

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  
«Шаблон класса процессор»

Выполнил:  
Студент группы ИП-113  
Шпилев Д. И.  
Работу проверил:  
старший преподаватель кафедры ПМиК  
Агалаков А.А.

Новосибирск 2024 г.

## Содержание

<b>1. Задание.....</b>	<b>3</b>
<b>1.1. Код программы .....</b>	<b>3</b>
<b>2.2 Результаты модульных тестов.....</b>	<b>5</b>
<b>3. Результаты модульных тестов .....</b>	<b>8</b>
<b>4. Вывод .....</b>	<b>8</b>

## 1. Задание

1. В соответствии с приведенной ниже спецификацией реализовать параметризованный абстрактный тип данных «Процессор», используя шаблон классов C++.
2. Протестировать каждую операцию, определенную на типе данных, используя средства модульного тестирования.
3. Если необходимо, предусмотрите возбуждение исключительных ситуаций. Спецификация типа данных «Процессор».

### ADT TProc

Данные Процессор (тип TProc) выполняет двухоперандные операции TOprtn = (None, Add, Sub, Mul, Div) и однооперандные операции - функции TFunc = (Rev, Sqr) над значениями типа T. Левый операнд и результат операции хранится в поле Lop\_Res, правый - в поле Rop. Оба поля имеют тип T. Процессор может находиться в состояниях: «операция установлена» - поле Operation не равно None (значение типа TOprtn) или в состоянии «операция не установлена» - поле Operation = None. Значения типа TProc - изменяемые. Они изменяются операциями: «Сброс операции» (OprtnClear), «Выполнить операцию» (OprtnRun), «Вычислить функцию» (FuncRun), «Установить операцию» (OprtnSet), «Установить левый операнд» (Lop\_Res\_Set), «Установить правый операнд» (Rop\_Set), «Сброс калькулятора» (ReSet). На значениях типа T должны быть определены указанные выше операции и функции.

### 1.1. Код программы

#### UProcsrc.h

```
#pragma once
#include "UANumber.h"

template <class T>
class TProc
{
public:
    enum Operation {
        None,
        Add,
        Sub,
        Mul,
        Div
    };

    enum Function {
        Sqr,
        Rev
    };

    TProc();
    void resetProc();
```

```

        void resetOper() { m_operation = None; }
        void doOperation();
        void doFunction(Function function);
        std::unique_ptr<TANumber> getLeftOperand() const noexcept { return
m_leftOperand->Clone(); }
        std::unique_ptr<TANumber> getRightOperand() const noexcept { return
m_rightOperand->Clone(); }
        void setLeftOperand(const std::unique_ptr<TANumber>& number);
        void setRightOperand(const std::unique_ptr<TANumber>& number);
        Operation getOperation() const noexcept { return m_operation; }
        void setOperation(Operation operation) noexcept { m_operation = operation; }

private:
        std::unique_ptr<TANumber> m_leftOperand;
        std::unique_ptr<TANumber> m_rightOperand;
        Operation m_operation;
};

template<class T>
inline TProc<T>::TProc()
{
    resetProc();
}

template<class T>
inline void TProc<T>::resetProc()
{
    m_leftOperand = std::make_unique<T>();
    m_rightOperand = std::make_unique<T>();
    m_operation = None;
}

template<class T>
inline void TProc<T>::doOperation()
{
    try {
        switch (m_operation)
        {
            case Add:
                m_leftOperand = *m_leftOperand + *m_rightOperand;
                break;
            case Sub:
                m_leftOperand = *m_leftOperand - *m_rightOperand;
                break;
            case Mul:
                m_leftOperand = *m_leftOperand * *m_rightOperand;
                break;
            case Div:
                m_leftOperand = *m_leftOperand / *m_rightOperand;
                break;
            default:
                break;
        }
    }
    catch (DivisionByZeroException& ex)
    {
    }
    catch (DifferentBaseOrPrecision& ex)
    {
    }
    //TODO: сделать нормальное перенаправление в ERROR
}

