Санкт-Петербургский государственный университет

Системное программирования

Группа 19Б.10-мм

Порсев Егор Валерьевич

Реализация и внедрение end-to-end системы обработки данных в инструмент Astminer

Отчёт по учебной практике

Научный руководитель:
доцент каф. СП, д.ф.-м.н., проф. Т.А. Брыксин

Консультант:
программист-исследователь JetBrains Е.С. Спирин

Санкт-Петербург
2021

Оглавление

1. Введение

Программное обеспечение стало неотъемлемой частью нашей жизни. Постоянно появляются новые приложения, программы и сайты, приходя на смену старым, код которых в свою очередь не утрачивается. Разработка программных продуктов только набирает обороты, пишется все больше и больше кода, создаются новые проекты [1]. Доступный код всех программ можно рассматривать как постоянно увеличивающийся объем данных, который можно использовать в любых целях.

Так развивается область машинного обучения для программной разработки [10], одной из целей которой является упрощения процесса написания кода. Например, помощь программисту релевантными подсказками и поиск ошибок в коде [6, 11, 12]. Многим алгоритмам машинного обучения требуются большие объемы обработанных данных, полученных из исходного кода программ.

Модели машинного обучения, упрощающие процесс разработки, используются в разных отраслях, и поэтому обучаются на данных, полученных из исходного кода на разных языках программирования (ссылки). Также для разных алгоритмов машинного обучения требуется поразному обработать данные [3, 6]. Может получиться так, что 2 алгоритма должны получать данные в двух абсолютно разных форматах. Поэтому исследователям приходится самим разрабатывать инструменты, собирающие и обрабатывающие данные, для области свое исследования.

Инструмент *Astminer* [9] решает задачу извлечения разных форматов данных из исходного кода. Он умеет обрабатывать исходный код, написанный на языках *Java*, *Python*, *JavaScript*, *C++* и извлекать данные в требуемом пользователем формате.

Пользователь может запускать *Astminer* через интерфейс командной строки, настраивая инструмент через переданные аргументы. Существует 4 независимых режима работы, каждый из которых реализован отдельным классом, но у всех их есть общие детали, например: запуск разбора исходного кода, сохранение результатов и т.д. Эти об-

щие детали выражаются в повторяющемся коде во всех реализациях режимов работы.

Поскольку *Astminer* применяется в разных областях, количество настраивающих его параметров только растет. Поэтому при запуске инструмента через командную строку пользователю может быть необходимо передать до 16 параметров. В таком случае все параметры будут в одной строке текста и никак не отделены по смыслу.

При добавлении новых форматов обработки данных количество настраиваемых параметров увеличится, что усложнит работу исследователю, который хочет запустить инструмент. Также программисту придется создавать новый режим работы инструмента, копируя существующий код, потому что основные этапы обработки данных во всех режимах работы схожие.

1.1. Цель и задачи

Целью является разработка и внедрение решения для настраиваемого пользователем запуска обработки данных в приложении *Astminer*

Для выполнения данной цели были поставлены следующие задачи:

1. Изучить существующую архитектуру и функциональность инструмента;
2. Выделить основные шаги обработки данных и требования к ним;
3. Спроектировать и реализовать решение, запускающее шаги обработки данных по настройкам пользователя;
4. Внедрить и апробировать реализованное решение;

2. Обзор

2.1. Базовые понятия

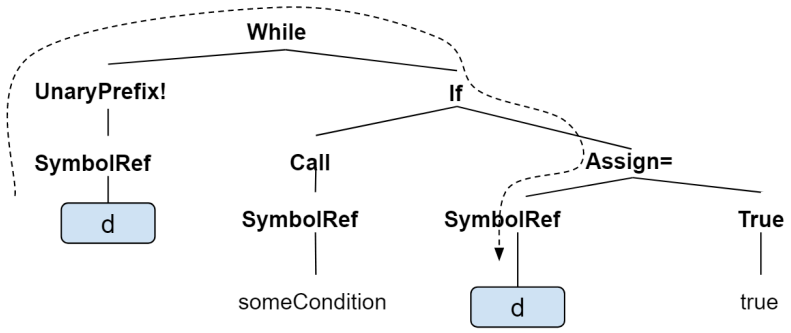
Абстрактное синтаксическое дерево (Abstract Syntax Tree, AST) – это представление исходного кода на данном языке программирования в виде дерева, каждый узел которого соответствует конструкции в исходном коде. Внутренние вершины помечены операторами данного языка, а листья – соответствующими операндами. Например, для кода на 1a AST представлено на 1b.

AST путь [11]– последовательность вершин AST, в которой каждая вершина, соединена ребром со следующей вершиной. В качестве начала и конца пути берутся листья дерева. Например, на 1b представлен путь, соединяющий вершины, помеченные идентификаторами *d*.

Представление кода в виде путей (path-based) [11] – множество AST путей, полученных из исходного кода.

```
while (!d) {
    if (someCondition()) {
        d = true;
    }
}
```

(a) JavaScript код



(b) Абстрактное синтаксическое дерево кода и AST путь

Рис. 1: Фрагмент кода на языке JavaScript и соответствующее ему AST с выделенным путем

2.2. Инструмент astminer

Astminer [1] [9] — это инструмент, позволяющий собирать датасеты из исходного кода программ. Он принимает код на языках программирования *Java*, *Python*, *JavaScript*, *C++* и извлекает представление кода в виде AST или path-based. А сам *Astminer* реализован на языке программирования *Kotlin*.

Рассмотрим основную функциональность *Astminer* и его основные части.

2.2.1. Промежуточные форматы представления данных

Поскольку *Astminer* обрабатывает синтаксические деревья, существует тип `Node` описывающий каждый узел синтаксического дерева. Деревья представляются как их корневой узел, поэтому в дальнейшем под деревом будет подразумеваться корневой узел этого дерева.

Результат разбора исходного кода представлен объектами класса `ParseResult`, которые хранят полученное синтаксическое дерево и путь к исходному файлу. Класс `ParseResult` представлен на Рис. 2.

При гранулярности "метод" *Astminer* работает с методами, которые представлены объектами типа `MethodInfo`, который изображен на Рис. 2. В `MethodInfo` хранится дерево, соответствующее объявлению метода, узел, который содержит имя метода, параметры метода в виде списка узлов синтаксического дерева и другая информация о методе, которая хранится в виде узлов.

Каждому дереву ставится в соответствие метка. Помеченные деревья представлены объектами класса `LabeledResult`, который содержит дерево, его метку и путь до исходного файла. Класс `LabeledResult` представлен на Рис. 2.

2.2.2. Разбор языков

Astminer поддерживает 3 вида парсеров: *ANTLR* [8], *GumTree* [4], *Fuzzy* [2], с помощью которых он извлекает AST из исходного кода. Для лю-

[1] Github: https://github.com/JetBrains-Research/astminer

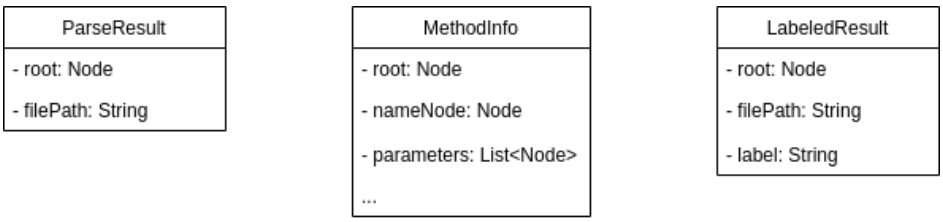


Рис. 2: Промежуточные форматы представления данных

бого языка программирования может быть произвольное количество видов парсеров, но они все реализуют единый интерфейс `Parser`, который изображен на Рис. 3

В интерфейсе `Parser` есть единственный метод `parseFile`, который принимает файл и возвращает `ParseResult`, который содержит разобранное синтаксическое дерево переданного файла. Для каждого типа парсера и языка программирования существует класс, реализующий интерфейс Parser, который играет роль адаптера к реальному парсеру, предоставляемому библиотеками *ANTLR*, *GumTree* и *Fuzzy*. Например, классы `PythonParser` и `JavaScriptParser` реализуют интерфейс `Parser` и при вызове метода `parseFile` запускают соответствующие *ANTLR* парсеры.

