Министерство науки и высшего образования Российской Федерации федеральное государственное автономное образовательное учреждение высшего образования «Санкт-Петербургский политехнический университет Петра Великого»
(ФГАОУ ВО СПбПУ)
Институт компьютерных наук и кибербезопасности
Высшая школа программной инженерии

КУРСОВАЯ РАБОТА

По дисциплине «Технология объектно-ориентированного программирования»

Разработка математического интерпретатора
(семестр 5)

Студент
группы з5130903/30301 В.А. Смолькин

Оценка выполненной студентом работы:

Преподаватель,
доцент, к.т.н. К.А. Туральчук

Санкт-Петербург – 2026

Содержание

1. Введение

Математический интерпретатор – это программный комплекс, способный *разбирать* и *выполнять* математические выражения, вводимые пользователем в текстовой форме. Проще говоря, интерпретатор позволяет вести интерактивные вычисления, схожие по удобству с работой на калькуляторе, но с поддержкой более сложных операций и структур данных. Области применения математических интерпретаторов весьма широки: они используются как в научных вычислениях, так и в образовательных целях, встроены в математические пакеты и языки программирования.

Например, система MATLAB представляет собой мощную интерактивную среду и язык программирования для численных расчётов и работы с матрицами, широко используемую инженерами и учёными. WolframAlpha – онлайн-система, позиционируемая как *вычислительный центр знаний*, способная по запросу пользователя проводить аналитические вычисления или выдавать фактические данные. Библиотека SymPy для языка Python является примером интерпретатора символьной математики (CAS), который может вычислять и упрощать выражения символически.

В рамках данной работы рассматривается разработка упрощённого CLI-интерпретатора (Command Line Interface) математических выражений. То есть создаётся консольное приложение, которое считывает текст команд пользователя, распознаёт формулы, выполняет указанные вычисления и выводит результат. Основной акцент делается на применении принципов объектно-ориентированного программирования (ООП) при реализации *математического ядра* интерпретатора. Поддерживаются базовые операции с различными типами математических объектов: векторами, матрицами, комплексными числами и рациональными дробями. Приложение обладает функционалом загрузки и исполнения команд из текстового файла, что позволяет проводить пакетную обработку выражений.

Таким образом, итогом работы является прототип консольного математического интерпретатора, демонстрирующий навыки ООП в C++ и способный выполнять заданный набор математических операций на основе исходного кода.

2. Формализация требований к заданию

Функциональные требования. Разрабатываемое приложение должно предоставлять пользователю возможность интерактивно вычислять математические выражения, вводя их в консоли. Поддерживаются два основных режима работы: интерактивный режим диалога и пакетный режим выполнения скрипта из файла. В интерактивном режиме пользователь последовательно вводит строки с выражениями или командами, получая немедленный результат. В файловом режиме программа считывает команды из указанного текстового файла и выводит результаты их выполнения построчно. На вход принимаются выражения, включающие следующие возможности и операторы:

- **Арифметические операции**: сложение (+), вычитание (-), умножение (*) и деление (/) между совместимыми по типу операндами. Операции соблюдают **приоритет**: умножение и деление выполняются раньше сложения и вычитания. Допускается использование круглых скобок для изменения порядка вычислений.
- **Типы данных**: поддерживаются скалярные числа (как действительные рациональные числа, в том числе представимые как дроби), комплексные числа (с использованием символа *i* для мнимой единицы), векторы и матрицы фиксированного размера. Скалярные действительные числа обрабатываются как рациональные (дроби) для сохранения точности: например, при вводе 0.5 или 1/2 результат хранится как дробь 1/2. Комплексные числа могут задаваться как выражения вида a + b*i (где *i* – мнимая единица, предопределённая в системе). Векторы задаются списком элементов в квадратных скобках, например [1 2 3] – вектор из трёх элементов. Матрицы задаются как элементы в квадратных скобках, разделённые точкой с запятой по строкам, например [1 0; 0 1] – матрица 2×2.
- **Операции с составными типами**: для векторов и матриц реализованы покомпонентные операции сложения и вычитания (возможно, только для одинаковых размеров). Матрицы поддерживают умножение на матрицу (при согласованных размерах, как матричное произведение), а также умножение на вектор (матрица столбцов на вектор-столбец) и на скаляр. Векторы можно умножать и делить на скаляр (поэлементное масштабирование). Операции между вектором и матрицей, или вектором разных размеров, а также сложение/вычитание вектора с скаляром – не определены и должны приводить к сообщению об ошибке. Комплексные числа можно складывать, вычитать, умножать и делить друг с другом, а также с рациональными числами (скалярными реалами) – при этом рациональные автоматически приводятся к комплексному типу (имагинярная часть 0). Дробные числа (рациональные) поддерживают все четыре арифметических операции между собой с выдачей результата в виде несократимой дроби.

- **Функции и специальные операции**: реализована функция транспонирования матрицы, обозначаемая как T(X) – где X является матрицей или вектором-строкой. В результате вычисляется транспонированная матрица (или вектор-столбец). Других встроенных функций (например, тригонометрических, экспоненциальных и т.п.) в данной версии интерпретатора не предусмотрено.

Требования к пользовательскому интерфейсу (CLI). Приложение должно быть консольным и обеспечивать простое взаимодействие. При запуске в интерактивном режиме программа выводит приветственное сообщение и приглашение для ввода (например, >>>). Пользователь может вводить либо выражения для вычисления, либо специальные команды. Обязательны следующие команды: команда выхода из приложения (например, ввод слова выход для завершения работы) и команда загрузки файла (например, файл <путь>), которая инициирует чтение и выполнение команд из указанного файла. Также должна быть реализована команда вывода справки (например, помощь), которая напомнит пользователю поддерживаемый синтаксис и примеры использования. После ввода каждого корректного выражения (не являющегося присваиванием) интерпретатор выводит результат вычисления. Если введено присваивание вида X = выражение, результат сохраняется в переменной X, а на экран ничего не выводится (кроме случаев, когда выражение содержит ошибку). Переменные, созданные присваиванием, хранят свое значение в памяти интерпретатора и могут использоваться в последующих выражениях. Имена переменных – буквенно-цифровые идентификаторы (например, *X, M1, result*), регистр символов значимости не имеет (можно считать, что идентификаторы регистронезависимые, если не оговорено иное).

Обработка ошибок. Система должна различать *синтаксические* ошибки ввода и *ошибки вычислений*. К синтаксическим ошибкам относятся: лексически некорректные символы или последовательности (например, недопустимый символ или комбинация вроде двух операторов подряд), несоответствие скобок, неправильный формат литералов векторов/матриц, лишние токены после полного разбора выражения и т.д. В случае такой ошибки интерпретатор должен вывести сообщение о синтаксической ошибке с указанием позиции (номера символа) в строке, где обнаружена проблема. К вычислительным ошибкам относятся ситуации, выявляемые на этапе исполнения: например, попытка выполнить операцию, не применимую к данным типам (сложить вектор и число и т.п.), несовместимость размеров матриц для операций, деление на ноль и т.д. В таких случаях выводится сообщение об ошибке вычисления с описанием проблемы. В обоих случаях работа интерпретатора не прерывается (кроме прерывания исполнения файла, где при ошибке можно перейти к чтению следующей строки). В режиме исполнения файла желательно также указывать номер строки в файле, на которой произошла ошибка, чтобы облегчить отладку входного скрипта.

Архитектура приложения. С точки зрения разработки, приложение разделено на несколько модулей, соответствующих отдельным компонентам функциональности:

- **CLI (интерфейс командной строки)** – модуль, отвечающий за взаимодействие с пользователем и операционной средой. Включает функцию main, которая обрабатывает аргументы командной строки (определяя, запущен ли режим файла) и организует цикл чтения пользовательского ввода. CLI реализует команды верхнего уровня (помощь, выход, файл) и передаёт математические выражения на обработку в ядро. Также на этом уровне формируются сообщения об ошибках на русском языке, выводимые пользователю, и происходит вывод результатов. В данном проекте CLI реализован в файле **MathCLI/main.cpp**, использует возможности стандартной библиотеки C++ для работы с файловой системой (загрузка из файла) и консолью.
- **Математическое ядро (MathCore)** – библиотечный модуль, инкапсулирующий всю логику разбора выражений (парсинг) и вычислений. Предоставляет класс интерпретатора mathcore::Interpreter, который имеет метод выполнения строки executeLine. Этот метод принимает строку с выражением или присваиванием, выполняет полный цикл обработки (лексический разбор, синтаксический разбор, вычисление) и возвращает результат (либо пустое значение, если строка была присваиванием). Ядро включает реализацию лексического анализатора (класс Tokenizer), синтаксического анализатора (методы класса Interpreter) и систему классов для представления данных (Value и его потомки). Для разных типов чисел и коллекций реализованы свои классы-наследники Value: рациональные числа, комплексные числа, векторы, матрицы. Они знают, как выполнять основные операции между собой, используя механизмы наследования и полиморфизма. Также ядро определяет типы исключений ParseError и EvalError (для синтаксических и вычислительных ошибок соответственно) – эти исключения бросаются при обнаружении ошибки и перехватываются на уровне CLI.
- **Модуль тестирования (MathTests)** – набор *юнит-тестов*, покрывающих ключевые элементы функциональности. Автоматические тесты проверяют корректность арифметических операций (например, сложение дробей, нормализацию дроби, умножение матриц и т.п.), а также имитируют выполнение последовательности команд, сравнивая полученный результат с ожидаемым. Тесты реализованы с использованием фреймворка MSTest (Microsoft C++ Unit Testing Framework) в Visual Studio, в коде оформлены как методы, вызывающие функции ядра и утверждающие (Assert), что результаты соответствуют ожидаемым значениям. Наличие тестового модуля обеспечивает регрессионную проверку – при внесении изменений в код можно быстро убедиться, что существующий функционал не нарушен.

Таким образом, требования к программе полностью определены: она должна соответствовать описанным возможностям по части вычислений и обеспечивать удобный интерфейс, а реализация должна быть организована модульно с использованием принципов ООП. Далее рассматривается проектирование системы и конкретные детали реализации.

3. Проектирование приложения «Математический интерпретатор»

На этапе проектирования были выбраны основные классы и структуры данных, отражающие предметную область – математические сущности и интерпретатор. Архитектура оформлена с использованием UML-диаграммы классов и функциональной модели в нотации IDEF0, которые приведены и описаны ниже.

3.1 UML-диаграмма классов

Для отображения статической структуры приложения разработана UML-диаграмма классов, представленная на рисунке 1. Диаграмма отражает основные классы библиотеки MathCore и их взаимосвязи, а также взаимодействие с модулем CLI. Класс Value является абстрактным базовым классом для всех типов значений, поддерживаемых интерпретатором. От него посредством наследования образованы классы RationalValue (рациональное число), ComplexValue (комплексное число), VectorValue (вектор) и MatrixValue (матрица). У класса Value объявлены виртуальные методы базовых операций – сложения, вычитания, умножения, деления и транспонирования (add, sub, mul, div, transpose). Каждый из наследников при необходимости *переопределяет* эти методы, реализуя соответствующую операцию для своего типа данных (либо оставляет реализацию по умолчанию, которая генерирует ошибку “операция не поддерживается”). Такое решение позволяет вызывать, например, метод mul у объекта, не зная статически, какого он типа – конкретная реализация выбирается во время выполнения (полиморфизм).

Класс Interpreter (пространство имён mathcore) реализует функциональность синтаксического анализа и вычисления выражений. В процессе работы он использует лексический анализатор (Tokenizer) для разбиения входной строки на токены. В контексте интерпретатора хранится *таблица переменных* (контекст) – ассоциативный массив имени переменной и её значения (std::map<std::string, ValuePtr>). При разборе присваивания интерпретатор сохраняет новое значение в контекст. Класс Interpreter не является наследником Value, он работает как управляющий класс. Тем не менее, связь между Interpreter и классами значений проявляется в том, что Interpreter создает новые объекты-наследники Value (например, парся число, он решает создать RationalValue или ComplexValue) и возвращает результат как указатель на базовый класс (ValuePtr обозначает std::shared_ptr<Value>). Таким образом, Interpreter агрегирует различные типы значений через базовый интерфейс.

Диаграмма также отражает, что модуль CLI использует класс Interpreter (композиция: в main создаётся объект интерпретатора). CLI не знает о внутреннем устройстве MathCore – общение происходит через публичный метод Interpreter::executeLine. Кроме того, классы исключений ParseError и EvalError (на рисунке они условно показаны как принадлежащие пространству имен MathCore) наследуются от стандартного std::exception (это не отражено на диаграмме ради упрощения) и используются Interpreter для сигнализации об ошибках. Модуль CLI перехватывает эти исключения и преобразует в сообщения пользователю.

