Separator.Length);

}

else

{

left = complex\_num;

}

if (mode == PartToEdit.Real)

{

if (left == "0" || left == "-0")

left = left.First() == '-' ? "-" + a.ToString() : a.ToString();

else left += a.ToString();

}

else

{

if (right == "0" || right == "-0")

right = right.First() == '-' ? "-" + a.ToString() : a.ToString();

else right += a.ToString();

}

if (right == "")

complex\_num = left;

else

complex\_num = left + Separator + right;

return complex\_num;

}

public override string AddZero()

{

if (complex\_num == "0" || complex\_num == "-0" || complex\_num.EndsWith(" 0") || complex\_num.EndsWith(" -0") || complex\_num.EndsWith(Separator))

return complex\_num;

complex\_num += "0";

return complex\_num;

}

public string AddNumberSeparator()

{

string left = "", right = "";

if (complex\_num.Contains(Separator))

{

left = complex\_num.Substring(0, complex\_num.IndexOf(Separator));

right = complex\_num.Substring(complex\_num.IndexOf(Separator) + Separator.Length);

}

else

{

left = complex\_num;

}

if (mode == PartToEdit.Real)

{

if (!left.Contains("."))

left += ".";

}

else

{

if (!right.Contains(".") && right.Length > 0)

right += ".";

}

if (right == "")

complex\_num = left;

else

complex\_num = left + Separator + right;

return complex\_num;

}

public override string AddSeparator()

{

if (!complex\_num.Contains(Separator))

{

complex\_num = complex\_num + Separator + "0";

mode = PartToEdit.Imag;

}

return complex\_num;

}

public override string RemoveSymbol()

{

string left = "", right = "";

if (complex\_num.Contains(Separator))

{

left = complex\_num.Substring(0, complex\_num.IndexOf(Separator));

right = complex\_num.Substring(complex\_num.IndexOf(Separator) + Separator.Length);

}

else

{

left = complex\_num;

}

if (mode == PartToEdit.Real)

{

if (left.Length == 1 || (left.Length == 2 && left[0] == '-'))

{

left = left[0] == '-' ? "-0" : "0";

}

else

{

left = left.Remove(left.Length - 1);

}

}

else

{

if (right.Length == 1 || (right.Length == 2 && right[0] == '-'))

{

right = right[0] == '-' ? "-0" : "0";

}

else

{

right = right.Remove(right.Length - 1);

}

}

if (right == "")

complex\_num = left;

else

complex\_num = left + Separator + right;

return complex\_num;

}

public override string Clear()

{

complex\_num = "0";

mode = PartToEdit.Real;

return complex\_num;

}

public override string ToString()

{

return complex\_num;

}

public override string Edit(Command command)

{

switch (command)

{

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:

ToggleMinus();

break;

case Command.cSeparator:

AddNumberSeparator();

break;

case Command.cBS:

RemoveSymbol();

break;

case Command.cCE:

Clear();

break;

case Command.cNumbSeparator:

AddSeparator();

break;

case Command.cToggleComplexMode:

ToggleMode();

break;

default:

break;

}

return complex\_num;

}

}

}

TFrac.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text.RegularExpressions;

namespace calculator

{

public class TFrac : ANumber

{

private long numerator;

private long denominator;

/// Числитель

public long Numerator

{

get

{

return numerator;

}

set

{

numerator = value;

}

}

/// Знаменатель

public long Denominator

{

get

{

return denominator;

}

set

{

denominator = value;

}

}

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;

}

public TFrac()

{

numerator = 0;

denominator = 1;

}

public TFrac(long a, long 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 = 0;

denominator = 1;

return;

}

numerator = a;

denominator = b;

long gcdRes = GCD(a, b);

if (gcdRes > 1)

{

numerator /= gcdRes;

denominator /= gcdRes;

}

}

public TFrac(string frac)

{

Regex FracRegex = new Regex(@"^-?(\d+)/(\d+)$");

Regex NumberRegex = new Regex(@"^-?\d+/?$");

if (FracRegex.IsMatch(frac))

{

List<string> FracSplited = frac.Split('/').ToList();

numerator = Convert.ToInt64(FracSplited[0]);

denominator = Convert.ToInt64(FracSplited[1]);

if (denominator == 0)

{

numerator = 0;

denominator = 1;

return;

}

long gcd = GCD(numerator, denominator);

if (gcd > 1)

{

numerator /= gcd;

denominator /= gcd;

}

return;

}

else if (NumberRegex.IsMatch(frac))

{

if (long.TryParse(frac, out long NewNumber))

numerator = NewNumber;

else

numerator = 0;

denominator = 1;

return;

}

else

{

numerator = 0;

denominator = 1;

return;

}

}

public TFrac Copy()

{

return (TFrac)this.MemberwiseClone();

}

public override void SetString(string str)

{

TFrac TempFrac = new TFrac(str);

numerator = TempFrac.numerator;

denominator = TempFrac.denominator;

}

public TFrac Add(TFrac a)

{

return new TFrac(numerator \* a.denominator + denominator \* a.numerator, denominator \* a.denominator);

}

public TFrac Mul(TFrac b)

{

return new TFrac(numerator \* b.numerator, denominator \* b.denominator);

}

public TFrac Sub(TFrac b)

{

return new TFrac(numerator \* b.denominator - denominator \* b.numerator, denominator \* b.denominator);

}

public TFrac Div(TFrac b)

{

return new TFrac(numerator \* b.denominator, denominator \* b.numerator);

}

public TFrac Square()

{

return new TFrac(numerator \* numerator, denominator \* denominator);

