Санкт-Петербургский государственный университет

Божнюк Александр Сергеевич

Выпускная квалификационная работа

Система модификации структуры исходного кода для интегрированных сред разработки

Уровень образования: бакалавриат

Направление *09.03.04 «Программная инженерия»*

Основная образовательная программа *СВ.5080.2019 «Программная инженерия»*

Научный руководитель:
д.т.н., профессор Д. В. Кознов

Консультанты:
инженер ключевых проектов ООО «МПГ Айти Солюшнз» А. А. Захаров,
к.т.н., доцент Ю. В. Литвинов

Рецензент:
ведущий инженер ключевых проектов ООО «Техкомпания Хуавэй» Н. В. Тропин

Санкт-Петербург
2023

Saint Petersburg State University

Alexander Bozhnyuk

Bachelor's Thesis

Writable PSI Generator for a Multi-Language IDE Platform

Education level: bachelor

Speciality *09.03.04 "Software Engineering"*

Programme *CB.5080.2019 "Software Engineering"*

Scientific supervisor:
Sc.D, prof. D.V. Koznov

Consultants:
Senior engineer at "MPG IT Solutions LLC" A.A. Zakharov,
C.Sc, docent Y.V. Litvinov

Reviewer:
Principal engineer at "Huawei Technologies CO.LTD" N.V. Tropin

Saint Petersburg
2023

Оглавление

Введение

Интегрированная среда разработки (Integrated Development Environment, IDE) является неотъемлемым инструментом любого программиста. Пожалуй, самыми известными средами разработки являются IntelliJ IDEA (JetBrains) [1] и Visual Studio (Microsoft) [2], которые предлагают большое количество сервисов для помощи в создании качественного программного обеспечения.

Одной из наиболее важных задач IDE является возможность быстро и корректно дополнять и исправлять исходный код программы. Для этого среда разработки предоставляет различные виды рефакторинга (refactorings) [3], а также сервисы быстрых исправлений (quick fixes) [4]. Рефакторинг делает возможным перестроить код с сохранением его семантики, например, переименовать класс, метод и атрибут (rename [5]), извлечь выделенный код в метод (extract method [6]) и др. Быстрое исправление позволяет по желанию пользователя устранить недостаток некоторого фрагмента кода, например, упростить условный оператор (simplify if statement).

Данные сервисы работают со структурой программы, анализируя и перестраивая её. Традиционно, внутренним представлением программы является абстрактное синтаксическое дерево (Abstract Syntax Tree, AST [7]), которое строится в результате синтаксического разбора программы (parsing). Но кроме этого данным сервисам требуется дополнительная семантическая информация (например, по вхождению метода или атрибута определить его описание), и эту информацию также удобно было бы сохранять в синтаксическом дереве. Поэтому IDE на основе AST строит другое дерево, которое дает внешним клиентам (сервисам IDE) доступ к такой информации о программе. В IntelliJ IDEA это дерево называется Program Structure Interface (PSI) [8, 9]. Данное название используется в текущей выпускной квалификационной работе. Таким образом дерево PSI хранит дополнительную семантическую информацию о программе и предоставляет клиентам многофункциональный программный интерфейс для доступа к ней (API), а дерево

AST является деталью реализации.

PSI-дерево является типизированным — каждый его узел имеет свой тип сообразно синтаксической конструкции языка, которую этот узел представляет, в рамках типовой системы языка программирования, на котором это дерево создается. Например, в контексте языка Java каждый PSI-узел определяется своим классом (`PsiFunction`, `PsiClass` и т. д.) [10].

Сервисы IDE, после манипуляций с PSI, переносят изменения программы в её исходный текст с тем, чтобы эти изменения стали видны пользователю. Для этого на основе изменений дерева формируются текстовые изменения, которые затем применяются к исходному тексту.

Как было указано выше, PSI-дерево является типизированным. Это удобно для разработчиков IDE, однако типы в таком дереве надо описать. Если IDE, например, разрабатывается на языке Java, то для доступа к соответствующим узлам дерева потребуется создать большое количество интерфейсов и классов [10]. Создание такого кода является очень трудоёмким из-за большого количества типов и чревато ошибками. По этой причине целесообразно использовать генеративный подход, т.е. на основе заранее созданной спецификации типов синтаксических конструкций языка программирования автоматически генерировать классы и интерфейсы доступа к узлам PSI-дерева.

В рамках Saint-Petersburg Research Center разрабатывается мультиязыковая платформа SRC IDE, предназначенная для разработки IDE различных языков программирования. В частности, на основе этой платформы в настоящий момент создаются IDE для Java и Python.

Для этой платформы SRC IDE важно иметь единую подсистему работы со структурой исходного кода программы. Для каждой конкретной IDE требуется свое PSI-дерево и средства для манипуляции с ним, а также компонента для отображения изменений в документе. Но принципы генерации интерфейсов доступа к PSI-дереву являются общими и могут быть реализованы в рамках платформы и повторно использованы в различных IDE.

1. Постановка задачи

Целью данной выпускной квалификационной работы является разработка подсистемы модификации структуры программы с генерируемым внешним интерфейсом в контексте мультиязыковой платформы, предназначенной для создания сред разработки (IDEs).

Для её достижения были поставлены следующие задачи.

1. Провести обзор предметной области — платформы SRC IDE, подходов к построению PSI-дерева и средств для его модификации, алгоритмов получения текстовых изменений.
2. Сформулировать требования к подсистеме.
3. Спроектировать механизм преобразований PSI-дерева и исходного документа, а также генератора классов и интерфейсов.
4. Реализовать генерацию внешних интерфейсов и классов для работы с системой модификации PSI-дерева.
5. Выполнить реализацию подсистемы преобразований PSI-дерева и исходного документа.
6. Апробировать созданную подсистему в контексте создания средств рефакторинга и быстрых исправлений.

2. Обзор предметной области

2.1. Платформа SRC IDE

Платформа SRC IDE предназначена для создания интегрированных сред разработки. На её основе разрабатывается Java IDE и Python IDE. В рамках этого раздела описываются компоненты платформы, существенные для реализуемой системы.

2.1.1. Модель кода

Модель кода (кодовая модель, Code Model) — подсистема, которая отвечает за представление синтаксиса и семантики исходного кода программы в IDE. Помимо этого модель кода предоставляет API для работы других подсистем, таких как навигация по исходному коду, рефакторинги (refactorings), быстрые исправления (quick fixes), автодополнение кода (code completion), поиск использований (find usages) и многих других.

Эта подсистема предоставляет внешним сервисам следующие возможности:

- информацию о созданных и открытых проектах;
- синтаксис и семантика исходного кода программы.

Модель кода большую часть информации хранит в индексах, которые являются специальными структурами данных для хранения знаний о синтаксисе и семантике исходного кода и дают возможность быстрого доступа к этим знаниям. В модели кода существует большое количество индексов разного значения, например:

- `ASTIndex` — индекс, который хранит абстрактное синтаксическое дерево для каждого файла проекта;
- `AllInheritorsIndex` — индекс, который позволяет по классу получить всех его наследников;

- `AllReturningIndex` — индекс, который позволяет по типу (сообразно системе типов языка программирования) получить все методы, которые этот тип возвращают.

Индексы строятся на этапе инициализации подсистемы модели кода. При этом, одни индексы могут строиться на основе информации из других индексов, тем самым образуя граф. Например, при построении `AllInheritorsIndex` и `AllReturningIndex` используются данные из `ASTIndex`.

Важно отметить, что подсистема, реализуемая в рамках данной работы, является частью модели кода.

2.1.2. Object Compression System

Индексы, описанные в предыдущем разделе, удобны при разработке IDE благодаря тому, что позволяют быстро получить необходимую информацию и имеют компактное представление. Последнее достигается за счет того, что индексы хранят информацию не в виде стандартных объектов языка Java, а в сжатом виде. Это достигается при помощи использования OCS-подсистемы (Object Compression System)[1]. OCS предназначена оптимизации хранения данных путем их сериализации и сжатия для работы с кэш-памятью процессора. При этом данная компонента предоставляет удобное API для работы со сжатыми данными как с обычными Java-объектами. В итоге использование OCS-подсистемы позволяет уменьшить потребление памяти за счет сжатия и увеличить производительность за счет уменьшения количества кэш-промахов.

OCS представляет объекты в виде набора колонок, каждая из которых соответствует полю класса, объекты которого подлежат сжатию. Обычный Java-объект, помимо данных, хранит также мета-информацию, которая не всегда нужна приложению. OCS-подсистема не сохраняет эту информацию, что позволяет сократить потребление памяти. Данные, лежащие в колонках, проходят через процедуру кодирования с дополнительным сжатием, что тоже способствует уменьшению зани-

[1]Название изменено с целью соблюдения соглашения о неразглашении.

маемой памяти. Таким образом, OCS-подсистема похожа на Protocol Buffers [11] или FlatBuffers [12], которые предоставляют похожую функциональность для сериализации объектов. Однако OCS предоставляет больше возможностей, которые перечислены ниже.

- Средства работы с наследованием, что имеет большое значение для приложений, написанных на Java.
- Функциональность по работе с циклическими ссылками Java-объектов. Это необходимо для представления в сериализованном виде различных структур данных вроде AST.
- Возможность использовать API с генерацией кода во время исполнения программы; благодаря этому разработчикам IDE не требуется производить дополнительных действий перед сборкой всей системы или прибегать к использованию специального языка описания, как, например, в случае Protocol Buffers.

OCS позволяет пометить классы, которые следует сериализовать, с помощью Java-аннотаций или специальных интерфейсов. Компонента во время исполнения будет генерировать прокси-классы с теми же методами, что и у классов упаковываемых объектов, и именно с объектами прокси-классов работает пользователь. На сериализуемые объекты накладываются определенные ограничения, но они не мешают при разработке функциональностей подсистемы модели кода.

Помимо генерируемых прокси-классов, OCS предоставляет структуры данных, которые по ключу позволяют получить значение или множество значений некоторых атрибутов — отображения (mappings). Отображения берут на себя поиск, сериализацию и десериализацию Java-объектов. Это реализация паттерна проектирования Repository, благодаря которому разработчикам подсистемы модели кода позволяется удобно строить свои структуры данных с использованием возможностей OCS без необходимости самим работать с колонками. Принцип схож с принципом работы Spring Data JPA [13] — платформа для Java,

которая позволяет взаимодействовать с сущностями баз данных без использования SQL-запросов. На основе отображений работает практически все множество индексов модели кода. Так, `ASTIndex` и `AllInheritorsIndex`, описанные ранее, представляют собой отображение из файла с исходным кодом программы в абстрактное синтаксическое дерево и отображение из класса в его наследников соответственно. При этом дерево и представления классов, наследников располагаются в OCS, разбитые по колонкам.

2.1.3. Language Server Protocol

На данный момент одной из самых важных технологий при построении IDE является протокол сервера языка (Language Server Protocol, LSP).

Эта технология была предложена компания Microsoft для разрабатываемого ею редактора кода Visual Studio Code [14]. Это открытый стандартизированный протокол, который позволяет построить IDE с помощью клиент-серверной архитектуры, отделив внутреннюю логику, специфичную для конкретных языков, в отдельные компоненты. Эти компоненты называются серверами языков (Language Server, LS). LSP полностью специфицирован [15]. Технология быстро нашла свое применение. Так, Microsoft стала взаимодействовать с Codenvy [16] и Red Hat [17], в результате чего появилась поддержка этого протокола в Eclipse Che [18] — Online IDE от Eclipse Cloud Development [19]. Также была реализована поддержка языка Java для Visual Studio Code [20], основанная на LS для Java от Eclipse [21]. Также JetBrains, которая раньше строила свои IDE на основе монолитной платформы [22], представила Fleet [23] — мультиязыковую IDE, которая также в своей основе использует LSP.

Language Server Protocol работает на основе протокола JSON-RPC. Как было сказано ранее, этот подход позволяет архитектурно разделить всю систему на две компоненты, которые взаимодействуют между собой через LSP.

1. Клиент, который может обрабатывать или посылать LSP-запросы. Как правило, это редактор кода или IDE. При этом клиент может запускаться и как десктопное приложение вроде Visual Studio Code, и как веб-приложение в браузере вроде Eclipse Che.

2. Сервер (Language Server, LS), в котором описывается вся логика, специфичная для конкретного языка программирования. Сервер предоставляет клиенту множество сервисов, таких как рефакторинги, быстрые исправления, автодополнение кода и пр. Существует большое количество серверов для поддержки различных языков программирования [24], что говорит о значительной популярности данной технологии.

Подход с использованием LSP позволяет обобщить сервисы инструментов разработки для многих языков. Если, например, редактор кода работает по LSP, то он может поддерживать множество языков программирования, так как для этого достаточно использовать необходимые языковые сервера и взаимодействовать с ними через этот протокол. Это позволяет достичь большей гибкости и расширяемости кодовой базы платформы за счет повторного использования общих компонент.

LS работает как отдельный процесс, а различные клиенты взаимодействуют с ним через протокол LSP. На рисунке 2 можно увидеть пример взаимодействия клиента и сервера по протоколу LSP.