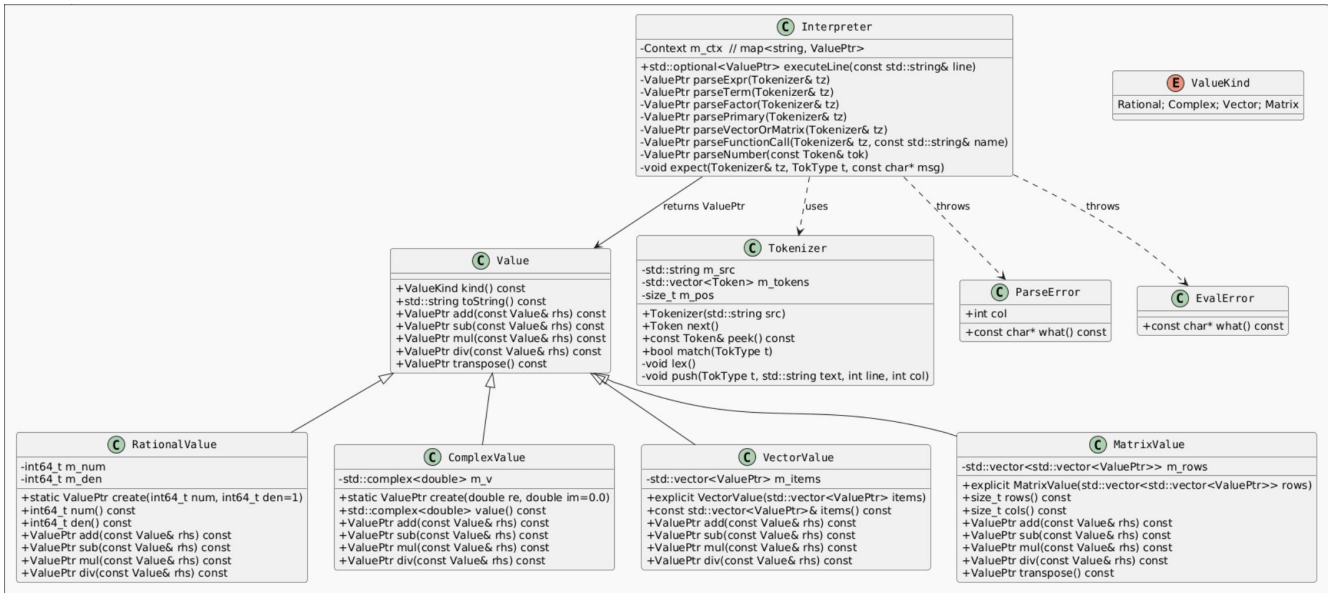


Рисунок 1 – UML-диаграмма классов математического интерпретатора (полноразмерная диаграмма приведена в Приложении Б).

Основные элементы диаграммы (рис. 1):

- **Value** – абстрактный базовый класс: содержит чисто виртуальные методы kind() (возвращает тип значения), toString() (форматирует значение в строку для вывода) и виртуальные методы операций (add, sub, mul, div, transpose) с реализацией по умолчанию “не поддерживается”.
- **RationalValue** – класс для рациональных чисел, хранит числитель (*m_num*) и знаменатель (*m_den*) типа int64_t. При создании выполняется нормализация дроби (сокращение на наибольший общий делитель, знаменатель хранится положительным). Метод toString() возвращает строку формата “числитель/знаменатель” для несократимых дробей или целое число (без дробной части), если знаменатель равен 1.
- **ComplexValue** – класс для комплексных чисел, на основе std::complex<double> (мнимая часть – double). Метод toString() формирует строку вида “a+bi” (или “a”, если мнимая часть почти 0, или “bi”, если действительная часть 0). Также в классе определена внутренняя вспомогательная функция для приведения другого значения к комплексному (если второе значение RationalValue, оно конвертируется к комплексному числу с нулевой мнимой частью).
- **VectorValue** – класс, представляющий вектор (одноразмерный массив) значений. Хранит std::vector<ValuePtr> – список элементов вектора, каждый элемент должен быть скаляром (проверяется при создании). Метод toString() выводит элементы через пробел в квадратных скобках. Операции add/sub реализованы покомпонентно для двух векторов одинакового размера; операции mul/div реализуют умножение или деление **каждого** элемента вектора на скаляр (переданный справа операнд должен быть скалярного типа, иначе операция не поддерживается).

- **MatrixValue** – класс для матриц (двумерный массив). Хранит std::vector<std::vector<ValuePtr>> – список строк, каждая строка – вектор ValuePtr, представляющий элементы в данной строке матрицы. В конструкторе проверяется, что все строки одинаковой длины и не пусты, элементы – скаляры. Метод toString() выводит матрицу в формате квадратных скобок, где строки разделены точкой с запятой и переводом строки. Операции: add/sub реализованы покомпонентно для матриц одинакового размера; mul перегружен для трёх случаев: матрица * скаляр (масштабирование всех элементов), матрица * вектор (умножение матрицы на столбец – возвращается вектор), матрица * матрица (матричное умножение при согласованных размерах). Метод div позволяет делить матрицу на скаляр (каждый элемент делится на число). Метод transpose возвращает транспонированную матрицу (строки и столбцы меняются местами).
- **Interpreter** – класс, осуществляющий разбор строк. Внутри содержит поле контекста (Context m_ctx), хранящее переменные. Имеет публичный метод executeLine(const std::string& line), который выполняет разбор строки: различает, является ли строка присваиванием (операция =) или просто выражением. В случае присваивания сохраняет вычисленное значение в контекст и возвращает пустое значение (null optional). Если строка – выражение, возвращает результат в виде ValuePtr (обёрнутого в std::optional для возможности “отсутствия результата”). Кроме того, внутри Interpreter реализованы методы парсера: parseExpr, parseTerm, parseFactor, parsePrimary и др., соответствующие правилам грамматики. Они последовательно вызывают друг друга, формируя дерево разбора и вычисляя по ходу значения выражения. Лексер Tokenizer используется для получения последовательности токенов из входной строки. В процессе вычислений Interpreter использует фабричные методы различных классов значений: например, в parseNumber при разборе числового литерала решается, создать ли RationalValue или ComplexValue. Также Interpreter обрабатывает встроенную функцию T(...) – вызывает метод transpose() у аргумента.
- **Tokenizer** – класс-лексический анализатор. В его задачу входит пройти по входной строке и разбить её на токены. Каждый токен (Token) имеет тип (перечисление TokType: число, идентификатор, символы операций и скобок и т.д.), текст и позицию в исходной строке. Tokenizer игнорирует пробелы, переводит подряд идущие цифры в токен числа (учитывая десятичную точку, чтобы числа типа 3.14 тоже считались единым токеном). Также он распознаёт идентификаторы (последовательности букв/цифр/подчёркивания, не начинающиеся с цифры). Специально символ *i* (мнимая единица) лексер трактует как идентификатор Ident. Для обработки комплексных литералов это оправдано: интерпретатор распознает “i” как известную переменную-комстанту. Класс Tokenizer предоставляет методы next() (взять следующий токен) и peek() (посмотреть текущий без извлечения) для использования парсером.

UML-диаграмма показывает отношения *наследования* (для классов значений), а также *ассоциации*: у Interpreter ассоциация с Tokenizer (использование), композиция с контекстом (хранит Context – множество переменных). Между Interpreter и Value-классами можно мыслить ассоциацию использования: интерпретатор создаёт объекты значений и вызывает их методы, однако это не явная связь, поэтому на диаграмме не проведена отдельная стрелка (вместо этого отражено через возвращаемые типы методов). Диаграмма является упрощённой: не показаны некоторые вспомогательные детали (например, класс Context раскрыт как просто map), но достаточной для понимания структуры проекта.

3.2 Функциональная модель IDEF0

Для описания функциональной стороны приложения построена диаграмма IDEF0 верхнего уровня. На рисунке 2 представлена контекстная диаграмма A-0, отражающая систему *«Математический интерпретатор»* как единый блок и его взаимодействие с внешними сущностями. Согласно нотации IDEF0, входы (Inputs) поступают слева, выходы (Outputs) выходят справа, механизмы (Mechanisms) располагаются снизу, а управляющие воздействия (Controls) – сверху.

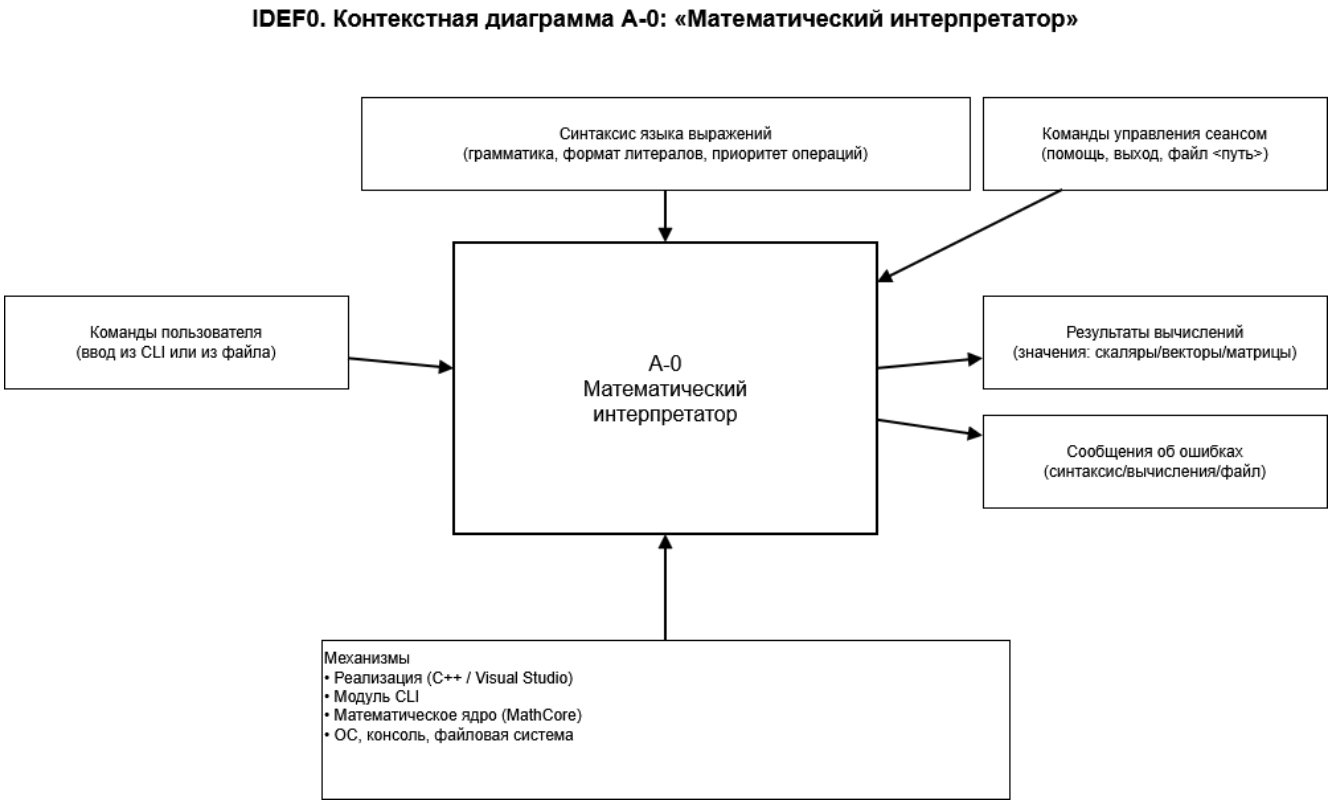


Рисунок 2 – Контекстная диаграмма IDEF0 для приложения-интерпретатора (отражает входы, выходы, механизмы и управление; исходник – в Приложении В).

На диаграмме A-0 (рис. 2) показано, что **входом** системы являются *“Команды пользователя”*, под которыми понимаются строки с математическими выражениями или управляющими командами (вводимые интерактивно либо прочитанные из файла). В качестве **выхода** система генерирует *“Результаты*

вычислений” – это либо значения выражений, выводимые пользователю, либо сообщения об ошибках, если выражение не удалось вычислить. На диаграмме в качестве выходов указаны отдельными стрелками *“Вычисленное значение”* и *“Сообщение об ошибке”*. Управляющим воздействием (Control) является *“Синтаксис языка выражений”* – по сути, это правило, по которому интерпретатор понимает введённые команды. Сюда входят формальные грамматические правила, соглашения о формате данных и допустимых операциях. Другими словами, синтаксис определяет логику работы блока. Еще один элемент управления – *“Команды управления сеансом”* (такие как выход или запрос справки) – они тоже влияют на поведение системы (например, команда “выход” завершает функцию интерпретации). Механизмами, обеспечивающими выполнение функции, являются компьютер и программная реализация: на схеме это отмечено как *“Алгоритмы интерпретатора (C++ код)”* и *“Аппаратное и программное обеспечение (ОС)”*. Эти механизмы указывают, с помощью чего осуществляется функция – то есть с помощью разработанного программного обеспечения на языке C++ в среде выполнения (операционная система, консоль, файловая система).