}

public TFrac Reverse()

{

return new TFrac(denominator, numerator);

}

public TFrac Minus()

{

return new TFrac(-numerator, denominator);

}

public bool Equal(TFrac b)

{

return numerator == b.numerator && denominator == b.denominator;

}

public static bool operator >(TFrac a, TFrac b)

{

return (Convert.ToDouble(a.numerator) / Convert.ToDouble(a.denominator)) > (Convert.ToDouble(b.numerator) / Convert.ToDouble(b.denominator));

}

public static bool operator <(TFrac a, TFrac b)

{

return (Convert.ToDouble(a.numerator) / Convert.ToDouble(a.denominator)) < (Convert.ToDouble(b.numerator) / Convert.ToDouble(b.denominator));

}

public static implicit operator string(TFrac v)

{

throw new NotImplementedException();

}

public long getNumeratorNum()

{

return numerator;

}

public long getDenominatorNum()

{

return denominator;

}

public string getNumeratorString()

{

return numerator.ToString();

}

public string getDenominatorString()

{

return denominator.ToString();

}

public override string ToString()

{

return getNumeratorString() + "/" + getDenominatorString();

}

}

}

TFracEditor.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace calculator

{

public class TFracEditor : AEditor

{

const string Separator = "/";

const string ZeroFraction = "0/";

const int max\_numerator\_length = 14;

const int max\_denominator\_length = 22;

private string fraction;

public override string Number

{

get

{

return fraction;

}

set

{

fraction = new TFrac(value).ToString();

}

}

public TFracEditor()

{

fraction = "0";

}

public TFracEditor(long a, long b)

{

fraction = new TFrac(a, b).ToString();

}

public TFracEditor(string frac)

{

fraction = new TFrac(frac).ToString();

}

public override bool IsZero()

{

return fraction.StartsWith(ZeroFraction) || fraction.StartsWith("-" + ZeroFraction) || fraction == "0" || fraction == "-0";

}

public override string ToggleMinus()

{

if (fraction[0] == '-')

fraction = fraction.Remove(0, 1);

else

fraction = '-' + fraction;

return fraction;

}

public override string AddNumber(int a)

{

if (!fraction.Contains(Separator) && fraction.Length > max\_numerator\_length)

return fraction;

else if (fraction.Length > max\_denominator\_length)

return fraction;

if (a < 0 || a > 9)

return fraction;

if (a == 0)

AddZero();

else if (IsZero())

fraction = fraction.First() == '-' ? "-" + a.ToString() : a.ToString();

else

fraction += a.ToString();

return fraction;

}

public override string AddZero()

{

if (IsZero())

return fraction;

if (fraction.Last().ToString() == Separator)

return fraction;

fraction += "0";

return fraction;

}

public override string RemoveSymbol()

{

if (fraction.Length == 1)

fraction = "0";

else if (fraction.Length == 2 && fraction.First() == '-')

fraction = "-0";

else

fraction = fraction.Remove(fraction.Length - 1);

return fraction;

}

public override string Clear()

{

fraction = "0";

return fraction;

}

public override string Edit(Command command)

{

switch (command)

{

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:

ToggleMinus();

break;

case Command.cSeparator:

AddSeparator();

break;

case Command.cBS:

RemoveSymbol();

break;

case Command.cCE:

Clear();

break;

default:

break;

}

return fraction;

}

/\* public void SetEditor(TFrac frac)

{

fraction = frac.ToString();

}\*/

public override string AddSeparator()

{

if (!fraction.Contains(Separator))

fraction += Separator;

return fraction;

}

public override string ToString()

{

return Number;

}

}

}

THistory.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace calculator

{

public class THistory

{

public struct Record

{

private string oper;

private string str\_res;

public Record(string operation, string str\_res)

{

str\_res = str\_res.Remove(0, 1);

this.oper = operation;

this.str\_res = str\_res;

}

public List<string> ToList()

{

return new List<string> { oper, str\_res };

}

}

List<Record> L;

public THistory()

{

L = new List<Record>();

}

public void AddRecord(string o, string record\_string)

{

Record record = new Record(o, record\_string);

L.Add(record);

}

public Record this[int i]

{

get

{

if (i < 0 || i >= L.Count)

throw new IndexOutOfRangeException();

return L[i];

}

set

{

if (i < 0 || i >= L.Count)

throw new IndexOutOfRangeException();

L[i] = value;

}

}

public void Clear()

{

L.Clear();

}

public int Count()

{

return L.Count();

}

}

}

TMemory.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace calculator

{

public class TMemory<T> where T : ANumber, new()

{

public enum Commands {

Store, Add, Clear, Copy

}

T number;

bool state;

public T FNumber

{

get

{

state = true;

return number;

}

set

{

number = value;

state = true;

}

}

public bool FState

{

get

{

return state;

}

set

{

state = value;

}

}

public TMemory()

{

number = new T();

state = false;

}

public TMemory(T num)

{

number = num;

state = false;

}

public T Add(T num)

{

state = true;

dynamic a = number;

dynamic b = num;

number = a.Add(b);

return number;

}

public void Clear()

{

number = new T();

state = false;

}

public (T, bool) Edit(Commands command, T newNumber) {

switch (command) {

case Commands.Store:

state = true;

number = newNumber;

break;

case Commands.Add:

dynamic a = number;

dynamic b = newNumber;

number = a.Add(b);

break;

case Commands.Clear:

Clear();

break;

}

return (number, state);

