Федеральное агентство связи
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»

Лабораторная работа №6

Выполнил:
студенты 4 курса
группы ИП-216
Андрущенко Ф.А.
Литвинов А. Е.
Русецкий А. С.

Проверил:
преподаватель кафедры ПМиК
Агалаков Антон Александрович

Новосибирск, 2025 г.

Задание

1. Разработать и реализовать класс «Ввод и редактирование комплексных чисел» (TEditor), используя класс C++.
2. Протестировать каждую операцию, определенную на типе данных, используя средства модульного тестирования Visual Studio по критерию C2.
3. Если необходимо, предусмотрите возбуждение исключительных ситуаций.

На Унифицированном языке моделирования UML (Unified Modeling Language) наш класс можно обозначить следующим образом:

РедакторР-ичныхЧисел
строка: String числоЕстьНоль: Boolean добавитьЗнак: String добавитьР-ичную цифру(a: Integer): String добавитьНоль: String забойСимвола: String очистить: String конструктор читатьСтрокаВформатеСтроки: String (метод свойства) писатьСтрокаВформатеСтроки(a: String) (метод свойства) редактировать(a: Integer): String
Обязанность: ввод, хранение и редактирование строкового представления р-ичных

4. Класс должен отвечать за посимвольный ввод, хранение и редактирование строкового представления комплексных чисел. Значение комплексного нуля - ‘0, i* 0,’. Класс должен обеспечивать:

- добавление цифры;
- добавление и изменение знака действительной и мнимой частей;
- добавление разделителя целой и дробной частей действительной
- и мнимой частей комплексного числа;
- добавление разделителя мнимой и действительной
- частей комплексного числа
- забой символа, стоящего справа (BackSpace);
- установку нулевого значения комплексного числа (Clear);
- чтение строкового представления комплексного числа;
- запись строкового представления комплексного числа.

Исходный код программы

```
class TEditor:
    # Constants
    DECIMAL_SEPARATOR = ","
    IMAGINARY_SEPARATOR = "i*"
    ZERO_REPRESENTATION = "0, i* 0,"

    def __init__(self):
        """Constructor - initializes with zero representation"""
        self._string = self.ZERO_REPRESENTATION
        self._has_decimal_real = True  # Zero representation already has comma
        self._has_decimal_imaginary = True  # Zero representation already has comma

    @property
    def string(self) -> str:
        """Read string in string format (property method)"""
        return self._string

    @string.setter
    def string(self, value: str) -> None:
        """Write string in string format (property method)"""
        self._string = value
        # Reset decimal flags when setting string
        parts = value.split(self.IMAGINARY_SEPARATOR)
        if len(parts) > 0:
            self._has_decimal_real = self.DECIMAL_SEPARATOR in parts[0]
        if len(parts) > 1:
            self._has_decimal_imaginary = self.DECIMAL_SEPARATOR in parts[1]
        else:
            self._has_decimal_imaginary = False

    def is_zero(self) -> bool:
        """Check if the number is complex zero"""
        return self._string == self.ZERO_REPRESENTATION

    def add_sign(self) -> str:
        """Add or remove minus sign from the string"""
        if self.is_zero():
            # For zero, just add sign to real part
            self._string = "-" + self._string
            return self._string

        parts = self._string.split(self.IMAGINARY_SEPARATOR)
        if len(parts) != 2:
            return self._string

        real_part, imaginary_part = parts

        # Toggle sign for real part
        real_part = real_part.strip()
        if real_part.startswith("-"):
```

```
            real_part = real_part[1:].strip()
        else:
            real_part = "-" + real_part

        # Reconstruct the string
        self._string = (
            f"{real_part} {self.IMAGINARY_SEPARATOR} {imaginary_part.strip()}"
        )
        return self._string

    def add_digit(self, digit: int) -> str:
        """Add a digit to the string if format allows"""
        if not 0 <= digit <= 9:
            raise ValueError("Digit must be between 0 and 9")

        if self.is_zero():
            # Replace zero with digit
            self._string = f"{digit}, i* 0,"
            self._has_decimal_real = True
            return self._string

        # Parse current complex number
        parts = self._string.split(self.IMAGINARY_SEPARATOR)
        if len(parts) != 2:
            return self._string

        real_part, imaginary_part = parts
        real_part = real_part.strip()
        imaginary_part = imaginary_part.strip()

        # Remove comma from real part to build the number
        if "," in real_part:
            real_part = real_part.replace(",", "")

        # Add digit to real part
        real_part += str(digit)

        # Add comma back to the end of real part
        real_part += ","
        self._has_decimal_real = True

        # Ensure imaginary part has comma
        if not self._has_decimal_imaginary:
            imaginary_part += ","
            self._has_decimal_imaginary = True

        self._string = f"{real_part} {self.IMAGINARY_SEPARATOR} {imaginary_part}"
        return self._string

    def add_zero(self) -> str:
        """Add zero to the string if format allows"""
        return self.add_digit(0)
```

```
def backspace(self) -> str:
    """Remove the rightmost character"""
    if len(self._string) > 0 and not self.is_zero():
        self._string = self._string[:-1]
        # If string becomes invalid, reset to zero
        if (
            len(self._string) < len(self.ZERO_REPRESENTATION)
            or "i*" not in self._string
        ):
            self._string = self.ZERO_REPRESENTATION
            self._has_decimal_real = True
            self._has_decimal_imaginary = True
        else:
            # Update decimal flags
            parts = self._string.split(self.IMAGINARY_SEPARATOR)
            if len(parts) >= 1:
                self._has_decimal_real = self.DECIMAL_SEPARATOR in parts[0]
            if len(parts) >= 2:
                self._has_decimal_imaginary = self.DECIMAL_SEPARATOR in parts[1]
    return self._string

def clear(self) -> str:
    """Set to zero complex number representation"""
    self._string = self.ZERO_REPRESENTATION
    self._has_decimal_real = True
    self._has_decimal_imaginary = True
    return self._string

def edit(self, command: int) -> str:
    """Execute editing command based on command number"""
    commands = {
        0: self.clear,
        1: self.backspace,
        2: self.add_sign,
        3: self.add_zero,
    }

    if command in commands:
        return commands[command]()
    elif 4 <= command <= 13:  # Commands for digits 0-9
        return self.add_digit(command - 4)
    else:
        raise ValueError(f"Unknown command: {command}")

def add_decimal_separator(self) -> str:
    """Add decimal separator to real or imaginary part"""
    if self.is_zero():
        self._string = "0, i* 0,"
        self._has_decimal_real = True
        self._has_decimal_imaginary = True
        return self._string
```

```
    parts = self._string.split(self.IMAGINARY_SEPARATOR)
    if len(parts) != 2:
        return self._string

    real_part, imaginary_part = parts
    real_part = real_part.strip()
    imaginary_part = imaginary_part.strip()

    # Add to real part if it doesn't have decimal
    if not self._has_decimal_real:
        real_part += self.DECIMAL_SEPARATOR
        self._has_decimal_real = True
    # Add to imaginary part if real part already has decimal
    elif not self._has_decimal_imaginary:
        imaginary_part += self.DECIMAL_SEPARATOR
        self._has_decimal_imaginary = True

    self._string = f"{real_part} {self.IMAGINARY_SEPARATOR} {imaginary_part}"
    return self._string

def add_imaginary_separator(self) -> str:
    """Add imaginary separator"""
    return self._string
```

