  
МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ  
ВЫСШЕГО ОБРАЗОВАНИЯ  
**«Дальневосточный федеральный университет»**

**ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ  
Департамент программной инженерии и искусственного интеллекта**

**РАЗРАБОТКА ТРАНСЛЯТОРА С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИЙ КОЛЛЕКТИВНОЙ ПРОМЫШЛЕННОЙ РАЗРАБОТКИ**

КУРСОВАЯ РАБОТА  
по дисциплине «Технологии коллективной промышленной разработки информационных систем» по образовательной программе подготовки бакалавров по направлению 09.03.04 «Программная инженерия»

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | |  | | | Выполнили:  студенты группы Б9121-09.03.04 | | | | | | | | |
|  | | | | | | | |  | | |  | | | | | Ваяй М. С. | | | |
|  | | | | | | | |  | | | (подпись) | | | | | Герус П. А. | | | |
|  | | | | | | | |  | | | (подпись) | | | | | Моисеев Д. А. | | | |
|  | | | | | | | |  | | | (подпись) | | | | | Проценко Д. В. | | | |
|  | | | | | | | |  | | | (подпись) | | | | |  | | | |
|  | | | | | | | |  | |  | Руководитель | | | | | | | | | |
|  | | | | | | | |  | |  | Старший преподаватель ДПИиИИ | | | | | | | | | |
|  | | | | | | | |  | |  | ученая степень, должность | | | | | | |  | Иваненко Ю. С. | |
|  | | | | | | | |  | |  | (подпись) | | | | | | |  | (ФИО) | |
|  | | | | | | | | | | | | | | | | | | | | |
| Регистрационный № | | | | | | | |  | |  | Защищен с оценкой | | | | | | | | | |
|  | | | | | | | |  | |  |  | | | | | | | | | |
|  | | | | |  |  | |  | |  | « | |  | » |  |  | | | | 2025 г. |
| (подпись) | | | | |  | (ФИО) | |  | |  |  | | | | | | | | | |
| « | |  | » |  | | | 2025 г. |  | |  |  | | | | | | | | | |

г. Владивосток  
 2025

Оглавление

[Введение 4](#_Toc192115722)

[1 Разработка плана проекта 6](#_Toc192115723)

[2 Разработка регламента проведения инспекции 7](#_Toc192115724)

[2.1 Критерии формальности инспекции 7](#_Toc192115725)

[2.2 Участники инспекции 7](#_Toc192115726)

[2.3 Этапы инспекции 8](#_Toc192115727)

[2.4 Порядок организации инспекции 8](#_Toc192115728)

[2.5 Порядок подготовки и проведения инспекции 9](#_Toc192115729)

[2.6 Перечень статусов и степени важности замечаний 9](#_Toc192115730)

[2.7 Порядок верификации учёта замечаний 9](#_Toc192115731)

[2.8 Метрики, характеризующие эффективность инспекций 9](#_Toc192115732)

[3 Разработка модели состояний задач 10](#_Toc192115733)

[3.1 Перечень возможных состояний задач и их интерпретация 10](#_Toc192115734)

[3.2 Правила создания новой задачи 10](#_Toc192115735)

[3.3 Правила перехода задачи из состояния в состояние 10](#_Toc192115736)

[4 Разработка презентации проекта 12](#_Toc192115737)

[5 Разработка требований к проекту 16](#_Toc192115738)

[5.1 Общие требования 16](#_Toc192115739)

[5.1.1 Функциональные требования 18](#_Toc192115740)

[5.1.2 Нефункциональные требования 18](#_Toc192115741)

[5.2 Требования к подсистеме «Пользовательский интерфейс» 18](#_Toc192115742)

[5.3 Требования к подсистеме «Модуль лексического анализа» 19](#_Toc192115743)

[5.4 Требования к подсистеме «Модуль синтаксического анализа» 19](#_Toc192115744)

[5.5 Требования к подсистеме «Модуль семантического анализа» 19](#_Toc192115745)

[5.5 Требования к подсистеме «Модуль генерации кода» 19](#_Toc192115746)

[6 Разработка архитектуры проекта 20](#_Toc192115747)

[7 Разработка измерений проекта 24](#_Toc192115748)

[7.1 Метрики эффективности процесса разработки 24](#_Toc192115749)

[7.2 Метрики качества программного продукта 25](#_Toc192115750)

[8 Разработка перечня задач проекта 26](#_Toc192115751)

[8.1 Задачи для подсистемы «Пользовательский интерфейс» 26](#_Toc192115752)

[8.2 Задачи для подсистемы «Модуль лексического анализа» 26](#_Toc192115753)

[8.3 Задачи для подсистемы «Модуль синтаксического анализа» 27](#_Toc192115754)

[8.4 Задачи для подсистемы «Модуль семантического анализа» 27](#_Toc192115755)

[8.5 Задачи для подсистемы «Модуль генерации кода» 27](#_Toc192115756)

[9 Разработка рекомендаций по кодированию 28](#_Toc192115757)

[9.1 Рекомендации и требования к оформлению кода 28](#_Toc192115758)

[10 Разработка плана тестирования проекта 29](#_Toc192115759)

[10.1 Тесты для функциональных требований 29](#_Toc192115760)

[10.2 Тесты для нефункциональных требований 29](#_Toc192115761)

[10.3 Тесты для подсистемы «Пользовательский интерфейс» 29](#_Toc192115762)

[10.4 Тесты для подсистемы «Модуль лексического анализа» 30](#_Toc192115763)

[10.5 Тесты для подсистемы «Модуль синтаксического анализа» 31](#_Toc192115764)

[10.6 Тесты для подсистемы «Модуль семантического анализа» 31](#_Toc192115765)

[10.7 Тесты для подсистемы «Модуль генерации кода» 31](#_Toc192115766)

[10.8 Матрица покрытия тестами требований 33](#_Toc192115767)

[11 Тестирование проекта 34](#_Toc192115768)

[11.1 Тесты для функциональных требований 34](#_Toc192115769)

[11.2 Тесты для нефункциональных требований 34](#_Toc192115770)

[11.3 Тесты для подсистемы «Пользовательский интерфейс» 35](#_Toc192115771)

[11.4 Тесты для подсистемы «Модуль лексического анализа» 36](#_Toc192115772)

[11.5 Тесты для подсистемы «Модуль синтаксического анализа» 37](#_Toc192115773)

[11.6 Тесты для подсистемы «Модуль семантического анализа» 38](#_Toc192115774)

[11.7 Тесты для подсистемы «Модуль генерации кода» 39](#_Toc192115775)

# Введение

Промышленная разработка информационных систем включает в себя множество этапов, начиная от разработки плана проекта, заканчивая тестированием проекта для чего, очевидно необходимо множество специалистов различных профилей, а также унифицированные методы коммуникации между ними, с помощью которых можно разделить обязанности членов команды по их специализации.

Исходя из описанного выше необходимо использовать определенные технологии коллективной разработки для повышения эффективности работы в группе и соответствия конечного продукта заявленным требованиям.

В данной курсовой работе рассматривается задача коллективной разработки программного средства «Транслятор из С# в Java» и составление технической документации к данному средству.

