Санкт-Петербургский Государственный Университет
Математическое обеспечение и администрирование информационных систем

Кафедра системного программирования

Свитков Сергей Андреевич

Отображение изменяемости метода на основе исторической информации в IntelliJ IDEA

Выпускная квалификационная работа

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

Рецензент:
аналитик ООО "Интеллиджей Лабс" Н. И. Поваров

Санкт-Петербург
2019

SAINT-PETERSBURG STATE UNIVERSITY
Software and Administration of Information Systems

Software Engineering

Sergey Svitkov

Visualization of methods changeability based on version control systems data in IntelliJ IDEA

Graduation Thesis

Scientific supervisor:
Candidate of Engineering Sciences Timofey Bryksin

Reviewer:
analyst, IntelliJ Labs Co. Ltd. Nikita Povarov

Saint-Petersburg
2019

Оглавление

Введение

Разработка программного обеспечения представляет собой сложный и долгий процесс, включающий в себя координацию команды (иногда и нескольких команд) разработчиков, оценку рисков, планирование, решение сложных технических задач и другие аспекты. Поскольку чаще всего над одним проектом или модулем работает команда разработчиков, разработка осуществляется с использованием системы контроля версий. Это позволяет контролировать изменения, производимые разными членами команды в коде проекта. Для улучшения качества кода применяются различные техники и практики, такие как статический анализ кода и код ревью. Однако, несмотря на это, некоторые ошибки всё же остаются незамеченными.

Ряд исследований показывает связь между частотой нахождения ошибок в фрагментах кода и тем, как часто данные фрагменты изменяются ([13], [3], [6]). Метод или функция может часто изменяться из-за различных технических (часть параметров могут быть вынесены в файл конфигурации), архитектурных или внешних (слишком частые изменения бизнес-логики) проблем проекта. Так, в статье [5] говорится о том, что если в фрагменте кода была найдена ошибка, то с высокой вероятностью в этом же фрагменте могут быть и другие ошибки. Также успешное исправление ошибки может произойти не с первой попытки: например, результаты статьи [7] показывают, что в 25% случаев исправление ошибки было неправильным и требовало повторного исправления.

Таким образом, можно сделать вывод о том, что часто изменяемые фрагменты кода заслуживают повышенного внимания со стороны разработчиков. Если привлечь внимание разработчика к такому фрагменту кода, то он сможет более пристально его изучить и исправить потенциальные проблемы в нём. Применение такой практики на регулярной основе позволит улучшить качество как отдельно взятых частей проекта (благодаря исправлению локальных ошибок), так и всего проекта в целом (изменение архитектуры проекта при необходимости). Чтобы

такое стало возможно, необходимо, чтобы инструмент, позволяющий осуществить это, был интегрирован в среду разработки, которую используют разработчики. Иначе вероятность его использования сильно понижается, поскольку, как показывает исследование [11], разработчики не склонны к использованию других инструментов (документация, трекеры задач), кроме среды разработки.

Постановка задачи

Исходя из проблем, сформулированных во введении, было принято решение реализовать инструмент для привлечения внимания разработчиков к потенциально проблемным фрагментам кода. Таким образом, целью работы является разработка расширения для среды разработки IntelliJ IDEA[1], позволяющего визуализировать историю изменений кода. Для достижения цели были поставлены следующие задачи:

- реализовать плагин, интегрирующийся с IntelliJ Platform SDK для анализа изменений в системе контроля версий и отображения полученных данных в редакторе кода;
- интегрировать решение с инструментом для отслеживания рефакторингов;
- оптимизировать решение для работы с большими репозиториями;
- опубликовать плагин в репозитории плагинов JetBrains, собрать и проанализировать статистику его использования.

[1] https://www.jetbrains.com/idea/

1. Обзор

1.1. Ближайший аналог

Статья *code_call_lens: Raising the Developer Awareness of Critical Code* [9] описывает расширение для среды разработки (Integrated development environment, IDE) Visual Studio Code, собирающее и визуализирующее информацию о количестве вызовов метода в работающем приложении. Статистика о вызовах метода в приложении, развернутом на “рабочем“ сервере, по мнению авторов, позволяет понять влияние конкретного фрагмента кода на работу приложения в целом. Так, если наиболее часто исполняемый метод перестанет корректно работать, то с большой вероятностью можно предположить, что работа приложения в целом будет так же нарушена. Представление собранной информации прямо в IDE позволяет привлечь внимание разработчиков к наиболее важным частям проекта не отвлекая их от решения задач (рис. 1).

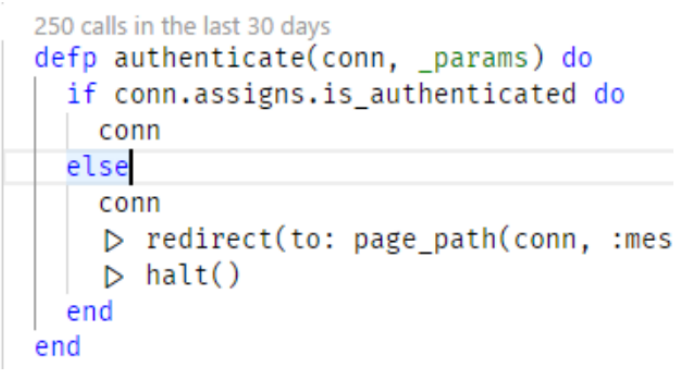

Рис. 1: Пример UI code_call_lens (заимствовано из [9])

Сбор статистики о частоте вызовов осуществляется путём добавления логгирующей компоненты к работающему приложению. Собранные данные отправляются на веб-сервер для хранения и анализа, после чего расширение через REST API получает информацию о частоте вызовов методов. Основная информация отображается в виде неизменяемых текстовых меток, кроме того, заголовок метода выделяется цветом в зависимости от частоты вызовов.