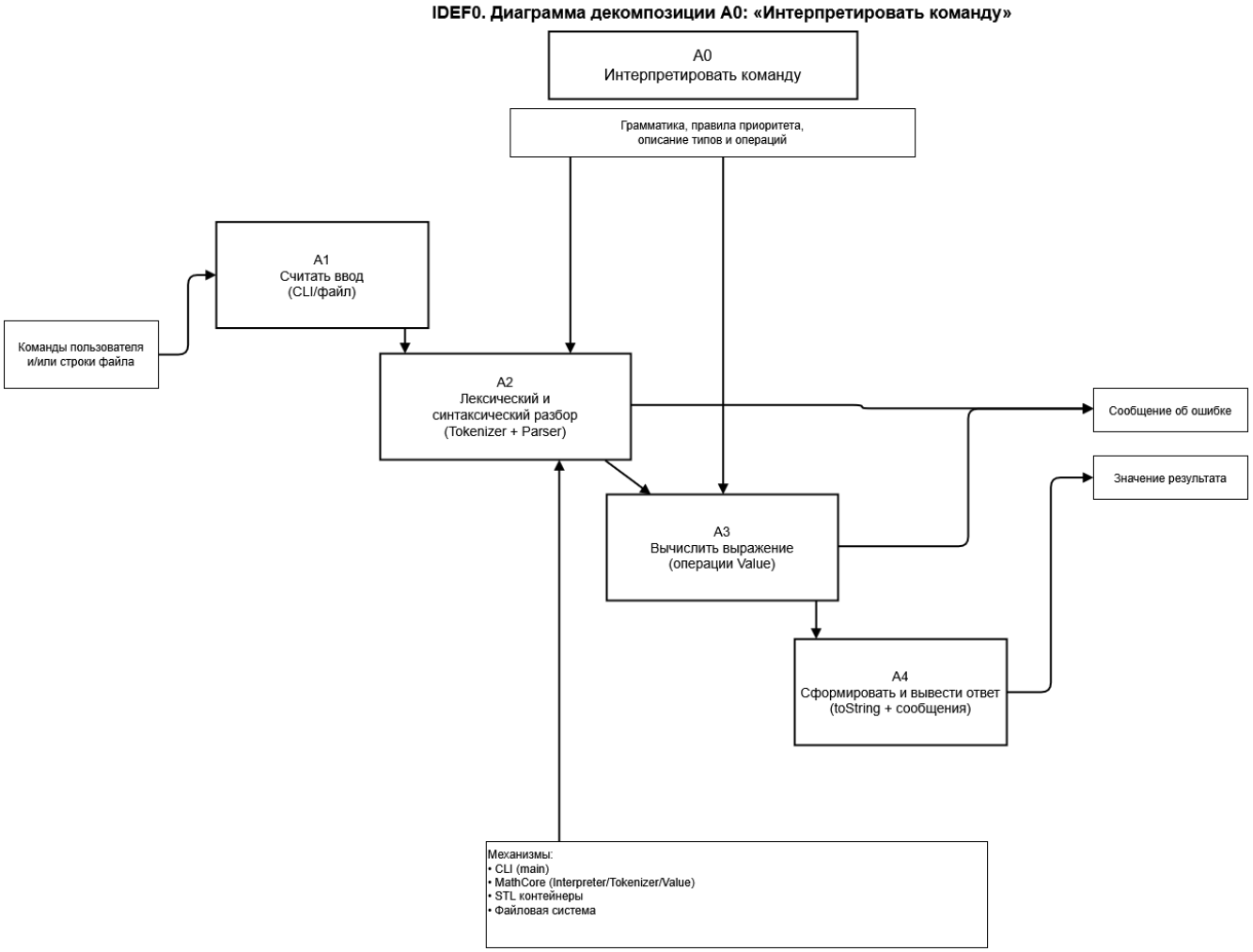


Рисунок 2 – Декомпозиция IDEF0 (исходник – в Приложении Г).

Декомпозиция функции интерпретации на подфункции представлена на диаграмме A0 (расшифровка контекстного блока) (рис. 3). Основные подзадачи, выполняемые системой, можно разделить следующим образом:

- **A1: Считывание команды** – прием ввода от пользователя или из файла. На этом этапе строка поступает во внутренний буфер, определяется, не является ли команда специальной (например, команда чтения файла вызывает отдельную обработку).
- **A2: Разбор выражения** – лексический и синтаксический анализ введённой строки. На этом этапе входная строка преобразуется в структуру данных (дерево выражения) или напрямую вычисляется методом рекурсивного спуска. Если в ходе разбора обнаружен синтаксический сбой, генерируется ошибка синтаксиса (стрелка управления на диаграмме, приводящая к выходу “Сообщение об ошибке”).
- **A3: Вычисление выражения** – непосредственное математическое вычисление результата на основе построенной структуры или последовательности операций. Здесь происходит выполнение операций над объектами Value. В случае возникновения исключительной ситуации (например, деление на ноль, несоответствие типов) генерируется ошибка вычисления.
- **A4: Вывод результата** – форматирование полученного результата в строковое представление и вывод на экран (или обработка команды *выход/помощь* без вычисления). Если было получено исключение, вместо значения формируется сообщение об ошибке, которое также выводится пользователю.

Диаграмма IDEF0, таким образом, описывает, как именно входные данные (команды) трансформируются в выходные (результаты) под воздействием управляющих факторов (синтаксис языка команд) и при помощи определённых механизмов (программа на C++ и вычислительные ресурсы). Эта модель помогает убедиться, что учтены все необходимые интерфейсы системы. Исходный файл диаграммы (формат Draw.io) приведён в Приложении В и может быть открыт в соответствующем редакторе для просмотра деталей.

4. Разработка приложения

В данном разделе описаны ключевые аспекты реализации интерпретатора на языке C++ с акцентом на применении принципов ООП, работу с файлами, обработку ошибок и использование библиотек STL. Код проекта структурирован по каталогам: MathCore (реализация библиотечных классов интерпретатора и математических объектов), MathCLI (файлы интерфейса командной строки) и MathTests (модуль автотестов). Ниже рассматриваются детали реализации по тематическим блокам.

4.1 Структура классов и инкапсуляция данных

Как было спроектировано ранее, все математические объекты представлены экземплярами классов-наследников от единого базового класса Value. Базовый класс определён как абстрактный (Value содержит чисто виртуальные методы) и не имеет собственных полей – он служит интерфейсом. Благодаря этому, реализация различных типов чисел изолирована в соответствующих классах: например, класс RationalValue инкапсулирует внутри два целых числа (числитель и знаменатель) и обеспечивает, что при всех операциях дробь поддерживается в нормализованном состоянии. Поля m_num и m_den объявлены как приватные, доступ к ним извне возможен только через константные методы-аксессоры num() и den() (если необходимо). Таким образом соблюдается принцип **инкапсуляции** – внутреннее представление рационального числа (как пары целых) скрыто от других частей программы; взаимодействие идёт через методы add, mul и т.д., которые сами заботятся о корректной реализации (например, сложение дробей приводит их к общему знаменателю внутри метода RationalValue::add). Аналогично, класс ComplexValue хранит приватно std::complex<double> (поле m_v), не раскрывая наружу деталей реализации комплексного числа – доступ предоставляется только через метод value() (возвращает копию std::complex) или через форматированный вывод toString().

Классы VectorValue и MatrixValue также демонстрируют инкапсуляцию: они содержат STL-контейнеры (std::vector) для хранения элементов, но сами детали этих контейнеров не влияют на внешний интерфейс. Для пользователя (и для остальной части программы) вектор представляется абстрактно – как объект, у которого можно вызвать, например, метод mul с другим объектом. Что произойдёт внутри – решает сам класс VectorValue: он пройдётся по своим элементам и умножит каждый на скаляр, – но внешнему коду это не видно. Подобное разделение отвечает принципу *сокрытия реализации* и позволяет в будущем изменить способ хранения данных (например, использовать другой контейнер для элементов матрицы) без изменения внешнего кода.

Отдельно стоит упомянуть класс Interpreter. Он хранит свое состояние – контекст с переменными – в структуре Context (которая, в свою очередь, содержит std::map<std::string, ValuePtr>). Поле m_ctx объявлено как приватное в Interpreter, и доступ к нему извне предоставляется только для чтения – через метод ctx() (возвращающий константную ссылку). В тестах это используется, чтобы проверить, что после выполнения определённых команд в контексте появились

нужные переменные. Однако напрямую изменять m_ctx или подменять карты переменных извне нельзя, что предотвращает нарушение внутренней логики интерпретатора. Все изменения контекста происходят контролируемо – только внутри метода executeLine при обработке присваивания.

Таким образом, **инкапсуляция** реализована через модификаторы доступа (поля данных скрыты, открыты только необходимые методы) и разделение на модули (CLI не имеет доступа к внутренностям MathCore кроме как через публичный API). Код проекта разбит на заголовочные файлы и cpp-файлы, используя механизм *precompiled headers* (pch.h) для ускорения сборки, но это техническая оптимизация.

4.2 Наследование и полиморфизм в математическом ядре

Принципы **наследования** и **полиморфизма** лежат в основе устройства математического ядра. Наследование в проекте используется для организации семейства типов значений: все виды поддерживаемых данных являются потомками единого интерфейса Value. Это позволяет в коде оперировать указателями типа ValuePtr (alias для std::shared_ptr<Value>) независимо от того, какое конкретно значение (число, вектор и т.д.) они хранят. Например, контекст переменных vars в интерпретаторе – это map<string, ValuePtr>: переменные могут ссылаться на любой объект, реализующий Value. В момент присваивания X = выражение интерпретатор вычисляет некоторое ValuePtr и кладёт в map. При последующем использовании переменной X ее значение (через ValuePtr) будет участвовать в вычислениях, вызывая свои виртуальные методы.

Полиморфизм проявляется прежде всего в механизме *виртуальных методов*. Когда интерпретатор разбирает, например, выражение A + B, где A и B – объекты типа ValuePtr, на этапе выполнения он будет вызывать A->add(*B). Метод add вызывается у того класса, объект которого хранится внутри A. Если A – рациональное число, вызов пойдёт в RationalValue::add; если A – вектор, то в VectorValue::add и т.д. Реализации этих методов различаются кардинально: у скаляров – простое арифм. сложение, у векторов – покомпонентное сложение. Однако интерпретатору не нужно предусматривать каждый вариант – достаточно вызвать виртуальный метод, и благодаря *динамическому связыванию* будет выбрана нужная реализация. Ниже приведён фрагмент кода, иллюстрирующий вызов операций в парсере.

```
ValuePtr Interpreter::parseExpr(Tokenizer& tz) {
    auto left = parseTerm(tz);
    while (true) {
        if (tz.match(TokType::Plus)) {
            auto right = parseTerm(tz);
            left = left->add(*right);
        }
        else if (tz.match(TokType::Minus)) {
            auto right = parseTerm(tz);
            left = left->sub(*right);
        }
        else break;
```

```
    }
    return left;
}

ValuePtr Interpreter::parseTerm(Tokenizer& tz) {
    auto left = parseFactor(tz);
    while (true) {
        if (tz.match(TokType::Star)) {
            auto right = parseFactor(tz);
            // Поддержка Scalar*Vector и Scalar*Matrix
            if (isScalar(left->kind()) && right->kind() ==
ValueKind::Vector) {
                left = right->mul(*left);
            }
            else if (isScalar(left->kind()) && right->kind() ==
ValueKind::Matrix) {
                left = right->mul(*left);
            }
            else {
                left = left->mul(*right);
            }
        }
        else if (tz.match(TokType::Slash)) {
            auto right = parseFactor(tz);
            left = left->div(*right);
        }
        else break;
    }
    return left;
}
```

Листинг 1 – Фрагмент реализации парсера (методы parseExpr и parseTerm класса Interpreter), демонстрирующий вызов виртуальных методов add, sub, mul, div для операндов типа ValuePtr. Особый случай: при умножении скаляра на вектор/матрицу производится перестановка операндов.

В листинге 1 видно, что метод parseTerm специально обрабатывает ситуацию, когда левый операнд – скаляр, а правый – вектор или матрица. Дело в том, что в классах VectorValue и MatrixValue реализовано умножение на скаляр, но обратный порядок (скаляр слева, вектор справа) стандартным виртуальным вызовом не покроется – RationalValue::mul не знает про вектора. Поэтому реализована небольшая хитрость: если обнаружено выражение типа число * вектор, интерпретатор меняет местами операнды и вызывает right->mul(*left), то есть, по сути, вектор умножается на число. Аналогично для матрицы. Этот пример показывает гибкость полиморфизма: добавление нового типа (например, если бы мы ввели класс PolynomialValue для многочленов) потребовало бы реализации операций в новом классе, но общая логика парсера могла бы остаться прежней – она оперирует базовым интерфейсом.

Использование **наследования** облегчает и расширение функциональности. Например, комплексные числа изначально не обязательны, но были добавлены в

проект: достаточно определить класс ComplexValue, унаследованный от Value, реализовать в нём виртуальные методы (add, mul и др.) и при разборе литерала обнаруживать суффикс i – интерпретатор начинает поддерживать комплексную арифметику без изменений в остальных классах. В коде это реализовано так: при создании объекта Interpreter в его контекст *автоматически добавляется* переменная i – мнимая единица. В конструкторе Interpreter() выполняется m_ctx.vars["i"] = ComplexValue::create(0.0, 1.0). Пользователь может использовать i как константу. При вводе выражения вроде 5 + 2i лексер прочитает токены 5, +, 2, i. Парсер воспримет 2 как число (RationalValue) и i как идентификатор – переменную из контекста (ComplexValue). В ходе вычисления 2 автоматически будет приведено к комплексному (в методе ComplexValue::add есть проверка: если второй операнд рациональный, он конвертируется к complex), и результатом станет ComplexValue.

Таким образом, благодаря полиморфизму, код операций написан обобщённо. Каждая пара взаимодействующих типов знает, как складываться или умножаться, а интерпретатор просто направляет вызов к нужному объекту. Если операция не поддержана, используется базовая реализация, которая выбрасывает исключение EvalError. Например, попытка сложить вектор и число: в VectorValue::add мы увидим, что если правый аргумент не Vector, то вызывается Value::add(rhs) – а этот метод всегда бросает ошибку *"операция '+' не поддерживается для данных типов"*. Исключение будет перехвачено и пользователь получит сообщение об ошибке вычисления.

Помимо операций, наследование используется для классов исключений: ParseError и EvalError наследуют от std::exception. Это позволяет поймать их в едином блоке catch (const std::exception& e) на верхнем уровне, если нужно, хотя в коде сделаны раздельные catch для разных видов ошибок (чтобы выдать разные формулировки сообщений). Классы ошибок инкапсулируют также информацию о позиции ошибки (для ParseError хранится номер колонки col, для EvalError может храниться сообщение). Это пример использования наследования для нетипичных объектов, но тоже демонстрирует ООП-подход: ошибки, как и обычные классы, образуют иерархию.

