МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«ТЮМЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ НАУК
Кафедра программного обеспечения

РЕКОМЕНДОВАНО К ЗАЩИТЕ В ГЭК
Заведующий кафедрой
К. т. н., доцент
_____ М. С. Воробьева
23.06. 2023 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
магистерская диссертация

ИССЛЕДОВАНИЕ И ПРОГРАММНАЯ РЕАЛИЗАЦИЯ МЕТОДА ОПРЕДЕЛЕНИЯ ПАТТЕРНОВ ПРОЕКТИРОВАНИЯ ДЛЯ ИСХОДНОГО КОДА НА ЯЗЫКЕ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

02.04.03 Математическое обеспечение и администрирование информационных систем
Магистерская программа «Разработка технологий Интернета вещей и больших данных»

Выполнил работу
студент 2 курса
очной формы обучения
Кева Юрий Андреевич

Научный руководитель
Д. п. н., профессор
Захарова Ирина Гелиевна

Рецензент
Старший преподаватель
Аврискин Михаил Владимирович

Тюмень
2023

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В настоящее время число IT специалистов постоянно возрастает по причине их востребованности. При этом их подготовка отличается из-за наличия значительного количества материалов и курсов для их обучения. Каждый программист пишет свой код по-своему, используя свои подходы, идеи и умения или, наоборот, отражая в коде недостаток квалификации.

Однако, несмотря на все разнообразие программ, сложились часто используемые конструкции кода, шаблоны, идиомы и принципы разработки. Они помогают ускорить командную разработку программного обеспечения (ПО), быстро решать часто возникающие задачи проектирования ПО, возникающие в процессе разработки, а также делать код более читаемым и расширяемым. Кроме «полезных» конструкций также существуют и другие шаблоны, отрицательно влияющие на разработку.

Все подобные конструкции называют *паттернами проектирования* [1]. Зачастую знание определенных паттернов проектирования считается обязательным для квалифицированного разработчика, ведь их знание и правильное использование позволяет достичь более качественного кода.

Среди паттернов есть те, которые хорошо описаны, как, например, паттерны проектирования по GoF'у [1], а есть те, которые наоборот не имеют конкретного описания, однако используются тоже часто.

Идея данной работы состоит в том, чтобы на основе анализа исходных кодов набора проектов находить часто используемые конструкции кода, которые являются известными или, наоборот, неизвестными паттернами.

Цель работы – разработать программный продукт для выявления паттернов проектирования на основе исходного кода одного или нескольких проектов, написанных на объектно-ориентированных языках программирования.

Для достижения данной цели были поставлены следующие **задачи**:

- Изучить существующие методы и подходы для анализа кода на наличие паттернов проектирования.
- Реализовать парсер для извлечения признаков, описывающих паттерны, и для создания структуры данных для их хранения.
- Разработать и реализовать алгоритмы для анализа паттернов.
- Разработать и протестировать программный продукт для анализа кода на паттерны проектирования.

ГЛАВА 1. ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

1.1ОПРЕДЕЛЕНИЕ ПАТТЕРНА ПРОЕКТИРОВАНИЯ

В настоящее время существует множество работ, исследующих исходный код на наличие паттернов. Однако такие работы сосредоточены на поиске конкретного паттерн.

Анализ кода на паттерны проектирования может быть применен так же, как и другие статические анализаторы кода: для поиска различных ошибок, которые встречаются в проектах с множеством антипаттернов, или наоборот анализ проекта на его качество, если иные паттерны проектирования.

Проектирование программ в объектно-ориентированной парадигме – сложная задача, ведь зачастую необходимо не только решить поставленную задачу, но и сделать программу и ее архитектуру достаточно гибкими и расширяемыми, чтобы в будущем можно было без лишних проблем поддерживать ПО и добавлять в него новый функционал.

Во время проектирования ПО часто возникают одни и те же задачи и проблемы, которые были решены уже давно. Именно поэтому более опытные разработчики используют уже готовые решения этих проблем, прибегая к устоявшимся конструкциям кода для их решения.

Такие повторяющиеся конструкции кода называются паттернами проектирования или шаблонами проектирования. Как правило они представляют собой описание конструкции кода, которая решает определенную задачу. Паттерн проектирования именует, абстрагирует и идентифицирует ключевые аспекты структуры общего решения, которые и позволяют применить его для создания повторно используемого дизайна [1].

Существует множество видов паттернов проектирования, начиная от стандартных паттернов проектирования, сформулированных «Бандой

Четырех» [2], и заканчивая антипаттернами, которые показывают, как проблему решать не надо. Рассмотрим наиболее популярные виды паттернов.

В первую очередь рассмотрим основные, или стандартные, паттерны проектирования, которые были сформулированы «Бандой Четырех». Данный вид паттернов представляет собой конечный список архитектурных паттернов проектирования. Чаще всего под паттернами проектирования понимаются именно эти паттерны. Данный вид шаблонов делится на 3 типа:

- Порождающие – шаблоны проектирования, которые абстрагируют процесс инстанцирования. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов.
- Структурные шаблоны – шаблоны, определяющие различные сложные структуры, которые изменяют интерфейс уже существующих объектов, позволяя облегчить разработку и оптимизировать программу.
- Поведенческие шаблоны – шаблоны, определяющие взаимодействие между объектами, увеличивая гибкость программы.

Далее приведем несколько примеров паттернов для каждого типа.

Абстрактная фабрика – порождающий шаблон. Данный паттерн описывает структуры программы для создания семейств взаимозависимых объектов без спецификации конкретных классов.

Строитель – порождающий шаблон. Позволяет отделить конструирование сложного объекта от его представления, благодаря чему в ходе одного и того же процесса конструирования могут получаться разные представления.

Фабричный метод – порождающий шаблон. Позволяет отделить интерфейс создания объектов и оставить подклассам решение о том, экземпляр какого класса должен быть создан.

Прототип – порождающий шаблон. Задает виды создаваемых объектов с помощью экземпляра прототипа, что позволяет создавать новые экземпляры класса путем копирования прототипа.

Адаптер – структурный шаблон, преобразующий интерфейс класса в другой интерфейс, на который рассчитаны другие классы. Решает задачу совместной работы классов, которые до этого были не совместимыми.

Мост – структурный шаблон, позволяет отделить абстракцию и ее реализацию для независимого использования обоих.

Фасад – структурный шаблон, объединяющий набор интерфейс в один общий, что упрощает взаимодействие с системой.

Приспособленец – структурный шаблон. Применяющий совместное использование для поддержки множества мелких объектов.

Команда – поведенческий шаблон. Инкапсулирует запрос в объекте, что позволяет параметризовать их для разных запросов, контролировать и хранить их.

Интерпретатор – поведенческий шаблон, определяющий представление грамматики для заданного языка, а также непосредственно интерпретатор приложений для этого языка.

Хранитель – поведенческий шаблон, позволяющий вынести внутренне состояние объекта без нарушения инкапсуляции.

Наблюдатель – поведенческий шаблон. Определяет зависимость между объектами между объектами таким образом, что при изменении объекта все зависимые объекты оповещаются и автоматически обновляются.

Кроме вышеуказанных паттернов существует множество других, которые решают более специфические задачи. Например, различные идиомы, которые решают специфические для конкретного языка проблемы, или антипаттерны, которые решают проблему ошибочным методом.

Примером идиомы для языка C++ может служить идиома *RAII – resource acquisition is initialization*. Данная идиома решает проблему управления памяти, посредством использования конструкторов и деструкторов для управления временем жизни объектов внутри класса.

Примером для антипаттернов является паттерн Полтергейст. Данный паттерн подразумевает наличие объекта без какого-либо состояния с минимальным временем жизни, существующего только для вызова функций из объекта другого класса

1.2. МЕТОДЫ ПОИСКА ПАТТЕРНОВ ПРОЕКТИРОВАНИЯ

Для поиска паттернов проектирования можно использовать различные методы, такие как методы статического анализа кода и методы машинного обучения. В статьях с близкой темой были рассмотрены объектно-ориентированные языки программирования и различные способы анализа кода на паттерны. Также есть множество методов обработки исходного кода для дальнейшего анализа.

В статье [3] для поиска паттернов проектирования были использованы технологии машинного обучения. Для выполнения поставленной задачи (классификации) были использованы следующие методы и алгоритмы: K-ближайших соседей, линейные модели и деревья решений. Однако такая модель была ограничена следующими паттернами: *singleton, adapter, composite, decorator* и *factory method*.

