

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет
Вычислительной техники

Кафедра
МОиПЭВМ

Направление подготовки 09.03.04 «Программная инженерия»

Направленность Программная инженерия

БАКАЛАВРСКАЯ РАБОТА

На тему

**Серверная часть комплекса программных
средств для автоматизации написания игровых
скриптов**

Студент

(подпись, дата)

Резяков Ильнур Равильевич
(ФИО полностью)

Руководитель

(подпись, дата)

Козлов А. Ю.
(фамилия, инициалы)

Нормоконтролёр

(подпись, дата)

Такташкин Д. В.
(фамилия, инициалы)

Работа допущена к защите (протокол заседания кафедры от 03.06.2026 г. № 12)

Заведующий кафедрой

(подпись)

Козлов А.Ю.
(фамилия, инициалы)

Работа защищена с отметкой _____ (протокол заседания ГЭК от _____ № _____)

Секретарь ГЭК

(подпись)

Попова Н.А.
(фамилия, инициалы)

Пенза, 2026

Федеральное государственное бюджетное образовательное учреждение
высшего образования
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет
Вычислительной техники

Кафедра
МО и ПЭВМ

«Утверждаю»
Заведующий кафедрой
А.Ю. Козлов
«26» декабря 2025 г

**ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
БАКАЛАВРА**

1. Студент Резяков Ильнур Равильевич гр. 22ВП1 факультета ВТ
<Фамилия, имя, отчество полностью>

2. Тема работы Серверная часть комплекса программных средств для автоматизации написания игровых скриптов

Тема утверждена приказом ПГУ №1583/о от "30" декабря 2025 г.

3. Руководитель работы д.т.н., профессор, заведующий кафедрой «МО и ПЭВМ» Козлов А.Ю.

4. Задание на работу (назначение разработки, исходные данные и т.п.)

1) *Назначение: Программное обеспечение автоматизации написания скриптов*

2) *Среда выполнения: операционная система Windows 11*

Языковые средства разработки: C#, UML

Инструментальные средства: MS Visual Studio 2022, Visual Paradigm

Технологии разработки: ООП, принципы SOLID, шаблоны проектирования.

3) *Основные функции приложения:*

а) автоматическая генерация игровых скриптов на основе введенных данных;

б) извлечение игровых метаданных;

в) поиск и навигация по данным;

г) сохранение сгенерированных скриптов в файл и итеративная запись;

5. Перечень подлежащих разработке вопросов:

1) *Анализ предметной области и требований к системе;*

2) *Проектирование программных средств;*

3) *Реализация программных средств;*

4) Тестирование программных средств;

5) Оформление пояснительной записки.

5. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Анализ предметной области и требований к системе	10%	06.04.2026 – 14.04.2026	
2	Проектирование программных средств	20%	15.04.2026 – 25.04.2026	
3	Реализация программных средств	30%	26.04.2026 – 12.05.2026	
4	Тестирование программных средств	20%	13.05.2026 – 22.05.2026	
5	Оформление пояснительной записки	20%	23.05.2026 – 03.06.2026	

Дата выдачи задания «20» декабря 2025г.

Руководитель бакалаврской работы _____ 26.12.2025 г.
(подпись, дата)

Задание к исполнению принял студент _____ 26.12.2025 г.
(подпись, дата)

Работу к защите допустить
Декан ФВТ _____ Л.Р. Фионова
(подпись, дата)

Реферат

Пояснительная записка содержит 98 листов, 40 рисунков, 11 таблиц и 22 использованных источника.

АВТОМАТИЗАЦИЯ СКРИПТОВ, ИТЕРАТИВНАЯ РАБОТА, ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ.

Объект исследования – программное обеспечение автоматизации игровых скриптов.

Цель выпускной квалификационной работы – спроектировать и разработать программные средства, позволяющие пользователю с помощью ввода данных получать готовые, рабочие игровые скрипты для изменения игры.

Технологии разработки – разработка всех средств и визуализации производится в среде разработки Microsoft Visual Studio 2022 на объектно-ориентированном языке программирования C#. Развертывание основного приложения осуществляется на операционных системах семейства Windows версии 10 и выше. В качестве среды исполнения программных средств выступает персональный компьютер пользователя.

Результаты работы – в ходе выполнения выпускной квалификационной работы были спроектированы и разработаны программные средства, позволяющие автоматизировать написание игровых скриптов.

					ПГУ 09.03.04 – 08ВП221.14 ПЗ			
Изм.	Лист	№ докум.	Подпись	Дата				
Разраб.		Резяков И. Р.			Серверная часть комплекса программных средств для автоматизации написания игровых скриптов. Пояснительная записка	Лит.	Лист	Листов
Пров.		Козлов А.Ю.					4	98
						Группа 22ВП1		
Н. Контр.		Такташкин Д.В						
Утв.		Козлов А.Ю.						

Содержание

Введение.....	6
1 Анализ предметной области и постановка задачи.....	7
1.1 Анализ предметной области	7
1.2 Архитектурно-техническая реализация системы автоматизации.....	8
1.2.1 Модульная организация и принципы проектирования.....	8
1.2.2 Анализа кода на основе абстрактного синтаксического дерева	8
1.2.3 Система экстракторов и реестр обработчиков.....	9
1.2.4 Стратегии генерации и сохранения кода.....	10
1.2.5 Интеграция с внешними источниками данных.....	10
1.3 Анализ программного обеспечения аналогичного назначения	10
1.4 Постановка задачи на разработку программных средств.....	14
1.5 Анализ функциональных требований.....	15
1.6 Анализ нефункциональных требований.....	18
1.7 Планирование разработки и оценка бюджета.....	18
2 Проектирование программного обеспечения для автоматизации генерации скриптов	25
2.1 Архитектура приложения.....	25
2.2 Проектирование моделей конфигураций и данных	28
3 Реализация программных средств.....	34
3.1 Синтаксический анализ кода	34
3.2 Реализация экстракции данных из произвольных JSON-структур.....	36
3.3 Реализация процесса извлечения и сохранения конфигураций рецептов	38
3.4 Реализация менеджера шаблонов.....	40

3.5 Реализация подсистемы извлечения игровых метаданных и графических ресурсов.....	43
4 Контроль стандартов качества программных решений	47
4.1 Выбор методов тестирования комплекса приложений	47
4.2 Тестирование верификации и генерации кода для стандартных игровых рецептов.....	48
4.3 Тестирование целостности данных при итеративном преобразовании JSON-структур.....	54
4.4 Метрики кода.....	58
4.4.1 Назначение и применение метрик.....	58
4.4.2 Цикломатическая сложность	58
4.4.3 Метрика SLOC.....	60
4.4.4 Метрика Холстеда.....	61
4.4.5 Комплексный показатель сопровождаемости.....	64
4.5 Приёмочное тестирование в натурной среде	64
4.5.1 Организация тестовой среды и подготовка испытаний	65
4.5.2 Методика проверки точности извлечения метаданных	65
4.5.3 Методика проверки работоспособности генерируемых скриптов	67
4.5.4 Результаты тестирования извлечения метаданных	71
4.5.5 Результаты тестирования созданного кода	72
Заключение	73
Список использованных источников	74
Приложение А (справочное) Глоссарий	77
Приложение Б (обязательное) Листинг приложения	79

Введение

Ещё лет десять назад игра была продуктом – её покупали, запускали, проходили. Сегодня всё иначе. «Minecraft», «Roblox» и десятки похожих проектов сделали из игрока не зрителя, а участника: он сам решает, по каким правилам существует его игровой мир. Раньше было чётко: вот разработчик, вот игрок. Сейчас эти роли перемешались – и обратно их уже не растащить. Но у этой свободы есть своя цена – чтобы что-то изменить под себя, нужно уметь программировать.

В центре этой работы – рецепты игры «Minecraft»: что из чего создаётся, что во что превращается. Именно с этим механизмом здесь предстоит работать – понять его структуру и научить программу писать код для его изменения вместо человека.

Проблема не надуманная. Игроки, которые хотят подправить рецепты под свои нужды, сталкиваются с одним и тем же: Forge API требует Java, ZenScript – своего синтаксиса, KubeJS – понимания JavaScript. Для большинства это глухая стена.

Цель работы – убрать эту стену. Игрок выбирает предметы, нажимает кнопку и получает готовый файл. Никакой документации KubeJS, никакого ручного кода.

Для этого нужно решить три задачи:

- а) разобраться в файловой структуре игры и найти способы получить из неё нужные данные;
- б) написать алгоритмы, которые сами собирают и сортируют информацию об игровых объектах;
- с) собрать программный интерфейс: загрузка изображений предметов с названиями плюс автоматическая генерация готового KubeJS-кода;
- д) провести всестороннее тестирование разработанного программного комплекса. Необходимо провести модульное, тестирование граничных условий, а также проверку при итеративных преобразованиях шаблонного рецепта.

1 Анализ предметной области и постановка задачи

1.1 Анализ предметной области

«Minecraft» давно перестал быть игрой с фиксированными правилами. Сообщество разрасталось годами, и вместе с ним запрос на инструменты, которые позволяют менять внутриигровые рецепты не вручную, а программно. KubeJS стал де-факто стандартом в этой нише: с его помощью авторы модификаций описывают, что из чего создаётся, задают условия и выходы. Проблема в другом – всё это требует знания JavaScript [1], и даже небольшая опечатка в синтаксисе ломает логику целого пакета модификаций. Чем крупнее проект, тем острее эта боль.

Отсюда и возникает нужда в инструменте другого уровня. Таком, который сам следит за правильностью структуры данных, генерирует код и проверяет параметры до того, как они попадут в игру. Важный нюанс: такое решение должно работать локально. Модифицированные клиенты не всегда имеют стабильный выход в сеть, а зависимость от облачного сервиса – это лишняя точка отказа.

Технически задача непростая. Рецепты в разных модификациях игры описываются по-разному: форматы вызовов отличаются, аргументы верифицируются по своим правилам, а код должен оставаться совместимым с игровой логикой конкретной версии. Сверху на это накладывается необходимость разбирать абстрактные синтаксические деревья (AST) – без этого нельзя надёжно анализировать и строить корректные вызовы [2].

Игровые рецепты – это не просто «положи три дерева, получи доску». В сложных модифицированных сборках встречаются компоненты с вероятностным выходом ресурсов, жидкостные ингредиенты, групповые теги предметов, многоступенчатые цепочки преобразований. Автору нужно всё это контролировать и желательно без того, чтобы каждый раз лезть в исходный код вручную.

Для решения этих задач выбран язык C# и архитектура на основе строго типизированных моделей конфигураций. Такой подход даёт несколько преимуществ сразу: генерация кода становится предсказуемой, систему легко расширять под новые рецепты, а подключить её к стороннему приложению можно без привязки к какому-либо интерфейсу отображения.

1.2 Архитектурно-техническая реализация системы автоматизации

1.2.1 Модульная организация и принципы проектирования

В основу проектирования API положены принципы модульной архитектуры и разделения ответственности [3]. Игровой рецепт представлен в виде объекта конфигурации – он хранит все данные, необходимые для последующей генерации кода. Такое разделение не случайно: вынося объекты конфигурации и модуль тестирования в отдельные слои, можно проверять каждый метод независимо, обращаясь только к публичному API. Верификация возвращает список ошибок. Пустой список означает успешное прохождение теста. Модуль тестирования при этом не имеет прямой связи с логикой генерации рецептов для игры «Minecraft».

1.2.2 Анализа кода на основе абстрактного синтаксического дерева

Для обработки файлов с рецептами KubeJS реализован механизм преобразования исходного кода в абстрактное синтаксическое дерево (Abstract Syntax Tree, AST). Это позволяет представить содержимое файла не как текст, а как список всех вызовов методов KubeJS, структуру, с которой уже можно работать программно.

Анализ на уровне AST даёт важное преимущество: форматирование, комментарии и порядок инструкций перестают иметь значение. Используемая библиотека синтаксического анализа поддерживает современный стандарт языка и корректно строит дерево даже для сложных вложенных конструкций и цепочек вызовов [4].

Первый шаг анализа – верификация структуры входного файла: система проверяет наличие ожидаемого вызова метода регистрации серверных событий. Если структура нарушена, генерируется диагностическое сообщение с указанием источника ошибки.

1.2.3 Система экстракторов и реестр обработчиков

Извлечение данных построено на наборе специализированных экстракторов – каждый из них отвечает за распознавание и преобразование одного конкретного типа рецептов. Все экстракторы реализуют единый интерфейс с двумя методами: проверка возможности обработать узел и непосредственное извлечение конфигурации.

Реестр обработчиков управляет очередностью их применения. При обходе AST система последовательно опрашивает зарегистрированные модули – пока один из них не подтвердит, что может обработать текущий элемент. Специфичные экстракторы для известных форматов проверяются первыми; универсальный обработчик для произвольного кода идёт последним и служит резервным механизмом сохранения.

Вспомогательный модуль собирает утилиты, общие для всех экстракторов: извлечение строковых значений из литералов и идентификаторов, рекурсивное преобразование объектных выражений в структурированные данные, обработка массивов и вложенных конструкций. Вынесение этой логики в отдельный компонент исключает её дублирование.

Для рецептов, основанных на механических процессах, реализован специализированный экстрактор с поддержкой цепочек вызовов и модификаторов. Экстрактор разбирает не один вызов, а всю цепочку. Базовый вызов регистрации рецепта идёт первым. Следующие методы меняют параметры выполнения: длительность обработки и условия сохранения инструментов.

Рецепт, который не вписался ни в один шаблон, уходит к универсальному экстрактору. Тот сохраняет узел как есть, не пытаясь разобрать его структуру. Синтаксически корректный код попадет в выходной файл в любом случае.

1.2.4 Стратегии генерации и сохранения кода

Исполняемый скрипт из внутренних представлений рецептов собирают генераторы кода. Каждый следит за форматированием: отступы, кавычки, разделители выставляются по правилам языка JavaScript.

Запись в файловую систему работает через стратегии. Базовый класс задает контракт. Дальше три варианта: записать новый файл полностью, дописать фрагмент в конец или атомарно заменить содержимое через временный файл.

Атомарная запись защищает данные от повреждения при сбоях. Содержимое сначала записывается во временный файл – и только после успешного завершения исходный документ заменяется переименованием. Если в процессе возникает ошибка, временный файл удаляется и никаких неполных записей не остаётся [5].

1.2.5 Интеграция с внешними источниками данных

Система поддерживает получение информации из внешних источников – в частности, из файлов журналов и экспортных данных игрового клиента. За это отвечает специализированный модуль, извлекающий перечни предметов и жидкостей вместе с сопутствующими метаданными: локализованными названиями, принадлежностью к конкретным игровым дополнениям, графическими представлениями.

Процесс извлечения включает чтение файлов метаданных в структурированном формате, сопоставление идентификаторов с файловыми ресурсами и загрузку бинарных данных изображений. Всё это агрегируется в единое хранилище, доступное для поиска и фильтрации по различным критериям.

1.3 Анализ программного обеспечения аналогичного назначения

В сфере модификации игры «Minecraft» существует ряд программных инструментов, предназначенных для автоматизации создания и управления игровыми рецептами. Для проведения сравнительного анализа выбраны два

наиболее релевантных решения: «CraftTweaker» и «KubeJS», которые обладают функциональностью, близкой к разрабатываемому программному обеспечению, но различаются архитектурными подходами и целевой аудиторией.

«CraftTweaker» (на рисунке 1) представляет собой фундаментальный модификационный фреймворк, позволяющий изменять рецепты, удаления предметов и другие аспекты игрового процесса посредством скриптового языка ZenScript [6].



Рисунок 1 – Графический интерфейс редактора скриптов CraftTweaker

Основные возможности:

- редактирование рецептов создания, плавки, обработки в машинах модификаций;
- удаление, замена или модификация существующих игровых рецептов;
- поддержка условий выполнения и приоритетов обработки;
- интеграция с более чем 200 популярными модификациями;
- система логгирования для отладки скриптов.

Преимущества:

- поддержка большого количества модификаций через официальные расширения;
- более 10 лет развития, стабильная работа на различных версиях игры «Minecraft»;

- возможность тонкой настройки поведения рецептов через условия и приоритеты;

- обширная база знаний, примеры скриптов и техническая поддержка.

Недостатки:

- сложный синтаксис ZenScript;
- отсутствие визуальных средств отладки;
- ручное управление зависимостями.

Модификация «KubeJS» – это современное решение для модификации игры «Minecraft», использующий JavaScript и предоставляющий унифицированный API для работы с рецептами, предметами, жидкостями и событиями игры.

Основные возможности:

- создание и модификация рецептов через декларативный JavaScript-синтаксис;
- регистрация новых предметов, блоков, жидкостей и сущностей;
- интеграция с системой тегов «Minecraft» для группировки предметов.

Преимущества:

- использование стандарта JavaScript;
- модульная архитектура;
- встроенная верификация;
- высокая скорость модифицирования;
- документация и примеры.

Недостатки:

- зависимость от версии игры «Minecraft»;
- отсутствие визуального конструктора;
- требование знания языка программирования JavaScript;
- ограниченные средства рефакторинга.

Сравнение аналогов приведено в таблице 1.

Таблица 1 – Сравнительный анализ аналогов с разрабатываемым программным обеспечением

Критерий	CraftTweaker	KubeJS	Разрабатываемое ПО
Целевая аудитория	Опытные разработчики, технические специалисты	Опытные разработчики	Начинающие разработчики
Язык	ZenScript	JavaScript	Нет
Порог входа	Высокий	Средний	Низкий
Визуальные средства	Присутствуют	Отсутствуют	Планируется базовый интерфейс
Автоматизация генерации	Отсутствует	Полуавтоматическая	Полностью автоматическая
Поддержка модификаций	Через расширения	Нативная интеграция	Универсальный адаптер
Отладка	Текстовые логи	Текстовые логи и верификация	Текстовые логи и верификация
Документация	Сообщество, форумы	Официальная структурированная	Встроенные подсказки и примеры
Масштабируемость	Средняя	Высокая	Ориентирована на типичные сценарии

Разрабатываемое программное обеспечение должно объединить сильные стороны обоих типов решений:

- простоту использования и стандартный синтаксис JavaScript как в «KubeJS»;

- автоматизацию рутинных операций через генерацию кода на основе конфигурации;

- минимальные требования к техническим знаниям пользователя;
- встроенные средства верификации и отладки;
- ориентацию на типовые задачи с технической направленностью.

При этом необходимо учесть недостатки аналогов и избежать:

- чрезмерного усложнения интерфейса и функциональности;
- отсутствия визуальной обратной связи при ошибках;
- необходимости ручного управления зависимостями и порядком

выполнения;

- избыточной гибкости, не востребованной в большинстве сценариев;
- высоких требований к квалификации пользователя.

Разрабатываемое программное обеспечение будет отличаться от существующих решений акцентом на автоматизацию типовых операций, интуитивной конфигурацией через структурированные данные и встроенными средствами контроля качества генерируемого кода, что позволит сократить время разработки и снизить вероятность ошибок при создании игровых рецептов.

1.4 Постановка задачи на разработку программных средств

Ручное написание скриптов для KubeJS требует знания JavaScript и отнимает значительное время. Одна опечатка – и конфигурация ломается. С простыми рецептами это поправимо, но в дополнениях наподобие «Create» рецепты исчисляются сотнями, и отслеживать их взаимосвязи вручную становится попросту нереально. Стандартизировать этот процесс без специального инструмента практически невозможно.

Разрабатываемое программное обеспечение решает эту задачу напрямую. Оно предназначено для авторов игровых модифицированных сборок на базе «Minecraft». Три основных сценария работы: настройка параметров рецептов,

импорт готовых конфигураций из игровых файлов и получение валидного кода на выходе.

Система должна справляться не только с простыми случаями. Обязательна поддержка многоуровневых структур – модификаторов цепочек вызовов, вероятностных выходов, жидкостных компонентов. Входные данные проходят верификацию на каждом этапе. Предметы автоматически сопоставляются с локализованными названиями, тегами и графическими ресурсами.

С технической стороны требуется несколько ключевых решений. Первое – модульная архитектура на основе анализа абстрактных синтаксических деревьев (Abstract Syntax Tree, AST): она обеспечивает преобразование исходного кода в типизированные конфигурации. Второе – полиморфная система генерации скриптов с гибким управлением отступами и форматированием. Третье – полностью локальная работа без зависимости от внешних сервисов.

Отдельного внимания требуют два аспекта: корректная обработка экспортированных метаданных и надёжные стратегии сохранения файлов. Результат – скрипты, полностью совместимые со стандартами KubeJS. Пользователю ничего не придётся исправлять или дописывать после генерации.

1.5 Анализ функциональных требований

Диаграмма вариантов использования нужна для визуализации функциональных требований к системе или программному продукту. Она позволяет понять, как пользователи будут взаимодействовать с системой, какие действия они будут выполнять, какие варианты использования системы существуют, а также кто участвует в этих вариантах использования [7]. Разработанная диаграмма вариантов использования для программного обеспечения автоматизации игровых скриптов приведена на рисунке 2. Основные функциональные возможности системы: получение внутриигровых данных, работа с шаблонами JSON (удаление, редактирование и создание), а также основная работа с конфигурациями игровых рецептов.

