Министерство цифрового развития, связи и массовых коммуникаций

Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования «Сибирский государственный университет

телекоммуникаций и информатики» (СибГУТИ)

Кафедра прикладной математики и кибернетики

**Лабораторная работа №5**

**«Абстрактный тип данных (ADT) p - ичное число»**

Выполнил:

Студент группы ИП-113

Шпилев Д. И.

Работу проверил:

старший преподаватель кафедры ПМиК

Агалаков А.А.

Новосибирск 2024 г.

**Содержание**

[1. Задание 3](#_Toc84522514)

[2. Исходный код программы 10](#_Toc84522515)

[2.1. Код программы 10](#_Toc84522516)

[2.2. Код тестов 17](#_Toc84522517)

[3. Результаты модульных тестов 24](#_Toc84522518)

[4. Вывод 26](#_Toc84522519)

1. **Задание**

Задание

1. Реализовать абстрактный тип данных «р-ичное число», используя класс, в

соответствии с приведенной ниже спецификацией.

2. Протестировать каждую операцию, определенную на типе данных по критерию С2, используя средства модульного тестирования Visual Studio.

3. Если необходимо, предусмотрите возбуждение исключительных ситуаций.

Спецификация типа данных «р-ичное число».

**ADT TPNumber**

**Данные**

Р-ичное число TPNumber - это действительное число (n) со знаком в системе

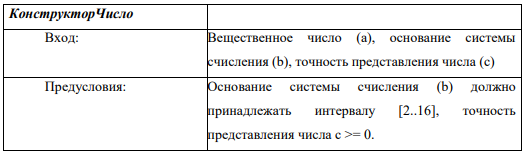
счисления с основанием (b) (в диапазоне 2..16), содержащее целую и дробную части.

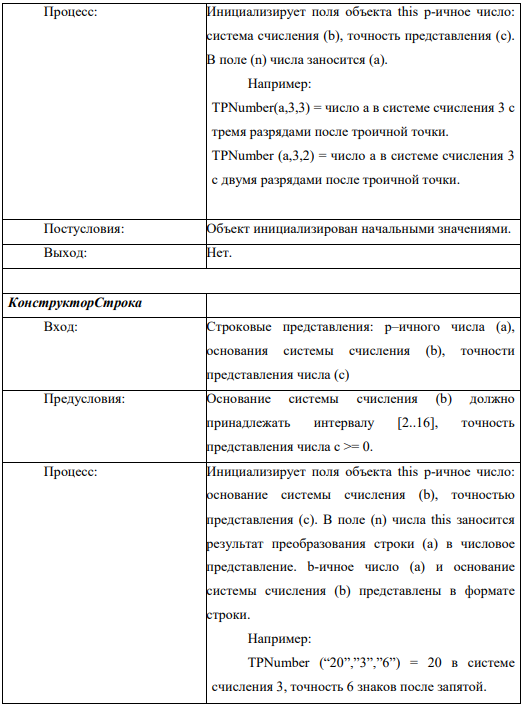
Точность представления числа – (c >= 0). Р-ичные числа неизменяемые.