Распознавание проводилось по признакам паттернов, однако в статье не было указано, какие именно признаки были выделены. Также было неизвестно какой язык программирования был выбран для проектов в выборке.

Результатом было сравнение трех выбранных методов распознавания паттернов. Автор статьи отмечает, что наилучший результат показали деревья решений, однако процент верно определенных паттернов указан не был.

В статье [4] для распознавания были выбраны иные паттерны:

- Observer
- Mediator
- Chain of responsibility
- Visitor

В данной работе паттерном являлась определенная зависимость между классами, методами и модификаторами доступа. Например паттерн observer был описан как класс с методами для подписки и отписки на уведомления, а

так же саам метод для уведомления, который уведомляет подписчиков о событии, или изменении состояния объекта.

Исходный код проектов был написан на Java. Для обработки кода и построения AST дерева используется парсер. Далее с помощью методов статического анализа кода ищется фрагмент кода, который может содержаться паттерн.

Аналогично ищутся другие паттерны, выбранные автором.

В результате распознавание не давало ложных результатов при соотнесении паттерна с кодом, однако процент ненайденных паттернов был велик.

Во многих статьях авторы ссылаются на сравнение основных методов анализа кода на паттерны, где описаны их преимущества и недостатки [5]

Большая часть методов поиска паттернов основана на статическом анализе кода. Эти методы сфокусированы на анализ структуры классов, которые соответствуют определенных критериям, из этого следует, что малейшие расхождения приведут к потере паттерна. Однако паттерны могут иметь разную реализацию, это зависит как от задачи, так и от используемых инструментов. Например, даже простейший паттерн singleton, может иметь несколько реализаций в зависимости от желания того, кто его реализовывал. Примеры разных реализаций:

1. При создании нового экземпляра класса будет проверяться наличие ссылки на объект класса, если она есть, то считается, что создавать другие экземпляры не требуется.
2. Вместо ссылки на экземпляр может проверяться специально созданное логическое значение, проверка которого может быть проще и, в некоторых ситуациях, надежнее, нежели проверка наличия самого экземпляра.

Другие методы статического анализа основаны на анализе программы как графа и поиске определенного подграфа соответственно, однако данный подход имеет те же проблемы с различными реализациями паттернов, что и предыдущий.

Кроме методов статического анализа и машинного обучения существуют также комбинированные подходы, которые анализируют структуру программы на наличие структур, похожих на паттерн, после чего с помощью методов машинного обучения проводится анализ найденной структуры.

Еще одним методом является динамический анализ кода, они основаны на анализе пути выполнения программы, однако анализировать все пути выполнения при условии, что данные фрагментарны – гораздо более сложная задача, результаты которой зависят от конкретных условий выполнения программы.

Некоторые методы являются многофазовыми и используют несколько проходов, например, статического анализа. Так, некоторые методы состоят из нескольких шагов:

1. Определение паттернов. На этом этапе специалисты определяют набор паттернов для распознавания, а также структуру и роли для них.
2. Определение метрики. Специалисты определяют метрики для оценки ролей, определенных на первом шаге. Для этой цели используется GQM (Goal Question Metric).
3. Машинное обучение. Специалисты подают на вход программы, содержащие паттерны, в метрическую систему измерения и получают измерения для каждой роли. Далее эти измерения подаются в симулятор машинного обучения. После обучения они проверяют предположение для каждой роли, и если результаты

проверки неудовлетворительны, они возвращаются ко второму шагу и пересматривают метрики.

4. Обнаружение паттернов: разработчики вводят роли кандидатов, определенные на четвертом шаге, в систему обнаружения паттернов, используя структуру паттерна, определенную на шаге 1. Эта система автоматически определяет закономерности.

Всего было выбрано 12 паттернов, однако на каком языке программирования были выбраны проекты, какую предобработки проходили программы и в каком виде собираются метрики – неизвестно.

В результате работы значение метрики recall составило от 80% до 100%.

В самой статье подробно описаны разные методы распознавания и их преимущества с недостатками, но используемый метод описан недостаточно детально.

Кроме программ для Java для распознавания используется код на языке C# [6]. Т. к. в работе используется машинное обучение, то авторами была собрана обучающая выборка, которая в итоге содержит 60 размеченных проектов на вышеуказанных языках. Тестовая выборка же содержала 15 проектов.

Выбор методов машинного обучения обусловлен возможностью более простым расширение в сторону добавления новых паттернов в систему. Для использования методов статического анализа на основе поиска по определенным правилам для расширения числа распознаваемых паттернов необходимо описывать правила, по которым будет вестись поиск, в то время как модель машинного обучения выделит эти признаки сама в ходе обучения.

Для распознавания паттернов проектирования в исходном коде на языках программирования C, C++, C# и Java был использован инструмент srcML [7], который создает XML-представление исходного кода. Далее из исходного кода извлекаются признаки, которые будут использоваться для

обучения модели. Для выявления метрик во время выполнения программы используется динамический анализ, который следует за статическим анализом. Значения метрик нормализуются в диапазоне от 0 до 1.

Список паттернов, выбранных для распознавания: observer, adapter, factory. Так же был реализован парсер для извлечения метрик из XML представлений.

Используемые в работе метрики:

- Количество классов.
- Количество полей.
- Количество статических полей.
- Количество методов.
- Количество статических методов.
- Количество интерфейсов.
- Количество абстрактных интерфейсов.

Далее представлена архитектура нейронной сети для выполнения задачи поиска паттернов:

- Входной слой – 19 нейронов
- Первый слой – 32 нейрона, ф-я активации – ReLU
- Второй слой – 16 нейронов
- Выходной слой – 3 нейрона, напрямую зависит от количеств распознаваемых паттернов.

Результат распознавания по метрике accuracy – 90%.

В работе [8] для распознавания паттернов проектирования были использованы методы машинного обучения на основе деревьев.

Набор данных был взят из репозитория P-MARt (pattern-like microarchitecture repository), в котором содержатся примеры паттернов, используемых в 9 проектах в открытом доступе.

Для распознавания были выбраны следующие паттерны: adapter и command. Автор мотивирует выбор их обще схожестью. Признаки каждого паттерна были получены с помощью IDE JBuilder. Всего авторами было выделено 36 признаков. После выделения признаков избыточные признаки были исключены.

Всего было использовано 2 метода: Naive bayes и GA-CFS. Также было использовано 3 алгоритма на деревьях.

Точность распознавания была указана около 96% по метрике accuracy.

В другой статье также была описана предварительная обработка исходного программного кода, сбор обучающей выборки и выделение признаков в программе для распознавания паттернов [9]. Для распознавания были выбраны следующие паттерны: builder, decorator, factory method, abstract factory, facade, memento, observer, prototype, proxy, singleton, visitor и adapter. Под признаками программы понимаются статические атрибуты исходного кода, такие как названия классов и методов, наследование, сложность конструкций if/else и т.д.

Авторы статьи решили собрать свой собственный корпус Java-проектов и выложить его в открытый доступ для дальнейших исследований, т. к. Существующие корпусы с размеченными паттернами проектирования либо слишком маленькие, либо на них ограничен открытый доступ. Для этого было собрано 14 000 проектов с Github, которые были очищены от unit-тестов, html и css файлов. Для разметки был разработан веб-инструмент, чтобы специалисты могли размечать паттерны в проектах. Среди оценщиков был достаточно высокий процент согласия в выборе метки для каждого паттерна.

В итоге получился размеченный корпус из 1400 проектов, содержащий хотя бы по 100 сущностей для каждого из 12 выбранных паттернов.

Аналогично предыдущим статьям из кода были выделены признаки. Всего было выделено 15 признаков:

1. ClassName – название Java-класса.
2. ClassModifiers – ключевые слова public, protected, private и т. д. – модификаторы доступа.
3. ClassImplements – бинарный признак, показывающий, реализует ли данный класс какой-либо интерфейс.
4. ClassExtends – бинарный признак, показывающий, является ли данный класс наследником другого класса.
5. MethodName – название метода.
6. MathodParam – параметры метода.
7. MethodReturnType – тип возвращаемого значения метода.
8. MethodBodyLineType – тип кода в методе: выражения присваивания, условные операторы и т. д.
9. MethodNumVariables – количество локальных переменных и параметров в методе.
10. MethodNumMethods – количество методов, которые вызываются внутри данного метода. Вызываемые методы могут принадлежать тому же классу, что и данный метод, а могут быть методами других классов.
11. MethodNumLine – количество строк в методе.
12. MethodIncomingMethod – количество методов, которые вызываются внутри данного метода. Вызываемые методы не принадлежат тому же классу, что и данный метод.