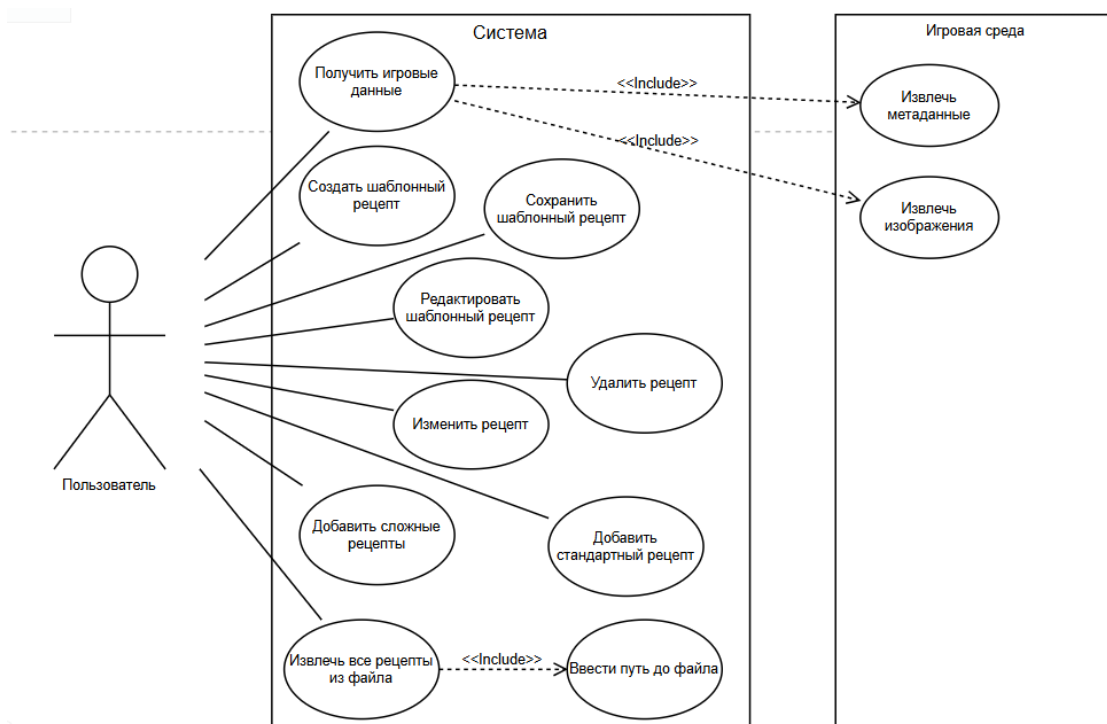


Рисунок 2 – Диаграмма вариантов использования разрабатываемого приложения

На диаграмме заметны две системы. Особое внимание стоит уделить системе игровой среды – варианты для извлечения изображений и метаданных. Они выполняются пользователем в специально настроенной среде игры «Minecraft» с модификацией «Icon Exporter». Спецификация варианта использования «Добавить стандартный рецепт» приведена на рисунке 3.

Прецедент: Добавить стандартный рецепт
Краткое описание: Создаёт рецепт на основе введенных данных пользователя
Актеры: Пользователь
Предусловия: -
Основной поток: <ol style="list-style-type: none"> 1. Пользователь выбирает тип создаваемого рецепта. 2. Вводит входные данные. 3. Вводит выходные данные. 4. Система создаёт рецепт.
Постусловия: Создан стандартный рецепт.

Рисунок 3 – Спецификация варианта использования «Добавить стандартный рецепт»

Спецификация варианта использования «Получить игровые данные» приведена на рисунке 4.

Прецедент: Получить игровые данные
Краткое описание: Импортирует в приложение игровые данные для удобной разработки и навигации
Актеры: Пользователь
Предусловия: -
Основной поток: <ol style="list-style-type: none"> 1. Пользователь выбирает получение данных. 2. Система проходит по содержимым папок. 3. Система сохраняет в приложение игровые данные в единый словарь. 4. Система создаёт рецепт.
Постусловия: Игровые данные импортированы.

Рисунок 4 – Спецификация варианта использования «Получить игровые данные»

Получение игровых данных требует сгенерированных игрой папок с ресурсами – без этого получение и категоризация работать не будут. Спецификация варианта использования «Изменить рецепт» приведена на рисунке 5.

Прецедент: Изменить рецепт
Краткое описание: Изменяет данные рецепта на основе фильтра поиска.
Актеры: Пользователь
Предусловия: -
Основной поток: <ol style="list-style-type: none"> 1. Пользователь выбирает тип изменение (по входным данным или выходным). 2. Пользователь вводит фильтр поиска рецептов (например по ID). 3. Пользователь вводит входные и выходные данные. 4. Система создает функцию изменения рецепта.
Постусловия: Функция изменения рецепта создана.

Рисунок 5 – Спецификация варианта использования «Изменить рецепт»

Проанализированные функциональные возможности приложения определяют главные основные способы взаимодействия пользователя с разрабатываемым программным комплексом: извлечение ресурсов, категоризация,.

1.6 Анализ нефункциональных требований

Приложение разрабатывается в среде Microsoft Visual Studio 2026 на языке C#. Целевая платформа: Windows 10 и выше; поддержка других операционных систем не предусматривается.

По своей роли система является внешним инструментом настройки: она работает напрямую с файловой структурой и экспортными метаданными игрового клиента «Minecraft», не встраиваясь в него. Генерируемые скрипты написаны под синтаксис «KubeJS» и исполняются внутри игровой среды, при условии, что на стороне сервера установлено соответствующее серверное дополнение.

Для хранения конфигураций, шаблонов и промежуточных данных используется формат JSON. Всё хранится локально. Никаких внешних серверов и баз данных приложение не требует.

1.7 Планирование разработки и оценка бюджета

Для оценки временных затрат, а также наглядного планирования процесса разработки программного обеспечения для автоматизации генерации скриптов была построена диаграмма Ганта с использованием инструментов управления проектами [8]. Общий процесс разработки приложения содержит пять ключевых вех. Каждый этап включает в себя определённый набор подзадач, выполнение которых необходимо для перехода на следующий уровень готовности продукта. Для реализации данного проекта целесообразно выделить следующие этапы:

- анализ предметной области и постановка задачи на разработку программного обеспечения для автоматизации написания скриптов;

- проектирование программного обеспечения для автоматизации генерации скриптов;
- реализация программного обеспечения;
- контроль качества и тестирование;
- оформление отчётной документации.

Краткое описание задач каждого этапа приведено на рисунке 6.

Название задачи	Длительность	Начало	Окончание	Предшественники
Начало проекта	0 дней	Пт 08.05.26	Пт 08.05.26	
▣ Анализ предметной области и постановка задачи	4,1 дней	Пт 08.05.26	Чт 14.05.26	1
Анализ предметной области	0,5 дней	Пт 08.05.26	Пт 08.05.26	1
Анализ аналогичных продуктов	0,5 дней	Пт 08.05.26	Пт 08.05.26	3
Анализ стратегий генерации кода	0,5 дней	Пн 11.05.26	Пн 11.05.26	4
Анализ функциональных требований	0,5 дней	Пн 11.05.26	Пн 11.05.26	5
Анализ нефункциональных требований	0,1 дней	Вт 12.05.26	Вт 12.05.26	6
Планирование разработки и бюджета	1 день	Вт 12.05.26	Ср 13.05.26	7
Оценка правового аспекта разработки ПО	1 день	Ср 13.05.26	Чт 14.05.26	8
Завершение анализа предметной области	0 дней	Чт 14.05.26	Чт 14.05.26	9
▣ Проектирование ПО для автоматизации написания скриптов	26 дней	Чт 14.05.26	Пт 19.06.26	9
Проектирование архитектуры приложения	2 дней	Чт 14.05.26	Пн 18.05.26	9
Проектирование моделей конфигураций	5 дней	Чт 14.05.26	Чт 21.05.26	9
Проектирование концептуальной модели данных	4 дней	Чт 21.05.26	Ср 27.05.26	13
Проектирование логических моделей данных	7 дней	Ср 27.05.26	Пт 05.06.26	14
Проектирование программных средств	10 дней	Пт 05.06.26	Пт 19.06.26	15;12
▣ Реализация ПО	22 дней	Пт 19.06.26	Вт 21.07.26	16
Реализация стратегий генерации кода	20 дней	Пт 19.06.26	Пт 17.07.26	16
Реализация CLI	1 день	Пт 19.06.26	Пн 22.06.26	16
Реализация шаблонных рецептов	10 дней	Пт 19.06.26	Пт 03.07.26	16
Реализация фильтров-обработчиков	2 дней	Пт 17.07.26	Вт 21.07.26	18
Завершение реализации ПО	0 дней	Вт 21.07.26	Вт 21.07.26	21;20;19
▣ Контроль качества и тестирование	8 дней	Вт 21.07.26	Пт 31.07.26	22
Roundtrip-тестирование	8 дней	Вт 21.07.26	Пт 31.07.26	22
Натурное тестирование стратегий	1 день	Вт 21.07.26	Ср 22.07.26	22
Автоматизированное тестирование граничных	6 дней	Вт 21.07.26	Ср 29.07.26	22
Завершение тестирования	0 дней	Пт 31.07.26	Пт 31.07.26	24;25;26
▣ Завершение работы	9 дней	Пт 31.07.26	Чт 13.08.26	27
Оформление отчёта	9 дней	Пт 31.07.26	Чт 13.08.26	27
Конец проекта	0 дней	Чт 13.08.26	Чт 13.08.26	29

Рисунок 6 – Сведения об этапах разработки приложения

На первом этапе содержатся задачи, направленные на проведение анализа предметной области модификаций, изучение синтаксиса JavaScript и структуры абстрактных синтаксических деревьев, изучение существующих решений для синтаксического анализа и генерации кода, а также формирование функциональных и нефункциональных требований к разрабатываемому приложению. Особое внимание уделяется оценке правовых аспектов разработки ПО в экосистеме Minecraft (EULA Mojang, ГК РФ).

Второй этап предполагает проектирование архитектуры приложения, разработку объектной модели конфигураций рецептов, проектирование моделей

данных. На данном этапе определяются паттерны проектирования, стратегии верификации и сериализации, а также механизмы обработки ошибок.

Третий этап предполагает непосредственную реализацию разрабатываемого программного обеспечения: создание синтаксического анализатора на основе библиотеки Esprima, реализацию реестра экстракторов, разработку иерархии конфигурационных классов, реализацию стратегий генерации кода и сохранение результатов в файловую систему. Также на данном этапе реализуется консольный интерфейс взаимодействия с пользователем и модуль извлечения метаданных из игровой среды.

Четвёртый этап предполагает комплексное тестирование разработанного ПО: roundtrip-тестирование (входные данные, затем генерация, затем синтаксический анализ сгенерированного кода, и получение данных, затем сравнение входных и выходных данных на равенство), натурное тестирование в игровой среде, автоматизированную проверку граничных условий и верификацию выходных файлов на синтаксическую корректность. Дополнительно проводится анализ устойчивости системы к некорректным входным данным и оценка производительности при обработке больших объёмов кода.

Пятый этап является заключительным и предполагает оформление отчёта по выполненной работе, подготовку демонстрационных материалов, документирование API и инструкций по использованию утилиты, а также финальную верификацию соответствия результатов исходным требованиям.

Для наглядного отражения временных характеристик разработки программного обеспечения была построена диаграмма Ганта. Она не выглядит достаточно разветвленной из-за общей линейности задач. В ней нет сложных структурных особенностей, поскольку работа представляет собой чёткую последовательность задач. Также объем работ не велик, это также делает диаграмму менее структурно сложной. Данная диаграмма приведена на рисунке 7.

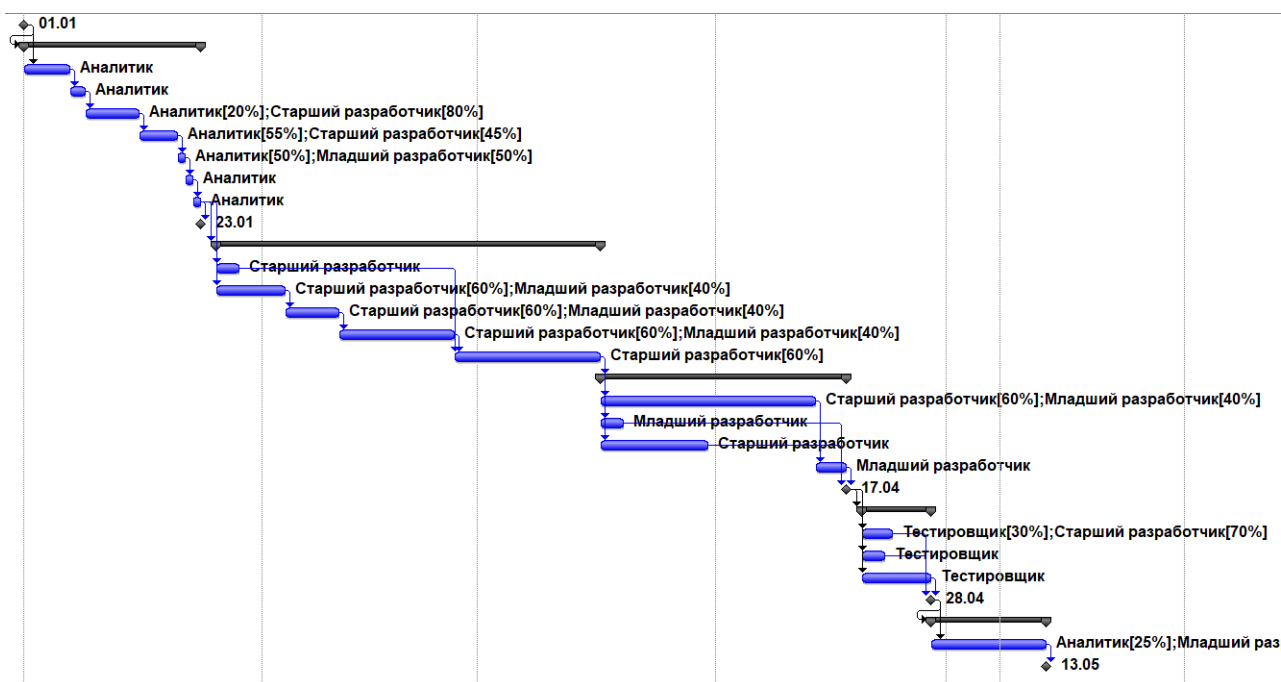


Рисунок 7 – Этапы разработки приложения на диаграмме Ганта

Согласно диаграмме, общая продолжительность проекта составляет 133 календарных дней, начиная с 01.01.2026, и заканчивая в 13.05.2026.

Критический путь проекта проходит через этапы проектирования моделей данных и реализации стратегий генерации кода, что определяет их приоритет при распределении ресурсов.

Анализ профилей загруженности ключевых исполнителей позволил построить графики загрузки с плановой потребностью для каждого сотрудника. На рисунке 8 представлен график загрузки старшего разработчика.

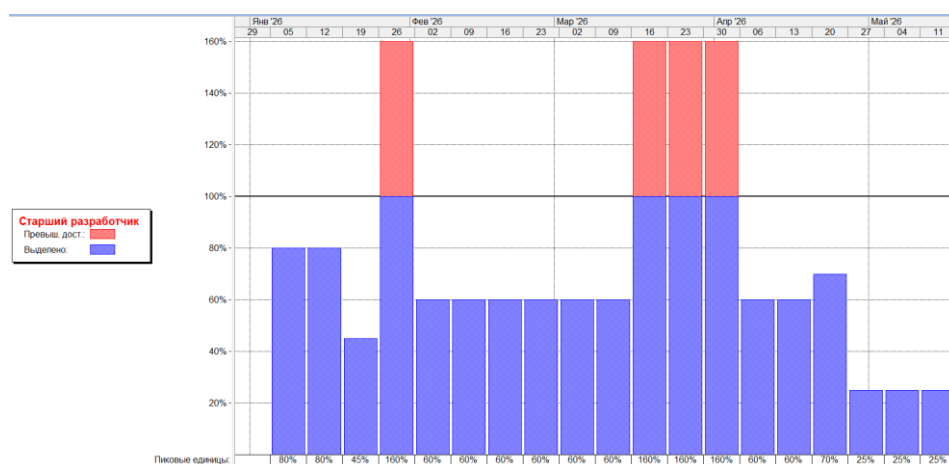


Рисунок 8 – График загрузки старшего разработчика

Общий период его участия в проекте составляет 133 дня. Перераспределение соответствующих задач затруднено, поскольку они опираются на уникальные экспертные компетенции, недоступные другим членам команды. Привлечение дополнительного специалиста на столь короткий период экономически нецелесообразно. Превышение допустимой загрузки младшего сотрудника на рисунке 9 характеризуется единичным коротким периодом.

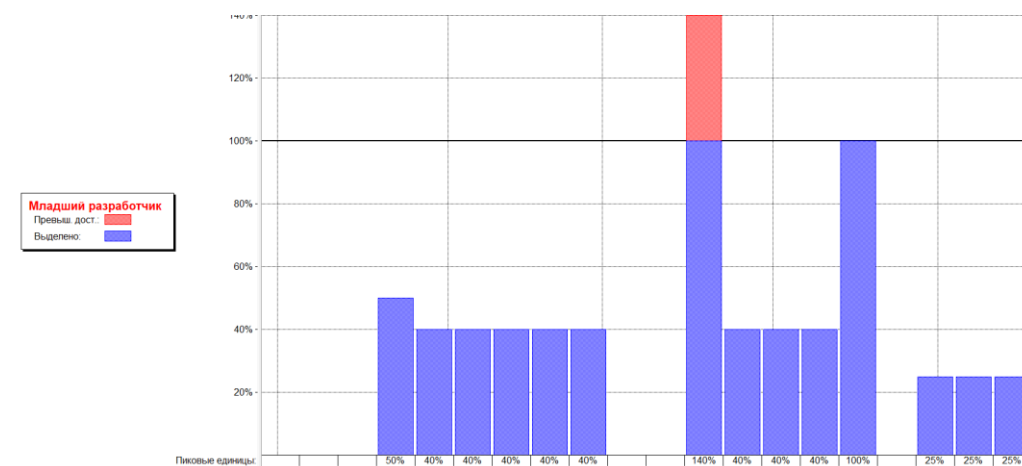


Рисунок 9 – График загрузки младшего разработчика

Такая перегрузка не создаёт системного риска для графика работ. Устранение возможно путём перераспределения части некритических задач (перенос которых не удлиняет общую продолжительность проекта) внутри текущего объема работ, либо смещением части неприоритетных работ на другие, менее загруженные дни, с неполной занятостью без нарушения зависимостей. Как демонстрирует рисунок 10, аналитик не испытывает перегрузки.

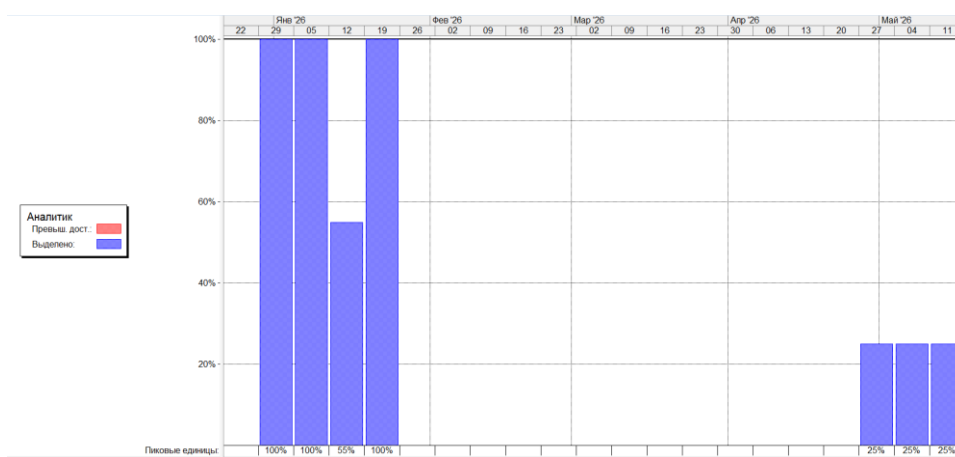


Рисунок 10 – График загрузки аналитика

Периоды загрузки только в начале и в конце работы над проектом свидетельствуют о возможностях для экономии, аналитик может быть нанят только в необходимые периоды для единичных работ на почасовую ставку. График загруженности тестировщика на рисунке 11 выявляет перегрузку

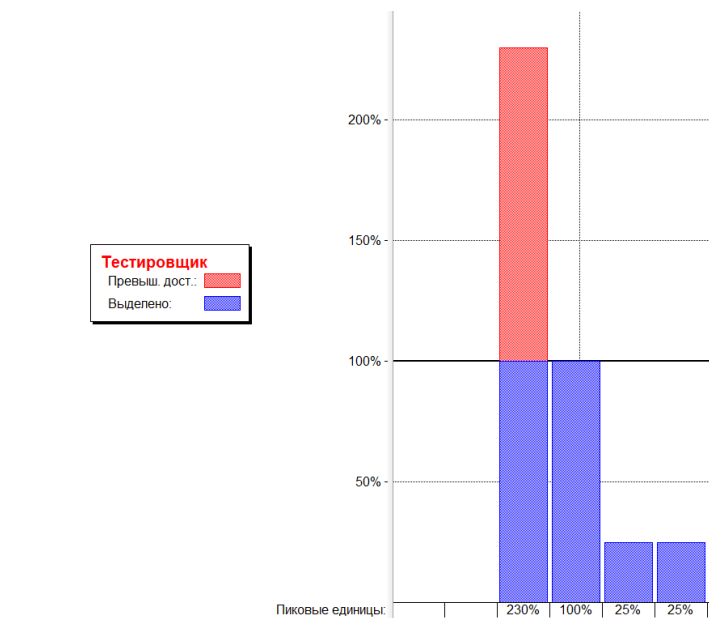


Рисунок 11 – График загруженности тестировщика

Перегрузка есть лишь на 1 периоде разработки в несколько дней, аналогичную с младшим разработчиком. Природа перегрузки, скорее всего, связана с пиковой концентрацией задач на завершающем этапе разработки. Она легко поддаётся выравниванию: достаточно перенести часть задач на соседние дни либо перераспределить нагрузку между младшим разработчиком и тестировщиком (при наличии соответствующих допусков или сертификации).

Таким образом суммарный объём перегрузок является незначительным в масштабе всего периода разработки проекта. С учётом изложенного, ресурсное планирование признаётся допустимым, а выявленные отклонения признаются некритичными для успешной реализации проекта.