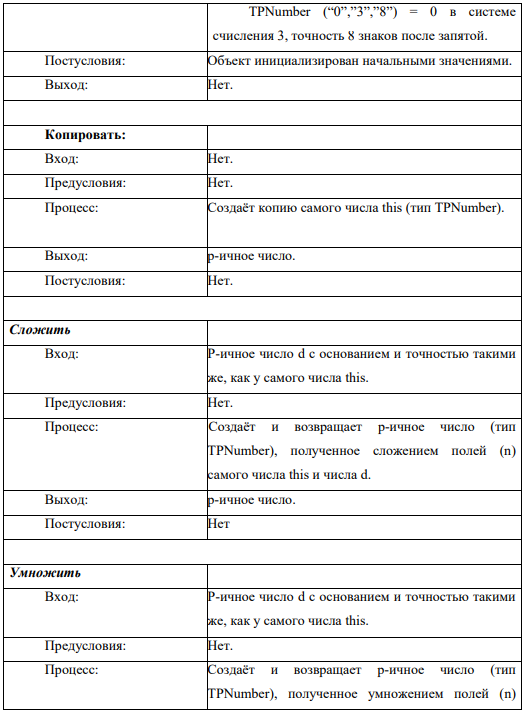
**Операции**

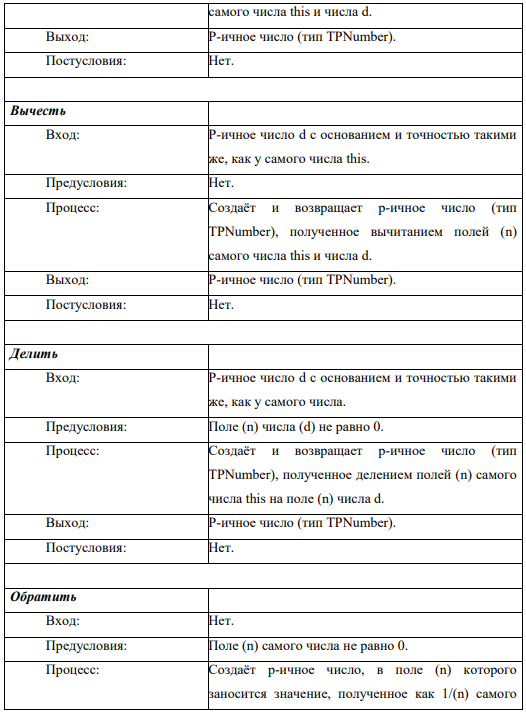
Операции могут вызываться только объектом р-ичное число (тип TPNumber),

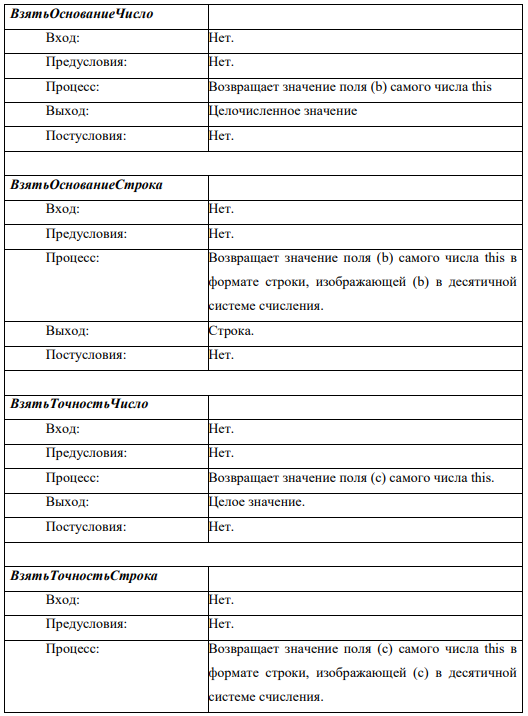
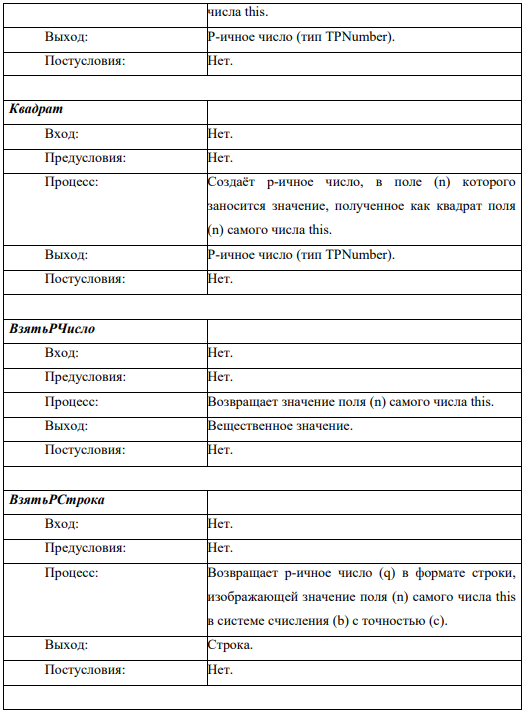
указатель на который в них передаётся по умолчанию. При описании операций этот объект называется this «само число».