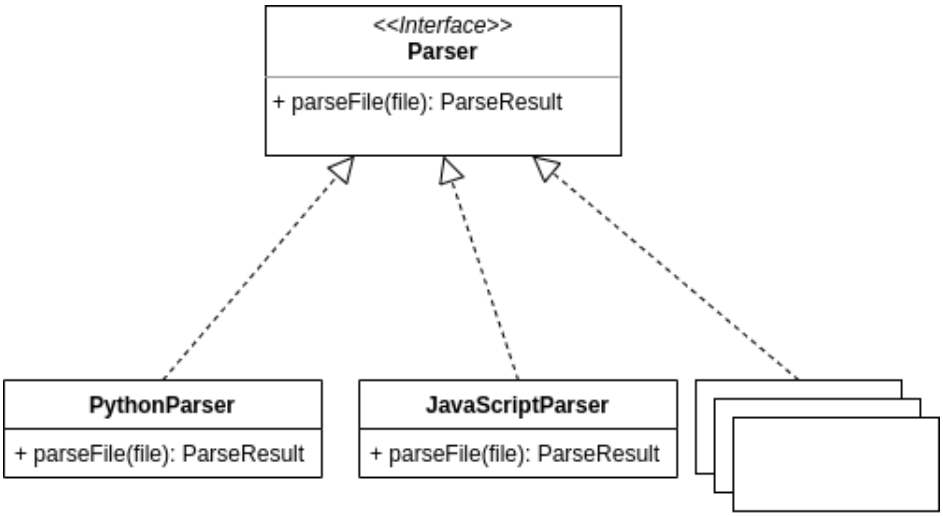


Рис. 3: Интерфейс `Parser` и реализующие его классы

`TreeMethodSplitter` – интерфейс, описывающий классы, которые позволяют собирать из AST файлов синтаксические деревья объявлений методов. Он имеет один метод, который получает синтаксическое дерево и извлекает из него поддеревья, соответствующие объявлениям

методов. Каждый парсер выдает AST своего вида, поэтому для каждого парсера существует класс, реализующий `TreeMethodSplitter`.

Astminer поддерживает два уровня гранулярности: уровень файлов и уровень функций. При файловом уровне гранулярности после парсинга дальнейшую обработку проходят AST файлов. При уровне гранулярности "функция" дальнейшую обработку проходят поддеревья, которые извлекаются с помощью `TreeMethodSplitter`.

2.2.3. Фильтрация

Далее из полученных деревьев остаются только те, которые удовлетворяют определенным условиям. Допустим пользователь может настроить, чтобы убирались деревья, превышающие определенный размер или соответствующие методом-конструкторам. Этот этап называется фильтрацией и выполняется фильтрами.

В *Astminer* фильтры реализованы только для уровня детализации "метод". Они представлены абстрактным классом `MethodFilterPredicate` и классами, которые его реализуют и содержат логику каждого фильтра. Фильтры поддерживают только синтаксические деревья, которые получаются после разбора парсером *GumTree*. Они проверяют есть ли определенные узлы в AST методов. Например, фильтр `ModifierFilterPredicate`, который проверяет есть ли у метода определенные модификаторы, ищет узлы, соответствующие модификаторам в синтаксическом дереве.

2.2.4. Извлечение меток

После фильтрации каждому полученному AST ставится в соответствие метка. Метка – это произвольная строка, которая сохраняется на диск вместе с данными. Например, дереву каждой функции можно ставить в соответствие имя этой функции или дереву файла можно ставить в соответствие название этого файла, чтобы можно было найти, откуда извлечены данные.

`LabelExtractor` – это интерфейс, имеющий один метод

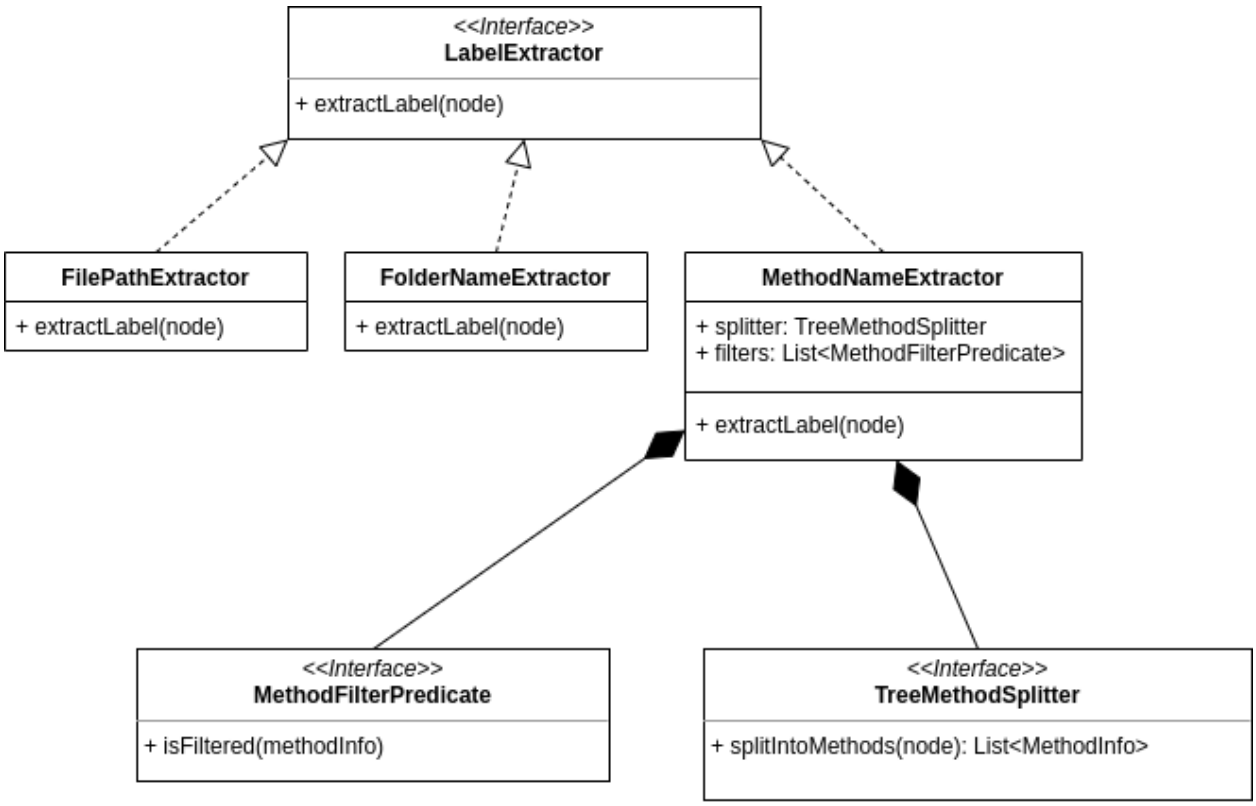


Рис. 4: Классы, реализующие `LabelExtractor`

`extractLabel`, который должен извлекать метку из синтаксического дерева. Для гранулярности "файл" реализованы 2 класса, извлекающие метки: первый класс `FilePathExtractor` ставит синтаксическому дереву в соответствие имя файла, второй `FolderNameExtractor` – имя папки, в которой лежит файл. Для уровня детализации "функция" интерфейс реализует класс `MethodNameExtractor`, который при помощи нужного `TreeMethodSplitter` находит поддеревья, соответствующие объявлениям методов, исключает ненужные пользователю методы при помощи фильтров (`MethodFilterPredicate`) и помечает каждый оставшийся метод его именем, которое он берет из терминального узла синтаксического дерева. Описанная архитектура изображена на Рис. 4.

2.2.5. Извлечение AST путей

С помощью *Astminer* можно извлекать пути из синтаксических деревьев, формируя представление кода в виде AST путей. У пользовате-

ля есть возможность установить ограничения, такие как максимальное число путей, максимальную ширину и длину каждого извлекаемого пути.

Извлечение путей из синтаксических деревьев выполняет класс `PathMiner`, у которого его метод `retrievePaths`, принимающий дерево и возвращающий список путей в этом дереве. Он используется, когда пользователь хочет получить представление кода в виде путей.

2.2.6. Сохранение на диск

AST. Запись синтаксических деревьев на диск производят классы, реализующие интерфейс `ASTStorage`. Классы `CsvAstStorage` и `DotAstStorage` позволяют сохранять деревья в *csv* и *dot* форматах. Они реализуют методы `store`, которые принимают дерево и его метку и записывают его в текстовый файл в выбранном формате. Примеры форматов изображены на 1 и 2.

AST пути. Для записи путей на диск используются классы `Code2VecPathStorage` и `CsvPathsStorage`, реализующие интерфейс `PathsStorage`. У каждого есть метод `store`, который сохраняет помеченный путь на диск, в заданном формате. Первый класс сохраняет пути в формате *code2vec*, а второй – *csv*.

Описанная архитектура изображена на Рис. 5.

2.2.7. Языковые фабрики

Языковые фабрики абстрагируют работу с парсерами и с классами, реализующими интерфейс `TreeMethodSplitter`. Для каждого вида парсера и языка программирования реализована языковая фабрика `HandlerFactory`. Она реализует метод `createHandler`, который получает файл и возвращает `LanguageHandler`, промежуточное представление файла.