Таким образом, целью курсовой работы является разработка программного средства «Транслятор из С# в Java» с использованием подходов коллективной промышленной разработки.

Для достижения поставленной цели необходимо решить следующие задачи:

* Разработать план проекта;
* Разработать регламент проведения инспекции;
* Разработать модель состояний задач;
* Разработать презентацию проекта;
* Разработать требования к проекту;
* Разработать архитектуру проекта;
* Разработать измерения проекта;
* Разработать перечь задач проекта;
* Разработать рекомендации по кодированию;
* Разработать план тестирования проекта;
* Протестировать проект.

# Разработка плана проекта

План проекта — это документ, содержащий подробную информацию о проекте: исполнителях, задачах и сроках. Документ является конечным результатом этапа планирования, утверждается до начала любых работ и становится самым главным и достоверным источником информации о грядущем проекте.

В нашем случае исполнителями являются следующие лица:

* Team Leader — Герус П.А.
* Coder 1 — Проценко Д.В.
* Build Engineer — Ваяй М.С.
* Technical Writer — Моисеев Д.А.

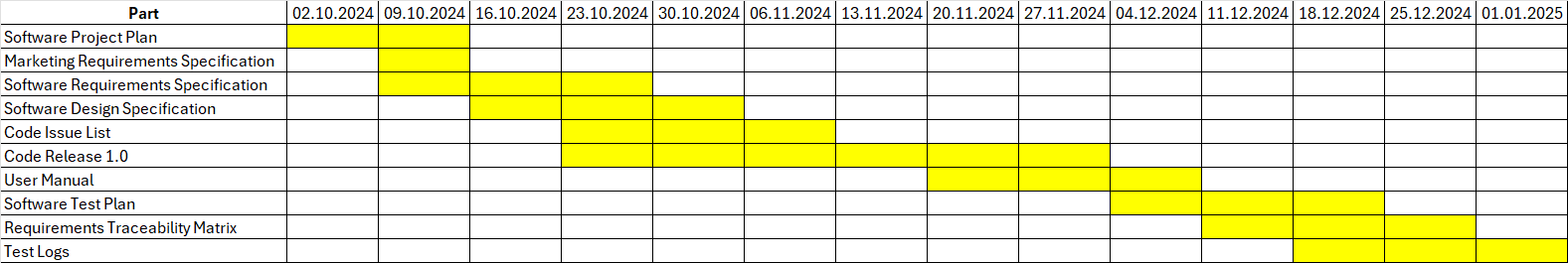
Был разработан перечень задач для выполнения и примерные сроки их реализации рисунке 1.

Рисунок 1 — План проекта

# Разработка регламента проведения инспекции

Верификация рабочих продуктов является неотъемлемой частью процесса по обеспечению их качества. Современной технологией программирования выработаны специальные стандарты, подходы и механизмы проведения верификаций рабочих продуктов в формате так называемых инспекций (peer reviews).

Инспекция — это мероприятие по обеспечению качества рабочих продуктов проектов по разработке ПО и иной деятельности, которая проводится разработчиками, возможно - с участием представителей заказчика. Концептуально инспекция имеет следующие цели:

* Обнаружить ошибки в функциях, логике, содержании или реализации рабочих продуктов на ранних этапах их разработки и предотвратить их наследование;
* Рационально донести замысел или реализацию продукта до всех заинтересованных лиц (через их участие);
* Оптимизировать, оценить или улучшить рабочий продукт.

## Критерии формальности инспекции

Неформальная инспекция проводится:

* В случае изменения участка документа, содержащего не более 5 строк, для текстовых документов;
* В случае изменения не более 5 элементов для документов дизайна.

Формальная инспекция проводится в случае невозможности проведения неформальной инспекции.

## Участники инспекции

Участники могут иметь следующие роли:

1. Автор — участник, внёсший изменения в рабочий продукт. Загружает изменения в систему контроля версий и инициирует инспекцию.
2. Председатель — тимлид (Team Leader), контролирующий процесс инспекции. Назначает инспектора и одобряет внесение ключевых изменений в основную версию продукта.
3. Инспектор — участник, проводящий проверку внесённых изменений. Оставляет замечания и выносит вердикт о внесении изменений в продукт.

В инспекции в обязательном порядке присутствуют два участника, имеющие роли автора и инспектора соответственно. При внесении изменений в дизайн рабочего продукта, необходимо присутствие председателя, который может также выполнять роль инспектора.

## Этапы инспекции

1. Инициация — создание автором запроса на внесение изменений в продукт и назначение инспектора.
2. Подготовка и проведение — анализ изменений и внесение замечаний инспектором.
3. Завершение — вынесение вердикта о внесении изменений в продукт.

## Порядок организации инспекции

Работа над продуктом ведётся в системе контроля версий GIT. Автор изменений оформляет Pull Request (Merge Request) и назначает инспектора, отправляет в рабочий чат в ВК сообщение с упоминанием инспектора.

Столкнувшись с изменением дизайна проекта, инспектор обращается к председателю инспекции в рабочем чате в ВК. По окончании своей работы инспектор одобряет Pull Request или отправляет на доработку автору, уведомляя его в рабочем чате в ВК.

## Порядок подготовки и проведения инспекции

Инспекция должна быть проведена в течение 7 дней с момента её инициации.

После анализа изменений инспектор оставляет в системе контроля версий замечания, обозначая степень их важности. При наличии замечаний, требующих исправлений, работа передаётся автору на доработку. При отсутствии подобных замечаний инспекция считается завершённой, и изменения вступают в силу.

## Перечень статусов и степени важности замечаний

1. Комментарий — рекомендация по улучшению продукта, не требующая обязательных изменений.
2. Ошибка — замечание, сообщающее о необходимости исправления.

## Порядок верификации учёта замечаний

После внесения повторных изменений инспектор просматривает замечания и проверяет соответствующие изменения. По окончании верификации выносится вердикт о внесении изменений в продукт или выносятся новые замечания.

## Метрики, характеризующие эффективность инспекций

Inspection Rate(IR): IR = Размер продукта / Общее время инспектирования.

Стратегическая цель метрики — повысить качество разрабатываемого ПО.

Изучаемый объект метрики — проведение инспекции, измеряемый атрибут — производительность инспектирования.

Единица измерения — <страница, требование, LOC, тест>/ час.

# Разработка модели состояний задач

Каждая задача, являясь отражением делового процесса, проходит определенные состояния. Сначала идет создание задачи, потом идет выполнение работ по задаче, после выполнения задача завершается.

## Перечень возможных состояний задач и их интерпретация

* Backlog — новые и отложенные задачи.
* In progress — задачи в процессе выполнения.
* Testing — задачи в состоянии проверки работоспособности изменений.
* To review — задачи в состоянии проверки изменений на соответствие требованиям проекта.
* Done — выполненные задачи.

## Правила создания новой задачи

Любой участник команды разработки может в любое время создавать задачи в рамках назначенной ему части проекта.