Модульные тесты для тестирования класса

```
import unittest
from ueditor import TEditor


class TestTEditor(unittest.TestCase):
    def setUp(self):
        self.editor = TEditor()

    def test_initial_state(self):
        self.assertEqual(self.editor.string, "0, i* 0,")
        self.assertTrue(self.editor.is_zero())

    def test_clear(self):
        self.editor.string = "123, i* 456,"
        result = self.editor.clear()
        self.assertEqual(result, "0, i* 0,")
        self.assertTrue(self.editor.is_zero())

    def test_backspace(self):
        self.editor.string = "123, i* 456,"
        result = self.editor.backspace()
        self.assertEqual(result, "123, i* 456")
        # Test backspace on zero
        self.editor.clear()
        result = self.editor.backspace()
        self.assertEqual(result, "0, i* 0,")

    def test_add_sign(self):
        result = self.editor.add_sign()
        self.assertEqual(result, "-0, i* 0,")
        result = self.editor.add_sign()  # Toggle back
        self.assertEqual(result, "0, i* 0,")

    def test_add_digit(self):
        result = self.editor.add_digit(5)
        self.assertEqual(result, "5, i* 0,")
        result = self.editor.add_digit(3)
        self.assertEqual(result, "53, i* 0,")

    def test_add_zero(self):
        result = self.editor.add_zero()
        self.assertEqual(result, "0, i* 0,")

    def test_edit_commands(self):
        # Test clear command
        result = self.editor.edit(0)
        self.assertEqual(result, "0, i* 0,")
        # Test add digit commands
        result = self.editor.edit(7)  # digit 3 (7-4=3)
        self.assertEqual(result, "3, i* 0,")
```

```
        # Test add sign command
        result = self.editor.edit(2)
        self.assertEqual(result, "-3, i* 0,")

    def test_add_decimal_separator(self):
        self.editor.string = "123 i* 456"
        result = self.editor.add_decimal_separator()
        self.assertEqual(result, "123, i* 456")
        result = self.editor.add_decimal_separator()  # Add to imaginary
        self.assertEqual(result, "123, i* 456,")

    def test_invalid_digit(self):
        with self.assertRaises(ValueError):
            self.editor.add_digit(10)

    def test_invalid_command(self):
        with self.assertRaises(ValueError):
            self.editor.edit(100)


if __name__ == "__main__":
    from rich import print
    from rich.panel import Panel
    from rich.console import Console
    from unittest import TextTestRunner, TestResult

    console = Console()

    class RichTestResult(TestResult):
        def __init__(self, stream, descriptions, verbosity):
            super().__init__(stream, descriptions, verbosity)

        def startTest(self, test):
            super().startTest(test)
            console.print(f"[cyan]Running:[/cyan] {test._testMethodName}")

        def addSuccess(self, test):
            super().addSuccess(test)
            console.print(f"[green]✓ PASS:[/green] {test._testMethodName}")

        def addFailure(self, test, err):
            super().addFailure(test, err)
            console.print(f"[red]✗ FAIL:[/red] {test._testMethodName}")

        def addError(self, test, err):
            super().addError(test, err)
            console.print(f"[magenta]💥 ERROR:[/magenta] {test._testMethodName}")

    runner = TextTestRunner(resultclass=RichTestResult, verbosity=0)
    result = runner.run(unittest.defaultTestLoader.loadTestsFromTestCase(TestTEditor))

    console.print(
```

```
        Panel.fit(
            f"[green]Passed: {result.testsRun - len(result.failures) -
len(result.errors)}[/green]\n"
            f"[red]Failed: {len(result.failures)}[/red]\n"
            f"[magenta]Errors: {len(result.errors)}[/magenta]\n"
            f"[yellow]Total: {result.testsRun}[/yellow]",
            title="Test Results",
        )
    )
```

Результат тестирования методов класса