```

```

template<class T>
inline void TProc<T>::doFunction(Function function)
{
    try {
        switch (function)
        {
            case Sqr:
                m_rightOperand = m_rightOperand->Square();
                break;
            case Rev:
                m_rightOperand = m_rightOperand->Invert();
                break;
            default:
                break;
        }
    }
    catch (DivisionByZeroException& ex)
    {
    }
    catch (DifferentBaseOrPrecision& ex)
    {
    }
    //TODO: сделать нормальное перенаправление в ERROR
}

template<class T>
inline void TProc<T>::setLeftOperand(const std::unique_ptr<TANumber>& number)
{
    const T* pB = dynamic_cast<const T*>(number.get());
    if (!pB)
    {
        throw TypeMismatchException();
    }
    m_leftOperand = number->Clone();
}

template<class T>
inline void TProc<T>::setRightOperand(const std::unique_ptr<TANumber>& number)
{
    const T* pB = dynamic_cast<const T*>(number.get());
    if (!pB)
    {
        throw TypeMismatchException();
    }
    m_rightOperand = number->Clone();
}

```

## 2.2 Результаты модульных тестов

```

TEST_CLASS(TProcTests)
{
public:

    TEST_METHOD(TestResetProc)
    {
        TProc<TComplex> processor;

        processor.resetProc();

        Assert::IsTrue(processor.getLeftOperand() != nullptr);
    }
}

```

```

        Assert::IsTrue(processor.getRightOperand() != nullptr);
        Assert::AreEqual(static_cast<int>(TProc<TComplex>::None),
static_cast<int>(processor.getOperation()));
    }

    TEST_METHOD(TestSetLeftOperand)
    {
        TProc<TComplex> processor;
        std::unique_ptr<TANumber> complexNumber = std::make_unique<TComplex>(1, 1);

        processor.setLeftOperand(complexNumber);

        Assert::AreEqual(complexNumber->numberString(), processor.getLeftOperand()-
>numberString());
    }

    TEST_METHOD(TestSetRightOperand)
    {
        TProc<TComplex> processor;
        std::unique_ptr<TANumber> complexNumber = std::make_unique<TComplex>(2, 2);

        processor.setRightOperand(complexNumber);

        Assert::AreEqual(complexNumber->numberString(), processor.getRightOperand()-
>numberString());
    }

    TEST_METHOD(TestAdditionOperation)
    {
        TProc<TComplex> processor;
        std::unique_ptr<TANumber> leftNumber = std::make_unique<TComplex>(1, 1);
        std::unique_ptr<TANumber> rightNumber = std::make_unique<TComplex>(2, 2);
        processor.setLeftOperand(leftNumber);
        processor.setRightOperand(rightNumber);
        processor.setOperation(TProc<TComplex>::Add);

        processor.doOperation();

        Assert::AreEqual(TComplex(3, 3).numberString(), processor.getLeftOperand()-
>numberString());
    }

    TEST_METHOD(TestSubtractionOperation)
    {
        TProc<TComplex> processor;
        std::unique_ptr<TANumber> leftNumber = std::make_unique<TComplex>(5, 5);
        std::unique_ptr<TANumber> rightNumber = std::make_unique<TComplex>(2, 2);
        processor.setLeftOperand(leftNumber);
        processor.setRightOperand(rightNumber);
        processor.setOperation(TProc<TComplex>::Sub);

        processor.doOperation();

        Assert::AreEqual(TComplex(3, 3).numberString(), processor.getLeftOperand()-
>numberString());
    }

    TEST_METHOD(TestMultiplicationOperation)
    {
        TProc<TComplex> processor;
        std::unique_ptr<TANumber> leftNumber = std::make_unique<TComplex>(1, 1);
        std::unique_ptr<TANumber> rightNumber = std::make_unique<TComplex>(2, 2);
        processor.setLeftOperand(leftNumber);
        processor.setRightOperand(rightNumber);
        processor.setOperation(TProc<TComplex>::Mul);
    }