## Правила перехода задачи из состояния в состояние

* В качестве системы отслеживания задач используется Git Hub Projects.
* Созданная задача имеет состояние «Backlog». Team Leader может назначать на задачу исполнителя и инспектора, также участник команды может взять роль исполнителя задачи на себя.
* Как только участник команды разработки готов приступить к задаче, на которую он назначен, он переводит её в состояние «In progress». На этом этапе исполнитель выполняет задачу.
* Когда задача выполнена, она переходит в состояние «Testing». На этом этапе исполнитель проверяет работоспособность изменений на наборе тестов.
* По завершении тестирования, задача переходит в состояние «To review». На этом этапе проходит инспекция изменений. В случае необходимости внесения изменений, состояния задачи меняется на «In progress», иначе на «Done».

# Разработка презентации проекта

Была разработана презентация проекта, состоящая из шести слайдов:

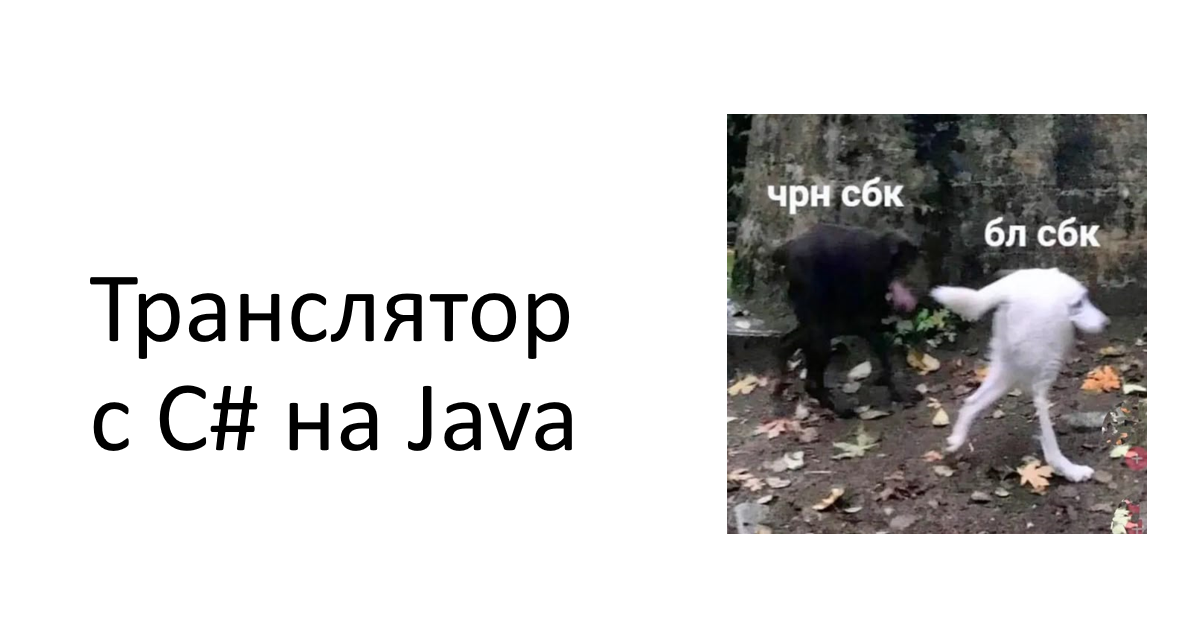
1. Титульный слайд (Рисунок 2);
2. Проблема пользователей (Рисунок 3);
3. Решение проблемы пользователей (Рисунок 4);
4. Команда разработки (Рисунок 5);
5. Принцип работы программного продукта (Рисунок 6);
6. Инструкция по использованию (Рисунок 7).

Рисунок 2 — Титульный слайд



Рисунок 3 — Проблема пользователей

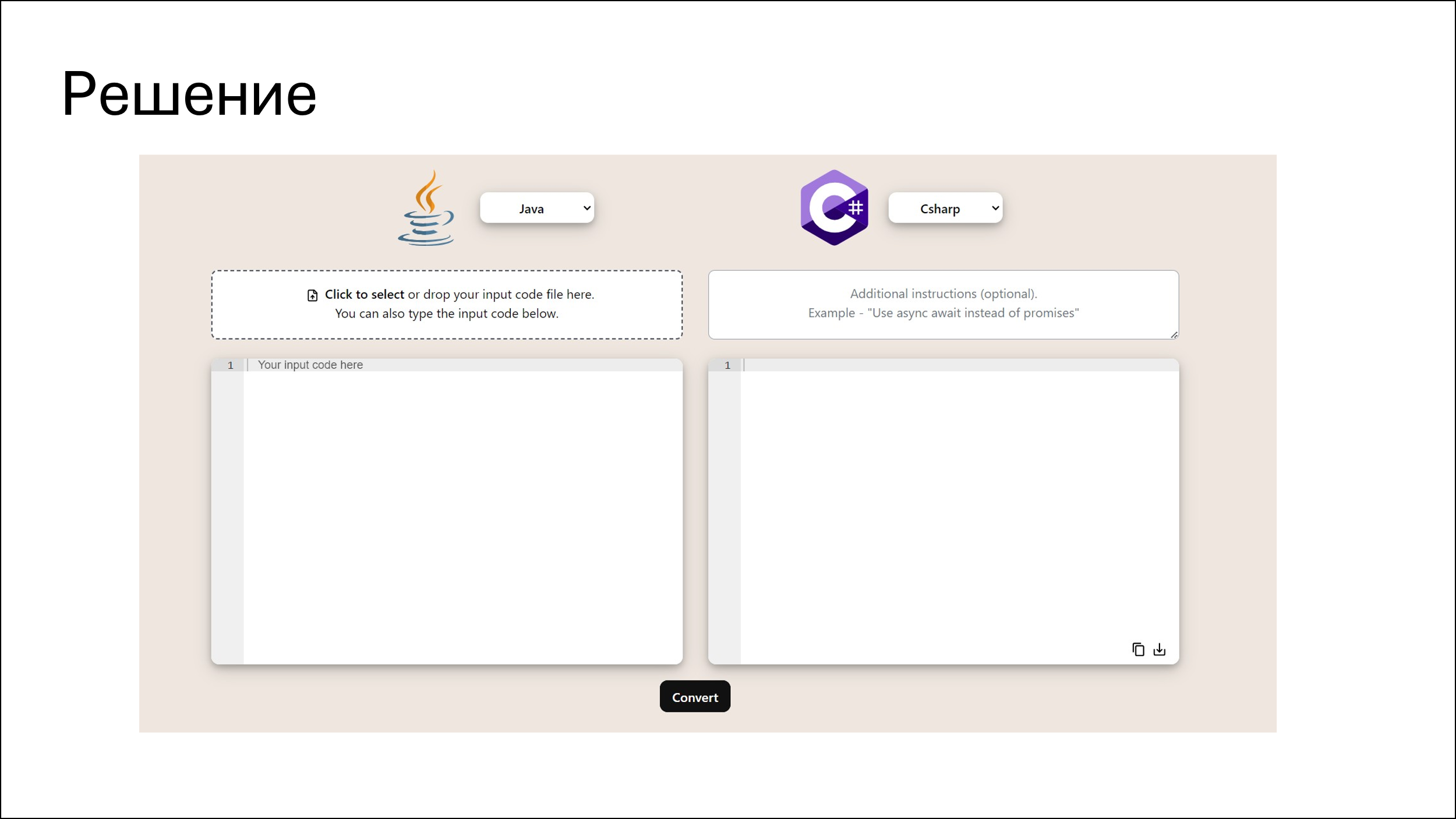


Рисунок 4 — Решение проблемы пользователей

Изображение выглядит как текст, снимок экрана, млекопитающее

Автоматически созданное описаниеРисунок 5 — Команда разработки

Рисунок 6 — Принцип работы программного продукта

Изображение выглядит как текст, кот, млекопитающее, снимок экрана

Автоматически созданное описаниеРисунок 7 — Инструкция по использованию

# Разработка требований к проекту

Программный продукт: транслятор, переводящий подмножество языка C# в эквивалентное подмножество языка Java.