По словам авторов, сбор данных не оказывает влияния на скорость работы приложения, поскольку выполняется в фоновом режиме. Кроме

того, сервер обрабатывает хранит и обрабатывает все полученные данные в оперативной памяти, что позволяет добиться высокой скорости и обновления статистики почти в режиме реального времени (задержка составляет около 1 секунды).

В своей работе, впрочем, авторы столкнулись с важной проблемой: при некоторых рефакторингах кода (например, при изменении имени метода, переносе метода в другой класс или изменении его параметров) статистика, собранная логгирующей компонентой, становится неактуальной. Это вызвано тем, что в качестве уникального идентификатора метода используется его сигнатура. Кроме того, подход, предлагаемый в статье, подразумевает реализацию логгирующей компоненты для каждой платформы, на основе которой работает отслеживаемое приложение. Это может быть затруднительно и времязатратно из-за особенностей платформы и инструментов, использованных при разработке.

Работа представляет собой пример отличного инструмента для визуального представления наиболее важных фрагментов кода и привлечения к ним внимания разработчиков. Исходя из этого, было принято решение использовать аналогичный подход к визуализации данных об изменяемости методов, но изменить механизм сбора статистики. Это необходимо для того, чтобы, во-первых, избежать потери данных при рефакторингах кода, а, во-вторых, сделать решение более обобщённым и не использовать отдельный сервер для сбора и анализа информации.

1.2. IntelliJ Platform

IntelliJ Platform представляет собой общую основу для всех IDE компании JetBrains (Intellij IDEA, WebStorm, RubyMine и др.) и предоставляет общий программный интерфейс (Application programming interface, API) для работы с разными языками программирования, позволяющий существенно облегчить разработку расширений для платформы. Далее рассмотрим компоненты IntelliJ Platform, используемые в ходе реализации решения.

Git4Idea

Модуль Git4Idea [10] предоставляет API для работы с системой контроля версий Git. С помощью данного модуля можно выполнять различные запросы к Git репозиторию, используя как низкоуровневое API, позволяющее полностью вручную сформировать запрос, так и набор уже готовых методов. К запросу можно добавить различные аргументы.

В рамках данной работы наибольший интерес представляет получение истории изменений проекта. Git4Idea позволяет обработать нужный фрагмент истории изменений, при этом не загружая все коммиты в память. Благодаря этому при обработке изменений не возникает проблем с количеством используемой памяти. Каждый коммит представлен экземпляром класса `GitCommit`. Объект данного класса хранит в себе информацию об изменениях, сделанных в одном коммите, в виде списка объектов класса `Change`, разбивая их по файлам. Так, например, если в одном коммите были сделаны изменения в 3 файлах, то в соответствующем коммиту экземпляре `GitCommit` будет содержаться список Change из трёх элементов. Change хранит имя файла, в котором было сделано изменение, тип изменения (NEW — файл был добавлен в коммите, DELETE — файл был удалён при коммите, MOVED — файл был перемещён при коммите, MODIFIED — содержимое файла было изменено) и ревизии файла до и после коммита.

Для отслеживания изменений, происходящих в репозитории в реальном времени, модуль предоставляет интерфейс для реализации обработчика событий. Таким образом, можно обрабатывать такие события, как изменение текущей ветки, коммит, откат коммитов и т.д. Важно также отметить, что обработчик срабатывает на изменения в репозитории, поэтому не имеет значения, было ли изменение сделано из визуального интерфейса среды разработки или нет.

Program Structure Interface

Program Structure Interface (PSI) представляет собой набор API для работы с языками, поддерживаемыми IntelliJ Platform. Код программы представляется в виде PSI-дерева — синтаксического дерева, которое содержит всю необходимую информацию. Пример структуры дерева изображен на рисунке 2. Узлы дерева представляют собой различные компоненты программы: классы, методы, поля, комментарии. Все узлы дерева наследуют общий интерфейс — PsiElement.

Поскольку данная работа подразумевает выделение методов, в которых были сделаны изменения, наибольший интерес представляет тип узлов PSI-дерева, соответсвующих методам. Такие узлы представлены классом `PsiMethod`, экземпляры которого содержат информацию о сигнатуре метода, а также о том, на какой позиции в документе начинается и заканчивается метод. Таким образом, путём рекурсивного обхода PSI-дерева для конкретного файла можно получить полную информацию о методах этого файла.

Из экземпляра любого класса-наследника `PsiElement` можно получить ссылку на часть программы, в которой объявлен элемент, соответствующий данному узлу. Такие ссылки позволяют осуществлять навигацию по проекту и, например, построить список ссылок на объявление наиболее часто изменяемых методов проекта, что может быть использовано для визуализации собранных данных.

Editor

Интерфейс Editor представляет собой API для работы с текстовым редактором внутри IDE. В API представлено большое количство разнообразных методов, предназначенных для добавления различных компонент в визуальный интерфейс редактора, получения информации о добавленных компонентах, добавления обработчиков событий Editor, получения InlayModel и т.д.

