Санкт-Петербургский государственный университет
Математическое обеспечение и администрирование информационных систем

Системное программирование

Соловьев Александр Александрович

Визуализация осуществлённых рефакторингов в IDE на основе IntelliJ Platform

Выпускная квалификационная работа

Научный руководитель:
к.т.н., доцент кафедры СП Брыксин Т.А.

Рецензент:
Заместитель начальника отдела разработки ПО СОРМ ООО “НТЦ Протей” Зимин А.С.

Санкт-Петербург
2020

SAINT-PETERSBURG STATE UNIVERSITY
Software and Administration of Information Systems

Software Engineering

Solovev Aleksandr

Visualization of applied refactorings in IDEs based on IntelliJ Platform

Graduation Thesis

Scientific supervisor:
Candidate of Engineering Sciences Timofey Bryksin

Reviewer:
Deputy Head of Software Development SORM PROTEI Co. Ltd Aleksandr Zimin

Saint-Petersburg
2020

Оглавление

Введение

Разработка программного обеспечения — длительный процесс, в ходе которого кодовая база проекта может сильно разрастаться и усложняться. Для улучшения качества исходного кода проекта проводится рефакторинг — изменение реализации некоторой функциональности, не затрагивающее при этом поведение программы [5]. Эта практика активно применяется на протяжении разработки проекта и призвана решать разные задачи: от улучшения читаемости каких-то её фрагментов до архитектурных изменений, открывающих возможности для реализации новой функциональности.

Знание о том, какие рефакторинги были проведены в репозитории, может быть использовано по-разному. Например, оно может быть полезно разработчику при рассмотрении изменений, сделанных в проекте в рамках некоторых доработок. К сожалению, полагаться на комментарии к коммитам в системах контроля версий не всегда возможно, поскольку разработчики зачастую оставляют недостаточно информативные сообщения. Поэтому интерес представляют инструменты, способные автоматически формировать список проведенных рефакторингов по некоторому репозиторию. В работах [1, 2, 6, 8, 10] были представлены разные подходы к решению этой задачи. Многие из них реализованы в качестве расширений к среде разработки Eclipse и умеют работать только с языком Java. Также с их помощью возможно просмотреть полученную на основе анализа проекта информацию в удобном для пользователя виде.

Поскольку ни один из инструментов, решающих задачу поиска рефакторингов в репозитории проекта, не был реализован для сред разработки на основе IntelliJ Platform[1], было решено выбрать наиболее подходящий для решения вышеописанной задачи подход и реализовать его для этой платформы, а также использовать полученное решение для расширения её возможностей.

[1] `https://www.jetbrains.com/ru-ru/opensource/idea/`

Постановка задачи

Целью данной работы является разработка расширения для IDE на основе IntelliJ Platform, позволяющего находить рефакторинги в истории проектов для разных языков программирования. Для достижения данной цели были поставлены следующие задачи:

- выбор инструмента для поиска рефакторингов в истории системы контроля версий, наиболее подходящего для поставленной задачи;
- проектирование модульной архитектуры решения для поддержки работы с разными языками;
- реализация библиотеки, интегрированной с IntelliJ Platform и способной детектировать рефакторинги для разных языков;
- апробация полученного решения.

1. Обзор

1.1. IntelliJ Platform

IntelliJ Platform[2] — это основа для построения IDE от компании JetBrains (IntelliJ IDEA, CLion и др.) и других разработчиков (Android Studio от Google), предоставляющая общий программный интерфейс для работы с различными языками программирования. Далее рассмотрены компоненты платформы, которые были использованы в рамках данной работы.

Program Structure Interface

Program Structure Interface (PSI) — это набор интерфейсов в IntelliJ Platform, отвечающих за разбор исходных файлов и построение по ним синтаксических и семантических моделей. Так, вне зависимости от языка для любого файла в проекте строится синтаксическое дерево, состоящее из различных PSI-элементов, все из которых реализуют интерфейс $PsiElement$. В узлах находятся различные компоненты, представляющие различные языковые конструкции вроде классов или методов, а в качестве корня всегда выступает некоторый наследник $PsiFile$.

Экземпляры $PsiElement$ содержат в себе различную информацию об элементах, которым они соответствуют. В общем случае к ней относятся данные о файлах, в которых они содержатся, о местоположении в тексте и о дочерних элементах, что в свою очередь позволяет рекурсивно обходить PSI-дерево. Элементы, представляющие конкретные сущности в языке, предоставляют специфическую для них информацию. Так, в экземплярах $PsiClass$ содержатся сведения о расширяемых ими классах и о реализуемых ими интерфейсах, а в $PsiMethod$ — о сигнатуре соответствующего метода.

[2] `https://www.jetbrains.com/ru-ru/opensource/idea/`

Git4Idea

Для работы с системой контроля версий существует встроенный в IDE модуль Git4Idea.[3] Он предоставляет интерфейс для работы с репозиторием открытого в среде разработки проекта. Для получения различной информации из его истории может быть использована компонента *GitHistoryUtils*. В частности, с помощью её метода *history* можно получить информацию о коммитах, которые были сделаны в репозитории проекта. Вышеуказанный метод способен принимать параметры команды `git log`.

Для хранения информации о коммитах предназначены экземпляры класса *GitCommit*. Наибольший интерес в рамках данной работы представляют данные о сделанных в рамках соответствующего коммита изменениях. Получить их можно в виде списка объектов *Change*, каждый из которых описывает изменения, которые были сделаны в некотором файле. *Change* позволяет получить тип изменения и пару экземпляров *ContentRevision* до и после коммита в случае, если файл был изменен или перемещен, и лишь один экземпляр, если он был создан или удалён. Каждый из полученных объектов содержит путь к соответствующему файлу и позволяет получить его содержимое в соответствующей ревизии.

Поддержка расширений

IntelliJ Platform разработана таким образом, чтобы обеспечить возможность написания и использования расширений для IDE на основе данной платформы. Так, возможно добавить поддержку некоторого непредусмотренного языка программирования, например, Rust.[4]

Для написания новых новых встраиваемых модулей в IntelliJ Platform предоставляется специальный интерфейс, позволяющий расширять возможности IDE. Однако, помимо этого, предусмотрена и возможность создания специальных точек расширения, которые могут быть добавле-

[3] urlhttps://github.com/JetBrains/intellij-community/tree/master/plugins/git4idea

[4] `https://github.com/intellij-rust/intellij-rust`

ны сторонними разработчиками в свои проекты. Это позволяет разрабатывать наборы подключаемых модулей, некоторые из которых расширяли бы поведение друг друга. Всё, что требуется для поддержки описанной функциональности, — указать интерфейс, который должен расширяться сторонними решениями, в файле-конфигурации расширения. Для взаимодействия с конкретными реализациями требуется использовать экземпляр класса $ExtensionPointName$.