Программный продукт предназначен для автоматического перевода, поданного на вход пользователем кода на языке программирования C# в код на языке Java.

Программный продукт - транслятор должен преобразовывать код, написанный на языке C#, в эквивалентный код на языке Java, сохраняя основную логику и функциональность.

Программный продукт состоит из следующих подсистем:

1. Пользовательский интерфейс.
2. Модуль лексического анализа.
3. Модуль синтаксического анализа.
4. Модуль семантического анализа.
5. Модуль генерации кода.

## Общие требования

Основные требования:

1. Поддержка синтаксиса

Транслятор должен поддерживать ключевые слова, операторы, типы данных и синтаксис C#, включая:

* Базовые типы данных (int, double, string, bool);
* Классы, структуры, интерфейсы, перечисления;
* Методы, свойства, поля;
* Операторы (арифметические, сравнения, логические);
* Условные операторы (if, else, switch);
* Циклы (for, while, do-while);
* Исключения (try-catch);
* Объектно-ориентированное программирование (наследование, полиморфизм).

1. Преобразование типов данных

Транслятор должен уметь преобразовывать типы данных C# в соответствующие типы данных Java, например:

* int в int;
* double в double;
* string в String;
* bool в Boolean;
* List<T> в ArrayList<T>;
* Dictionary<K, V> в HashMap<K, V>;
* Транслятор должен учитывать нюансы преобразования, например, для DateTime использовать Date или Calendar в Java.

3. Преобразование логики

Транслятор должен уметь преобразовывать логику кода, например:

* Вызовы методов, конструкторов, статических методов;
* Условные операторы и циклы;
* Обработку исключений;
* Использование стандартных библиотек .NET (например, [System.IO](https://vk.com/away.php?to=http%3A%2F%2FSystem.IO&cc_key=)) в соответствующие классы Java (например, [java.io](https://vk.com/away.php?to=http%3A%2F%2Fjava.io&cc_key=)).

4. Создание исходного кода Java

Транслятор должен генерировать корректный исходный код Java, который может быть скомпилирован и запущен.

5. Обработка ошибок

* Транслятор должен уметь выявлять и сообщать о синтаксических ошибках в исходном коде C#;
* Транслятор должен уметь выявлять и сообщать о несовместимости между типами данных C# и Java;
* Транслятор должен уметь предоставлять исчерпывающие сообщения об ошибках с указанием местоположения ошибки в исходном коде C#;
* Транслятор не должен быть просто строковым преобразователем. Он должен учитывать контекст, семантику кода C# и обеспечивать его корректное преобразование в Java;
* Транслятор должен быть как можно более полным, чтобы обеспечить максимальную совместимость с исходным кодом C#.  
  Транслятор должен быть гибким и расширяемым, чтобы его можно было легко адаптировать к новым функциям и обновлениям языка C# и Java.

### Функциональные требования

Требование FU\_001 Продукт должен транслировать код с языка С# на Java.

Требование FU\_002 Продукт должен формировать код на языке Java.

### Нефункциональные требования

Требование UF\_001 Продукт должен быть написан на языке C# версии.

Требование UF\_002 Контроль версий продукта должен вестись в GitHub.

Требование UF\_003 Для каждой системы должны быть подготовлены unit-тесты.

## Требования к подсистеме «Пользовательский интерфейс»

Требование UI\_001 На сайте должны быть и исправно функционировать следующие элементы интерфейса:

1. кнопка добавления файла с исходным кодом на C#;
2. поле для отображения исходного кода;
3. поле для отображения логов и прочих ошибок;
4. поле для отображения транслированного кода на java;
5. кнопка отправки запроса на перевод;

Требование UI\_002 Если пользователь ввёл некорректные данные, то по нажатии кнопки отправки запроса на трансляцию (см. UI\_001) пользователь должен получать сообщение с описанием ошибки.

Требование UI\_003 Система должна принимать файлы с расширением .cs при добавлении файла по нажатии кнопки (см. UI\_001).

## Требования к подсистеме «Модуль лексического анализа»

LX\_001: модуль должен получать на вход текст программы на C# и разбивать его на токены, сохраняя результат в формате JSON.

LX\_002: каждый токен должен содержать тип лексемы, значение, номер строки и столбца.

LX\_003: в случае нахождения некорректной лексемы, токен помечается как ошибка, но работа модуля продолжается.

LX\_004: при обнаружении ошибки необходимо сформировать сообщение и передать его до интерфейса пользователя.

## Требования к подсистеме «Модуль синтаксического анализа»

SY\_001: при нахождении синтаксической ошибки формируется сообщение с её описанием, которое передаётся в интерфейс пользователя.

## 5.5 Требования к подсистеме «Модуль семантического анализа»

SE\_001: при обнаружении семантической ошибки формируется сообщение с описанием и передаётся в интерфейс пользователя.

## Требования к подсистеме «Модуль генерации кода»

Требование CG\_001 Каждому токену языка С# и каждой конструкции языка должна быть выделена единственным образом конструкция языка Java.

# Разработка архитектуры проекта

Архитектура программного обеспечения относится к фундаментальным структурам программной системы и дисциплине создания таких структур и систем. Каждая структура включает элементы программного обеспечения, отношения между ними, а также свойства как элементов, так и отношений. Архитектура программной системы — это метафора, аналогичная архитектуре здания. Он функционирует как план для системы и проекта разработки, в котором излагаются задачи, которые должны быть выполнены командами разработчиков.

Одним из способов представления архитектуры проекта является диаграмма потоков данных.

Диаграммы потоков данных (Data Flow Diagrams — DFD) представляют собой иерархию функциональных процессов, связанных потоками данных. Цель такого представления — продемонстрировать, как каждый процесс преобразует свои входные данные в выходные, а также выявить отношения между этими процессами.

Была разработана диаграмма потоков данных между подсистемами разрабатываемого транслятора рисунке 8. На вход интерфейс принимает файл, содержащий код на языке Java. Далее полученный код передается лексическому анализатору, который преобразует валидный код в массив токенов. Полученный массив токенов предоставляется синтаксическому анализатору для проверки синтаксиса входного языка. После проверки строится дерево разбора, которое в свою очередь уже будет обрабатывать сематический анализатор. Обработанное семантическим анализатором дерево передается генератору кода, который, в свою очередь, отдает интерфейсу код на языке C++.