Для оценки финансовых затрат разрабатываемого программного комплекса необходимо назначить цены используемых ресурсов. Итоговая стоимость затрат представлена на рисунке 12. Стоимость работ измеряется пропорционально объёмам проделанной сотрудниками работы. Нулевые значения стоимости указывают на задачи вехи с нулевой длительностью.

Название задачи	Фиксированные затраты	Начисление фикс. затрат	Общие затраты
Реализация стратегий генерации кода	0,00 Р	Пропорциональное	47 542,86 Р
Реализация шаблонных рецептов	0,00 Р	Пропорциональное	32 000,00 Р
Проектирование логических моделей	0,00 Р	Пропорциональное	28 160,00 Р
Оформление отчёта	0,00 Р	Пропорциональное	25 300,00 Р
Проектирование программных средств	0,00 Р	Пропорциональное	24 960,00 Р
Проектирование моделей конфигурации	0,00 Р	Пропорциональное	17 920,00 Р
Анализ стратегий генерации кода	0,00 Р	Пропорциональное	15 200,00 Р
Автоматизированное тестирование	0,00 Р	Пропорциональное	14 000,00 Р
Проектирование концептуальной модели	0,00 Р	Пропорциональное	12 800,00 Р
Roundtrip-тестирование	0,00 Р	Пропорциональное	11 360,00 Р
Анализ предметной области	0,00 Р	Пропорциональное	9 600,00 Р
Проектирование архитектуры приложения	0,00 Р	Пропорциональное	9 600,00 Р
Анализ функциональных требований	0,00 Р	Пропорциональное	8 280,00 Р
Реализация фильтров-обработчиков	0,00 Р	Пропорциональное	6 400,00 Р
Натурное тестирование стратегий	0,00 Р	Пропорциональное	6 000,00 Р
Анализ аналогичных продуктов	0,00 Р	Пропорциональное	4 800,00 Р
Реализация CLI	0,00 Р	Пропорциональное	4 800,00 Р
Планирование разработки и бюджет	0,00 Р	Пропорциональное	2 400,00 Р
Оценка правового аспекта разработки	0,00 Р	Пропорциональное	2 400,00 Р
Анализ нефункциональных требований	0,00 Р	Пропорциональное	2 000,00 Р
Начало проекта	0,00 Р	Пропорциональное	0,00 Р
Завершение анализа предметной области	0,00 Р	Пропорциональное	0,00 Р
Завершение реализации ПО	0,00 Р	Пропорциональное	0,00 Р
Завершение тестирования	0,00 Р	Пропорциональное	0,00 Р
Конец проекта	0,00 Р	Пропорциональное	0,00 Р
	0,00 Р		285 522,86 Р

Рисунок 12 – Стоимость затрат

Таким образом, исходя из оценки финансовых и временных затрат, предполагаемые расходы на разработку составляют 285522,86 рублей, а общая продолжительность – 133 дней. Такой срок хорошо укладывается в рамки одного академического семестра, то есть проект вполне возможно выполнить в учебных условиях. При этом финансовые затраты остаются весьма скромными для программного продукта сопоставимой сложности.

Сдержанная стоимость во многом объясняется выбранными техническими средствами: вся работа должна вестись с применением открытых и бесплатных инструментов. Начиная от среды разработки и заканчивая библиотеками для контроля системы версий. Поэтому потребности в закупке коммерческих лицензий не возникло.

Дополнительным плюсом оказалась возможность реализовать проект силами малой команды, одним-двумя исполнителями без узких специалистов со стороны. Наличие такой малой командой также значительно уменьшает финансовую нагрузку. Всё это в совокупности делает разработку оправданной и усиливает финансовую сторону.

2 Проектирование программного обеспечения для автоматизации генерации скриптов

2.1 Архитектура приложения

Архитектура разрабатываемого программного обеспечения построена на основе многоуровневой модульной структуры, обеспечивающей строгое разделение ответственности между компонентами системы [9]. Приложение функционирует исключительно в серверном режиме без реализации графического интерфейса, взаимодействуя с пользователем через консольный интерфейс командной строки и предоставляя функциональность в виде подключаемой библиотеки. Основа архитектуры состоит из пяти логических уровней.

Уровень интеграции с игровой средой представлен подключаемой библиотекой, ответственной за сбор внешних ресурсов и метаданных из файловой структуры клиента игры «Minecraft». Не требует взаимодействия по именованным каналам или запуска сторонних процессов.

GamePathProvider – централизованный менеджер путей, обеспечивающий конфигурацию корневого каталога игры и определение путей к метаданным и иконкам.

ItemFluidWithIconExtractor – модуль извлечения файлов и бинарных ресурсов-изображений. Формирует словарь, связывающий идентификаторы предметов с локализованными названиями, тегами и графическими данными.

Уровень конфигураций и бизнес-логики содержит строго типизированные модели данных, реализующие верификации и правила предметной области. Построен на принципах объектно-ориентированного наследования и полиморфизма.

BaseRecipeConfiguration – абстрактный базовый класс, определяющий единый контракт для всех типов рецептов. Включает свойства, методы верификации и генерации.

CreateCompactingConfig, CreateMixingConfig, и др. – конкретные реализации для рецептов модификации «Create». Содержат ограничения, основанные на бизнес-логике, проверяют допустимое количество входов и выходов, поддерживаемые модификаторы и типы ингредиентов.

MinecraftStandardRecipeConfig – конфигурация для базовых рецептов плавки, резки и обработки. Реализует проверки диапазонов, запрет на дубликаты и условную поддержку модификаторов.

RawCodeBlock – контейнер для сохранения исходного AST-узла (RawNode) без семантического разбора. Обеспечивает полное восстановление оригинального кода при экспорте.

Уровень синтаксического анализа и экстракции обеспечивает преобразование исходного JavaScript-кода в типизированные внутренние представления. Реализует паттерн «Цепочка ответственности» для обработки абстрактного синтаксического дерева (AST) [10, 11].

JsRecipeParser – главный синтаксический анализатор, использующий библиотеку «Esprima» для анализа. Выполняет верификацию структуры файла и обход узлов лямбда-выражений.

AstHelper – набор статических утилит для безопасного извлечения строковых, числовых и объектных значений из узлов AST.

ExtractorRegistry – реестр экстракторов, реализующий приоритетную проверку узлов. Порядок регистрации определяет порядок обработки: специфичные экстракторы проверяются первыми, общий (RawCodeExtractor) – последним [12].

CreateExtractor – модуль анализа вызовов вида «event.recipes.create.*». Реализует рекурсивный обход цепочек модификаторов, нормализацию имён методов на соответствующие конфигурации.

CustomExtractor, RemovalExtractor – модули анализа произвольных рецептов (event.custom) и правил удаления (event.remove), поддерживающих вложенную логику фильтрации.

`RawCodeExtractor` – финальный обработчик, сохраняющий неинтерпретируемые или сложные выражения в виде исходного AST-узла для последующего точного восстановления кода.

Уровень генерации и сохранения отвечает за преобразование внутренних конфигураций в валидный JavaScript-код и его запись в файловую систему.

Метод `GenerateJsCode()` – реализован в каждом наследнике `BaseRecipeConfiguration`. Выполняет сериализацию параметров, подстановку отступов, форматирование массивов и добавление модификаторов.

Класс `EmptyFunctionGenerator` отвечает за формирование базовой обёртки скрипта. Его вызов гарантирует, что выходной файл будет иметь корректную структуру.

Для записи данных на диск предусмотрены стратегии сохранения. Абстрактный `FileSaverBase` задаёт общий интерфейс, а его наследник `FileSaverToEnd` реализует механизм записи в конец файла. В начале вычисляется точка вставки перед закрывающей скобкой затем данные атомарно записываются через временный файл [13].

В основу работы заложен строго однонаправленный конвейер. Точкой входа служит начало анализа входного файла. Тот поступает в метод класса `JsRecipeParser`, где библиотека «Esprima» строит по его содержимому абстрактное синтаксическое дерево. Дальнейшим обходом узлов и их передачей нужным экстракторам кода управляет `ExtractorRegistry`. Каждый успешно извлеченный узел преобразуется в экземпляр одного из наследников `BaseRecipeConfiguration`, после чего запускается проверка целостности. Если верификация пройдена, конфигурация попадает в `GenerateJsCode()`. Этот метод создает отформатированный фрагмент на JavaScript, который записывается на диск текущей стратегией сохранения.

Подобное устройство даёт архитектуре два ключевых преимущества. Чтобы добавить поддержку нового типа рецепта, достаточно унаследовать класс `BaseRecipeConfiguration` и зарегистрировать соответствующий экстрактор. А также даёт снижение числа синтаксических ошибок в скриптах. Оно достигается

за счёт того, что типовые проверки и полиморфная сериализация берут на себя заботу о корректности выходного JavaScript-кода ещё на этапе компоновки внутреннего представления, имитируя эффект типобезопасности.

2.2 Проектирование моделей конфигураций и данных

Проектирование программных средств автоматизации требует строгой формализации входных и выходных данных, а также внутренних структур, обеспечивающих корректную обработку игровых рецептов. Поскольку разрабатываемая система функционирует без использования внешних хранилищ и опирается на локальную файловую систему, архитектура данных базируется на объектно-ориентированных моделях конфигураций, словарях метаданных и правилах семантической верификации. Данная структура обеспечивает независимость от конкретной среды выполнения, высокую расширяемость и точность генерации скриптов. На рисунке 13 показана диаграмма классов разрабатываемых конфигураций.

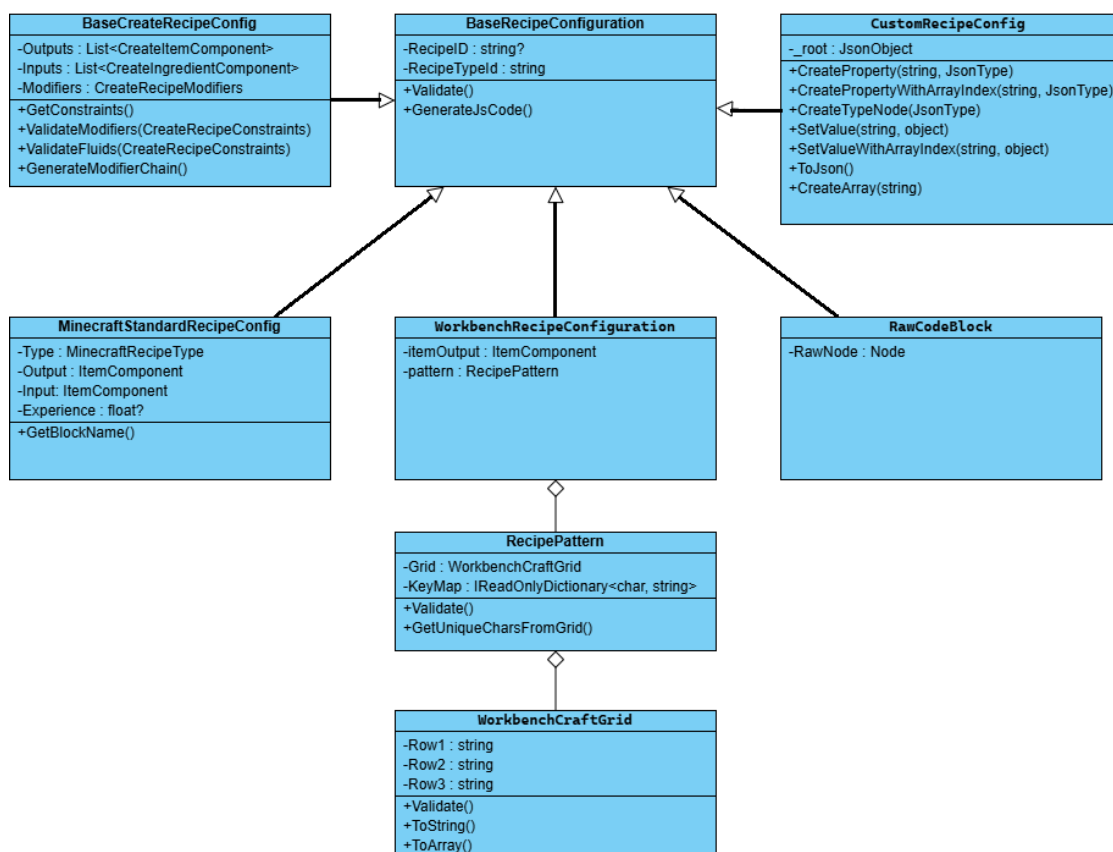


Рисунок 13 – Диаграмма классов конфигураций рецептов

Основными сущностями выступают: конфигурационные модели рецептов, компоненты предметов и жидкостей, модификаторы цепочек вызовов, а также метаданные игровых объектов. Взаимодействие между сущностями строится на принципах композиции и агрегации: каждый рецепт включает списки входных и выходных компонентов, которые могут содержать дополнительные параметры, такие как вероятности выпадения, объёмы или системные теги.

Система также оперирует реестром метаданных, сопоставляющим уникальные идентификаторы предметов с локализованными названиями, принадлежностью к программным модификациям, классификационными тегами и графическими ресурсами, полученными из внешних экспортов игрового клиента. На данном этапе определяются только связи между компонентами и правила их совместного использования. В таблице 2 описаны концептуальные сущности, которые являются основой для генерации рабочего и валидного JavaScript-кода.

Таблица 2 – Описание концептуальных сущностей

Название	Идентификатор	Назначение в системе
Конфигурация рецепта	BaseRecipeConfiguration	Абстрактный базовый тип, определяющий общий контракт для всех видов модификации игры, включая методы верификации и сериализации
Специализированный рецепт из модификации «Create»	BaseCreateRecipeConfig	Абстрактный базовый тип, определяющий общий контракт для большинства видов рецептов из модификации «Create».

Продолжение таблицы 2

Наименование сущности	Идентификатор	Назначение в системе
Конкретный специализированный рецепт из модификации «Create»	CreateCompactingConfig, CreateMillingConfig...	Конкретные реализации, хранящие параметры входов/выходов, модификаторы и отраслевые ограничения.
JSON-конфигурация	CustomRecipeConfig	Единый стандарт для большинства игровых модификаций. Преобразует свои данные в JSON-объект.

Для преобразования исходного кода во внутренние типизированные конфигурации рецептов в системе предусмотрен специализированный слой экстракторов, реализующий паттерн «Цепочка ответственности». Под экстрактором понимается программный компонент, способный по заданному узлу абстрактного синтаксического дерева восстановить семантику конкретной операции предметной области и построить соответствующий экземпляр класса-наследника `BaseRecipeConfiguration`. Входными данными для экстракторов служат AST-структуры, полученные синтаксическим анализатором на основе библиотеки «Esprima», а выходными строго типизированные объекты конфигураций, прошедшие первичную верификацию.

Все экстракторы регистрируются в централизованном реестре `ExtractorRegistry`, порядок добавления в который определяет приоритет обработки: от наиболее специфичных (например, экстрактор рецептов модификации «Create») до универсального `RawCodeExtractor`, размещаемого последним. Во время обхода синтаксического дерева реестр по очереди проверяет каждый узел всеми зарегистрированными экстракторами. Тот из них,

кто первым распознает знакомую семантическую конструкцию, сразу же разбирает узел и возвращает готовую конфигурацию. Дальнейшие проверки этого узла прекращаются. Если же ни один специализированный экстрактор не сработал, финальный RawCodeExtractor сохраняет узел как есть, чтобы позже точно воспроизвести его в выходном коде. Это позволяет системе не ломаться на незнакомых и экзотических выражениях, полностью сохраняя структуру скриптов. Весь набор экстракторов работает как связка между синтаксическим анализом и бизнес-логикой: он наполняет объектные данные реальными данными, после чего те проходят унифицированную обработку. На рисунке 14 представлена диаграмма классов экстракторов рецептов.

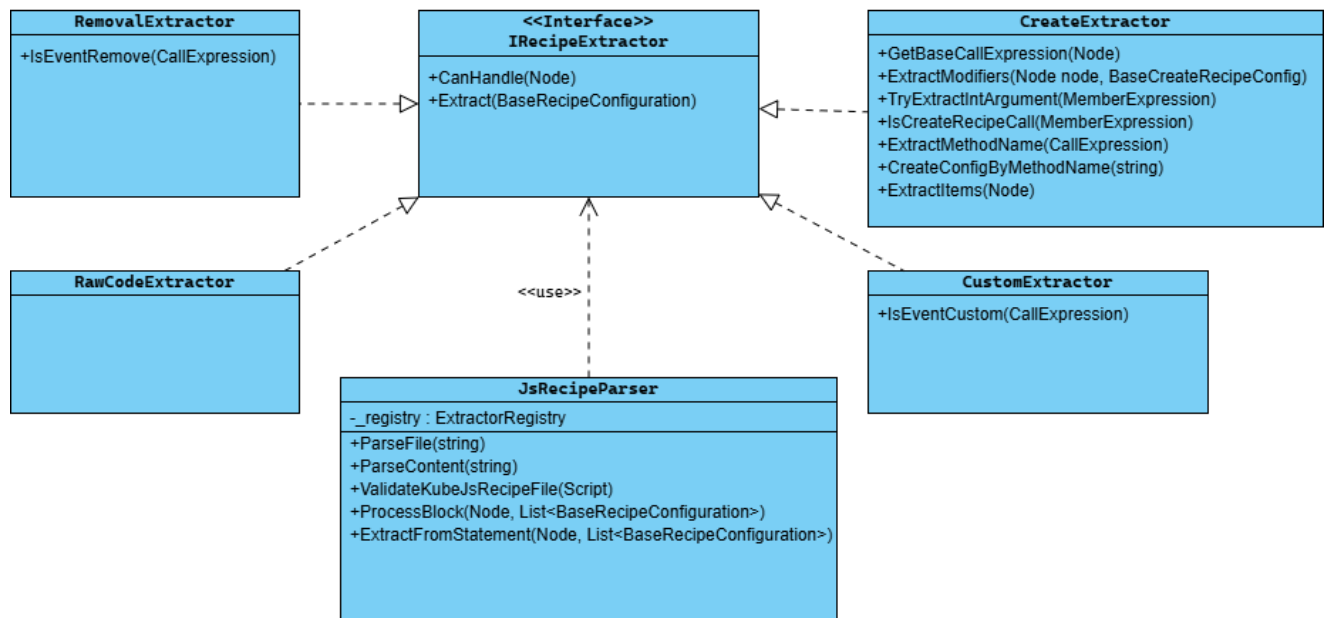


Рисунок 14 – Диаграмма классов экстракторов рецептов

В таблице 3 описаны сущности из диаграммы классов на рисунке 14. Основной упор делается на разборе классов для анализа AST.

Таблица 3 – Уровень семантического и синтаксического анализа

Наименование	Идентификатор	Назначение в системе
Базовый интерфейс для экстракторов	IRecipeExtractor	Определяет единый контракт для всех наследников.

Продолжение таблицы 3

Наименование	Идентификатор	Назначение в системе
Синтаксический анализатор	JsRecipeParser	Определяет массив из экстракторов, а затем использует их для извлечения конфигураций из исходного кода
Экстрактор рецептов модификации «Create»	CreateExtractor	Извлекает конфигурации из вызовов методов модификации «Create». Учитывает цепочки модификаторов вызовов
Экстрактор JSON объектов	CustomExtractor	Извлекает JSON объект из исходного кода

Представленная в таблице иерархия экстракторов, объединённая общим интерфейсом, конкретизирует описанный ранее принцип цепочки ответственности применительно к задаче семантического анализа игровых скриптов. Синтаксический анализатор JsRecipeParser выступает в роли координатора, последовательно передавая узлы абстрактного синтаксического дерева зарегистрированным реализациям интерфейса и получая на выходе типизированные конфигурационные объекты. Специализированный экстрактор CreateExtractor отвечает за разбор вызовов, характерных для модификации «Create», с учётом цепочек модификаторов, тогда как CustomExtractor обеспечивает извлечение JSON-структур произвольных рецептов, не привязанных к конкретному программному расширению. Совместное применение этих компонентов гарантирует полноту покрытия синтаксических конструкций и одновременно сохраняет возможность расширения системы

новыми экстракторами без изменения существующего кода синтаксического анализатора.

Таким образом, в настоящей главе была спроектирована статическая основа программной системы, охватывающая архитектурные уровни, модели конфигураций и механизмы семантического экстракции. Многоуровневая архитектура, разделенная на слои интеграции, логики, семантического анализа и генерации кода, задала чёткие границы ответственности и обеспечила независимую эволюцию каждого компонента. Ключевые абстракции сформировали иерархию типизированных моделей, способных единообразно описывать стандартные игровые механизмы и специализированные процессы из модификации «Create».

Детально проработанный слой экстракторов, организованный в виде цепочки ответственности с централизованным реестром `ExtractorRegistry`, решил задачу надёжно перехода от исходного JavaScript-кода к внутренним объектным представлениям. Благодаря приоритетной обработке специфичных конструкций и наличию универсального `RawCodeExtractor`, система оказалась устойчивой к неизвестным и редко встречающимся синтаксическим формам, полностью сохраняя исходный код для последующего восстановления. Взаимодействие `JsRecipeParser`, `AstHelper` и набора специализированных экстракторов (`CreateExtractor`, `CustomExtractor`, `RemovalExtractor`) позволило охватить широкий спектр рецептов – от простой плавки до комплексных JSON-конфигураций с фильтрацией.

Динамические аспекты функционирования разрабатываемого программного обеспечения – процедуры синтаксического анализа, разрешения неоднозначностей, проверки ограничений и генерации результирующего кода – опираются на созданный в данной главе статический каркас. Именно эти алгоритмические решения, их детальная проработка и тестирование станут предметом рассмотрения следующего раздела, посвящённого проектированию алгоритмического обеспечения системы.