}

}

}

TPNumber.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace calculator

{

public class TPNumber : ANumber

{

public class ADT\_Convert\_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();

long leftSide = (long)n;

double rightSide = n - leftSide;

if (rightSide < 0)

rightSide \*= -1;

string leftSideString = Int\_to\_p(leftSide, p);

string rightSideString = Flt\_to\_p(rightSide, p, c);

return leftSideString + (rightSideString == String.Empty ? "" : ".") + rightSideString;

}

public static char Int\_to\_char(int d)

{

if (d > 15 || d < 0)

{

throw new IndexOutOfRangeException();

}

string allSymbols = "0123456789ABCDEF";

return allSymbols.ElementAt(d);

}

public static string Int\_to\_p(long n, int p)

{

if (p < 2 || p > 16)

throw new IndexOutOfRangeException();

if (n == 0)

return "0";

if (p == 10)

return n.ToString();

bool isNegative = false;

if (n < 0)

{

isNegative = true;

n \*= -1;

}

string buf = "";

while (n > 0)

{

buf += Int\_to\_char((int)n % p);

n /= p;

}

if (isNegative)

buf += "-";

char[] chs = buf.ToCharArray();

Array.Reverse(chs);

return new string(chs);

}

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 class ADT\_Convert\_p\_10

{

public static double Dval(string p\_num, int p)

{

if (p < 2 || p > 16)

throw new IndexOutOfRangeException();

double buf = 0d;

if (p\_num.Contains("."))

{

string[] lr = p\_num.Split('.');

if (lr[0].Length == 0)

throw new Exception();

char[] chs = lr[0].ToCharArray();

Array.Reverse(chs);

for (int i = 0; i < chs.Length; i++)

{

if (Char\_to\_num(chs[i]) > p)

throw new Exception();

buf += Char\_to\_num(chs[i]) \* Math.Pow(p, i);

}

char[] chsr = lr[1].ToCharArray();

for (int i = 0; i < chsr.Length; i++)

{

if (Char\_to\_num(chsr[i]) > p)

throw new Exception();

buf += Char\_to\_num(chsr[i]) \* Math.Pow(p, -(i + 1));

}

}

else

{

char[] chs = p\_num.ToCharArray();

Array.Reverse(chs);

for (int i = 0; i < chs.Length; i++)

{

if (Char\_to\_num(chs[i]) > p)

throw new Exception();

buf += Char\_to\_num(chs[i]) \* Math.Pow(p, i);

}

}

return buf;

}

public static double Char\_to\_num(char ch)

{

string allNums = "0123456789ABCDEF";

if (!allNums.Contains(ch))

throw new IndexOutOfRangeException();

return allNums.IndexOf(ch);

}

public static double Convert(string p\_num, int p, double weight)

{

return 0d;

}

}

public double Number;

public int Notation;

public int Precision;

public TPNumber()

{

Number = 0f;

Notation = 10;

Precision = 5;

}

public TPNumber(double num, int not, int pre)

{

if (not < 2 || not > 16 || pre < 0 || pre > 10)

{

Number = 0f;

Notation = 10;

Precision = 5;

}

else

{

Number = num;

Notation = not;

Precision = pre;

}

}

public TPNumber(TPNumber anotherTPNumber)

{

Number = anotherTPNumber.Number;

Notation = anotherTPNumber.Notation;

Precision = anotherTPNumber.Precision;

}

public TPNumber(string str, int not, int pre)

{

try

{

Number = ADT\_Convert\_p\_10.Dval(str, not);

Notation = not;

Precision = pre;

}

catch

{

throw new System.OverflowException();

}

}

public TPNumber Add(TPNumber a)

{

TPNumber tmp = a.Copy();

if (a.Notation != this.Notation || a.Precision != Precision)

{

tmp.Number = 0.0;

return tmp;

}

tmp.Number = Number + a.Number;

return tmp;

}

public TPNumber Mul(TPNumber a)

{

TPNumber tmp = a.Copy();

if (a.Notation != this.Notation || a.Precision != this.Precision)

{

tmp.Number = 0.0;

return tmp;

}

tmp.Number = Number \* a.Number;

return tmp;

}

public TPNumber Div(TPNumber a)

{

TPNumber tmp = a.Copy();

if (a.Notation != Notation || a.Precision != Precision)

{

tmp.Number = 0.0;

return tmp;

}

tmp.Number = Number / a.Number;

return tmp;

}

public TPNumber Sub(TPNumber a)

{

TPNumber tmp = a.Copy();

if (a.Notation != Notation || a.Precision != Precision)

{

tmp.Number = 0.0;

return tmp;

}

tmp.Number = Number - a.Number;

return tmp;

}

public object Square()

{

return new TPNumber(Number \* Number, Notation, Precision);

}

public object Reverse()

{

return new TPNumber(1 / Number, Notation, Precision);

}

public bool IsZero()

{

return Number == 0;

}

public TPNumber Copy()

{

return (TPNumber)this.MemberwiseClone();

}

public override string ToString()

{

string str;

try

{

str = ADT\_Convert\_10\_p.Do(Number, Notation, Precision);

}

catch

{

throw new System.OverflowException();

}

return str;

}

public override void SetString(string str)

{

try

{

Number = ADT\_Convert\_p\_10.Dval(str, Notation);

}

catch

{

throw new System.OverflowException();

}

}

private bool check(double a, int b, int c)

{

string a\_str = a.ToString();

if (!checkOnBase(a\_str, b))

{

return false;

}

if (!checkOnC(a\_str, c))

{

return false;

}

if (!checkOnSymbol(a\_str))

{

return false;

}

return true;

}

private bool check(string a, string b, string c)