InlayModel представляет собой компонент редактора, позволяющий добавлять визуальные элементы. В версии 2018.3 IntelliJ Platform в

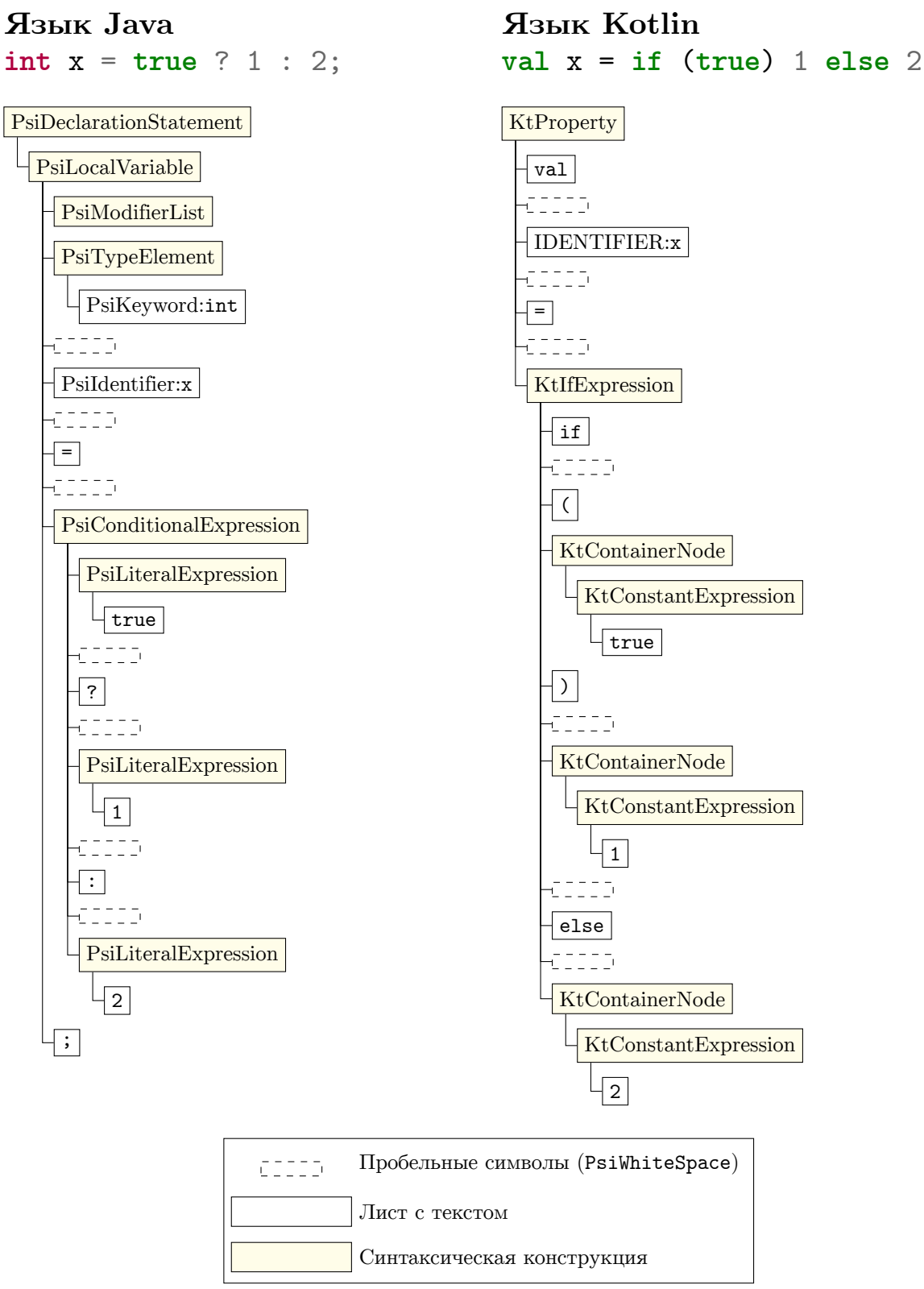

Рис. 2: Примеры PSI-деревьев, заимствовано из работы [15]

InlayModel была добавлена возможность создания блочных визуальных элементов в интерфейсе редактора, что позволяет отображать информацию об изменениях метода в визуальном интерфейсе IDE. Для

отрисовки элементов используются инструменты из пакета `java.awt`, благодаря этому можно представить информацию об изменяемости метода не только с помощью текста, но и с помощью графиков.

1.3. RefactoringMiner

Для сохранения корректности собранных данных рефакторинги, происходящие в проекте, нужно отслеживать и обрабатывать. Существует ряд инструментов, предназначенных для нахождения рефакторингов в истории изменений проекта: RefFinder [12], JDEvAn [14], RefactoringCrawler [2], RefactoringMiner [1]. Первые три инструмента представляют собой плагины для IDE Eclipse, последний — самостоятельную утилиту, основанную на компонентах Eclipse.

Согласно исследованию, проведённому авторами RefactoringMiner, этот инструмент обладает лучшими результатами среди перечисленных инструментов (точность 0.98, полнота 0.87) и представлен в виде библиотеки, поэтому был выбран для использования в рамках данной работы. Инструмент поддерживает отслеживание различных типов рефакторингов, связанных с пакетами, классами, методами и переменными (см. таблицу 1).

Таблица 1: Типы рефакторингов, поддерживаемые RefactoringMiner

Элемент в коде	Рефакторинги
Пакет	Изменить пакет (перенести, переименовать, разбить)
Класс	Переместить класс, Переименовать класс, Извлечь родительский класс или интерфейс
Метод	Извлечь метод, Встроить метод, Подъём метода, Спуск метода, Переименовать метод, Перенести метод, Извлечь и перенести метод
Поле	Подъем поля, Спуск поля, Перенести поле

Алгоритм поиска рефакторингов устроен следующим образом: синтаксический анализатор (парсер) разбивает код, имеющийся в двух ре-

визиях коммита, на сущности. На первом этапе происходит тривиальная проверка: все сущности с одинаковыми именами в старой и новой версии считаются одними и теми же. На втором этапе происходит поиск рефакторингов путем сравнения сущностей в соответствии с набором эвристик.