Промежуточное представление файла `LanguageHandler` скрывает всю логику разбора файла и извлечения синтаксических деревьев методов. Оно имеет поле `parseResult`, которое хранит разобранный файл в

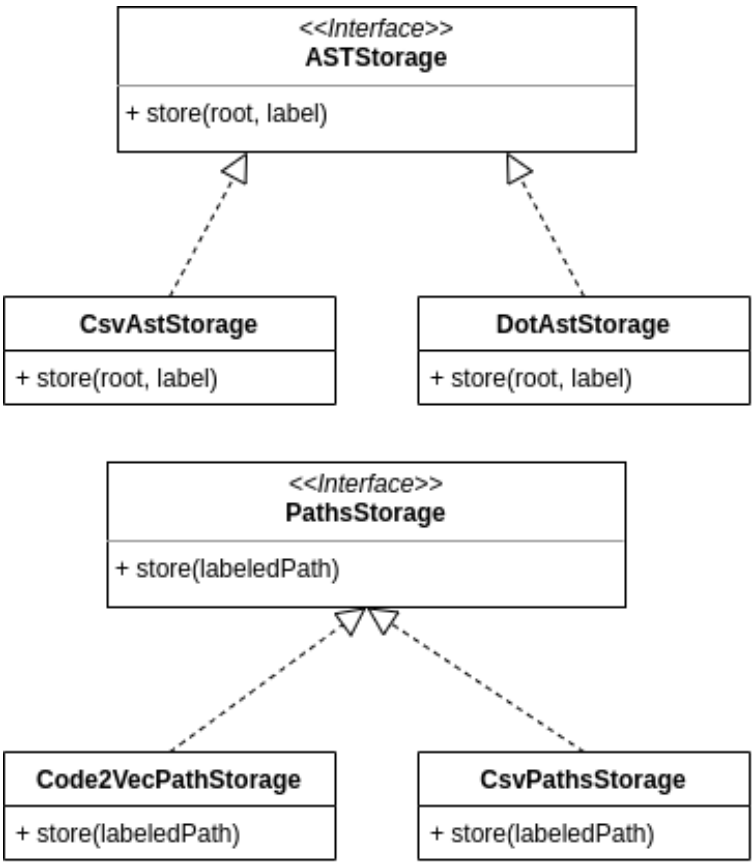


Рис. 5: `ASTStorage` и `PathsStorage`

виде `ParseResult`, и метод `splitIntoMethods`, который вызывает нужный `TreeMethodSplitter` и возвращает список поддеревьев, соответствующих объявлениям методов в файле.

2.2.8. Режимы работы *Astminer*

Каждый режим работы *Astminer* реализован классом, который отвечает за свою CLI команду. Каждый такой класс инициализирует классы нужных шагов обработки данных и запускает их в требуемом порядке. Так, например, класс, отвечающий за запуск извлечения путей, инициализирует парсер, `LabelExtractor`, `PathMiner` и хранилище (`PathsStorage`), соответствующее выбранному формату данных.

Основные шаги обработки данных совпадают во всех режимах работы, но поскольку все режимы работы реализованы независимыми классами, эти сходства выражаются в повторяющемся коде. Повторяющийся код двух реализаций режимов работы изображен на Рис. 7a 7b.

Листинг (1) Пример csv формата

```
id,ast
testMethodSplitting.java,1 1{1 2{2 3{}3 4{}1 5{4...
```

Листинг (2) Пример dot формата

```
digraph testMethodSplitting_java {
0 — {1 2};
1 — {3 4 5};
3 — {};
4 — {};
5 — {6 7 8 9 10 11 12 13 14 15 16};
...
```

Рис. 6: Форматы записи AST на диск

2.2.9. Недостатки архитектуры *Astminer*

Astminer запускается через интерфейс командной строки. Каждый этап обработки настраивается переданными аргументами. Один из примеров запуска изображен на 8. Поскольку в большом списке параметров искать и исправлять ошибки не просто, это затрудняет работу с *Astminer*. При добавлении новых видов обработки данных будут добавляться в этот список новые параметры, что сделает *Astminer* еще тяжелее для использования.

Также проблема повторяющегося кода, упомянутая ранее в 2.2.8, только усугубится, когда будет добавлен новый тип обработки данных. Добавление поддержки нового типа обработки данных в каждый режим работы реализуется одинаково, поэтому при добавлении во все режимы работы *Astminer* будет увеличено количество похожих строк кода.

2.3. Похожие инструменты

PSIMiner [7] – это похожий на *Astminer* инструмент, позволяющий извлекать данные *Java* файлов из их PSI деревьев, которые строит IntelliJ IDEA. Он позволяет сохранять PSI деревья в JSON формате и извлекать пути из них, сохраняя их в code2seq формате. Тут понятие пути

```
val outputDir = File(outputDirName)
/* ... */
for (extension in extensions) {
    val outputDirForLanguage = outputDir.resolve(extension)
    val storage = /* ... */
    val parser = getParser(extension, javaParser)
    parser.parseFiles(/* ... */) {
        val labeledParseResults = /* ... */
        storage.store(labeledParseResults)
    }
    storage.close()
}
```

(a) Фрагменты реализации класса `Code2VecExtractor`

```
val outputDir = File(outputDirName)
/* ... */
for (extension in extensions) {
    val outputDirForLanguage = outputDir.resolve(extension)
    val storage = /* ... */
    val parser = getParser(extension, javaParser)
    parser.parseFiles(/* ... */) {
        val labeledParseResults = /* ... */
        storage.store(labeledParseResults)
    }
    storage.close()
}
```

(b) Фрагменты реализации класса `ProjectParser`

Рис. 7: Фрагменты реализации двух классов, соответствующих двум режимам работы *Astminer*

совпадает с этим же понятием для astminer в пункте 2.1.

Обработка данных в PSIMiner делится на 4 шага: парсинг, фильтрация, извлечение метки и сохранение в хранилище. Шаги применяются в указанной последовательности и каждый шаг получает, результат действия предыдущего шага.

На этапе парсинга запускается среда разработки IntelliJ IDEA, которая строит PSI деревья. Это возможно, потому что PSIMiner представляет из себя плагин для IntelliJ IDEA. Из полученных PSI дере-

```
--lang java c cpp py
--java-parser antlr
--project  path/to/project
--output output
--maxL 10
--maxW 4
--maxContexts 1000
--maxTokens 100
--maxPaths 500
--granularity method
--hide-method-name
--split-tokens
--filter-modifiers private,protected
--filter-annotations override
--remove-constructors
```

Рис. 8: Аргументы запуска *Astminer*

вьев извлекаются поддеревья нужного типа: при уровне гранулярности ”класс” извлекаются деревья, соответствующие объявлениям классов, при уровне ”функция” – соответствующие объявлениям функций, аналогично при уровне детализации ”файл”.

На этапе фильтрации деревья проверяются предикатами и далее проходят только те, которые прошли проверку. Например, поддерживаются фильтры, проверяющие является ли метод абстрактным или конструктором.

Далее пройденные проверки деревья помечаются метками. На этом этапе из деревьев извлекаются строки, например, названия методов, которые в дальнейшем сохраняются вместе с деревьями.

На этапе сохранения в хранилище полученные помеченные деревья сохраняются на диск, в описанных форматах. Хранилища извлекают пути из PSI деревьев или сохранение деревья в JSON формате.

GumTree [4] – это фреймворк для работы с представлением исходного кода в виде деревьев и для подсчета отличий между такими деревьями. Он поддерживает 6 языков программирования: *C*, *Java*, *JavaScript*, *Python*, *R* и *Ruby*.

GumTree определяет независимый от языка вид AST, к которому должны быть приведены все AST, полученные после парсинга исходного кода на произвольном языке программирования. Все алгоритмы GumTree связанные с визуализацией или вычислением различий межу деревьями далее работают с этим каноническим видом AST. Это архитектурное решение абстрагирует основную логику GumTree от конкретных языков программирования и упрощает добавление новых языков.

Вывод. PSIMiner решает схожую задачу, что и *Astminer*: извлечение AST и path-based представления кода, и выделяет схожие интуитивные шаги обработки данных. Но PSIMiner поддерживает только один язык программирования и один вид синтаксических деревьев, и его архитектура не предусматривает добавление новых языков программирования. А архитектура GumTree, наоборот, упрощает добавление новых языков программирования. Поэтому архитектура *Astminer* была спроектирована на основе архитектурных идей GumTree и PSIMiner.

2.4. Формат и чтение конфигурации

Далее будут рассмотрены языки, на которых можно писать конфигурационные файлы, и библиотеки для их чтения.

Kotlinx Serialization [5] – библиотека на языке Kotlin, которая позволяет сериализовывать и десериализовывать объекты. С помощью ее можно читать конфигурационный файл и потом работать с ним как с экземпляром определенного класса.

Kotlin Serialization поддерживает полиморфную десериализацию, то есть она может десериализовывать интерфейсы, определяя конкретный тип объекта. Например, при изображенном на рис. 9a интерфейсе и двух его реализациях JSON объект на рис. 9b будет иметь тип `Dog` при полиморфной десериализации как объект типа `Animal`. То есть библиотека Kotlin Serialization по значению поля *type* определит конкретный тип объекта. Это упрощает добавление новых классов в существующие сериализуемые иерархии, так как достаточно реализовать сам класс, наследующий сериализуемый тип, а библиотека сама определит как ра-

ботать с новым классом.

```
interface Animal

@Serializable
class Dog : Animal

@Serializable
class Cat : Animal
```

(a) Сериализуемый интерфейс `Animal` и две реализации: `Dog` и `Cat`

```
{
    ''type'': ''Dog''
}
```

(b) JSON объект, который при полиморфной десериализации интерфейса `Animal` станет экземпляром класса `Dog`

Рис. 9: Пример использования полифорфной сериализации

Библиотека Kotlin Serialization была выбрана для чтения конфигурационого файла как официальная библиотека для сериализации на Kotlin. Выбор этой библиотеки ограничивает выбор языков, которые можно использовать для конфигурационного файла.