{

int b\_int = Convert.ToInt32(b);

int c\_int = Convert.ToInt32(c);

if (!checkOnBase(a, b\_int))

{

return false;

}

if (!checkOnC(a, c\_int))

{

return false;

}

if (!checkOnSymbol(a))

{

return false;

}

return true;

}

private bool checkOnBase(string a, int b)

{

foreach (char iter in a)

{

int move = Math.Abs('A' - iter);

int iter\_int = iter - '0';

if (iter >= 'A' && iter <= 'Z')

{

iter\_int = 10 + move;

}

if (iter == ',')

{

continue;

}

if (iter\_int >= b)

{

return false;

}

}

return true;

}

private bool checkOnC(string a, int c)

{

if (checkPoint(a) && c > 0)

{

string[] spliter = a.Split(',');

if (spliter[1].Length == c)

{

return true;

}

}

return false;

}

private bool checkPoint(string n)

{

int i;

for (i = 0; i < n.Length && n[i] != ','; i++) { }

if (i < n.Length)

{

return true;

}

return false;

}

private bool checkOnSymbol(string a)

{

foreach (char iter in a)

{

if (iter >= 'a' && iter <= 'z')

{

return false;

}

}

return true;

}

}

}

TPNumberEditor.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Text.RegularExpressions;

using System.Threading.Tasks;

namespace calculator

{

public class TPNumberEditor : AEditor

{

private string pNumber;

public int Notation;

public int Precision;

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

const string Separator = ".";

public override string Number

{

get

{

return pNumber;

}

set

{

pNumber = new TComplex(value).ToString();

}

}

public TPNumberEditor()

{

pNumber = "0";

Notation = 10;

Precision = 5;

}

public TPNumberEditor(double num, int not, int pre)

{

if (not < 2 || not > 16 || pre < 0 || pre > 10)

{

pNumber = "0";

Notation = 10;

Precision = 5;

}

else

{

Notation = not;

Precision = pre;

pNumber = TPNumber.ADT\_Convert\_10\_p.Do(num, not, pre);

}

}

public override bool IsZero()

{

return ZeroPNumber.IsMatch(pNumber);

}

public override string ToggleMinus()

{

if (pNumber.ElementAt(0) == '-')

pNumber = pNumber.Remove(0, 1);

else

pNumber = "-" + pNumber;

return pNumber;

}

public override string AddNumber(int num)

{

if (num < 0 || num >= Notation)

return pNumber;

if (num == 0)

AddZero();

else if (pNumber == "0" || pNumber == "-0")

pNumber = pNumber.First() == '-' ? "-" + TPNumber.ADT\_Convert\_10\_p.Int\_to\_char(num).ToString() : TPNumber.ADT\_Convert\_10\_p.Int\_to\_char(num).ToString();

else

pNumber += TPNumber.ADT\_Convert\_10\_p.Int\_to\_char(num).ToString();

return pNumber;

}

public override string RemoveSymbol()

{

if (pNumber.Length == 1)

pNumber = "0";

else if (pNumber.Length == 2 && pNumber.First() == '-')

pNumber = "-0";

else

pNumber = pNumber.Remove(pNumber.Length - 1);

return pNumber;

}

public override string AddSeparator()

{

if (!pNumber.Contains(Separator))

pNumber += Separator;

return pNumber;

}

public override string AddZero()

{

if (pNumber.Contains(Separator) && pNumber.Last().ToString() == Separator)

return pNumber;

if (pNumber == "0" || pNumber == "0.")

return pNumber;

pNumber += "0";

return pNumber;

}

public override string ToString()

{

return pNumber;

}

public override string Clear()

{

pNumber = "0";

return pNumber;

}

public override string Edit(Command command)

{

switch (command)

{

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:

ToggleMinus();

break;

case Command.cSeparator:

AddSeparator();

break;

case Command.cBS:

RemoveSymbol();

break;

case Command.cCE:

Clear();

break;

default:

break;

}

return Number;

}

}

}

Tests.cs

using Microsoft.VisualStudio.TestTools.UnitTesting;

using calculator;

namespace calculator\_tests

{

[TestClass]

public class UnitTest1

{

[TestMethod]

public void InitString1()

{

string fracString = "1/2";

TFrac fracClass = new TFrac(fracString);

Assert.AreEqual(fracString, fracClass.ToString());

}

[TestMethod]

public void InitString2()

{

string fracString = "111/2";

TFrac fracClass = new TFrac(fracString);

Assert.AreEqual(fracString, fracClass.ToString());

}

[TestMethod]

public void InitString3()

{

string fracString = "-100/60";

TFrac fracClass = new TFrac(fracString);

string Expect = "-5/3";

Assert.AreEqual(Expect, fracClass.ToString());

}

[TestMethod]

public void InitString4()

{

string fracString = "00000003/000004";

TFrac fracClass = new TFrac(fracString);

string Expect = "3/4";

Assert.AreEqual(Expect, fracClass.ToString());

}

[TestMethod]

public void InitString5()

{

string fracString = "-00000003/000004";

TFrac fracClass = new TFrac(fracString);

string Expect = "-3/4";

Assert.AreEqual(Expect, fracClass.ToString());

}

[TestMethod]

public void InitNumber1()

{

TFrac fracClass = new TFrac(1, 2);

string Expect = "1/2";

Assert.AreEqual(Expect, fracClass.ToString());

}

[TestMethod]

public void InitNumber2()

{

TFrac fracClass = new TFrac(100, 100);

string Expect = "1/1";

Assert.AreEqual(Expect, fracClass.ToString());