1.4. SQLite

Данные, полученные в ходе разбора истории проекта, нужно сохранять при перезапуске IDE. Сохранение с помощью сериализации объектов не подходит в рамках данной работы, поскольку объем данных может быть довольно большим. В таком случае коллекция объектов, полученных при десериализации, будет требовать большого количества свободной памяти. Этого можно избежать, если использовать для сохранения полученной статистики базу данных.

Для хранения данных была выбрана реляционная СУБД SQLite [4]. SQLite является встраиваемой СУБД, то есть не использует парадигму "клиент-сервер", а компонуется с программой. Это позволяет избежать необходимости использования отдельного сервера для её запуска. Вся база данных представлена одним файлом. Кроме того, данная СУБД является кросс-платформенной, то есть, поддерживается во всех популярных операционных системах (Linux, MacOS, Windows). Начиная с версии 3.24 данная СУБД поддерживает UPSERT запросы, т.е. INSERT запросы с возможностью произвести UPDATE значений реляционного кортежа в случае возникновения конфликта по первичному ключу при их исполнении. С помощью таких запросов появляется, например, возможность инкрементального подсчёта статистики. Язык запросов, используемый SQLite, соответствует большинству SQL стандартов, однако является слабо типизированным.

2. Реализация

В данной секции описаны детали реализации работы. Расширение реализовано на языке Java, исходный код размещён на Github в репозитории “topias” организации JetBrains Research[2].

2.1. Предлагаемый алгоритм

Для решения задачи был разработан алгоритм, представленный на рисунке 3. Более подробно он представлен на рисунке 4.

Для сбора данных об изменениях в системе контроля версий при открытии проекта в IDE происходит запрос к Git репозиторию. Поскольку хэш-код последнего обработанного коммита сохраняется для каждой ветки репозитория, происходит проверка наличия ещё не обработанных коммитов. Если такие имеются, то для каждого из них осуществляется разбор сделанных изменений с выделением методов, которые были изменены. Это происходит в соответствии с типом изменения. Полученные данные помещаются в отдельную коллекцию. Также происходит добавление добавленных, удаленных и перемещённых методов в хранилище методов в оперативной памяти. Это необходимо для того, чтобы обнаружить рефакторинги, если RefactoringMiner их пропустит.

После происходит поиск рефакторингов, при их обнаружении формируется коллекция объектов с информацией о том, какие методы были затронуты. Каждый из объектов коллекции содержит информацию о методе до и после рефакторинга. Полученная коллекция сравнивается с хранилищем методов, в случае если метод есть и в коллекции и в хранилище, то из хранилища удаляется информация о нём. На следующем шаге происходит обновление данных об измененных методах в соответствии с имеющейся информацией о примененных рефакторингах. Полученный результат записывается в базу данных. Информация об изменениях записывается в таблицу-журнал, добавленные методы добавляются в словарь методов, а записи словаря о методах, к кото-