4.3 Работа с файлами и обработка ошибок

Модуль CLI отвечает за реализацию функционала команд файл и корректное завершение работы. Работа с файлами организована с помощью стандартной библиотеки: класс std::ifstream используется для чтения команд из файла построчно. В функции main анализируется аргумент командной строки – если программа запущена с указанием имени файла, интерпретатор сразу вызывает функцию executeFile для этого файла и затем завершает выполнение. Если без аргументов – начинается интерактивный режим.

Команда файл <путь> реализована непосредственно в цикле ввода: при чтении строки от пользователя, прежде чем пытаться парсить как выражение, программа проверяет, не начинается ли строка с ключевого слова "файл". Для упрощения распознавания выполняется тримминг пробелов и, если путь заключён в кавычки,

снятие этих кавычек. После этого вызывается функция executeFile(interp, path), где interp – экземпляр mathcore::Interpreter, а path – путь к файлу. Эта функция открывает файл, затем читает из него построчно и передаёт каждую строку интерпретатору через executeLine. В случае отсутствия файла или ошибок открытия выдаются сообщения об ошибке (с помощью std::cout). Если при выполнении одной из строк файла возникает ParseError или EvalError, они перехватываются внутри executeFile, и выводится сообщение с указанием номера строки в файле. Затем интерпретатор продолжает выполнять следующую строку (т.е. ошибки в файле не прерывают чтение, за исключением случая какой-то непредвиденной исключительной ситуации, которая ловится последним catch (const std::exception&)).

При реализации чтения файла учтены нюансы: пропуск пустых строк, обработка символов возврата каретки \r (в Windows) – эти символы удаляются функцией trimCmd. Это делает работу с файлами более надёжной.

В интерактивном режиме обработка ошибок чуть проще: код main просто вызывает interp.executeLine(line) внутри try/catch. Если пойман ParseError, выводит *"Синтаксическая ошибка (позиция X): <описание>"*. Если EvalError – *"Ошибка вычисления: <описание>"*. Если какое-то иное исключение – *"Неизвестная ошибка: <текст>"*. После вывода ошибка не приводит к закрытию приложения – цикл продолжится и пользователь сможет ввести новую команду.

Важно отметить, что сообщения об ошибках формируются на русском языке для удобства пользователя. Строки сообщений зашиты прямо в код (например, текст *"Нельзя сложить векторы разных размеров."* находится в реализации метода VectorValue::add). Это допустимо для учебного проекта, хотя не самый гибкий подход с точки зрения локализации.

В ходе разработки особое внимание уделялось предотвращению аварийных ситуаций (assert/panic). Например, перед выполнением операции транспонирования проверяется тип: метод Interpreter::parseFunctionCall вызывает arg->transpose(), и только класс MatrixValue переопределяет transpose – для остальных классов вызов приведёт к броску исключения EvalError (*"Операция 'T' не поддерживается..."*). Благодаря этому, если пользователь случайно попробует взять транспонирование скаляра или комплекса, программа не упадёт, а выдаст понятную ошибку. Другой пример: при попытке создания пустого вектора или матрицы (например, [] без элементов) генерируется EvalError (*"Пустой литерал матрицы/вектора."*), то есть такие случаи отсечены.

Подводя итог, реализация ввода-вывода и обработки исключений в приложении следует принципу *безопасности*: любые неправильные действия пользователя интерпретируются и сообщаются ему, не приводя к краху программы. Работа с файлами облегчает тестирование и демонстрацию – можно подготовить файл со списком команд, и программа выполнит их подряд, как если бы их вводили вручную.

4.4 Использование стандартной библиотеки C++

В проекте широко применяется **стандартная библиотека C++ (STL)**, что повышает эффективность разработки и надежность кода. Несколько примеров использования STL в данном приложении:

- Контейнеры std::vector – основной тип для хранения коллекций однородных элементов. В классе VectorValue используется std::vector<ValuePtr> для хранения списка элементов вектора. В классе MatrixValue – вектор векторов. Методы резервирования памяти (reserve) применяются, чтобы избежать лишних реаллокаций при известном размере результата (например, при сложении векторов резервируется требуемое число элементов заранее).
- Контейнер std::map – используется в Context для хранения переменных по именам. Ассоциативный массив автоматически упорядочивает пары “имя–значение” по имени, что не критично для функционала, но упрощает поиск переменной. Вставка новой переменной или поиск существующей реализованы через методы operator[] и find карты. Например, m_ctx.vars[name] = value автоматически создаст ключ, если его не было, или обновит существующее значение.
- Умные указатели std::shared_ptr – тип ValuePtr определён как std::shared_ptr<Value>. Выбор shared_ptr продиктован тем, что значения могут передаваться между функциями и храниться в нескольких местах (например, одна и та же матрица может быть присвоена двум переменным, хотя в текущей реализации такого не происходит, так как executeLine всегда создает новый объект для результата выражения). Shared_ptr обеспечивает автоматическое освобождение памяти, когда объект больше нигде не используется. Кроме того, он облегчает возвращение результата из функции (возврат ValuePtr по значению не приводит к потере управляемого объекта). В тестах, после вычисления, ValuePtr можно разыменовать, чтобы проверить содержимое.
- Класс std::optional – используется в методе Interpreter::executeLine. Он возвращает std::optional<ValuePtr>. Это сделано для удобства: в случае, если строка была присваиванием, метод возвращает std::nullopt (пустое optional), иначе возвращает полученное ValuePtr, обёрнутое в optional. В коде CLI можно проверить auto res = interp.executeLine(line); if (res && *res) { ... } – таким образом, если результат есть и не является нулевым указателем, то его можно вывести. Использование optional вместо, например, возвращения нулевого shared_ptr или использования исключений для обозначения “ничего не произошло” считается более *типа-безопасным*: ясно, что nullopt – это нормальная ситуация (присваивание), а не ошибка.
- Прочие утилиты: функции <cctype> для проверки символов (например, std::isspace и std::isdigit в лексере), <numeric> для нахождения НОД (в RationalValue::normalize используется std::gcd из <numeric> для сокращения дроби). Стримы <sstream> для форматирования вывода (в toString()

комплексных чисел используется std::ostringstream с заданной точностью, чтобы получить строковое представление числа).

Стоит отметить, что использование STL значительно ускорило разработку: например, вместо ручного написания списка токенов, std::vector<Token> позволяет динамически добавлять токены. Методы push_back и индексация делают код лексера лаконичным. Аналогично, работа с std::string (конкатенация, извлечение подстрок) облегчила парсинг (например, при обработке кавычек в пути файла используется метод substr строки).

Разработанное приложение рассчитано на запуск в среде Windows (в коде есть подключение <Windows.h> и вызов SetConsoleOutputCP(65001) для установки кодировки UTF-8 в консоли). Это сделано, чтобы корректно отображались русские сообщения и символы. Тем не менее, сам код достаточно переносимый; за исключением этих вызовов настройки кодировки, остальная логика должна работать и в других ОС (при условии использования UTF-8 терминала по умолчанию).

Подводя итог, этап разработки подтвердил правильность выбранной архитектуры: классовая структура удобна в сопровождении, а применение STL позволило избежать излишнего “изобретения велосипедов” – многие стандартные задачи (хранение коллекций, работа со строками, вычисление НОД) решаются готовыми средствами языка C++.

5. Тестирования приложения

Тестирование проводилось в два этапа: автоматическое модульное тестирование основных компонентов и ручное тестирование сценариев использования через интерфейс CLI. Ниже описаны оба подхода и приведены результаты.

5.1 Автоматическое тестирование (юнит-тесты)

Для проекта разработан набор юнит-тестов (см. каталог MathTests в репозитории). Тесты реализованы на C++ с использованием библиотеки MSTest, которая позволяет оформлять тестовые функции посредством макросов TEST_CLASS и TEST_METHOD. Запуск тестов выполнялся в среде Visual Studio. Цель автоматических тестов – проверить корректность вычислительных модулей MathCore в отрыве от пользовательского интерфейса. В частности, написаны следующие тесты:

- **RationalTests** – тестирование класса рациональных чисел. Проверяется функция нормализации дроби: при создании RationalValue(2,4) ожидается, что внутренняя дробь сократится до 1/2 и метод toString() вернёт строку "1/2". Также проверяется корректность арифметических операций. Например, тест *Addition* складывает 1/3 и 1/6: оба операнда создаются через RationalValue::create, затем вызывается a->add(*b). Ожидаемый результат – 1/2, что подтверждается сравнением строки результата с "1/2". Эти тесты гарантируют, что операции над дробями соблюдают правила арифметики и результат правильно сокращается.
- **InterpreterSmokeTests** – “сквозной” тест интерпретатора на основе заданного сценария. В тесте *SampleFromTask* создаётся экземпляр mathcore::Interpreter и последовательно вызываются executeLine с теми же строками, что приведены в примерах задания. А именно:
    1. V1 = [1 2 3] – создаётся вектор V1.
    2. M1 = [1 0 0; 0 1 0; 0 0 1] – создаётся матрица M1 (единичная 3×3).
    3. V2 = M1 * V1 – результатом должно быть копирование V1 (так как M1 – единичная).
    4. R = 1 / 3 – создаётся дробь 1/3.
    5. V3 = V2 * R – каждый элемент V2 (это [1 2 3]) умножается на 1/3, в результате ожидается [1/3 2/3 1].
    6. M2 = T(M1) – транспонирование единичной матрицы, должно получиться опять единичная матрица.
       Далее тест вызывает it.executeLine("V3") и проверяет, что возвращённое значение есть и равняется [1/3 2/3 1]. Также проверяется it.executeLine("M2") – должно вернуть [1 0 0; 0 1 0; 0 0 1].

Указанный тест охватывает множество аспектов: создание и операции с векторами, матрицами, дробями, а также применение функции транспонирования.

Успешное прохождение теста (все Assert-ы выполняются) свидетельствует о правильности ключевых вычислений. Все юнит-тесты в проекте проходят успешно, что говорит о работоспособности основных модулей ядра.

При написании тестов учитывались также граничные ситуации. Например, тест на сложение дробей проверяет случай, когда необходимо приведение к общему знаменателю и сокращение. Можно было бы добавить тесты на ошибки (например, что вызов executeLine("X") для неизвестной переменной бросает EvalError), однако в рамках данного проекта основные проверки выполнены. Автотесты значительно упростили отладку: при обнаружении несоответствий (например, на ранней стадии сложение дробей давало не сокращённый результат), сразу вносились исправления в код (добавлена нормализация).

5.2 Ручное тестирование (пример работы)

После успешного прохождения модульных тестов приложение было испытано вручную в консольном режиме. Цель ручного тестирования – убедиться, что взаимодействие с пользователем интуитивно и все требования по функциональности удовлетворены. Ниже приведён пример консольной сессии работы интерпретатора, демонстрирующий разные возможности:

```
Математический интерпретатор (введите 'помощь' для справки)
>>> помощь
Команды:
  помощь                - показать справку
  выход                 - завершить
  файл <путь>           - выполнить команды из файла
Синтаксис:
  X = выражение
  выражение
Примеры:
  V1 = [ 1 2 3 ]
  M1 = [ 1 0 0; 0 1 0; 0 0 1 ]
  V2 = M1 * V1
  R = 1 / 3
  V3 = V2 * R
  M2 = T(M1)
  V3
  M2
>>> V1 = [1 2 3]
>>> M1 = [1 0 0; 0 1 0; 0 0 1]
>>> V2 = M1 * V1
>>> R = 1/3
>>> V3 = V2 * R
>>> M2 = T(M1)
>>> V3
[ 1/3 2/3 1 ]
>>> M2
[
1 0 0;
0 1 0;
0 0 1
```

```
]
>>> C = 4 * i
>>> C
4i
>>> Z = C + 1      # сложение комплексного с рациональным
>>> Z
1+4i
>>> A = [2 4] + [3 6]
>>> A
[ 5 10 ]
>>> B = [2 4] + [1 2 3]
Ошибка вычисления: Нельзя сложить векторы разных размеров.
>>> выход
```

Листинг 2 – Пример диалога пользователя с интерпретатором. Введённые команды показаны после приглашения >>>, ответы программы – без префикса. Демонстрируются базовые арифметические операции, работа с векторами/матрицами, комплексными числами и обработка ошибок.

В этом сценарии сначала вызывается команда помощь, программа выводит справочную информацию по управлению и синтаксису, включая пример из задания. Далее пользователь поочередно вводит примеры из справки: создаётся вектор V1, матрица M1, вычисляются V2, R, V3, M2. Вывод команды V3 показывает результат вектора, элементы которого представлены как дроби (1/3, 2/3, 1). Вывод M2 демонстрирует транспонированную матрицу (в данном случае совпадает с исходной единичной; формат вывода – столбцы разделены пробелом, строки разделены ; и переведены на новую строку, заключены в [...]). Далее вводятся команды с комплексными числами: C = 4 * i создаёт комплексное число 4i, вывод C подтверждает отображение в виде 4i. Затем переменная Z получает значение C + 1 (то есть 4i + 1). Результат Z выводится как 1+4i. Этот вывод подтверждает, что интерпретатор правильно сложил комплексное с действительным: последнее автоматически трактовано как 1+0i, результат 1+4i.

Затем проверяется операция с векторами: A = [2 4] + [3 6] – два вектора длины 2, результат выводится как [5 10] (покомпонентное сложение). После этого продемонстрирована ситуация ошибки: попытка сложить вектор длины 2 с вектором длины 3 ([2 4] + [1 2 3]). Программа отобразила сообщение об ошибке вычисления: *"Нельзя сложить векторы разных размеров."* – как и требовалось, операция выявила несовместимость и сгенерировала исключение EvalError, которое было перехвачено и выведено. В конце команда выход завершает сеанс работы.