1.2. Инструменты поиска рефакторингов

Данный раздел содержит описание подходов к поиску рефакторингов в истории репозитория проекта, которые рассматривались для использования в данной работе. Стоит заранее определить два параметра, которыми можно характеризовать, насколько успешно некоторый алгоритм справляется со своей задачей: точность и полнота. Точность определяет, какая доля выявленных рефакторингов была определена корректно. Полнота — какая доля проведенных рефакторингов была успешно определена.

JDEvAn

Построение списка рефакторингов в JDEvAn (Java Design Evolution Analysis) [10] происходит следующим образом. В качестве входных данных инструмент принимает две версии проекта. Для каждой из них строится представление в виде графа, в качестве узлов которого выступают пакеты, классы, поля, операции и так далее. Рёбра же соответствуют различным отношениям между данными элементами, например, отношению наследования или использования. Стоит заметить, что данные модели включают в себя информацию и о тех компонентах, которые никак не различаются в разных версиях проекта. Далее полученные представления анализируются при помощи алгоритма UMLDiff [9], который по ним определяет, какие изменения были сделаны между версиями проектов.

Минусом данного подхода является то, что для его работы требуется

построение модели для всего проекта. Значения точности и полноты, полученные в работе [9], были примерно равны 0.95.

Ref-Finder

В используемом в работе [8] подходе предлагается рассматривать программу как набор утверждений, некоторые из которых могут описывать какие-то языковые единицы, а другие — отношения между ними. Рефакторинги же можно рассматривать как некоторые логические выражения, исполнение которых означало бы, что между версиями проекта были проведены соответствующие рефакторинги. На подобных соображениях и строится работа Ref-Finder, который для пары версий проекта строит их представление при помощи предикатов, после чего последовательно проверяет, выполняются ли логические выражения, задающие рефакторинги.

Как и JDEvAn, данный подход обрабатывает полностью обе версии проекта. Полученные авторами значения точности и полноты равны 0.94 и 0.97 соответственно, однако в других работах были получены значительно отличающиеся результаты. Так, в работе [4] точность Ref-Finder была равна 0.35, а его полнота — 0.24. Связано это с тем, что значения этих показателей сильно зависят от того, на какой тестовой выборке происходит их определение, что в свою очередь означает, что на практике подобный инструмент также может нестабильно работать на разных проектах.

RefactoringCrawler

В представленном в работе [2] подходе определение списка рефакторингов происходит в два шага. Первым идёт стадия точного синтаксического анализа. На ней для старой и новой версий проекта строятся легковесные абстрактные синтаксические деревья (AST), в которых в качестве листьев выступают объявления методов и полей, для узлов же строится некоторое представление на основе шинглов [3], что позволяет сравнить соотносящиеся с сущностями в программе узлы де-

ревьев и составить список потенциальных рефакторингов. Шинглы — это такое представление строк, при котором незначительное изменение строки ведет к незначительному изменению шингла. Далее, на стадии семантического анализа для каждого из кандидатов определяется тип проведенного рефакторинга. Происходит это при помощи некоторых синтаксических проверок и подсчета вероятностей принадлежности к конкретному типу.

Как и в предыдущих случаях, RefactoringCrawler анализирует все Java-файлы проекта. Точность и полнота в посвященной данному инструменту работе была получена для трёх репозиториев и была не менее 0.90 и 0.86, соответственно.

RMiner

В отличие от ранее описанных подходов, RMiner [1] анализирует только те файлы, которые различаются между ревизиями. Для каждой из них определяется множество объявленных типов, в которых содержится информация об их полях и методах и о том, где данная сущность была объявлена, будь то некоторый пакет или указание другого типа. Последнее означает, что для каждого файла ревизии строится некоторое AST. Далее происходит проверка сверху-вниз идентичных по имени и месту объявления узлов, после чего оставшиеся из них сверяются снизу-вверх с использованием техники, позволяющей успешно определять ситуации, в которых некоторые выражения были заменены на переданные в метод параметры, и наоборот. Во время этой сверки постепенно формируется список пар сопоставленных узлов, каждая из которых впоследствии будет проверена на принадлежность к определенному рефакторингу при помощи некоторых четко определенных правил.

Как было сказано выше, данный инструмент анализирует лишь измененные файлы, что положительно сказывается на его времени работы. Отдельное внимание в статье [1] было уделено построению точного

и полного оракула[5] для корректного определения точности и полноты. Полученные в статье значения были примерно равны 0.98 и 0.87, соответственно.

RefDiff 2.0

Описанный в работе [6] подход так же, как и RMiner, ограничивается лишь теми файлами, в которых произошли какие-либо изменения. Для обеих ревизий по затронутым коммитом файлам строится языконезависимое представление — Code Structure Tree (CST). CST представляет собой напоминающий AST граф, узлами которого являются значимые языковые единицы с уникальным именем. Так, для Java узлами в CST могут быть представлены перечисления, классы, интерфейсы и методы. Ребра в данном дереве представляют либо отношения наследования, либо вызов одной сущности из другой. Также для дальнейшего сравнения содержимого узлов происходит токенизация файлов. Далее узлы полученных деревьев постепенно проходят ряд проверок, самая первая, как и в случае с RMiner, рекурсивно сверяет их на полное соответствием идентификаторов, далее же происходит попытка соотнести оставшиеся узлы сначала через сходство их кода, а после — через сходство их дочерних узлов. По полученным парам узлов и узлам без пар строятся списки проведенных рефакторингов.

Достоинствами данного метода является то, что он не требует исходного кода всего проекта, а также то, что лишь построение дерева является языкозависимой операцией. Для замера точности и полноты был использован представленный в работе по RMiner [1] оракул, и их значения составили 0.96 и 0.80 соответственно.

1.3. Сравнение инструментов поиска рефакторингов

Выбор метода в данной работе достаточно важен, поэтому ему было уделено много внимания. Одним из важных критериев при выборе инструмента для поиска рефакторингов является возможность работы

[5] `http://refactoring.encs.concordia.ca/oracle/`

инструмента с кодом на разных языках программирования. Это связано с тем, что IDE на основе IntelliJ Platform способны поддерживать не только язык Java. Поскольку предполагалось, что полученное в ходе данной работы решение может иметь различные сценарии применения, другим важным критерием является его время работы. Не менее существенно и то, чтобы сам инструмент хорошо справлялся со своей задачей.