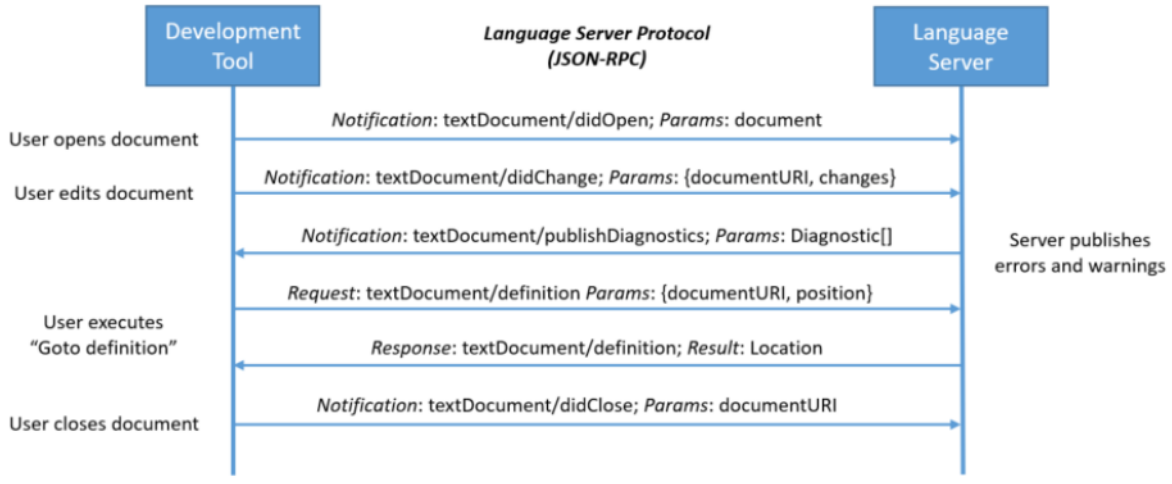

Рис. 1: Взаимодействие клиента и сервера по протоколу LSP

В рамках платформы SRC IDE реализуются два LS — для языков Python и Java. Так, сервисы рефакторинга и быстрых изменений работают на основе обработки запроса `textDocument/codeAction`, который приходит от клиента и является частью протокола LSP [25]. При этом определено, в каком формате сервер должен отправить ответ клиенту. Однако, сам процесс построения ответа может отличаться от сервера к серверу и учитывать специфику того или иного языка программирования.

2.2. Построение и модификация PSI

В рамках этого раздела описывается, какими свойствами должно обладать PSI-дерево. Приводятся различные способы его построения и взаимодействия с ним.

2.2.1. Lossless Syntax Tree

Обычно синтаксическое дерево содержит только ту информацию, которая необходима для компиляции программы. Пробелы и комментарии никак не учитываются. Однако для рефакторинга и других функций IDE необходимо учитывать пользовательское форматирование кода, включая, в частности, комментарии, как описывается в статье [26].

В [27] приведены следующие требования, которым должно удовлетворять дерево PSI.

- Оно должно максимально подробно представлять исходный код файла. При этом важно иметь информацию о пробелах и комментариях внутри самого дерева.
- Должен существовать способ отображать элементы дерева в области исходного документа, и наоборот.

Для удовлетворения этих требований существует структура данных Lossless Syntax Tree (LST). Это дерево, которое хранит всю информацию о содержимом исходного кода файла, с учетом пробелов и комментариев. На рисунке 2 можно увидеть пример LST.

Каждый узел такого дерева хранит свою длину и отступ относительно начала документа (`offset`). Такая структура, например, позволяет легко получить узлы дерева, которые относятся к тексту, выделенному мышкой: достаточно пройти по дереву и выбрать узел с нужными отступами и длиной.

Пробелы и комментарии хранятся в отдельных LST-узлах, которые называются тривиальными узлами (Trivia Nodes).

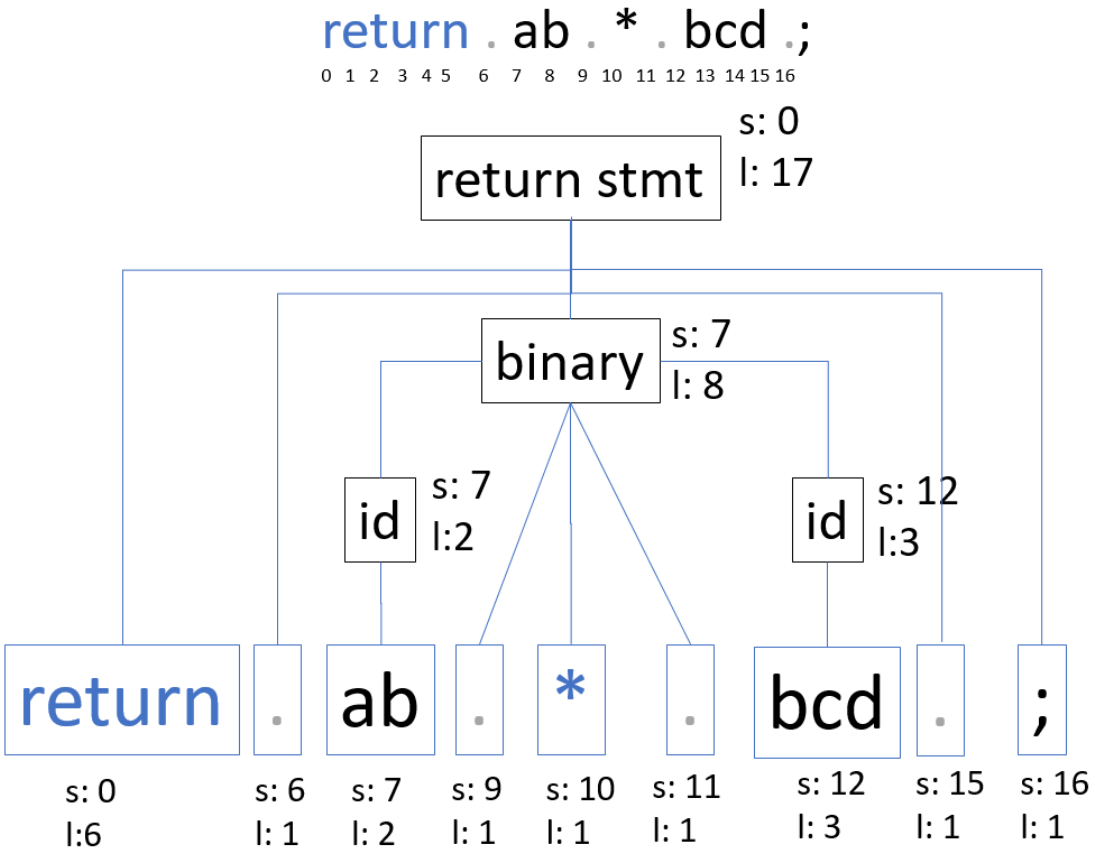

Рис. 2: Пример Lossless Syntax Tree

2.2.2. Mutable Tree

Один из самых очевидных способов спроектировать PSI-дерево и систему модификации на нем — изменяемое дерево (Mutable Tree). У такого подхода есть следующие достоинства:

- удобное API для модификации дерева;
- легкость самостоятельной реализации различных модификаций дерева;
- обеспечение актуальности версии дерева, с которым работает внешний сервис.

Однако, у данного подхода также имеются недостатки, перечисленные ниже.

- При модификации дерева необходимо обновлять длины и отступы у узлов. В худшем случае предстоит обойти почти все дерево с целью обновления всей этой информации.

- Семантическая информация может потерять свою актуальность. Так, если в PSI для Java хранилась информация о том, что в строке `this.x = x` первый `x` относится к полю, а второй — к локальной переменной, то если убрать объявление локальной переменной, эта информация перестанет быть актуальной, так как второй x теперь отсылается на поле. Инвалидация семантической информации может запутать клиентов, так как в какой-то момент времени, который тяжело уловить, семантическая информация может резко измениться.

- Если узел дерева, с которым работал клиент, оказался как-то удален из дерева, то клиент может столкнуться с проблемами, которые довольно тяжело отлаживать.

- Такое дерево плохо работает с многопоточным кодом, так как нуждается в синхронизации.

Применение подхода с изменяемым деревом можно обнаружить в инструментах для рефакторинга наподобие Smalltalk Refactoring Browser [28], CRefactory [29], а также в компиляторах. При этом само дерево в таких инструментах недоступно для внешних сервисов, оно только для внутреннего использования, так как отдавать его в публичное пользование довольно опасно.

Важно отметить, что, несмотря на перечисленные недостатки, этот подход можно успешно применять для построения системы модификации исходного кода. Так, он используется в IntelliJ Platform (JetBrains). Там существует дерево, которое реализуется классом `ASTNode`. Это дерево является изменяемым и предназначено для внутреннего использования. На его основе строится PSI-дерево, с которым и работает пользователь [30].

2.2.3. Immutable Tree

Приняв во внимание недостатки предыдущего подхода с изменяемым деревом, вполне понятным шагом является рассмотреть возможность сделать дерево неизменяемым. Этот подход имеет множество достоинств:

- потокобезопасность, которая является следствием отсутствия необходимости работать с синхронизацией потоков;
- фиксированные отступы и длины у узлов PSI-дерева.

Важной задачей все еще является построение системы модификации такой структуры данных. В рамках статьи [27] рассматривается два подхода: Rewriter и Persistent Tree.

2.2.4. Immutable Tree: Rewriter

Подход построения системы модификации над неизменяемым деревом, рассматриваемый в данном разделе, заключается в делегировании всех изменений в дереве отдельной сущности. Эта сущность реализуется объектом `Rewriter`. Данный подход применяется в Eclipse Java Development Tools (JDT) [31] и C/C++ Development Tools (CDT) [32].

`Rewriter` принимает на вход изменения PSI-дерева для всего файла с исходным кодом. Изменения могут быть следующими:

- два PSI-узла, где первый элемент пары — узел, который требуется заменить в PSI-дереве, а второй элемент — узел, на который требуется произвести замену;
- узел, который требуется удалить из PSI-дерева.

Переданные изменения PSI-дерева будут применены после вызова терминального метода `rewrite`. После обработки изменений объект Rewriter выдает текстовые изменения, которые описывают, что нужно поменять в документе с исходным кодом.

В листинге 1 показан пример того, как можно использовать объект `Rewriter` для удаления поля класса.

```
1 // Delete field
2 rewriter.removeNode(field);
3 // Get text changes
4 TextEdit edit = rewriter.rewrite();
5 // Apply changes to document
6 document.applyEdit(edit);
```

Листинг 1: Пример работы объекта Rewriter в JDT

На рис. 3 показано, как будет работать Rewriter в данном случае.

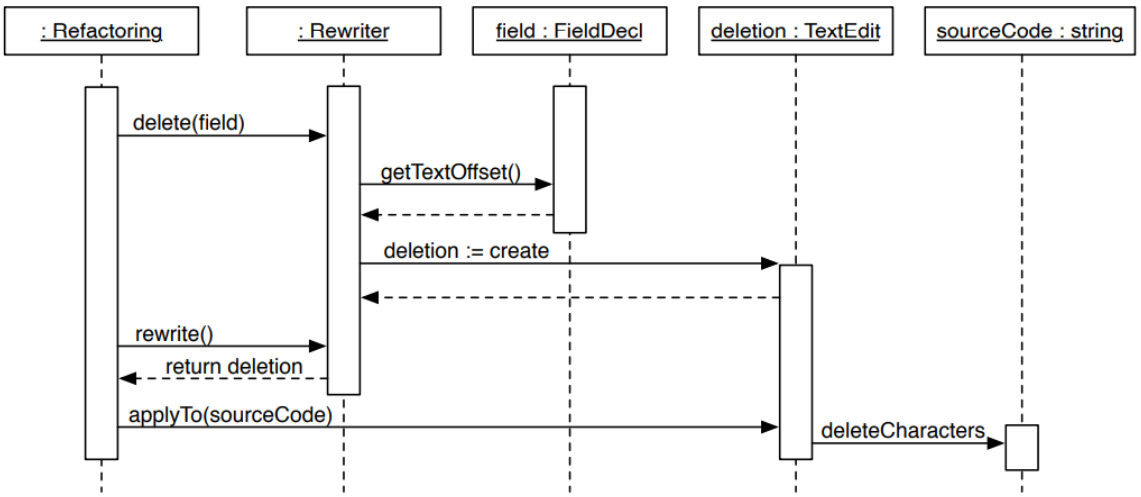

Рис. 3: Диаграмма последовательности при использовании Rewriter

Объект Rewriter эффективен для работы с неизменяемым деревом, который используется. Однако у него есть определенные ограничения. Так, при работе с Rewriter у клиента будет доступ только к первой версии дерева, но никак не к промежуточным и финальной, что может быть не всегда удобно. Для этого существует способ, который описан в следующем разделе.

2.2.5. Immutable Tree: Persistent Tree

Суть данного подхода заключается в построении персистентной структуры данных — структуры данных, при изменении которой клиент сможет работать с её промежуточными состояниями. При этом сама структура старается использовать неизмененные части повторно. Такого способа построения PSI-дерева придерживались разработчики из компании Microsoft при разработке Roslyn [33] — платформы с открытым исходным кодом, включающей компиляторы и средства анализа исходного кода.

Разработчики Roslyn построили API для модификации PSI-дерева следующим образом: создали методы модификации для каждого типа узла с учетом синтаксических особенностей языка. Например, узел `Method` будет иметь метод `withName(Identifier identifier)`, который будет создавать новый узел `Method` с новым именем. Такой способ позволяет изменять дерево и не нарушать синтаксис языка. Правда, такие методы довольно тяжело писать вручную, а потому приходится прибегать к генерации, что и было сделано в Roslyn.

Как было указано ранее, Persistent Tree способно переиспользовать старые узлы, которые не были изменены. На рисунке 4 зеленым изображены старые узлы, а красным — измененные.