}

[TestMethod]

public void InitNumber3()

{

TFrac fracClass = new TFrac(-100, -99);

string Expect = "100/99";

Assert.AreEqual(Expect, fracClass.ToString());

}

[TestMethod]

public void InitNumber4()

{

TFrac fracClass = new TFrac(0, 0);

string Expect = "0/1";

Assert.AreEqual(Expect, fracClass.ToString());

}

[TestMethod]

public void InitNumber5()

{

TFrac fracClass = new TFrac(50, -5);

string fracCompar = "-10/1";

Assert.AreEqual(fracCompar, fracClass.ToString());

}

[TestMethod]

public void Add1()

{

TFrac fracClass1 = new TFrac(1, 4);

TFrac fracClass2 = new TFrac(-3, 4);

fracClass2 = fracClass1.Add(fracClass2);

string answer = "-1/2";

Assert.AreEqual(answer, fracClass2.ToString());

}

[TestMethod]

public void Add2()

{

TFrac fracClass1 = new TFrac(-1, 2);

TFrac fracClass2 = new TFrac(-1, 2);

fracClass2 = fracClass1.Add(fracClass2);

string answer = "-1/1";

Assert.AreEqual(answer, fracClass2.ToString());

}

[TestMethod]

public void Add3()

{

TFrac fracClass1 = new TFrac(-6, 2);

TFrac fracClass2 = new TFrac(6, 2);

fracClass2 = fracClass1.Add(fracClass2);

string answer = "0/1";

Assert.AreEqual(answer, fracClass2.ToString());

}

[TestMethod]

public void Add4()

{

TFrac fracClass1 = new TFrac(50, 3);

TFrac fracClass2 = new TFrac(0, 1);

fracClass2 = fracClass1.Add(fracClass2);

string answer = "50/3";

Assert.AreEqual(answer, fracClass2.ToString());

}

[TestMethod]

public void Add5()

{

TFrac fracClass1 = new TFrac(0, 1);

TFrac fracClass2 = new TFrac(0, 1);

fracClass2 = fracClass1.Add(fracClass2);

string answer = "0/1";

Assert.AreEqual(answer, fracClass2.ToString());

}

[TestMethod]

public void Multiply1()

{

TFrac fracClass1 = new TFrac(-1, 2);

TFrac fracClass2 = new TFrac(-1, 2);

fracClass2 = fracClass1.Mul(fracClass2);

string answer = "1/4";

Assert.AreEqual(answer, fracClass2.ToString());

}

[TestMethod]

public void Multiply2()

{

TFrac fracClass1 = new TFrac(1, 6);

TFrac fracClass2 = new TFrac(0, 1);

fracClass2 = fracClass1.Mul(fracClass2);

string answer = "0/1";

Assert.AreEqual(answer, fracClass2.ToString());

}

[TestMethod]

public void Multiply3()

{

TFrac fracClass1 = new TFrac(1, 6);

TFrac fracClass2 = new TFrac(1, 6);

fracClass2 = fracClass1.Mul(fracClass2);

string answer = "1/36";

Assert.AreEqual(answer, fracClass2.ToString());

}

[TestMethod]

public void Multiply4()

{

TFrac fracClass1 = new TFrac(-1, 6);

TFrac fracClass2 = new TFrac(12, 1);

fracClass2 = fracClass1.Mul(fracClass2);

string answer = "-2/1";

Assert.AreEqual(answer, fracClass2.ToString());

}

[TestMethod]

public void Multiply5()

{

TFrac fracClass1 = new TFrac(-1, 6);

TFrac fracClass2 = new TFrac(12, 1);

fracClass2 = fracClass1.Mul(fracClass2);

string answer = "-2/1";

Assert.AreEqual(answer, fracClass2.ToString());

}

[TestMethod]

public void Substract1()

{

TFrac fracClass1 = new TFrac(0, 1);

TFrac fracClass2 = new TFrac(1, 1);

fracClass2 = fracClass1.Sub(fracClass2);

string answer = "-1/1";

Assert.AreEqual(answer, fracClass2.ToString());

}

[TestMethod]

public void Substract2()

{

TFrac fracClass1 = new TFrac(5, 1);

TFrac fracClass2 = new TFrac(1, 1);

fracClass2 = fracClass1.Sub(fracClass2);

string answer = "4/1";

Assert.AreEqual(answer, fracClass2.ToString());

}

[TestMethod]

public void Substract3()

{

TFrac fracClass1 = new TFrac(1, 2);

TFrac fracClass2 = new TFrac(1, 2);

fracClass2 = fracClass1.Sub(fracClass2);

string answer = "0/1";

Assert.AreEqual(answer, fracClass2.ToString());

}

[TestMethod]

public void Substract4()

{

TFrac fracClass1 = new TFrac(-1, 6);

TFrac fracClass2 = new TFrac(-1, 6);

fracClass2 = fracClass1.Sub(fracClass2);

string answer = "0/1";

Assert.AreEqual(answer, fracClass2.ToString());

}

[TestMethod]

public void Substract5()

{

TFrac fracClass1 = new TFrac(-1, 6);

TFrac fracClass2 = new TFrac(2, 6);

fracClass2 = fracClass1.Sub(fracClass2);

string answer = "-1/2";

Assert.AreEqual(answer, fracClass2.ToString());

}

[TestMethod]

public void Divide1()

{

TFrac fracClass1 = new TFrac(5, 6);

TFrac fracClass2 = new TFrac(1, 1);

fracClass2 = fracClass1.Div(fracClass2);

string answer = "5/6";