Так, использование JDEvAn, Ref-Finder и RefactoringCrawler приводило бы к достаточно долгой обработке одиночного коммита, даже если изменению подвергся единственный файл. Насколько успешно алгоритмы находят рефакторинги, частично понятно из таблицы 1, заимствованной из оригинальной работы по RefDiff [7], в которой представлено сравнение с Ref-Finder, RefactoringCrawler и актуальной на тот момент версией RMiner.

Метод	Точность	Полнота
RefDiff	1.000	0.877
RMiner	0.956	0.728
RefactoringCrawler	0.419	0.356
Ref-Finder	0.264	0.624

Таблица 1: Сравнение показателей точности и полноты инструментов поиска рефакторингов в истории систем контроля версий (из [7]).

Так как прочие подходы имели низкие показатели точности и полноты, а также требовали полный исходный код проекта для анализа каждого коммита, было решено свести дальнейшие исследования лишь к сравнению RefDiff и RMiner. Так, наиболее актуальные данные по точности/полноте, основанные на оракуле, предложенном в [1], представлены в таблице 2. Из неё видно, что RefDiff лишь немного уступает в качестве производимого поиска RMiner.

В работе [6] также было указано, что в то время, как RMiner обычно обрабатывает коммиты быстрее, чем RefDiff, последний является более стабильным в плане времени. В рамках данной работы было также проведено сравнение времени обработки одного коммита RefDiff и RMiner на коммитах девяти различных репозиториев. Его результаты

Метод	Точность	Полнота
RefDiff 1.0	0.792	0.802
RefDiff 2.0	0.964	0.804
RMiner	0.988	0.813

Таблица 2: Сравнения показателей точности и полноты (из [6]).

были схожи с полученными в [6], однако также рассматривалось время обработки всех коммитов в одном репозитории. Несмотря на то, что RMiner обрабатывал большинство коммитов в среднем примерно вдвое быстрее, чем RefDiff, это не всегда вело к более быстрой обработке целого репозитория. Так, например, при обработке репозитория RxJava[6] RMiner потратил вдвое больше времени на обработку репозитория, чем RefDiff, в то время как spring-boot[7] был обработан им в полтора раза быстрее. Среднее время обработки одного коммита представлено в таблице 3.

Репозиторий	Среднее		Максимальное	
	RefDiff 2.0	Rminer	RefDiff 2.0	Rminer
helidon	174	418	21 983	112 807
hutool	81	34	1 833	782
igniter	15.6	14.8	75	343
nacos	65	234	2 076	133 827
resilience4j	79	146	11 756	30 207
RxJava	262	490	26 139	448 664
spring-boot	136	88	150 289	92 117
spring-cloud-alibaba	102	35	14 191	2 972
vhr	26.5	14.2	1 079	591

Таблица 3: Время обработки одного коммита в миллисекундах, полученное в данной работе.

Поскольку инструменты схожи и в производительности, и в качестве находимых ими рефакторингов, ключевым фактором в выборе инструмента стала языконезависимость реализации RefDiff и возможность его более простой интеграции с IntelliJ Platform.

[6] `https://github.com/ReactiveX/RxJava`

[7] `https://github.com/spring-projects/spring-boot`

2. Реализация

В данном разделе описано полученное в ходе данной работы решение. Исходный код размещён на Github.[8] Для решения поставленной задачи была разработана архитектура, представленная на рисунке 1. Красным выделены компоненты, которые были реализованы в рамках данной работы.

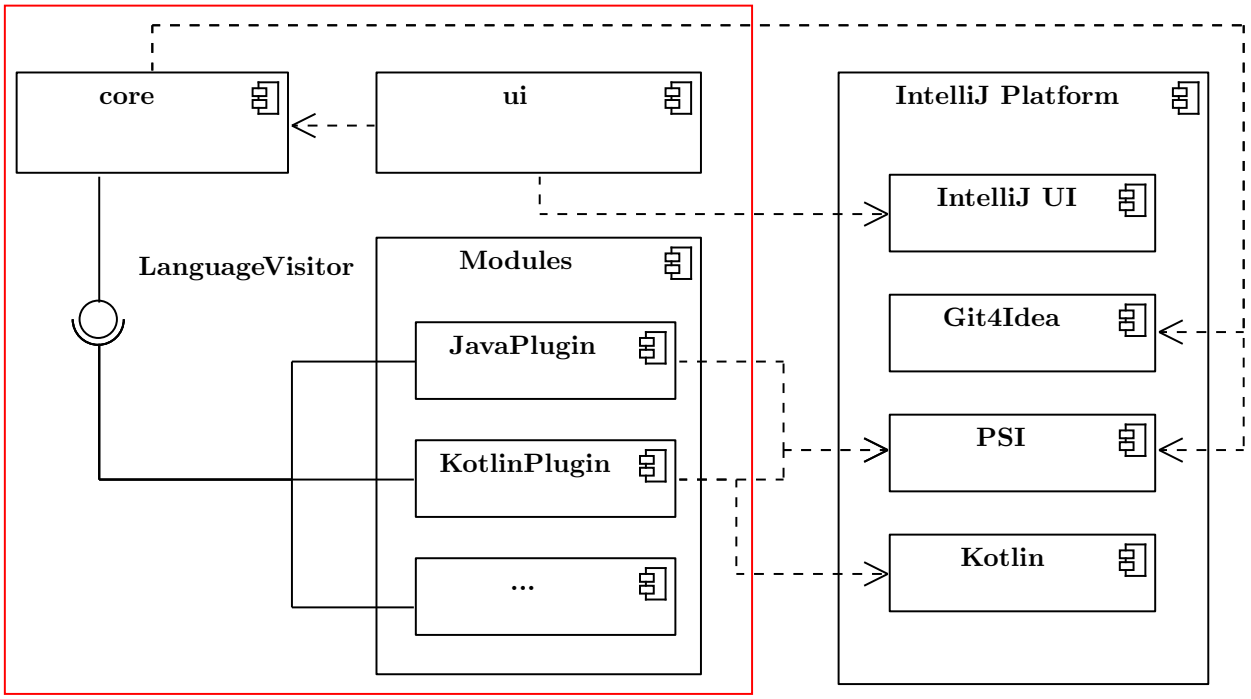

Рис. 1: Диаграмма основных модулей решения.

В разделе 2.1 содержится описание компоненты *core*, в которой реализованы алгоритм RefDiff, а также классы, на основе которых разрабатываются ответственные за построение CST для некоторых языков расширения. Расширения на диаграмме являются частью компоненты *Modules*, и в качестве их примеров в данной работе были разработаны модули для языков Java и Kotlin. Их описание представлено в разделах 2.2 и 2.3 соответственно. Ответственная за непосредственное встраивание в IDE и визуализацию найденных рефакторингов компонента *ui* описана в разделе 2.4.