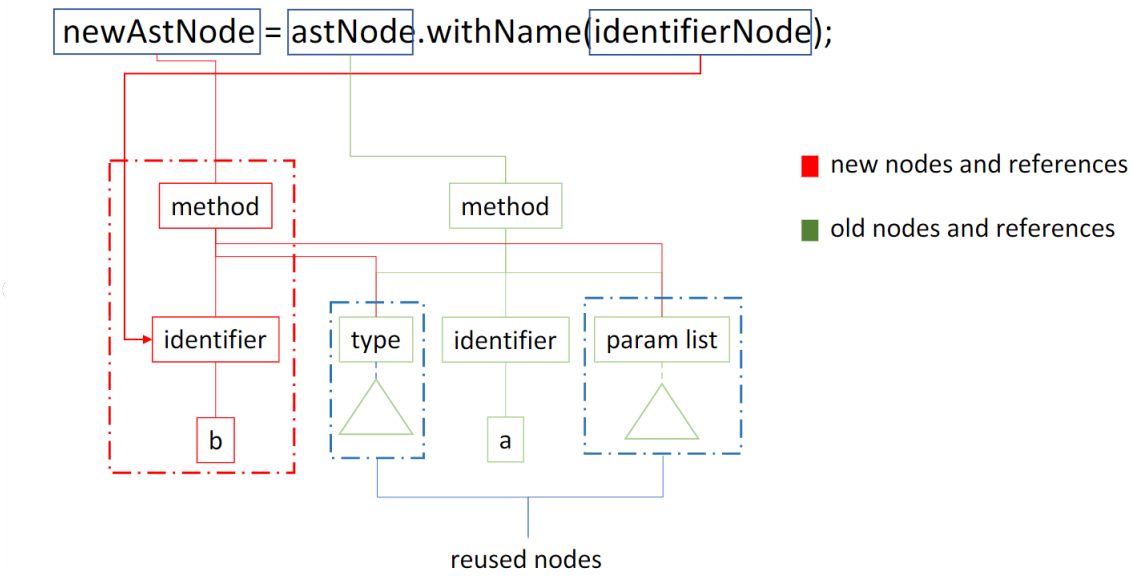

Рис. 4: Переиспользование узлов при модификации

При реализации такой структуры данных возникает следующая проблема. Для работы с деревом принято в каждом его узле хранить ссылку на узла-родителя и на узлов-детей. При этом сам узел неизменяем, нет возможности установить ему нового родителя или ребенка без пересоздания. Тогда получается два сценария, описанные ниже.

- Узел принимает в конструктор родителя, дерево строится сверху-вниз, как в листинге 2. Тогда получается установить узлу `b` родителя `a`, но не удается установить узлу `a` ребенка `b`.
- Узел принимает в конструкторе детей, дерево строится снизу-вверх, как в листинге 3. Тогда получается узлу `a` установить ребенка `b`, но не удается установить узлу `b` родителя.

```
1    Node a = new Node(null);
2    Node b = new Node(a);
```

Листинг 2: Установка узлу b родителя a

```
1    Node b = new Node(null);
2    Node a = new Node(b);
```

Листинг 3: Установка узлу a ребенка b

Таким образом, при наличии циклических ссылок между родителями и детьми нельзя добиться неизменяемости с корректной модификацией. Более того, для корректного переиспользования узлов нельзя хранить ссылки на родителей (при изменении одного узла придется изменять все поддерево с целью обновления родителя). Но при этом для удобной итерации по дереву все еще требуются ссылки на родителей.

Разработчики Roslyn предложили решение этой проблемы — красные и зеленые деревья [34]. Persistent Tree представляет включает в себя оба этих дерева.

1. Зеленое дерево, которое является неизменяемым и строится при парсинге снизу-вверх, как обычное синтаксическое дерево. Узлы этого дерева (будем их называть зелеными узлами) содержат в себе информацию о длине, о типе и так далее. Также каждый узел этого дерева хранит ссылки на детей, но при этом не хранит ссылку на родителя. В действительности, эта структура является деталью реализации, которая сокрыта от клиента. Пользователь никак не взаимодействует с зеленым деревом. Поэтому такое дерево даже не является типизированным, оно нужно, чтобы хранить внутреннюю информацию.

2. Красное дерево является неизменяемым, типизированным и строится сверху-вниз ленивым способом, по запросу. Именно оно является PSI-деревом, с которым работает клиент. Узлы такого дерева (красные узлы) хранят ссылки на соответствующие зеленые узлы. Так, красное дерево образует фасад над зеленым деревом. Каждый красный узел обязательно хранит отступ и ссылку на родителя.

Работа с такой структурой данных может выглядеть так, как показано на листинге 4.

```
1    // Create green tree
2    GreenNode b = new GreenNode();
3    GreenNode c = new GreenNode();
4    GreenNode a = new GreenNode(b, c);
5    // Then create a red node
6    RedNode root = new RedNode(offset:0,
7                               parent:null,
8                               gNode:a);
```

Листинг 4: Работа с красными и зелеными узлами

Можно заметить, что в этом листинге был создан только один красный узел. Красные узлы для детей создаются только в том случае, если этого просит клиент (например, вызвав метод `getChildren()`). Как

видно на второй схеме рисунка 5, красный узел `root` пока один и он ссылается на зеленый узел `a`. На третьей схеме видно, как будет вести себя данная структура при запросе на детей узла `root`. Ленивым образом будет создан новый красный узел со ссылкой на зеленый узел `b`, ему устанавливается `root` в качестве родителя.

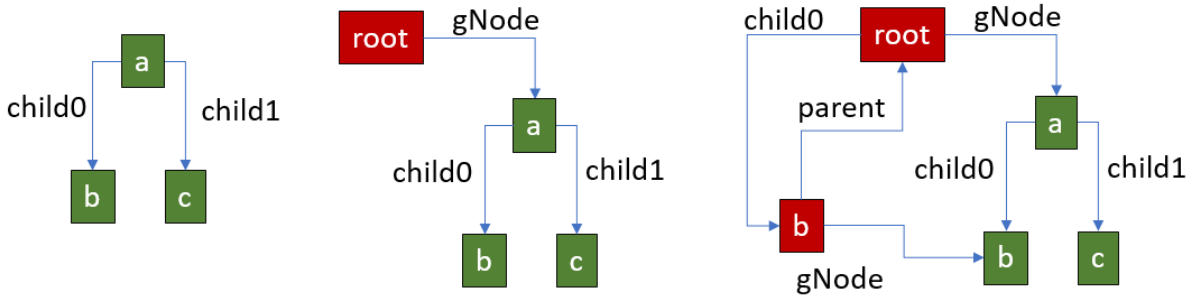

Рис. 5: Красные и зеленые узлы

Если говорить о модификации, то она производится на красных узлах через методы, о которых говорилось выше. Каждый метод модификации создает новый красный узел с новым ребенком, при этом переиспользуя старые узлы, которые не поменялись. Для этого создается новый зеленый узел с нужными детьми. Получается, что клиент при каждом вызове метода модификации будет получать новую версию поддерева.

Все описанное выше позволяет модифицировать отдельные PSI-узлы, получая их новые версии. Однако помимо этого необходимо преобразовывать дерево всего файла. Для этого в Roslyn применяется `Rewriter`, который был рассмотрен в предыдущем разделе, но с некоторыми улучшениями. `Rewriter` в Roslyn принимает поддеревья, которые нужно заменить. При этом если раньше он возвращал текстовые изменения для документа, то сейчас он может возвращать новую версию PSI-дерева файла, в котором уже применены все необходимые изменения.

Сами изменения производятся на уровне зеленых узлов. Так как зеленые узлы не хранят отступы, то не возникает проблем с обновлением данных об отступах и длинах в узлах. Пользователь сможет получить информацию о правильных отступах по запросу, когда ленивым обра-

зом построится часть красного дерева. Этот подход опирается на то, что клиенту не нужно все дерево, а только какая-то его часть. Худший случай может возникнуть, если пользователь в первый раз запросит узлы, находящиеся в конце документа, и придется один раз построить красное дерево, что может оказаться дороже по количеству операций, чем проход по всему дереву с обновлением всех отступов.

Такой подход к построению системы модификации PSI имеет следующие достоинства.

- Клиент может работать с множеством версий PSI-дерева.
- Отсутствуют трудности с обновлением длин и отступов ввиду создания новых PSI-узлов с новыми длинами и отступами после модификации.
- После применения изменений происходит обновление семантической информации, и так как теперь этапы модификации четко разделены, пользователь знает, когда он работает с актуальной семантической информацией, и когда она может обновляться.
- Методы модификации не позволяют построить дерево, нарушающее синтаксис языка.
- Потокобезопасность ввиду использования неизменяемой структуры данных.

Главной проблемой такой реализации является тот самый худший случай, при котором приходится строить все красное дерево. Однако он потенциально довольно редок. К тому же, такая ситуация может возникнуть всего один раз за все время работы IDE.

Отдельно стоит отметить, что Microsoft Roslyn особым способом работают с тривиальными узлами. Так, они вводят следующий инвариант: тривиальные узлы могут быть детьми токенов и только их. Это сильно упрощает создание многих компонентов системы. Например, у Roslyn `Rewriter` создан путем генерации кода.

2.2.6. Выводы

По результату обзора можно увидеть, что последний рассмотренный подход обладает наибольшим количеством положительных сторон, которые перечислены ниже.

- Безопасное API для пользователя, которое не позволяет нарушить правила синтаксиса языка.
- Безопасная работа с семантической информацией.
- Дерево, которое соответствует требованиям для модели кода IDE.
- Безопасная работа в многопоточных условиях.

Поэтому было принято решение при проектировании системы опираться на подход Persistent Tree от Roslyn с учетом особенностей работы модели кода SRC IDE.

2.3. Преобразование исходного документа

После выполнения преобразований над PSI-деревом необходимо перенести изменения в исходный документ, чтобы они стали видны пользователю инструмента разработки. При этом, чем меньше последовательность и чем меньше сами изменения в контексте исходного документа, тем лучше для скорости работы внешних сервисов. Такая проблема известна как задача получения кратчайшей последовательности текстовых изменений, которые можно применить к исходному документу.

Подобную проблему решает `diff` [35] — утилита семейства операционных систем Unix. Она принимает два текстовых файла (или две строки) и генерирует сценарий преобразования одного файла (строки) в другой.

Текущая задача отличается тем, что сравнивать необходимо не текстовые файлы, а деревья. Конечно, так как сравниваются два LST-дерева, их можно перевести в текст и применить текстовый `diff`, однако перевод дерева в текст подразумевает прохождение по всем его узлам, что занимает продолжительное время. Более того, если учитывать структуру PSI-дерева, появляется возможность получать более мелкие текстовые изменения.

В работе [36] описывается множество подходов для сравнения деревьев с рассмотрением действий по добавлению (add), удалению (delete) и обновлению (update) узла (например, изменению текстового значения у токена). Однако самый быстрый алгоритм из этого семейства, RTED [37], имеет асимптотику $O(n^3)$, где n — количество узлов в дереве. Средства разработки довольно часто вынуждены иметь дело с большими документами, а потому время получения текстовых изменений в таком случае может быть достаточно долгим.

Дальнейшие решения стараются не только улучшить асимптотику, но и при сравнении определять перемещения (move) поддеревьев. Это важно, потому что такая операция довольно часто встречается в рамках работы с IDE (Rearrange Code Refactoring — пример того сервиса ре-

факторинга, где код в основном перемещается). Алгоритм, описанный в [38], вычисляет последовательность текстовых изменений для деревьев, которые представляют содержимое LaTeX-файлов. Он лучше предыдущих в плане ассимптотики, но при этом у него есть ограничения, которые ограничивают его применимость в контексте IDE. Например, он работает из предположения, что в листьях дерева содержится большое количество текста, что не так для IDE.

Развитием алгоритма [38] является алгоритм ChangeDistiller [39], который больше подходит для AST. Однако, и этот алгоритм исходит из того, что в листьях располагается большое количество текста.

Это ограничение преодолевает алгоритм GumTree [40], который производит сравнение деревьев с генерацией последовательности изменений (в том числе с move) более точно за счет исправления недочетов алгоритмов [38] и [39]. Более того, у него достаточная для нужд IDE асимптотическая сложность $O(n^2)$.

В итоге, именно GumTree было решено взять ввиду его точности и достойной производительности. Более того, на основе этого алгоритма построен инструмент для сравнения деревьев [41], который имеет открытый исходный код. Это значительно облегчает адаптацию алгоритма под существующую кодовую базу.

3. Требования

Данный раздел описывает функциональные и нефункциональные требования, которым должна удовлетворять разрабатываемая подсистема.

3.1. Функциональные требования

Были выделены следующие функциональные требования.

- Подсистема должна позволять посредством генерации получить классы и интерфейсы для работы с PSI любого языка программирования.
- Подсистема должна предоставлять возможность дописывать в полученные путем генерации файлы код, который не будет исчезать при повторном запуске генератора.
- Подсистема должна позволять производить модификацию отдельных узлов PSI-дерева. Каждый PSI-узел дерева должен предоставлять методы для получения и модификации детей этого PSI-узла, которые соотносятся с синтаксической структурой конструкции, которую этот узел представляет.
- Подсистема должна не позволять получить синтаксически неверную конструкцию языка в результате использования методов модификации узлов PSI-дерева.
- Подсистема должна предоставлять фабрику для построения новых PSI-узлов из существующих, при этом методы этой фабрики не позволяют построить синтаксически неверную конструкцию языка.
- Подсистема должна позволять переписывать PSI-дерево всего файла без необходимости модификации конкретных узлов в дереве. В результате переписывания не должно получаться PSI-дерево, которое представляет программу с синтаксическими ошибками.

- Подсистема должна позволять вычислять последовательность текстовых изменений, которые можно применить к исходному документу для того, чтобы пользователь IDE мог увидеть результаты преобразований над структурой программы. Эта последовательность не может быть упорядоченной, т.е. результат применения ее элементов к исходному документу не должен зависеть от порядка, в котором находятся эти элементы. Данное ограничение существует из-за особенностей работы клиентской части платформы SRC IDE, в которой не специфицирован порядок применения текстовых изменений. Текстовые изменения могут быть только следующих типов: добавление (add), удаление (remove) и replace (обновление). Каждое текстовое изменение содержит два числа — позиции начала и конца изменения в исходном документа, а также строковое значение — новый текст, который должен появиться в документе. При этом, если текстовое изменение является добавлением, то позиции начала и конца должны совпадать, если оно является удалением, то строковое значение должно быть пустым. Это ограничение диктуется спецификацией Language Server Protocol [15].

- Подсистема должна позволять работать с различным версиями деревьев и поддеревьев, которые получаются в результате модификации PSI-узлов.

- Подсистема должна позволять корректно выполнять модификацию PSI-дерева и документа в условиях параллельного исполнения сервисов рефакторингов и быстрых исправлений.

- Подсистема должна допускать повторное использование при разработке новых IDE в контексте платформы SRC IDE.