13. MethodIncomingName – названия методов, которые вызывает данный метод.

14. MethodOutgoingMethod – количество методов, вызывающих данный метод.

15. MethodOutgoingName – названия методов, вызывающих данный метод.

Процесс распознавания состоит из трех шагов: предобработка, построение модели и классификация.

Предобработка включает построение графа вызовов SCG (Static Call Graph) и SSLR представления после парсинга кода. SSLR – это представление признаков исходного кода на естественном языке, к которому далее будет применен алгоритм Word2Vec, чтобы сгенерировать JEM (Java Embedded Model). JEM послужит входными данными для классификатора, обученного на паттернах проектирования.

В результате получена точность около 80% по метрикам precision и recall.

В статье [10] описан метод поиска паттернов проектирования с использованием MLDA, или подход к многоуровневому обнаружению. Данный подход включает в себя 3 уровня обнаружения: парсинг, поиск и сравнение сигнатур. Задача парсинга – извлечь информацию из кода программы, после чего построить ее модель. На этапе поиска сопоставляется структура, полученная на этапе парсинга, с описанием паттерна. Сравнение сигнатур представляет собой сравнение набора правил, в виде которых сигнатуры представлены. Для такого сравнения используется инструмент CLIPS [11].

Для парсинга был использован открытый инструмент Javaparser [12], с помощью которого строится AST дерево программы. Всего было выделено 5 типов связи классов в коде, которые парсер обрабатывает: наследование,

зависимость, агрегация, ассоциация и реализация. Сами паттерны описываются как эти зависимости между классами.

На этапе поиска строится структура данных, с параллельной проверкой зависимостей между классами. Если в процессе поиска обнаруживается требуемое отношение, то будут искаться остальные, пока не будет сформирована полная структура паттерна.

На этапе сравнения сигнатур отсеиваются ложноположительные результаты, полученные на предыдущем этапе. Для этого применяется технология CLIPS для составления набора правил для задания нужных паттернов, а так же составления зависимостей в иерархии кода.

Всего в работе были собраны 990 классов и 80 интерфейсов, для которых были определены зависимости. Для поиска были выбраны 23 основных паттерна проектирования. В результате метрики precision и recall были около 85%.

ГЛАВА 2. ЗАДАЧА ПОИСКА ПАТТЕРНОВ ПРОЕКТИРОВАНИЯ

2.1. ФОРМАЛЬНАЯ ПОСТАНОВКА ЗАДАЧИ

По результатам анализа статей на данную тему был сделан вывод, что для поиска паттернов не требуется полный исходный код, т. к. достаточно выделить основные элементы, отражающую структуру кода и его иерархию.

Исходя из этого была поставлена следующая задача:

Дано: $S = \{s_1,\ s_2 \dots s_n\}$,

где $s_i -$ исходный код программы в виде AST

Пусть $c -$

конкретный участок кода, являющийся поддеревом дерева $s \in$

S. Определим функцию $Eq(c_i, c_j) = \begin{cases} c_i = c_j \to 1 \\ c_i \neq c_j \to 0 \end{cases}$

Пусть $P = \{p_1, p_2 \dots p_k\}$,

где $p_i = (s, C), C = (c_1, c_2 \dots c_l)$: $\forall c_i, c_j \in C\ Eq(c_i, c_j) = 1$.

Необходимо найти такой $p \in P$: $s\ \|C\| = max$

Такая постановка позволяет реализовать поиск для большинства классических объектно-ориентированных языков программирования, таких как Java, C#, C++ и т. п.

2.2. МЕТОДЫ И АЛГОРИТМЫ РЕШЕНИЯ

Для того, чтобы найти паттерны проектирования, необходимо определить структуру для описания паттерна, а также методы для их извлечения, анализа и дальнейшего использования. Т. к. для решения задачи было решено выбрать методы статического анализа кода, то структура описания паттернов должна передавать структуру программы для того, чтобы ее можно было использовать для анализа кода на паттерны.

Основной структурой для описания программы на любом языке программирования является AST дерево [13], поэтому определим соответствующую структуру для описания паттернов. Это позволит относительно просто извлекать паттерны из программы, а также сопоставлять паттерн и программу, чтобы определить, есть ли в этой программе определенные паттерны или нет.

Далее определим структуру для описания паттерна, исходя и определения. Т. к. паттерн — это часто повторяющая конструкция кода, то структура должна передавать частотность использования той или иной конструкции кода. Следующим этапом нужно определить, какие конструкции кода нужно описывать. В большинстве своем паттерны описывают архитектурные решения и иерархии классов, поэтому внутренняя реализация того или иного метода роли не играет, кроме случаев, когда описываются алгоритмы, которые являются паттернами для вычислений. Исходя из этого выделим основные конструкции кода, общие для большинства объектно-ориентированных языков: класс, поле и метод.

Следующим этапом после выбора необходимых данных для описания паттерна определим саму структуру. Т. к. будущая структура основана на AST деревьях, то также определим дерево и узел этого дерево. Узел дерева будет содержать данные о конструкции кода вместе с частотой ее появления в коде, а само дерево будет содержать общую информацию о себе, например, глубину дерева.

Определим узел дерева паттернов. Узел должен содержать следующие данные:

- Частотность узла, чтобы описывать, как часто данный узел встречается в программах, на основе которых было составлено дерево.
- Вид узла, определяющий то, какую конструкцию кода описывает данный узел, а именно: класс, поле или метод.
- Атрибуты узла, описывающие соответствующие атрибуты из AST дерева, например атрибут static.
- Дочерние узлы, для сохранения иерархии кода программы.
- Родительский узел, необходимый для упрощения навигации по дереву
- Отношение узла к родительскому, описывающее тип связи между узлами, таким образом можно передать не только объявление внутренних членов класса, тип полей и сигнатуру методов, но и наследование и шаблонные параметры.

На основе такой структуры узла составим дерево. Каждое объявление класса является узлом дерева вида «Класс», дочерними узлами которого являются узлы вида «Поле», «Метод» и «Класс», которыми являются объявления соответствующих структур внутри этого класса, а также родительскими классами в рамках наследования и шаблонными параметрами в рамках шаблонов. Единственным потомком узла вида «Поле» является узел вида «Класс», описывающий тип поля. Если этот класс не объявлен в программе, т. е. является стандартным или библиотечным, или является рекурсивным, то этот узел также является листом дерева. Аналогично потомками узла вида «Метод» являются узлы вида «Класс», описывающие сигнатуру метода. Первым потомком всегда является узел для возвращаемого типа. На рисунке 1 изображен пример такого дерева для кода из приложения 1.

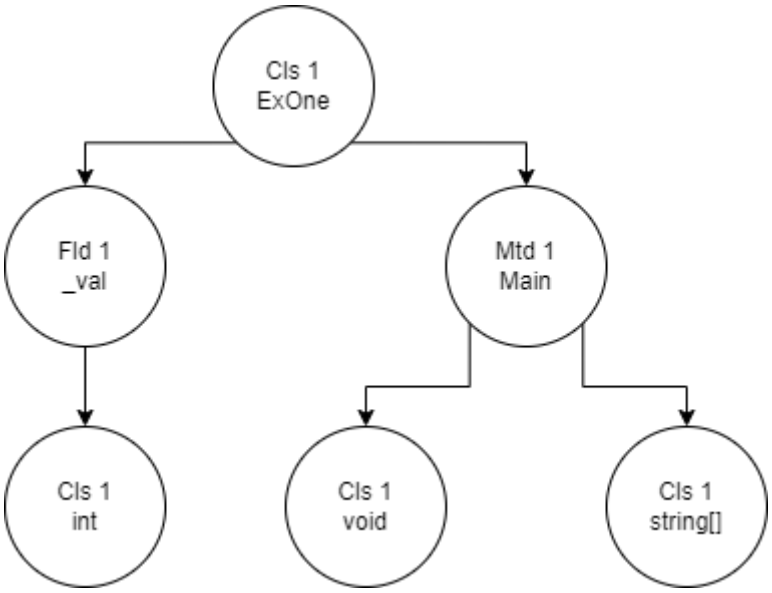

Рис. 1. Пример дерева паттернов.