[8] `https://github.com/Alexander-Solovyov/IdeaRefDiff`

2.1. Базовый модуль

Основные компоненты модуля изображены на рисунке 2. Компоненты RefDiff, GitHelper и GitSourceTree были заимствованы из репозитория, содержащего реализацию RefDiff для Eclipse[9], и были переработаны для работы с git4idea. Задача компонент RefDiff и GitHelper состоит в получении измененных файлов до и после некоторого коммита, а задачейGitSourceTree является хранение этих самых файлов.

Первым делом при вызове $computeDiffForCommit$ по переданному экземпляру $Project$ запрашивается соответствующий ему объект $GitRepository$. Далее при помощи метода $GitHistoryUtils.history()$ на основе переданного хэша коммита происходит запрос $GitCommit$, содержащего информацию по единственному коммиту. Далее у него запрашивается список изменений, для каждого из которых извлекается информация о файлах из ревизий, предшествующих коммиту и следующих за ним. По ним формируется экземпляр $GitSourceTree$, реализующий интерфейс $SourceFileSet$, который впоследствии используется в нескольких частях решения для получения списка файлов и их содержимого. После конструирования данных экземпляров происходит вызов $CstComparator.compare()$, который осуществляет поиск проведенных рефакторингов, его реализация была полностью заимствована из репозитория оригинального проекта.

Как было сказано в секции 1.2, единственным шагом, зависящим от используемого языка программирования, является построение структуры Code Structure Tree. Для корректного формирования узлов данного дерева требуются различные мета-данные, которые невозможно получить без знания о конкретных классах, использующихся для представления сущностей некоторого языка программирования. В связи с этим, реализовать построение CST в общем случае невозможно. Однако, есть ряд операций, необходимых для конструирования дерева, при этом не зависящих от того, каким образом некоторый язык был реализован в IntelliJ Platform.

[9] `https://github.com/aserg-ufmg/RefDiff`

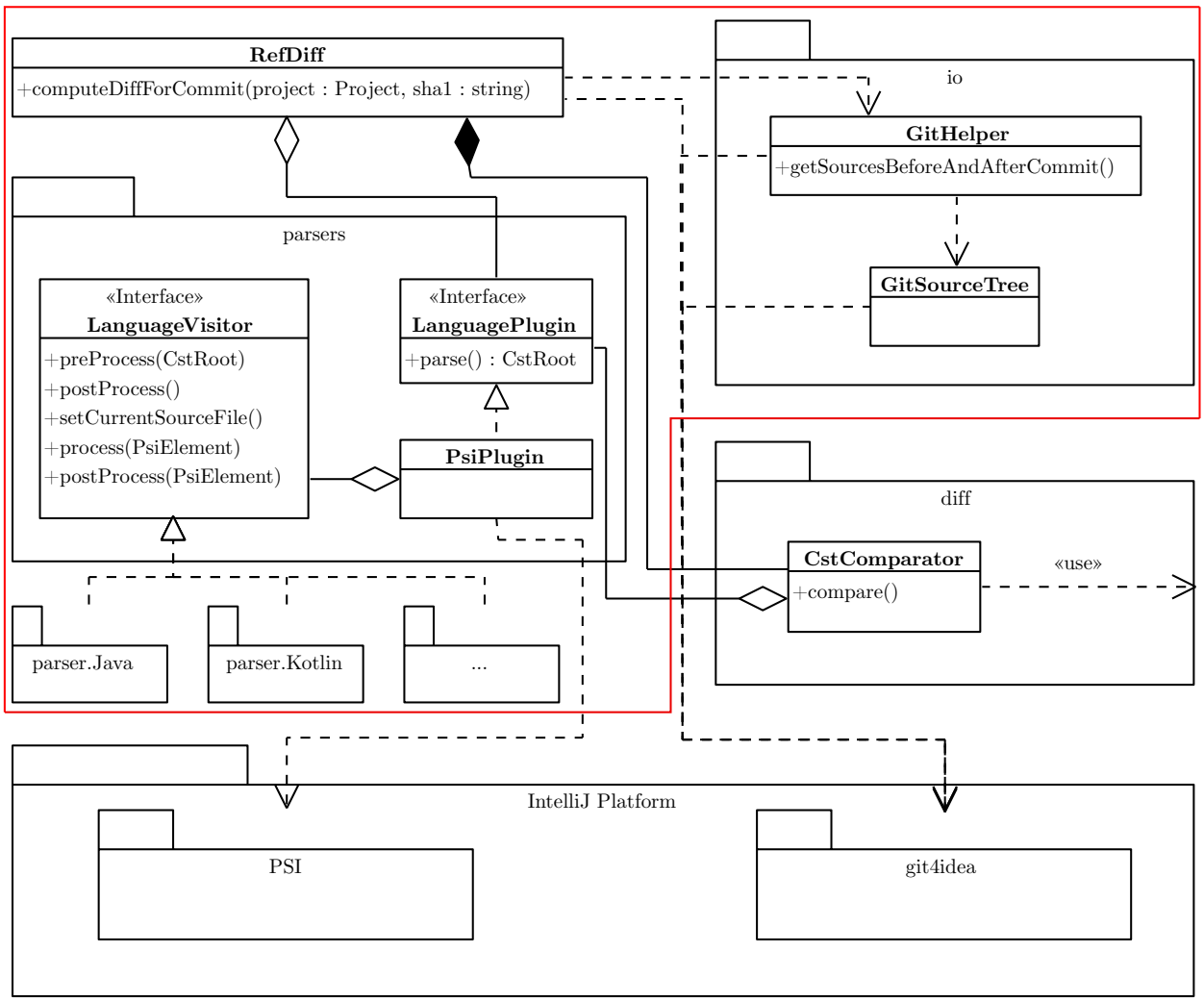

Рис. 2: Архитектуры базового модуля.

В связи с этим было решено реализовать класс $PsiPlugin$, который брал бы на себя решение таких задач, как токенизация и обход дерева, а также добавить интерфейс $LanguageVisitor$, реализации которого должны отвечать за обработку элементов конкретного языка. Подобный подход был выбран не сразу и основывается на том, что позволяет при реализации конкретного языка без каких-либо проблем использовать имеющиеся в IntelliJ Platform реализации $PsiElementVisitor$ под конкретные языки. В $LanguageVisitor$ есть две группы методов: первая — методы для его начальной конфигурации и конечной обработки, а вторая — методы, связанные с обработкой конкретного файла и элемента. Также стоит добавить, что работа с самим $PsiPlugin$ происходит

через реализуемый им интерфейс $LanguagePlugin$, который при желании можно использовать для написания расширения, полностью реализующего логику построения CST дерева по списку файлов и не ограниченного необходимостью работать через интерфейс $LanguageVisitor$.