```

```

        processor.doOperation();

        Assert::AreEqual(TComplex(0, 4).numberString(), processor.getLeftOperand()-
>numberString());
    }

    TEST_METHOD(TestDivisionOperation)
    {
        TProc<TComplex> processor;
        std::unique_ptr<TANumber> leftNumber = std::make_unique<TComplex>(1, 1);
        std::unique_ptr<TANumber> rightNumber = std::make_unique<TComplex>(1, -1);
        processor.setLeftOperand(leftNumber);
        processor.setRightOperand(rightNumber);
        processor.setOperation(TProc<TComplex>::Div);

        processor.doOperation();

        Assert::AreEqual(TComplex(0, 1).numberString(), processor.getLeftOperand()-
>numberString());
    }

    TEST_METHOD(TestSquareFunction)
    {
        TProc<TComplex> processor;
        std::unique_ptr<TANumber> rightNumber = std::make_unique<TComplex>(2, 3);
        processor.setRightOperand(rightNumber);

        processor.doFunction(TProc<TComplex>::Sqr);

        Assert::AreEqual(TComplex(-5, 12).numberString(),
processor.getRightOperand()->numberString());
    }

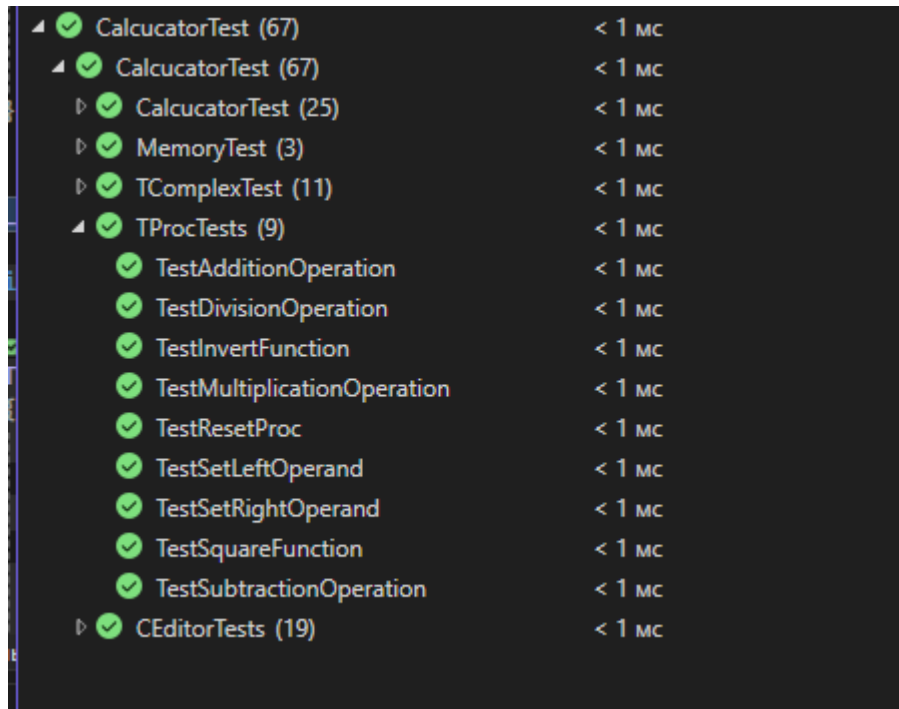
    TEST_METHOD(TestInvertFunction)
    {
        TProc<TComplex> processor;
        std::unique_ptr<TANumber> rightNumber = std::make_unique<TComplex>(2, 3);
        processor.setRightOperand(rightNumber);

        processor.doFunction(TProc<TComplex>::Rev);

        Assert::AreEqual(TComplex(0.153846, -0.230769).numberString(),
processor.getRightOperand()->numberString());
    }
};

```

### 3. Результаты модульных тестов



▲ ✓ CalcucatorTest (67)	< 1 MC
▲ ✓ CalcucatorTest (67)	< 1 MC
▷ ✓ CalcucatorTest (25)	< 1 MC
▷ ✓ MemoryTest (3)	< 1 MC
▷ ✓ TComplexTest (11)	< 1 MC
▲ ✓ TProcTests (9)	< 1 MC
✓ TestAdditionOperation	< 1 MC
✓ TestDivisionOperation	< 1 MC
✓ TestInvertFunction	< 1 MC
✓ TestMultiplicationOperation	< 1 MC
✓ TestResetProc	< 1 MC
✓ TestSetLeftOperand	< 1 MC
✓ TestSetRightOperand	< 1 MC
✓ TestSquareFunction	< 1 MC
✓ TestSubtractionOperation	< 1 MC
▷ ✓ CEditorTests (19)	< 1 MC

### 4. Вывод

По итогам данной лабораторной работе были сформированы практические навыки реализации абстрактных типов данных в соответствии с заданной спецификацией с помощью классов C++ и их модульного тестирования.