3 Реализация программных средств

3.1 Синтаксический анализ кода

Центральным компонентом синтаксического анализа в разрабатываемой системе выступает класс `JsRecipeParser`, реализующий преобразование исходных JavaScript-файлов во внутреннее представление, пригодное для дальнейшей обработки. Данный класс инкапсулирует логику проверки структурной корректности входного скрипта, выделения лямбда-выражения, содержащего определения рецептов, и координирует применение цепочки экстракторов к обнаруженным синтаксическим конструкциям. На вход класс принимает либо путь к файлу, либо непосредственно строку с кодом, а на выходе формирует список объектов, наследующих `BaseRecipeConfiguration`, каждый из которых представляет один распознанный рецепт. На рисунке 15 представлена диаграмма деятельности для класса `JsRecipeParser` [14]. Она позволяет описывать бизнес процессы в виде набора шагов [15].

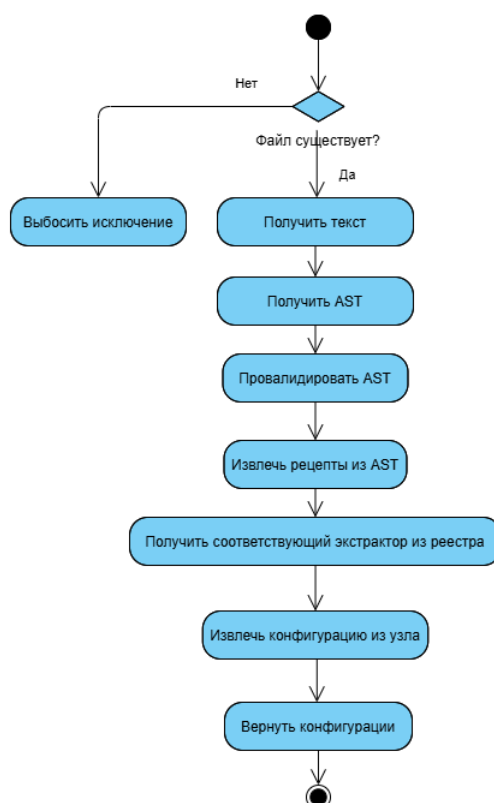


Рисунок 15 – Диаграмма деятельности для класса `JsRecipeParser`

Первым этапом работы JsRecipeParser является верификация структуры файла на предмет соответствия поддерживаемому шаблону. Ожидается, что весь код представлен единственным вызовом вида `ServerEvents.recipes(event => { ... })`. Несколько утверждений верхнего уровня, отсутствие ожидаемого вызова, иная сигнатура метода немедленно прерывают разбор с формированием диагностического исключения, в котором указывается точная причина несоответствия. Такой строгий контроль на входе исключает неоднозначности и гарантирует, что последующая обработка будет выполняться только над корректно структурированными данными.

После успешного прохождения верификации синтаксический анализатор извлекает тело лямбда-функции, переданной в качестве аргумента, и передаёт каждый оператор этого тела реестру экстракторов `ExtractorRegistry`. Реестр последовательно опрашивает зарегистрированные экстракторы, и первый, способный обработать узел абстрактного синтаксического дерева, возвращает готовый объект конфигурации. Таким образом, синтаксический анализатор не содержит собственной логики интерпретации конкретных типов рецептов, а делегирует эту задачу специализированным компонентам, обеспечивая гибкость и расширяемость системы.

Алгоритмическая последовательность действий JsRecipeParser наглядно отражена на диаграмме деятельности, представленной на рисунке 15. Диаграмма демонстрирует ветвление на этапе верификации, обход инструкций тела лямбды, обращение к реестру экстракторов и накопление итоговой коллекции конфигураций.

Реализованный таким образом синтаксический анализатор полностью выполняет возложенную на него задачу: он преобразует плоский текст скрипта KubeJS в множество типизированных объектов, готовых к верификации и последующей генерации выходного JavaScript-кода. Строгая проверка входного формата в сочетании с делегированием семантического анализа реестру экстракторов обеспечивает высокую надёжность и сопровождаемость компонента, а чёткое разделение ответственности делает возможной

независимую доработку как самого синтаксического анализатора, так и подключаемых к нему экстракторов.

3.2 Реализация экстракции данных из произвольных JSON-структур

Когда рецепт описан произвольной JSON-структурой, за его разбор отвечает класс CustomExtractor. Такой формат встречается во многих игровых дополнениях. Специализированные экстракторы для «Create» работают с фиксированным набором методов. CustomExtractor устроен иначе: ему не важно, какое дополнение сгенерировало вызов. «GregTech» и любое другое расширение, поддерживающее JSON-рецепты, он обработает одинаково.

Проверка начинается с одного вопроса: является ли текущий узел AST вызовом функции custom на объекте event. За это отвечает приватный метод IsEventCustom. Метод запрашивает у AstHelper два значения: имя объекта и имя свойства. Оба должны точно совпасть с ожидаемыми, иначе узел будет пропущен. Совпадение должно быть точным: только при полном соответствии экстрактор берётся за обработку узла. Конструкции схожей формы, но иной семантики при этом отсеиваются.

Метод Extract преобразует подходящий узел в типизированную конфигурацию CustomRecipeConfig. Первый аргумент вызова должен быть ObjectExpression. Если это так, AstHelper.ExtractObjectExpression строит из него иерархию JsonNode. Готовый JSON-объект уходит в CustomRecipeConfig.FromJsonObject. Тот собирает экземпляр конфигурации, сохраняя исходную структуру данных в неизменном виде. Если аргумент не является ObjectExpression или объект окажется пустым, метод возвращает null [16].

Логика работы класса показана на диаграмме деятельности (рисунок 16). На ней видны точки ветвления: проверка узла на корректность, извлечение ObjectExpression для дальнейших преобразований, построение JsonNode и финальное создание конфигурации.

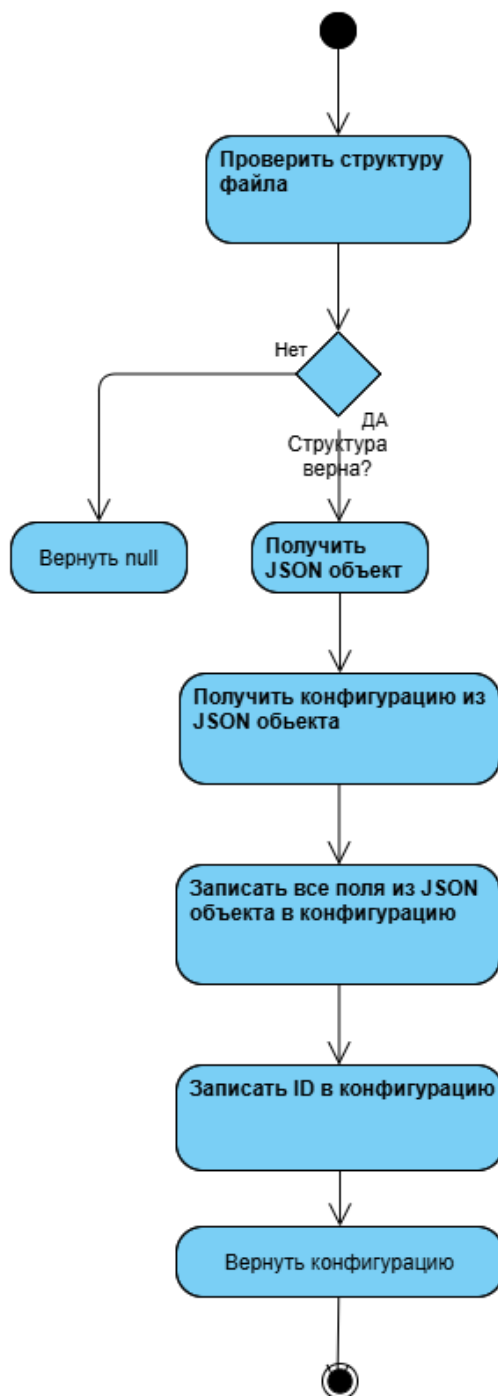


Рисунок 16 – Диаграмма деятельности класса CustomExtractor

Класс решает задачу полностью: JSON-описание рецепта на входе, типизированная конфигурация на выходе. Некорректные аргументы отсеиваются раньше, чем успевают попасть в следующий этап обработки. Расширение-источник не имеет значения. Компонент работает в любом сценарии без правок.

3.3 Реализация процесса извлечения и сохранения конфигураций рецептов

Один из практичных сценариев работы системы: взять готовый JavaScript-файл, преобразовать его в редактируемую конфигурацию и сохранить как параметризованный шаблон. В этом процессе участвуют несколько уже описанных компонентов; их связывает класс `CustomRecipeConfig`, который одновременно хранит данные и служит объектом редактирования. Рисунок 17 показывает упрощённую версию сценария: один рецепт в исходном файле, акцент на сохранении результата как шаблона без финальной генерации кода.

Сценарий запускает пользователь: через командную строку он передаёт путь к нужному JavaScript-файлу. Программа перенаправляет его синтаксическому анализатору `JsRecipeParser`. Тот читает файл и сразу проверяет структуру: файл должен соответствовать единственному поддерживаемому формату вызова. Диаграмма последовательности показывает принцип работы системы [17].

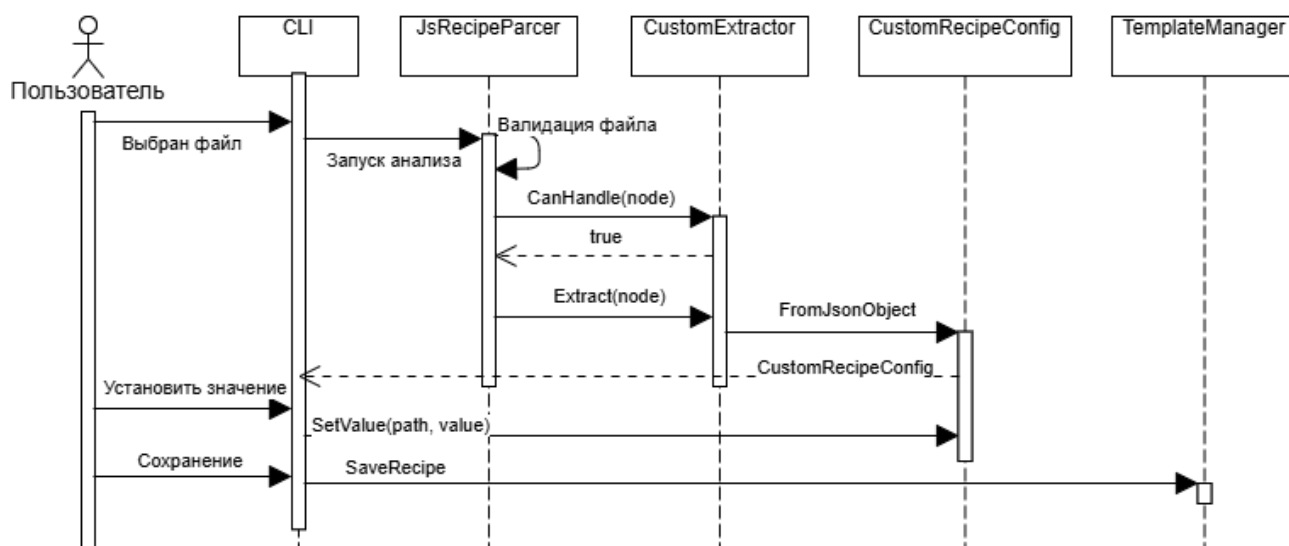


Рисунок 17 – Диаграмма последовательности для сценария сохранения шаблонного рецепта

Пройдя проверку, анализатор обходит тело лямбда-функции. Каждый найденный узел AST передаётся реестру экстракторов, который опрашивает зарегистрированные компоненты по очереди. В данном сценарии узел

перехватывает CustomExtractor: его метод распознаёт вызов event.custom(...) по сигнатуре. Найдя совпадение, анализатор вызывает Extract. Метод достаёт JSON-объект из аргумента функции и передаёт его в CustomRecipeConfig.FromJsonObject. На выходе получается экземпляр CustomRecipeConfig: точная копия структуры из исходного кода. Уже написанный рецепт сразу становится отправной точкой для правок.

Получив список конфигураций, программа показывает пользователю извлечённую структуру. Нужные поля меняются через метод установки значения: он принимает строковый путь к элементу JSON-дерева и новое значение. Знать JavaScript или синтаксис KubeJS при этом не нужно. Пользователь работает с понятиями предметной области: ингредиенты, результаты, типы обработки. Всё, что связано с сериализацией, спрятано внутри модели.

После редактирования пользователь сохраняет конфигурацию как шаблон. Программа передаёт вызов менеджеру шаблонов: имя, категорию и сам объект конфигурации. Менеджер запрашивает у конфигурации JSON-представление через toJson(), добавляет метаданные (название, автор, описание) и записывает файл в нужную директорию. Один экземпляр CustomRecipeConfig покрывает оба случая: оперативное редактирование и долгосрочное хранение в читаемом формате. Ручное форматирование JSON здесь не нужно вовсе.

Сравнение с привычным подходом объясняет, почему это важно. Раньше новый рецепт писался с нуля: отступы, кавычки, вложенные структуры вручную. Для многокомпонентных рецептов дополнений наподобие «Create» это занимало десятки минут. Одна пропущенная запятая – и игра не загружает файл, а внятной диагностики в «Minecraft» практически нет.

С разработанной системой процесс другой. Рабочий рецепт извлекается один раз, правятся только нужные поля (например, входной предмет или количество), результат сохраняется как шаблон. На повторное использование уходят секунды, а ошибки из-за ручного набора исключены.

Со временем в файловой системе накапливается библиотека шаблонов. Новый участник команды не тратит время на изучение синтаксиса KubeJS: он открывает готовые заготовки и заполняет нужные поля. Когда состав рецептов часто пересматривается, это ощутимо ускоряет работу. Редактирование, сохранение и генерация кода доступны в любой момент; каждое изменение можно проверить сразу и при необходимости отменить.

Механизм извлечения, редактирования и сохранения в CustomRecipeConfig работает корректно на техническом уровне. Но важнее другое: он меняет сам подход к разработке рецептов. Внутренняя сложность скрыта, структура верифицируется автоматически, шаблоны хранятся рядом и всегда доступны. Разработчику не нужно держать в голове синтаксические детали. Можно сосредоточиться на логике рецептов.

3.4 Реализация менеджера шаблонов

Центральным компонентом, обеспечивающим долговременное хранение и повторное использование извлечённых из кода конфигураций, выступает статический класс TemplateManager. Его назначение заключается в преобразовании экземпляра CustomRecipeConfig в структурированный шаблон с автоматической классификацией полей и последующей записью полученного представления в файловую систему, а также в обратном восстановлении конфигурации из такого шаблона. Диаграмма деятельности, представленная на рисунке 18, отображает последовательность операций, выполняемых этим методом. Основной упор делается на проверку условностей и соблюдение корректности. Обязательна проверка на спец-символы, поскольку шаблонные рецепты KubeJS не поддерживают их, а работают лишь с буквенными символами. Затем происходит извлечение структуры, для семантического сравнения, сохранение метаданных. И только после всех операций происходит сохранение в корректный JSON для последующего сохранения. Это позволяет в будущем экономить время, просто записывая все данные в поля, вместо полноценного написания всего кода.

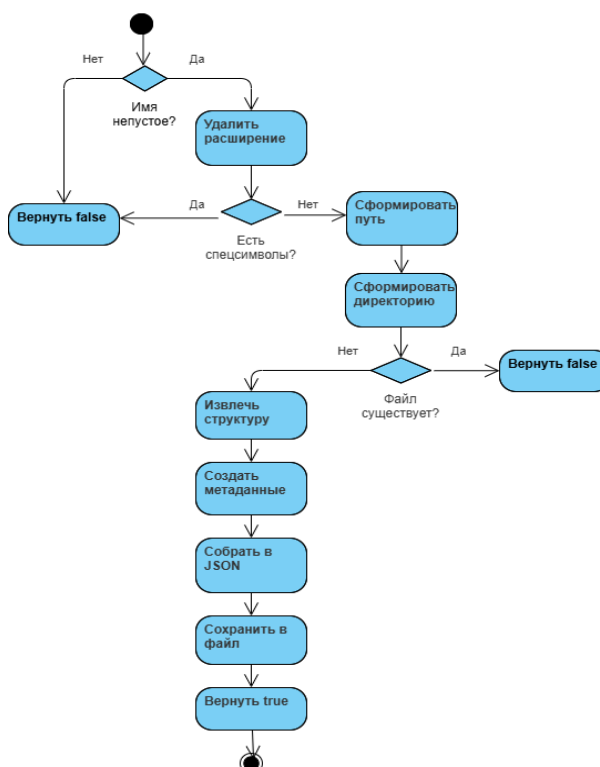


Рисунок 18 – Диаграмма деятельности метода сохранения шаблона

Первой операцией при вызове `SaveTemplate` является строгая проверка переданного имени шаблона. Имя не должно быть пустой строкой или состоять исключительно из пробельных символов; кроме того, из него удаляется расширение, если оно было указано, и выполняется фильтрация недопустимых символов. Разрешёнными остаются только буквенно-цифровые знаки и символ подчёркивания. При нарушении любого из этих условий метод немедленно возвращает логическое значение `false`, сигнализируя о невозможности сохранения. Данная проверка предотвращает создание файлов с некорректными именами, способных вызвать проблемы при последующей загрузке или отображении в пользовательском интерфейсе.

После успешной верификации имени формируется полный путь к целевому файлу. На основе базового каталога и переданной категории строится промежуточная директория, которая при необходимости создаётся вместе со всеми родительскими папками. Это гарантирует, что иерархическая организация шаблонов по категориям будет поддерживаться автоматически, без дополнительных действий со стороны пользователя. Далее выполняется

проверка существования файла: если файл с таким именем уже присутствует, а флаг перезаписи не установлен, то сохранение прерывается, что позволяет избежать случайной потери ранее сохранённых данных.

Ключевым отличием шаблона от непосредственного сохранения готового рецепта является этап извлечения структуры с классификацией полей, реализованный во внутреннем методе `ExtractStructure`. Для каждого пути, полученного из `CustomRecipeConfig`, определяется его тип и текущее значение, после чего в соответствии с набором эвристических правил полю присваивается одна из трёх категорий: фиксированная, стандартная или требуемая. Поля, классифицированные как фиксированные, сохраняют своё исходное значение и в дальнейшем не подлежат редактированию. К ним относится, например, тип рецепта. Поля стандартной категории сохраняют текущее значение либо подставляют значение по умолчанию, если оно отсутствует; они могут быть изменены пользователем, но уже содержат разумную заготовку. Поля требуемого типа намеренно очищаются или помечаются как требующие обязательного заполнения, тем самым формируя минимально необходимый набор входных данных, которые пользователь должен предоставить для получения корректного рецепта. Такая автоматическая классификация полностью освобождает разработчика от необходимости вручную определять, какие элементы структуры критичны, а какие могут оставаться неизменными.

Следующим шагом на основе извлечённых путей и метаданных, включающих имя, категорию, описание и сведения об авторе, формируется единый JSON-объект. Метаданные помещаются в секцию метаданных. Классифицированные пути помещаются в секцию структуры, что обеспечивает чёткое разделение служебной информации и данных шаблона. Полученный объект сериализуется с форматированием отступов и безопасным экранированием символов, после чего атомарно записывается в файл. Завершая свою работу, метод возвращает `true`, подтверждая успешное сохранение шаблона.

Реализованный таким образом подход к сохранению формирует фундамент для итеративной разработки скриптов, радикально сокращая время, затрачиваемое на создание новых рецептов. При ручном написании JavaScript-кода приходится держать в голове, и синтаксис, и отступы, и структуру JSON. С шаблоном весь процесс сводится к заполнению нескольких обязательных полей. Шаблон сохраняется один раз, а затем используется многократно: достаточно загрузить его, поправить нужные значения и сразу получить готовый код. Разработка превращается в короткий повторяющийся цикл «загрузка – правка – генерация». Когда таких шаблонов накапливается много, эффект только усиливается: типовые структуры рецептов больше не пишут с нуля, а лишь подгоняют под очередную задачу.

Отдельно стоит сказать об ошибках: автоматическая классификация полей убирает целый пласт проблем. Пользователю не нужно помнить, как именно называются свойства JSON, в какой они вложенности и какого типа значения допустимы – это всё уже зашито в шаблоне, а поля, обязательные для заполнения, явно отмечены. Здесь контроль корректности выполняется на уровне методов `CreateProperty` и `SetValue`, которые гарантируют соответствие сохраняемых данных ожидаемой структуре. В результате вероятность получения неработоспособного скрипта на выходе сводится к минимуму, а процесс отладки, традиционно болезненный в среде игры «Minecraft» из-за отсутствия внятных диагностических сообщений KubeJS, практически исключается. Таким образом, `TemplateManager` задаёт новый уровень абстракции, при котором шаблон рецепта становится самодокументированным и безопасным в использовании элементом, непосредственно поддерживающим итеративный и командный стиль разработки.

3.5 Реализация подсистемы извлечения игровых метаданных и графических ресурсов

Завершающим компонентом уровня интеграции с игровой средой является библиотека `LogExtractorLibrary`, центральное место в которой занимает класс

ItemFluidWithIconExtractor. Он унаследован от абстрактного базового класса BaseExtractor, задающего единый контракт, и отвечает за формирование структурированного набора сведений обо всех предметах и жидкостях, присутствующих в клиентской части игры. В отличие от конфигураций рецептов, которые существуют лишь в оперативной памяти в процессе обработки скриптов, результаты работы данного экстрактора образуют долгоживущий кэш метаданных, используемый на всех последующих этапах – от разрешения идентификаторов предметов при синтаксическом анализе до отображения локализованных названий и иконок в пользовательском интерфейсе.