Единственный доступный для вызова извне метод расширения — $parse()$, принимающий в качестве аргумента некоторый экземпляр, реализующий интерфейс $SourceFileSet$, по которому строится список файлов для обработки. В начале и конце данного метода происходит обращение к методам преобработки и постобработки $LanguageVisitor$, между ними же происходит обработка файлов. Для каждого из них формируется экземпляр $PsiFile$, по которому строится список токенов. Также он используется для рекурсивного обхода всех PSI-элементов в файле, и для них вызывается $LanguageVisitor.process()$, который также определяет, нужно ли спускаться дальше. По завершении этих шагов формируется некоторый экземпляр $CstRoot$, который и возвращается пользователю.

2.2. Реализация расширения для Java

На основе модульной архитектуры, спроектированной в рамках данной работы, было реализовано расширение, позволяющее строить CST для проектов на Java. Его устройство представлено на рисунке 3.

Разработанное расширение довольно компактно и состоит из трёх компонент: $JavaPlugin$, $JavaCstModel$ и $JavaPsiVisitor$. Класс $JavaCstModel$ предназначен для хранения конструируемого $CstRoot$ и некоторой вспомогательной информации, которая не должна теряться при смене обрабатываемого в рамках одного вызова $parse$ файла. Также он отвечает за добавление в граф ребер на основе установленных между различными сущностями связей. $JavaPsiVisitor$ отвечает за обработку элементов в рамках одного файла. Он является наследником $JavaElementVisitor$ и переопределяет реализацию для методов $visitClass$ и $visitMethod$, в каждом из которых он конструирует узел на основе имеющейся информации. Затем полученный узел сохраняет-

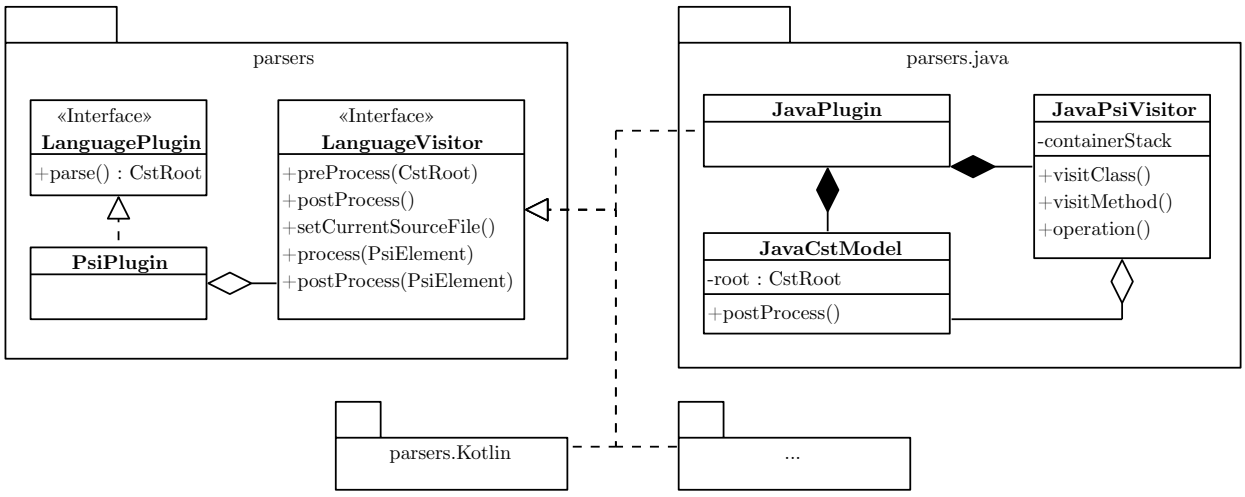

Рис. 3: Диаграмма классов расширения для Java.

ся в качестве потомка родительского узла, который либо соответствует $CstRoot$, либо некоторому классу, в котором определен обрабатываемый элемент. Стоит также заметить, что анонимные классы в рамках данной реализации не обрабатываются.

$JavaPlugin$, реализующий интерфейс $LanguageVisitor$, отвечает за конструирование модели при вызове предобработки и создание нового экземпляра обработчика элементов для каждого нового файла. Все остальные вызовы он делегирует либо модели, либо экземпляру $JavaPsiVisitor$ с единственным исключением в методе *process*, где, помимо самого обращения для обработки элемента, определяется, необходимо ли продолжать рекурсивную обработку для данного элемента. Рекурсивный спуск не происходит лишь для анонимных классов.

2.3. Реализация расширения для Kotlin

Для проверки того, что представленное в данной работе решение способно находить рефакторинги не только для языка Java, было разработано расширение для языка Kotlin.[10] Устройство и логика работы этого расширения схожи с устройством и логикой работы расширения для Java. Основное отличие заключено в компоненте $KotlinPsiVisitor$,

[10] `https://github.com/JetBrains/kotlin`

которая наследуется от $KtVisitorVoid$ и переопределяет методы для обхода экземпляров классов $KtClass$ и $KtFunction$. Столь малые отличия от реализации расширения для Java являются результатом того, что и для Java, и для Kotlin имеются представления посредством PSI-элементов.

2.4. Интеграция с IDE

В данном разделе описывается модуль, отвечающий за визуализацию найденных рефакторингов. Он представлен на диаграмме 4. Также данный модуль отвечает за взаимодействие с модулями, реализующими логику построения CST. Взаимодействие обеспечивается за счет создания специального интерфейса $LanguagePluginCreator$ и использования его в качестве точки расширения посредством $ExtensionPointName$ и указания соответствующей информации в конфигурации проекта. При таком подходе реализовавшие данный интерфейс модули, которые подключены к IDE на платформе IntelliJ, автоматически используются при поиске рефакторингов в истории проекта для соответствующего языка программирования.

2.4.1. Визуализация

Несмотря на то, что в IntelliJ Platform есть различные компоненты, использующиеся для представления различий между несколькими файлами, в рамках данной работы они не использовались. Связано это с тем, что при проведении некоторых рефакторингов могут затрагиваться несколько файлов сразу. Например, информация об ExtractMove рефакторинге, при котором происходит извлечение метода и перемещение его в другую компоненту, содержится в старой и новой версии файла, из которого произошло извлечение, а также в файле, в который метод был извлечен. В таком случае соответствующие компоненты неспособны представить всю полученную информацию об обнаруженных рефакторингах. Поэтому для решения этой задачи был реализован специальный набор компонент.