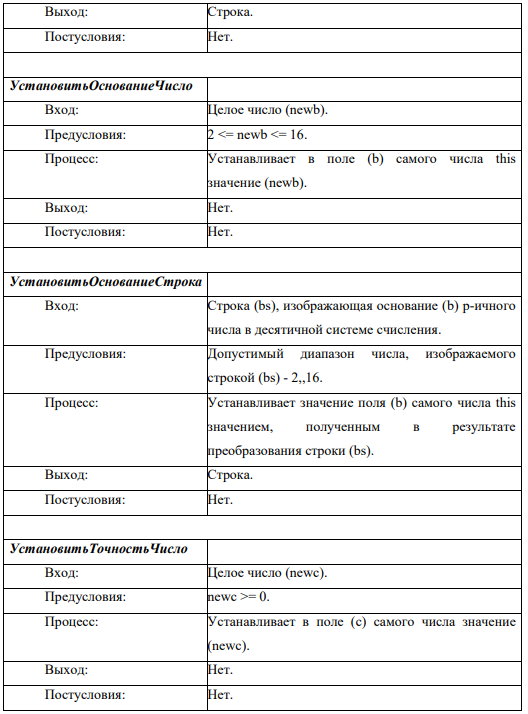


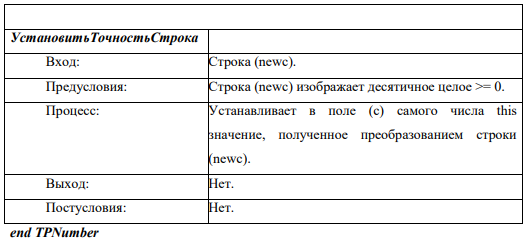












1. **Исходный код программы**
   1. **Код программы**

***TPNumber.h***

#pragma once

#include <string>

#include <stdexcept>

#include <cctype>

namespace STP {

class TPNumber

{

public:

TPNumber(double value, int base, int precision);

TPNumber(const std::string &value, const std::string &base, const std::string &precision);

virtual ~TPNumber() = 0;

virtual TPNumber\* operator+(const TPNumber& d) const = 0;

virtual TPNumber\* operator-(const TPNumber& d) const = 0;

virtual TPNumber\* operator\*(const TPNumber& d) const = 0;

virtual TPNumber\* operator/(const TPNumber& d) const = 0;

virtual TPNumber\* Invert() const = 0;

virtual TPNumber\* Square() const noexcept = 0;

void setBase(const std::string& base);

void setBase(const int& base);

void setPrecision(const std::string& precision);

void setPrecision(const int& precision);

double number() const noexcept { return number\_; }

std::string numberString() const noexcept { return numberString\_; }

int base() const noexcept { return base\_; }

std::string baseString() const noexcept { return std::to\_string(base\_); }

int precision() const noexcept { return precision\_; }

std::string precisionString() const noexcept { return std::to\_string(precision\_); }

protected:

std::string numberString\_;

double number\_;

int base\_;

int precision\_;

private:

std::string ConvertToBase(const double &value, const int &base, int precision) const noexcept;

double StringToDouble(const std::string &value, const int &base) const;

};

}

***TPNumber.cpp***

#include "TPNumber.h"

#include <stdexcept>

#include <cmath>

namespace STP {

TPNumber::TPNumber(double value, int base, int precision)

: base\_(base), precision\_(precision) {

if (base < 2 || base > 16) {

throw std::invalid\_argument("Base must be in the range [2..16]");

}

if (precision < 0) {

throw std::invalid\_argument("Precision must be non-negative");

}

number\_ = value;

numberString\_ = ConvertToBase(value, base, precision);

}

TPNumber::TPNumber(const std::string& value, const std::string& base, const std::string& precision)

{

int baseInt, precisionInt;

try {

baseInt = std::stoi(base);

precisionInt = std::stoi(precision);

}

catch (const std::invalid\_argument&) {

throw std::invalid\_argument("Invalid base or precision string");

}

if (baseInt < 2 || baseInt > 16) {

throw std::invalid\_argument("Base must be in the range [2..16]");

}

if (precisionInt < 0) {

throw std::invalid\_argument("Precision must be non-negative");

}

base\_ = baseInt;

precision\_ = precisionInt;

number\_ = StringToDouble(value, baseInt);

numberString\_ = value;

}

TPNumber::~TPNumber() = default;

std::string TPNumber::ConvertToBase(const double& value, const int& base, int precision) const noexcept

{

std::string result;

double absValue = std::fabs(value);

long long intPart = static\_cast<long long>(absValue);

double fracPart = absValue - intPart;

if (intPart == 0) {

result = "0";

}

else {

while (intPart > 0) {

short digit = intPart % base;

result.insert(0, 1, digit < 10 ? static\_cast<char>('0' + digit) : static\_cast<char>('A' + digit - 10));

intPart /= base;

}

}

if (precision > 0) {

result += ".";

while (precision-- > 0) {

fracPart \*= base;

short digit = static\_cast<long long>(fracPart);

fracPart -= digit;

result += (digit < 10 ? static\_cast<char>('0' + digit) : static\_cast<char>('A' + digit - 10));

}

}

if (value < 0) result.insert(0, 1, '-');

return result;