Алгоритмическая основа метода ExtractAsync построена на последовательном проходе по двум внешним источникам: JSON-файлу, содержащему атрибуты игровых объектов (идентификатор, локализованное имя, модификация-источник, тип, теги), и директории, хранящей растровые изображения в формате PNG. На первом этапе выполняется десериализация метаданных в словарь, ключом которого выступает строковый идентификатор. Это позволяет в дальнейшем для каждого графического файла мгновенно отыскать соответствующий набор атрибутов. Параллельно инициализируются списки предметов и жидкостей, куда будут помещаться экземпляры MinecraftItem.

Второй этап представляет собой обход всех изображений, полученных сканированием папки иконок. Для каждого файла из имени извлекается идентификатор с помощью вспомогательного метода, реализующего синтаксический разбор строк с особым форматом. Попутно выполняется фильтрация: исключаются файлы, содержащие NBT-теги (определяются по наличию фигурных скобок), и дублирующие текстуры жидкостей (определяются по префиксу «flowing»). Такая фильтрация гарантирует, что в итоговую коллекцию попадут только статичные изображения, соответствующие реальным игровым объектам, а не их динамическим состояниям.

Если для извлечённого идентификатора в словаре метаданных найдена запись, то её поля используются для заполнения объекта `MinecraftItem`. В противном случае формируется запись с резервными значениями: идентификатор дублируется в качестве названия, источник помечается как неизвестный, а тип определяется по префиксу имени файла. Такой подход обеспечивает устойчивость к неполноте метаданных и позволяет обрабатывать иконки, добавленные модификациями, которые не экспортировали соответствующий JSON. После связывания метаданных и пиксельных данных изображение загружается в память как массив байтов, и готовый объект помещается в список предметов либо жидкостей в зависимости от типа, указанного в метаданных или имени файла. Итоговый результат возвращается в виде экземпляра `ItemsAndFluidsData`, содержащего оба списка.

Реализованная таким образом подсистема полностью изолирует от остального приложения детали взаимодействия с файловой структурой игры. Основное приложение получает готовый словарь объектов, не нуждаясь в знании форматов имён файлов `IconExporter` или структуры JSON-экспорта. Это не только упрощает разработку и тестирование ядра системы, но и гарантирует, что при изменении внутреннего представления метаданных в будущих версиях игры модификации потребуются лишь в отдельной библиотеке, в то время как основной проект `KubeAutomation` останется неизменным. Стоит отметить, что игровая среда часто дублирует информацию из-за необходимости хранить состояния игровых объектов. Хотя такое дублирование не влияет на корректность извлечения, оно создаёт избыточность, которую важно учитывать при фильтрации. Эта избыточность влияет на опыт взаимодействия с системой: `KubeJS` изначально не поддерживает работу с NBT (Named Binary Tag) данными. Описанная последовательность действий наглядно представлена на диаграмме деятельности, приведённой на рисунке 19. На ней выделены узлы ветвления, соответствующие проверкам существования файла метаданных, наличия записи в словаре и фильтрам по NBT и дублирующим текстурам, а также цикл обхода изображений и формирование результирующих списков.

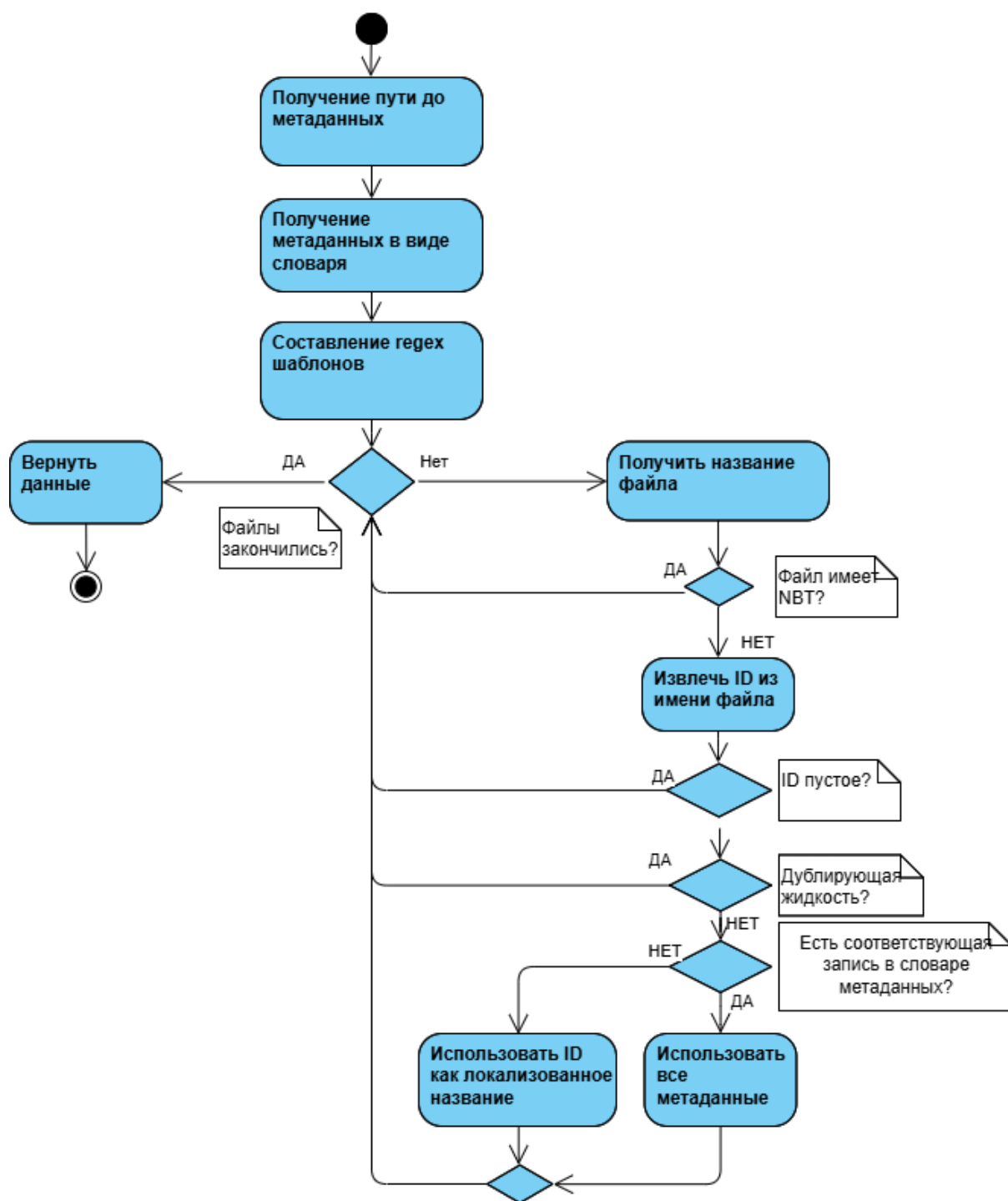


Рисунок 19 – Диаграмма деятельности метода извлечения игровых данных

Загрузка полного набора предметов и жидкостей с их иконками открывает возможность для последующего визуального конструирования рецептов, а точная привязка идентификаторов к локализованным названиям исключает ошибки, связанные с ручным вводом строковых литералов, что в совокупности повышает надёжность и удобство работы с системой.

4 Контроль стандартов качества программных решений

4.1 Выбор методов тестирования комплекса приложений

Комплекс приложений для генерации игровых скриптов позволяет автоматически и интерактивно генерировать код. Отличительная особенность комплекса в модульной архитектуре с чётким разделением ответственности [18].

Генератор скриптов реализован как модульное С#-приложение со слоистой архитектурой. Ввиду наличия доступа к исходному коду и необходимости проверки внутренней логики, для данного компонента применено модульное тестирование методом «белого ящика» с использованием техник покрытия условий и анализа граничных значений согласно ГОСТ Р ИСО/МЭК 25010 [19].

Библиотека получения игровых данных представляет собой наиболее сложное сочетание решений. Это С#-библиотека, подключаемая в главное приложение. Она состоит из нескольких функций, которым требуются дополнительные данные. Например функция извлечения игровых предметов, названий и изображений требует наличия установленной сторонней игровой модификации «icon-exporter.jar» в игровой среде, который создаёт папку со всеми ресурсами в корне игры при вызове отдельных функций внутри игровой сессии. Модификация функционирует в изолированной среде игровой сессии, что делает невозможным прямое модульное тестирование. В связи с этим для данной библиотеки применено приёмочное тестирование в натурной среде с фиксацией результатов [20].

Так как генерируемый код для изменения игры находится внутри игровой среды, и исполняется самим сервером, то игровые рецепты могут быть изменены, удалены, добавлены только во время работы сессии. Например, KubeJS создаёт уникальный идентификатор игрового рецепта самостоятельно. Автоматическая верификация сгенерированных рецептов затруднена ввиду динамической природы игрового окружения и отсутствия решений для

программной проверки рецепта. В связи с этим, финальная верификация осуществляется приёмочным тестированием натурным методом [21].

4.2 Тестирование верификации и генерации кода для стандартных игровых рецептов

Объектом тестирования выступает класс `MinecraftStandardRecipeConfig`, предназначенный для генерации строк кода KubeJS, описывающих стандартные рецепты обработки предметов в игре. Класс предоставляет два публичных метода для верификации входных параметров и генерирование кода на основе верифицированных параметров.

Управляющий граф прецедента «Генерировать рецепт» изображен на рисунке 20.

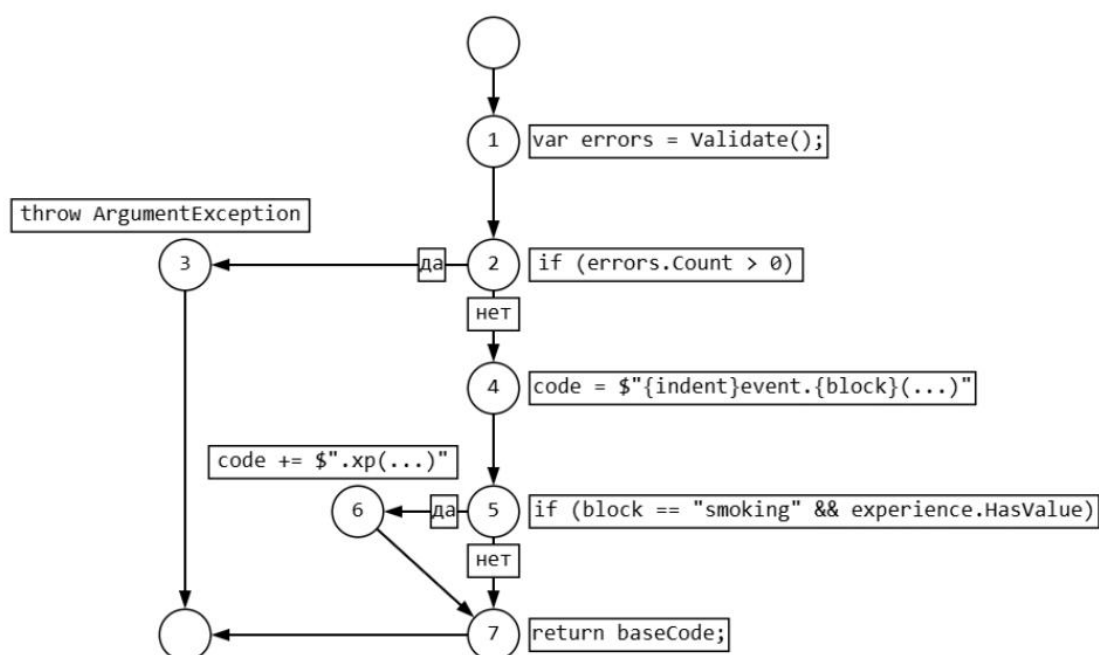


Рисунок 20 – Управляющий граф прецедента «Генерировать рецепт»

Для обеспечения полноты тестирования применён метод «белого ящика», предполагающий анализ внутренней структуры кода. Ключевым элементом функции генерации является условный оператор, определяющий добавление модификатора опыта. Данное составное условие содержит два атомарных под-условия.

Для достижения полного покрытия условий необходимо обеспечить выполнение каждого под-условия как на истинное, так и на ложное значение [22]. Соответствующие тестовые сценарии представлены в таблице 4.

Таблица 4 – Результат тестирования методом покрытия маршрутов

Маршрут	Тест	Предполагаемый результат	Фактический результат
1, 2, 3	Ошибка верификации входных данных.	Функция выбросила исключение.	Успешно. На рисунке 21.
1, 2, 4, 5, 7	Создан базовый код без дополнительных данных об опыте.	Функция вернула строку без дополнительных данных	Успешно. На рисунке 22.
1, 2, 4, 5, 6, 7	Создан код с дополнительными данными об опыте.	Функция вернула строку с дополнительными данными	Успешно. На рисунке 23.

Для исчерпывающей верификации требуется минимизировать вероятность пропуска ошибок, связанных с некорректной комбинацией логических состояний. Представленные в таблице 4 маршруты графа иллюстрируют типовые пути выполнения, охватывающие как ситуацию ошибки верификации, так и штатные ветви с добавлением дополнительных данных или без такового. Полученные фактические результаты подтверждают ожидаемое поведение функции генерации во всех трёх случаях, что свидетельствует о корректной реализации логики и отсутствии регрессионных изменений на проверенных маршрутах.

На рисунке 21 изображена часть справочной информации из системы автоматизированного тестирования с успешными проверками граничных значений и покрытия маршрутов. Основной упор делается на сравнении ожидаемого и фактического результата. Система тестируется как при подаче верных значений, так и при проверке неверных значений. Это гарантирует высокую отказоустойчивость

```

[GROUP] Валидация: Null и пустые параметры
[TEST] Validate: outputItem не может быть null
PASS: Ошибка обнаружена
[TEST] Validate: inputItem не может быть null
PASS: Ошибка обнаружена
[TEST] Validate: block не может быть пустым
PASS: Ошибка обнаружена
[TEST] Validate: block не может быть пустым
PASS: Ошибка обнаружена
[TEST] Validate: Blocks не содержит
PASS: Ошибка обнаружена
[GROUP] Валидация: Граничные значения Output Amount
[TEST] Validate: OutputAmount=0 (Ниже минимума)
PASS: Результат совпадает с ожидаемым
[TEST] Validate: OutputAmount=1 (Минимум (граница))
PASS: Результат совпадает с ожидаемым
[TEST] Validate: OutputAmount=32 (Середина диапазона)
PASS: Результат совпадает с ожидаемым
[TEST] Validate: OutputAmount=64 (Максимум (граница))
PASS: Результат совпадает с ожидаемым
[TEST] Validate: OutputAmount=65 (Выше максимума)
PASS: Результат совпадает с ожидаемым
[GROUP] Валидация: Input Amount должен быть 1
[TEST] Validate: InputAmount=0 (Ноль)
PASS: Результат совпадает с ожидаемым
[TEST] Validate: InputAmount=1 (Единственное допустимое)
PASS: Результат совпадает с ожидаемым
[TEST] Validate: InputAmount=2 (Больше единицы)
PASS: Результат совпадает с ожидаемым
[TEST] Validate: InputAmount=64 (Максимум)
PASS: Результат совпадает с ожидаемым
[GROUP] Валидация: Experience границы [0, 100]
[TEST] Validate: Experience=-1 (Отрицательный)
PASS: Результат совпадает с ожидаемым

```

Рисунок 21 – Результаты автоматизированного тестирования входных данных

На рисунке 22 изображена проверка всех типов блоков.

```

[GROUP] Генерация: Все типы блоков
[TEST] Generate: smelting
PASS: event.smelting('minecraft:iron', 'minecraft:ore')
[TEST] Generate: blasting
PASS: event.blasting('minecraft:gold', 'minecraft:ore')
[TEST] Generate: smoking
PASS: event.smoking('minecraft:cooked_beef', 'minecraft:beef')
[TEST] Generate: stonecutting
PASS: event.stonecutting('minecraft:stone_slab', 'minecraft:stone')

```

Рисунок 22 – Результаты автоматизированного тестирования генерации кода
без дополнительных данных

На рисунке 23 изображена проверка вветвления для прецедента.

```

[GROUP] Генерация: Experience для smoking
[TEST] Generate: Smoking exp=0
      PASS: Содержит .xp(0)
[TEST] Generate: Smoking exp=0,5
      PASS: Содержит .xp(0.5)
[TEST] Generate: Smoking exp=1
      PASS: Содержит .xp(1)
[TEST] Generate: Smoking exp=100
      PASS: Содержит .xp(100)
[GROUP] Форматирование: Культура и отступы
[TEST] Generate: InvariantCulture exp=0,75
      PASS: Используется точка (InvariantCulture)

```

Рисунок 23 – Результат автоматизированного тестирования генерации кода с вветвлением для прецедента «Создать рецепт»

Приведённые в таблице 4 маршруты показывают, какие пути выполнения кода были задействованы при тестировании. Это наглядно подтверждает, что все логические ветви функции покрыты проверками. Однако одних маршрутов недостаточно: из-за сложности системы нужно дополнительно протестировать граничные значения, которые принимают входные параметры методов. К этим параметрам относятся тип рецепта, выходные и входные предметы, а также необязательный опыт. Для каждого из них сформулированы явные ограничения.

Детальный анализ границ помогает убедиться, что экстремальные значения величины обрабатываются корректно. В первую очередь, проверялись ситуации, когда количество предметов выходит за нижнюю и верхнюю границу разрешённого диапазона, а также случай для нулевого значения опыта – такой вариант разрешен логикой предметной области только для отдельных типов рецептов.

Для простейших рецептов возможны лишь четыре варианта. Совместное применение комбинаторного тестирования граничных значений и маршрутного покрытия даёт достаточную уверенность в надёжности модуля.

В таблице 5 указаны все ограничения параметров для прецедента «Создать рецепт».

Таблица 5 – Параметры прецедента «Создать рецепт»

Параметр	Описание	Допустимые значения	Ограничение
block	Игровой объект, на котором выполняется создаваемый рецепт	smelting, blasting, smoking, stonecutting	Только из списка Blocks
outputItem.Amount	Количество выходного предмета в рецепте	[1, 64]	Целое число во включительном диапазоне
inputItem.Amount	Количество входного предмета в рецепте	1	Строго равно единице
experience	Количество получаемого игроком опыта при успешном совершении рецепта	[0, 100]	Целое число во включительном диапазоне. Только для block = smoking
inputItem.ItemId	Идентификатор входного предмета	$\neq$ outputItem.ItemId	Идентификаторы должны различаться

Эти ограничения, продиктованные правилами самой игры, нужны для того, чтобы сгенерированный рецепт заведомо работал в игровой сессии без ошибок. Граф строится для прецедента «Создать рецепт», но учитывает вызов прецедента «Верифицировать данные» как составной узел. В дополнение представленным данным в таблице 6, следует подчеркнуть, что проверка значений охватывает сценарии с некорректными типами данных, однако в рамках исследования внимание уделяется целочисленным границам, поскольку правила явно специфицируют дискретные множества допустимых величин.

Таблица 6 – Проверка граничных значений

Проверка параметра	Диапазон	Подано	Ожидаемый результат	Фактический результат
outputItem.Amount	[1, 64]	0	Ошибка	Ошибка
outputItem.Amount	[1, 64]	1	Успешно	Успешно
outputItem.Amount	[1, 64]	64	Успешно	Успешно
outputItem.Amount	[1, 64]	65	Ошибка	Ошибка
experience	[0, 100]	-1	Ошибка	Ошибка
experience	[0, 100]	0	Успешно	Успешно
experience	[0, 100]	100	Успешно	Успешно
experience	[0, 100]	101	Ошибка	Ошибка

В таблице 7 приведены сводные результаты автоматизированного модульного тестирования.

Таблица 7 – Сводные результаты автоматизированного модульного тестирования

Группа тестов	Всего	Пройдено	Провалено	Покрытие
Верификация пустых значений	2	2	0	100%
Верификация граничных значений для количества предметов	5	5	0	100%
Верификация количества опыта	10	10	0	100%
Генерация с определенными типами	4	4	0	100%
Генерация с количеством типов	4	4	0	100%
Форматирование	1	1	0	100%
ИТОГО	32	32	0	100%

Результаты тестирования показали, что модуль генерации простейших рецептов работает успешно. Тестирование не показало несоответствия между ожидаемым и фактическим результатом работы модуля приложения генерации кода.

4.3 Тестирование целостности данных при итеративном преобразовании JSON-структур

Целью данного этапа тестирования является обеспечение целостности данных при преобразовании объектов конфигурации рецептов в формат JSON и обратно в исходные данные при генерации JSON. Это важная часть итеративного подхода при разработке игровых скриптов, балансировке значений и реализации игрового дизайна. Ключевыми задачами выступают: проверка корректности сериализации вложенных структур, верификация сохранения метаданных шаблонов, гарантия отсутствия потерь данных при циклических преобразованиях «данные в JSON, а затем обратно в данные».

Система тестирования построена по модульному принципу. Каждый тестовый сценарий выполняется в изолированной временной директории, что исключает влияние тестов на основную файловую систему. Компонент CustomRecipeConfig отвечает за построение JSON-структуры, TemplateManager отвечает за сохранение и загрузку с классификацией полей, а функция семантического сравнения JsonSemanticEquals() обеспечивает корректную верификацию результатов независимо от порядка свойств.

Тестовые сценарии охватывают три уровня проверки: юнит-тесты отдельных методов доступа к данным, интеграционные тесты взаимодействия компонентов сериализации, функциональные тесты бизнес-логики классификации полей шаблонов. Каждый сценарий включает детальное логирование входных данных, промежуточных состояний и результатов для упрощения отладки. В таблице 8 приведена классификация всех тестовых сценариев для тестирования надежности итеративных преобразований, созданных в ходе разработки программного комплекса.