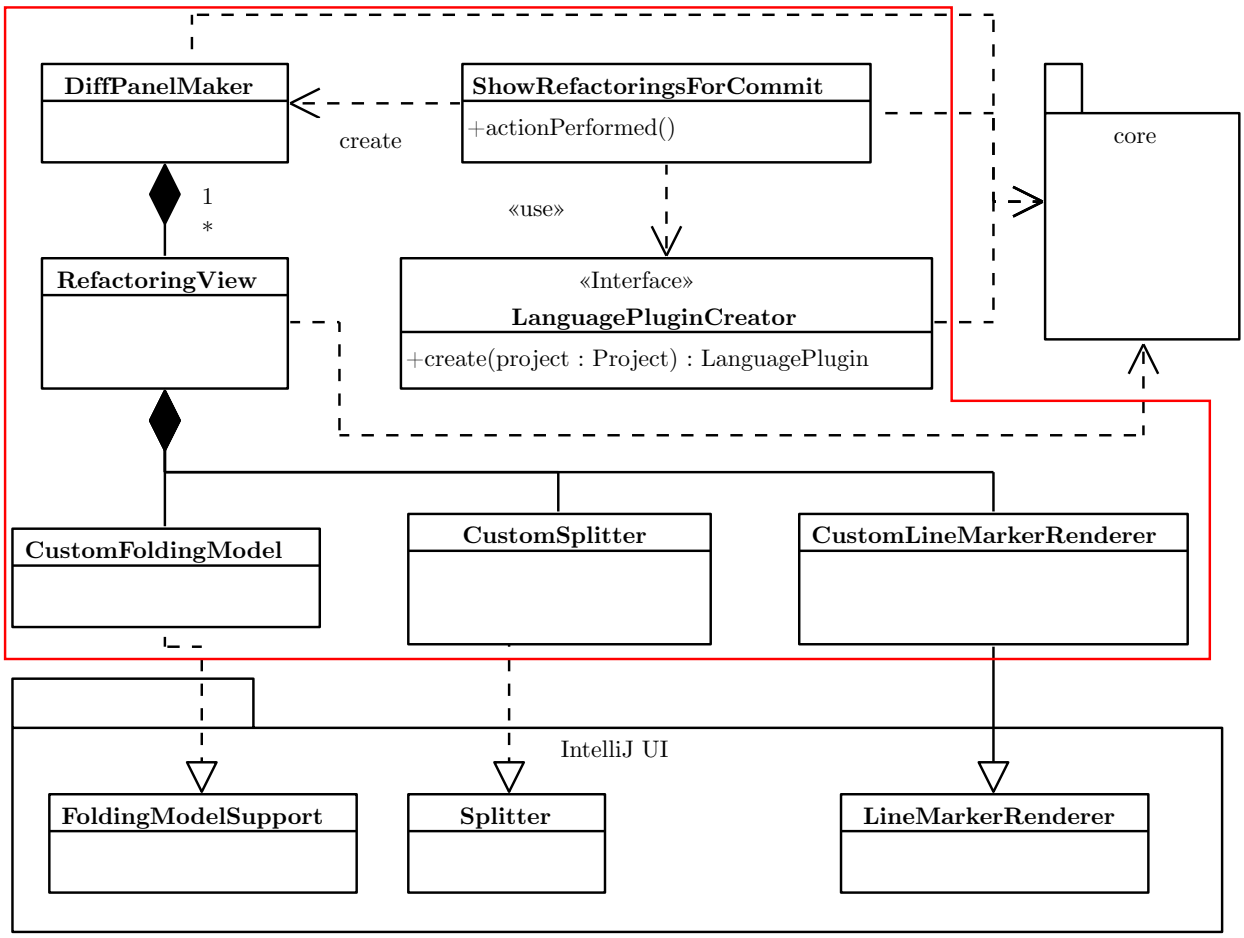

Рис. 4: Архитектура компоненты, осуществляющей визуализацию.

Основными компонентами, отвечающими за визуализацию, являются $RefactoringView$ и $DiffPanelMaker$. $RefactoringView$ отвечает за представление информации о конкретном рефакторинге. Для этого в нём используется ряд компонент из IntelliJ Platform SDK. Основной из них является $Editor$, в экземплярах которого представляется код файла либо до, либо после рефакторинга. Для того, чтобы можно было сразу увидеть, какие участки исходного кода были затронуты изменениями, было решено добавить подсветку. Синим было решено помечать модифицированные при рефакторинге сущности, а зеленым — новые сущности, которые отсутствовали до рефакторинга. Для этого использовались компоненты $RangeHighlighter$ в сочетании с $CustomLineMarkerRenderer$, реализующей интерфейс $LineMarkerRenderer$. Для того, чтобы окрашенные части в редакто-

рах были визуально соединены, был создан наследник $DividerImpl$ с переопределенной операцией отрисовки.

$DiffPanelMaker$ отвечает за конструирование окна с информацией о рефакторингах, обнаруженных в некотором коммите. В частности, он отвечает за конструирование $RefactoringView$ для новых рефакторингов и за переключение между ними.