[2]https://github.com/ml-in-programming/topias

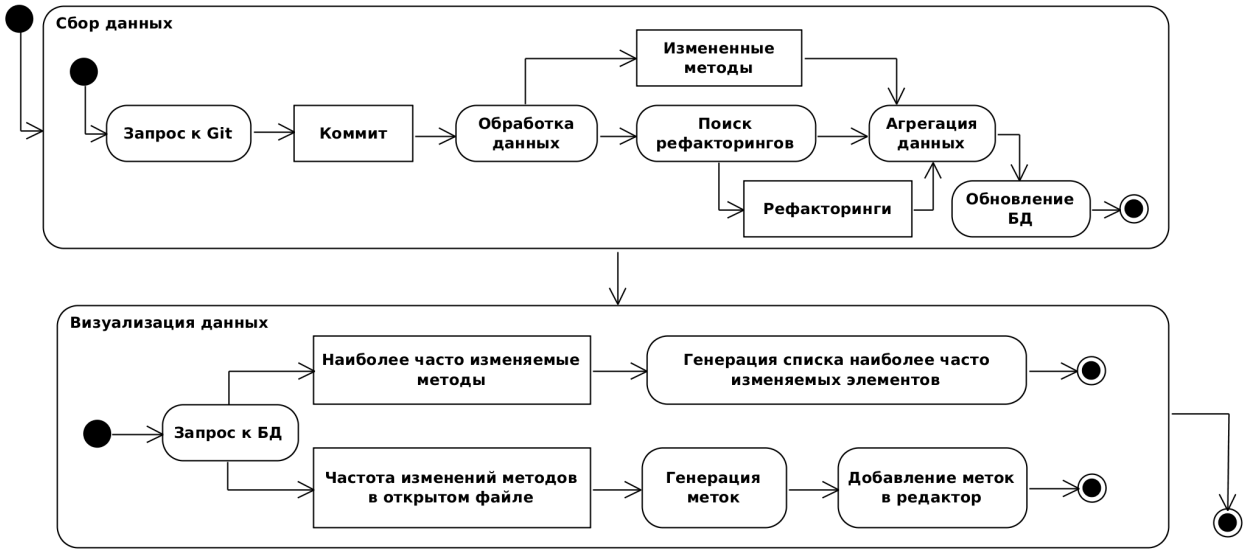

Рис. 3: Алгоритм работы разработанного решения

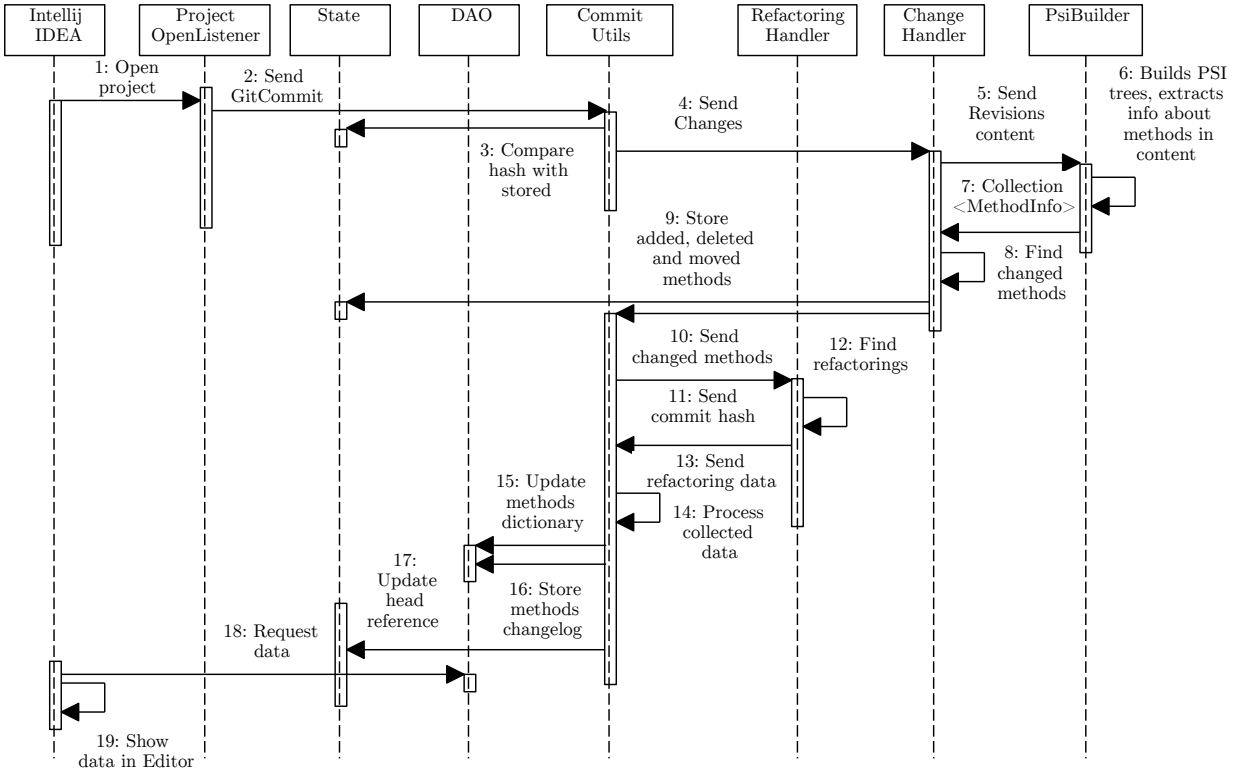

Рис. 4: Схема исполнения

рым были применены рефакторинги, обновляются.

Обработка данных осуществляется в фоновом режиме, поэтому это не оказывает ощутимого влияния на пользовательский опыт, получаемый при работе с IDE. Кроме того, пользователю показывается индикатор прогресса, который позволяет понять, какая часть данных уже

обработана.

По завершении разбора изменений собранные данные визуализируются. Для этого используются неизменяемые метки с информацией о количестве изменений за период времени, а также список наиболее часто изменяемых методов проекта. Период времени, за который отображаются данные, можно изменить в настройках расширения. Выборка данных и генерация визуальных элементов также осуществляется в фоновом режиме, а метки, добавляемые в интерфейс редактора, не оказывают ощутимого влияния на скорость работы IDE.

2.2. Архитектура

Для реализации предложенного алгоритма была разработана архитектура, представленная на рисунке 5. Красным выделены компоненты, реализованные в рамках данной работы, синим — использованные внешние инструменты.

Обработка событий, происходящих в IDE, осуществляется с помощью подписки на внутреннюю шину событий. Модуль Components позволяет обработать открытие проекта и подписать классы, которым необходимо получать уведомления о событиях IDE, на шину. При открытии проекта в IDE происходит проверка наличия в проекте системы контроля версий. При её наличии с помощью API Git4Idea определяется текущая ветка репозитория и выполняется запрос к репозиторию с целью выяснить, имеются ли необработанные данные. Если изменения в выбранной ветке уже разбирались, то обработка начинается с первого необработанного коммита, иначе за последние 30 или 7 дней в зависимости от выбранного в настройках плагина периода. В случае наличия неразобранных коммитов запускается их обработка, которая происходит в хронологическом порядке, поскольку это необходимо для корректной обработки рефакторингов.

Выделение методов, измененных в коммите, происходит с помощью PSI. По двум ревизиям измененного файла строятся PSI-деревья, рекурсивный обход которых позволяет получить информацию о методах,