Ручное тестирование показало, что программа корректно обрабатывает все предусмотренные случаи. Команда файл также была проверена: для этого использовался файл **example.txt** (прилагается в репозитории), содержащий несколько команд. При вводе файл example.txt интерпретатор прочёл файл и вывел результаты вычислений, аналогично тому, как если бы эти команды вводились вручную. Это подтверждает работоспособность пакетного режима.

Дополнительно были проверены нестандартные ситуации, например: ввод неизвестной команды (ни выражение, ни специальное слово) – интерпретатор воспринимает как выражение и выдаёт синтаксическую ошибку; деление на ноль – при вводе 1/0 возвращается ошибка вычисления “Деление на ноль.”; использование незаданной переменной – при вводе, например, X + 1 без предшествующего определения X, выдается ошибка вычисления “Неизвестная переменная: X”. Все эти сообщения реализованы и соответствуют ожидаемому поведению.

В целом, тестирование подтвердило соответствие работы программы заданным требованиям. Приложение стабильно функционирует и корректно реагирует на некорректный ввод. Примеры использования и выявленные ошибки были отражены в руководстве пользователя (справке), чтобы пользователь знал, какие возможности имеются.

6. Заключение

Разработка данного приложения продемонстрировала на практике применение ключевых навыков объектно-ориентированного программирования и языка C++. В процессе реализации были закреплены следующие концепции ООП:

- **Абстракция и моделирование предметной области:** выделены сущности (число, матрица, интерпретатор и пр.), для каждой создан класс с чётко определённой ответственностью. Это позволило отразить математические объекты в коде естественным образом.
- **Инкапсуляция:** реализация каждого класса скрывает детали (например, внутреннее устройство дроби или алгоритм умножения матриц) за публичным интерфейсом. Внешние компоненты пользуются методами, не вмешиваясь во внутренние данные.
- **Наследование:** построена иерархия классов значений, позволяющая расширять систему новыми типами без модификации существующего кода. Общий интерфейс Value обеспечил единообразие обращения с различными математическими объектами.
- **Полиморфизм:** за счёт виртуальных методов достигнуто гибкое управление – операции над объектами вызываются без проверки типов через if, всё происходит автоматически. Это облегчило написание парсера и добавление новых операций.

Кроме ООП, проект потребовал уверенного владения **C++**: работа с указателями (smart pointers), использование стандартных контейнеров и алгоритмов, обработка исключений, организация проекта в Visual Studio (раздельная компиляция, precompiled headers), взаимодействие с файловой системой, настройка кодировки консоли и т.д. Таким образом, в рамках курсовой работы были затронуты многие аспекты разработки на C++.

Конечный результат – прототип CLI-интерпретатора – успешно выполняет поставленные задачи: поддерживает базовые вычисления с несколькими типами данных, интерактивный ввод и выполнение скриптов. Это приложение можно рассматривать как отправную точку для дальнейшего развития. Возможные направления **развития проекта** включают:

- Расширение набора поддерживаемых операций и функций. Например, добавить операции возведения в степень, скалярное произведение векторов, вычисление определителя матрицы, дополнительные математические функции (sin, cos, sqrt и др.).
- Улучшение языка интерпретации: поддержка вещественных чисел (с плавающей запятой) помимо рациональных, введение переменных-скалярных констант (например, *pi*), поддержка более удобного ввода комплексных (распознавание сразу токена “2i” вместо 2 * i).

- Развитие интерфейса: создание графической оболочки или веб-интерфейса, где пользователь может вводить выражения с удобной визуализацией результатов (например, матрицы в виде таблицы).
- Оптимизация производительности: текущая реализация подходит для небольших размеров данных. Для работы с большими матрицами можно было бы использовать более эффективные структуры (библиотеки линейной алгебры). Также можно внедрить *кеширование* результатов или оптимизацию парсера (например, компиляция выражений в дерево команд).
- Расширение возможностей тестирования: добавить более полное покрытие, включая property-based тесты, тесты на корректность сообщений об ошибках, стресс-тесты на больших данных.

Выполнение данной работы позволило не только получить работающую программу, но и отточить методологию разработки: начиная от постановки требований, формализации архитектуры (UML, IDEF0), затем реализации модулей и, наконец, тестирования. Все этапы оказались важными: проектирование классов по UML выявило необходимую иерархию, а написание тестов – помогло в обнаружении и исправлении ошибок. Таким образом, цели курсовой работы достигнуты – создан функциональный математический интерпретатор, а автор приобрёл ценный опыт применения технологий объектно-ориентированного программирования на практике.

Список использованных источников

1. **C++** // Википедия: энциклопедия. – URL: https://ru.wikipedia.org/wiki/C%2B%2B (дата обращения: 11.01.2026).
2. **Документация по C++** // Microsoft Learn. – URL: https://learn.microsoft.com/ru-ru/cpp/cpp/?view=msvc-170 (дата обращения: 11.01.2026).
3. **C++ Standard Library** // cppreference.com – справочник по стандартной библиотеке C++. – URL: https://en.cppreference.com/w/cpp/standard_library.html (дата обращения: 11.01.2026).
4. **Справочник по стандартной библиотеке C++ (STL)** // Microsoft Learn. – URL: https://learn.microsoft.com/ru-ru/cpp/standard-library/cpp-standard-library-reference?view=msvc-170 (дата обращения: 11.01.2026).
5. **Стандартная библиотека шаблонов (STL)** // Википедия: энциклопедия. – URL: https://ru.wikipedia.org/wiki/Стандартная_библиотека_шаблонов (дата обращения: 11.01.2026).
6. **Введение в UML** // Biconsult.ru (PDF). – URL: https://biconsult.ru/img/bi_portal/vvedenie_v_UML.pdf (дата обращения: 11.01.2026).
7. **ГОСТ Р 52573-2006. Информационная технология. Язык моделирования UML** (содержит описание Unified Modeling Language и применение в проектировании). – URL: https://gis-lab.info/docs/law/gost52573_2006.pdf (дата обращения: 11.01.2026).
8. **MATLAB – платформа для численных вычислений** // MathWorks. – URL: https://www.mathworks.com/products/matlab.html (дата обращения: 11.01.2026).
9. **SymPy – библиотека Python для символьной математики** // sympy.org (дата обращения: 11.01.2026).
10. **WolframAlpha – вычислительная онлайновая система** // wolframalpha.com (дата обращения: 11.01.2026).

Приложения

Приложение А. Листинги кода

MathCLI/main.cpp

```
#include <Windows.h>
#include <iostream>
#include <fstream>
#include <string>
#include <filesystem>

#include "MathCore/Interpreter.h"
#include "MathCore/Errors.h"

static void enableUtf8Console() {
    // Для UTF-8 в консоли Windows:
    // - cp 65001 (UTF-8)
    // - проект собран с /utf-8
    SetConsoleOutputCP(65001);
    SetConsoleCP(65001);
}

static void printHelp() {
    std::cout
        << "Команды:\n"
        << "  помощь                - показать справку\n"
        << "  выход                 - завершить\n"
        << "  файл <путь>           - выполнить команды из файла\n"
        << "Синтаксис:\n"
        << "  X = выражение\n"
        << "  выражение\n"
        << "Примеры:\n"
        << "  V1 = [ 1 2 3 ]\n"
        << "  M1 = [ 1 0 0; 0 1 0; 0 0 1 ]\n"
        << "  V2 = M1 * V1\n"
        << "  R = 1 / 3\n"
        << "  V3 = V2 * R\n"
        << "  M2 = T(M1)\n"
        << "  V3\n"
        << "  M2\n";
}

static void executeFile(mathcore::Interpreter& interp, const
std::filesystem::path& p) {
    if (!std::filesystem::exists(p)) {
        std::cout << "Ошибка: файл не найден: " << p.u8string() << "\n";
        return;
    }

    std::ifstream in(p);
    if (!in) {
```

```
        std::cout << "Ошибка: не удалось открыть файл: " << p.u8string() <<
"\n";
        return;
    }

    std::string line;
    int lineNo = 0;
    while (std::getline(in, line)) {
        ++lineNo;
        if (line.empty()) continue;

        try {
            auto res = interp.executeLine(line);
            if (res && *res) std::cout << (*res)->toString() << "\n";
        }
        catch (const mathcore::ParseError& e) {
            std::cout << "Синтаксическая ошибка (строка " << lineNo << ",
позиция " << e.col << "): " << e.what() << "\n";
        }
        catch (const mathcore::EvalError& e) {
            std::cout << "Ошибка вычисления (строка " << lineNo << "): " <<
e.what() << "\n";
        }
        catch (const std::exception& e) {
            std::cout << "Неизвестная ошибка (строка " << lineNo << "): "
<< e.what() << "\n";
        }
    }
}

static std::string trimCmd(std::string s) {
    auto is_ws = [](unsigned char ch) { return std::isspace(ch) != 0; };
    while (!s.empty() && is_ws(static_cast<unsigned char>(s.front())))
s.erase(s.begin());
    while (!s.empty() && is_ws(static_cast<unsigned char>(s.back())))
s.pop_back();
    // На случай '\r' (иногда попадает в конец строки)
    if (!s.empty() && s.back() == '\r') s.pop_back();
    return s;
}

int main(int argc, char** argv) {
    enableUtf8Console();

    mathcore::Interpreter interp;

    // Режим файла: MathCLI.exe <filePath>
    if (argc >= 2) {
        executeFile(interp, std::filesystem::path(argv[1]));
        return 0;
    }
```

```
    std::cout << "Математический интерпретатор (введите 'помощь' для
справки)\n";

    while (true) {
        std::cout << ">>> ";
        std::string line;
        if (!std::getline(std::cin, line)) break;

        if (line == "выход") break;
        if (line == "помощь") { printHelp(); continue; }

        // команда: файл <путь>
        const std::string cmdFile = u8"файл ";
        if (line.rfind(cmdFile, 0) == 0) {
            std::string raw = trimCmd(line.substr(cmdFile.size()));

            // Поддержка пути в кавычках: файл "D:\...\example.txt"
            if (raw.size() >= 2 && raw.front() == '"' && raw.back() == '"')
{
                raw = raw.substr(1, raw.size() - 2);
            }

            auto path = std::filesystem::u8path(raw);
            executeFile(interp, path);
            continue;
        }

        try {
            auto res = interp.executeLine(line);
            if (res && *res) std::cout << (*res)->toString() << "\n";
        }
        catch (const mathcore::ParseError& e) {
            std::cout << "Синтаксическая ошибка (позиция " << e.col << "):
" << e.what() << "\n";
        }
        catch (const mathcore::EvalError& e) {
            std::cout << "Ошибка вычисления: " << e.what() << "\n";
        }
        catch (const std::exception& e) {
            std::cout << "Неизвестная ошибка: " << e.what() << "\n";
        }
    }

    return 0;
}
```

MathCore/Include/MathCore/ComplexValue.h

```
#pragma once
#include "MathCore/Value.h"
#include "MathCore/Errors.h"

#include <complex>
```

```
#include <string>

namespace mathcore {

    class ComplexValue final : public Value {
    public:
        static ValuePtr create(double re, double im);

        ValueKind kind() const override { return ValueKind::Complex; }
        std::string toString() const override;

        std::complex<double> value() const { return m_v; }

        ValuePtr add(const Value& rhs) const override;
        ValuePtr sub(const Value& rhs) const override;
        ValuePtr mul(const Value& rhs) const override;
        ValuePtr div(const Value& rhs) const override;

    public:
        explicit ComplexValue(std::complex<double> v) : m_v(v) {}
        std::complex<double> m_v{};
    };

} // namespace mathcore
```

MathCore/Include/MathCore/Errors.h

```
#pragma once
#include <stdexcept>
#include <string>

namespace mathcore {

    struct ParseError : public std::runtime_error {
        int line;
        int col;
        explicit ParseError(int line_, int col_, const std::string& msg)
            : std::runtime_error(msg), line(line_), col(col_) {}
    };

    struct EvalError : public std::runtime_error {
        explicit EvalError(const std::string& msg) :
std::runtime_error(msg) {}
    };

} // namespace mathcore
```

MathCore/Include/MathCore/Interpreter.h

```
#pragma once
#include "MathCore/Value.h"
#include "MathCore/Tokenizer.h"
#include "MathCore/Errors.h"
```

```
#include "MathCore/VectorMatrix.h"
#include "MathCore/RationalValue.h"
#include "MathCore/ComplexValue.h"

#include <map>
#include <optional>
#include <string>

namespace mathcore {

    struct Context {
        std::map<std::string, ValuePtr> vars;
    };

    class Interpreter {
    public:
        Interpreter();

        // Выполняет одну строку: либо присваивание, либо выражение.
        // Возвращает значение выражения, если строка не присваивание.
        std::optional<ValuePtr> executeLine(const std::string& line);

        // Доступ к контексту (например, для тестов)
        const Context& ctx() const { return m_ctx; }

    private:
        // Parser
        ValuePtr parseExpr(Tokenizer& tz);
        ValuePtr parseTerm(Tokenizer& tz);
        ValuePtr parseFactor(Tokenizer& tz);

        ValuePtr parsePrimary(Tokenizer& tz);
        ValuePtr parseVectorOrMatrix(Tokenizer& tz);
        ValuePtr parseFunctionCall(Tokenizer& tz, const std::string& name);

        ValuePtr parseNumber(const Token& tok);

        void expect(Tokenizer& tz, TokType t, const char* msg);

        Context m_ctx;
    };

} // namespace mathcore
```