Таблица 8 – Классификация тестовых сценариев

№	Наименование теста	Проверяемый компонент	Критерий успеха
1	Прямая проверка функции GetValue()	CustomRecipeConfig.GetValue()	Корректное извлечение значений по путям с индексами массивов
2	Roundtrip-тестирование простого шаблона	SaveRecipe, LoadTemplate	JSON идентичен оригиналу после полного цикла
3	Roundtrip-тестирование вложенных объектов	Вложенные массивы. Например «sequence[n].ingredients[n]»	Сохранение структуры и значений глубоко вложенных элементов
4	Сохранение метаданных	TemplateMetadata сериализация/десериализация	Автор, описание, категория сохраняются без изменений

В таблице 9 приведены результаты тестов. Зафиксированные отладочными средствами системы автоматизированного тестирования. Перед запуском тесты были сгруппированы по классификации из таблицы 8. В ходе каждого сценария система тестирования автоматически собирала подробную информацию: какие параметры подавались на вход, в каком промежуточном состоянии находились объекты и каким получился итог.

Отдельно проверялось, выдерживают ли компоненты многократные циклы обработки – итеративные преобразования сложных структур с вложенными массивами и метаданными приводят к потере данных или ссылок. Поэтому здесь требовалась особая аккуратность. Каждая итерация фиксировалась и сравнивалась с эталоном для выявления мельчайших расхождений. Чтобы исключить влияние посторонних факторов, тесты выполнялись в изолированной среде, не связанной с рабочей инфраструктурой.

Таблица 9 – Результаты комплексного тестирования всех сценариев

Тест	Ожидаемый результат	Фактический результат	Статус	Результат
Прямая проверка GetValue	pattern[0] = AAA	pattern[0] = AAA	Пройден	На рисунке 24
Roundtrip: простой рецепт	JSON идентичен	JSON идентичен	Пройден	На рисунке 24
Roundtrip: сложный рецепт	Вложенность сохранена	Вложенность сохранена	Пройден	На рисунке 25
Сохранение метаданных	Автор = TestAuthor	Автор = TestAuthor	Пройден	На рисунке 26
ИТОГО	4/4	4/4	100%	Успешно

Рисунок 24 иллюстрирует полученные результаты.

```
[TEST] Start: Прямая проверка GetValue
[DEBUG] pattern[0] = 'AAA' (type: String)
[DEBUG] pattern[1] = 'ABA' (type: String)
[TEST] Pass

[TEST] Start: Roundtrip: Простой рецепт (РЕЖИМ РЕЦЕПТОВ)
[DATA] Original JSON:
{
  "type": "minecraft:crafting_shaped",
  "pattern": [
    "AAA",
    "ABA"
  ]
}
[DEBUG] ExtractStructureFull: 3 нутей
[PATH] Path='type' Type=String Field=Fixed Value='minecraft:crafting_shaped'
[PATH] Path='pattern[0]' Type=String Field=Fixed Value='AAA'
[PATH] Path='pattern[1]' Type=String Field=Fixed Value='ABA'
[SAVE] SaveRecipe result: True
[DEBUG] Loaded paths: 3
[PATH] Path='type' Type=String Value='minecraft:crafting_shaped' (CLR: JsonElement)
[PATH] Path='pattern[0]' Type=String Value='AAA' (CLR: JsonElement)
[PATH] Path='pattern[1]' Type=String Value='ABA' (CLR: JsonElement)
[SET] type = 'minecraft:crafting_shaped' (type: String)
[SET] pattern[0] = 'AAA' (type: String)
[SET] pattern[1] = 'ABA' (type: String)
[LOAD] Success: True, Error:
[DATA] Loaded JSON:
{
  "type": "minecraft:crafting_shaped",
  "pattern": [
    "AAA",
    "ABA"
  ]
}
[TEST] Pass | Duration: 56ms
```

Рисунок 24 – Результат тестирования функции получения значения

Следующий тест касался более сложных структур. На рисунке 25 видно, как система справляется с многократно вложенными данными.

```
[DEBUG] Extracted 8 paths for saving:
[PATH] type = 'create:sequenced_assembly'
[PATH] loops = '3'
[PATH] sequence[0].type = ''
[PATH] sequence[0].ingredients[0].item = ''
[PATH] sequence[1].type = ''
[PATH] sequence[1].ingredients[0].item = ''
[PATH] sequence[2].type = ''
[PATH] sequence[2].ingredients[0].item = ''
[STEP 3] Загрузка шаблона (LoadTemplate)...
[DEBUG] Loaded paths: 8
[PATH] Path='type' Type=String Value='create:sequenced_assembly' (CLR: JsonElement)
[PATH] Path='loops' Type=Int Value='3' (CLR: JsonElement)
[PATH] Path='sequence[0].type' Type=String Value='' (CLR: JsonElement)
[PATH] Path='sequence[0].ingredients[0].item' Type=String Value='' (CLR: null)
[PATH] Path='sequence[1].type' Type=String Value='' (CLR: JsonElement)
[PATH] Path='sequence[1].ingredients[0].item' Type=String Value='' (CLR: null)
[PATH] Path='sequence[2].type' Type=String Value='' (CLR: JsonElement)
[PATH] Path='sequence[2].ingredients[0].item' Type=String Value='' (CLR: null)
[SET] type = 'create:sequenced_assembly' (type: String)
[SET] loops = '3' (type: Double)
[SET] sequence[0].type = '' (type: String)
[SET] sequence[1].type = '' (type: String)
[SET] sequence[2].type = '' (type: String)
[LOAD] Success: True, Error: none
```

Рисунок 25 – Результат тестирования сложных вложенных структур

Завершал серию тест, связанный с сохранением метаданных шаблона (рисунок 26).

```
[DATA] Raw saved JSON (first 500 chars):
{
  "$metadata": {
    "Name": "test_meta",
    "Id": "test_meta",
    "Category": "test",
    "Description": "Тестовое описание",
    "Author": "TestAuthor",
    "Version": "1.0",
    "CreatedDate": "2026-05-25T11:21:59.6574888Z",
    "ModifiedDate": "2026-05-25T11:21:59.6574889Z"
  },
  "$structure": {
    "paths": [
      {
        "Path": "type",
        "Type": 0,
        "FieldType": 0,
        "Value": "test:type"
      }
    ]
  }
}
[STEP 3] Загрузка шаблона (LoadTemplate)...
[DEBUG] Loaded paths: 1
[PATH] Path='type' Type=String Value='test:type' (CLR: JsonElement)
[SET] type = 'test:type' (type: String)
[LOAD] Success: True
[LOAD] ErrorMessage: 'none'
[STEP 4] Валидация метаданных...
[META LOADED] Name: 'test_meta', Category: 'test'
[META LOADED] Description: 'Тестовое описание', Author: 'TestAuthor'
[OK] Author: 'TestAuthor'
[OK] Description: 'Тестовое описание'
[OK] Category: 'test'
[OK] Name: 'test_meta'
[TEST] Pass | Duration: 13ms
```

Рисунок 26 – Результат тестирования сохранения метаданных

Все четыре сценария отработали без ошибок. Удалось подтвердить, что метод получения данных корректно обрабатывает пути, содержащие индексы массивов. Итеративные преобразования выполняются без потерь. Эти результаты говорят о том, что система шаблонной генерации работоспособна, и её можно применять для задач генерации кода.

Итог: автоматизированные тесты целостности данных позволяют надёжно проверять механизмы сериализации JSON-структур. За счёт методики кругового тестирования и последующего семантического сравнения результатов ошибки, связанного с потерей данных, отлавливаются на ранних стадиях. Стопроцентное прохождение всех сценариев говорит о работоспособности всех компонентов генерации скриптов.

4.4 Метрики кода

4.4.1 Назначение и применение метрик

Метрики кода представляют собой совокупность количественных оценок программного обеспечения, служащих для анализа сложности, объёма и качества исходного кода. Применение метрик позволяет объективно оценить структурную сложность разработанных модулей, спрогнозировать трудозатраты на поддержку и выявить потенциально проблемные участки кода, требующие рефакторинга. В данной работе метрики кода рассмотрены на примере ключевого метода модуля генерации стандартных рецептов.

4.4.2 Цикломатическая сложность

Цикломатическая сложность является одной из метрик программного обеспечения, которая позволяет количественно оценить логическую сложность программы. Цикломатическая сложность может быть также применена для отдельных функций, модулей, методов и классов в пределах программы.

При вычислении цикломатической сложности используется граф потока управления программы. Узлы графа соответствуют неделимым группам команд программы, они соединены ориентированными рёбрами, если группа команд,

соответствующая второму узлу, может быть выполнена непосредственно после группы команд первого узла. Для вычисления цикломатической сложности $Z(G)$ применяется формула:

$$Z(G) = e - v + 2 \times p,$$

где e – число дуг ориентированного графа G ;

v – число вершин;

p – число компонентов связности графа.

На рисунке 27 представлен управляющий граф прецедента «Создать рецепт» [14].

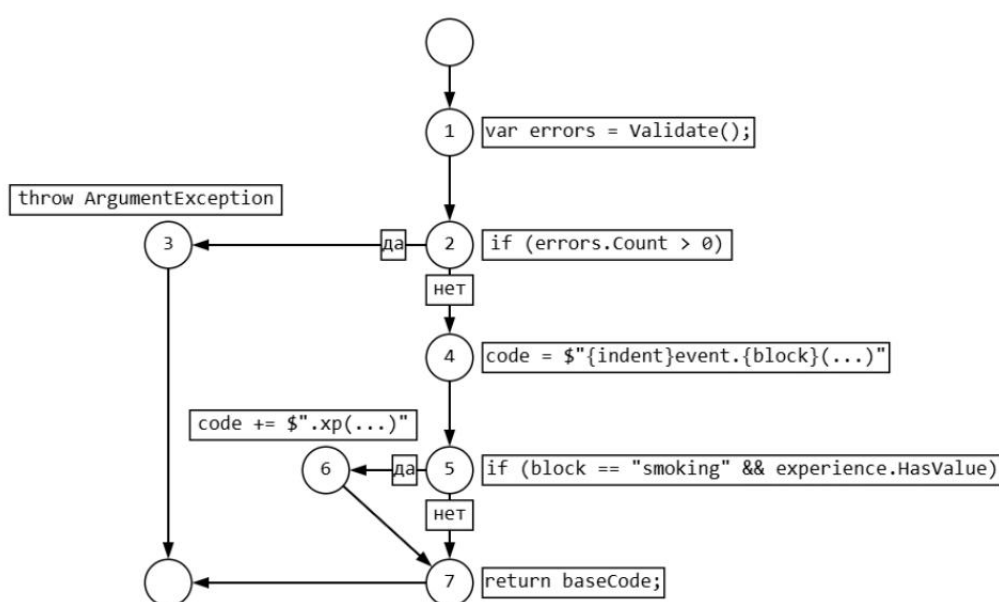


Рисунок 27 – Управляющий граф прецедента «Создать рецепт»

Из изображения графа видно, что e – число дуг ориентированного графа G равно 10, v – число вершин, равно 9, p – число компонентов связности графа равно 2, так внутри данного метода используется другой метод. Проводя вычисления по формуле, получаем, что цикломатическая сложность равна 5. На рисунке 28 представлена шкала для цикломатической сложности.



Рисунок 28 – Шкала цикломатической сложности

Данный результат позволяет говорить, что метод обладает низкой структурной сложностью. В случае, когда полученное значение циклической сложности больше десяти, это свидетельствует о структурной сложности программ.

4.4.3 Метрика SLOC

Количество строк кода (англ. Source Lines of Code – SLOC) – это метрика программного обеспечения, используемая для измерения его объёма с помощью подсчёта количества строк в тексте исходного кода. Как правило, этот показатель используется для прогноза трудозатрат на разработку конкретной программы на конкретном языке программирования, либо для оценки производительности труда уже после того, как программа написана.

Физическими строками считаются все непустые строки текстового файла. Измеряя логические строки кода, предпринимается попытка подсчитать количество операторов в программе, но, разумеется, их определение зависит от конкретного языка программирования. На рисунке 29 представлен текст функции генерации кода.

```
public override string GenerateJsCode()
{
    Validate();

    const string indent = "    ";
    string blockName = GetBlockName();

    // Формируем базовый вызов: event.smelting('output', 'input')
    var baseCode = $"{indent}event.{blockName}('{Output.ToJsString()}', '{Input.ToJ

    // Добавляем .xp() только для smoking
    if (Type == MinecraftRecipeType.Smoking && Experience.HasValue)
    {
        baseCode += $"{indent}.xp({Experience.Value.ToString(CultureInfo.InvariantCulture)})"
    }

    return baseCode;
}
```

Рисунок 29 – Прецедент «Создать рецепт»

В данной функции, представленной на рисунке 29, имеем 7 логических строк кода и 2 строки комментария [14].

Значение в 7 логических строк кода является малым, что свидетельствует о низких возможных трудозатратах при поддержке кода.

4.4.4 Метрика Холстеда

Основу данной метрики составляют 4 измеряемых характеристики программы:

- число уникальных операторов программы, включая символы-разделители, имена процедур и знаки операций (словарь операторов);
- число уникальных операндов программы (словарь операндов);
- общее число операторов в программе;
- общее число операндов в программе.

Для проведения метрического анализа по Холстеду в рамках прецедента «Создать рецепт» произведена идентификация всех лексических единиц исходного кода. В процессе выделения операторов учитывались не только явные знаки арифметических, логических и реляционных знаков, но также ключевые слова языка программирования C#, слова-разделители, скобки, точки с запятой, вызовы функций, а также операторы присваивания и сравнения. В свою очередь операндами признавались все идентификаторы переменных, константных выражений, литеральных значений, параметров методов, а также возвращаемые значения функций и промежуточные результаты выражений. При подсчете уникальных элементов словаря операторов учитывались однократно, тогда как общее число операторов и операндов вычислялось суммированием всех их вхождений в тексте программы. Полученный в результате такой лексической разметки перечень операторов и операндов с указанием частоты их появления систематизирован в таблице 10. Данная таблица служит первичной эмпирической основой для последующего вычисления метрики Холстеда. Важно отметить, что для корректности расчётов в операнды не включались имена типов данных и модификаторы доступа, если они не участвуют в операциях. Приведённая таблица фиксирует частотное распределение операторов и операндов в рамках прецедента «Создать рецепт» и выступает исходной точкой

для расчёта словарных длин и итоговых метрических показателей по формулам Холстеда.

Таблица 10 – Операнды и операторы

Операторы	Число вхождений	Операнды	Число вхождений
var	2	errors	3
=	3	block	2
Validate()	1	outputItem	2
;	6	inputItem	2
.	7	experience	3
>	1	Count	1
Throw	1	indent	2
New	1	baseCode	3
ArgumentExcep tion()	1	InvariantCultur e	1
Join()	1	HasValue	1
String	1	Value	1
ToJsString()	2		
{}	4		
==	1		
+=	1		
\$""	2		
&&	1		
()	2		
ToString()	1		
n1 = 19	N1 = 39	n2 = 11	N2 = 21

Длина программы рассчитывается по формуле:

$$N = N1 + N2,$$

где N1 – число вхождений операторов;

$N2$ – число вхождений операндов.

Словарь программы рассчитывается по формуле:

$$n = n1 + n2,$$

где $n1$ – число уникальных операторов;

$n2$ – число уникальных операндов.

Вычислительная сложность рассчитывается по формуле:

$$HV = N \times \log_2 n,$$

где N – длина программы;

n – словарь программы.

Расчёты метрики Холстеда приведены в таблице 11.

Таблица 11 – Расчёт метрики Холстеда

Метрика	Результат
Длина программы	$39 + 21 = 60$
Словарь программы	$19 + 11 = 30$
Вычислительная сложность	$60 \times \log_2 30 = 294,6$

В метрике Холстеда код со значением меньше 100 считается тривиальным, со значением меньше 300 считается легким, и так далее. На рисунке 30 показана шкала по метрике Холстеда.

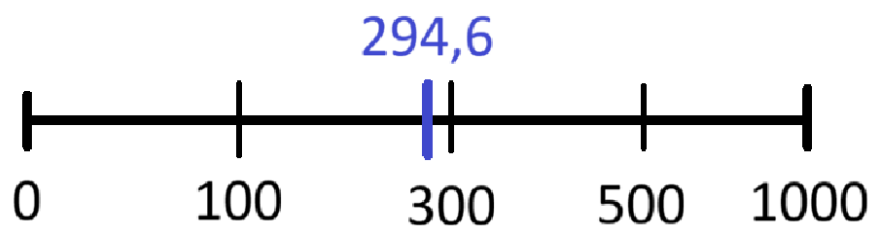


Рисунок 30 – Шкала вычислительной сложности прецедента «Создать рецепт»

Значение, равное 294,6 говорит о низкой вычислительной сложности написанного кода, так как по шкале оно меньше 300. Следовательно, разработчик может быть уверен, что данный прецедент не создаст избыточной нагрузки на анализ и отладку, а поддержка кода потребует минимальных усилий.

4.4.5 Комплексный показатель сопровождаемости

Комплексный показатель сопровождаемости кода (Maintainability Index) – рассчитывается по формуле:

$$MI = \max(0, 171 - 5.2 \times \ln HV - 0.23 \times CC - 16.2 \times \ln LoC) \times \frac{100}{171},$$

где CC (Cyclomatic Complexity) – структурная сложность кода равна 5;

LoC (Lines of Code) – количество строк кода равно 8;

HV – метрика Холстеда равна 294,6.

Комплексный показатель кода составляет 62. Данный показатель принимает значения от 0 до 100 и с его помощью определяют относительную сложность поддержки кода. На рисунке 31 изображена шкала комплексного показателя сопровождаемости кода.

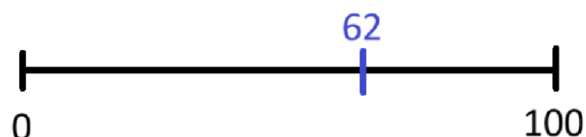


Рисунок 31 – Шкала сопровождаемости кода

Полученное значение для представленного варианта использования находится в допустимых границах и означает среднюю сложность поддержки кода. Таким образом, алгоритм кода имеет приемлемый индекс сопровождаемости.

4.5 Приёмочное тестирование в натурной среде

Приёмочное тестирование в натурной среде представляет собой критически важный этап верификации программного обеспечения, направленный на подтверждение корректности работы системы в реальных условиях эксплуатации [20].

Для разрабатываемого комплекса автоматизации генерации кода данный этап тестирования был сфокусирован на двух ключевых аспектах: проверке точности извлечения игровых метаданных и подтверждении работоспособности

сгенерированных скриптов в игровой среде выполнения. Согласно методологии тестирования программного обеспечения, натурные испытания позволяют выявить проблемы, которые невозможно обнаружить при изолированном модульном тестировании, особенно в контексте взаимодействия с внешними системами и средами выполнения [21].

4.5.1 Организация тестовой среды и подготовка испытаний

Тестирование проводилось в среде игры «Minecraft» версии «1.21.1 NeoForge» с установленными модификациями «KubeJS», «Icon Exporters» и «JEI» в корневую директорию игры в папку «mods». Подготовка тестовой среды включала стандартную установку игрового клиента, настройку пути к корневой директории игры и создание конфигурационного файла, указывающего на расположение игровых ресурсов. Важно отметить, что проверка корректности установки модов и настройки игровой среды не входила в зону ответственности разрабатываемого программного комплекса, так как его функциональность ограничена обработкой уже существующих данных и генерацией конфигурационных файлов.

4.5.2 Методика проверки точности извлечения метаданных

Процесс верификации подсистемы извлечения метаданных осуществлялся путем сравнения данных, полученных с помощью разработанной утилиты, с фактическими данными, присутствующими в игровой среде. Методика включала следующие этапы:

Первоначально выполнялось извлечение данных с использованием реализованного модуля ItemFluidWithIconExtractor, который анализирует файлы, сгенерированные модификацией «icon-exporter.jar», и строит внутреннее представление предметов и жидкостей. Полученные данные проверялись на соответствие реальным элементам игры путем ручной выборки и сравнения идентификаторов, локализованных названий и графических ресурсов. Согласно требованиям метрологической оценки программного обеспечения, проверка

точности данных является обязательным этапом приёмочного тестирования для систем, основанных на обработке внешних источников информации [22]. На рисунке 32 показан результат работы программы при вызове функции извлечения данных. Он показывает лишь первые 15 совпадений из списка всех предметов. Главным показателем работоспособности функции извлечения является игровой идентификатор, который является уникальным для всей игровой среды. Сравнивая полученный идентификатор с оригинальным можно сделать вывод, что функция работает успешно.

```
Предупреждение: Нет метаданных для ID minecraft:bundle, файл
d Circuit New 1.21\icon-exports-x16\minecraft__bundle.png
Найдено 3437 предметов и 56 жидкостей.
Загружено 3493 элементов.

=== Первые 15 элементов из базы данных ===
ID: aeronautics:adjustable_burner
  Название: Hot Air Burner
  Мод: Create Aeronautics
  Тип: item
  Теги:
  Размер изображения: 638 байт

ID: aeronautics:andesite_propeller
  Название: Andesite Propeller
  Мод: Create Aeronautics
  Тип: item
  Теги:
  Размер изображения: 558 байт
```

Рисунок 32 – Успешное извлечение метаданных и изображений

Для проверки необходимо зайти в игровую сессию, и поиске ввести идентичное название. После ввода названия в поисковую строку система отображает соответствующий предмет с его иконкой и названием. Затем, полученные из игры данные сопоставляются с теми, что были ранее извлечены утилитой. Если визуальное представление и текстовые атрибуты полностью совпадают, то тест считается пройденным. Такая процедура повторяется для пятнадцати выборочных предметах, покрывающий основной набор внутриигровых ресурсов. На рисунке 33 изображен результат сравнения с консольной информацией.



Рисунок 33 – Сравнение с игровым предметом

Особое внимание уделялось проверке корректности обработки предметов из различных модификаций, включая базовую игру «Minecraft», модификацию «Create» и дополнительные модификации к нему. Для каждого извлечённого элемента проверялись его уникальный идентификатор, локализованное название, принадлежность к модификации и соответствие графического представления. В процессе тестирования не было выявлено несоответствий между извлечёнными данными и реальными элементами игры, что подтверждает корректность реализованного алгоритма извлечения.