3.2. Нефункциональные требования

Подсистема должна удовлетворять следующим нефункциональным требованиям:

- разрабатываться на языке программирования Java;
- оформление исходного кода должно соответствовать стандартам, принятым в Saint-Petersburg Research Center;
- быть понятной в использовании внешним клиентам;
- быть масштабируемой в рамках платформы SRC IDE;
- работать на разных операционных системах с минимальными изменениями;
- в качестве среды разработки следовало использовать IntelliJ IDEA.

4. Архитектура

Данный раздел представляет архитектуру подсистемы. В ней приводятся UML-диаграммы, по которым видно, из каких компонент состоит подсистема. С помощью диаграмм последовательностей описывается главный случай использования подсистемы. Приводится описание API реализуемой подсистемы, которое доступно доступно клиентам. Под клиентами подразумеваются как внешние сервисы вроде рефакторингов и быстрых исправлений, так и пользователи генератора внешнего интерфейса — разработчики из модели кода (более подробно об этой подсистеме рассказывается в разделе 2.1.1).

Предлагаемая подсистема состоит, в свою очередь, из двух следующих подсистем:

- подсистема генерации внешних интерфейсов и классов;
- подсистема модификации PSI и исходного документа.

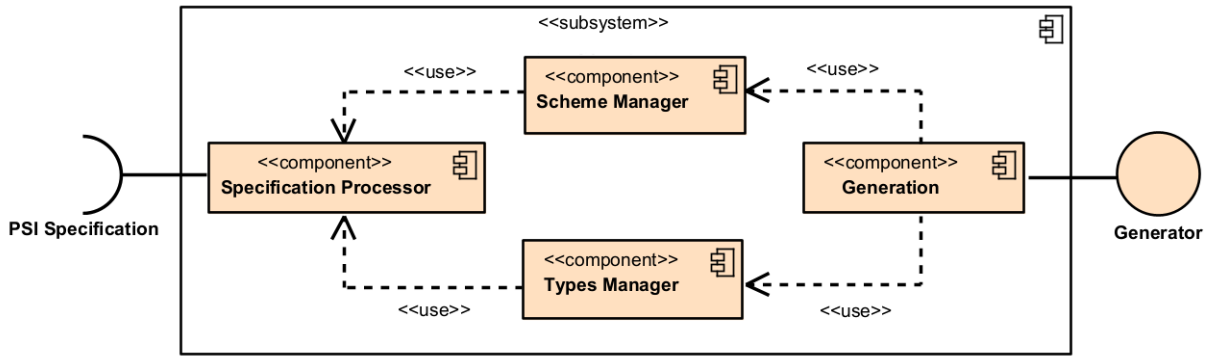

Рис. 6: Подсистема генерации интерфейсов и классов

На рисунке 6 представлена диаграмма компонент для подсистемы генерации интерфейсов и классов. Она состоит из следующих компонент.

- `Specification Processor` — отвечает за обработку и валидацию составленной пользователем спецификации типов синтаксических конструкций языка программирования, для которого строится PSI.

- `Scheme Manager` — хранит знания о схемах генерации интерфейсов и классов языка Java, созданные на основе информации о типах от Specification Processor: какие интерфейсы реализуются, какой порядок у детей при генерации и др. Scheme Manager реализует паттерн проектирования Singleton.

- `Types Manager` — хранит знания о семантике типов синтаксических конструкций: типы детей, свойства и др. Как и `Scheme Manager`, реализует паттерн проектирования Singleton.

- `Generation` — основная компонента, которая реализует функциональность, связанную с генерацией PSI. Предоставляет интерфейс `Generator`, который отвечает за генерацию конкретного файла.

Пользователю, желающему сгенерировать PSI для своего IDE, достаточно написать спецификацию типов синтаксических конструкций соответствующего языка программирования и запустить генератор. При необходимости представленную подсистему можно расширить для поддержки других языков программирования.

На рисунке 7 представлена диаграмма компонент UML, описывающая подсистему модификации PSI и исходного документа (редактируемого текста программы).

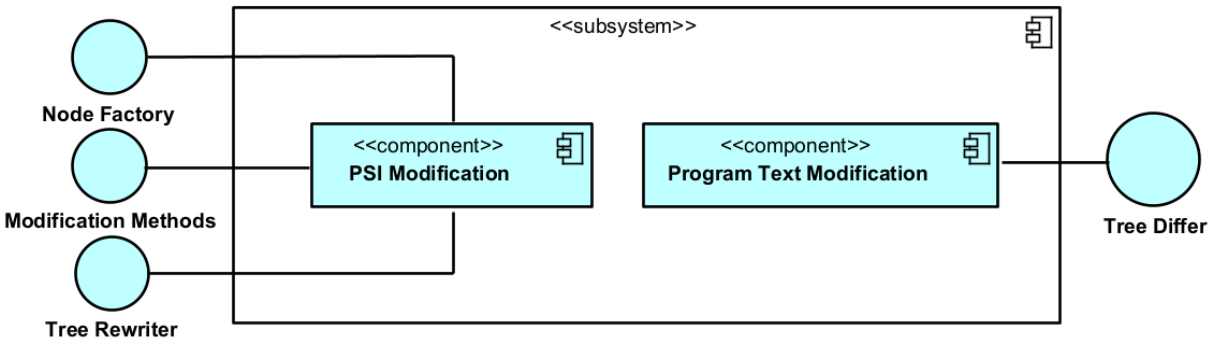

Рис. 7: Подсистема модификации PSI с получением текстовых изменений

Компонента `PSI Modification` предоставляет внешним клиентам разные способы модификации и построения новых узлов PSI. Клиенты могут взаимодействовать с этой компонентой через следующие сущности.

- `Node Factory` — фабрика, которая предоставляет методы как для конструирования узлов PSI из других узлов. Эта фабрика генерируется, но при этом для неё есть возможность «вручную» дописывать дополнительные методы.

- `Modification Methods` — сгенерированные методы, которые есть у каждого интерфейса и класса для модификации атрибутов конструкции. Эти методы позволяют создать новую версию узла, позволяя заменять существующих детей.

- `Tree Rewriter` — сущность, которая позволяет создать новую копию PSI заменив (replace) или удалив (remove) некоторые узлы. Реализует паттерн проектирования Builder.

Компонента `Program Text Modification` предоставляет клиентам возможность получения текстовых изменений документа после преобразований над PSI. Клиенту предоставляется `Tree Differ`, который, получив два дерева, создает набор текстовых изменений, которые клиент может применить к документу с исходным кодом.

На рисунке 8 показана диаграмма последовательностей UML, в котором описан главный сценарий использования подсистемы данной подсистемы. Этот сценарий состоит из следующих шагов.

- Модификация узлов дерева (1) или построение новых узлов с помощью фабрики (2). В результате у внешнего сервиса доступны новые узлы дерева.

- Модификация всего PSI при помощи интерфейса `Tree Rewriter`, в результате чего создаётся новая, модифицированная копия PSI. Сначала сервис инициализирует этот интерфейс корнем нового PSI (3), далее через `replace`/`remove` методы указывает, какие преобразования нужно совершить (4). С помощью метода `rewrite` (5) активируется процесс модификации, в результате сервис получает новый PSI.

- Получение набора текстовых изменений при помощи интерфейса `Tree Differ`, который принимает на вход корни старого и нового PSI, сравнивает их и создаёт спецификацию текстовых изменений (6).

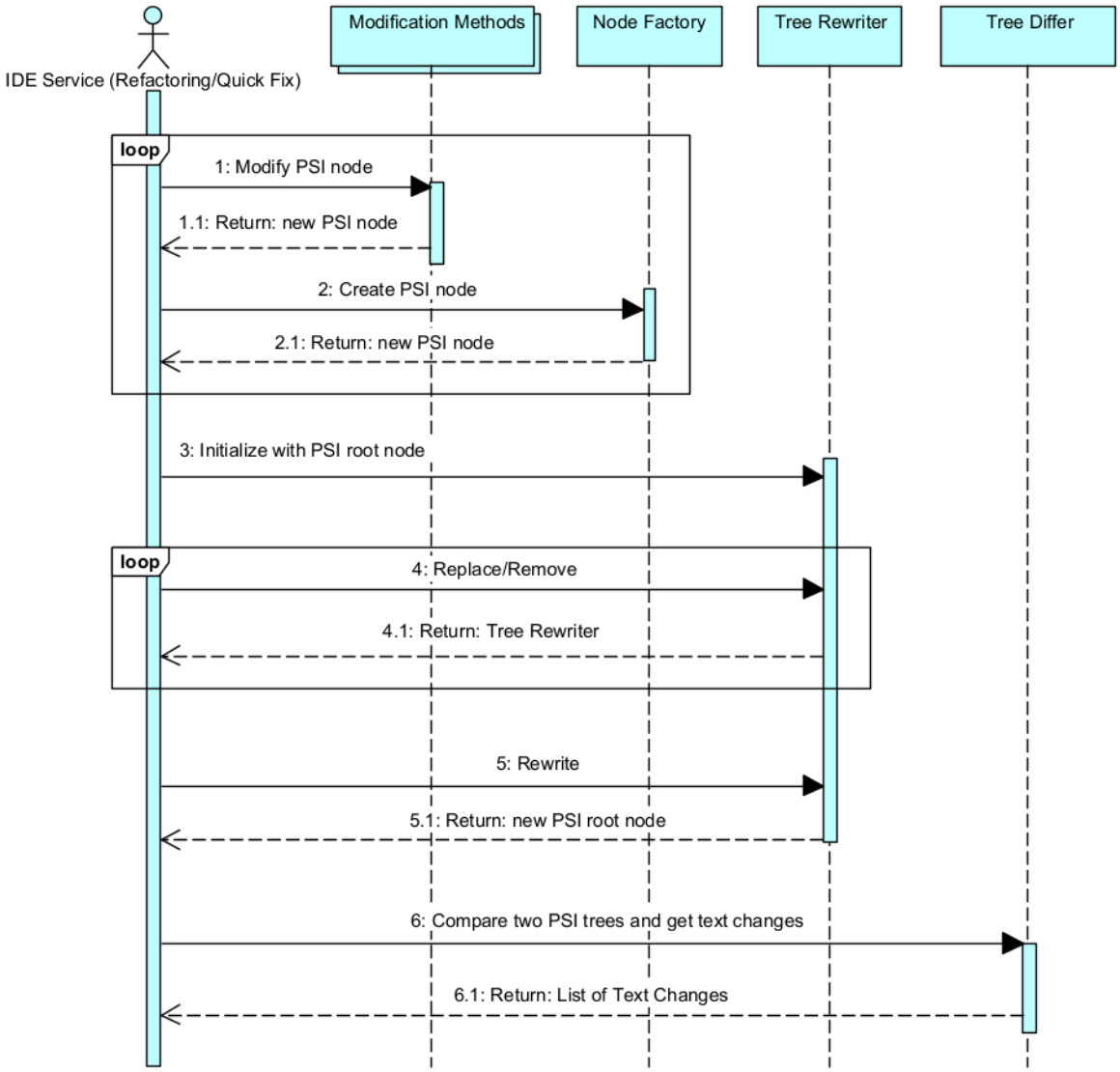

Рис. 8: Основной сценарий использования подсистемы модификации PSI и исходного документа

Теперь рассмотрим более подробно API, который предоставляется клиентам реализуемой подсистемой[2].

Каждый интерфейс, который описывает тип синтаксический конструкции, имеет методы для получения и модификации своих детей. Под модификацией подразумевается получение новой копии узла с замененным ребенком. Типы детей описываются в спецификации. Например в листинге 5 описан интерфейс для бинарного выражения в языке Python. Все эти методы получаются как результат генерации.

```
1  public interface IPyBinaryExpression extends IPyExpression {
2      // Generated methods start
3      IPyExpression getLeftExpression();
4
5      IPyToken getOperatorToken();
6
7      IPyExpression getRightExpression();
8
9      IPyBinaryExpression withLeftOperand(IPyExpression left);
10
11     IPyBinaryExpression withOperatorToken(IPyToken operator);
12
13     IPyBinaryExpression withRightOperand(IPyExpression right);
14     // Generated methods end
15 }
```

Листинг 5: Пример интерфейса узла PSI дерева

Частью компоненты `PSI Modification` является фабрика, методы которой также генерируются. Через эти методы фабрика предоставляет функциональность для конструирования новых узлов PSI на основе существующих узлов. На листинге 6 представлен пример такого метода.

[2]Названия интерфейсов и методов изменены с целью соблюдения соглашения о неразглашении.

```
class PyNodeFactory {
    // Generated methods start
    // ...
    public static IPyBinaryExpression newBinaryExpression(
        IPyExpression left,
        IPyToken operator,
        IPyExpression right) { ... }
    // ...
    // Generated methods end
}
```

Листинг 6: Пример метода фабрики

Компонента PSI Modification предоставляет интерфейс Tree Rewriter. Как было указано ранее, он позволяет полностью переписывать дерево. Соответствующий интерфейс на языке Java описан в листинге 7.

```
public interface TreeRewriter {
    TreeRewriter replace(PSINode oldNode, PSINode newNode);

    TreeRewriter remove(PSINode oldNode);

    PSINode rewrite();

    PSINode rewrite(PSIDecorator ... decorators);
}
```

Листинг 7: Интерфейс Tree Rewriter

Методы `replace` и `remove` предназначены для накапливания информации о действиях, которые надо применить к дереву. Метод `rewrite` активирует процесс преобразований над деревом, в результате которого получается новая модифицированная версия дерева.