Язык YAML [2] – это текстовый формат для сериализации данных, ориентированный на удобство чтения. Он позволяет сериализовывать примитивные типы и списки из них. В число примитивных типов входят числа, строки и булевские переменные, а в число списков массивы и ассоциативные списки. Пример кода на YAML представлен на Рис. 10.

Язык JSON [3] – это читабельный язык для сериализации данных. Он похож на YAML, но отличается синтаксисом. В JSON каждый объект должен быть обособлен фигурными скобками, а каждый массив –

[2] Официальный сайт YAML – https://yaml.org/

[3] Официальный сайт JSON – https://www.json.org/json-en.html

```
name: John Smith
age: 33
friends:
  - Many Smith
  - Susan Williams
```

Рис. 10: Пример кода на YAML

```
{
    ''name'': ''John Smith'',
    ''age'': 33,
    ''friends'': [''Mary Smith'', ''Susan Williams'']
}
```

Рис. 11: Пример кода на JSON

квадратными. Также все строки должны быть обособлены двойными кавычками. Пример кода на JSON изображен на 11.

Язык TOML [4] – это текстовый формат для конфигурационных файлов, похожий на YAML. Основное его отличие от YAML заключается в том, как в нем выражается вложенность. Если в документе есть вложенный объект с именем *object*, то он должен быть помечен `[object]`. Если у *object* есть вложенный объект *a*, то он должен быть помечен `[object.a]`. Для того, чтобы обозначить массив объектов с именем *list*, то каждый объект массива должен быть выделен `[[list]]`.

Библиотека kaml. Для библиотеки kotlinx serialization существует библиотека kaml [5], добавляющая поддержку сериализации YAML.

2.5. Вывод

Была реализована архитектура, в которой сначала из различных видов AST извлекается общий вид данных, как в GumTree, который далее проходит этапы фильтрации, извлечения меток и сохранения на диск, как в PSIMiner. Причем каждый шаг обработки данных представлен отдельным классом, который реализует интерфейс соответствующего этапа обработки данных.

Конфигурационный файл, написанный на YAML получается более кратким, чем на JSON, так как JSON содержит обязательные фигурные скобки для объектов и квадратные для списков. TOML и YAML очень похожи, но запись массивов из объектов в YAML более краткая, так как в TOML для каждого элемента массива указывается полное

[4] Github: https://github.com/toml-lang/toml
[5] Github: https://github.com/charleskorn/kaml

имя массива `[[object.inner.list]]`. Поэтому в качестве языка для конфигурационного файла был взят YAML, а в качестве библиотек для сериализации YAML были взяты kotlinx serialization и kaml.

3. Реализация

3.1. Основные шаги обработки данных и требования к ним

При разных уровнях гранулярности извлекаются различные объекты: файлы и функции. Используются разные методы обработки данных, одни из которых работают с файлами, другие – с функциями. Итоговая архитектура обобщает процесс обработки данных для каждого уровня гранулярности.

3.1.1. Уровень гранулярности ”файл”

После разбора исходного кода деревья файлов представляются в виде объектов типа `ParseResult`. Каждый разобранный файл проходит фильтрацию, то есть остаются только те файлы, которые удовлетворяют определенному набору условий, который задал пользователь. После чего из файлов извлекаются метки, в результате чего получается помеченное дерево, представленное объектом типа `LabeledResult`. Далее синтаксические деревья файлов могут быть записаны на диск в 2 форматах, или из них могут быть извлечены пути, которые потом будут записаны на диск. Если извлечение путей и сохранение синтаксических деревьев на диск описать одним интерфейсом, то получается такая последовательность шагов изображенная на Рис. 12.

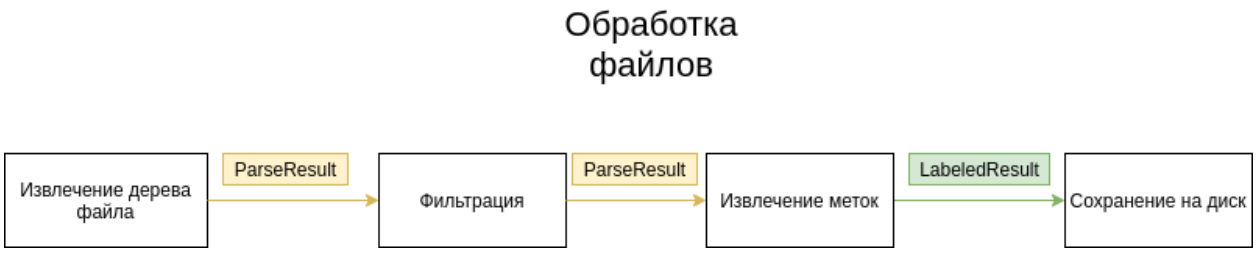


Рис. 12: Шаги обработки данных при уровне гранулярности ”файл”

3.1.2. Уровень гранулярности ”функция”

Извлеченные из файлов функции проходят схожие шаги обработки: они сначала проверяются фильтрами, а потом из их поддеревьев извлекаются метки. После извлечения меток функции представляются объектами типов `LabeledResult`, как в предыдущем пункте, что позволяет использовать те же реализации последнего этапа сохранения на диск, что и при обработке файлов. Полученная последовательность шагов изображена на Рис. 13.

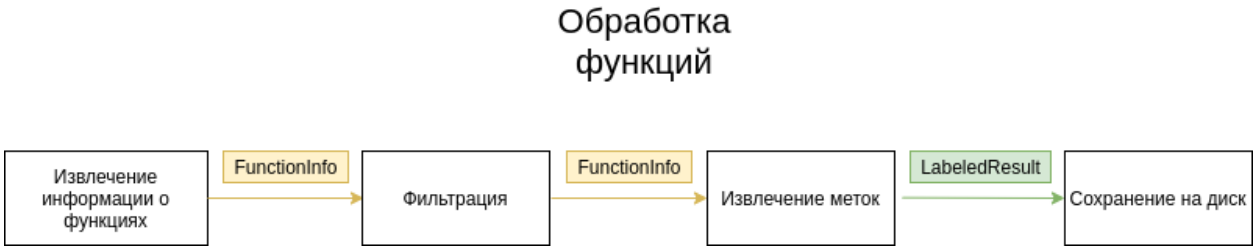


Рис. 13: Шаги обработки данных при уровне гранулярности ”функция”

3.1.3. Обобщение процесса обработки данных

В обоих случаях обработка состоит из 4 шагов: извлечение объектов, фильтрация, извлечение меток из объектов и сохранение их на диск в произвольном формате. В результате извлечения объектов при разных уровнях гранулярности получаются разные извлеченные сущности: файлы или функции. Обобщая оба случая, получаем что сначала из исходного кода извлекаются произвольные сущности типа `T`, которые потом проходят фильтрацию. После извлечения меток получается дерево с меткой, представленное объектом типа `LabeledResult`, который потом сохраняется на диск в произвольном формате. Причем этап извлечения данных из исходного кода должен извлекать всю релевантную для фильтров и извлечения метки информацию, чтобы эти этапы не взаимодействовали с синтаксическими деревьями и были независимы от разбираемых языков. Полученная последовательность шагов изображена на Рис. 15.

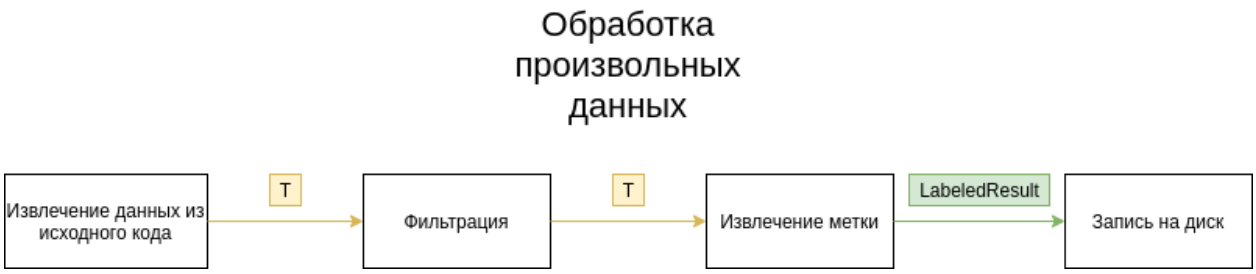


Рис. 14: Общий процесс обработки данных

3.2. Реализация архитектуры

Спроектированная мной архитектура явно разграничивает этапы обработки файлов и функций

3.2.1. Представление разобранного файла и функций

Разобранный файл представляется классом `ParseResult`, который был описан в обзоре.

Функция, извлеченная из файла описывается интерфейсом `FunctionInfo`. В этом интерфейсе есть все поля, которые могут потребоваться дальше для фильтрации или извлечения меток из функций: корень дерева объявления функции *root*, имя файла, из которого взята функция *filePath*, имя функции, список модификаторов функции и аннотаций и т.д. Для каждого парсера и вида синтаксического дерева есть реализация интерфейса `FunctionInfo`, которая хранит информацию, извлеченную из конкретного синтаксического дерева. За счет того, что в `FunctionInfo` уже есть вся найденная нужная информация, такие шаги обработки данных как фильтрация и извлечение метки не взаимодействуют с синтаксическим деревом.