Для каждого узла на рисунке обозначен его вид, cls, fld, mtd для классов, полей и методов соответственно, частотность, во всех узлах она равна 1, т. к. дерево строилось в проекте с одним классом, и имя конструкции, которой этот узел соответствует (отсутствует в дереве)

Подобная структура дерева позволит описать большую часть программ на объектно-ориентированных языках, как и передать их архитектуру. Также эта структура является достаточно гибкой для изменений, что позволит быстро изменить само дерево, добавив или исключив ту или иную конструкцию кода.

После определения структуры требуется определить операции над ней. Исходя из задачи нужно иметь следующие операции: создание дерева и сопоставление дерева с исходным кодом. При этом создавать дерево паттернов нужно несколькими путями: 1) создание дерева из исходного кода — это дерево будет полностью соответствовать исходной программе; 2) путем объединения деревьев, или *слияния* деревьев. Результатом объединения будет новое дерево, которое отражает все используемые конструкции кода в программах, на которых строились изначальные деревья.

Операция создания дерева паттернов из исходного кода представляет собой преобразование AST дерева входной программы в дерево паттернов, где узлы AST дерева нужного вида заносятся в дерево паттернов с сохранением структуры входной программы и единичной частотой.

Операция создания дерева из двух деревьев паттернов представляет собой слияние входных деревьев с суммированием частоты сливаемых узлов. Однако, т. к. некоторых случаях порядок узлов не имеет значение, то слияние должны проходить деревья с наиболее похожей структурой. Схожесть двух деревьев паттернов – показатель, являющийся отношением количества схожих узлов деревьев к максимальному количеству узлов в поддеревьях. В свою очередь узлы считаются схожими, если они имеют одинаковый вид и одинаковый тип связи с родительским узлом. Вычисление схожести позволит выбрать оптимальную стратегию слияния двух деревьев, в результате которого слияние пройдут большинство узлов.

Операция сопоставления необходима, чтобы проверить, используется ли во входном проекте паттерны из других проектов. Данная операция представляет собой вычисление схожести дерева паттернов, которое было составлено на выборке проектов, и дерева паттернов, полученного из проекта, который нужно проверить на паттерны.

После определения структуры необходимо построить AST деревья из входных проектов. Так как дерево паттернов универсально, то язык программирования проектов, на основе которых оно составляется, не имеет значения, поэтому был реализован алгоритм для парсинга, который так же является универсальным для любого ЯП, который поддерживает стандартные объектно-ориентированные конструкции. Алгоритм работы парсера изображен на рисунке 2. На вход парсера подается путь к директории с проектом, расширение файлов, которые надо прочитать и имя языка программирования, на котором написан проект. Выходом данного алгоритма является AST дерево проекта.

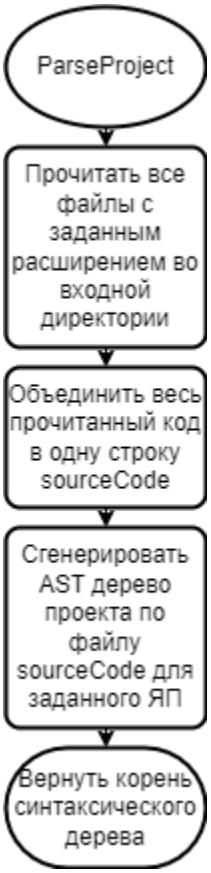

Рис.2. Алгоритм парсинга проекта.

Следующим этапом является превращение AST дерева в дерево паттернов. На этом этапе требуется извлечь из синтаксического дерева всю необходимую информацию и на ее основе создать дерево паттернов. Перед созданием дерева необходимо собрать информацию о каждом существующем в проекте классе, чтобы избежать ошибок, которые могут быть связаны с рекурсивными типами, библиотеками и стандартными типами данных. Пусть эта информация хранится в словаре clsInfoMap типа (string, ClassDeclaration), где ключ – имя класса, а значение – ссылка на узел AST дерева. Алгоритм создания дерева паттернов представлен на рисунке 3. Представленный алгоритм имеет несколько подалгоритмов: ParseClass, ParseField, ParseMethod. Эти алгоритмы изображены на рисунке 4.

Рис.3. Алгоритм создания дерева паттернов

Данный алгоритма представляет собой рекурсивный обход синтаксического дерева проекта, результатом которого является сформированное дерево паттернов.

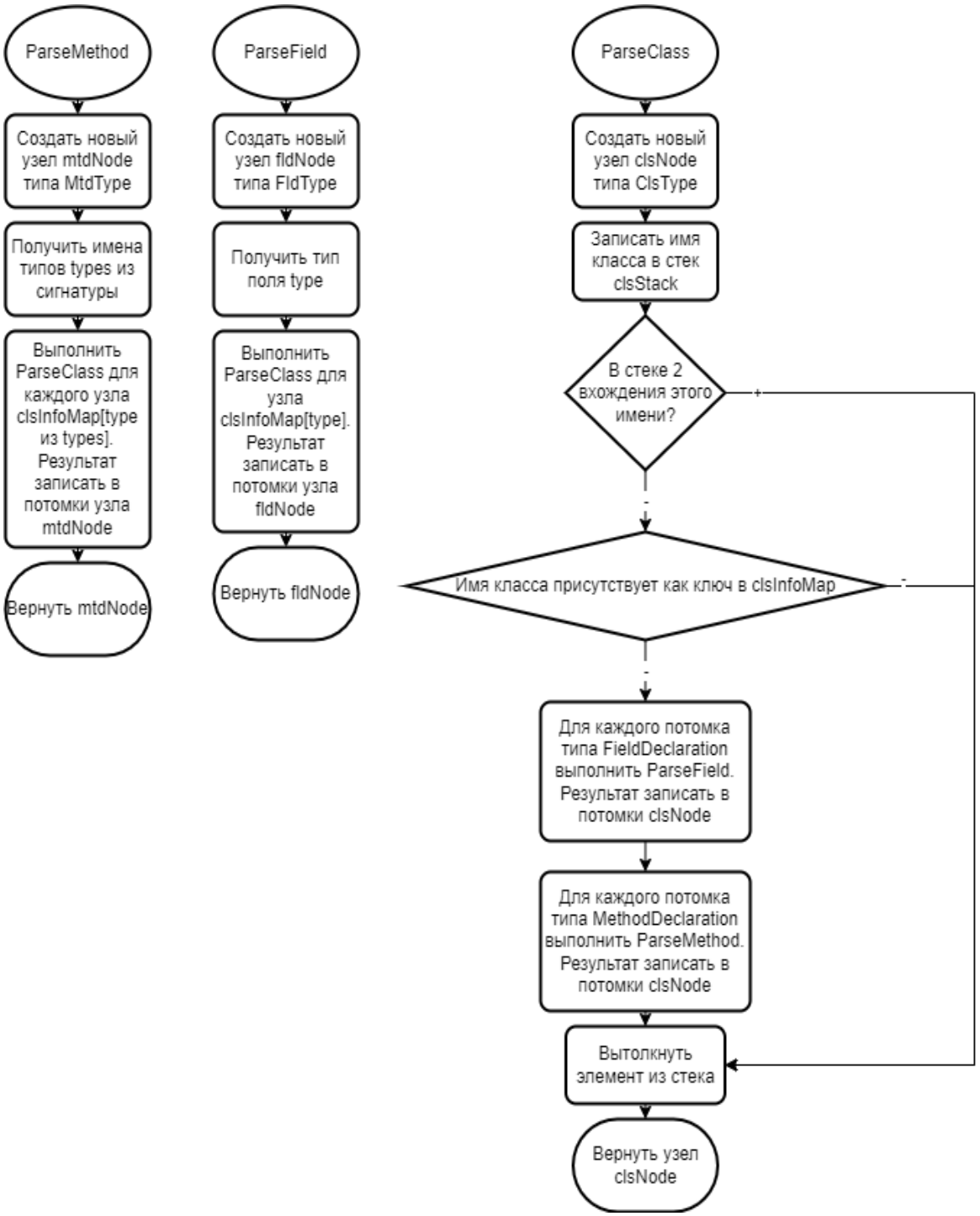

Рис.4. Алгоритмы ParseClass, ParseField, ParseMethod.