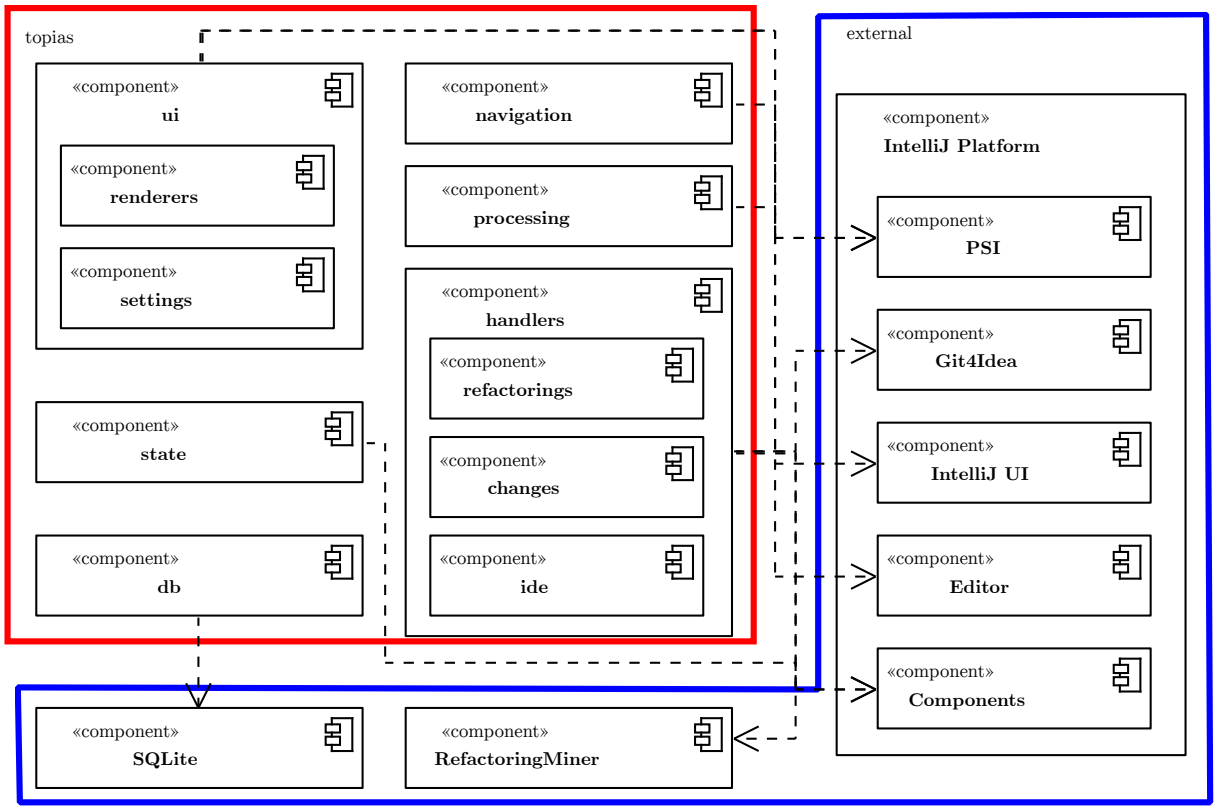

Рис. 5: Архитектура компонентов проекта

содержащихся в них. Объект класса `Change`, представляющий изменение, содержит информацию о номерах строчек, которые были изменены. С помощью этого происходит выбор измененных методов. Таким образом формируется коллекция измененных в рамках коммита методах. Кроме того, если метод был добавлен, перемещён или удален, то он дополнительно сохраняется в хранилище в оперативной памяти, чтобы после разбора рефакторингов можно было убедиться в том, что это действительно новый или удаленный метод, а не метод, подвергнувшийся, например, рефакторингу “Перемещение метода”. Это сделано в связи с тем, что инструмент RefactoringMiner, который используется для отслеживания рефакторингов, не всегда обнаруживает их корректно.

Для каждого коммита с помощью API RefactoringMiner можно получить коллекцию, содержащую информацию о произведенных рефакторингах. Каждый элемент полученной коллекции является наследником интерфейса `Refactoring`, который содержит полную информацию о методе до и после рефакторинга. Обработка каждого объекта произ-

водится в соответствии с его типом. Поскольку информация о методе в наследниках `Refactoring` представлена с помощью экземпляров классов из API среды разработки Eclipse, то обработчик преобразовывает полученные объекты к более удобному и простому формату. После завершения обработки коммита с помощью RefactoringMiner имеется коллекция с помощью которой можно установить однозначное соответствие между методами старой ревизии, к которым были применены рефакторинги, и методами новой ревизии. Производится обход коллекции с целью удалить информацию о методах, к которым были применены рефакторинги, из хранилища методов. После этого с помощью имеющихся коллекций можно скорректировать данные, хранящиеся в базе данных.

2.3. База данных

Взаимодействие с базой данных происходит через JDBC драйвер. От использования инструментов ORM (object-relational mapping), таких как Hibernate[3] или MyBatis[4], было принято решение отказаться ввиду относительной простоты схемы базы данных и производимых к ней запросов.

Схема базы данных представлена тремя основными таблицами:

- журнал изменений методов, каждая запись которой представляет информацию о дате, авторе изменения, идентификаторе метода, который был изменен, а также ветке, в которой было сделано изменение;

- словарь методов, в котором хранится идентификатор метода, по которому данная таблица связана с журналом изменений с целью облегчить обновление данных при рефакторинге метода, полная сигнатура метода и файл, в котором данный метод расположен;

- таблица со статистикой, в которой собраны данные о количестве

[3]https://hibernate.org/

[4]http://www.mybatis.org/mybatis-3/

изменений, сгруппированные по сигнатуре метода, дате, округленной до дня, и ветке.

Также имеется представление таблицы со статистикой, объединенной со словарем методов для удобства получения данных при генерации визуальных элементов.

Полная информация о каждом методе, который встречался при разборе хотя бы одного из коммитов, хранится в таблице-словаре. Данная таблица содержит уникальный идентификатор метода, его полную сигнатуру и имя файла, в котором метод расположен. Индекс в таблице сделан по сигнатуре метода, поскольку при заполнении журнала изменений нужно получить идентификатор метода. В словарь добавляется информация о методах, которые были помечены как добавленные, удаляются записи о методах, которые были удалены в коммите, а информация о методах, к которым были применены рефакторинги, обновляется. После этого можно получить идентификаторы методов, которые были изменены в рамках коммита, и сформировать данные для заполнения журнала изменений.