Можно заметить, что у компоненты `Rewriter` есть еще один метод `rewrite`, который принимает в себя набор объектов `PSIDecorator`. `PSIDecorator` — это функциональный интерфейс, который содержит

один метод `decorate`. Он необходим для того, чтобы производить дополнительные действия над деревом, которые могут быть специфичны для конкретного языка. `Rewriter` реализует паттерн проектирования Decorator. В листинге 8 показан соответствующий интерфейс. В разделах 5.2.3 и 5.2.4 присутствуют примеры использования этого интерфейса.

```
1 public interface PSIDecorator {
2     PSINode decorate(PSINode node);
3 }
```

Листинг 8: Интерфейс PSIDecorator

Компонента `Program Text Modification` предоставляет интерфейс `Tree Differ`. Он состоит из одного метода для получения списка текстовых изменений на основе двух версий дерева. Эти текстовые изменения описывают порядок действий, которые нужно воспроизвести, чтобы на основе первого документа получить второй. Интерфейс представлен на листинге 9.

```
1 public interface TreeDiffer {
2     List<TextChange> getTextChanges(PSINode src, PSINode dst);
3 }
```

Листинг 9: Интерфейс Tree Differ

Подсистема генерации, несмотря на свое не самое тривиальное устройство, предоставляет пользователям всего один интерфейс `Generator` с одним методом для получение содержимого файла. При этом он параметризуется вспомогательным интерфейсом `GenSpec`, который предоставляет генератору необходимую информацию для формирования исходного кода правильным образом. В листинге 10 представлен интерфейс генератора.

```
1 public interface Generator<S extends GenSpec> {
2     String generate();
3 }
```

Листинг 10: Интерфейс генерации

5. Особенности реализации

Данный раздел знакомит с реализацией подсистем, которые были представлены в разделе 4. Описывается, какие были приняты технические решения, приводится их обоснование, а также описываются проблемы, решенные при реализации.

Из-за соглашения о неразглашении привести ссылку на исходный код с реализацией не представляется возможным. Названия сущностей изменены с целью соблюдения этого соглашения.

5.1. Генератор внешнего интерфейса

Как описывалось в разделе 4, подсистема генерации ставит перед собой задачу быстрого получения интерфейсов и классов, которые являются API всей подсистемы. В первую очередь генерации подлежат интерфейсы и классы, которые описывают узлы PSI-дерева. Далее идут вспомогательные сущности, упрощающие взаимодействие с системой.

5.1.1. Спецификация

Спецификация представляет из себя текстовый файл, в котором описывается структура узлов PSI-дерева сообразно синтаксическим конструкциям языка программирования, для которого разрабатывается IDE.

В качестве формата для описания узлов PSI был выбран формат JSON. Такой выбор был сделан в силу следующих причин.

- Популярность формата — большинство разработчиков знает, как он устроен, и для них не будет проблемой создать спецификацию.
- Инструментарий — существуют библиотеки, которые позволяют легко генерировать и обрабатывать JSON-файлы. Так, в рамках этой работы используется библиотека GSON [42]. Это библиотека для языка Java, разработанная в компании Google, которая позволяет конвертировать JSON-объекты в Java-объекты, и наоборот.

Следует отметить, что среды разработки, подобные IntelliJ IDEA, позволяют довольно удобно редактировать JSON вручную.

Файл спецификации является списком JSON-объектов, каждый из которых описывает тип синтаксической конструкции — тип узла PSI-дерева. Каждый JSON-объект имеет множество атрибутов. Рассмотрим некоторые из них.

- `name` — строка с именем, типа узла. Необходим для названия интерфейса, класса имплементации, имени фабрики и других генерируемых сущностей. Обязательный атрибут.

- `extends` — строка с именем типа-родителя узла. Необходим для поддержки иерархии наследования типов узлов в PSI. С точки зрения синтаксиса языка Java — имя класса/интерфейса, от которого будет наследован класс/интерфейс, описывающий текущий тип. Обязательный атрибут.

- `is_abstract` — логическое (`true`/`false`) значение, является ли тип абстрактным. Если тип является абстрактным, то он будет описываться абстрактным классом. Для него не будет создаваться метод фабрики. Однако этот класс может содержать некоторые методы, которые будут переопределяться типами-наследниками. Этот атрибут не является обязательным, если его не указать, он будет иметь значение `false` по умолчанию.

- `implements` — список строк, который описывает, какие интерфейсы реализует генерируемый класс для этого типа. Он бывает необходим, если требуется указать, что класс реализует интерфейсы, существующие в кодовой модели и не являющиеся интерфейсами других типов в PSI-дереве. Для заполнения требуется указать полное квалифицированное имя (fully quialified name) интерфейса: пакет и имя. Если реализуемый интерфейс сам описывает тип в дереве, то пакет указывать не нужно. Этот атрибут не является обязательным, по умолчанию представляет собой список с интерфейсом того же типа, что и класс.

- `super_interfaces` — список строк который описывает интерфейсы, от которых унаследован интерфейс для описания текущего типа. Синтаксис идентичен предыдущему атрибуту. Он не является обязательным, по умолчанию список с интерфейсом того же типа, что и класс.

- `children` — один из важнейших обязательных атрибутов, который описывает, какие могут быть дети у узла данного типа. Представляет собой упорядоченный список JSON-объектов, где каждый объект специфицирует информацию о ребенке и его роли и имеет множество атрибутов, важные как для генерации, так и системы модификации в целом. Рассмотрим некоторые из этих атрибутов.

  · `name` — название ребенка в контексте текущего типа. Это обязательный строковый атрибут. Он необходим для названий методов получения (геттер) и модификации ребенка (сеттер), имени аргумента внутри метода фабрики.

  · `child_type` — обязательный строковый атрибут, тип узла, соответствующий типу, описанному в спецификации. Используется при генерации (тип возвращаемого значения для метода получения ребенка, например).

  · `is_mandatory` — логическое значение, описывающее, является ли ребенок (или дети) в данной роли обязательными. От этого зависит, можно ли получить `null` (или пустой список) из метода получения ребенка (или детей) и можно ли подавать `null` (или пустой список) вместо ребенка (или детей). Этот атрибут не является обязательным, по умолчанию ставится значение `true`.

  · `is_list` — логическое значение, описывающее, может ли ребенок появляться в списке, или нет. От этого зависят возвращаемые типы и типы аргументов у методов для взаимодействия с ребенком. Например, у узла типа `ArgumentList` можно есть дети с ролью `Argument`, для них `is_list` является

`true` и потенциальный метод будет возвращать `List<Argument>`.

Данная спецификация достаточно полно описывает синтаксическую структуру языка, удобна для обработки при генерации.

Единственная проблема, которую имеет такой JSON-формат, заключается большом объеме информации, из-за чего заполнять вручную файл оказываетеся затруднительно. Однако процесс заполнения можно ускорить: часть этого файла можно автоматически генерировать. Например, в
IntelliJ Platform есть свои классы для реализации узлов Python PSI [43]. Воспользовавшись Reflection API и разобрав эти классы, удалось получить первую версию файла спецификации через генерацию при помощи библиотеки GSON. Дальше файл все равно пришлось заполнять вручную, но уже не было необходимости его писать с самого начала. Подобным образом было сделано и для Java IDE, только там за основу были взяты классы из дерева Eclipse JDT.

5.1.2. Обработка спецификации

Как было указано в разделе 4, за обработку спецификации отвечает компонента `Specification Processor`. Далее описывается её основной принцип работы.

Спецификация обрабатывается в несколько проходов.

1. Синтаксический разбор и валидация файла спецификации.
2. Инициализация компоненты `Types Manager`.
3. Инициализация компоненты `Scheme Manager`.

Стоит отметить, что второй и третий проход не зависят друг от друга, но при этом они оба зависят от результата первого прохода.

Синтаксический разбор JSON-файла производится средствами библиотеки GSON. Результатом этого разбора являются POJO (Plain Old Java Object) объекты, которые образуют взаимное отображение вместе с JSON-объектами исходного файла. Важной частью этапа разбора

также является проверка файла спецификации на корректность — валидация. Это необходимо для предотвращения ошибок при заполнении файла, которые могли бы предотвратить к неожиданному поведению системы. Например, производятся следующие проверки:

- проверка наличия всех обязательных атрибутов;
- проверка типов всех атрибутов;
- проверка отсутствия лишних атрибутов у JSON-объектов.

Для инициализации компоненты `Types Manager` обрабатывается результат первого этапа. Производится анализ, во время которого создаются объекты, которые инкапсулируют информацию о семантике типов синтаксических конструкций, но не о построении нужных классов и интерфейсов. Они хранятся внутри менеджера типов. Во время этого прохода также происходит валидация. Например, проверяется, что типы детей описаны в самом файле, правильные связи атрибутов между собой и пр.

Инициализация компоненты `Scheme Manager` происходит похожим образом. Создаются POJO объекты, которые инкапсулируют информацию о том, каким образом генерировать те или иные классы, связанные с конкретным типом. В конечном итоге, ими распоряжается `Scheme Manager`. Как и в предыдущие разы, на этом этапе тоже валидируются некоторые атрибуты.

Как отмечалось в разделе 4, `Types Manager` и `Scheme Manager` реализуют паттерн проектирования Singleton, т.е. они инициализируются только один раз по первому запросу. Такое решение было принято потому, что информация внутри менеджеров не зависит от дальнейшего состояния всей системы, а значит все проходы обработки спецификации, которые могут происходить не очень быстро, можно произвести всего один раз и только тогда, когда это действительно необходимо.

5.1.3. Генерация

В разделе 4 указывалось, что генерацию Java-файлов внешнего интерфейса осуществляет компонента `Generation`, которая предоставляет внешним клиентам интерфейс `Generator`. Каждый объект, который реализует этот интерфейс, отвечает за генерацию одного файла (интерфейса или класса). Таким образом клиент (в данном случае — разработчик подсистемы модели кода, см. раздел 2.1.1) при необходимости генерации нового Java-файла может создать класс, который реализует интерфейс `Generator`. Этот класс будет отвечать за генерацию нового Java-файла. Такой подход позволяет клиенту расширять внешний интерфейс для подсистемы модификации и добавлять новые возможности в подсистему модели кода. Далее описан принцип работы компоненты `Generation`.

Для того, чтобы генератор мог корректно выполнять свою работу, ему необходимо иметь данные о том, какие Java-файлы необходимо создать и каким образом это нужно сделать. По этой причине с каждым генератором связан объект, реализующий интерфейс `GenSpec`.

Каждый объект `GenSpec` содержит всю информацию, необходимую для генерации файла. При инициализации генератора создается и объект `GenSpec`, который, в свою очередь, используют `Types Manager` и `Scheme Manager` для заполнения своих полей. На рисунке 9 представлена UML-диаграмма, которая показывает внутреннее представление компоненты генерации.

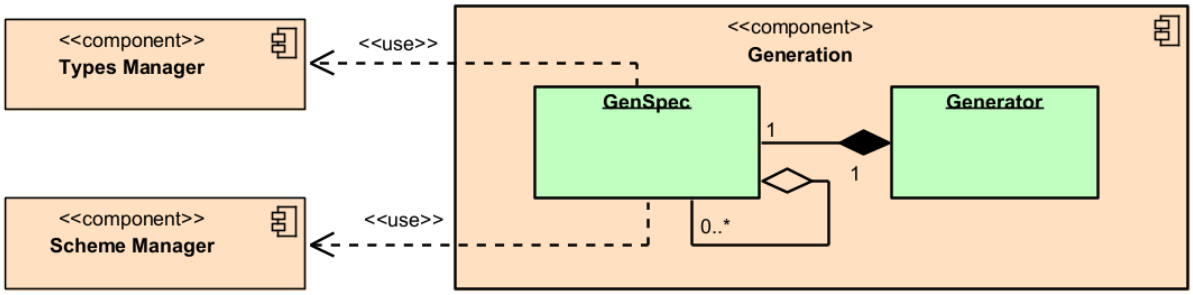

Рис. 9: Внутреннее представление компоненты `Generation`

Генерация работает следующим образом. Инициализируется `StringBuilder`, с помощью которого на основе информации из `GenSpec` строится строка, являющаяся содержимым генерируемого файла. Эта строка содержит в себе название пакета, импорты, "шапку" класса или интерфейса, поля, конструкторы и методы. В итоге, она записывается в файл, путь к которому содержится в `GenSpec`. Если этого файла не существует, то он создается.

Перед разработкой генератора были рассмотрены разные инструменты для написания кода (например, библиотека генерации кода JavaPoet [44]). Однако эти генераторы были слишком усложнены для задачи, либо не обладали достаточной гибкостью. Использовать `StringBuilder` оказалось самым простым и подходящим решением, так как он предоставлял все необходимое для формирования содержимого файла. Также `StringBuilder` подошел для решения задачи, описанной далее. Это тоже стало одним из решающих факторов выбора такого подхода.

Обычно генераторы кода создают файлы, в которые нельзя дописывать код вручную. Причина заключается в том, что при перегенерации, которую порой необходимо производить, весь файл перезаписывается и дописанный код затирается. Разработанный в рамках этой работы генератор решает эту проблему, реализуя шаги, представленные ниже.

1. В начале генератор проверяет, существует ли уже файл на диске.

2. Если файла не существует, то он генерируется. При этом делается разметка кода, приводя к разбиению файла на зоны. Внутри одной группы этих зон в последующем производится перегенерация, поэтому в них нельзя писать код (например, зона сгенерированных методов, зона сгенерированных полей и пр.). В других зонах программист может писать код, они не перегенерируется.

3. Если файл существует, генератор по маркерам распознает зоны, в которых необходимо произвести перегенерацию. Зоны перегенерации подменяются на новые, остальные остаются неизменными.

4. Обновленный файл получается конкатенацией содержимого генерируемых и не генерируемых зон.

Таким образом, реализованная подсистема получилась удобной, поскольку программист может реализовать дополнительную логику для определенных узлов PSI-дерева. Особенно эта возможность хорошо себя показала для класса фабрики PSI-узлов, куда было добавлено множество методов, написанных вручную.

В итоге, на основе подсистемы генерации были описаны и сгенерированы внешние интерфейсы для работы с PSI в Java IDE и Python IDE. На рисунке 10 можно увидеть, как на основе спецификации типов синтаксических конструкций языка получился интерфейс. Тут же можно увидеть разделение кода на зоны, где происходит или не происходит генерация.