На вход каждого алгоритма подается узел определённого типа:

- На вход ParseAST подается узел типа Compilation Unit, который означает, по сути, корень AST дерева.
- На вход ParseClass подается узел типа Class Declaration, который является узлом объявления класса.

- На вход ParseField подается узел типа Field Declaration.
- На вход ParseMethod подается узел типа Method Declaration.

Итогом совокупной работы всех алгоритмов является дерево паттернов, дублирующие синтаксическое дерево. В данном дереве паттернов частота вех узлов равна 1. Листьями такого дерева являются узлы классов, которые по той или иной причине не имеют потомков (класс является базовым, как тип int, библиотечным, как List<T>, или рекурсивным, как PatternTreeNode) Пример дерева паттернов представлен на рисунке 5.

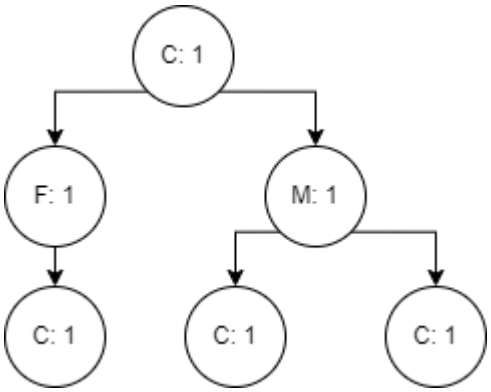

Рис.5. Пример дерева паттернов. Запись вида X: N внутри узла обозначает, что узел имеет вид X и частоту N соответственно, где C, F и M – виды узлов для узлов класса, поля и метода соответственно.

Второй операцией является слияние деревьев паттернов. Данная операция необходимо для составления финального дерева паттернов для всех проектов, на основе которых извлекались паттерны. Такое дерево содержит все возможные варианты архитектуры классов проектов с соответствующей частотностью. На вход подается 2 корня деревьев паттернов, на выходе получаем корень 3-го дерева. Алгоритм изображен на рисунке 6.

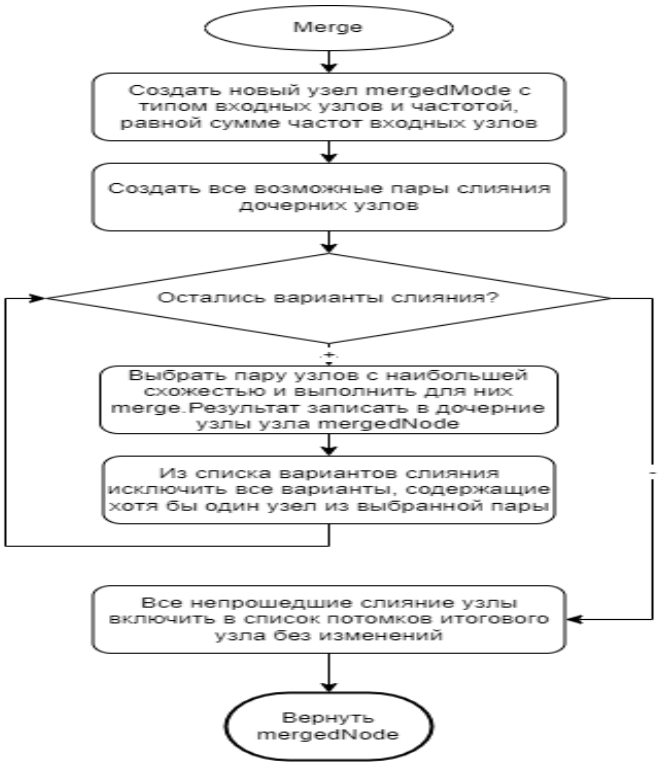

Рис.6. Алгоритм слияния деревьев

Так как с точки зрения архитектуры порядок объявления не имеет значения, то порядок дочерних узлов тоже не важен. В таком случае необходимо производить слияние наиболее похожих поддеревьев. Для этого было введено понятие схожести двух деревьев паттернов. Схожесть – показатель, показывающий, как много узлов можно слить между собой из общего числа узлов. Чем выше показатель, тем более деревья схожи между собой. Сам показатель представляет собой пару значений: (количество схожих узлов, общее число узлов). С помощью этого показателя можно выбрать оптимальную стратегию слияния двух деревьев. Данный показатель рассчитывается с помощью рекурсивного обхода двух деревьев паттернов.

ГЛАВА 3. ПРОЕКТИРОВАНИЕ И ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЯ

3.1. АРХИТЕКТУРА ПРИЛОЖЕНИЯ

Перед реализацией приложения необходимо спроектировать его архитектуру. Адекватная архитектура позволяет избежать множество проблем, возникающих при реализации приложения.

Из-за специфики работы приложения грузка данных, парсинг и любые другие функции, такие как анализ кода на паттерны должны работать независимо друг от друга. Еще одним требованием является гибкость настройки приложения. Это необходимо для возможности быстро изменять важные параметры приложения, не прибегая к изменениям кода.

Приложение должно иметь следующие основные функции:

- Загрузка проектов из открытых репозиториев.
- Парсинг проектов и формирование дерева паттернов.

Исходя из поставленных требований к архитектуре и функционала была спроектирована архитектура, которая позволяет реализовать приложение, функции которого не будет напрямую зависеть от других его функций, т. е., например, приложение будет иметь возможность пропустить выполнение загрузки проектов из открытых репозиториев и начать парсинг ранее загруженных проектов. Спроектированная архитектура в общем виде изображена на рисунке 7.

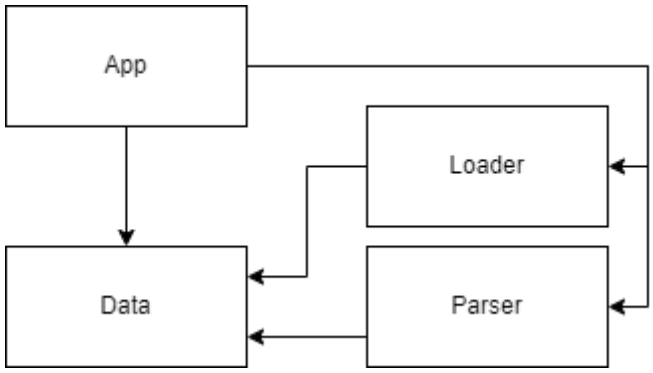

Рис. 7. Общая архитектура приложения

Приложение имеет следующие основные модули:

- Управляющий модуль (App) – управляющий модуль приложения, выполняющий задачи парсинга файла конфигурации, взаимодействия с интерфейса и исполнение выбранных функций приложения.
- Модуль модели данных (Data) – модуль, в котором объявлены все структуры данных необходимые для работы, а также методы и алгоритмы для работы с ними.
- Модуль загрузки данных (Loader) – модуль, представляющий функцию загрузки проектов из открытых репозиториев.
- Модуль парсинга проектов (Parser) – модуль, задачей которого является парсинг загруженных проектов и конструирование дерева паттернов на их основе.

3.2. ФОРМИРОВАНИЕ НАБОРА ПРОЕКТОВ

Модуль Loader представляет собой набор функций для поиска подходящих открытых репозиториев, а после их загрузкой. Все функции реализованы в виде методов специального класса «Loader», который является единственным классом в модуле.

Задача модуля Loader – загрузить проекты из открытых репозиториев согласно заданным требованиям:

- Язык программирования. По умолчанию задан язык C#
- Минимальный размер в мегабайтах. По умолчанию равен 50 мегабайтам
- Максимальный размер в мегабайтах. По умолчанию равен -1, что означает неограниченный максимальный размер.

Принцип работы данного модуля следующий: модуль сканирует открытые репозитории, выбирая все подходящие проекты, после чего клонирует определенное количество проектов в выбранную директорию, создавая для каждого проекта файл с метаинформацией, который содержит путь до проекта и язык программирования, на котором он написан.

Входными параметрами данного модуля являются:

- Список критериев, указанный выше. Если какой-то из критериев не указан, то берется стандартное значение.
- Количество проектов для выгрузки.
- Путь до директории, в которую будут клонироваться репозитории.

Все параметры берутся из файла конфигурации приложения.