Assert.AreEqual(answer, fracClass2.ToString());

}

[TestMethod]

public void Divide2()

{

TFrac fracClass1 = new TFrac(1, 1);

TFrac fracClass2 = new TFrac(5, 6);

fracClass2 = fracClass1.Div(fracClass2);

string answer = "6/5";

Assert.AreEqual(answer, fracClass2.ToString());

}

[TestMethod]

public void Divide3()

{

TFrac fracClass1 = new TFrac(0, 1);

TFrac fracClass2 = new TFrac(5, 6);

fracClass2 = fracClass1.Div(fracClass2);

string answer = "0/1";

Assert.AreEqual(answer, fracClass2.ToString());

}

[TestMethod]

public void Divide4()

{

TFrac fracClass1 = new TFrac(2, 3);

TFrac fracClass2 = new TFrac(7, 4);

fracClass2 = fracClass1.Div(fracClass2);

string answer = "8/21";

Assert.AreEqual(answer, fracClass2.ToString());

}

[TestMethod]

public void Divide5()

{

TFrac fracClass1 = new TFrac(2, 3);

TFrac fracClass2 = new TFrac(2, 3);

fracClass2 = fracClass1.Div(fracClass2);

string answer = "1/1";

Assert.AreEqual(answer, fracClass2.ToString());

}

[TestMethod]

public void Reverse1()

{

TFrac fracClass = new TFrac(-2, 3);

fracClass = fracClass.Reverse() as TFrac;

string answer = "-3/2";

Assert.AreEqual(answer, fracClass.ToString());

}

[TestMethod]

public void Reverse2()

{

TFrac fracClass = new TFrac(0, 1);

fracClass = fracClass.Reverse() as TFrac;

string answer = "0/1";

Assert.AreEqual(answer, fracClass.ToString());

}

[TestMethod]

public void Reverse3()

{

TFrac fracClass = new TFrac(5, 6);

fracClass = fracClass.Reverse() as TFrac;

string answer = "6/5";

Assert.AreEqual(answer, fracClass.ToString());

}

[TestMethod]

public void Square1()

{

TFrac fracClass = new TFrac(2, 3);

fracClass = fracClass.Square() as TFrac;

string answer = "4/9";

Assert.AreEqual(answer, fracClass.ToString());

}

[TestMethod]

public void Square2()

{

TFrac fracClass = new TFrac(0, 1);

fracClass = fracClass.Square() as TFrac;

string answer = "0/1";

Assert.AreEqual(answer, fracClass.ToString());

}

[TestMethod]

public void Square3()

{

TFrac fracClass = new TFrac(-2, 3);

fracClass = fracClass.Square() as TFrac;

string answer = "4/9";

Assert.AreEqual(answer, fracClass.ToString());

}

[TestMethod]

public void Equal1()

{

TFrac fracClass1 = new TFrac(1, 3);

TFrac fracClass2 = new TFrac(1, 3);

Assert.IsTrue(fracClass1.Equal(fracClass2));

}

[TestMethod]

public void Equal2()

{

TFrac fracClass1 = new TFrac(0, 6);

TFrac fracClass2 = new TFrac(1, 6);

Assert.IsFalse(fracClass1.Equal(fracClass2));

}

[TestMethod]

public void Equal3()

{

TFrac fracClass1 = new TFrac(-1, 6);

TFrac fracClass2 = new TFrac(-1, 6);

Assert.IsTrue(fracClass1.Equal(fracClass2));

}

[TestMethod]

public void Equal4()

{

TFrac fracClass1 = new TFrac(-1, 7);

TFrac fracClass2 = new TFrac(1, 7);

Assert.IsFalse(fracClass1.Equal(fracClass2));

}

[TestMethod]

public void Equal5()

{

TFrac fracClass1 = new TFrac(1, 6);

TFrac fracClass2 = new TFrac(0, 1);

Assert.IsFalse(fracClass1.Equal(fracClass2));

}

[TestMethod]

public void Greater1()

{

TFrac fracClass1 = new TFrac(1, 6);

TFrac fracClass2 = new TFrac(0, 1);

Assert.IsTrue(fracClass1 > fracClass2);

}

[TestMethod]

public void Greater2()

{

TFrac fracClass1 = new TFrac(0, 1);

TFrac fracClass2 = new TFrac(0, 1);

Assert.IsFalse(fracClass1 > fracClass2);

}

[TestMethod]

public void Greater3()

{

TFrac fracClass1 = new TFrac(-1, 6);

TFrac fracClass2 = new TFrac(0, 1);

Assert.IsFalse(fracClass1 > fracClass2);

}

[TestMethod]

public void Greater4()

{

TFrac fracClass1 = new TFrac(17, 3);

TFrac fracClass2 = new TFrac(16, 3);

Assert.IsTrue(fracClass1 > fracClass2);

}

[TestMethod]

public void Greater5()

{

TFrac fracClass1 = new TFrac(-2, 3);

TFrac fracClass2 = new TFrac(-1, 3);

Assert.IsFalse(fracClass1 > fracClass2);

}

}

[TestClass]

public class FracEditorTest

{

[TestMethod]

public void TestInit1()

{

TFracEditor testClass = new TFracEditor();

string input = "3/4";

testClass.Number = input;

Assert.AreEqual(input, testClass.Number);

}

[TestMethod]

public void TestInit2()

{

TFracEditor testClass = new TFracEditor();

string input = "-16/3";

testClass.Number = input;

Assert.AreEqual(input, testClass.Number);

}

[TestMethod]

public void TestInit3()