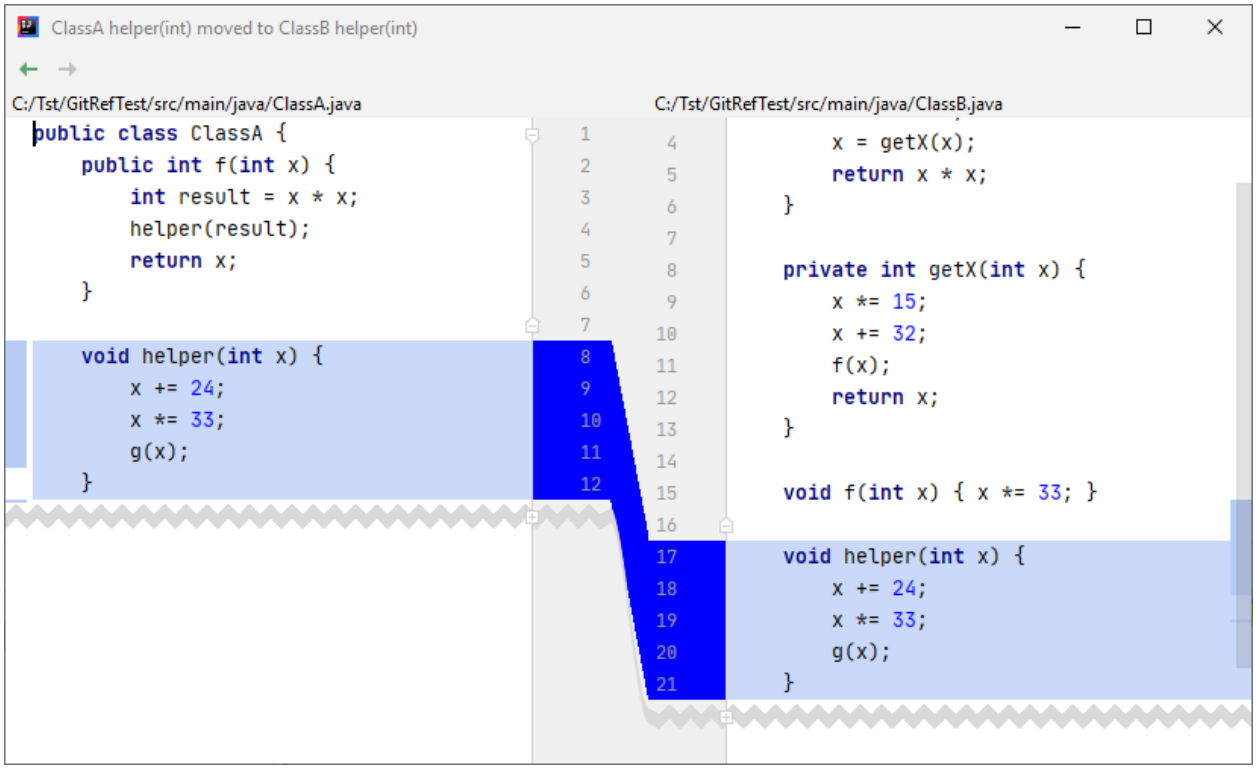

Рис. 5: Визуализация рефакторинга с перемещением метода $helper()$ из $ClassA$ в $ClassB$ в проекте на Java.

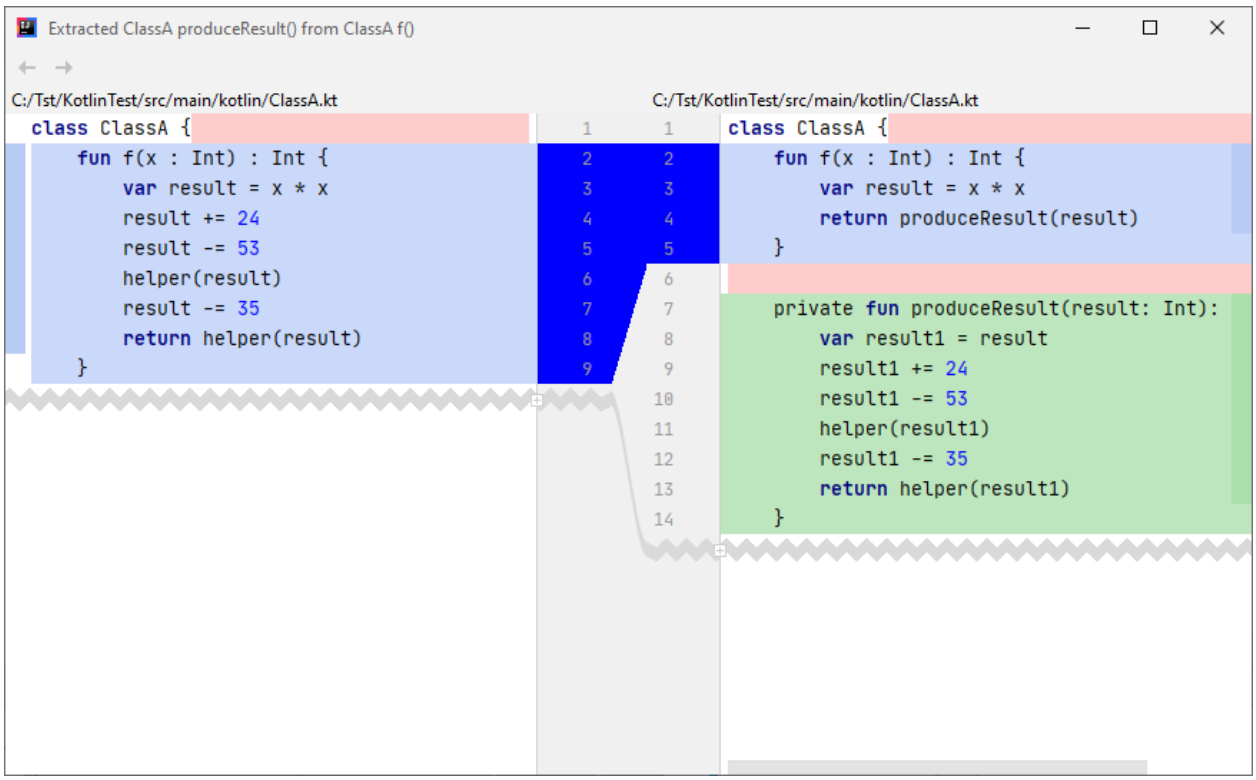

Рис. 6: Визуализация рефакторинга с извлечением метода *produceResult*() из метода *f* в проекте на Kotlin.

3. Апробация

Для проведения апробации полученное решение было предоставлено пяти разработчикам с опыта работы в индустрии от одного до пяти лет. С его помощью они искали рефакторинги в проектах на языке Java. После тестирования были собраны отзывы, а также оценки полезности и удобства использования расширения по пятибальной шкале. При определении полезности разработчиками учитывались различные факторы, основным из которых было то, находятся ли осуществлённые рефакторинги, о которых им известно, или нет. Так, полезность могла получить оценку “5” только при нахождении всех рефакторингов, а оценка “1” выставлялось, если никакие рефакторинги, о которых было известно, найдены не были. Оценка удобства была субъективной и выдавалась разработчиками по их ощущениям. Полученные результаты представлены на рисунке 7.