Данный модуль был реализован с помощью языка программирования C# с помощью библиотеки Ocktokit [17]. Данный модуль представляет собой

единственный класс с функциями для получения списка репозиториев, их фильтрации, а после клонирования выбранных репозиториев. UML диаграмма представлена на рисунке 8.

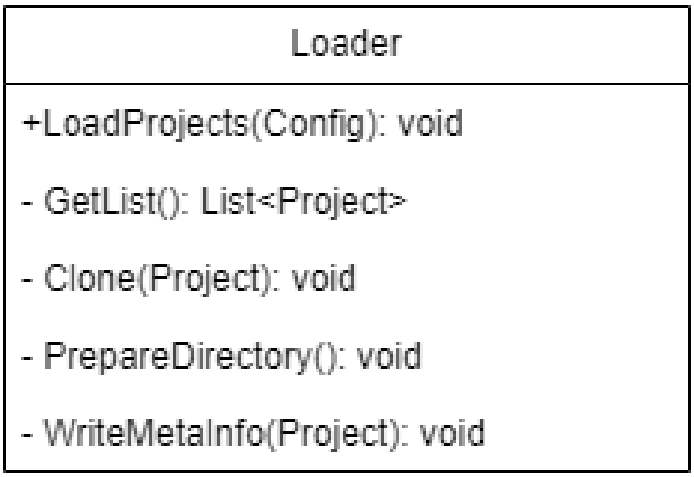

Рис. 8. Диаграмма класса Loader

3.3. ПРОЕКТИРОВАНИЕ ПАРСЕРА

Для извлечения дерева паттернов из проектов нужно реализовать парсер, который будет способен обрабатывать проекты на разных объектно-ориентированных языках. Это позволит добиться того, что можно будет выделить какие-то идиомы, выделить особенности проектирования на каком-либо языке программирования и т. д.

Для выполнения этого требования нужно разделить парсинг проекта в AST дерево и конструирование дерева паттернов, а также определить универсальный набор конструкций кода, который подходит для как можно более широкого числа объектно-ориентированных языков, что позволит построить независимое от конкретного языка дерево паттернов.

Такой подход разделения дерева паттернов и языка программирования имеет недостатки в виде потери некоторых важных конструкций, которые могут определить идиому, однако он также позволяет выделить более общие паттерны, независящие от реализации на конкретном языке программирования.

Исходя из требований выше необходимо решить 3 задачи:

1. Реализовать парсер для проектов, преобразующий исходный код в AST дерево.
2. Реализовать обход полученного AST дерева с его преобразование в дерево паттернов.
3. Сохранить результат в файл для дальнейшего использования.

Для выполнения первой задачи требуется выбрать соответствующую библиотеку, которая позволит компилировать исходный код на объектно-ориентированном языке программирования в AST дерево. Такая библиотека необходима, так как такая задача является слишком объемной и сложной.

Выбор был сделан в пользу библиотеки Tree-sitter[18], которая позволяет по заданной грамматике преобразовать текст в соответствующее AST дерево. Для данной библиотеки уже реализованы грамматики для большого числа популярных языков программирования, что позволяет без проблем реализовать парсер для любого нужного языка из поддерживаемых.

Вторая задача заключается в обходе полученного AST дерева, поиске узлов, которые соответствуют определенной конструкции кода, а после переносом этой конструкции в дерево паттернов.

Третья же задача заключается в сохранении результатов парсинга в файл, чтобы его можно было использовать в любой момент времени. Это позволит сохранять деревья паттернов, содержащие интересную, по той или иной причине, информацию для дальнейшего ее использования, как, например, сбор «библиотеки» паттернов и т. п.

При построении дерева паттернов были сделаны некоторые допущения, так как они не сильно влияют на архитектуру проекта, однако очень сильно усложняют разработку.

Одним из таких допущений является игнорирование объявления внутренних классов. Такие классы не сильно распространены и не оказывают большого влияния на архитектуру, однако они заметно усложняют анализ деревьев паттернов и алгоритмы для работы с ними.

Также при реализации парсера появились следующие проблемы:

- C#, выбранный как основной язык разработки, не поддерживается библиотекой.
- Сама библиотека является достаточно низкоуровневой и написанной на языке C, с дополнительными биндингами для других языков.

Для решения данных проблем можно либо найти другую библиотеку со схожим функционалом, либо реализовать парсинг проектов в AST отдельно от обхода AST, а после использовать «внешнюю» компоненту парсера как отдельную утилиту. Т. к. аналога библиотеки tree-sitter найдено не было, то было решено разделить парсер на две компоненты: утилита для парсинга проектов, выполняющая все функции парсера, и внутренний модуль приложения, который читает файлы с деревьями паттернов.

Модуль парсинга реализован с помощью второго способа, таким образом основное приложение вызывает внешнюю утилиту, с реализацией парсера. Архитектура парсера изображена на рисунке 9.

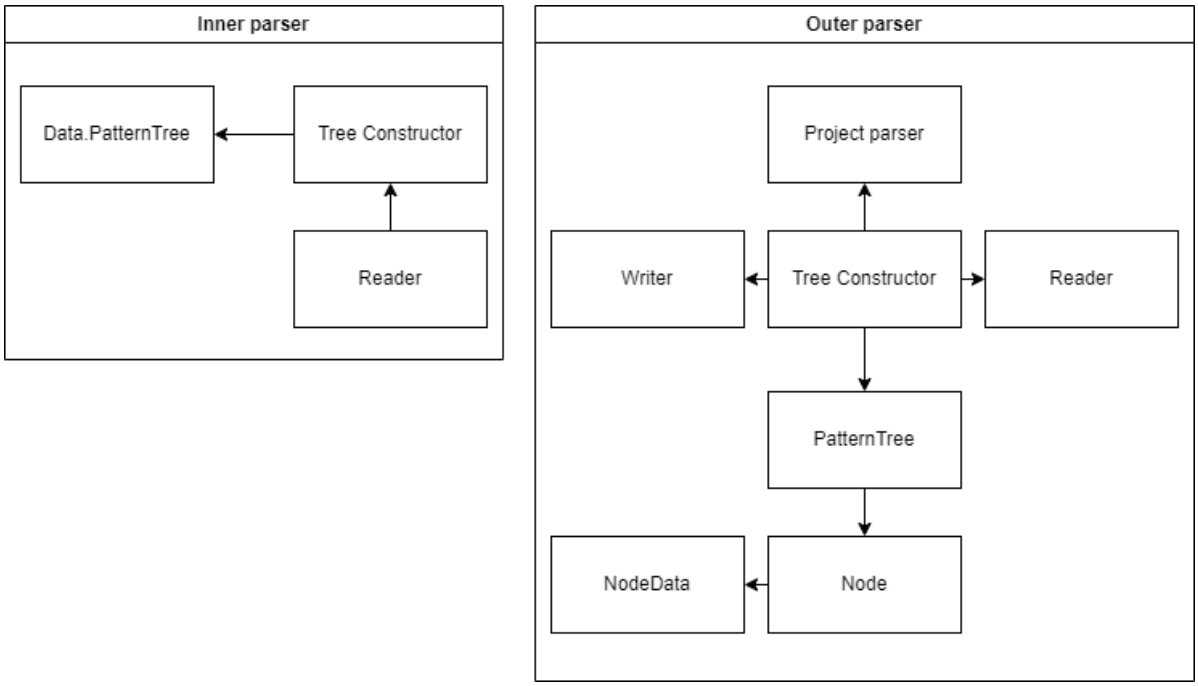

Рис. 9. Архитектура парсера

Внутренний модуль парсера реализует чтение файла с деревом паттернов и его конструирование из файла. Внешний модуль реализует парсинг проектов по заданной структуре, дублирующую структуру из основной программы. Таким образом сам процесс парсинга полностью делегирован внешней утилите. Однако у данного подхода есть проблема,

которая заключается в дублировании объявлений структур дерева в обоих модулях.

Проблема реализации парсера с помощью библиотеки tree-sitter, которая заключается в ее низкоуровневости и реализации самой библиотеки на языке C, была решена с помощью механизма FFI (Foreign Function Interface) [19]. Данный механизм обеспечивает взаимодействие между разными языками программирования, позволяя вызывать функции, написанные на одном языке программирования, используя другой язык

Использование подобного механизма позволило использовать высокоуровневые абстракции, например, классы, при условии, что оригинальный язык, на котором была написана нужная библиотека таких абстракций не имеет.