4.5.3 Методика проверки работоспособности генерируемых скриптов

Проверка сгенерированных скриптов осуществлялась по следующей методике: после генерации JavaScript-кода с помощью разработанного программного комплекса файлы размещались в соответствующей директории. Затем запускался игровой клиент. Проверялось наличие новых рецептов в игровом процессе. При проверке важно было убедиться в том, что скрипты загружаются игрой корректно, и в корректной интерпретации их KubeJS – одно без другого не имеет смысла.

Тестирование охватило два рабочих вида рецептов: стандартные и из модификации «Create». По каждому проверялись синтаксис, идентификаторы

предметов и соответствие игровой логике. Отдельно тестировался сложный вариант с вероятностными выходами, жидкостными компонентами и цепочками модификаторов.

Две рабочие функции, полученные из тестового модуля, приведены на рисунке 34.

```
[TEST] Generate: stonecutting
PASS: event.stonecutting('minecraft:stone_slab', 'minecraft:stone')
[GROUP] Генерация: Experience для smoking
[TEST] Generate: Smoking exp=0
PASS: Содержит .xp(0)
[TEST] Generate: Smoking exp=0,5
PASS: Содержит .xp(0.5)
[TEST] Generate: Smoking exp=1
PASS: Содержит .xp(1)
[TEST] Generate: Smoking exp=100
PASS: Содержит .xp(100)
[GROUP] Форматирование: Культура и отступы
[TEST] Generate: InvariantCulture exp=0,75
PASS: Используется точка (InvariantCulture)

=====
StandartRecipeTests: 35 тестов завершено
Пройдено: 35
Провалено: 0
```

Рисунок 34 – Валидная функция резки камня

Функция для пресса, также успешно прошедшая тест, показана на рисунке 35.

```
[TEST] Start: CreateCompactingConfig - Генерация
[GENERATED] event.recipes.create.compacting(['minecraft:diamond'], ['minecraft:coal_block'])
[TEST] Pass
```

Рисунок 35 – Сгенерированная валидная функция для пресса

Итоговый код, который непосредственно вносит изменения в игру, изображен на рисунке 36.

```
ServerEvents.recipes(event =>
{
    event.stonecutting('minecraft:stone_slab', 'minecraft:stone')

    event.recipes.create.compacting(['minecraft:diamond'], ['minecraft:coal_block'])
})
```

Рисунок 36 – KubeJS-скрипт для модификации игры

Чтобы применить изменения, нужно выполнить в серверной консоли две ключевые команды. Для этого требуются права администратора (или

включенный «режим читов» в одиночной игре). На рисунке 37 изображена команда для обновления файлов.



Рисунок 37 – Команда для обновления файлов

Данная команда позволяет игре увидеть новые, только что созданные файлы для модификации игровой среды. Если файл не создавался, а изменялся, то его достаточно сохранить. Затем необходимо перезапустить игровой сервер с помощью команды перезагрузки. Для одиночной игровой среды это также актуально, поскольку игра использует интегрированный сервер [1]. На рисунке 38 изображена команда для перезапуска сервера.



Рисунок 38 – Команда для перезапуска сервера

После перезапуска должно появиться сообщение во внутриигровом чате или серверной консоли об успешной модификации игры. Если сообщение отсутствует или появляются сообщения об ошибке, то тест считается неуспешным. Для визуального контроля использовалась модификация «JEI», отображающая все доступные рецепты в интерактивном интерфейсе. Оба рецепта были протестированы в игровой среде – игровой персонаж смог их повторить на рабочих станциях. На рисунке 39 показаны результаты работы кода.



Рисунок 39 – Добавлен рецепт для резки камня «1 к 1»

На рисунке 40 показан результат применения второго рецепта. Модификация «JEI» показывает данный рецепт в интерактивном пользовательском интерфейсе. Это свидетельствует об успешной модификации игры.

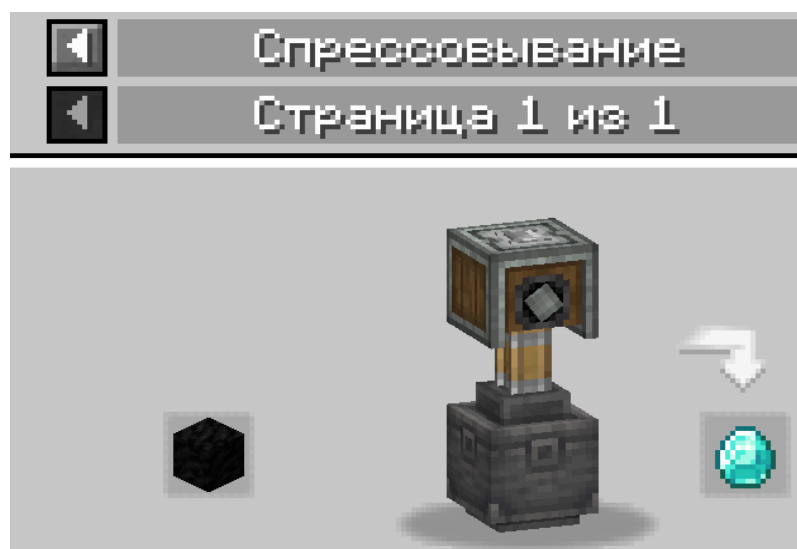


Рисунок 40 – Добавлен рецепт для спрессовывания

Результаты тестирования подтверждают корректность функционирования сгенерированных программных модулей в целевой среде исполнения. В ходе

испытаний были протестированы два основных типа технологических цепочек: стандартные механизмы обработки ресурсов и специализированные процессы модификации «Create». Для каждого типа рецепта была выстроена сквозная проверка: от автоматической генерации и размещения файлов в проекте до перезагрузки сервера и визуального контроля результата в игре.

Стандартные механики, например «резка камня», корректно добавили новые правила преобразования ресурсов – это видно на рисунке 39. Специализированные процессы из модификации «Create», в частности «пресс», также отработали без нареканий, обеспечив ожидаемое преобразование материалов (рисунок 40). Все полученные результаты выдержали проверку синтаксиса JavaScript и требований KubeJS. При загрузке не возникло ни одной ошибки. Добавленные элементы стабильно работают в игровой среде.

Поскольку изменения применялись штатными серверными средствами, получилось сразу же проверить работоспособность каждой созданной конфигурации. По итогам тестирования не было зафиксировано ни одного случая синтаксической ошибки или отклонения от заданной логики работы. Это свидетельствует о высокой надежности реализованного алгоритма генерации кода для поддерживаемых типов рецептов. Полученные данные полностью соответствуют поставленной цели по созданию инструмента, обеспечивающего автоматизацию разработки игровых механик без необходимости ручного программирования.

4.5.4 Результаты тестирования извлечения метаданных

В ходе тестирования подсистемы извлечения метаданных были получены следующие результаты. Система успешно обработала 3493 элемента, включая 3437 предметов и 56 жидкостей. Все извлечённые данные соответствовали реальным элементам, присутствующим в игровой среде.

В процессе тестирования не было выявлено случаев несоответствия между извлечёнными идентификаторами и реальными данными в игре. Все предметы и жидкости, присутствующие в игре, были корректно отражены в внутреннем

представлении системы. Это подтверждает высокую точность реализованного алгоритма извлечения данных и его соответствие требованиям к достоверности информации.

4.5.5 Результаты тестирования созданного кода

Все сгенерированные скрипты загрузились без синтаксических ошибок. Рецепты в игровой среде работали как ожидалось: новые правила применялись и сохранялись корректно. По каждому типу рецептов проведена отдельная проверка.

Для рецептов дополнения «Create» тестировались сложные структуры: цепочки вызовов с модификаторами и обработка жидкостных компонентов. Результаты приёмочного тестирования в натурной среде подтверждают соответствие разработанного программного комплекса установленным требованиям и его готовность к практическому применению. Система точно извлекает игровые метаданные и генерирует валидные скрипты, устраняя необходимость ручного написания кода [20]. Это соответствует цели разработки: надёжный инструмент автоматизации генерации скриптов KubeJS с верифицированными данными [21].

Заключение

В ходе выпускной квалификационной работы разработано программное обеспечение для автоматизированной генерации кода, применяемого при настройке игрового процесса «Minecraft». Работа прошла все этапы: от анализа предметной области до приёмочного тестирования в натурной среде. Отдельно оценивались временные и финансовые затраты.

На стадии анализа требований изучена предметная область настройки игровых механик, проведён сравнительный анализ существующих решений аналогичного назначения, сформулированы функциональные и нефункциональные требования.

Спроектированы конфигурации рецептов для разных типов: простые, рецепты модификации «Create», шаблонные и для удаления. Они инкапсулируют внутри себя логику преобразования себя из данных в готовый и рабочий JavaScript-код с соблюдением форматирования и отступов. Разработанная модульная архитектура позволяет легко добавлять новые конфигурации рецептов не меняя исходный код.

Разработанная система экстракторов с использованием библиотеки «Esprima» позволяет пользователю программного комплекса легко и быстро редактировать уже существующие скрипты. Итеративная разработка значительно сокращает время работы при модифицировании игрового сервера или клиента.

Автоматизированная система тестирования позволяет в один клик проверять корректность работы системы генерации кода. В системе также разработано тестирование круговых преобразований шаблонных рецептов. Она проверяет на равенство семантик двух конфигураций, первой и изначальной, и второй, которая прошла процесс сохранения и загрузки JSON. Это позволяет обеспечить корректность при итеративной разработке.

Разработанный программный комплекс корректно работает и обеспечивает высокую скорость разработки при низких требованиях к знанию языков программирования.

Список использованных источников

1. KubeJS. Официальный сайт с документацией [Электронный ресурс]. URL: <https://kubejs.com/> (дата обращения: 21.03.2026).
2. Microsoft Learn. Пакет SDK для платформы компилятора .NET (API-интерфейсы) [Электронный ресурс]. URL: <https://learn.microsoft.com/ru-ru/dotnet/csharp/roslyn-sdk/> (дата обращения: 22.03.2026).
3. Мартин Р., Мартин М. Принципы, паттерны и методики гибкой разработки на языке C#. – СПб.: Символ-Плюс, 2011. – 757 с.
4. GitHub. Esprima .NET. Репозиторий стандартного ECMAScript-анализатора для .NET [Электронный ресурс]. URL: <https://github.com/sebastienros/esprima-dotnet> (дата обращения: 28.03.2026).
5. Microsoft Learn. File.Replace Method (System.IO). Документация .NET. [Электронный ресурс]. URL: <https://learn.microsoft.com/ru-ru/dotnet/api/system.io.file.replace>. (дата обращения: 29.03.2026).
6. Blamejared. Документация CraftTweaker [Электронный ресурс]. URL: <https://docs.blamejared.com/> (дата обращения: 29.03.2026).
7. Сырцев В. Использование диаграммы вариантов использования UML при проектировании программного обеспечения // Хабр [Электронный ресурс]. URL: <https://habr.com/ru/articles/566218/> (дата обращения: 01.04.2026).
8. Богданов В. В. Управление проектами в Microsoft Project 2007. Учебный курс – СПб.: Питер Пресс, 2008. – 594 с.
9. Ларман К. Применение UML и шаблонов проектирования. – Издательский дом «Вильямс», 2006. – 620 с.
10. Гагарина Л.Г., Федоров А.Р., Федоров П.А. Введение в архитектуру программного обеспечения: Учебное пособие. – М.: ФОРУМ, 2020. – 320 с.
11. Игнатъев В. Н. Статический анализ на абстрактных синтаксических деревьях с помощью конечных автоматов. – Труды ИСП РАН. 2025. – 54 с.
12. Андреев С. В., Болотников М. А., Ханафиев Н. А. Оценка применения паттерна Chain of Responsibility для создания гибкой системы

обработки запросов – Новые горизонты науки: стратегии и направления развития. Белгород: АПНИ, 2024. – 28 с.

13. Douceur J. R. A five-year study of file-system metadata – Proceedings of the 5th USENIX Conference on File and Storage Technologies, 2007. – 32 с.

14. GitHub. Исходный код разрабатываемого комплекса приложений [Электронный ресурс]. URL: <https://github.com/Coatllicence/KubejsAutomation>.

15. Санкт-Петербургский государственный университет. Диаграммы деятельности (Activity Diagrams). // Национальный Открытый Университет «ИНТУИТ» [Электронный ресурс]. URL: <https://intuit.ru/studies/courses/1041/218/lecture/27261?page=2> (дата обращения: 09.04.2026).

16. Коптева А. В., Князев И. В. Анализ проблемы преобразования данных формата JSON в строго типизированных языках программирования на примере Golang – Научные труды 2021. – 1 - 5 с.

17. Ensi Tech. Диаграмма последовательности (Sequence Diagrams) // Документация по применению [Электронный ресурс]. URL: <https://docs.ensi.tech/analyst-guides/tools/diagrams/uml/sequence-diagram> (дата обращения: 14.04.2026).

18. Шлаев, Д. В. Инновационные подходы к визуализации и разработке с применением унифицированного языка моделирования (UML) // учебное пособие. 2024 [Электронный ресурс]. URL: <https://kf.lib.bmstu.ru/Catalog/Details/IPRBooks/156601> (дата обращения: 18.04.2026).

19. ГОСТ Р ИСО/МЭК 25010–2015. Информационные технологии. Системная и программная инженерия. Требования к качеству систем и программных продуктов. – Введ. 2016–03–01. – М. : Стандартинформ, 2016.

20. Турнецкая Е. Л., Аграновский А. В. Программная инженерия. Тестирование и контроль качества программного обеспечения: учебное пособие для вузов. – СПб.: Лань, 2025. – 172 с.

21. Кулямин В. В. Методы верификации программного обеспечения. / В. В. Кулямин – Институт системного программирования РАН, 2006. – 117 с.
22. Проскуряков А. В. Качество и тестирование программного обеспечения. Метрология программного обеспечения. Учебное пособие. / В. А. Проскуряков. – Издательство «Южный федеральный университет», 2022. – 197 с.

Приложение А
(справочное)
Глоссарий

API (Application Programming Interface) – набор готовых классов, функций, процедур, констант и структур, предоставляемых приложением или операционной системой для использования во внешних программных продуктах.

AST (Абстрактное синтаксическое дерево, AST) – дерево. Представление исходного кода, написанного на некотором языке программирования, в котором каждый узел обозначает конструкцию, встречающуюся в коде. Используется в разработанном ПО для статического анализа и извлечения параметров скриптов KubeJS.

Верификация – процесс проверки данных или программного кода на соответствие заданным правилам, ограничениям и синтаксическим стандартам. В работе применяется для контроля корректности параметров рецептов до этапа генерации итогового скрипта.

JSON (JavaScript Object Notation) – текстовый формат обмена данными, основанный на JavaScript. Применяется в работе для структурирования метаданных игровых элементов, хранения конфигураций и описания произвольных структур рецептов.

Рецепт (в контексте игровой среды и библиотеки «KubeJS») – это формализованное описание правила преобразования внутриигровых ресурсов в конечный продукт. Такое правило задается в виде структурированного файла (на языке JavaScript или в формате JSON). В разработанном приложении рецепты выступают целевым артефактом генерации.

KubeJS – скриптовый фреймворк для модификации игры «Minecraft», предоставляющий программный интерфейс на языке JavaScript для изменения внутриигровых рецептов, регистрации новых предметов и блоков, а также обработки игровых событий без необходимости компиляции исходного Java-кода.

ООП (Объектно-ориентированное программирование) – методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса.

Приложение Б
(обязательное)
Листинг приложения

```

using KubeAutomation;
using KubeAutomation.FileSaveStrategies;
using KubeAutomation.GenerationStrategies.Generators;
using KubeAutomation.GenerationStrategies.Parsing;
using KubeAutomation.GenerationStrategies.RecipeConfigurations;
using
KubeAutomation.GenerationStrategies.RecipeConfigurations.Create;
using KubeAutomation.Tests;
using LogExtractorLibrary;
using System.IO.Pipes;
using System.Text.Json;

class Program
{
    static readonly MyForm form = MyForm.GetInstance();

    [STAThread]
    static async Task Main()
    {
        // Запускаем форму в отдельном потоке
        Thread formThread = new(() =>
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(form);
        });
        formThread.SetApartmentState(ApartmentState.STA);
        formThread.IsBackground = true;
        formThread.Start();

        //GenCreateRecipes();

        //TestCreateChains();

        GamePathProvider.SetGameRootPath("C:\\Users\\f0578\\AppData\\Roami
ng\\.tlauncher\\legacy\\Minecraft\\Create Cog and Circuit New
1.21");

        await RunItemDemo();

        //TestJS();

        Console.WriteLine("Система автоматизации KubeJS
скриптов");

        Console.WriteLine("=====
");

        var inputCollector = new
GregTechRecipeCodeGenerator.RecipeInputCollector();
        var fileSaver = new RecipeFileSaver();

```

```

while (true)
{
    Console.WriteLine("\nХотите создать новый рецепт?
(да/нет/получить)");

    string res = Console.ReadLine()?.Trim().ToLower();

    if (res == "получить" || res == "п" || res == "g" ||
res == "get")
    {
        // перенесено в форму
        //_ = StartExternalAppAsync(); // Запускает задачу
в фоне, не блокируя поток
    }

    if (res == "t" || res == "т" || res == "test" || res
== "тест")
    {
        TestRunner.RunAll();
        Console.WriteLine("\n\n\n Введите любой символ
для продолжения...");
        Console.ReadKey();
        continue;
    }

    if (res == "пр" || res == "gr")
    {
        try
        {
            var extractor = new RecipesExtractor();
            Console.WriteLine($"Экстрактор готов к работе:
{extractor.IsValid()}");

            var recipes = await extractor.ExtractAsync();
            Console.WriteLine($"Получено {recipes.Count}
рецептов.");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Ошибка при извлечении
рецептов: {ex.Message}");
        }
    }

    if (res == "н" || res == "нет" || res == "n" || res ==
"no")
        break;

    try
    {
        var config = inputCollector.Collect();
    }
}

```

```

        string script =
GregTechRecipeCodeGenerator.Generate(config);

        string? dir = null;
        while (dir == null)
        {
            Console.WriteLine("Введите путь сохранения");
            dir = Console.ReadLine()?.Trim();

            if (dir == "н" || dir == "нет" || dir == "н"
|| dir == "no")
            {
                dir =
"C:\\Users\\f0578\\OneDrive\\Desktop\\Предметы";
                break;
            }

            //fileSaver.Save(script, config.RecipeId, dir);

            Console.WriteLine("\nРецепт успешно создан!");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Ошибка при создании рецепта:
{ex.Message}");
        }
    }
    ServerRecipeType s2 = new();

    Console.WriteLine(EmptyFunctionGenerator.GenerateTitle(s2));

    Console.WriteLine("Программа завершена.");
}

public static void TestJS()
{
    Console.WriteLine("=== Тест парсера JS ===\n");

    string testFilePath =
Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "..", "..",
"..", "test_recipes.js");

    if (!File.Exists(testFilePath))
    {
        testFilePath = "test_recipes.js";
    }

    Console.WriteLine($"Путь к файлу:
{Path.GetFullPath(testFilePath)}");
    Console.WriteLine($"Файл существует:
{File.Exists(testFilePath)}\n");
}

```

```

        if (!File.Exists(testFilePath))
        {
            Console.WriteLine("[ERROR] Файл не найден! Создайте
test_recipes.js в корне проекта.");
            Console.ReadKey();
            return;
        }

        try
        {
            var parser = new JsRecipeParser();

            Console.WriteLine("Чтение файла...");
            var configs = parser.ParseFile(testFilePath);
            Console.WriteLine($"Элементы в порядке файла (всего:
{configs.Count})");

            int removalCount = 0, customCount = 0, rawCount = 0,
otherCount = 0;

            for (int i = 0; i < configs.Count; i++)
            {
                var config = configs[i];
                Console.WriteLine($"\\n[{i + 1}]");

                if (config is RecipeRemovalConfig removal)
                {
                    removalCount++;
                    Console.WriteLine($"      [Удаление] фильтров:
{removal.Filters.Count}");
                    foreach (var f in removal.Filters)
                    {
                        Console.WriteLine($"mod:{f.Mod ?? "-"}
out:{f.Output ?? "-"} type:{f.Type ?? "-"}");
                    }
                }
                else if (config is CustomRecipeConfig custom)
                {
                    customCount++;
                    var type = custom.GetValue("type") as string
?? "-";
                    var id = custom.RecipeId ?? "-";
                    Console.WriteLine($"      [custom] type:{type}
id:{id}");
                }
                else if (config is CreateFillingConfig filling)
                {
                    Console.WriteLine($"      [Create:Filling]
out:{filling.Outputs[0].ItemId} in:{filling.Inputs.Count}");
                }
                else if (config is CreateCrushingConfig crushing)
                {

```

```

        Console.WriteLine($"      [Create:Crushing]
out:{crushing.Outputs.Count} in:{crushing.Inputs}");
    }
    else if (config is CreateDeployingConfig
deploying)
    {
        Console.WriteLine($"      [Create:Deploying]
out:{deploying.Outputs[0].ItemId} in:{deploying.Inputs.Count}");
    }
    else if (config is RawCodeBlock raw)
    {
        rawCount++;
        var code = raw.RawNode?.ToString() ??
"[empty]";
        Console.WriteLine($"      [Сырой код]
{raw.Reason}: {code.Substring(0, Math.Min(40, code.Length))}...");
    }
}

    Console.WriteLine("\n-----
-----");
    Console.WriteLine($"[СТАТИСТИКА]:");
    Console.WriteLine($"      Удаление: {removalCount}");
    Console.WriteLine($"      Custom: {customCount}");
    Console.WriteLine($"      Сырой код: {rawCount}");
    Console.WriteLine($"      Другие: {otherCount}");
    Console.WriteLine($"      Всего: {configs.Count}");
    Console.WriteLine("\n-----
-----");

    // Генерация обратно в JS
    Console.WriteLine($"\\n=== Генерация обратно в JS
===");

    foreach (var config in configs)
    {
        Console.WriteLine(config.GenerateJsCode());
    }
}
catch (Exception ex)
{
    Console.WriteLine($"\\n[ERROR] {ex.Message}");
    if (ex.InnerException != null)
    {
        Console.WriteLine($"      Внутренняя ошибка:
{ex.InnerException.Message}");
    }
}

    Console.WriteLine("\\n=== Тест завершён ===");
    Console.WriteLine("Нажмите любую клавишу для выхода...");
    Console.ReadKey();