Статистика о количестве изменений хранится в отдельной таблице, а её подсчёт полностью делегирован базе данных и осуществляется инкрементально с помощью UPSERT запросов. После вставки очередной порции данных в журнал изменений происходит агрегация этих данных по сигнатуре метода, дате и названию ветки репозитория, и обновление статистики. Хранение подсчитанной статистики по дням позволяет легко получать статистические данные и строить гистограммы с количеством изменений по дням выбранного периода, а также производить нормализацию данных и определять наиболее часто изменяемые методы всего проекта.

2.4. Визуализация

Для визуализации истории изменений используются неизменяемые метки с информацией о количестве изменений, добавляемые в редактор над сигнатурой метода, а также список ссылок на наиболее часто

изменяемые методы проекта, отображающийся в отдельном окне в интерфейсе IDE.

2.4.1. Неизменяемые метки с информацией

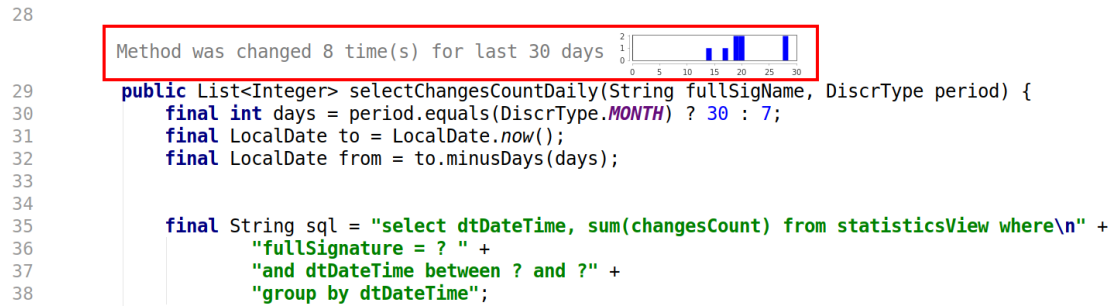

Рис. 6: Представление информации об изменениях метода

Информация об изменениях методов визуализируется с помощью добавления новых элементов в `InlayModel` редактора кода. Метка представляет собой текст и гистограмму с количеством изменений на каждом из дней интервала (рис. 6). Гистограмма генерируется с помощью библиотеки JFreeChart [8].

2.4.2. Список наиболее часто изменяемых методов проекта

Список из 10 наиболее часто изменяемых методов проекта за выбранный период времени отображается в отдельном окне, по умолчанию расположенном в нижней части интерфейса IDE (рис. 7, 8). Методы отсортированы по убыванию числа изменений. Окно может быть перемещено пользователем на правую или левую панель инструментов IDE. При клике на элемент списка курсор перемещается на объявление выбранного метода. Навигация реализована с помощью PSI.