3.4. МОДЕЛЬ ДАННЫХ

Модуль модели данных хранит общие структуры и алгоритмы, которые относятся непосредственно к задаче поиска паттернов, а также к смежным задачам, таким как выгрузка проектов. Архитектура модуля данных изображена рисунке 10.

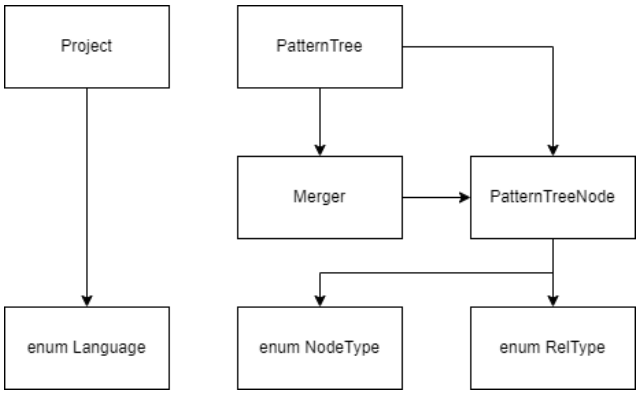

Рис. 10. Архитектура модуля данных

Данный модуль содержит следующие элементы:

- Project – структура, описывающая проект, загруженный с помощью модуля Loader. Содержит путь к проекту и язык программирования, который является для проекта основным
- PatternTree – структура, хранящая корень дерева паттернов и предоставляющая интерфейс взаимодействия с ним.
- PatternTreeNode – структура узла дерева
- Merger – класс, реализующий алгоритм слияния деревьев
- Enum NodeType – перечисление, определяющее виды узлов дерева паттернов
- Enum RelType – перечисление, определяющее список отношения узла к родительскому
- Enum Language – перечисление, определяющее список поддерживаемых языков программирования. На данный момент поддерживается только 1 язык: C#

3.5 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

Итоговые приложения были реализованы с использованием двух языков программирования: C# и Rust. Выбор данных языков обусловлен их универсальностью, достаточной простотой и безопасностью.

На языке C# было реализовано основное приложение, его интерфейс и часть алгоритмов для работы со структурой дерева. Интерфейс представляет собой консоль с выбором необходимой функции для выполнения{!}.

На языке Rust был реализован парсер, который через механизмы FFI обращается к интерфейсу библиотеки Tree-sitter, благодаря которой выполняется парсинг исходных кодов проектов. Выбор в пользу этого языка был сделан по причине его безопасности наличия высокоуровневых абстракций, которые сокращают время разработки, а также его возможностям разработки в функциональном стиле

Итоговое приложение является консольным, однако оно имеет простой интерфейс, позволяющий выбирать, что нужно выполнить и в каком порядке. Пример главного меню приложения представлен на рисунке 11.

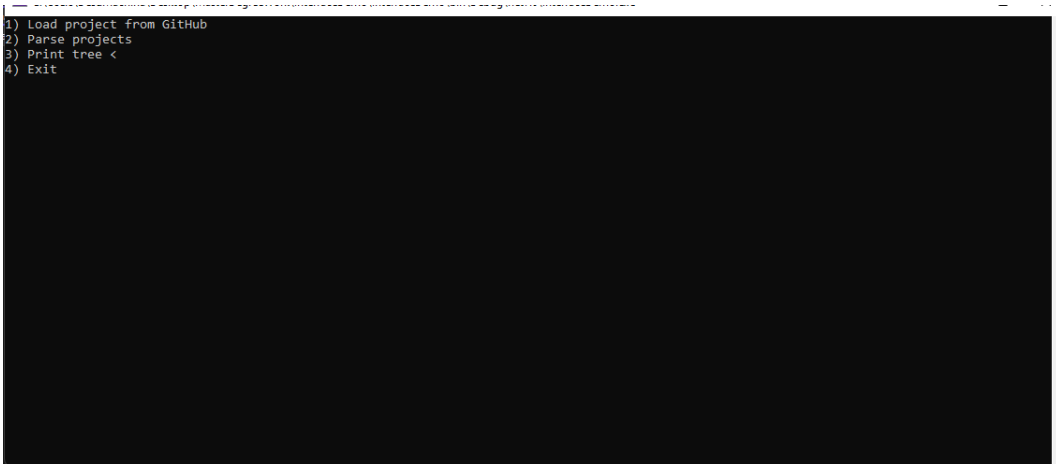

Рис. 11. Главное меню приложения.

Далее из главного меню производится выбор нужной функции и ввод данных, не представленных в конфигурации. Само приложение работает по принципу конечного автомата, что позволяет легко добавлять новые функции,

добавляя новое состояние. Упрощенная диаграмма классов представлена на рисунке 12. На данной диаграмме основные модули приложения, описанные выше, отображены другими блоками. На данной диаграмме все State-классы наследуются от интерфейса IState

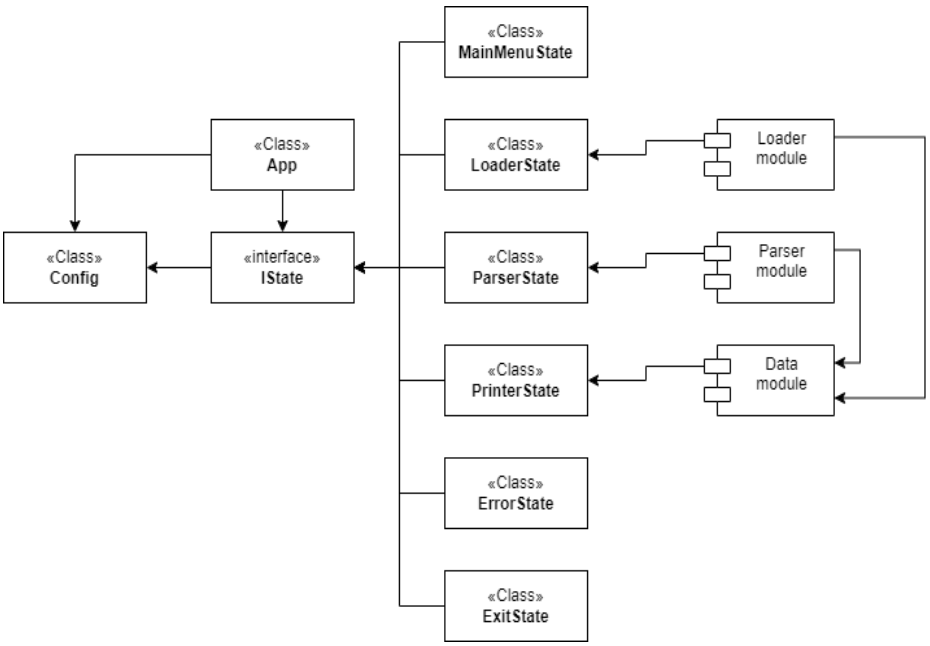

Рис. 12. Диаграмма классов управляющего модуля приложения.

Схема конечного автомата, на основе которого был реализован интерфейс представлен на рисунке 13.

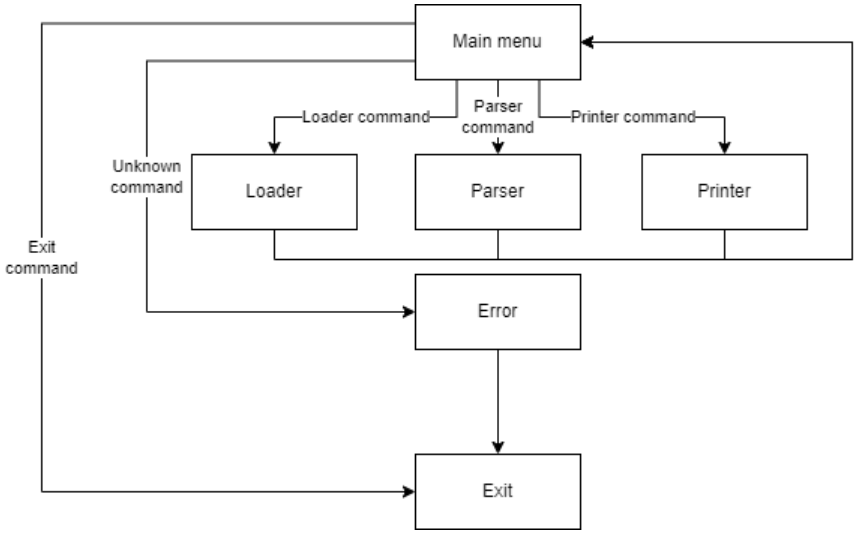

Рис. 13. Схема автомата.