}

double TPNumber::StringToDouble(const std::string& value, const int& base) const

{

double result = 0.0;

bool isFraction = false;

double fractionMultiplier = 1.0;

bool isNegative = (value[0] == '-');

size\_t startIndex = (isNegative || value[0] == '+') ? 1 : 0;

for (size\_t i = startIndex; i < value.size(); ++i) {

char ch = value[i];

if (ch == '.') {

isFraction = true;

continue;

}

char upperCh = std::toupper(ch);

short digit;

if (upperCh >= '0' && upperCh <= '9') {

digit = upperCh - '0';

}

else if (upperCh >= 'A' && upperCh <= 'F') {

digit = upperCh - 'A' + 10;

}

else {

throw std::invalid\_argument("Invalid character in the number string");

}

if (digit >= base) {

throw std::invalid\_argument("Digit out of range for the specified base");

}

if (isFraction) {

fractionMultiplier /= base;

result += digit \* fractionMultiplier;

}

else {

result = result \* base + digit;

}

}

return isNegative ? -result : result;

}

void TPNumber::setBase(const std::string& base)

{

int newBase;

try {

newBase = std::stoi(base);

if (newBase < 2 || newBase > 16) {

throw std::out\_of\_range("Base must be between 2 and 16");

}

}

catch (const std::invalid\_argument&) {

throw std::invalid\_argument("Invalid argument: the string is not a valid integer");

}

catch (const std::out\_of\_range&) {

throw std::out\_of\_range("The integer value is out of range");

}

base\_ = newBase;

numberString\_ = ConvertToBase(number\_, base\_, precision\_);

}

void TPNumber::setBase(const int& base)

{

if (base < 2 || base > 16) {

throw std::out\_of\_range("Base must be between 2 and 16");

}

base\_ = base;

numberString\_ = ConvertToBase(number\_, base\_, precision\_);

}

void TPNumber::setPrecision(const std::string& precision)

{

int newPrecision;

try {

newPrecision = std::stoi(precision);

}

catch (const std::invalid\_argument&) {

throw std::invalid\_argument("Invalid argument: the string is not a valid integer");

}

if (newPrecision < 0) {

throw std::invalid\_argument("Precision must be non-negative");

}

precision\_ = newPrecision;

numberString\_ = ConvertToBase(number\_, base\_, precision\_);

}

void TPNumber::setPrecision(const int& precision)

{

if (precision < 0) {

throw std::invalid\_argument("Precision must be non-negative");

}

precision\_ = precision;

numberString\_ = ConvertToBase(number\_, base\_, precision\_);

}

}

***TPNumber.h***

#pragma once

#include "TPNumber.h"

namespace STP {

class PNumber :

public TPNumber

{

public:

PNumber(double value, int base, int precision)

: TPNumber(value, base, precision) {}

PNumber(const std::string& value, const std::string &base, const std::string &precision)

: TPNumber(value, base, precision) {}

virtual TPNumber\* operator+(const TPNumber& d) const override;

virtual TPNumber\* operator-(const TPNumber& d) const override;

virtual TPNumber\* operator\*(const TPNumber& d) const override;

virtual TPNumber\* operator/(const TPNumber& d) const override;

virtual PNumber operator+(const PNumber& d) const;

virtual PNumber operator-(const PNumber& d) const;

virtual PNumber operator\*(const PNumber& d) const;

virtual PNumber operator/(const PNumber& d) const;

virtual TPNumber\* Invert() const override;

virtual TPNumber\* Square() const noexcept override;

~PNumber() override = default;

};

}

***TPNumber.cpp***

#include "PNumber.h"

namespace STP {

TPNumber\* PNumber::operator+(const TPNumber& d) const

{

const PNumber\* pNumberD = dynamic\_cast<const PNumber\*>(&d);

if (pNumberD == nullptr) {

throw std::invalid\_argument("Operands must be of type PNumber");

}

if (base\_ != pNumberD->base\_ || precision\_ != pNumberD->precision\_) {

throw std::invalid\_argument("Bases and precisions must match");

}

double resultValue = number\_ + pNumberD->number\_;

return new PNumber(resultValue, base\_, precision\_);

}

TPNumber\* PNumber::operator-(const TPNumber& d) const

{

const PNumber\* pNumberD = dynamic\_cast<const PNumber\*>(&d);

if (pNumberD == nullptr) {

throw std::invalid\_argument("Operands must be of type PNumber");

}

if (base\_ != pNumberD->base\_ || precision\_ != pNumberD->precision\_) {

throw std::invalid\_argument("Bases and precisions must match");

}

double resultValue = number\_ - pNumberD->number\_;

return new PNumber(resultValue, base\_, precision\_);