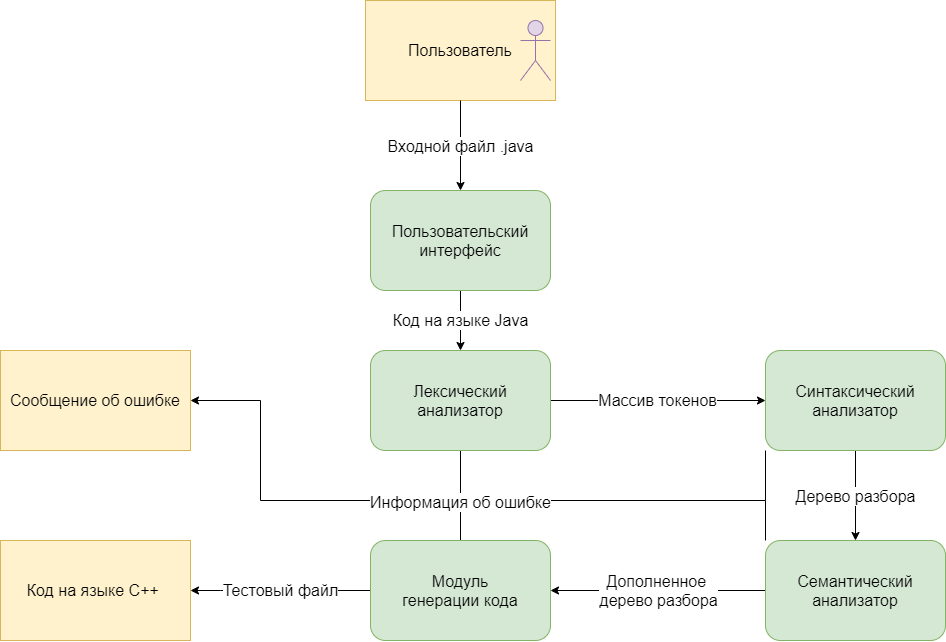


Рисунок 8 — Диаграмма потоков данных

Также составим диаграмму, описывающую сценарий взаимодействия пользователя с нашей системой, то есть «Use case» диаграмму на рисунке 9.

Сценарии использования транслятора:

1. Загрузить файл с исходным кодом.  
   Пользователь предоставляет файл с кодом на языке C# для анализа и преобразования.
2. Просмотреть ошибки в исходном коде.  
   Пользователь видит сообщения об ошибках в исходном коде (лексических, синтаксических, семантических) и их местоположение.
3. Исправить исходный код.  
   Пользователь исправляет ошибки в коде (вне приложения) и повторно загружает файл.
4. Запустить трансляцию.  
   Пользователь инициирует процесс трансляции кода из C# в Java.
5. Просмотреть сгенерированный код.  
   Пользователь может просмотреть результирующий код на Java в интерфейсе.
6. Сохранить сгенерированный код.  
   Пользователь сохраняет преобразованный файл в указанное место.

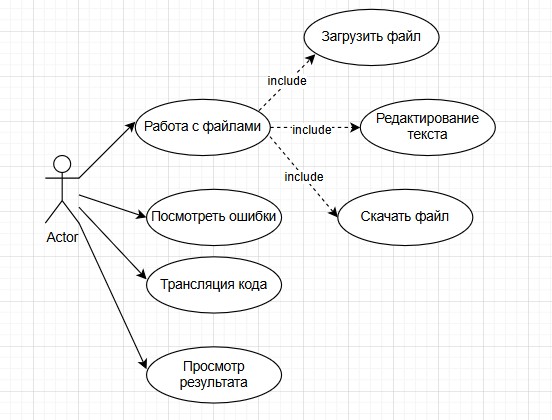


Рисунок 9 — “Use Case” диаграмма

Архитектурно-контекстная диаграмма рисунке 10 отражает программные и информационные компоненты системы, которые используются в программном средстве. Программные компоненты изображаются прямоугольниками. Стрелки являются связями, между компонентами.