3.2.2. Фильтры

Для явного деления на два типа фильтров были созданы два интерфейса: `FileFilter` и `FunctionFilter`. Классы реализующие `FileFilter` проверяют должен ли разобранный файл записаться на диск, они имеют метод `validate`, который принимает `ParseResult` и возвращает `Boolean`. Классы реализующие `FunctionFilter` проверяют функции,

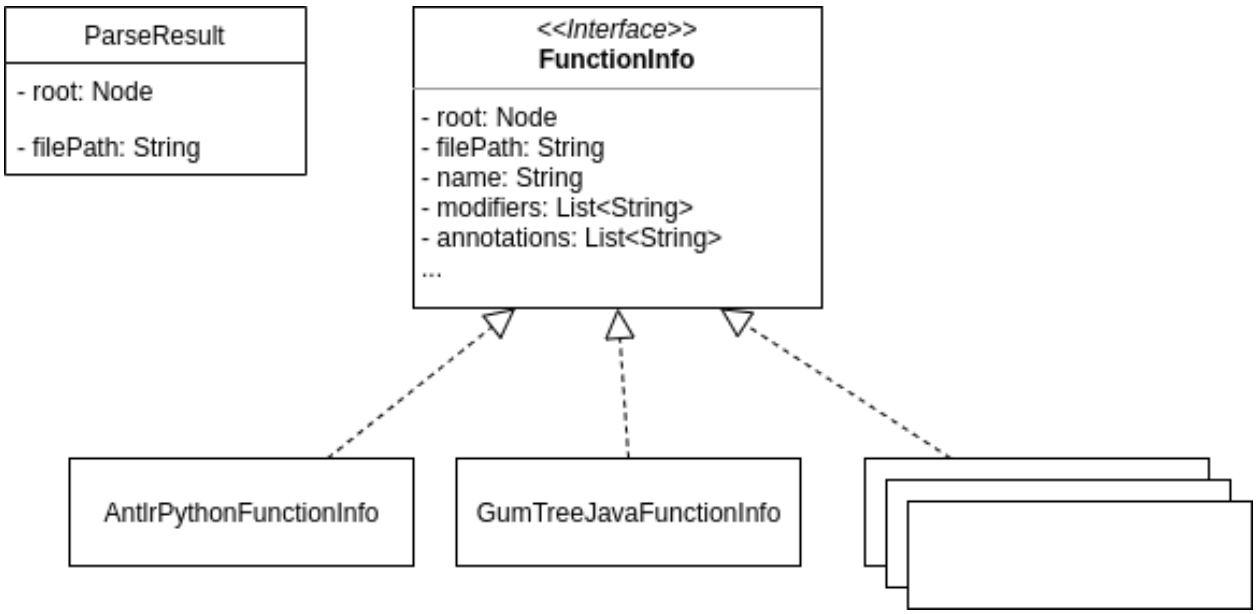


Рис. 15: `FunctionInfo` и `ParseResult`

они имеют метод `validate`, который принимает `FunctionInfo` и возвращает `Boolean`.

Существуют фильтры, которые могут работать и с файлами, и с функциями. Такие фильтры должны реализовать оба интерфейса. Сейчас существует два таких фильтра: `TreeSizeFilter`, который проверяет количество узлов в деревьях, и `WordsNumberFilter`, который проверяет количество слов в токенах деревьев.

Архитектура, реализующая фильтры представлена на Рис. 16.

3.2.3. Извлечение меток

Объекты, извлекающие метки, представлены двумя интерфейсами: `FileLabelExtractor` и `FunctionLabelExtractor`. Первый интерфейс предназначен для извлечение меток из файлов, он имеет один метод `process`, который принимает разобранный файл (`ParseResult`) и возвращает дерево с меткой (`LabeledResult`). Второй интерфейс предназначен для извлечения меток из функций, он имеет аналогичный метод `process`, который принимает `FunctionInfo` и также возвращает помеченное дерево. `FilePathExtractor` и `FolderNameExtractor` из архитектуры описанной ранее теперь реализуют интерфейс `FileLabelExtractor`. Описанная архитектура изображена на Рис. 17

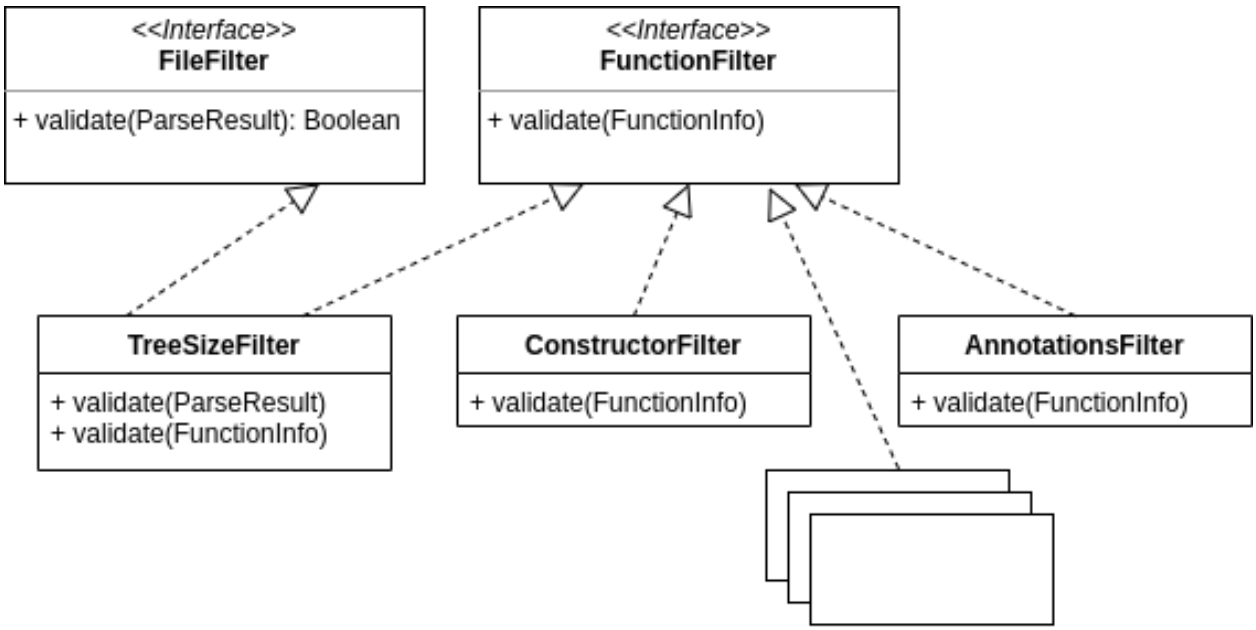


Рис. 16: Архитектура фильтров

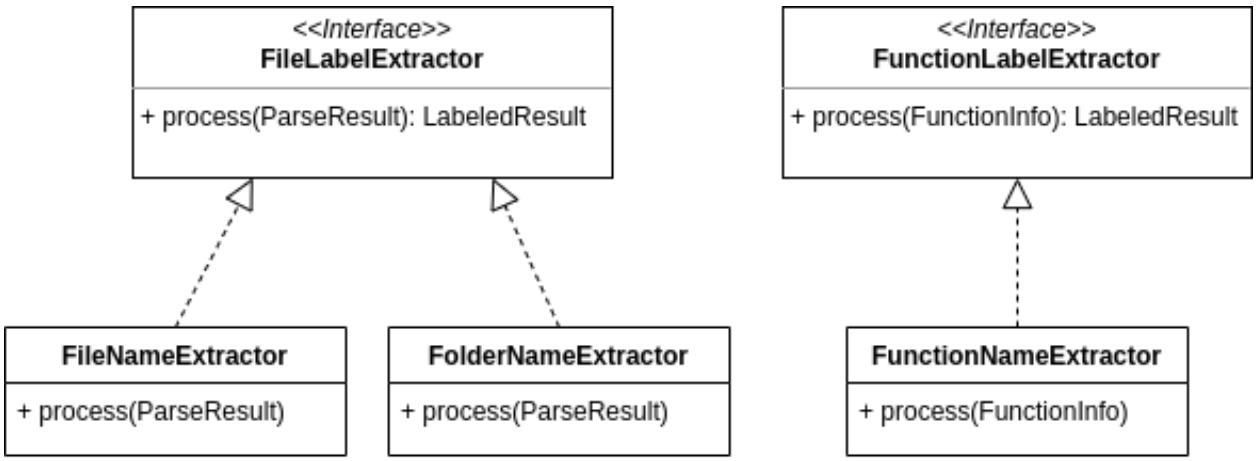


Рис. 17: Архитектура, реализующая извлечение меток

3.2.4. Хранилища

Хранилище отвечает за формат данных, который должен быть записан на диск. Все хранилища реализуют интерфейс `Storage`, который имеет метод `store`, получающий помеченное дерево. В отличие от старой архитектуры, теперь извлечение путей выполняется хранилищем `Code2VecPathStorage`, которое хранит объект класса `PathMiner`. Это позволяет абстрагировать хранилища от предыдущих шагов обработки данных, что позволяет использовать любое хранилище с любыми шагами обработки. Интерфейс `Storage` и реализующие его классы изоб-

ражены на 18.