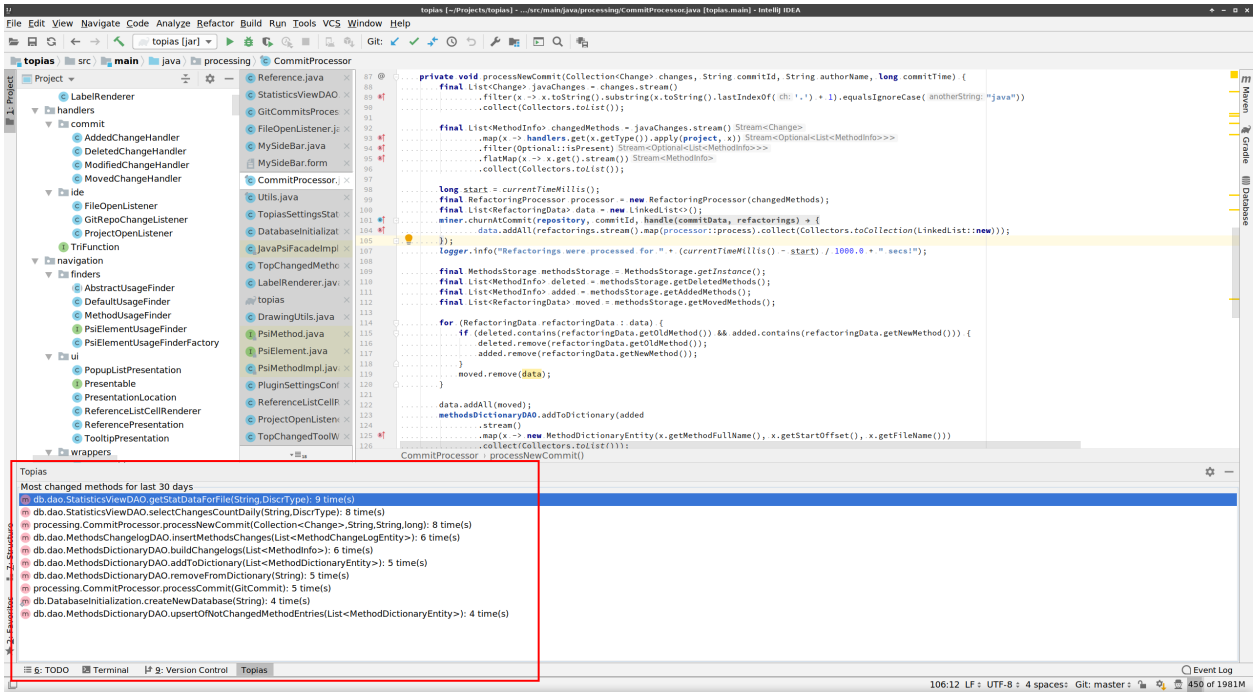

Рис. 7: Список наиболее часто изменяемых методов проекта, расположение

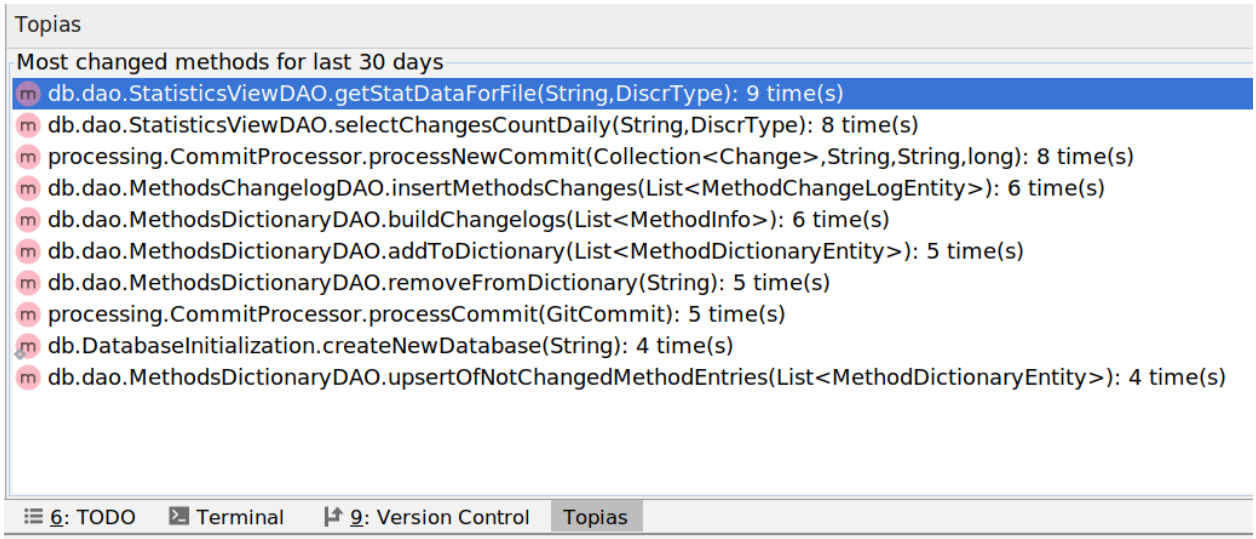

Рис. 8: Список наиболее часто изменяемых методов проекта

Заключение

В ходе данной работы были достигнуты следующие результаты:

- реализован плагин для сбора и визуализации статистики о частоте изменений методов в системе контроля версий;
  - `https://github.com/ml-in-programming/topias`;
  - `https://github.com/ml-in-programming/topias/releases/tag/v1.0-beta`;
- решение интегрировано с RefactoringMiner для отслеживания рефакторингов;
- собранная статистика отображается в виде текстовых меток и графиков частоты изменений;
- сделан доклад на конференции SEIM-2019.

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 / ACM. — 2018. — P. 483–494.

[2] Automatic detection of refactorings for libraries and frameworks / Danny Dig, Can Comertoglu, Darko Marinov, Ralph Johnson // Proceedings of Workshop on Object Oriented Reengineering (WOOR'05). — 2005.

[3] Classifying code changes and predicting defects using changegenealogies / Kim Herzig, Sascha Just, Andreas Rau, Andreas Zeller // Technical Report, Tech. Rep. — 2013.

[4] Consortium SQLite. SQLite // SQLite web-site. — Access mode: `https://www.sqlite.org/index.html` (online; accessed: 20.05.2019).

[5] Detect related bugs from source code using bug information / Deqing Wang, Mengxiang Lin, Hui Zhang, Hongping Hu // 2010 IEEE 34th Annual Computer Software and Applications Conference / IEEE. — 2010. — P. 228–237.

[6] Giger Emanuel. Fine-grained Code Changes and Bugs: Improving Bug Prediction : Ph. D. thesis / Emanuel Giger ; Verlag nicht ermittelbar. — 2012.

[7] How do fixes become bugs? / Zuoning Yin, Ding Yuan, Yuanyuan Zhou et al. // Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering / ACM. — 2011. — P. 26–36.

[8] JFree. JFreeChart // JFree web-site. — Access mode: `http://www.jfree.org/jfreechart/` (online; accessed: 20.05.2019).

[9] Janes Andrea, Mairegger Michael, Russo Barbara. code_call_lens: raising the developer awareness of critical code // Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering / ACM. — 2018. — P. 876–879.

[10] JetBrains. Git4Idea // Git4Idea Github repository. — Access mode: `https://github.com/JetBrains/intellij-community/tree/master/plugins/git4idea` (online; accessed: 20.05.2019).

[11] Johnson Philip M. You can't even ask them to push a button: Toward ubiquitous, developer-centric, empirical software engineering // The NSF Workshop for New Visions for Software Design and Productivity: Research and Applications. — 2001.

[12] Ref-Finder: a refactoring reconstruction tool based on logic query templates / Miryung Kim, Matthew Gee, Alex Loh, Napol Rachatasumrit // Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering / ACM. — 2010. — P. 371–372.

[13] What is the connection between issues, bugs, and enhancements?: lessons learned from 800+ software projects / Rahul Krishna, Amritanshu Agrawal, Akond Rahman et al. // Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice / ACM. — 2018. — P. 306–315.

[14] Xing Zhenchang, Stroulia Eleni. The JDEvAn tool suite in support of object-oriented evolutionary development // Companion of the 30th international conference on Software engineering / ACM. — 2008. — P. 951–952.

[15] Суворов Егор Фёдорович. Улучшение структурной замены кода в IntelliJ IDEA.