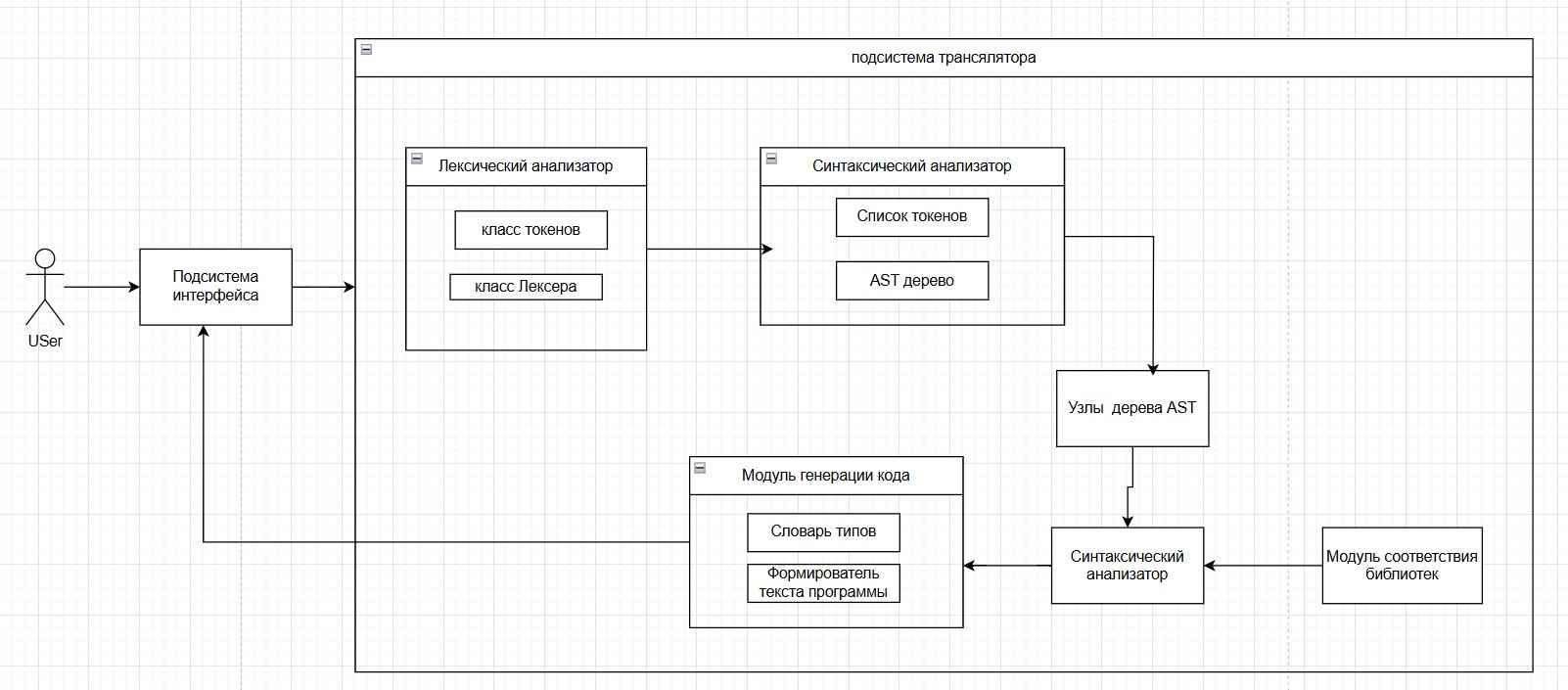


Рисунок 10 — Контекстная диаграмма

# Разработка измерений проекта

Контроль за производственным процессом и его результатами является ключевым видом деятельности на современном предприятии, производящем программное обеспечение на заказ. В силу специфики такого продукта, как программное обеспечение, для оценки эффективности процесса и качества конечного продукта применяются особые методы. Комплекс мероприятий, направленных на количественную оценку эффективности работы компании, называется программой измерений компании. Программа измерений выполняется как в рамках отдельных проектов, осуществляемых компанией, так и в рамках определённых видов деятельности компании. На крупных предприятиях программа измерений осуществляется специальным отделом по обеспечению качества (SQA — Software Quality Assurance team).

## Метрики эффективности процесса разработки

1. Problem Resolution Rate (PRR)

PRR = Количество дней на обработку задачи.

Стратегическая цель метрики — сократить сроки разработки модулей ПО.

Изучаемый объект метрики — задача.

Измеряемый атрибут — время обработки.

Единица измерения — день.

2. Faults Screening (FS)

FS = (Общее количество ошибок − Число ошибок, найденных до бета-тестирования) × 100% / Общее количество ошибок.

Стратегическая цель метрики — повысить качество разрабатываемого ПО.

Изучаемый объект метрики — проект.

Измеряемый атрибут — эффективность обнаружения дефектов.

Единица измерения — %.

## Метрики качества программного продукта

1. Beta Testing Faults (BTF)

BTF = Число обнаруженных на этапе бета-тестирования ошибок / LOC.

Стратегическая цель метрики — повысить качество разрабатываемого ПО.

Изучаемый объект метрики — продукт.

Измеряемый атрибут — плотность неполадок.

Единица измерения — неполадка / LOC.

# Разработка перечня задач проекта

[UF-1] Придумать как разнести разделить проект на модули.

Создать модули для подсистем.

[UF-2] Подготовить пайплайн для тестов.

Подготовить файлы с кодом на языке С#, скрипты для автоматического запуска тестов программной системы на них.

## Задачи для подсистемы «Пользовательский интерфейс»

[UI-1] Разработать прототип интерфейса.

[UI-1] Добавить основные элементы интерфейса.

Необходимо добавить на сайт поля блоки с кодом, инструменты загрузки файла и кнопку начала трансляции (см. UI\_001).

[UI-2] Вывод однородных ошибок.

Необходимо собрать ошибки всех модулей и выводить их на экран пользователя в одном формате в случае их возникновения (см. UI\_002).

[UI-3] Загрузка файла для трансляции.

Код на C# должен быть закружен в систему в виде файла, в интерфейсе нужен соответствующий функционал (см. UI\_003).

## Задачи для подсистемы «Модуль лексического анализа»

[LX-1] Разработать начальную версию.

Вход: любой код на языке C#.

Выход: JSON-файл с токенами.

[LX-2] Обеспечить вывод ошибок.

Реализация требования LX\_004.

[LX-3] Написать и проверить тесты для лексера.

Подготовить начальный набор тестов для модуля.

## Задачи для подсистемы «Модуль синтаксического анализа»

[SY-1] Разработать начальную версию.

Вход: JSON-файл с токенами.

Выход: дерево разбора.

[SY-2] Обеспечить вывод ошибок.

Реализация требования SY\_002.

[SY-3] Написать и проверить тесты для синтаксического анализатора.

Подготовить начальный набор тестов для модуля.

## Задачи для подсистемы «Модуль семантического анализа»

[SE-1] Начальная версия семантического анализатора.

Вход: дерево разбора.

Выход: дерево разбора и сообщение о семантической ошибке, если таковая имеется.

[SE-2] Обеспечить вывод ошибок.

[SE-3] Написать и проверить тесты для семантического анализатора.

Подготовить начальный набор тестов для модуля.

## Задачи для подсистемы «Модуль генерации кода»

[CG-1] Начальная версия генератора кода.

Вход: дерево разбора.

Выход: текст на языке Java (см. CG\_002).

[CG-2] Написать и проверить тесты для сгенерированного кода.

Подготовить начальный набор тестов для модуля.

# Разработка рекомендаций по кодированию

Для создания качественного кода на любом языке программирования, обладающего таким свойствами, как удобочитаемость (readability) и понятность (understandability), необходимо следовать хорошо определённым стандартам и руководствам. Особенно это актуально при коллективной разработке программ. Любой стандарт кодирования призван определить набор правил, которые способствуют разработке более единообразного кода и минимизации числа общераспространенных ошибок в нем, не ущемляя при этом права разработчика на творчество.

В основе рекомендаций — руководство по оформлению кода Microsoft C# Coding Conventions. Все правила должны соблюдаться в проекте. Ниже перечислены самые важные рекомендации и требования по оформлению кода.

## Рекомендации и требования к оформлению кода

* **CamelCase** для имён переменных и параметров методов;
* **PascalCase** для имён классов, методов и свойств;
* **SCREAMING\_SNAKE\_CASE** для констант;
* **Имена методов должны начинаться с глагола** и отражать их функциональность;
* **Использование фигурных скобок** { } даже для однострочных операторов;
* Не использовать **"magic numbers"** (неименованные константы);
* Классы и методы должны быть снабжены **XML-документацией** для автоматической генерации описаний.

# Разработка плана тестирования проекта

## Тесты для функциональных требований

Тест TEST\_FU\_001

Тестируемые требования: FU\_001

Код, написанный на C# успешно транслируется на язык Java.

Тест TEST\_FU\_002

Тестируемые требования: FU\_002

Успешно транслированный код сформирован на языке Java.

## Тесты для нефункциональных требований

Тест TEST\_UF\_001

Тестируемые требования: UF\_001

Проверка исходного кода продукта показывает, что код написан на языке C#

Тест TEST\_UF\_002

Тестируемые требования: UF\_002

Репозиторий продукта существует на GitHub.

Тест TEST\_UF\_003

Тестируемые требования: UF\_003

Unit-тесты присутствуют для всех модулей

## Тесты для подсистемы «Пользовательский интерфейс»

Тест TEST\_UI\_001

Тестируемые требования: UI\_001

В окне приложения расположены и исправно функционируют следующие элементы интерфейса:

* кнопка добавления файла с исходным кодом на С#,
* кнопка сохранения файла с исходным кодом на Java,
* кнопка очистки содержимого поля ввода кода на С#,
* checkbox удаления старых логов,
* поле для отображения исходного кода,
* поле для отображения транслированного кода на Java,
* кнопка отправки запроса на перевод,
* поле для отображения логов.

Тест TEST\_UI\_002

Тестируемые требования: UI\_002

В зависимости от типа ошибки при вводе некорректного кода вывод соответствующего сообщения.