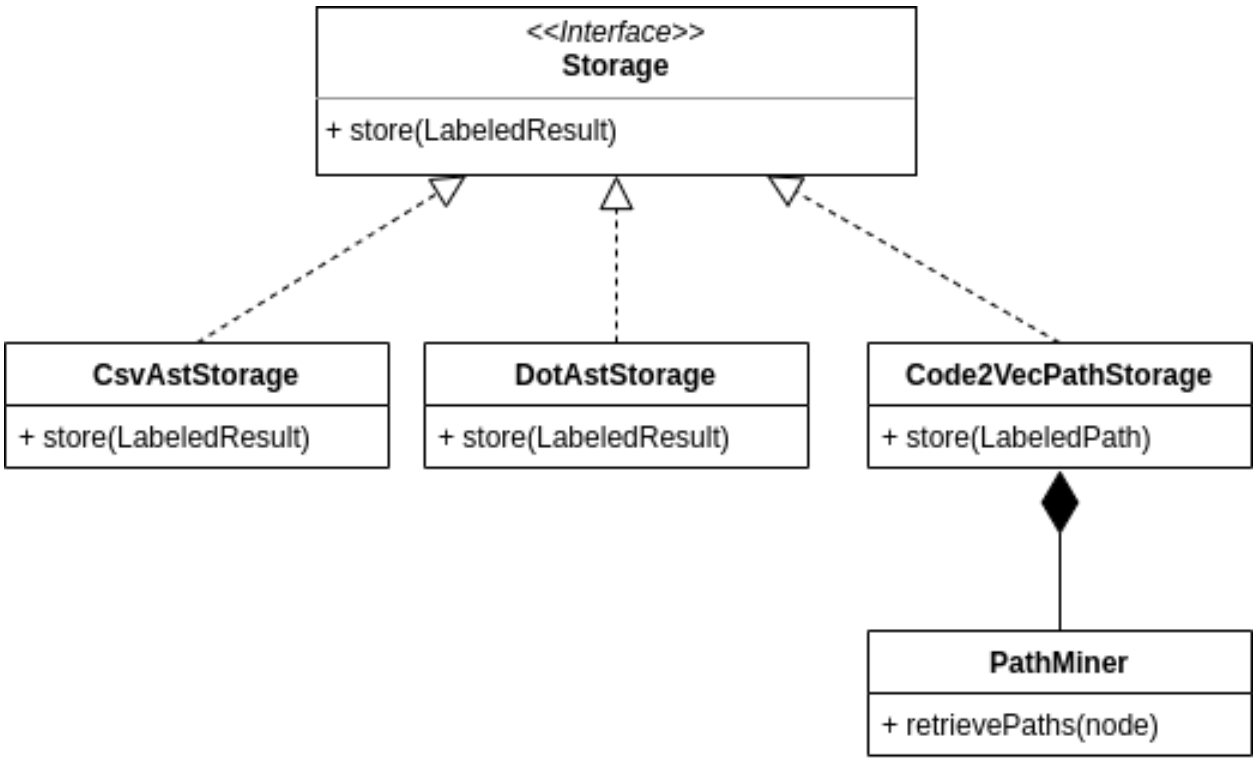


Рис. 18: Архитектура хранилищ

3.2.5. `PipelineBranch`

`PipelineBranch` – это интерфейс, объединяющий в себя фильтрацию и извлечение меток. Фильтры и извлечение меток реализовали разные интерфейсы в зависимости от того, для какого уровня гранулярности они предназначены. `PipelineBranch` сделан, чтобы объединить все виды извлечения меток и фильтрации под одним интерфейсом. Интерфейс `PipelineBranch` имеет метод `process`, который принимает разобранный файл (`LanguageHandler`) и возвращает список помеченных деревьев. Существует две реализации этого интерфейса: `FilePipelineBranch` и `FunctionPipelineBranch`. `FilePipelineBranch` применяется при уровне детализации "файл", он содержит фильтры, работающие с файлами (`FileFilter`), и `FileLabelExtractor`. Аналогично устроен класс `FunctionPipelineBranch`, который применяется при уровне гранулярности "функция". Описанная архитектура изображена на Рис. 19.

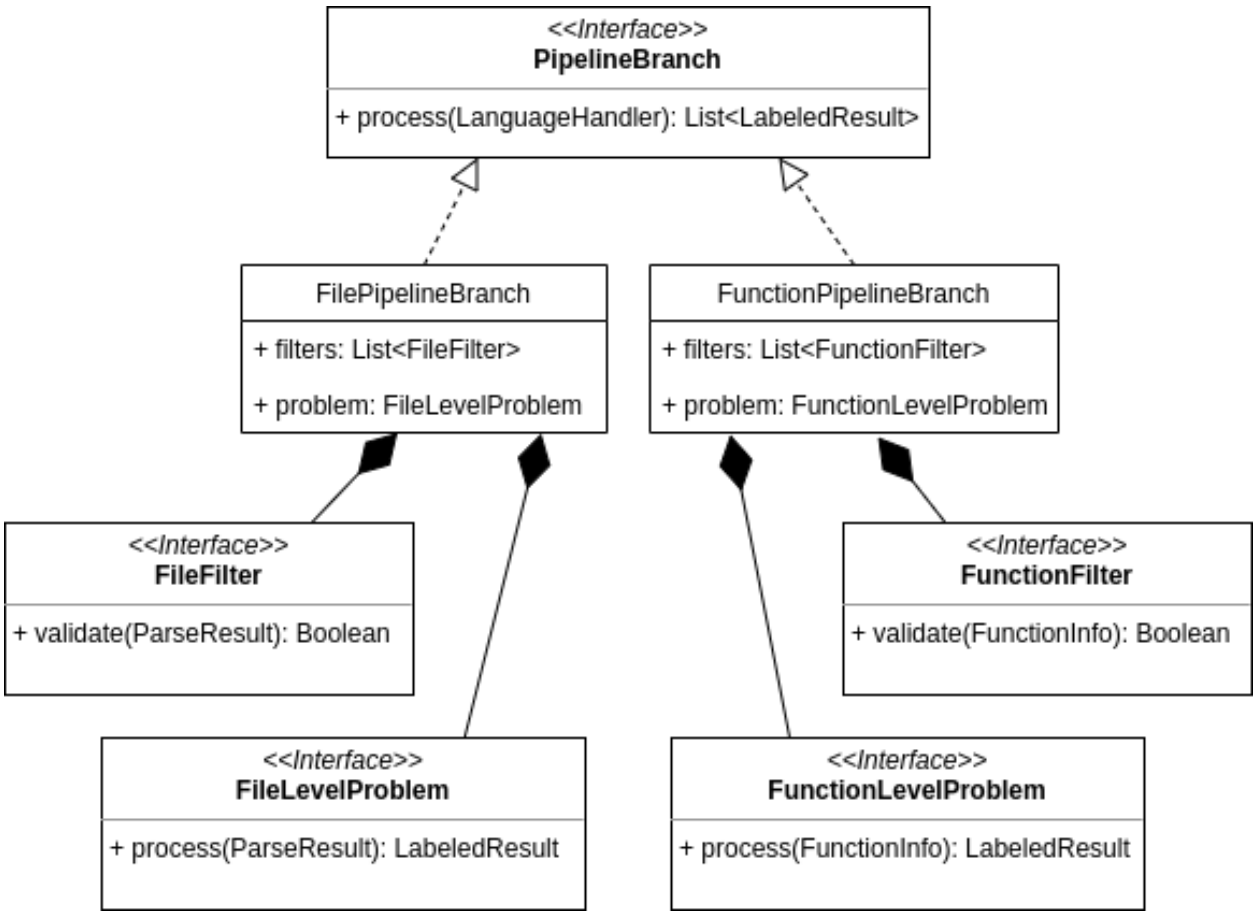


Рис. 19: `PipelineBranch`

3.2.6. `Pipeline`

Класс `Pipeline`, изображенный на Рис. 20, нужен для запуска работы всего инструмента. Он содержит нужные `HandlerFactory` для разбора требуемых языков программирования, `PipelineBranch` для фильтрации полученных синтаксических деревьев и извлечения из них меток, а также `Storage` для записи помеченных деревьев на диск. Все три компонента не зависят друг от друга, что значит, что в качестве них можно использовать любую реализацию их интерфейса.