MathCore/Include/MathCore/RationalValue.h

```
#pragma once
#include "MathCore/Value.h"
#include "MathCore/Errors.h"

#include <cstdint>
#include <numeric>
```

```
#include <string>

namespace mathcore {

    class ComplexValue; // forward

    class RationalValue final : public Value {
    public:
        static ValuePtr create(int64_t num, int64_t den = 1);

        ValueKind kind() const override { return ValueKind::Rational; }
        std::string toString() const override;

        int64_t num() const { return m_num; }
        int64_t den() const { return m_den; }

        ValuePtr add(const Value& rhs) const override;
        ValuePtr sub(const Value& rhs) const override;
        ValuePtr mul(const Value& rhs) const override;
        ValuePtr div(const Value& rhs) const override;

    public:
        RationalValue(int64_t num, int64_t den);

        static void normalize(int64_t& num, int64_t& den);

        int64_t m_num{};
        int64_t m_den{ 1 };
    };

} // namespace mathcore
```

MathCore/Include/MathCore/Tokenizer.h

```
#pragma once
#include "MathCore/Errors.h"
#include <string>
#include <vector>

namespace mathcore {

    enum class TokType {
        End,
        Ident,
        Number,
        LBracket, RBracket,
        LParen, RParen,
        Semicolon,
        Plus, Minus, Star, Slash,
        Equal
    };
```

```
    struct Token {
        TokType type{ TokType::End };
        std::string text;
        int line{ 1 };
        int col{ 1 };
    };

    class Tokenizer {
    public:
        explicit Tokenizer(std::string src);

        const Token& peek() const { return m_tokens[m_pos]; }
        Token next();
        bool match(TokType t);

    private:
        void lex();
        void push(TokType t, std::string text, int line, int col);

        std::string m_src;
        std::vector<Token> m_tokens;
        size_t m_pos{ 0 };
    };

} // namespace mathcore
```

MathCore/Include/MathCore/Value.h

```
#pragma once
#include <memory>
#include <string>

namespace mathcore {

    enum class ValueKind { Rational, Complex, Vector, Matrix };

    class Value;
    using ValuePtr = std::shared_ptr<Value>;

    class Value {
    public:
        virtual ~Value() = default;

        virtual ValueKind kind() const = 0;
        virtual std::string toString() const = 0;

        // Базовые операции: по умолчанию не поддерживаются.
        virtual ValuePtr add(const Value& rhs) const;
        virtual ValuePtr sub(const Value& rhs) const;
        virtual ValuePtr mul(const Value& rhs) const;
        virtual ValuePtr div(const Value& rhs) const;
        virtual ValuePtr transpose() const; // для матриц
```

```
    };

} // namespace mathcore
```

MathCore/Include/MathCore/VectorMatrix.h

```
#pragma once
#include "MathCore/Value.h"
#include "MathCore/Errors.h"
#include "MathCore/RationalValue.h"
#include "MathCore/ComplexValue.h"

#include <vector>
#include <string>
#include <sstream>

namespace mathcore {

    inline bool isScalar(ValueKind k) { return k == ValueKind::Rational ||
k == ValueKind::Complex; }

    inline ValuePtr scalarMul(const Value& a, const Value& b) { return
a.mul(b); }
    inline ValuePtr scalarDiv(const Value& a, const Value& b) { return
a.div(b); }
    inline ValuePtr scalarAdd(const Value& a, const Value& b) { return
a.add(b); }
    inline ValuePtr scalarSub(const Value& a, const Value& b) { return
a.sub(b); }

    class VectorValue final : public Value {
    public:
        explicit VectorValue(std::vector<ValuePtr> items);

        ValueKind kind() const override { return ValueKind::Vector; }
        std::string toString() const override;

        const std::vector<ValuePtr>& items() const { return m_items; }

        ValuePtr add(const Value& rhs) const override;
        ValuePtr sub(const Value& rhs) const override;
        ValuePtr mul(const Value& rhs) const override; // * scalar
        ValuePtr div(const Value& rhs) const override; // / scalar

    private:
        std::vector<ValuePtr> m_items;
    };

    class MatrixValue final : public Value {
    public:
        explicit MatrixValue(std::vector<std::vector<ValuePtr>> rows);
```

```
        ValueKind kind() const override { return ValueKind::Matrix; }
        std::string toString() const override;

        size_t rows() const { return m_rows.size(); }
        size_t cols() const { return m_rows.empty() ? 0 : m_rows[0].size();
}
        const std::vector<std::vector<ValuePtr>>& data() const { return
m_rows; }

        ValuePtr add(const Value& rhs) const override;
        ValuePtr sub(const Value& rhs) const override;
        ValuePtr mul(const Value& rhs) const override; // * scalar / vector
/ matrix
        ValuePtr div(const Value& rhs) const override; // / scalar
        ValuePtr transpose() const override;

    private:
        std::vector<std::vector<ValuePtr>> m_rows;
    };

} // namespace mathcore
```

MathCore/Src/ComplexValue.cpp

```
#include "pch.h"
#include "MathCore/ComplexValue.h"
#include "MathCore/RationalValue.h"

#include <cmath>
#include <sstream>

namespace mathcore {

    ValuePtr ComplexValue::create(double re, double im) {
        return std::make_shared<ComplexValue>(std::complex<double>(re,
im));
    }

    std::string ComplexValue::toString() const {
        const double re = m_v.real();
        const double im = m_v.imag();

        // Упрощённый вывод
        std::ostringstream oss;
        oss.setf(std::ios::fixed);
        oss.precision(10);

        if (std::abs(im) < 1e-12) {
            oss << re;
            return oss.str();
        }
        if (std::abs(re) < 1e-12) {
```

```
            oss << im << "i";
            return oss.str();
        }

        oss << re;
        if (im >= 0) oss << "+";
        oss << im << "i";
        return oss.str();
    }

    static std::complex<double> asComplex(const Value& v) {
        if (v.kind() == ValueKind::Complex) return static_cast<const
ComplexValue&>(v).value();
        if (v.kind() == ValueKind::Rational) {
            auto& r = static_cast<const RationalValue&>(v);
            return { static_cast<double>(r.num()) /
static_cast<double>(r.den()), 0.0 };
        }
        throw EvalError("Ожидался скаляр (рациональный или комплексный).");
    }

    ValuePtr ComplexValue::add(const Value& rhs) const {
        if (rhs.kind() == ValueKind::Rational || rhs.kind() ==
ValueKind::Complex) {
            return create((m_v + asComplex(rhs)).real(), (m_v +
asComplex(rhs)).imag());
        }
        return Value::add(rhs);
    }

    ValuePtr ComplexValue::sub(const Value& rhs) const {
        if (rhs.kind() == ValueKind::Rational || rhs.kind() ==
ValueKind::Complex) {
            const auto res = m_v - asComplex(rhs);
            return create(res.real(), res.imag());
        }
        return Value::sub(rhs);
    }

    ValuePtr ComplexValue::mul(const Value& rhs) const {
        if (rhs.kind() == ValueKind::Rational || rhs.kind() ==
ValueKind::Complex) {
            const auto res = m_v * asComplex(rhs);
            return create(res.real(), res.imag());
        }
        return Value::mul(rhs);
    }

    ValuePtr ComplexValue::div(const Value& rhs) const {
        if (rhs.kind() == ValueKind::Rational || rhs.kind() ==
ValueKind::Complex) {
            const auto d = asComplex(rhs);
```

```
            if (std::abs(d.real()) < 1e-18 && std::abs(d.imag()) < 1e-18)
throw EvalError("Деление на ноль.");
            const auto res = m_v / d;
            return create(res.real(), res.imag());
        }
        return Value::div(rhs);
    }

} // namespace mathcore
```

MathCore/Src/Interpreter.cpp

```
#include "pch.h"
#include "MathCore/Interpreter.h"

#include <cctype>

namespace mathcore {

    Interpreter::Interpreter() {
        // Встроенная константа i = 0 + 1i
        m_ctx.vars["i"] = ComplexValue::create(0.0, 1.0);
    }

    static bool isAssignStart(const Token& t1, const Token& t2) {
        return t1.type == TokType::Ident && t2.type == TokType::Equal;
    }

    std::optional<ValuePtr> Interpreter::executeLine(const std::string&
line) {
        Tokenizer tz(line);

        // Пустая строка
        if (tz.peek().type == TokType::End) return std::nullopt;

        // Присваивание: IDENT '=' expr
        Token t1 = tz.peek();
        Token t2;
        {
            Tokenizer tz2(line);
            t1 = tz2.next();
            t2 = tz2.peek();
        }

        if (isAssignStart(t1, t2)) {
            const std::string name = t1.text;
            tz.next(); // ident
            tz.next(); // '='
            auto v = parseExpr(tz);
            if (tz.peek().type != TokType::End) throw
ParseError(tz.peek().line, tz.peek().col, "Лишние токены в конце строки.");
            m_ctx.vars[name] = v;
```

```
            return std::nullopt;
        }

        // Иначе — просто выражение
        auto v = parseExpr(tz);
        if (tz.peek().type != TokType::End) throw
ParseError(tz.peek().line, tz.peek().col, "Лишние токены в конце строки.");
        return v;
    }

    void Interpreter::expect(Tokenizer& tz, TokType t, const char* msg) {
        if (!tz.match(t)) {
            const auto& p = tz.peek();
            throw ParseError(p.line, p.col, msg);
        }
    }

    // expr := term (('+'|'-') term)*
    ValuePtr Interpreter::parseExpr(Tokenizer& tz) {
        auto left = parseTerm(tz);
        while (true) {
            if (tz.match(TokType::Plus)) {
                auto right = parseTerm(tz);
                left = left->add(*right);
            }
            else if (tz.match(TokType::Minus)) {
                auto right = parseTerm(tz);
                left = left->sub(*right);
            }
            else break;
        }
        return left;
    }

    // term := factor (('*'|'/') factor)*
    ValuePtr Interpreter::parseTerm(Tokenizer& tz) {
        auto left = parseFactor(tz);
        while (true) {
            if (tz.match(TokType::Star)) {
                auto right = parseFactor(tz);

                // Поддержка Scalar*Vector и Scalar*Matrix
                if (isScalar(left->kind()) && right->kind() ==
ValueKind::Vector) {
                    left = right->mul(*left);
                }
                else if (isScalar(left->kind()) && right->kind() ==
ValueKind::Matrix) {
                    left = right->mul(*left);
                }
                else {
                    left = left->mul(*right);
```

```
            }
        }
        else if (tz.match(TokType::Slash)) {
            auto right = parseFactor(tz);
            left = left->div(*right);
        }
        else break;
    }
    return left;
}

// factor := '-' factor | primary
ValuePtr Interpreter::parseFactor(Tokenizer& tz) {
    if (tz.match(TokType::Minus)) {
        auto v = parseFactor(tz);
        // 0 - v
        auto zero = RationalValue::create(0);
        return zero->sub(*v);
    }
    return parsePrimary(tz);
}

ValuePtr Interpreter::parsePrimary(Tokenizer& tz) {
    const auto& t = tz.peek();

    if (tz.match(TokType::Number)) {
        return parseNumber(t);
    }

    if (tz.match(TokType::Ident)) {
        // function call: IDENT '(' expr ')'
        if (tz.peek().type == TokType::LParen) {
            return parseFunctionCall(tz, t.text);
        }

        // variable
        auto it = m_ctx.vars.find(t.text);
        if (it == m_ctx.vars.end()) throw EvalError("Неизвестная
переменная: " + t.text);
        return it->second;
    }

    if (tz.match(TokType::LParen)) {
        auto v = parseExpr(tz);
        expect(tz, TokType::RParen, "Ожидалась ')'.");
        return v;
    }

    if (tz.match(TokType::LBracket)) {
        return parseVectorOrMatrix(tz);
    }
```

```
        throw ParseError(t.line, t.col, "Ожидалось выражение.");
    }

    ValuePtr Interpreter::parseFunctionCall(Tokenizer& tz, const
std::string& name) {
        expect(tz, TokType::LParen, "Ожидалась '('.");
        auto arg = parseExpr(tz);
        expect(tz, TokType::RParen, "Ожидалась ')'.");
        if (name == "T") return arg->transpose();
        throw EvalError("Неизвестная функция: " + name);
    }

    static bool hasDot(const std::string& s) {
        for (char c : s) if (c == '.') return true;
        return false;
    }

    ValuePtr Interpreter::parseNumber(const Token& tok) {
        // Поддержка десятичных как рациональных: 3.25 = 325/100 -> 13/4
        const std::string& s = tok.text;

        if (!hasDot(s)) {
            // int64
            int64_t n = 0;
            bool neg = false;
            size_t i = 0;
            if (i < s.size() && s[i] == '+') ++i;
            if (i < s.size() && s[i] == '-') { neg = true; ++i; }
            for (; i < s.size(); ++i) {
                if (!std::isdigit(static_cast<unsigned char>(s[i])))
                    throw ParseError(tok.line, tok.col, "Некорректное
число.");
                n = n * 10 + (s[i] - '0');
            }
            if (neg) n = -n;
            return RationalValue::create(n);
        }

        // decimal -> rational
        // form: [digits]? '.' digits
        std::string a, b;
        size_t p = s.find('.');
        a = (p == 0) ? "0" : s.substr(0, p);
        b = s.substr(p + 1);
        if (b.empty()) b = "0";

        bool neg = false;
        if (!a.empty() && a[0] == '-') { neg = true; a = a.substr(1); }
        if (a.empty()) a = "0";

        int64_t intPart = 0;
        for (char c : a) {
```

```
            if (!std::isdigit(static_cast<unsigned char>(c))) throw
ParseError(tok.line, tok.col, "Некорректное число.");
            intPart = intPart * 10 + (c - '0');
        }

        int64_t fracPart = 0;
        int64_t den = 1;
        for (char c : b) {
            if (!std::isdigit(static_cast<unsigned char>(c))) throw
ParseError(tok.line, tok.col, "Некорректное число.");
            fracPart = fracPart * 10 + (c - '0');
            den *= 10;
        }

        int64_t num = intPart * den + fracPart;
        if (neg) num = -num;
        return RationalValue::create(num, den);
    }

    ValuePtr Interpreter::parseVectorOrMatrix(Tokenizer& tz) {
        // Внутри: элементы разделяются пробелами, строки матрицы
отделяются ';'
        // Пример: [ 1 0; 0 1 ]
        // Закрывающая ']' уже НЕ съедена (мы съели '[' до вызова)
        std::vector<std::vector<ValuePtr>> rows;
        rows.push_back({});

        while (true) {
            if (tz.peek().type == TokType::RBracket) {
                tz.next(); // ]
                break;
            }

            if (tz.match(TokType::Semicolon)) {
                rows.push_back({});
                continue;
            }

            auto v = parseExpr(tz);
            if (!isScalar(v->kind())) throw EvalError("Элемент
вектора/матрицы должен быть скаляром.");
            rows.back().push_back(v);
        }

        // Удалим возможную пустую последнюю строку
        if (!rows.empty() && rows.back().empty()) rows.pop_back();
        if (rows.empty()) throw EvalError("Пустой литерал
матрицы/вектора.");

        // Если одна строка — это вектор
        if (rows.size() == 1) {
            return std::make_shared<VectorValue>(rows[0]);
```

```
        }
        return std::make_shared<MatrixValue>(rows);
    }

} // namespace mathcore
```