ЗАКЛЮЧЕНИЕ

В результате работы были изучены существующие методы поиска паттернов, основанные как на методах машинного обучения, так и на методах статического анализа кода.

В ходе работы был реализован метод поиска паттернов на основе анализа синтаксических деревьев на наличие повторяющихся конструкций, которые могут являться не именованными паттернами. Результат достигнут трансформацией синтаксического дерева в дерево описания паттернов, содержащее необходимую информацию для описания структуры проектов. Данный метод работает вне зависимости от языка программирования и позволяет применять его к любому языку, если для него реализован соответствующий парсер.

В дальнейшем этот метод можно расширить, добавив новые алгоритмы для работы с деревьями паттернов, а также увеличив информативность этих деревьев.

СПИСОК ЛИТЕРАТУРЫ

1. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. П75 Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб: Питер, 2001. — 368 с.: ил. (Серия «Библиотека программиста»)
2. Степанчук И. А., Логинов Н. И. ПАТТЕРНЫ ПРОГРАММИРОВАНИЯ: ИСТОРИЯ ПОЯВЛЕНИЯ, ОСНОВНАЯ ИДЕЯ, КЛАССИФИКАЦИЯ //INTERNATIONAL INNOVATION RESEARCH: Сборник статей XII Международной научно-практической конференции в 3 частях. Том Часть 1. – 2018. – С. 138-144.
3. Алаа Ш. Методика подготовки и распознавания паттернов программного обеспечения с использованием моделей машинного обучения // Политехнический молодежный журнал / МГТУ им. Н. Э. Баумана, Москва, 2019.
4. Heuzeroth D. [et al.] Automatic design pattern detection // 11th IEEE International Workshop on Program Comprehension, 2003. С. 94-103.
5. Uchiyama S. [et al.] Detecting design patterns in object-oriented program source code by using metrics and machine learning // Journal of Software Engineering and Applications, 2014. С. 983.
6. Oberhauser R. A machine learning approach towards automatic software design pattern recognition across multiple programming languages // ICSEA 2020. С. 37.
7. srcML markup documentation. URL: https://www.srcml.org/documentation.html (дата обращения: 20.04.2023).

8. Mhawish M. Y., Gupta M. Software metrics and tree-based machine learning algorithms for distinguishing and detecting similar structure design patterns // SN Applied Sciences, 2020. С. 1-10.
9. Nazar N., Aleti A. Feature-based software design pattern detection // arXiv:2012.01708, 2020.
10. Al-Obeidallah M. G., Petridis M., Kapetanakis S. A Structural Rule-Based Approach fpr Design Patterns Recovery // International Conference on Software Engineering Research, Management and Applications. Springer, Cham, 2017. С. 107-124.
11. CLIPS: A Tool for Building Expert Systems, 2016. URL: http://www.clipsrules.net/.Cited5Jan2017
12. Javaparser: Java 1-15 parser and Abstract Syntax Tree for Java. URL: https://github.com/javaparser/javaparser
13. Ахо, Альфред В., Лам, Моника С., Сети, Рави, Ульман, Джеффри Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд. : пер. с англ -СПб. :Диалектика, 2020 – 1184 с.
14. Макконел С. Совершенный код. Мастер-класс. Пер с англ. – СПб.: БХВ 2022 – 896 с.
15. Фримен Э. [и др.] Паттерны проектирования. СПб.: Питер, 2011. 656 с.
16. Heuzeroth D. [et al.] Automatic design pattern detection // 11th IEEE International Workshop on Program Comprehension, 2003. С. 94-103.
17. Home Octokit.net Documentation. URL: https://octokitnet.readthedocs.io/en/latest/ (дата обращения: 20.04.2023).
18. Tree-sitter | Introduction URL: https://tree-sitter.github.io/tree-sitter/
19. FFI - The Rustonomicon URL: https://doc.rust-lang.org/nomicon/ffi.html

Приложение

Конструирование дерева во внешнем парсере.

```
pub struct ProjectParser<'a> {
    pub(crate) wanted_node_types: Vec<String>,
    pub(crate) source: String,
    pub(crate) class_map: HashMap<String, &'a PTNode>
}
impl ProjectParser<'_> {
    fn get_node_name(&self, cursor: &mut TreeCursor) -> String {
        if cursor.goto_first_child() {
            while cursor.node().kind() != "identifier" {
                if !cursor.goto_next_sibling() {
                    break;
                }
            }
            let name =
&self.source[cursor.node().start_byte()..cursor.node().end_byte()];
            cursor.goto_parent();
            return String::from(name)
        }
        String::from("NoName")
    }
    fn parse_class(&self, cursor: &mut TreeCursor) -> PTNode {
        let node_name = self.get_node_name(cursor);
        PTNode { node_type: NodeType::ClassNode, name:
self.get_node_name(cursor), con_type: ConnectionType::Default, childs:
Vec::new() }
    }
    fn parse_field(&self, cursor: &mut TreeCursor) -> PTNode {
        PTNode { node_type: NodeType::FieldNode, name:
self.get_node_name(cursor), con_type: ConnectionType::Default, childs:
Vec::new() }
    }
    fn parse_method(&self, cursor: &mut TreeCursor) -> PTNode {
        PTNode { node_type: NodeType::MethodNode, name:
self.get_node_name(cursor), con_type: ConnectionType::Default, childs:
Vec::new() }
    }
    fn parse_generic(&self, cursor: &mut TreeCursor) -> PTNode {
        let node = cursor.node();
        PTNode { node_type: NodeType::ClassNode, name:
String::from(&self.source[node.start_byte()..node.end_byte()]), con_type:
ConnectionType::TemplateParameter, childs: Vec::new() }
    }
    fn parse_predefined(&self, cursor: &mut TreeCursor) -> PTNode {
        let node = cursor.node();
```

```
        PTNode { node_type: NodeType::ClassNode, name:
String::from(&self.source[node.start_byte()..node.end_byte()]), con_type:
ConnectionType::Default, childs: Vec::new() }
    }
    fn parse_childs(&self, cursor: &mut TreeCursor) -> Vec<PTNode> {
        let mut res = Vec::new();
        if cursor.goto_first_child() {
            let mut node = self.parse_ast_node(cursor);
            match node {
                ASTNode::Wanted(x) => res.push(x),
                ASTNode::Unwanted(mut x) => res.append(&mut x)
            }
            while cursor.goto_next_sibling() {
                node = self.parse_ast_node(cursor);
                match node {
                    ASTNode::Wanted(x) => res.push(x),
                    ASTNode::Unwanted(mut x) => res.append(&mut x)
                }
            }
            cursor.goto_parent();
        }
        res
    }
    fn parse_node(&self, cursor: &mut TreeCursor) -> PTNode {
        let mut res = match cursor.node().kind() {
            "class_declaration" => self.parse_class(cursor),
            "field_declaration" =>  self.parse_field(cursor),
            "method_declaration" => self.parse_method(cursor),
            "generic_name" => self.parse_generic(cursor),
            "predefined_type" => self.parse_predefined(cursor),
            //"identifier" => self.parse_identifier(cursor),
            _ => PTNode::default()
        };
        res.childs.append(&mut self.parse_childs(cursor));
        res
    }
    fn parse_ast_node(&self, cursor: &mut TreeCursor) -> ASTNode {
        if self.wanted_node_types.contains(&String::from(cursor.node().kind())) {
            ASTNode::Wanted(self.parse_node(cursor))
        }
        else {
            ASTNode::Unwanted(self.parse_childs(cursor))
        }
    }
    pub fn parse_project(&self) -> Vec<PTNode> {
        let language = unsafe { tree_sitter_c_sharp()} ;
        let mut parser = Parser::new();
        let mut res = Vec::new();
        parser.set_language(language).unwrap();
```

```
        let ast = parser.parse(String::clone(&self.source),
None).unwrap();        let test_node = ast.root_node();
        let mut cursor: TreeCursor = test_node.walk();
        let node = self.parse_ast_node(&mut cursor);
        match node {
            ASTNode::Wanted(x) => res.push(x),
            ASTNode::Unwanted(mut x) => res.append(&mut x)
        }
        res
    }
}
```