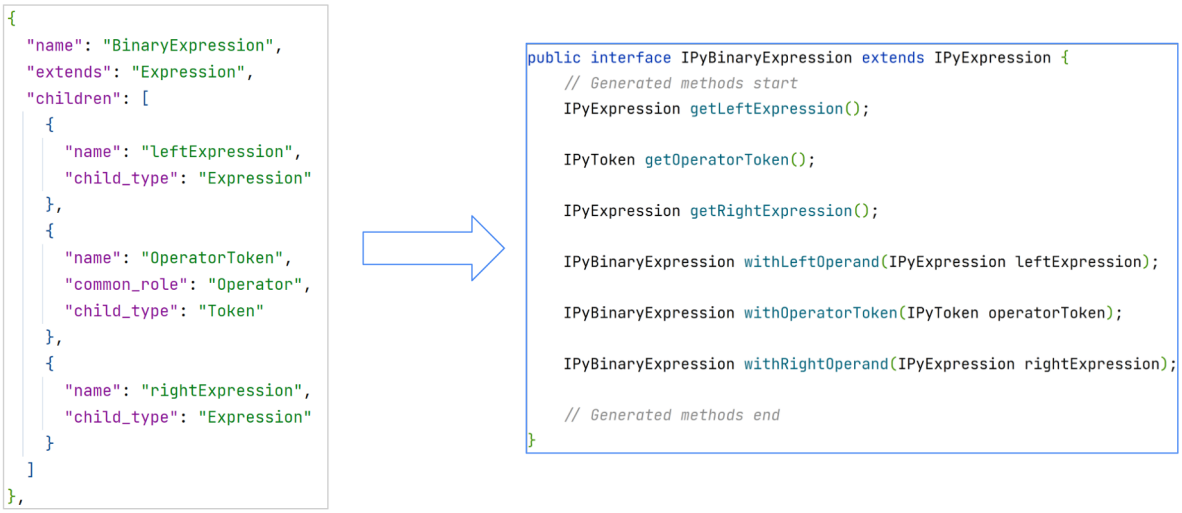

Рис. 10: Пример генерации интерфейса на основе спецификации

5.2. Модификация PSI-дерева и исходного документа

Данная подсистема представляет собой механизм, который работает при вызове методов внешнего интерфейса реализуемой подсистемы, которые описаны в разделе 4.

Как было указано в разделе 2, за основу был взят подход от Microsoft Roslyn с красными и зелеными деревьями [34]. Однако он был значительно изменен с целью адаптации под потребности разработчиков внешних сервисов, а также с учетом особенностей кодовой модели платформы.

5.2.1. Красные и зеленые узлы в кодовой модели

Как рассказывалось в разделе 2, зеленое дерево является деталью реализации, с которой клиент не может взаимодействовать. Также в нем описывалась OCS-подсистема, предназначенная для сжатия Java-объектов, которая позволяет уменьшить потребление памяти и увеличить скорость обращения к объектам за счет меньшего количества кэш-промахов.

Зеленый узел описывается интерфейсом `IASTNode`, который способен возвращать различные целочисленные значения (идентификатор типа конструкции узла, идентификатор роли) или строковые значения (строковое значение внутри узла, если это, например, токен). Это объект, который можно положить в OCS. Так, после синтаксического разбора зеленые узлы дерева располагаются в колонках OCS. Объекты, которые лежат в колонках, описываются классом `ASTColumnNode` и реализуют интерфейс `IASTNode`.

Как указывалось в разделе 2, красное дерево является фасадом над зеленым деревом. В действительности, это и есть PSI-дерево. Узел красного дерева реализуется интерфейсом `PSINode`. Интерфейсы и классы, которые являются подтипами `PSINode`, генерируются. Процесс генерации был описан в предыдущей секции. Это дерево хранит позицию в документе, которая высчитывается на основе позиции родителя и длин

зеленых узлов. Также красный узел имеет зеленый узел, поверх которого он построен. При этом красному узлу неизвестно где находится его зеленый узел: в памяти или в OCS.

Все, что описано в данной секции, реализовано внутри компоненты `PSI Modification`. При этом подтипы `PSINode` предоставляют методы модификации, что на рисунке 7 в разделе 4 описывается интерфейсом `Modification Methods`. На рисунке 11 представлена UML-диаграмма, которая уточняет диаграмму на рисунке 7 в контексте деталей реализации, описанной в текущей секции.

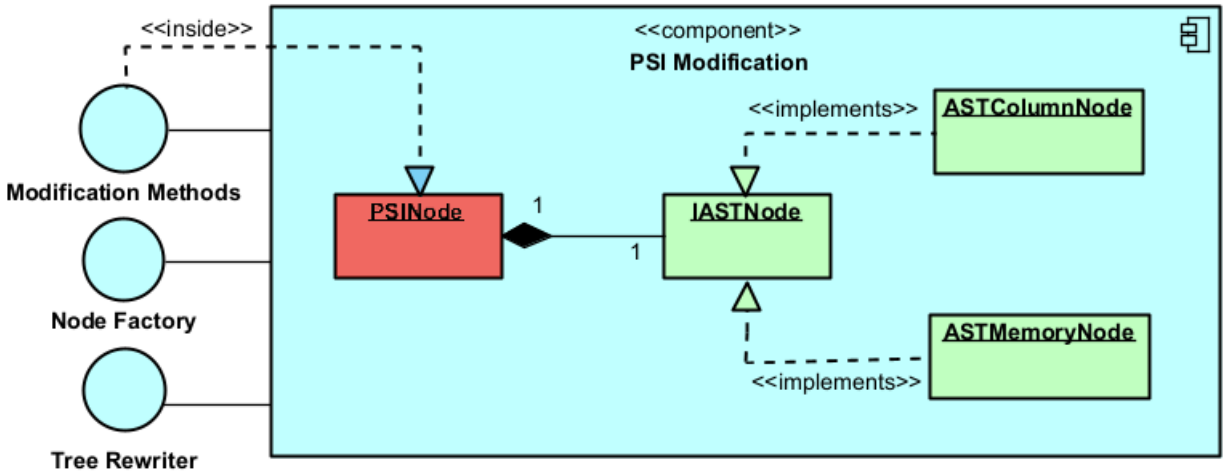

Рис. 11: Компонента `PSI Modification`: красные и зеленые узлы

Данные, которые лежат в колонках, не подлежат модификации. Поэтому, при преобразовании дерева и создании новых узлов, появляются новые зеленые узлы, которые лежат уже памяти. Они описываются классом `ASTMemoryNode`. При этом были реализованы два подхода к тому, как восстанавливать в узлы в памяти.

1. Восстанавливать в памяти полное дерево от корня. Это оказалось необходимо для Java IDE. Дело в том, что кодовая модель использует парсер из Eclipse JDT [31] и переводит получившиеся JDT узлы в `ASTColumnNode`. Проблема в том, что JDT дерево не является Lossless Syntax Tree (см. раздел 2) — оно не хранит тривиальные узлы и токены. Разработчики Java IDE решили, что им не нужно хранить в колонке токены и тривиальные узлы. Поэтому был реализован конвертер, который преобразовывает узлы

`ASTColumnNode` в `ASTMemoryNode` в случае, если внешний сервис решит модифицировать PSI. При этом происходит развешивание токенов и тривиальных узлов. На рисунке 12 можно увидеть пример работы этого механизма при модификации бинарного выражения. Можно заметить, что после модификации происходит повторное использование старых узлов с целью экономии памяти.

2. Восстанавливать в памяти только те узлы, которые подверглись модификации. Этот подход подошел при разработке Python IDE, так как парсер для Python[3], реализованный в рамках кодовой модели, уже развешивает токены и тривиальные узлы, которые тоже хранятся в колонках OCS. На рисунке 13 можно увидеть пример работы этого механизма при модификации бинарного выражения.

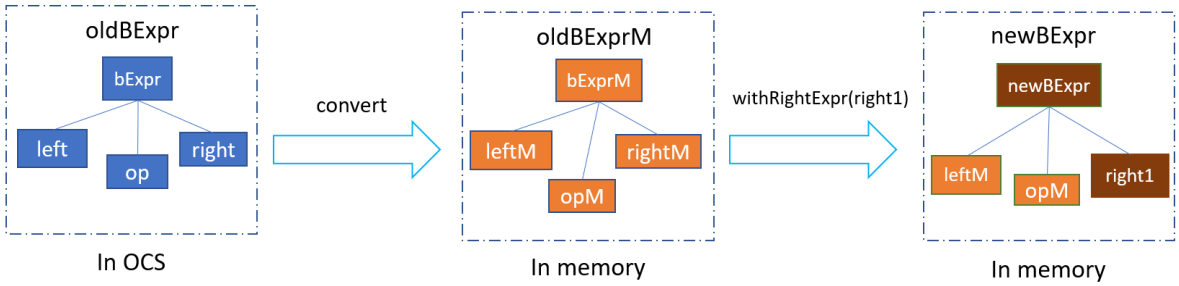

Рис. 12: Зеленые узлы при модификации бинарного выражения в кодовой модели Java IDE

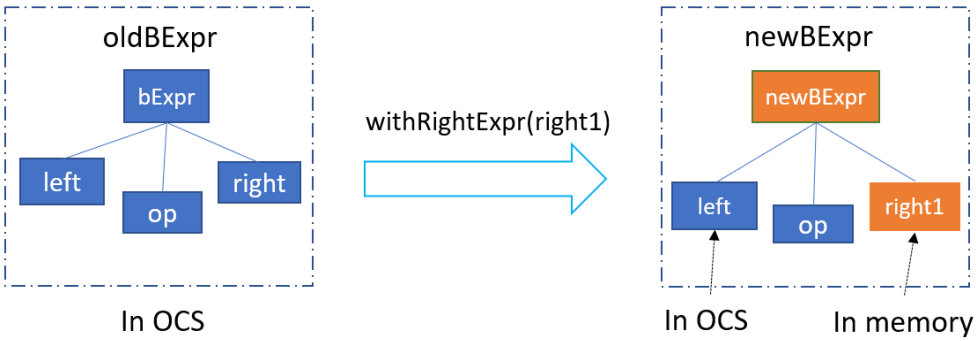

Рис. 13: Зеленые узлы при модификации бинарного выражения в кодовой модели Python IDE

[3]Стоит отметить, что данный парсер был разработан студентом 4 курса Математико-Механического факультета Владиславом Мирошниковым.

В итоге, получилось два механизма работы с зелеными узлами, которые можно использовать в зависимости от того, какие технические решения принимаются при разработке кодовой модели IDE.

5.2.2. Модификация и построение новых узлов

В данной секции описывается, как работают методы модификации узлов и методы фабрики. Это интерфейсы `Modification Methods` и `Node Factory`, которые предоставляются компонентой `PSI Modification` из диаграммы компонентов на рисунке 7.

Следует отметить, что каждый метод модификации узла вызывает метод фабрики, куда он в качестве параметра подставляет нового ребенка. В листинге 11 представлен пример метода для модификации правого операнда у бинарного выражения в Python PSI. Этот метод генерируется при помощи механизма, описанный в разделе 5.1.

```
1 public IPyBinaryExpression withRightExpr(IPyExpression right) {
2     return PyNodeFactory.newBinaryExpression(
3         getLeftExpr(),
4         getOperand(),
5         right
6     );
7 }
```

Листинг 11: Пример реализации метода модификации

Рассмотрим, как работают методы фабрики. Основной задачей метода фабрики является построение из красных детей нового красного узла `PSINode`. Для этого используется объект `ASTNodeBuilder`, который реализует паттерн проектирования Builder и позволяет различными способами построить новый зеленый узел. На основе построенного зеленого узла как раз и создается необходимый красный узел. Пример использования `ASTNodeBuilder` можно увидеть на листинге 12. Здесь с помощью метода `withChild(PSINode node)` в PSI-узел добавляется новый ребенок. Метод `build()` выполняет упаковку этих детей в новый зеленый узел, на основе которого создается также новый красный узел.

```
1  public IPyBinaryExpression newBinaryExpression(
2          IPyExpression leftExpr,
3          IPyToken operator,
4          IPyExpression rightExpr
5      ) {
6      new PyASTNodeBuilder().
7              withChild(leftExpr).
8              withChild(operator).
9              withChild(rightExpr).
10             build();
11 }
```

Листинг 12: Пример реализации метода модификации

5.2.3. Пробелы и комментарии

В разделе 2.2.1 рассказывалось про то, что PSI является Lossless Syntax Tree (LST). Одно из важных свойств LST заключается в том, что оно хранит информацию о пробелах и комментариях в тривиальных узлах (Trivia Nodes).

Microsoft Roslyn ввели специальный инвариант для работы с тривиальными узлами: тривиальные узлы являются детьми токенов и только их. Таким образом, у токена могут быть среди детей предшествующие (leading) и последующие (trailing) тривиальные узлы. При этом токены можно модифицировать, при необходимости добавляя новые тривиальные узлы. У такого подхода есть преимущества: некоторые компоненты, вроде Rewriter, реализовать проще, потому что нет никакой необходимости учитывать тривиальные узлы, так как они ”спрятаны” под токенами и их размещение является ответственностью разработчика внешнего сервиса.

Такой подход оказался неприменим в рамках разрабатываемой платформы по следующим причинам.

- Разработчикам внешних сервисов требовалось проектировать разную логику работы с тривиальными узлами в зависимости от того, являются ли они предшествующими или последующими. Так, это имело значение для сервиса по автоматическому дополнению кода, которому важно определить ближайшие нетривиальные узлы для корректной работы.

- Вследствие того, что у токенов появляются предшествующие тривиальные узлы, у любого PSI-узла приходится рассматривать два типа позиций в документе: основную и ту, откуда начинается самый крайний слева тривиальный узел. Также приходится рассматривать и разные длины у узла: с учетом тривиальный узлов по краям и без. Это приводит к сложному API, с которым неудобно работать разработчикам внешних сервисов.

- При таком подходе к тривиальным узлам разработчикам рефакторингов и быстрых исправлений приходилось думать о том, как правильно их размещать. Например, в методах фабрики появлялись параметры с токенами, в которых требовалось устанавливать токены корректно размещенными пробелами. Таким образом, было принято решение, что именно подсистема модификации PSI-дерева должна отвечать за размещение тривиальных узлов, а не подсистемы рефакторингов и быстрых исправлений.