MathCore/Src/RationalValue.cpp

```
#include "pch.h"
#include "MathCore/RationalValue.h"
#include "MathCore/ComplexValue.h"

#include <cmath>

namespace mathcore {

    static int64_t abs64(int64_t x) { return x < 0 ? -x : x; }

    void RationalValue::normalize(int64_t& num, int64_t& den) {
        if (den == 0) throw EvalError("Деление на ноль (знаменатель равен
0).");
        if (den < 0) { den = -den; num = -num; }
        const int64_t g = std::gcd(abs64(num), abs64(den));
        if (g != 0) { num /= g; den /= g; }
    }

    RationalValue::RationalValue(int64_t num, int64_t den) : m_num(num),
m_den(den) {
        normalize(m_num, m_den);
    }

    ValuePtr RationalValue::create(int64_t num, int64_t den) {
        return std::make_shared<RationalValue>(num, den);
    }

    std::string RationalValue::toString() const {
        // normalize() уже гарантирует: m_den > 0 и дробь сокращена.
        if (m_den == 1) return std::to_string(m_num);

        const bool neg = (m_num < 0);

        // Безопасное взятие модуля для INT64_MIN
        const uint64_t un = neg
            ? (static_cast<uint64_t>(-(m_num + 1)) + 1ULL)
            : static_cast<uint64_t>(m_num);

        const uint64_t ud = static_cast<uint64_t>(m_den);

        const uint64_t whole = un / ud;
        const uint64_t rem = un % ud;

        // Делится нацело -> просто целое число
```

```
        if (rem == 0) {
            const std::string s = std::to_string(whole);
            return neg ? ("-" + s) : s;
        }

        // Правильная дробь (целая часть 0) -> "-a/b" или "a/b"
        if (whole == 0) {
            const std::string s = std::to_string(rem) + "/" +
std::to_string(ud);
            return neg ? ("-" + s) : s;
        }

        // Смешанная дробь -> "-q r/d" или "q r/d"
        const std::string s =
            std::to_string(whole) + "+(" + std::to_string(rem) + "/" +
std::to_string(ud) + ")";

        return neg ? ("-" + s) : s;
    }

    static double toDouble(const RationalValue& r) {
        return static_cast<double>(r.num()) / static_cast<double>(r.den());
    }

    ValuePtr RationalValue::add(const Value& rhs) const {
        if (rhs.kind() == ValueKind::Rational) {
            auto& r = static_cast<const RationalValue&>(rhs);
            // a/b + c/d = (ad + cb)/bd
            const int64_t n = m_num * r.den() + r.num() * m_den;
            const int64_t d = m_den * r.den();
            return RationalValue::create(n, d);
        }
        if (rhs.kind() == ValueKind::Complex) {
            return ComplexValue::create(toDouble(*this), 0.0)->add(rhs);
        }
        return Value::add(rhs);
    }

    ValuePtr RationalValue::sub(const Value& rhs) const {
        if (rhs.kind() == ValueKind::Rational) {
            auto& r = static_cast<const RationalValue&>(rhs);
            const int64_t n = m_num * r.den() - r.num() * m_den;
            const int64_t d = m_den * r.den();
            return RationalValue::create(n, d);
        }
        if (rhs.kind() == ValueKind::Complex) {
            return ComplexValue::create(toDouble(*this), 0.0)->sub(rhs);
        }
        return Value::sub(rhs);
    }

    ValuePtr RationalValue::mul(const Value& rhs) const {
```

```
        if (rhs.kind() == ValueKind::Rational) {
            auto& r = static_cast<const RationalValue&>(rhs);
            return RationalValue::create(m_num * r.num(), m_den * r.den());
        }
        if (rhs.kind() == ValueKind::Complex) {
            return ComplexValue::create(toDouble(*this), 0.0)->mul(rhs);
        }
        return Value::mul(rhs);
    }

    ValuePtr RationalValue::div(const Value& rhs) const {
        if (rhs.kind() == ValueKind::Rational) {
            auto& r = static_cast<const RationalValue&>(rhs);
            if (r.num() == 0) throw EvalError("Деление на ноль.");
            return RationalValue::create(m_num * r.den(), m_den * r.num());
        }
        if (rhs.kind() == ValueKind::Complex) {
            return ComplexValue::create(toDouble(*this), 0.0)->div(rhs);
        }
        return Value::div(rhs);
    }

} // namespace mathcore
```

MathCore/Src/Tokenizer.cpp

```
#include "pch.h"
#include "MathCore/Tokenizer.h"
#include <cctype>

namespace mathcore {

    Tokenizer::Tokenizer(std::string src) : m_src(std::move(src)) {
        lex();
    }

    void Tokenizer::push(TokType t, std::string text, int line, int col) {
        m_tokens.push_back(Token{ t, std::move(text), line, col });
    }

    void Tokenizer::lex() {
        int line = 1, col = 1;
        for (size_t i = 0; i < m_src.size();) {
            unsigned char ch = static_cast<unsigned char>(m_src[i]);

            if (ch == '\n') { ++line; col = 1; ++i; continue; }
            if (std::isspace(ch)) { ++col; ++i; continue; }

            const int startCol = col;

            switch (ch) {
```

```
            case '[': push(TokType::LBracket, "[", line, startCol); ++i;
++col; continue;
            case ']': push(TokType::RBracket, "]", line, startCol); ++i;
++col; continue;
            case '(': push(TokType::LParen, "(", line, startCol); ++i;
++col; continue;
            case ')': push(TokType::RParen, ")", line, startCol); ++i;
++col; continue;
            case ';': push(TokType::Semicolon, ";", line, startCol); ++i;
++col; continue;
            case '+': push(TokType::Plus, "+", line, startCol); ++i; ++col;
continue;
            case '-': push(TokType::Minus, "-", line, startCol); ++i;
++col; continue;
            case '*': push(TokType::Star, "*", line, startCol); ++i; ++col;
continue;
            case '/': push(TokType::Slash, "/", line, startCol); ++i;
++col; continue;
            case '=': push(TokType::Equal, "=", line, startCol); ++i;
++col; continue;
            default: break;
            }

            // number: digits [ '.' digits ]
            if (std::isdigit(ch) || ch == '.') {
                size_t j = i;
                bool seenDot = false;
                if (m_src[j] == '.') { seenDot = true; ++j; }
                while (j < m_src.size() &&
std::isdigit(static_cast<unsigned char>(m_src[j]))) ++j;
                if (j < m_src.size() && m_src[j] == '.' && !seenDot) {
                    seenDot = true;
                    ++j;
                    while (j < m_src.size() &&
std::isdigit(static_cast<unsigned char>(m_src[j]))) ++j;
                }
                auto txt = m_src.substr(i, j - i);
                push(TokType::Number, txt, line, startCol);
                col += static_cast<int>(j - i);
                i = j;
                continue;
            }

            // ident: [A-Za-z_][A-Za-z0-9_]*
            if (std::isalpha(ch) || ch == '_') {
                size_t j = i;
                while (j < m_src.size()) {
                    unsigned char c = static_cast<unsigned char>(m_src[j]);
                    if (!(std::isalnum(c) || c == '_')) break;
                    ++j;
                }
                auto txt = m_src.substr(i, j - i);
```

```
                push(TokType::Ident, txt, line, startCol);
                col += static_cast<int>(j - i);
                i = j;
                continue;
            }

            throw ParseError(line, startCol, "Недопустимый символ во
входной строке.");
        }

        push(TokType::End, "", line, col);
    }

    Token Tokenizer::next() {
        if (m_pos >= m_tokens.size()) return Token{};
        return m_tokens[m_pos++];
    }

    bool Tokenizer::match(TokType t) {
        if (peek().type == t) { next(); return true; }
        return false;
    }

} // namespace mathcore
```

MathCore/Src/Value.cpp

```
#include "pch.h"
#include "MathCore/Value.h"
#include "MathCore/Errors.h"

namespace mathcore {

     ValuePtr Value::add(const Value&) const { throw EvalError("Операция
'+' не поддерживается для данных типов."); }
     ValuePtr Value::sub(const Value&) const { throw EvalError("Операция
'-' не поддерживается для данных типов."); }
     ValuePtr Value::mul(const Value&) const { throw EvalError("Операция
'*' не поддерживается для данных типов."); }
     ValuePtr Value::div(const Value&) const { throw EvalError("Операция
'/' не поддерживается для данных типов."); }
     ValuePtr Value::transpose() const { throw EvalError("Операция 'T'
(транспонирование) не поддерживается для данного типа."); }

} // namespace mathcore
```

MathCore/Src/VectorMatrix.cpp

```
#include "pch.h"
#include "MathCore/VectorMatrix.h"

namespace mathcore {
```

```
    static void ensureScalar(const Value& v) {
        if (!isScalar(v.kind())) throw EvalError("Ожидался скаляр
(рациональный или комплексный).");
    }

    VectorValue::VectorValue(std::vector<ValuePtr> items) :
m_items(std::move(items)) {
        for (auto& x : m_items) {
            if (!x) throw EvalError("Вектор содержит пустой элемент.");
            ensureScalar(*x);
        }
    }

    std::string VectorValue::toString() const {
        std::ostringstream oss;
        oss << "[ ";
        for (size_t i = 0; i < m_items.size(); ++i) {
            if (i) oss << " ";
            oss << m_items[i]->toString();
        }
        oss << " ]";
        return oss.str();
    }

    ValuePtr VectorValue::add(const Value& rhs) const {
        if (rhs.kind() != ValueKind::Vector) return Value::add(rhs);
        auto& v = static_cast<const VectorValue&>(rhs);
        if (v.items().size() != m_items.size()) throw EvalError("Нельзя
сложить векторы разных размеров.");
        std::vector<ValuePtr> out;
        out.reserve(m_items.size());
        for (size_t i = 0; i < m_items.size(); ++i)
out.push_back(scalarAdd(*m_items[i], *v.items()[i]));
        return std::make_shared<VectorValue>(std::move(out));
    }

    ValuePtr VectorValue::sub(const Value& rhs) const {
        if (rhs.kind() != ValueKind::Vector) return Value::sub(rhs);
        auto& v = static_cast<const VectorValue&>(rhs);
        if (v.items().size() != m_items.size()) throw EvalError("Нельзя
вычесть векторы разных размеров.");
        std::vector<ValuePtr> out;
        out.reserve(m_items.size());
        for (size_t i = 0; i < m_items.size(); ++i)
out.push_back(scalarSub(*m_items[i], *v.items()[i]));
        return std::make_shared<VectorValue>(std::move(out));
    }

    ValuePtr VectorValue::mul(const Value& rhs) const {
        if (!isScalar(rhs.kind())) return Value::mul(rhs);
        std::vector<ValuePtr> out;
```

```
        out.reserve(m_items.size());
        for (auto& x : m_items) out.push_back(scalarMul(*x, rhs));
        return std::make_shared<VectorValue>(std::move(out));
    }

    ValuePtr VectorValue::div(const Value& rhs) const {
        if (!isScalar(rhs.kind())) return Value::div(rhs);
        std::vector<ValuePtr> out;
        out.reserve(m_items.size());
        for (auto& x : m_items) out.push_back(scalarDiv(*x, rhs));
        return std::make_shared<VectorValue>(std::move(out));
    }

    MatrixValue::MatrixValue(std::vector<std::vector<ValuePtr>> rows) :
m_rows(std::move(rows)) {
        if (m_rows.empty()) throw EvalError("Матрица не может быть
пустой.");
        const size_t c = m_rows[0].size();
        if (c == 0) throw EvalError("Матрица не может иметь 0 столбцов.");
        for (auto& r : m_rows) {
            if (r.size() != c) throw EvalError("Все строки матрицы должны
иметь одинаковую длину.");
            for (auto& x : r) {
                if (!x) throw EvalError("Матрица содержит пустой
элемент.");
                ensureScalar(*x);
            }
        }
    }

    std::string MatrixValue::toString() const {
        std::ostringstream oss;
        oss << "[\n";
        for (size_t i = 0; i < m_rows.size(); ++i) {
            if (i) oss << ";\n";
            for (size_t j = 0; j < m_rows[i].size(); ++j) {
                if (j) oss << " ";
                oss << m_rows[i][j]->toString();
            }
        }
        oss << "\n]";
        return oss.str();
    }

    ValuePtr MatrixValue::add(const Value& rhs) const {
        if (rhs.kind() != ValueKind::Matrix) return Value::add(rhs);
        auto& m = static_cast<const MatrixValue&>(rhs);
        if (rows() != m.rows() || cols() != m.cols()) throw
EvalError("Нельзя сложить матрицы разных размеров.");
        std::vector<std::vector<ValuePtr>> out(rows(),
std::vector<ValuePtr>(cols()));
        for (size_t i = 0; i < rows(); ++i)
```

```
            for (size_t j = 0; j < cols(); ++j)
                out[i][j] = scalarAdd(*m_rows[i][j], *m.data()[i][j]);
        return std::make_shared<MatrixValue>(std::move(out));
    }

    ValuePtr MatrixValue::sub(const Value& rhs) const {
        if (rhs.kind() != ValueKind::Matrix) return Value::sub(rhs);
        auto& m = static_cast<const MatrixValue&>(rhs);
        if (rows() != m.rows() || cols() != m.cols()) throw
EvalError("Нельзя вычесть матрицы разных размеров.");
        std::vector<std::vector<ValuePtr>> out(rows(),
std::vector<ValuePtr>(cols()));
        for (size_t i = 0; i < rows(); ++i)
            for (size_t j = 0; j < cols(); ++j)
                out[i][j] = scalarSub(*m_rows[i][j], *m.data()[i][j]);
        return std::make_shared<MatrixValue>(std::move(out));
    }

    ValuePtr MatrixValue::mul(const Value& rhs) const {
        // Matrix * Scalar
        if (isScalar(rhs.kind())) {
            std::vector<std::vector<ValuePtr>> out(rows(),
std::vector<ValuePtr>(cols()));
            for (size_t i = 0; i < rows(); ++i)
                for (size_t j = 0; j < cols(); ++j)
                    out[i][j] = scalarMul(*m_rows[i][j], rhs);
            return std::make_shared<MatrixValue>(std::move(out));
        }

        // Matrix * Vector
        if (rhs.kind() == ValueKind::Vector) {
            auto& v = static_cast<const VectorValue&>(rhs);
            if (cols() != v.items().size()) throw EvalError("Нельзя
умножить: число столбцов матрицы не равно размеру вектора.");
            std::vector<ValuePtr> out(rows());
            for (size_t i = 0; i < rows(); ++i) {
                // sum_j a[i][j] * v[j]
                ValuePtr acc = RationalValue::create(0);
                for (size_t j = 0; j < cols(); ++j) {
                    auto prod = scalarMul(*m_rows[i][j], *v.items()[j]);
                    acc = scalarAdd(*acc, *prod);
                }
                out[i] = acc;
            }
            return std::make_shared<VectorValue>(std::move(out));
        }

        // Matrix * Matrix
        if (rhs.kind() == ValueKind::Matrix) {
            auto& b = static_cast<const MatrixValue&>(rhs);
            if (cols() != b.rows()) throw EvalError("Нельзя умножить
матрицы: A.cols != B.rows.");
```