Тест TEST\_UI\_003

Тестируемые требования: UI\_003

Приложение должен принимать на вход файл только с расширением .cs

## Тесты для подсистемы «Модуль лексического анализа»

Тест TEST\_LX\_001

Тестируемые требования: LX\_001, LX\_002

Проверка разбиения кода C# на токены, содержащие всю описанную в требованиях информацию, и их соответствие формату JSON. На вход модулю подаётся строка, содержащая код на языке C#. Производится попытка разбиения на токены и сохранение их в формате JSON.

Тест TEST\_LX\_002

Тестируемые требования: LX\_003, LX\_004

Проверка работоспособности в случае нахождения лексемы, не попадающей в список выделенных, токен помечается ошибкой. При этом модуль не прекращает работу и формирует сообщение с описанием ошибки и доставляет его до интерфейса пользователя.

## Тесты для подсистемы «Модуль синтаксического анализа»

Тест TEST\_SY\_001

Тестируемые требования: SY\_001

Проверка корректности сообщения при ошибке с недостающей точкой с запятой. На вход подаётся файл, в котором не хватает точки с запятой в строке.

Тест TEST\_SY\_002

Тестируемые требования: SY\_001

Проверка корректности сообщения при ошибке с отсутствующей в грамматике конструкцией языка. На вход подаётся файл, в котором не хватает скобок в блоке условного оператора.

## Тесты для подсистемы «Модуль семантического анализа»

Тест TEST\_SE\_001

Тестируемые требования: SE\_001

Проверка корректности сообщения при ошибке с использованием оператора сравнения в условной конструкции if (x = y).

Тест TEST\_SE\_002

Тестируемые требования: SE\_001

Проверка корректности трансляции при попытке с присвоением переменной типа int значения типа double.

## Тесты для подсистемы «Модуль генерации кода»

Тест TEST\_CG\_001

Тестируемые требования: CG\_001

Проверка подсистемы на единообразие выводимого результата для всех токенов и конструкций языка, присутствующих в грамматике. На вход подаётся файл с кодом, в котором создаются две переменные типа int: а = 5 и b = 2; а также функция formula, получающая на вход два параметра типа int a и b и возвращающая результат формулы *(3 + (a - 1)) \* b / 2*. Также в файле с кодом содержится цикл for для переменной типа int i от 0 до 5, внутри которого с помощью функции Console.write выводится текущее значение i и цикл while с условием i < 5, внутри которого сначала выводится текущее значение переменной i с помощью функции Console.write, а затем значение i инкрементируется; кроме этого в файле содержится две переменные типа float: а = 1.5 и b = 2.0. Затем в условном операторе if проверяется условие a < b && b == 2.0, в блоке if с помощью функции Console.write выводится переменная a, в блоке else выводится переменная b. Помимо этого в функции main создаётся по одной переменной следующих типов: double и char; а также две переменные типа boolean, одной из которых присваивается значение true, другой - значение false. Помимо конструкции if проверяются также конструкии if с операторами сравнения >, >=, <= и !=, а также с логическими операторами ! и ||. Помимо этого, проверяются все операторы присваивания: сначала создаётся переменная типа int k = 1, после этого этой переменной присваиваются значения с помощью операторов +=, -=, \*=, /=, %=, а также проверяются операторы ++ и --. Для проверки всех математических операторов создаётся переменная типа float f, которой присваивается значение формулы *(1.0 + 0.5 - 0.25) \* 2.0 / (5 % 2).*

## Матрица покрытия тестами требований

Была построена матрица покрытия тестами требований, представлена на рисунке 11.

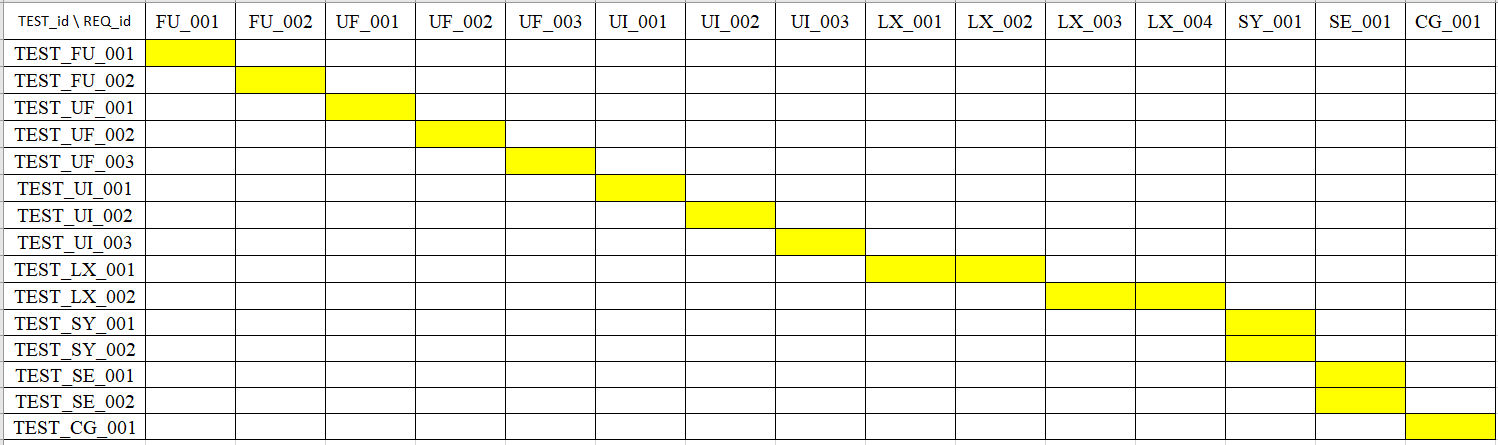


Рисунок 11 — Матрица покрытия тестами требований

# Тестирование проекта

## Тесты для функциональных требований

**Тест TEST\_FU\_001**

Тестируемые требования: FU\_001

Код, написанный на C# успешно транслируется на язык Java.

Тестируемая версия продукта: 1.0.1

Ожидаемый результат: Код, написанный на C# успешно транслируется на язык Java.

Видимый результат: написанный на C# успешно транслируется на язык Java.

Резюме: Тест пройден.

**Тест TEST\_FU\_002**

Тестируемые требования: FU\_002

Успешно транслированный код сформирован на языке Java.

Тестируемая версия продукта: 1.0.1

Ожидаемый результат: Код на языке Java.

Видимый результат: Код на языке Java.

Резюме: Тест пройден.

## Тесты для нефункциональных требований

**Тест TEST\_UF\_001**

Тестируемые требования: UF\_001

Проверка исходного кода продукта показывает, что код написан на языке C#.

Тестируемая версия продукта: 1.0.1

Ожидаемый результат: Исходный код написан на C#.

Видимый результат: Исходный код написан на C#.

Резюме: Тест пройден.

**Тест TEST\_UF\_002**

Тестируемые требования: UF\_002

Репозиторий продукта существует на GitHub.

Тестируемая версия продукта: 1.0.1

Ожидаемый результат: Репозиторий продукта существует на GitHub.