}

TPNumber\* PNumber::operator\*(const TPNumber& d) const

{

const PNumber\* pNumberD = dynamic\_cast<const PNumber\*>(&d);

if (pNumberD == nullptr) {

throw std::invalid\_argument("Operands must be of type PNumber");

}

if (base\_ != pNumberD->base\_ || precision\_ != pNumberD->precision\_) {

throw std::invalid\_argument("Bases and precisions must match");

}

double resultValue = number\_ \* pNumberD->number\_;

return new PNumber(resultValue, base\_, precision\_);

}

TPNumber\* PNumber::operator/(const TPNumber& d) const

{

const PNumber\* pNumberD = dynamic\_cast<const PNumber\*>(&d);

if (pNumberD == nullptr) {

throw std::invalid\_argument("Operands must be of type PNumber");

}

if (base\_ != pNumberD->base\_ || precision\_ != pNumberD->precision\_) {

throw std::invalid\_argument("Bases and precisions must match");

}

if (pNumberD->number\_ == 0.0) {

throw std::invalid\_argument("Division by zero");

}

double resultValue = number\_ / pNumberD->number\_;

return new PNumber(resultValue, base\_, precision\_);

}

PNumber PNumber::operator+(const PNumber& d) const

{

if (base\_ != d.base\_ || precision\_ != d.precision\_) {

throw std::invalid\_argument("Bases and precisions must match");

}

double resultValue = number\_ + d.number\_;

return PNumber(resultValue, base\_, precision\_);

}

PNumber PNumber::operator-(const PNumber& d) const

{

if (base\_ != d.base\_ || precision\_ != d.precision\_) {

throw std::invalid\_argument("Bases and precisions must match");

}

double resultValue = number\_ - d.number\_;

return PNumber(resultValue, base\_, precision\_);

}

PNumber PNumber::operator\*(const PNumber& d) const

{

if (base\_ != d.base\_ || precision\_ != d.precision\_) {

throw std::invalid\_argument("Bases and precisions must match");

}

double resultValue = number\_ \* d.number\_;

return PNumber(resultValue, base\_, precision\_);

}

PNumber PNumber::operator/(const PNumber& d) const

{

if (base\_ != d.base\_ || precision\_ != d.precision\_) {

throw std::invalid\_argument("Bases and precisions must match");

}

if (d.number\_ == 0.0) {

throw std::invalid\_argument("Division by zero");

}

double resultValue = number\_ / d.number\_;

return PNumber(resultValue, base\_, precision\_);

}

TPNumber\* PNumber::Invert() const

{

if (number\_ == 0.0) {

throw std::invalid\_argument("Cannot invert zero");

}

double resultValue = 1.0 / number\_;

return new PNumber(resultValue, base\_, precision\_);

}

TPNumber\* PNumber::Square() const noexcept

{

double resultValue = number\_ \* number\_;

return new PNumber(resultValue, base\_, precision\_);

}

}

* 1. **Код тестов**

***TPNumberTest.cs***

#include "CppUnitTest.h"

#include "../Lab5/PNumber.h"

using namespace STP;

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace TPNumberTest

{

TEST\_CLASS(TPNumberTest)

{

public:

TEST\_METHOD(ConstructorDoubleBasePrecision)

{

PNumber num1(10.0, 10, 2);

Assert::AreEqual(10.0, num1.number());

Assert::AreEqual(10, num1.base());

Assert::AreEqual(2, num1.precision());

PNumber num2(234.153, 5, 8);

Assert::AreEqual(std::string("1414.03403030"), num2.numberString());

Assert::AreEqual(5, num2.base());

Assert::AreEqual(8, num2.precision());

}

TEST\_METHOD(ConstructorDoubleBasePrecisionExceptions)

{

Assert::ExpectException<std::invalid\_argument>([] {PNumber num1(10.0, 20, 2); });

Assert::ExpectException<std::invalid\_argument>([] {PNumber num1(10.0, 1, 2); });

Assert::ExpectException<std::invalid\_argument>([] {PNumber num1(10.0, 6, -1); });

}

TEST\_METHOD(ConstructorStringBasePrecisionExceptions)

{

Assert::ExpectException<std::invalid\_argument>([] {PNumber num1("10.0", "20", "2"); });

Assert::ExpectException<std::invalid\_argument>([] {PNumber num1("10.0", "1", "2"); });

Assert::ExpectException<std::invalid\_argument>([] {PNumber num1("10.0", "6", "-1"); });

Assert::ExpectException<std::invalid\_argument>([] {PNumber num1("1~0.0", "6", "1"); });

Assert::ExpectException<std::invalid\_argument>([] {PNumber num1("80.0", "6", "1"); });