- Ни реализованный в подсистеме модели кода парсер языка Python, ни парсер языка Java от JDT, не поддерживают этот инвариант.

В результате пришлось отказаться от подхода Microsoft Roslyn. На рисунке 14 слева показано, как размещались тривиальные узлы в Microsoft Roslyn, а справа — как это было сделано в реализованной подсистеме. Был выбран вариант это сделать более классическим образом — размещать тривиальные узлы на уровне токенов.

Теперь подсистема модификации берет на себя правильное размещение тривиальных узлов. Это необходимо делать даже при построении нового зеленого узла в методе фабрики. Если этого не делать, то, напри-

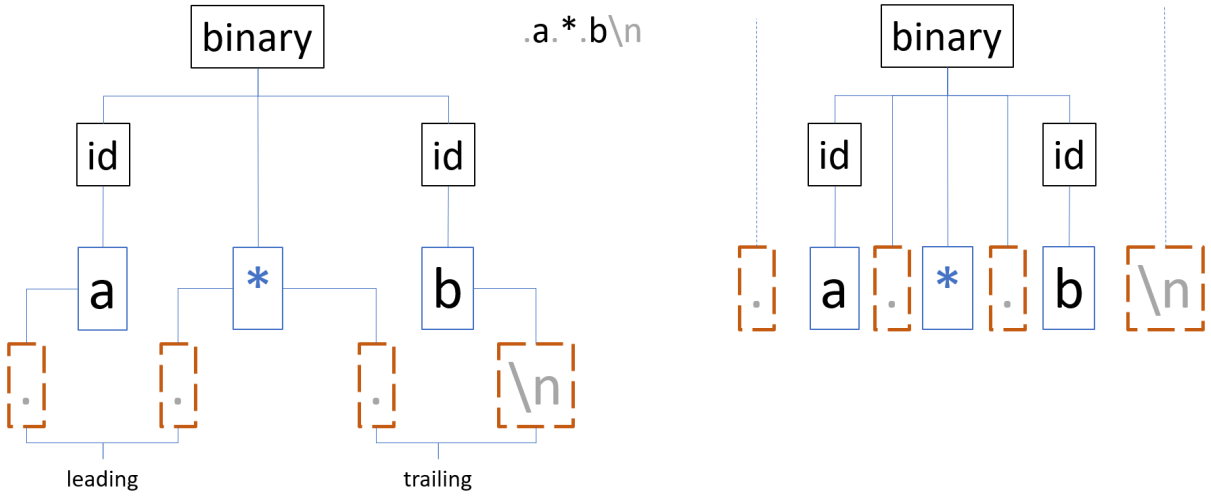

Рис. 14: Тривиальные узлы в Roslyn и в реализованной подсистеме

мер, при построении новой функции для Python не поставится пробел после ключевого слова `def`, что приведет к синтаксической ошибке.

Был реализован нормализатор пробелов, который берет узел и расставляет тривиальные узлы вокруг его детей. Нормализатору заранее определяются правила, по которым он обрабатывает детей. Эти правила включают в себя следующую информацию:

- тип ребенка, который требуется обработать;
- способ добавления тривиальных узлов (вокруг ребенка, до или после);
- информацию для ответа на вопрос о том, какие тривиальные узлы нужно расставить (пробел или, например, переход на новую строку).

Важно отметить, что этот нормализатор представляет собой декоратор дерева (см. раздел 4) — функциональный интерфейс, который принимает PSI-узел и возвращает обработанный PSI-узел и реализует паттерн проектирования Decorator.

5.2.4. Tree Rewriter

Как описывалось в разделе 4, Tree Rewriter — объект, который предоставляется компонентой `PSI Modification` для переписывания всего дерева. Через методы `replace` и `remove` он накапливает действия, которые необходимо совершить над деревом и совершает их при вызове метода `rewrite`. Рассмотрим более подробно принцип его работы.

Объект `Rewriter` оперирует двумя структурами данных — словарем замен
(Replace Map) и списком узлов на удаление (Remove List). Словарь замен представляет собой хеш-таблицу, в которой хранятся данные о том, какие узлы надо заменить на какие. Именно она заполняется при вызове метода `replace`. Список на удаление — обычный список, который хранит в себе узлы, подлежащие удалению. Он заполняется во время вызова метода `remove`.

Объект `Rewriter` работает следующим образом. Он обрабатывает все PSI-дерево через прямой обход (Preorder Traversal). Это значит, что обработке подлежит сначала сам PSI-узел, а потом его дети слева направо. При обработке узла происходит проверка отображения замен на наличие узла там. Если узел найден, он заменяется. Если нет, то активируется проход по его детям. При этом при помощи информации в спецификации узла определяется, можно ли произвести замену. Если можно, то замена производится. В противном случае бросается Java-исключение.

При проходе по детям формируется новый список детей, на основе которого будет сформирован новый узел. Однако, если после прохода список не поменялся, новый родитель не формируется, потому что в этом нет необходимости. Если при проходе по детям очередной ребенок лежит в списке на удаление — он не кладется в новый список детей. При этом проверяется, может ли узел быть удален, при помощи проверки свойства `is_mandatory`, которое описывается в спецификации.

Важной особенностью является то, что в при этом выполняется обход не всего PSI-дерева, а только некоторой его части. Методы `replace`

и `remove` не только заполняют соответствующие структуры данных, но и запоминают области, которые занимают узлы в исходном документе. Область представляет собой пару из двух позиций в документе, где начинается и заканчивается узел. Если обозримый во время обхода узел не пересекается ни с одной из сохраненных областей, то его нет смысла рассматривать, так как у него точно нет подлежащих замене или удалению потомков, а потому он пропускается.

В Microsoft Roslyn тоже реализован объект`Rewriter`, но другим способом. Он генерируется, и при обходе вызывает методы фабрики для перестраивания дерева. Это дает гарантию того, что переписывание дерева не приведет к новым синтаксическим ошибкам в исходном коде. В рамках разрабатываемой системы такой подход оказался не применим из-за измененного способа работы с пробелами и комментариями (см. раздел 5.2.3). `Rewriter`, работающий таким образом, нарушает пользовательское форматирование кода из-за нормализации пробелов, что приводит к нежелательному поведению рефакторингов и быстрых исправлений с точки зрения пользователя IDE.

У объекта `Rewriter` есть дополнительный метод `rewrite`, который принимает набор декораторов (экземпляров класса `PSIDecorator`), которые применяются к переписанному дереву после отработки объекта `Rewriter`. Это необходимо для того, чтобы разработчики внешних сервисов имели возможность производить над деревом дополнительные действия, специфичные для конкретного языка программирования. Так, для языка Python был реализован декоратор `IndentFixer`, который отвечает за исправление отступов после переписывания дерева. Он необходим, так как отступы играют важную роль в синтаксисе языка Python.

5.2.5. Преобразование исходного документа

Компонента `Program Text Modification` (см. раздел 4) отвечает за получение последовательности текстовых изменений, которые можно применить к исходному документу для того, чтобы получить его обновленную версию. Для клиентов предоставляется интерфейс `Tree Differ`, с помощью которого можно на основе двух версий PSI-дерева получить неупорядоченную последовательность текстовых изменений. Эту последовательность сервис может применить к исходному документу, чтобы изменения в PSI-дереве стали видны пользователю IDE. В этом разделе описан принцип работы этой компоненты.

Реализованы следующие три вида текстовых изменений:

- вставка текста(add);
- удаление текста (delete);
- замена текста (replace).

Эти же виды текстовых изменений определены также и в Language Server Protocol [15], на стороне которого реализуются сервисы рефакторинга и быстрых исправлений.

Текстовое изменение реализуется объектом, который содержит следующую информацию.

- Позицию в документе, в которой изменение начинается.
- Позицию в документе, в которой изменение заканчивается. В случае добавления эта позиция равна позиции начала данного изменения.
- Текст, на который необходимо заменить область в документе. В случае удаления эта строка является пустой.

Алгоритм, описанный ниже, способен определять и перемещения поддеревьев (move). Такой тип изменения эквивалентен последовательности из удаления и добавления поддеревьев, благодаря чему достигается совместимость с LSP.

В разделе 2.3 в качестве подхода для получения текстовых изменений был выбран алгоритм GumTree [40]. Ему на вход подаются два PSI-дерева. GumTree работает в два этапа.

1. Устанавливаются отображения (mappings) между узлами первого и второго дерева. На рисунке 15 показан пример из статьи [40], где установлены отображения для двух деревьях, построенных по участкам исходного кода на языке Java.

2. Анализ построенных отображений и построение последовательности текстовых изменений.

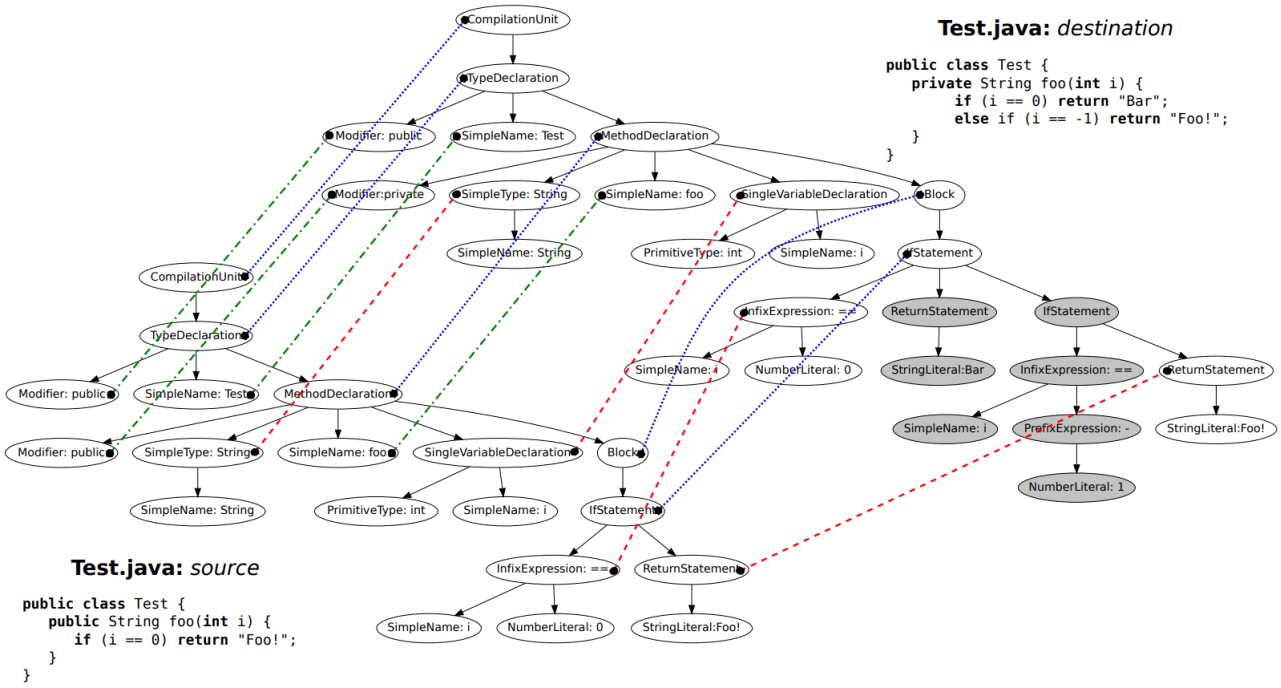

Рис. 15: Примеры построенных отображений при работе GumTree

Установление отображений происходит в два этапа.

1. Жадный алгоритм нахождения изоморфных деревьев возрастающей высоты. Между изоморфными деревьями как раз и устанавливается отображение. Изоморфность деревьев определяется следующим образом. Для поддеревьев лениво высчитывается хеш-код. Алгоритм вычисления и хранения хеш-кодов описан в статье [45]. Эти хеш-коды находятся в B-дереве, что позволяет более быстро их получать. Таким образом, определение изоморфности двух деревьев имеет сложность $O(1)$, представляя собой сравнение хеш-кодов. Отображения, найденные на этом этапе, называются якори.

2. Алгоритм насыщения множества отображений. Вводится новое правило для определения новых отображений — один узел отображается в другой, если их потомки включают в себя большое количество якорей. По этому правилу в результате анализа дерева снизу вверх определяются новые отображения.

Этап генерации кратчайшей последовательности текстовых изменений более подробно описан в статье [38]. Обходятся оба дерева и, в зависимости от построенных отображений, строится последовательность текстовых изменений. Отображения анализируются в несколько фаз, в каждой из которых определяются типы элементов последовательности текстовых изменений. При этом учитывается особенность Language Server Protocol, в котором нельзя полагаться на порядок текстовых изменений, так как они происходят одновременно. Поэтому и алгоритм требовалось адаптировать так, чтобы последовательность была неупорядоченной. Например, несколько добавлений подряд в одной области склеиваются в один.

Алгоритм построен так, чтобы его можно было использовать для PSI-деревьев разных языков. Сама компонента не знает, деревья какого языка программирования она анализирует.

6. Апробация системы

Данный раздел ставит своей целью показать, насколько разработанная подсистема универсальна и удобна в рамках платформы SRC IDE.

6.1. Метрика

Java IDE и и Python IDE, построенные на платформе SRC IDE, образуют семейство программных продуктов (software product line), разрабатываемое на основе повторно используемых активов [46].

В рамках данной работы было создано два повторно используемых актива — генератор внешнего интерфейса и подсистема модификации PSI и исходного документа.

При использовании этих подсистем разными продуктами семейства происходило их улучшение: исправлялись ошибки, добавлялись новые возможности. В работе [47] такой процесс называют насыщением повторно используемых активов при разработке различных продуктов семейства.

Каждое улучшение актива происходит по запросу от клиентов, которые пользуются созданной системой. В контексте текущей работы клиенты — это либо разработчики, которые пользуются системой в рамках разработки таких сервисов, как рефакторинги и быстрые исправления, либо разработчики из команды кодовой модели, которые используют генератор при разработке новой IDE. Если созданные повторно используемые активы достаточно хорошо выполняют свою задачу, то с разработкой нового продукта семейства количество запросов на улучшение должно уменьшаться. Это будет маркером того, что подсистема стала более пригодной для использования в условиях разработки новых продуктов.