Видимый результат: Репозиторий продукта существует на GitHub.

Резюме: Тест пройден.

**Тест TEST\_UF\_003**

Тестируемые требования: UF\_003

Unit-тесты присутствуют для всех модулей.

Тестируемая версия продукта: 1.0.1

Ожидаемый результат: Unit-тесты присутствуют для всех модулей.

Видимый результат: Unit-тесты присутствуют для всех модулей.

Резюме: Тест пройден.

## Тесты для подсистемы «Пользовательский интерфейс»

**Тест TEST\_UI\_001**

Тестируемые требования: UI\_001

В окне приложения расположены и исправно функционируют следующие элементы интерфейса:

* кнопка добавления файла с исходным кодом на С#,
* кнопка сохранения файла с исходным кодом на Java,
* кнопка очистки содержимого поля ввода кода на С#,
* checkbox удаления старых логов,
* поле для отображения исходного кода,
* поле для отображения транслированного кода на Java,
* кнопка отправки запроса на перевод,
* поле для отображения логов.

Тестируемая версия продукта: 1.0.1

Ожидаемый результат: Все элементы интерфейса присутствуют в приложении.

Видимый результат: Все элементы интерфейса присутствуют в приложении.

Резюме: Тест пройден.

**Тест TEST\_UI\_002**

Тестируемые требования: UI\_002

В зависимости от типа ошибки при вводе некорректного кода вывод соответствующего сообщения.

Тестируемая версия продукта: 1.0.1

Ожидаемый результат: Вывод исходного кода и сообщения об ошибке, которое содержит описание ошибки, соответствующей определённому виду.

Видимый результат: Вывод исходного кода и сообщения об ошибке, которое содержит описание ошибки, соответствующей определённому виду. Резюме: Тест пройден.

**Тест TEST\_UI\_003**

Тестируемые требования: UI\_003

Приложение должен принимать на вход файл только с расширением .cs

Тестируемая версия продукта: 1.0.1

Ожидаемый результат: возможность прикрепить только файлы с расширением .cs

Видимый результат: возможность прикрепить только файлы с расширением .cs

Резюме: Тест пройден.

## Тесты для подсистемы «Модуль лексического анализа»

**Тест TEST\_LX\_001**

Тестируемые требования: LX\_001, LX\_002

Проверка разбиения кода C# на токены, содержащие всю описанную в требованиях информацию, и их соответствие формату JSON. На вход модулю подаётся строка, содержащая код на языке C#. Производится попытка разбиения на токены и сохранение их в формате JSON.

Тестируемая версия продукта: 1.0.1

Ожидаемый результат: Массив токенов формата JSON.

Видимый результат: Массив токенов формата JSON.

Резюме: Тест пройден.

**Тест TEST\_LX\_002**

Тестируемые требования: LX\_003, LX\_004

Проверка работоспособности в случае нахождения лексемы, не попадающей в список выделенных, токен помечается ошибкой. При этом модуль не прекращает работу и формирует сообщение с описанием ошибки и доставляет его до интерфейса пользователя.

Тестируемая версия продукта: 1.0.1

Ожидаемый результат: Модуль ловит лексему, не прекращает работу и выводит сообщение, которое показывается пользователю.

Видимый результат: Модуль ловит лексему, не прекращает работу и выводит сообщение, которое показывается пользователю.

Резюме: Тест пройден.

## Тесты для подсистемы «Модуль синтаксического анализа»

**Тест TEST\_SY\_001**

Тестируемые требования: SY\_001

Проверка корректности сообщения при ошибке с недостающей точкой с запятой. На вход подаётся файл, в котором не хватает точки с запятой в строке.

Тестируемая версия продукта: 1.0.1

Ожидаемый результат: Сообщение о синтаксической ошибке в строке.

Видимый результат: Сообщение о синтаксической ошибке в строке.

Резюме: Тест пройден.

**Тест TEST\_SY\_002**

Тестируемые требования: SY\_001

Проверка корректности сообщения при ошибке с недостающей точкой с запятой. На вход подаётся файл, в котором не хватает точки с запятой в строке.

Тестируемая версия продукта: 1.0.1

Ожидаемый результат: Сообщение о синтаксической ошибке в строке.

Видимый результат: Сообщение о синтаксической ошибке в строке.

Резюме: Тест пройден.

## Тесты для подсистемы «Модуль семантического анализа»

**Тест TEST\_SE\_001**

Тестируемые требования: SE\_001

Проверка корректности сообщения при ошибке с использованием оператора сравнения в условной конструкции if (x === y).

Тестируемая версия продукта: 1.0.1

Ожидаемый результат: Сообщение об семантической ошибке.

Видимый результат: [ERROR]: оператор требует 2 аргумента. Строка: 11, столбец: 24

Резюме: Тест пройден.

**Тест TEST\_SE\_002**

Тестируемые требования: SE\_001

Проверка корректности сообщения при ошибке с присвоением переменной типа int значения типа double.

Тестируемая версия продукта: 1.0.1

Ожидаемый результат: Неявное преобразование данных и успешная трансляция кода.

Видимый результат: Неявное преобразование данных и успешная трансляция кода.

Резюме: Тест пройден.

## Тесты для подсистемы «Модуль генерации кода»

**Тест TEST\_CG\_001**

Тестируемые требования: CG\_001

Проверка подсистемы на единообразие выводимого результата для всех токенов и конструкций языка, присутствующих в грамматике. На вход подаётся файл с кодом, в котором создаются две переменные типа int: а = 5 и b = 2; а также функция formula, получающая на вход два параметра типа int a и b и возвращающая результат формулы *(3 + (a - 1)) \* b / 2*. Также в файле с кодом содержится цикл for для переменной типа int i от 0 до 5, внутри которого с помощью функции Console.write выводится текущее значение i и цикл while с условием i < 5, внутри которого сначала выводится текущее значение переменной i с помощью функции Console.write, а затем значение i инкрементируется; кроме этого в файле содержится две переменные типа float: а = 1.5 и b = 2.0. Затем в условном операторе if проверяется условие a < b && b == 2.0, в блоке if с помощью функции Console.write выводится переменная a, в блоке else выводится переменная b. Помимо этого в функции main создаётся по одной переменной следующих типов: double и char; а также две переменные типа boolean, одной из которых присваивается значение true, другой - значение false. Помимо конструкции if проверяются также конструкии if с операторами сравнения >, >=, <= и !=, а также с логическими операторами ! и ||. Помимо этого, проверяются все операторы присваивания: сначала создаётся переменная типа int k = 1, после этого этой переменной присваиваются значения с помощью операторов +=, -=, \*=, /=, %=, а также проверяются операторы ++ и --. Для проверки всех математических операторов создаётся переменная типа float f, которой присваивается значение формулы *(1.0 + 0.5 - 0.25) \* 2.0 / (5 % 2)*.

Тестируемая версия продукта: 1.0.1

Ожидаемый результат: код на Java с единообразным результатом для всех токенов и конструкций.

Видимый результат: код на Java с единообразным результатом для всех токенов и конструкций.

Резюме: Тест пройден.