Assert::ExpectException<std::invalid\_argument>([] {PNumber num1("10.0", "Z", "2"); });

}

TEST\_METHOD(ConstructorStringBasePrecision)

{

PNumber num2("-A.B9", "16", "4");

Assert::AreEqual(-10.7227, num2.number(), 0.0001);

Assert::AreEqual(16, num2.base());

Assert::AreEqual(4, num2.precision());

}

TEST\_METHOD(SetBaseFromInt)

{

PNumber num1(10.0, 10, 2);

num1.setBase(2);

Assert::AreEqual(2, num1.base());

}

TEST\_METHOD(SetBaseFromIntExceptions)

{

PNumber num1("10.0", "3", "2");

Assert::ExpectException<std::out\_of\_range>([&] { num1.setBase(20); });

Assert::ExpectException<std::out\_of\_range>([&] { num1.setBase(1); });

}

TEST\_METHOD(SetBaseFromString)

{

PNumber num1(10.0, 10, 2);

num1.setBase("8");

Assert::AreEqual(8, num1.base());

}

TEST\_METHOD(SetBaseFromStringExceptions)

{

PNumber num1("10.0", "3", "2");

Assert::ExpectException<std::out\_of\_range>([&] { num1.setBase("20"); });

Assert::ExpectException<std::out\_of\_range>([&] { num1.setBase("1"); });

Assert::ExpectException<std::invalid\_argument>([&] { num1.setBase("q`sdfa"); });

}

TEST\_METHOD(SetPrecisionFromInt)

{

PNumber num1(10.0, 10, 2);

num1.setPrecision(4);

Assert::AreEqual(4, num1.precision());

}

TEST\_METHOD(SetPrecisionFromIntExceptions)

{

PNumber num1("10.0", "3", "2");

Assert::ExpectException<std::invalid\_argument>([&] { num1.setPrecision(-1); });

}

TEST\_METHOD(SetPrecisionFromString)

{

PNumber num1(10.0, 10, 2);

num1.setPrecision("5");

Assert::AreEqual(5, num1.precision());

}

TEST\_METHOD(SetPrecisionFromStringExceptions)

{

PNumber num1("10.0", "3", "2");

Assert::ExpectException<std::invalid\_argument>([&] { num1.setPrecision("-1"); });

Assert::ExpectException<std::invalid\_argument>([&] { num1.setPrecision("`dsgfa"); });

}

TEST\_METHOD(Addition)

{

PNumber num1(10.0, 10, 2);

PNumber num4(20.0, 10, 2);

PNumber result = num1 + num4;

Assert::AreEqual(30.0, result.number());

Assert::AreEqual(10, result.base());

Assert::AreEqual(2, result.precision());

}

TEST\_METHOD(OperationsExceptinsBase)

{

PNumber num1(10.0, 10, 2);

PNumber num4(20.0, 12, 2);

Assert::ExpectException<std::invalid\_argument>([&] { PNumber result = num1 + num4; });

Assert::ExpectException<std::invalid\_argument>([&] { PNumber result = num1 - num4; });

Assert::ExpectException<std::invalid\_argument>([&] { PNumber result = num1 \* num4; });

Assert::ExpectException<std::invalid\_argument>([&] { PNumber result = num1 / num4; });

}

TEST\_METHOD(OperationsExceptinsPrecision)

{

PNumber num1(10.0, 10, 4);

PNumber num4(20.0, 10, 2);

Assert::ExpectException<std::invalid\_argument>([&] { PNumber result = num1 + num4; });

Assert::ExpectException<std::invalid\_argument>([&] { PNumber result = num1 - num4; });

Assert::ExpectException<std::invalid\_argument>([&] { PNumber result = num1 \* num4; });

Assert::ExpectException<std::invalid\_argument>([&] { PNumber result = num1 / num4; });

}

TEST\_METHOD(OperationsExceptinDivisionByZero)

{

PNumber num1(10.0, 10, 4);

PNumber num4(0.0, 10, 4);

Assert::ExpectException<std::invalid\_argument>([&] { PNumber result = num1 / num4; });

}

TEST\_METHOD(Subtraction)

{

PNumber num1(10.0, 10, 2);

PNumber num4(20.0, 10, 2);

PNumber result = num4 - num1;

Assert::AreEqual(10.0, result.number());

Assert::AreEqual(10, result.base());

Assert::AreEqual(2, result.precision());

}

TEST\_METHOD(Multiplication)

{

PNumber num1(10.0, 10, 2);

PNumber num4(20.0, 10, 2);