{

TFracEditor testClass = new TFracEditor();

string input = "0/8";

testClass.Number = input;

string result = "0/1";

Assert.AreEqual(result, testClass.Number);

}

[TestMethod]

public void TestInit4()

{

TFracEditor testClass = new TFracEditor();

string input = "-17/4";

testClass.Number = input;

Assert.AreEqual(input, testClass.Number);

}

[TestMethod]

public void TestInit5()

{

TFracEditor testClass = new TFracEditor();

string input = "0/1";

testClass.Number = input;

Assert.AreEqual(input, testClass.Number);

}

[TestMethod]

public void TestInit6()

{

TFracEditor testClass = new TFracEditor();

string input = "666/6666";

testClass.Number = input;

string result = "111/1111";

Assert.AreEqual(result, testClass.Number);

}

[TestMethod]

public void TestInit7()

{

TFracEditor testClass = new TFracEditor();

string input = "aaaa";

testClass.Number = input;

string result = "0/1";

Assert.AreEqual(result, testClass.Number);

}

[TestMethod]

public void TestInit8()

{

TFracEditor testClass = new TFracEditor();

string input = "0/1";

testClass.Number = input;

Assert.AreEqual(input, testClass.Number);

}

[TestMethod]

public void TestInit10()

{

TFracEditor testClass = new TFracEditor();

string input = "16/000000";

testClass.Number = input;

string result = "0/1";

Assert.AreEqual(result, testClass.Number);

}

[TestMethod]

public void hasZero1()

{

TFracEditor testClass = new TFracEditor("14/3");

Assert.AreEqual(false, testClass.IsZero());

}

[TestMethod]

public void hasZero2()

{

TFracEditor testClass = new TFracEditor("16/00000");

Assert.AreEqual(true, testClass.IsZero());

}

[TestMethod]

public void ToogleMinus1()

{

TFracEditor testClass = new TFracEditor("14/3");

testClass.ToggleMinus();

string result = "-14/3";

Assert.AreEqual(result, testClass.ToString());

}

[TestMethod]

public void ToogleMinus2()

{

TFracEditor testClass = new TFracEditor("-14/3");

testClass.ToggleMinus();

string result = "14/3";

Assert.AreEqual(result, testClass.ToString());

}

[TestMethod]

public void AddDeleteTest1()

{

TFracEditor testClass = new TFracEditor("123/123");

testClass.AddNumber(0);

testClass.AddNumber(1);

testClass.AddNumber(3);

testClass.AddSeparator();

testClass.ToggleMinus();

string result = "-1/1013";

Assert.AreEqual(result, testClass.ToString());

}

[TestMethod]

public void AddDeleteTest2()

{

TFracEditor testClass = new TFracEditor(123, 123);

testClass.RemoveSymbol();

testClass.RemoveSymbol();

testClass.RemoveSymbol();

testClass.RemoveSymbol();

testClass.RemoveSymbol();

testClass.RemoveSymbol();

testClass.RemoveSymbol();

testClass.AddNumber(1);

testClass.AddNumber(2);

testClass.AddNumber(3);

testClass.AddNumber(4);

testClass.AddNumber(5);

testClass.AddSeparator();

testClass.AddNumber(1);

testClass.AddNumber(1);

testClass.AddNumber(1);

testClass.AddNumber(1);

string result = "12345/1111";

Assert.AreEqual(result, testClass.ToString());

}

[TestMethod]

public void AddDeleteTest3()

{

TFracEditor testClass = new TFracEditor(1234567, 12345678);

for (int i = 0; i < 100; ++i)

testClass.RemoveSymbol();

for (int i = 0; i < 100; ++i)

testClass.AddSeparator();

testClass.AddNumber(1);

testClass.AddNumber(1);

testClass.AddNumber(1);

testClass.AddNumber(1);

string result = "1111";

Assert.AreEqual(result, testClass.ToString());

}

[TestMethod]

public void AddDeleteTest4()

{

TFracEditor testClass = new TFracEditor("0/1");

for (int i = 0; i < 100; ++i)

testClass.AddNumber(i);

string result = "123456789";

Assert.AreEqual(result, testClass.ToString());

}

[TestMethod]

public void Clear()

{

TFracEditor testClass = new TFracEditor("2345678/345678");

testClass.Clear();

string result = "0";

Assert.AreEqual(result, testClass.ToString());

}

}

[TestClass]

public class TMemoryTest

{

[TestMethod]

public void InitAndOutput1()

{

TFrac frac = new TFrac(22, 33);

TMemory<TFrac> memory = new TMemory<TFrac>(frac);

string answer = "2/3";

Assert.AreEqual(answer, memory.FNumber.ToString());

}

[TestMethod]

public void InitAndOutput2()

{

TFrac frac = new TFrac();

TMemory<TFrac> memory = new TMemory<TFrac>(frac);

string answer = "0/1";

Assert.AreEqual(answer, memory.FNumber.ToString());

}

[TestMethod]

public void InitAndOutput3()

{

TFrac frac = new TFrac(-1, 5);

TMemory<TFrac> memory = new TMemory<TFrac>(frac);

string answer = "-1/5";

Assert.AreEqual(answer, memory.FNumber.ToString());

}

[TestMethod]

public void Sum1()

{

TFrac frac = new TFrac(-1, 5);

TMemory<TFrac> memory = new TMemory<TFrac>(frac);

TFrac summator = new TFrac(1, 2);

memory.Add(summator);

string answer = "3/10";

Assert.AreEqual(answer, memory.FNumber.ToString());

}

[TestMethod]

public void Sum2()

{

TFrac frac = new TFrac(8, 9);