```

```

    }

    private static void TestCreateChains()
    {
        Console.WriteLine("=== Тест: Цепочки модификаторов Create  

        ===\n");

        // 1. Создаём конфиг с модификаторами
        var config = new CreateMixingConfig();
        config.Inputs.Add(new CreateItemIngredientComponent {
            ItemId = "minecraft:coal" });
        config.Inputs.Add(new CreateFluidIngredientComponent {
            FluidId = "minecraft:water", Amount = 1000 });
        config.Outputs.Add(new CreateItemComponent { ItemId =
            "minecraft:diamond", Count = 1, Chance = 0.5f });
        config.Outputs.Add(new CreateItemComponent { ItemId =
            "minecraft:stick", Count = 1, Chance = 0.5f });

        config.Modifiers.Heat = CreateHeatType.Superheated;

        Console.WriteLine("1. Исходный конфиг:");
        Console.WriteLine($"    Inputs: {config.Inputs.Count},
        Outputs: {config.Outputs.Count}");
        Console.WriteLine($"    Heat: {config.Modifiers.Heat},
        Chance: {config.Outputs[0].Chance}");
        Console.WriteLine();

        // 2. Генерация в JS
        string generatedCode = config.GenerateJsCode();
        Console.WriteLine("2. Сгенерированный код:");
        Console.WriteLine(generatedCode);
        Console.WriteLine();

        // 3. Сохраняем во временный файл
        string tempFile = Path.Combine(Path.GetTempPath(),
            "test_create_chain.js");
        string fullJs = $"ServerEvents.recipes(event => {{\n
        {generatedCode}\n}});";
        File.WriteAllText(tempFile, fullJs);
        Console.WriteLine($"3. Временный файл: {tempFile}");
        Console.WriteLine();

        // 4. Парсим обратно
        var parser = new JsRecipeParser();
        var configs = parser.ParseFile(tempFile);
        Console.WriteLine($"4. Распознано элементов:
        {configs.Count}");

        if (configs.Count > 0 && configs[0] is CreateMixingConfig
            parsedConfig)
        {
            Console.WriteLine("5. Распознанный конфиг:");

```

```

        Console.WriteLine($"    Inputs:
{parsedConfig.Inputs.Count}, Outputs:
{parsedConfig.Outputs.Count}");
        Console.WriteLine($"    Heat:
{parsedConfig.Modifiers.Heat}, Chance:
{parsedConfig.Outputs[0].Chance}");

        // Проверка соответствия
        bool matches = config.Inputs.Count ==
parsedConfig.Inputs.Count &&
                        config.Outputs.Count ==
parsedConfig.Outputs.Count &&
                        config.Modifiers.Heat ==
parsedConfig.Modifiers.Heat &&
                        config.Outputs[0].Chance ==
parsedConfig.Outputs[0].Chance;

        Console.WriteLine($"    Соответствие: {(matches ? "OK"
: "FAIL")}");
    }
    else
    {
        Console.WriteLine("5. Не удалось распознать
CreateMixingConfig");
        if (configs.Count > 0)
        {
            Console.WriteLine($"    Фактический тип:
{configs[0].GetType().Name}");
        }
    }

    // 6. Очистка
    try { File.Delete(tempFile); } catch { }
    Console.WriteLine("\n=== Тест завершён ===");
}
// Асинхронная функция для запуска внешнего приложения и
получения данных из именного канала
public static async Task StartExternalAppAsync()
{
    //const string pipeName = "MyMinecraftItemsPipe"; // Не
используется напрямую теперь
    //string logPath =
"C:\\Users\\nojda\\AppData\\Roaming\\.tlauncher\\legacy\\Minecraft
\\game\\logs\\debug.log"; // Не нужно

    await Task.Delay(1000);

    var processStartInfo = new
System.Diagnostics.ProcessStartInfo
    {
        FileName =
@"C:\\Users\\nojda\\source\\repos\\LogExtractor\\LogExtractor\\bin\\Debug\\
net8.0\\LogExtractor.exe",

```

```

        Arguments = "all",
        UseShellExecute = false,
        CreateNoWindow = true,
        RedirectStandardOutput = false,
        RedirectStandardError = false
    };

    var process =
System.Diagnostics.Process.Start(processStartInfo);
    if (process == null)
    {
        Console.WriteLine("Ошибка: не удалось запустить
внешнее приложение.");
        return;
    }

    Console.WriteLine("Внешнее приложение запущено. Ожидание
подключения к каналам...");

    // === Подключаемся к каналу предметов/жидкостей ===
    await ReceiveItemsAndFluidsAsync(process);

    // === Подключаемся к каналу рецептов ===
    await ReceiveRecipesAsync(process);

    Console.WriteLine("Все данные получены из каналов.");
}

private static async Task
ReceiveItemsAndFluidsAsync(System.Diagnostics.Process process)
{
    const string pipeName = "MyMinecraftItemsPipe";

    try
    {
        using var client = new NamedPipeClientStream(".",
pipeName, PipeDirection.In);
        await client.ConnectAsync(30000); // Асинхронное
подключение
        Console.WriteLine("Подключено к именованному каналу
предметов/жидкостей.");

        using var reader = new StreamReader(client,
System.Text.Encoding.UTF8, detectEncodingFromByteOrderMarks:
false, bufferSize: 1024, leaveOpen: false);
        string lengthStr = await reader.ReadLineAsync();
        if (!int.TryParse(lengthStr, out int itemsDataLength))
        {
            Console.WriteLine("Ошибка: не удалось прочитать
длину данных предметов из канала.");
            return;
        }
    }
}

```

```

        var buffer = new byte[itemsDataLength];
        int totalRead = 0;
        while (totalRead < itemsDataLength)
        {
            int read = await client.ReadAsync(buffer,
totalRead, itemsDataLength - totalRead);
            if (read == 0) break;
            totalRead += read;
        }

        if (totalRead != itemsDataLength)
        {
            Console.WriteLine($"Ошибка: прочитано {totalRead}
байт, ожидалось {itemsDataLength}.");
            return;
        }

        string itemsJson =
System.Text.Encoding.UTF8.GetString(buffer);
        var itemIds =
JsonSerializer.Deserialize<List<string>>(itemsJson, new
JsonSerializerOptions());

        // Читаем ID жидкостей
        string lengthStr2 = await reader.ReadLineAsync();
        if (!int.TryParse(lengthStr2, out int
fluidsDataLength))
        {
            Console.WriteLine("Ошибка: не удалось прочитать
длину данных жидкостей из канала.");
            return;
        }

        var buffer2 = new byte[fluidsDataLength];
        totalRead = 0;
        while (totalRead < fluidsDataLength)
        {
            int read = await client.ReadAsync(buffer2,
totalRead, fluidsDataLength - totalRead);
            if (read == 0) break;
            totalRead += read;
        }

        if (totalRead != fluidsDataLength)
        {
            Console.WriteLine($"Ошибка: прочитано {totalRead}
байт, ожидалось {fluidsDataLength}.");
            return;
        }

        string fluidsJson =
System.Text.Encoding.UTF8.GetString(buffer2);

```

```

        var fluidIds =
JsonSerializer.Deserialize<List<string>>(fluidsJson, new
JsonSerializerOptions());

        // Читаем количество изображений
        string countItemsStr = await reader.ReadLineAsync();
        string countFluidsStr = await reader.ReadLineAsync();

        if (!int.TryParse(countItemsStr, out int totalItems)
|| !int.TryParse(countFluidsStr, out int totalFluids))
        {
            Console.WriteLine("Ошибка: не удалось прочитать
количество изображений.");
            return;
        }

        Console.WriteLine($"Ожидается {totalItems} изображений
предметов и {totalFluids} изображений жидкостей.");

        // Читаем изображения предметов
        var itemImages = new Dictionary<string, byte[]>();
        for (int i = 0; i < totalItems; i++)
        {
            string imgSizeStr = await reader.ReadLineAsync();
            if (!int.TryParse(imgSizeStr, out int imgSize))
            {
                Console.WriteLine($"Ошибка: не удалось
прочитать размер изображения предмета {i}.");
                return;
            }

            var imgBuffer = new byte[imgSize];
            int totalImgRead = 0;
            while (totalImgRead < imgSize)
            {
                int read = await client.ReadAsync(imgBuffer,
totalImgRead, imgSize - totalImgRead);
                if (read == 0) break;
                totalImgRead += read;
            }

            if (totalImgRead != imgSize)
            {
                Console.WriteLine($"Ошибка: прочитано
{totalImgRead} байт изображения, ожидалось {imgSize}.");
                return;
            }

            string id = itemIds[i];
            itemImages[id] = imgBuffer;
        }

        // Читаем изображения жидкостей

```

```

var fluidImages = new Dictionary<string, byte[]>();
for (int i = 0; i < totalFluids; i++)
{
    string imgSizeStr = await reader.ReadLineAsync();
    if (!int.TryParse(imgSizeStr, out int imgSize))
    {
        Console.WriteLine($"Ошибка: не удалось
прочитать размер изображения жидкости {i}.");
        return;
    }

    var imgBuffer = new byte[imgSize];
    int totalImgRead = 0;
    while (totalImgRead < imgSize)
    {
        int read = await client.ReadAsync(imgBuffer,
totalImgRead, imgSize - totalImgRead);
        if (read == 0) break;
        totalImgRead += read;
    }

    if (totalImgRead != imgSize)
    {
        Console.WriteLine($"Ошибка: прочитано
{totalImgRead} байт изображения, ожидалось {imgSize}.");
        return;
    }

    string id = fluidIds[i];
    fluidImages[id] = imgBuffer;
}

Console.WriteLine($"Получено {itemIds.Count} ID
предметов и {fluidIds.Count} ID жидкостей с изображениями.");

MyForm.GetInstance().UpdateData(itemImages,
fluidImages);
}
catch (TimeoutException)
{
    Console.WriteLine("Ошибка: не удалось подключиться к
каналу предметов/жидкостей в течение 30 секунд.");
    if (!process.HasExited)
    {
        try { process.Kill(); }
        catch { /* Игнорируем ошибки при завершении */ }
    }
}
catch (IOException ex)
{
    Console.WriteLine($"Ошибка именованного канала
(предметы/жидкости): {ex.Message}");
}

```

```

        if (!process.HasExited)
        {
            try { process.Kill(); }
            catch { /* Игнорируем ошибки при завершении */ }
        }
    }
    catch (JsonException ex)
    {
        Console.WriteLine($"Ошибка обработки JSON
(предметы/жидкости): {ex.Message}");
        if (!process.HasExited)
        {
            try { process.Kill(); }
            catch { /* Игнорируем ошибки при завершении */ }
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Произошла ошибка
(предметы/жидкости): {ex.Message}");
        if (!process.HasExited)
        {
            try { process.Kill(); }
            catch { /* Игнорируем ошибки при завершении */ }
        }
    }
}

private static async Task
ReceiveRecipesAsync(System.Diagnostics.Process process)
{
    const string pipeName = "MyMinecraftRecipesPipe";

    try
    {
        using var client = new NamedPipeClientStream(".",
pipeName, PipeDirection.In);
        await client.ConnectAsync(30000); // Асинхронное
подключение
        Console.WriteLine("Подключено к именованному каналу
рецептов.");

        using var reader = new StreamReader(client,
System.Text.Encoding.UTF8, detectEncodingFromByteOrderMarks:
false, bufferSize: 1024, leaveOpen: false);
        string lengthStr = await reader.ReadLineAsync();
        if (!int.TryParse(lengthStr, out int dataLength))
        {
            Console.WriteLine("Ошибка: не удалось прочитать
длину данных рецептов из канала.");
            return;
        }
    }
}

```

```

        var buffer = new byte[dataLength];
        int totalRead = 0;
        while (totalRead < dataLength)
        {
            int read = await client.ReadAsync(buffer,
totalRead, dataLength - totalRead);
            if (read == 0) break;
            totalRead += read;
        }

        if (totalRead != dataLength)
        {
            Console.WriteLine($"Ошибка: прочитано {totalRead}
байт, ожидалось {dataLength}.");
            return;
        }

        string json =
System.Text.Encoding.UTF8.GetString(buffer);
        var recipes =
JsonSerializer.Deserialize<List<string>>(json, new
JsonSerializerOptions());

        if (recipes != null)
        {
            Console.WriteLine($"Получено {recipes.Count} ID
рецептов.");

            MyForm.GetInstance().UpdateRecipes(recipes);
        }
    }
    catch (TimeoutException)
    {
        Console.WriteLine("Ошибка: не удалось подключиться к
каналу рецептов в течение 30 секунд.");
        if (!process.HasExited)
        {
            try { process.Kill(); }
            catch { /* Игнорируем ошибки при завершении */ }
        }
    }
    catch (IOException ex)
    {
        Console.WriteLine($"Ошибка именованного канала
(рецепты): {ex.Message}");
        if (!process.HasExited)
        {
            try { process.Kill(); }
            catch { /* Игнорируем ошибки при завершении */ }
        }
    }
    catch (JsonException ex)
    {

```

```

        Console.WriteLine($"Ошибка обработки JSON (рецепты):
{ex.Message}");
        if (!process.HasExited)
        {
            try { process.Kill(); }
            catch { /* Игнорируем ошибки при завершении */ }
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Произошла ошибка (рецепты):
{ex.Message}");
        if (!process.HasExited)
        {
            try { process.Kill(); }
            catch { /* Игнорируем ошибки при завершении */ }
        }
    }
}

private static void GenCreateRecipes()
{
    Console.WriteLine("=== Примеры Create-рецептов ===\n");

    // 1. Compacting
    var compacting = new CreateCompactingConfig();
    compacting.Inputs.Add(new CreateItemIngredientComponent {
ItemId = "minecraft:coal_block" });
    compacting.Outputs.Add(new CreateItemComponent { ItemId =
"minecraft:diamond", Count = 1 });
    compacting.Modifiers.Heat = CreateHeatType.Heated;
    Console.WriteLine("// Compacting (с нагревом)");
    Console.WriteLine(compacting.GenerateJsCode());
    Console.WriteLine();

    // 2. Deploying
    var deploying = new CreateDeployingConfig();
    deploying.Inputs.Add(new CreateItemIngredientComponent {
ItemId = "minecraft:coal_block" });
    deploying.Inputs.Add(new CreateItemIngredientComponent {
ItemId = "minecraft:sand" });
    deploying.Outputs.Add(new CreateItemComponent { ItemId =
"minecraft:diamond", Count = 1 });
    deploying.Modifiers.KeepHeldItem = true;
    Console.WriteLine("// Deploying (с сохранением
предмета)");
    Console.WriteLine(deploying.GenerateJsCode());
    Console.WriteLine();

    // 3. Emptying
    var emptying = new CreateEmptyingConfig();
    emptying.Inputs.Add(new CreateItemIngredientComponent {
ItemId = "minecraft:water_bucket" });

```

```

        emptying.Outputs.Add(new CreateItemComponent { ItemId =
"minecraft:bucket", Count = 1 });
        emptying.Outputs.Add(new CreateItemComponent { ItemId =
"minecraft:water", Count = 1000 });
        Console.WriteLine("// Emptying (опустошение)");
        Console.WriteLine(emptying.GenerateJsCode());
        Console.WriteLine();

        // 4. Filling
        var filling = new CreateFillingConfig();
        filling.Inputs.Add(new CreateItemIngredientComponent {
ItemId = "minecraft:bucket" });
        filling.Inputs.Add(new CreateFluidIngredientComponent {
FluidId = "minecraft:water", Amount = 1000 });
        filling.Outputs.Add(new CreateItemComponent { ItemId =
"minecraft:water_bucket", Count = 1 });
        Console.WriteLine("// Filling (наполнение)");
        Console.WriteLine(filling.GenerateJsCode());
        Console.WriteLine();

        // 5. Crushing
        var crushing = new CreateCrushingConfig();
        crushing.Inputs.Add(new CreateItemIngredientComponent {
ItemId = "minecraft:coal_block" });
        crushing.Outputs.Add(new CreateItemComponent { ItemId =
"minecraft:diamond", Count = 1 });
        crushing.Outputs.Add(new CreateItemComponent { ItemId =
"minecraft:emerald", Count = 1, Chance = 0.5f });
        crushing.Modifiers.ProcessingTime = 500;
        Console.WriteLine("// Crushing (дробление с шансами и
временем)");
        Console.WriteLine(crushing.GenerateJsCode());
        Console.WriteLine();

        // 6. Mixing
        var mixing = new CreateMixingConfig();
        mixing.Inputs.Add(new CreateItemIngredientComponent {
ItemId = "minecraft:coal" });
        mixing.Inputs.Add(new CreateFluidIngredientComponent {
FluidId = "minecraft:water", Amount = 1000 });
        mixing.Outputs.Add(new CreateItemComponent { ItemId =
"minecraft:diamond", Count = 1 });
        mixing.Modifiers.Heat = CreateHeatType.Superheated;
        Console.WriteLine("// Mixing (смешивание с флюидами и
нагревом)");
        Console.WriteLine(mixing.GenerateJsCode());
        Console.WriteLine();

        // 7. Sandpaper Polishing
        var sandpaper = new CreateSandpaperPolishingConfig();
        sandpaper.Inputs.Add(new CreateItemIngredientComponent {
ItemId = "minecraft:coal_block" });

```

```

        sandpaper.Outputs.Add(new CreateItemComponent { ItemId =
"minecraft:diamond", Count = 1, Chance = 0.7f });
        Console.WriteLine("// Sandpaper Polishing (с шансом)");
        Console.WriteLine(sandpaper.GenerateJsCode());
        Console.WriteLine();

        Console.WriteLine("=== Конец примеров ===");
    }

    private static Dictionary<string, MinecraftItem> _itemDatabase
= new();

    private static async Task LoadItemsAsync()
    {
        var extractor = new ItemFluidWithIconExtractor();
        if (!extractor.IsValid())
        {
            Console.WriteLine("Путь к иконкам недействителен.");
            return;
        }

        var data = await extractor.ExtractAsync();
        // Объединяем Items и Fluids в один словарь
        foreach (var item in data.Items)
        {
            _itemDatabase[item.Id] = item;
        }
        foreach (var fluid in data.Fluids)
        {
            _itemDatabase[fluid.Id] = fluid;
        }

        Console.WriteLine($"Загружено {_itemDatabase.Count}
элементов.\n");
    }

    private static void ShowSampleItems(int count = 10)
    {
        Console.WriteLine($"=== Первые {count} элементов из базы
данных ===");
        int shown = 0;
        foreach (var kvp in _itemDatabase)
        {
            if (shown >= count) break;

            Console.WriteLine($"ID: {kvp.Key}");
            Console.WriteLine($"    Название:
{kvp.Value.LocalizedName}");
            Console.WriteLine($"    Мод: {kvp.Value.ModName}");
            Console.WriteLine($"    Тип: {kvp.Value.Type}");
            Console.WriteLine($"    Теги: {string.Join(", ",
kvp.Value.Tags)}");

```

```

        Console.WriteLine($"    Размер изображения:
{kvp.Value.ImageData.Length} байт");
        Console.WriteLine();

        shown++;
    }
}

private static void SearchAndDisplay(string query)
{
    var results = FindItemsByName(query);

    Console.WriteLine($"\\n=== Результаты поиска по запросу
'{query}' ===");
    if (results.Count == 0)
    {
        Console.WriteLine("Ничего не найдено.");
        return;
    }

    foreach (var item in results)
    {
        Console.WriteLine($"ID: {item.Id}");
        Console.WriteLine($"    Название:
{item.LocalizedName}");
        Console.WriteLine($"    Мод: {item.ModName}");
        Console.WriteLine($"    Тип: {item.Type}");
        Console.WriteLine($"    Теги: {string.Join(", ",
item.Tags)}}");
        Console.WriteLine();
    }
}

private static List<MinecraftItem> FindItemsByName(string
nameQuery)
{
    var results = new List<MinecraftItem>();
    // Поиск без учёта регистра
    string lowerQuery = nameQuery.ToLowerInvariant();

    foreach (var kvp in _itemDatabase)
    {
        // Проверяем как локализованное название, так и ID
        if
(kvp.Value.LocalizedName.ToLowerInvariant().Contains(lowerQuery)
||
        kvp.Key.ToLowerInvariant().Contains(lowerQuery))
        {
            results.Add(kvp.Value);
        }
    }

    return results;
}

```

```

    }

    private static void DemoItemSearch()
    {
        Console.WriteLine("=== Демонстрация поиска ===");
        Console.WriteLine("Введите часть названия или ID предмета/жидкости для поиска (или 'exit' для выхода):");

        while (true)
        {
            Console.Write("> ");
            string input = Console.ReadLine()?.Trim();

            if (string.IsNullOrEmpty(input) ||
                input.Equals("exit", StringComparison.OrdinalIgnoreCase))
            {
                break;
            }

            SearchAndDisplay(input);
        }
    }

    // В вашем Main или вызывающем методе:
    private static async Task RunItemDemo()
    {
        await LoadItemsAsync();
        if (_itemDatabase.Count == 0)
        {
            Console.WriteLine("Не удалось загрузить данные. Проверьте пути и файлы.");
            return;
        }
        ShowSampleItems(15);
        DemoItemSearch();
    }
}

```