Поэтому, в качестве метрики будет использоваться количество запросов на улучшение созданных повторно используемых активов.

6.2. Использование в рамках Python IDE

Была создана спецификация для генератора, и на ее основе были сгенерированы типы для PSI-дерева языка Python. Помимо этого были сгенерированы фабрика, Visitor для обхода дерева, и другие вспомогательные Java-файлы для работы с PSI для Python.

Реализуемая подсистема активно использовалась командой разработки рефакторингов и быстрых исправлений для Python IDE. За время использование системы были успешно разработаны разные сервисы по упрощению и модификации исходного кода. Ниже представлены некоторые из них.

- Rearrange Code — рефакторинг, который позволяет переставлять местами конструкции программы. Благодаря этому сервису можно быстро перемещать по исходному коду выделенные функции и классы, менять аргументы функций местами. Также имеется возможность выносить функции из классов во внешнюю область видимости в случае, если функция является самой верхней или самой нижней в классе.

- Introduce Variable — рефакторинг, который позволяет на основе выбранного выражения определить новую переменную, в которую оно будет присвоено.

- Min Max If — быстрое исправление, которое позволяет превратить конструкцию вида `if a < b: return a else return b` в `return min(a, b)`. Стоит отметить, что на момент реализации системы в PyCharm (JetBrains) [48] такого быстрого исправления не было.

- Annotated Assignment — быстрое исправление, которое позволяет убрать аннотацию в случае цепного присваивания. (Например, `a: int = b = d = 3` превращается `a = b = d = 3`). Это быстрое исправление необходимо, потому что в Python типовая аннотация в цепном присваивании является синтаксической ошибкой. При этом на момент разработки системы PyCharm хоть и показывал ошибку, но не предлагал быстрого исправления.

В результате использования системы в рамках разработки Python IDE было выполнено 21 улучшение созданных повторно используемых активов по запросу от разработчиков.

6.3. Использование системы в рамках Java IDE

Как и в случае Python IDE, были сгенерированы Java-файлы для работы с реализованной подсистемой.

Далее подсистема модификации PSI и исходного документа использовалась командой разработки рефакторингов и быстрых исправлений для Java IDE. Ниже представлены некоторые разработанные сервисы по изменению структуры исходного кода.

- Remove Empty Finally Block — быстрое исправление, которое позволяет удалить пустой finally-блок у конструкции try catch. Оно позволяет уменьшить количество потенциальных ошибок в программе.

- Replace For Loop With While Loop — быстрое исправление, которое позволяет превратить цикл for в цикл while в случае, если у цикла for не указана часть для инициализации и обновления переменных. Оно необходимо для упрощения читаемости исходного кода программы.

- Simplify Trivial If — быстрое исправление, которое позволяет заменить if statement с возвращением true или false в зависимости от выполнения условия на return с проверкой этого условия. Оно позволяет улучшить читаемость исходного кода программы.

В результате использования предложенной подсистемы в рамках разработки Java IDE было выполнено 7 улучшений созданных повторно используемых активов по запросу от разработчиков.

6.4. Выводы

По итогам апробации можно сделать следующие выводы.

- Подсистема генерации была успешно использована как общий актив в рамках таких продуктов, как Java IDE и Python IDE.
- Подсистема модификации PSI и исходного документа также показала достойные результаты как общий актив в рамках Java IDE и Python IDE. На ее основе команды разработки рефакторингов и быстрых исправлений смогли выполнить необходимые им задачи в рамках реализации новых возможностей для платформы SRC IDE.
- При использовании предложенной системы в новых продуктах, основанных на платформе SRC IDE, количество запросов на изменение уменьшалось, что говорит об успешном наращивании созданных общих активов и их достаточной универсальности.

Следует отметить, что программисты из команд разработки Java IDE и Python IDE дали позитивные отзывы об использовании созданной подсистемы.

Заключение

В ходе выпускной квалификационной работы были достигнуты следующие результаты.

- Выполнен обзор предметной области:
  - изучена текущая платформа SRC IDE;
  - рассмотрены механизмы модификации PSI-дерева: изменяемое дерево, неизменяемое дерево через Rewriter, а также персистентное дерево с Rewriter, разработанное Microsoft, которое и было выбрано;
  - рассмотрены подходы для получения текстовых изменений: RTED, ChangeDistiller, GumTree. Был выбран подход GumTree с учетом особенностей SRC IDE.
- Сформулированы функциональные и нефункциональные требования к подсистеме, в рамках которых производились проектирование архитектуры и реализация.
- Спроектирована архитектура генератора классов и интерфейсов, механизма преобразований над PSI-деревом и исходным документом (Java/IntelliJ IDEA).
- Реализована генерация необходимых интерфейсов и классов языка Java для системы модификации PSI-дерева на основе спецификации в формате JSON.
- Реализованы механизм модификации дерева в виде персистентного дерева и Rewriter-а и механизм получения текстовых изменений в виде алгоритма GumTree.
- Выполнена апробация в рамках написания подсистем рефакторингов и исправлений кода в средах разработки для Java и Python.

- Результаты данной работы были приняты для публикации в Proceedings of ISP RAS и для защиты на конференции SYRCoSE 2023 (г. Пенза).

Благодарности

Автор выражает отдельную благодарность коллегам, помогавшим ему при выполнении данной работы.

- Захарову Александру Александровичу — за всестороннюю консультацию по всем вопросам по ходу разработки подсистемы.
- Тропину Николаю Владимировичу — за внимательную проверку как написанного автором исходного кода, так и текста работы.
- Лукьяновой Ольге Евгеньевне — за то, что, как лидер команды, позволила автору взяться за реализацию этой не тривиальной подсистемы и представить результат в виде выпускной квалификационной работы

Также, автор благодарит своего научного руководителя Кознова Дмитрия Владимировича за помощь при создании текста с презентацией и в проектировании выпускной работы по выполненной производственной задаче, а также за советы по апробации работы, Литвинова Юрия Викторовича за проверку текста и контроль прогресса работы в течение последнего этапа выполнения работы.

Список литературы

[1] IntelliJ IDEA. — URL: https://www.jetbrains.com/idea/ (дата обращения: 27 марта 2023 г.).

[2] Visual Studio. — URL: https://visualstudio.microsoft.com/ (дата обращения: 27 марта 2023 г.).

[3] Refactorings. — URL: https://www.jetbrains.com/help/idea/tutorial-introduction-to-refactoring.html (дата обращения: 27 марта 2023 г.).

[4] Quick Fixes. — URL: https://www.jetbrains.com/help/rider/Code_Analysis__Quick-Fixes.html (дата обращения: 27 марта 2023 г.).

[5] Rename Refactoring. — URL: https://www.jetbrains.com/help/idea/rename-refactorings.html (дата обращения: 27 марта 2023 г.).

[6] Extract Method Refactoring. — URL: https://www.jetbrains.com/help/idea/extract-method.html (дата обращения: 27 марта 2023 г.).

[7] Alfred V. Aho Ravi. Sethi Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. — 1986. — P. 69–70.

[8] Program Structure Interface. — URL: https://plugins.jetbrains.com/docs/intellij/psi.html (дата обращения: 27 марта 2023 г.).

[9] The IntelliJ Platform: A Framework for Building Plugins and Mining Software Data / Zarina Kurbatova, Yaroslav Golubev, Vladimir Kovalenko, Timofey Bryksin // 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW). — 2021. — P. 14–17.

[10] IntelliJ Community Github: PSI. — URL: https://github.com/JetBrains/intellij-community/tree/master/java/

java-psi-api/src/com/intellij/psi (дата обращения: 27 марта 2023 г.).

[11] Protocol Buffers. — URL: https://protobuf.dev/ (дата обращения: 26 апреля 2023 г.).

[12] FlatBuffers. — URL: https://github.com/google/flatbuffers (дата обращения: 26 апреля 2023 г.).

[13] Spring Data JPA. — URL: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/ (дата обращения: 29 апреля 2023 г.).

[14] Microsoft Visual Studio Code. — URL: https://code.visualstudio.com/ (дата обращения: 25 апреля 2023 г.).

[15] Language Server Protocol Overview. — URL: https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/ (дата обращения: 18 декабря 2022 г.).

[16] Codenvy. — URL: https://github.com/codenvy/codenvy (дата обращения: 25 апреля 2023 г.).

[17] Red Hat. — URL: https://www.redhat.com/en (дата обращения: 25 апреля 2023 г.).

[18] Eclipse Che. — URL: https://www.eclipse.org/che/ (дата обращения: 25 апреля 2023 г.).

[19] Eclipse Cloud Development. — URL: https://ecdtools.eclipse.org/ (дата обращения: 25 апреля 2023 г.).

[20] Java support for Visual Studio Code. — URL: https://github.com/redhat-developer/vscode-java (дата обращения: 25 апреля 2023 г.).

[21] Eclipse JDT Language Server. — URL: https://github.com/eclipse/eclipse.jdt.ls (дата обращения: 25 апреля 2023 г.).

[22] IntelliJ Platform GitHub. — URL: https://github.com/JetBrains/intellij-community (дата обращения: 25 апреля 2023 г.).

[23] JetBrains Fleet. — URL: https://www.jetbrains.com/fleet/#distributive (дата обращения: 25 апреля 2023 г.).

[24] Language Servers. — URL: https://microsoft.github.io/language-server-protocol/implementors/servers/ (дата обращения: 18 декабря 2022 г.).

[25] Language Server Protocol: Code Action Request. — URL: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#textDocument_codeAction (дата обращения: 14 апреля 2023 г.).

[26] Retaining comments when refactoring code / Peter Sommerlad, Guido Zgraggen, Thomas Corbat, Lukas Felber. — 2008. — 01. — P. 653–662.

[27] Overbey Jeffrey L. Immutable Source-Mapped Abstract Syntax Tree: A Design Pattern for Refactoring Engine APIs // Proceedings of the 20th Conference on Pattern Languages of Programs. — PLoP '13. — USA : The Hillside Group, 2013. — 8 p.

[28] Roberts Don, Brant John, Johnson Ralph. A Refactoring Tool for Smalltalk. // TAPOS. — 1997. — 01. — Vol. 3. — P. 253–263.

[29] Garrido Alejandra. Program Refactoring in the Presence of Preprocessor Directives : Ph. D. thesis / Alejandra Garrido. — USA : University of Illinois at Urbana-Champaign, 2005. — AAI3199001.

[30] IntelliJ Platform SDK — Modifying the PSI. — URL: https://plugins.jetbrains.com/docs/intellij/modifying-psi.html (дата обращения: 18 декабря 2022 г.).

[31] Eclipse Java development tools (JDT). — URL: https://www.eclipse.org/jdt/ (дата обращения: 18 декабря 2022 г.).

[32] C/C++ development tools (CDT). — URL: https://projects.eclipse.org/projects/tools.cdt (дата обращения: 18 декабря 2022 г.).

[33] Roslyn GitHub. — URL: https://github.com/dotnet/roslyn (дата обращения: 18 декабря 2022 г.).

[34] Persistence, Facades and Roslyn's Red-Green Trees. — URL: https://learn.microsoft.com/en-us/archive/blogs/ericlippert/persistence-facades-and-roslyns-red-green-trees (дата обращения: 18 декабря 2022 г.).

[35] GNU Diffutils. — URL: https://www.gnu.org/software/diffutils/ (дата обращения: 5 апреля 2023 г.).

[36] Bille Philip. A survey on tree edit distance and related problems // Theoretical Computer Science. — 2005. — Vol. 337, no. 1. — P. 217–239. — URL: https://www.sciencedirect.com/science/article/pii/S0304397505000174.

[37] Pawlik Mateusz, Augsten Nikolaus. RTED: A Robust Algorithm for the Tree Edit Distance. — 2011. — 1201.0230.

[38] Change Detection in Hierarchically Structured Information / Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, Jennifer Widom // Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data. — SIGMOD '96. — New York, NY, USA : Association for Computing Machinery, 1996. — P. 493–504. — URL: https://doi.org/10.1145/233269.233366.

[39] Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction / Beat Fluri, Michael Wursch, Martin PInzger, Harald Gall // IEEE Transactions on Software Engineering. — 2007. — Vol. 33, no. 11. — P. 725–743.

[40] Fine-Grained and Accurate Source Code Differencing / Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc et al. // Proceedings of the 29th

ACM/IEEE International Conference on Automated Software Engineering. — ASE '14. — New York, NY, USA : Association for Computing Machinery, 2014. — P. 313–324. — URL: https://doi.org/10.1145/2642937.2642982.

[41] GumTree Tool. — URL: https://github.com/GumTreeDiff/gumtree (дата обращения: 5 апреля 2023 г.).

[42] GSON Library. — URL: https://github.com/google/gson (дата обращения: 4 апреля 2023 г.).

[43] IntelliJ Platform Python PSI. — URL: https://github.com/JetBrains/intellij-community/tree/master/python/python-psi-api/src/com/jetbrains/python/psi (дата обращения: 4 апреля 2023 г.).

[44] JavaPoet Library. — URL: https://github.com/square/javapoet (дата обращения: 4 апреля 2023 г.).

[45] Chilowicz Michel, Duris Etienne, Roussel Gilles. Syntax tree fingerprinting for source code similarity detection // 2009 IEEE 17th International Conference on Program Comprehension. — 2009. — P. 243–247.

[46] A Framework for Software Product Line Practice, version 5.0. — Software Engineering Institute, 2006. — . — URL: https://doi.org/10.1145/2642937.2642982.

[47] Попова Т.Н. Кознов Д.В. Тинова А.А. Романовский К.Ю. Эволюция общих активов в семействе средств реинжиниринга программного обеспечения // Системное программирование, 1. — 2005. — P. 184–198.

[48] JetBrains PyCharm. — URL: https://www.jetbrains.com/pycharm/ (дата обращения: 4 апреля 2023 г.).