3.2.7. Вид конфигурационного файла

Был выбран вид конфигурационного файла, в котором явно указано, из каких шагов состоит обработка данных. Благодаря этому исследователь, пользуясь инструментом *Astminer*, понимает аспекты внутренней архитектуры, что сэкономит время исследователю, если он захочет рас-

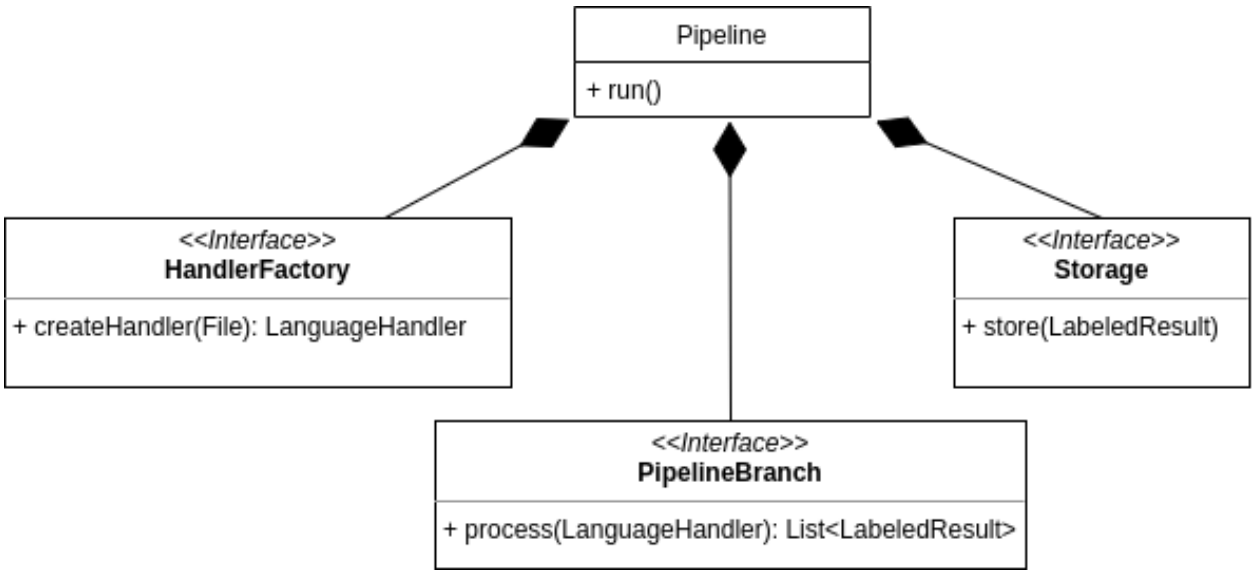


Рис. 20: `Pipeline`

ширить *Astminer*, добавив новую функциональность. У каждого шага обработки есть несколько реализаций, каждая из которых может иметь настраиваемые параметры.

Файл с настройками пользователя имеет поля *inputDir*, *outputDir* и 4 секции: *parser*, *filters*, *problem*, *storage*. Поля *inputDir* и *outputDir* содержат путь до папки с исходным кодом и папки, где пользователь хочет видеть результат работы *Astminer*.

В секции *parser* указывается тип парсера и языки программирования, которые этот парсер должен разбирать. Если пользователь выберет несколько языков программирования, то *Astminer* будет использовать по парсеру на каждый язык. В примере 21 будет использовано два *ANTLR* парсера для языков программирования *Java* и *JavaScript*.

Секция *filters* предназначена для настройки фильтрации результатов парсинга. В ней указывается список фильтров и их параметров, которые будут применены. Поддерживаются все фильтры описанные выше, которые названы *by tree size*, *by modifiers*, *by annotations*, *no constructors*, *by function name length*, *by words number*. В примере 21 будут отброшены все функции, длина имени которых превышает 10 символов и которые в синтаксическом дереве своего объявления содержат токены, состоящие больше чем из 100 слов.

Что сделует использовать в качестве меток пользователь может ука-

зать в секции *labelExtractor*. По выбранному типу метки будет определен уровень гранулярности, которому должны соответствовать переданные фильтры. Поддерживаются все описанные ваше типы извлечения меток: *file name*, *folder name*, *function name*. В примере 21 будет выбран уровень гранулярности "функция" и в качестве меток будут использоваться имена функций.

В последней секции *storage* пользователь должен указать формат выходных данных. Поддерживаются форматы для синтаксических деревьев *CsvAST*, *DotAST*, а также формат для представления кода в виде путей *Code2vec* [3]. В примере ниже будет выбран *csv* формат для синтаксических деревьев.

```
inputDir: 'input'
outputDir: 'output'

parser:
  name: 'antlr'
  extensions: ['java', 'js']

filters:
  - name: 'by function name length'
    maxWordsNumber: 10
  - name: 'by words number'
    maxTokenWordsNumber: 100

labelExtractor:
  name: 'function name'

storage:
  name: 'CsvAST'
```

Рис. 21: Пример конфигурационного файла

3.3. Решение, запускающее шаги обработки данных по настройкам пользователя

3.3.1. Конфигурационные классы

Конфигурационный файл представлен классом `PipelineConfig`, который имеет поля *inputDir*, *outputDir*, *parser*, *filters*, *problem*, *storage*. Этот класс полностью описывает структуру YAML конфигурационного файла, который имеет такие же поля. На примере 21 поля *inputDir* и *outputDir* после десериализации будут равны строкам *"input"* и *"output"*.

Поле *parser* хранит объект класса `ParserConfig`, который описывает настройку парсеров. Он имеет поля *name* и *extensions*, которые хранят тип выбранного парсера и список языков программирования. На примере выше после десериализации поле *name* будет иметь строковое значение *"antlr"*, а *extensions* будет хранить список из строк *"java"* и *"js"*.

Поле *filters* хранит список объектов типа `FilterConfig`, зарегистрированного для полиморфной сериализации. Каждый класс, реализующий интерфейс `FilterConfig`, является конфигурационным классом для определенного типа фильтра. У каждого класса, конфигурирующего фильтр, есть свое уникальное название, по которому осуществляющая сериализацию библиотека Kotlinx Serialization определяет конкретный тип сериализованного объекта. На примере 21 первый фильтр с полем *name* равным *"by function name length"* будет десериализован как объект класса `FunctionNameWordsNumberFilterConfig` с полем `maxWordsNumber` равным *10*. Второй фильтр с полем *name* равным *"by words number"* будет десериализован как объект класса `WordsNumberFilterConfig`, имеющий поле `maxTokenWordsNumber` равное *100*.

Поле *labelExtractor* хранит объект типа `LabelExtractorConfig`. Аналогично фильтрам, для каждого из 3 классов извлекателей меток существует конфигурационный класс, реализующий интерфейс `LabelExtractorConfig` и настраивающий свой класс, извлекающий метки. По полю `name`, которое индивидуально для каждого класса, реали-

зующего `ProblemConfig` сериализатор понимает конкретный тип объекта. На примере 21 объект проблемы будет десериализован и будет экземпляром класса `FunctionNameExtractorConfig`, который настраивает проблему `FunctionNameExtractor`.

Аналогично проблемам, поле *storage* имеет тип `StorageConfig`. Для каждого из 3 реализующих интерфейс `Storage` классов существует свой конфигурационный класс, реализующий `StorageConfig` и имеющий свое уникальное название. В примере выше объект *storage* будет десериализован как объект класса `CsvAstStorageConfig`.