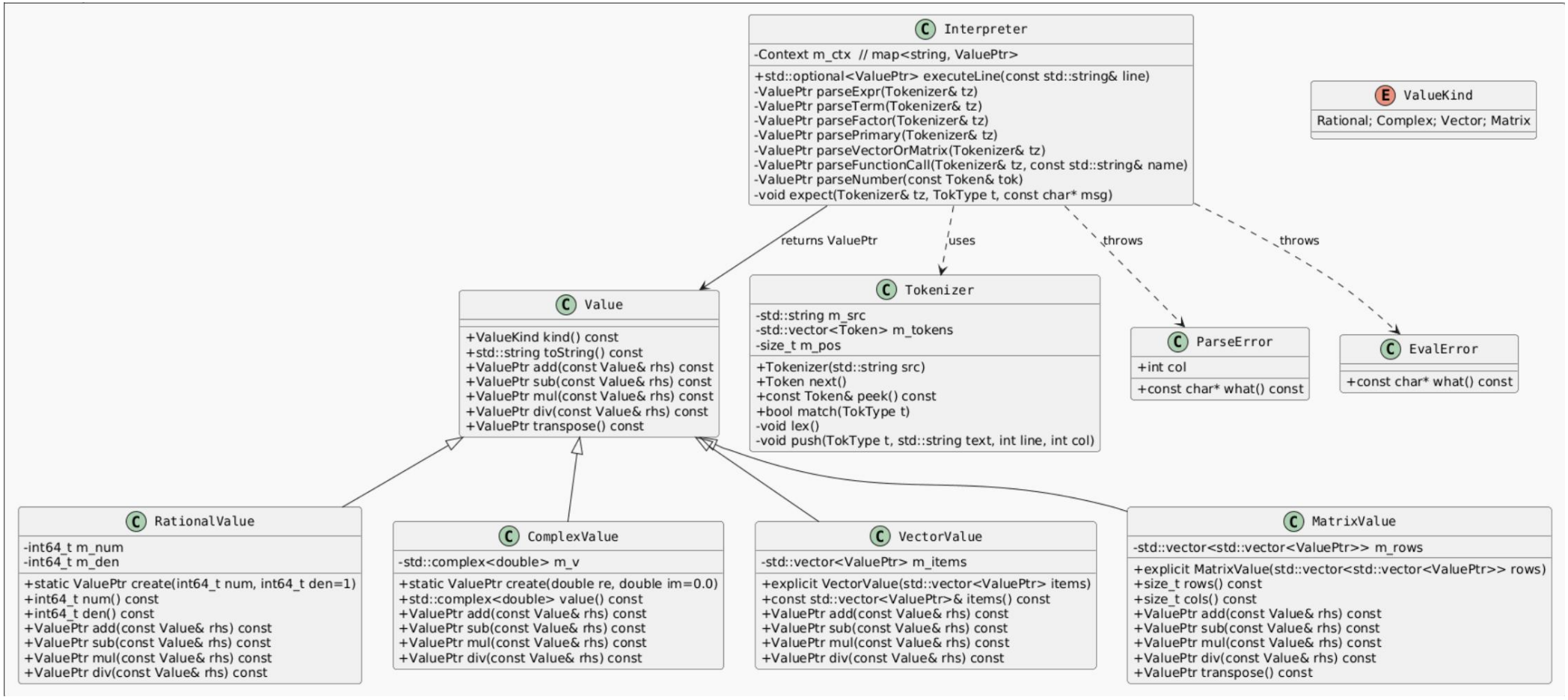
```
            std::vector<std::vector<ValuePtr>> out(rows(),
std::vector<ValuePtr>(b.cols()));
            for (size_t i = 0; i < rows(); ++i) {
                for (size_t k = 0; k < b.cols(); ++k) {
                    ValuePtr acc = RationalValue::create(0);
                    for (size_t j = 0; j < cols(); ++j) {
                        auto prod = scalarMul(*m_rows[i][j],
*b.data()[j][k]);
                        acc = scalarAdd(*acc, *prod);
                    }
                    out[i][k] = acc;
                }
            }
            return std::make_shared<MatrixValue>(std::move(out));
        }

        return Value::mul(rhs);
    }

    ValuePtr MatrixValue::div(const Value& rhs) const {
        if (!isScalar(rhs.kind())) return Value::div(rhs);
        std::vector<std::vector<ValuePtr>> out(rows(),
std::vector<ValuePtr>(cols()));
        for (size_t i = 0; i < rows(); ++i)
            for (size_t j = 0; j < cols(); ++j)
                out[i][j] = scalarDiv(*m_rows[i][j], rhs);
        return std::make_shared<MatrixValue>(std::move(out));
    }

    ValuePtr MatrixValue::transpose() const {
        std::vector<std::vector<ValuePtr>> out(cols(),
std::vector<ValuePtr>(rows()));
        for (size_t i = 0; i < rows(); ++i)
            for (size_t j = 0; j < cols(); ++j)
                out[j][i] = m_rows[i][j];
        return std::make_shared<MatrixValue>(std::move(out));
    }

} // namespace mathcore
```

MathTests/MathTests.cpp

```
#include "pch.h"
#include "CppUnitTest.h"

#include "MathCore/Interpreter.h"
#include "MathCore/RationalValue.h"
#include "MathCore/VectorMatrix.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace MathTests {
```

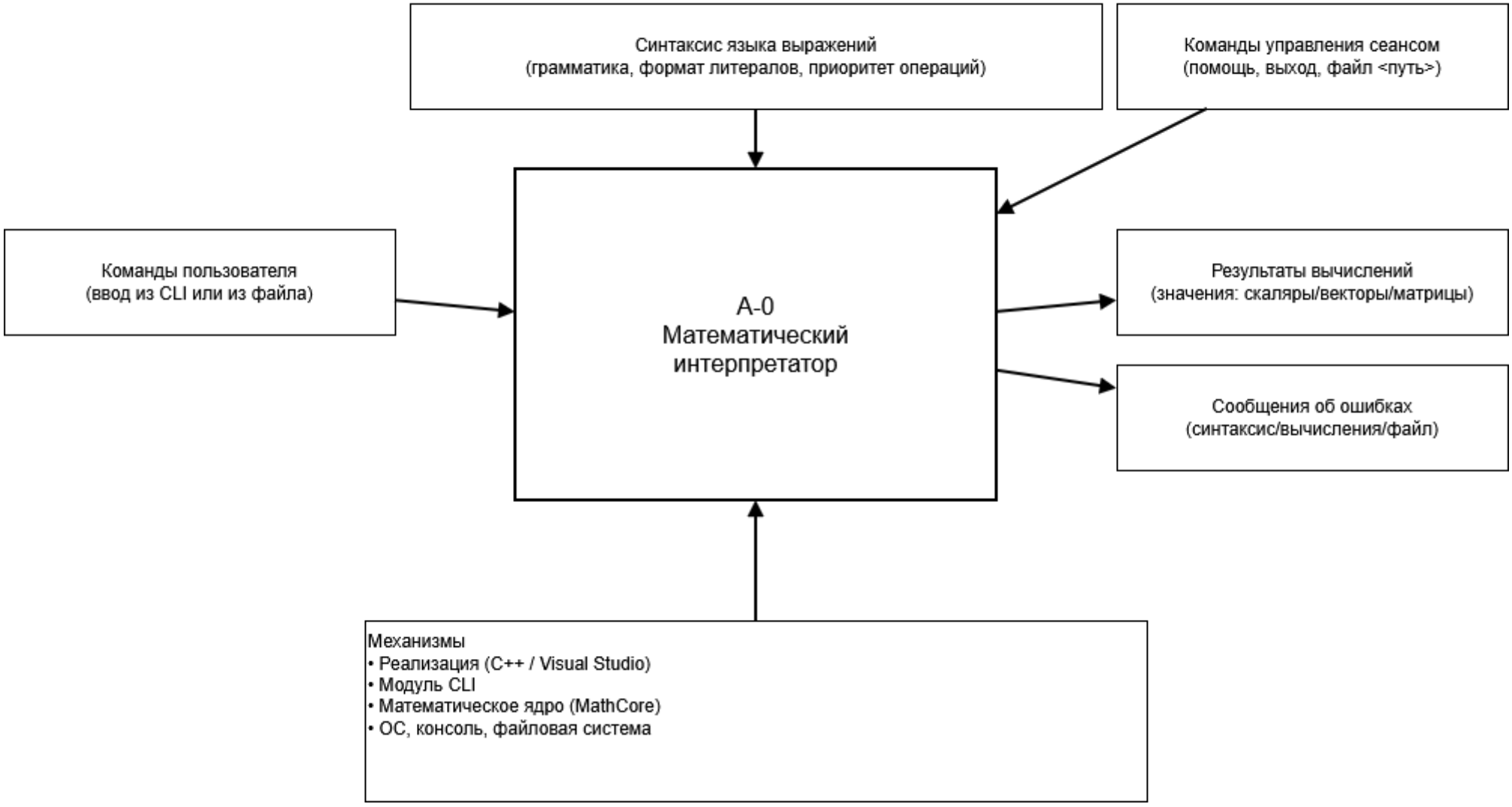
```
    TEST_CLASS(RationalTests) {
public:
    TEST_METHOD(Normalization) {
        auto r = mathcore::RationalValue::create(2, 4);
        Assert::AreEqual(std::string("1/2"), r->toString());
    }

    TEST_METHOD(Addition) {
        auto a = mathcore::RationalValue::create(1, 3);
        auto b = mathcore::RationalValue::create(1, 6);
        auto c = a->add(*b);
        Assert::AreEqual(std::string("1/2"), c->toString());
    }
    };

    TEST_CLASS(InterpreterSmokeTests) {
public:
    TEST_METHOD(SampleFromTask) {
        mathcore::Interpreter it;

        it.executeLine("V1 = [ 1 2 3 ]");
        it.executeLine("M1 = [ 1 0 0; 0 1 0; 0 0 1 ]");
        it.executeLine("V2 = M1 * V1");
        it.executeLine("R = 1 / 3");
        it.executeLine("V3 = V2 * R");
        it.executeLine("M2 = T(M1)");

        auto v3 = it.executeLine("V3");
        Assert::IsTrue(v3.has_value());
        Assert::AreEqual(std::string("[ 1/3 2/3 1 ]"), (*v3)->toString());

        auto m2 = it.executeLine("M2");
        Assert::IsTrue(m2.has_value());
        Assert::AreEqual(std::string("[ 1 0 0; 0 1 0; 0 0 1 ]"), (*m2)-
>toString());
    }
    };

} // namespace MathTests
```

Приложение Б. UML-диаграмма приложения



Приложение В. IDEF0-диаграмма приложения

IDEF0. Контекстная диаграмма A-0: «Математический интерпретатор»



Приложение Г. IDEF0-диаграмма приложения (декомпозиция)