PNumber result = num1 \* num4;

Assert::AreEqual(200.0, result.number());

Assert::AreEqual(10, result.base());

Assert::AreEqual(2, result.precision());

}

TEST\_METHOD(Division)

{

PNumber num1(10.0, 10, 2);

PNumber num4(20.0, 10, 2);

PNumber result = num4 / num1;

Assert::AreEqual(2.0, result.number());

Assert::AreEqual(10, result.base());

Assert::AreEqual(2, result.precision());

}

TEST\_METHOD(AdditionRef)

{

TPNumber\* num1 = new PNumber(10.0, 10, 2);

TPNumber \*num4 = new PNumber(20.0, 10, 2);

TPNumber \*result = \*num1 + \*num4;

Assert::AreEqual(30.0, result->number());

Assert::AreEqual(10, result->base());

Assert::AreEqual(2, result->precision());

}

TEST\_METHOD(OperationsExceptinsBaseRef)

{

TPNumber\* num1 = new PNumber(10.0, 4, 2);

TPNumber\* num4 = new PNumber(20.0, 10, 2);

Assert::ExpectException<std::invalid\_argument>([&] { TPNumber \*result = \*num1 + \*num4; });

Assert::ExpectException<std::invalid\_argument>([&] { TPNumber \*result = \*num1 - \*num4; });

Assert::ExpectException<std::invalid\_argument>([&] { TPNumber \*result = \*num1 \* \*num4; });

Assert::ExpectException<std::invalid\_argument>([&] { TPNumber \*result = \*num1 / \*num4; });

}

TEST\_METHOD(OperationsExceptinsPrecisionRef)

{

TPNumber\* num1 = new PNumber(10.0, 10, 4);

TPNumber\* num4 = new PNumber(20.0, 10, 2);

Assert::ExpectException<std::invalid\_argument>([&] { TPNumber\* result = \*num1 + \*num4; });

Assert::ExpectException<std::invalid\_argument>([&] { TPNumber\* result = \*num1 - \*num4; });

Assert::ExpectException<std::invalid\_argument>([&] { TPNumber\* result = \*num1 \* \*num4; });

Assert::ExpectException<std::invalid\_argument>([&] { TPNumber\* result = \*num1 / \*num4; });

}

TEST\_METHOD(OperationsExceptinDivisionByZeroRef)

{

TPNumber\* num1 = new PNumber(10.0, 10, 4);

TPNumber\* num4 = new PNumber(0.0, 10, 4);

Assert::ExpectException<std::invalid\_argument>([&] { TPNumber\* result = \*num1 / \*num4; });

}

TEST\_METHOD(SubtractionRef)

{

TPNumber\* num1 = new PNumber(10.0, 10, 2);

TPNumber\* num4 = new PNumber(20.0, 10, 2);

TPNumber\* result = \*num1 - \*num4;

Assert::AreEqual(-10.0, result->number());

Assert::AreEqual(10, result->base());

Assert::AreEqual(2, result->precision());

}

TEST\_METHOD(MultiplicationRef)

{

TPNumber\* num1 = new PNumber(10.0, 10, 2);

TPNumber\* num4 = new PNumber(20.0, 10, 2);

TPNumber\* result = \*num1 \* \*num4;

Assert::AreEqual(200.0, result->number());

Assert::AreEqual(10, result->base());

Assert::AreEqual(2, result->precision());

}

TEST\_METHOD(DivisionRef)

{

TPNumber\* num1 = new PNumber(10.0, 10, 2);

TPNumber\* num4 = new PNumber(20.0, 10, 2);

TPNumber\* result = \*num4 / \*num1;

Assert::AreEqual(2.0, result->number());

Assert::AreEqual(10, result->base());

Assert::AreEqual(2, result->precision());

}

TEST\_METHOD(Invert)

{

TPNumber \*num1 = new PNumber(10.0, 10, 2);

TPNumber\* result = num1->Invert();

Assert::AreEqual(0.1, result->number());

Assert::AreEqual(10, result->base());

Assert::AreEqual(2, result->precision());

delete num1;

delete result;

}

TEST\_METHOD(InvertException)

{

TPNumber\* num1 = new PNumber(0.0, 10, 2);

Assert::ExpectException<std::invalid\_argument>([&] { TPNumber\* result = num1->Invert(); });

}

TEST\_METHOD(Square)

{

TPNumber\* num1 = new PNumber(10.0, 10, 2);

TPNumber\* result = num1->Square();

Assert::AreEqual(100.0, result->number());

Assert::AreEqual(10, result->base());