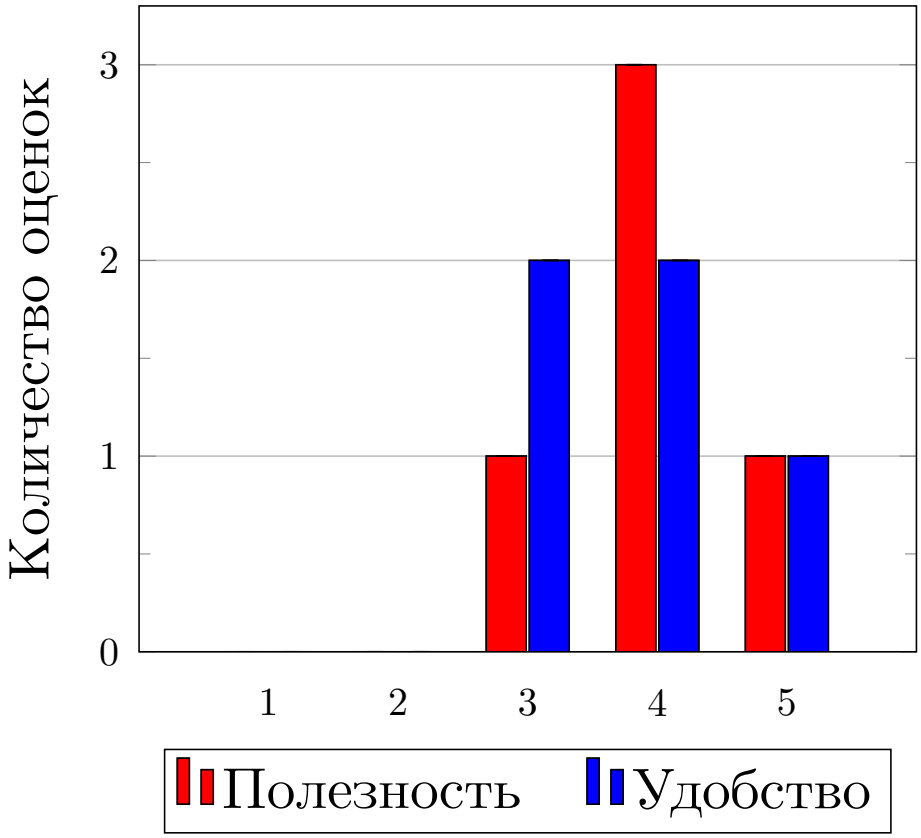

Рис. 7: Полученные оценки при апробации решения.

В целом, работа расширения была оценена положительно. Разработ-

чиками было отмечено, что удавалось обнаружить большинство проведенных рефакторингов, но некоторые из ожидаемых все же не были найдены, однако это было ожидаемо, поскольку изначально инструмент RefDiff находит лишь фиксированный набор типов рефакторингов. Также одним из замечаний было то, что, несмотря на простоту и удобство используемого подхода к отображению данных, при визуализации затрагивающих большие компоненты рефакторингов информативность значительно ухудшается. Кроме того, разработчиками был выдвинут ряд предложений по улучшению удобства использования решения, многие из них касались изменения различных космитических аспектов. Например, одним из предложений было выделять цветом не участвующие в рефакторингах сущности, а лишь те их участки, которые различаются между ревизиями.

Заключение

В ходе данной работы были достигнуты следующие результаты:

- Проведен обзор предметной области. Рассмотрены алгоритмы поиска рефакторингов в истории проектов, проведено их сравнение.
- На основе сравнения выбран наиболее подходящий инструмент для поиска рефакторингов в истории проектов в IDE.
- Спроектирована модульная архитектура, позволяющая поддержать работу выбранного инструмента с различными языками.
- Реализована библиотека, интегрированная с IntelliJ Platform и позволяющая определять рефакторинги для различных языков, для которых существуют представления посредством PSI интерфейса в IntelliJ Platform.
- На основе разработанной архитектуры реализованы расширения, позволяющие находить рефакторинги в репозиториях проектов на Java и Kotlin.
- Проведена апробация решения. Плагин протестирован пользователями, от них получена преимущественно положителная оценка, также получены замечания и предложения по улучшению решения.

Развитие полученного в ходе данной работы результата возможно по нескольким направлениям:

- Развитие самого алгоритма RefDiff. Любые улучшения алгоритма, которые не затрагивают его независимости от используемого языка программирования, могут быть успешно реализованы в данном проекте.
- Улучшение визуального представления результатов работы инструмента. Как показала апробация, реализованный в рамках данной работы пользовательский интерфейс может быть упрощен и доработан.

- Разработка расширений для поддержки поиска рефакторингов в отличных от Java и Kotlin языках программирования.

Список литературы

[1] Accurate and Efficient Refactoring Detection in Commit History / Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari et al. // Proceedings of the 40th International Conference on Software Engineering. — 2018. — P. 483–494. — Access mode: http://doi.acm.org/10.1145/3180155.3180206.

[2] Automated Detection of Refactorings in Evolving Components / Danny Dig, Can Comertoglu, Darko Marinov, Ralph Johnson // Proceedings of the 20th European Conference on Object-Oriented Programming. — 2006. — P. 404–428. — Access mode: https://doi.org/10.1007/11785477_24.

[3] Broder Andrei. On the Resemblance and Containment of Documents // Proceedings of the Compression and Complexity of Sequences 1997. — 1997. — 06. — P. 21–29.

[4] Comparing Approaches to Analyze Refactoring Activity on Software Repositories / Gustavo Soares, Rohit Gheyi, Emerson Murphy-Hill, Brittany Johnson // J. Syst. Softw. — 2013. — Apr. — Vol. 86, no. 4. — P. 1006–1022. — Access mode: https://doi.org/10.1016/j.jss.2012.10.040.

[5] Opdyke William, Johnson Ralph. Refactoring Object-Oriented Frameworks. — USA : University of Illinois at Urbana-Champaign, 1992. — 07. — P. 1–2.

[6] RefDiff 2.0: A Multi-language Refactoring Detection Tool / Danilo Silva, João Silva, Gustavo Santos et al. // IEEE Transactions on Software Engineering. — 2020. — 01. — Vol. PP. — P. 1–17.

[7] Silva Danilo, Valente Marco Tulio. RefDiff: Detecting Refactorings in Version Histories // Proceedings of the 14th International Conference on Mining Software Repositories. — 2017. — P. 269–279. — Access mode: https://doi.org/10.1109/MSR.2017.14.

[8] Template-Based Reconstruction of Complex Refactorings / Kyle Prete, Napol Rachatasumrit, Nikita Sudan, Miryung Kim // Proceedings of the 2010 IEEE International Conference on Software Maintenance. — 2010. — P. 1–10. — Access mode: https://doi.org/10.1109/ICSM.2010.5609577.

[9] Xing Zhenchang, Stroulia Eleni. UMLDiff: An Algorithm for Object-Oriented Design Differencing // Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. — 2005. — P. 54–65. — Access mode: https://doi.org/10.1145/1101908.1101919.

[10] Xing Zhenchang, Stroulia Eleni. The JDEvAn Tool Suite in Support of Object-Oriented Evolutionary Development // Companion of the 30th International Conference on Software Engineering. — 2008. — P. 951–952. — Access mode: https://doi.org/10.1145/1370175.1370203.