Описанная архитектура изображена на 22

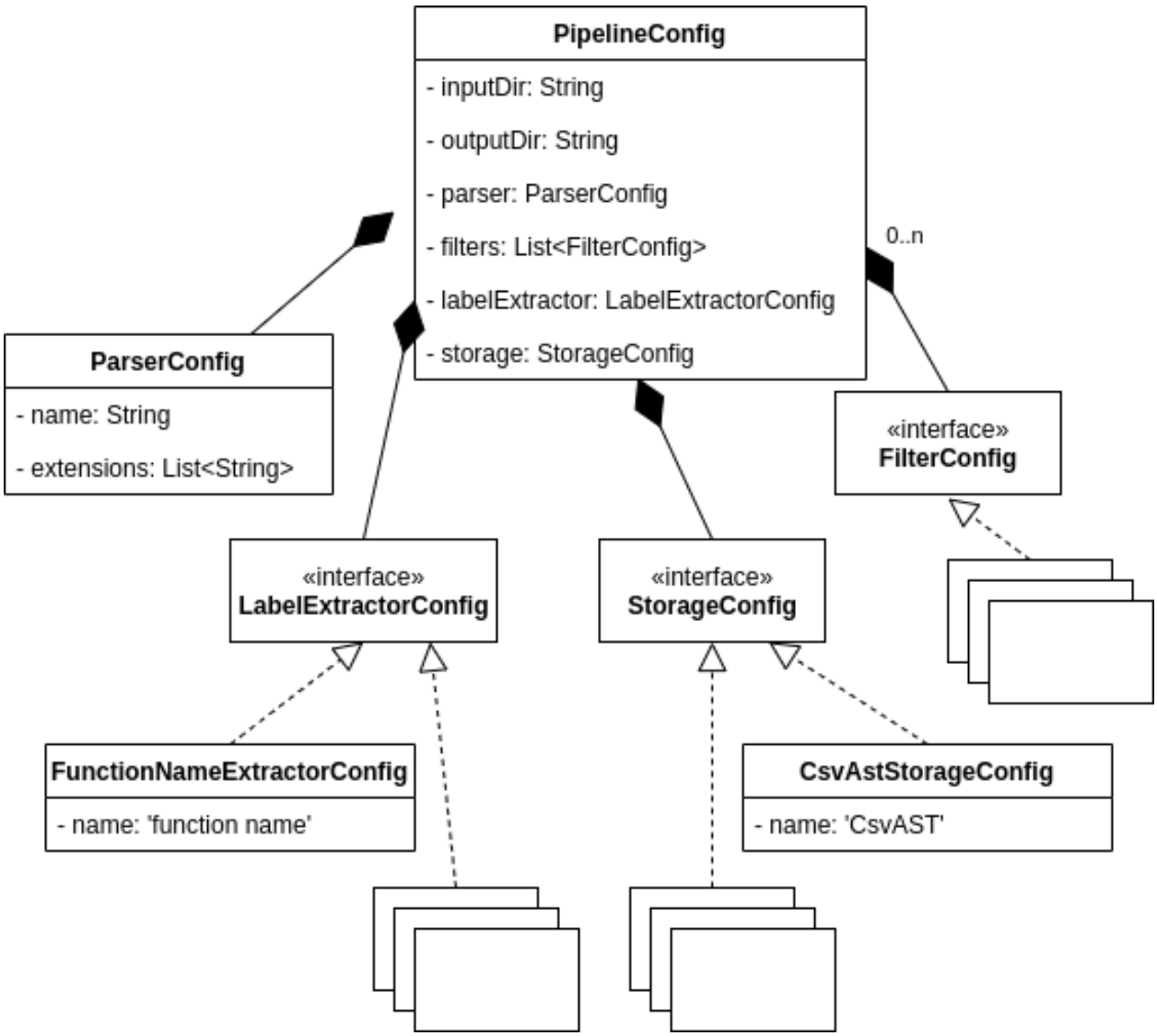


Рис. 22: Классы, описывающие конфигурационный файл

3.3.2. Настройка Pipeline по полученным конфигурациям

Класс Pipeline имеет конструктор, принимающий конфигурацию PipelineConfig, который в зависимости от переданных настроек выбирает требуемые шаги обработки данных. Он получает нужные фабрики HanlerFactory, соответствующие переданному типу парсера и языкам программирования в ParserConfig. Конструктор также инициализирует нужный объект типа Storage с помощью кода, приведенного на 23.

```
when (this) {
    is CsvAstStorageConfig -> CsvAstStorage(...)
    is DotAstStorageConfig -> DotAstStorage(...)
    is Code2VecPathStorageConfig -> Code2VecPathStorage(...)
}
```

Рис. 23: Инициализация хранилища

Далее по уровню гранулярности выбранной проблемы конструктор инициализирует нужный PipelineBranch, передавая ему весь конфигурационный объект PipelineConfig: при файловом уровне гранулярности инициализируется объект типа FilePipelineBranch, при функциональном уровне – FunctionPipelineBranch.

Конструктор каждого класса, реализующего PipelineBranch, продолжает процесс инициализации: он создает объекты-фильтры и объект-проблему. Когда конструктор класса FilePipelineBranch получает конфигурационный объект PipelineConfig, он создает все требуемые фильтры, которые указал пользователь. Если хоть один фильтр предназначен для проблемы с функциональным уровнем детализации, конструктор FilePipelineBranch бросает исключение, которое будет выведено пользователю. Также FilePipelineBranch инициализурует класс, извлекающий метки, который описан в конфигурационном объекте. Аналогично, поступает FunctionPipelineConfig.

4. Эксперимент

4.1. Условия эксперимента

В рамках эксперимента была добавлена новая функциональность: возможность сохранять AST на диск в JSON формате. Целью эксперимента является проверка простоты добавления новых типов обработки данных.

4.2. Исследовательские вопросы

- Реализовать архитектуру, позволяющую проще добавлять новые виды обработки данных
- Реализовать удобный пользовательский интерфейс, в который можно проще добавлять новые параметры

4.3. Результаты

4.3.1. Добавление реализации JSON хранилища

Чтобы добавить новое хранилище, надо создать новый класс, который реализует интерфейс `Storage` и содержит всю логику записи на диск AST в JSON формате. Был создан класс JsonAstStorage, изображенный на Рис. 24

```
class JsonAstStorage(val outputDirectoryPath: String) : Storage {
    override fun store(labeledResult: LabeledResult<out Node>) {
        /* ... */
    }
}
```

Рис. 24: Класс `JsonAstStorage`

4.3.2. Создание конфигурационного класса для нового хранилища

Далее, чтобы новое хранилище можно было настраивать из конфигурации, необходимо создать конфигурационный класс для нового хранилища, который знает как его инициализировать. Так как наше хранилище не имеет настраиваемых параметров, то конструктор конфигурационного класса также не имеет параметров. Новый конфигурационный класс реализует метод `createStorage`, который инициализирует хранилище `JsonAstStorage`. Строчка `@SerialName(''JsonAST'')` указывает, что в конфигурационном файле хранилище будет иметь имя *JsonAst*. Конфигурационный класс представлен на Рис. 25.

```
@Serializable
@SerialName(''JsonAST'')
class JsonAstStorageConfig : StorageConfig() {
    override fun createStorage(outputDirectoryPath: String) =
        JsonAstStorage(outputDirectoryPath)
}
```

Рис. 25: Конфигурационный класс `JsonAstStorageConfig`

4.3.3. Запуск *Astminer* с новым форматом сохранения данных

После реализации класса `JsonAstStorage` и конфигурационного класса `JsonAstStorageConfig` новое хранилище можно использовать. Достаточно указать в конфигурационном файле *JsonAST* как имя хранилища. Пример конфигурационного файла, который запустит *Astminer* и сохранит данные на диск в новом формате приведет на Рис. 26.

4.4. Обсуждение результатов

При добавлении нового формата сохранения данных не были скопированы существующие строки кода. А в старой архитектуре пришлось бы изменять каждый из 4 классов, реализующих режимы работы

```
inputDir: 'input'
outputDir: 'output'

parser:
  name: 'antlr'
  extensions: ['java', 'js']

labelExtractor:
  name: 'file name'

storage:
  name: 'JsonAST'
```

Рис. 26: Пример конфигурационного файла для использования хранилища `JsonAstStorage`

Astminer, добавляя настраиваемые параметры этого формата и условия, инициализирующие новый формат данных. Поскольку в каждый режим работы новый формат обработки данных добавлялся бы одинаково, увеличилось бы количество скопированных строк кода. Следовательно, процесс добавления новых типов обработки данных в *Astminer* и их настраиваемых параметров в конфигурацию запуска был упрощен.

5. Заключение

Было разработано и внедрено end-to-end решение для настраиваемого запуска шагов обработки данных в приложении *Astminer*.

С ходе выполнения цели, были выполнены задачи:

- Изучена существующая архитектура и функциональность инструмента;
- Выделены основные шаги обработки данных и требования к ним;
- Спроектирована и реализовано решение, запускающее инструмент по настройкам пользователя;
- Было внедрено и апробировано реализованное решение;

Изменение пользовательского интерфейса должно быть отражено в документации, которая находится в процессе написания и требует взаимодействия всех команды разработчиков. Поэтому все изменения пользовательского интерфейса еще не влиты в главную ветку, а находятся в ветке *pipeline*: https://github.com/JetBrains-Research/astminer/tree/pipeline.

Список литературы

[1] The 2020 State of the Octoverse : Rep. : Github Inc., 2020. — URL: https://octoverse.github.com/.

[2] Code Property Graph. — 2021. — URL: https://github.com/ShiftLeftSecurity/codepropertygraph/.

[3] Code2Vec: Learning Distributed Representations of Code / Uri Alon, Meital Zilberstein, Omer Levy, Eran Yahav // Proc. ACM Program. Lang. — 2019. — . — Vol. 3, no. POPL. — P. 40:1–40:29. — URL: http://doi.acm.org/10.1145/3290353.

[4] Fine-grained and accurate source code differencing / Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc et al. // ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014. — 2014. — P. 313–324. — URL: http://doi.acm.org/10.1145/2642937.2642982.

[5] Kotlinx Serialization. — 2021. — URL: https://github.com/Kotlin/kotlinx.serialization.

[6] Miltiadis Allamanis Marc Brockschmidt Mahmoud Khademi. Learning to Represent Programs with Graphs. — 2018. — URL: https://openreview.net/pdf?id=BJOFETxR-.

[7] PSIMiner. — 2021. — URL: https://github.com/JetBrains-Research/psiminer.

[8] Parr Terence. ANTLR. — 2021. — URL: https://www.antlr.org/.

[9] PathMiner: a library for mining of path-based representations of code / Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, Alberto Bacchelli // Proceedings of the 16th International Conference on Mining Software Repositories / IEEE Press. — 2019. — P. 13–17.

[10] Shafiq Saad. Machine Learning for Software Engineering: A Systematic Mapping. — 2020. — URL: https://arxiv.org/abs/2005.13299.

[11] Uri Alon Meital Zilberstein Omer Levy Eran Yahav. A General Path-Based Representation for Predicting Program Properties. — 2018.

[12] Visual Studio IntelliCode. — 2021. — URL: https://visualstudio.microsoft.com/services/intellicode/.