Assert::AreEqual(2, result->precision());

delete num1;

delete result;

}

TEST\_METHOD(DivisionByZero)

{

TPNumber\* num1 = new PNumber(10.0, 10, 2);

TPNumber\* num4 = new PNumber(0.0, 10, 2);

Assert::ExpectException<std::invalid\_argument>([&] {TPNumber\* result = \*num1 / \*num4; });

delete num1;

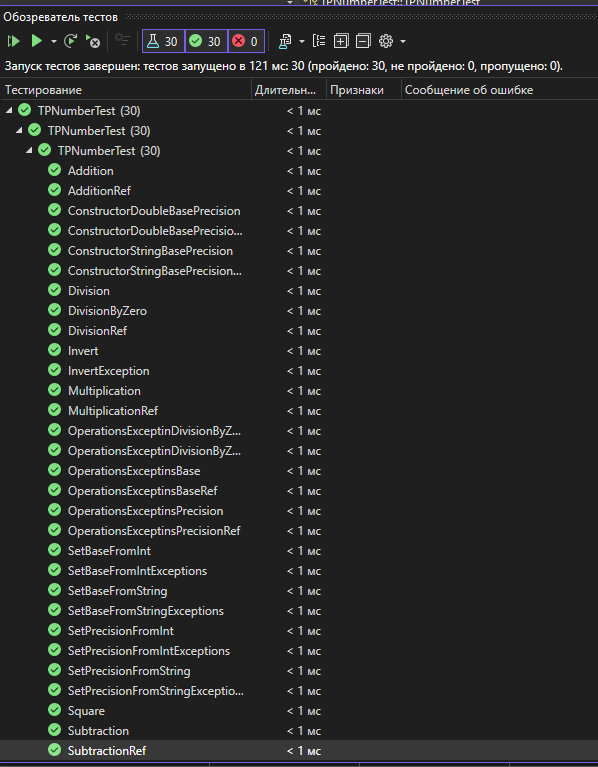
delete num4;

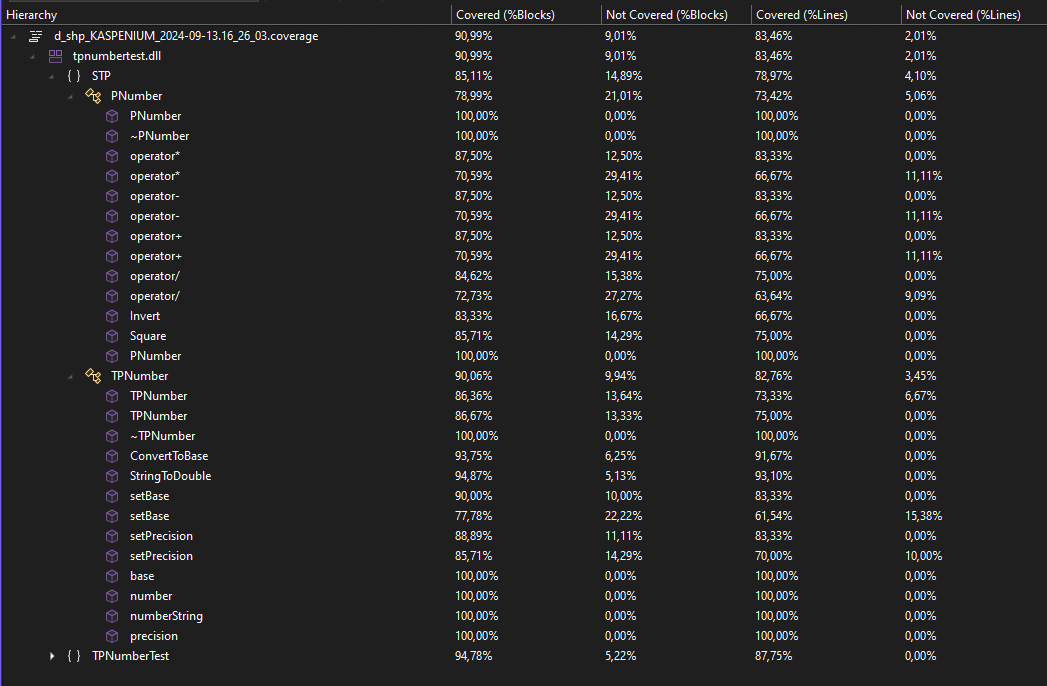
}

};

}

1. **Результаты модульных тестов**

****

****

1. **Вывод**

По итогам данной лабораторной работе были сформированы практические навыки разработки на С++ и модульного тестирования классов средствами Visual Studio.