TMemory<TFrac> memory = new TMemory<TFrac>(frac);

TFrac summator = new TFrac(-16, 3);

memory.Add(summator);

string answer = "-40/9";

Assert.AreEqual(answer, memory.FNumber.ToString());

}

[TestMethod]

public void TestFState1()

{

TFrac frac = new TFrac(8, 9);

TMemory<TFrac> memory = new TMemory<TFrac>(frac);

memory.Clear();

bool expected = false;

Assert.AreEqual(expected, memory.FState);

}

[TestMethod]

public void TestFState2()

{

TFrac frac = new TFrac(8, 9);

TMemory<TFrac> memory = new TMemory<TFrac>(frac);

bool expected = false;

Assert.AreEqual(expected, memory.FState);

}

[TestMethod]

public void TestFState3()

{

TFrac frac = new TFrac(8, 9);

TMemory<TFrac> memory = new TMemory<TFrac>(frac);

memory.Add(frac);

bool expected = true;

Assert.AreEqual(expected, memory.FState);

}

}

[TestClass]

public class TProcTest

{

[TestMethod]

public void Init1()

{

TFrac leftFrac = new TFrac();

TFrac rightFrac = new TFrac();

ADT\_Proc<TFrac> proc = new ADT\_Proc<TFrac>(leftFrac, rightFrac);

string answer = "0/1";

Assert.AreEqual(answer, proc.Left\_Result\_operand.ToString());

Assert.AreEqual(answer, proc.Right\_operand.ToString());

}

[TestMethod]

public void Init2()

{

TFrac leftFrac = new TFrac(11, 3);

TFrac rightFrac = new TFrac();

ADT\_Proc<TFrac> proc = new ADT\_Proc<TFrac>(leftFrac, rightFrac);

string answer = "11/3";

Assert.AreEqual(answer, proc.Left\_Result\_operand.ToString());

}

[TestMethod]

public void Init3()

{

TFrac leftFrac = new TFrac(16, 4);

TFrac rightFrac = new TFrac(17, 9);

ADT\_Proc<TFrac> proc = new ADT\_Proc<TFrac>(leftFrac, rightFrac);

string answer = "17/9";

Assert.AreEqual(answer, proc.Right\_operand.ToString());

}

[TestMethod]

public void Operation1()

{

TFrac leftFrac = new TFrac(1, 2);

TFrac rightFrac = new TFrac(1, 2);

ADT\_Proc<TFrac> proc = new ADT\_Proc<TFrac>(leftFrac, rightFrac);

proc.Operation = ADT\_Proc<TFrac>.Operations.Add;

proc.DoOperation();

string answer = "1/1";

Assert.AreEqual(answer, proc.Left\_Result\_operand.ToString());

}

[TestMethod]

public void Operation2()

{

TFrac leftFrac = new TFrac(3, 4);

TFrac rightFrac = new TFrac(5, 6);

ADT\_Proc<TFrac> proc = new ADT\_Proc<TFrac>(leftFrac, rightFrac);

proc.Operation = ADT\_Proc<TFrac>.Operations.Sub;

proc.DoOperation();

string answer = "-1/12";

Assert.AreEqual(answer, proc.Left\_Result\_operand.ToString());

}

[TestMethod]

public void Operation3()

{

TFrac leftFrac = new TFrac(12, 7);

TFrac rightFrac = new TFrac(5, 9);

ADT\_Proc<TFrac> proc = new ADT\_Proc<TFrac>(leftFrac, rightFrac);

proc.Operation = ADT\_Proc<TFrac>.Operations.Mul;

proc.DoOperation();

string answer = "20/21";

Assert.AreEqual(answer, proc.Left\_Result\_operand.ToString());

}

[TestMethod]

public void Operation4()

{

TFrac leftFrac = new TFrac(56, 7);

TFrac rightFrac = new TFrac(-22, 3);

ADT\_Proc<TFrac> proc = new ADT\_Proc<TFrac>(leftFrac, rightFrac);

proc.Operation = ADT\_Proc<TFrac>.Operations.Div;

proc.DoOperation();

string answer = "-12/11";

Assert.AreEqual(answer, proc.Left\_Result\_operand.ToString());

}

[TestMethod]

public void TestFState1()

{

TFrac leftFrac = new TFrac(56, 7);

TFrac rightFrac = new TFrac(-22, 3);

ADT\_Proc<TFrac> proc = new ADT\_Proc<TFrac>(leftFrac, rightFrac);

proc.DoFunction(ADT\_Proc<TFrac>.Functions.Rev);

string answer = "-3/22";

Assert.AreEqual(answer, proc.Right\_operand.ToString());

}

[TestMethod]

public void TestFState2()

{

TFrac leftFrac = new TFrac(56, 7);

TFrac rightFrac = new TFrac(-22, 3);

ADT\_Proc<TFrac> proc = new ADT\_Proc<TFrac>(leftFrac, rightFrac);

proc.DoFunction(ADT\_Proc<TFrac>.Functions.Sqr);

string answer = "484/9";

Assert.AreEqual(answer, proc.Right\_operand.ToString());

}

}

}

Вывод

В рамках проектной работы были проведены объектно-ориентированный анализ, проектирование и реализация приложения «Калькулятор». В процессе работы было изучено: отношения между классами: ассоциация, агрегация, зависимость, их реализация средствами языка программирования высокого уровня; этапы разработки приложений в технологии ООП; элементы технологии визуального программирования; диаграммы языка UML для документирования разработки. Также были закреплены знания по разработке тестов для методов класса в проекте.