Санкт-Петербургский государственный университет

ЮМАТОВ Владимир Константинович

Выпускная квалификационная работа

Инструментальная поддержка процесса разработки проекта OpenJDK

Уровень образования: бакалавриат

Направление *09.03.04 «Программная инженерия»*

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

Научный руководитель:
к.ф.-м.н., доцент Д. В. Луцив

Консультант:
ст. преп. А. П. Козлов

Рецензент:
Старший Программист ООО «Азул Системс» Ю. А. Нестеренко

Санкт-Петербург
2023

Saint Petersburg State University

Vladimir Iumatov

Bachelor's Thesis

Tooling support for the OpenJDK development process

Education level: bachelor

Speciality *09.03.04 «Software Engineering»*

Programme *CB.5080.2019 «Software Engineering»*

Scientific supervisor:
C.Sc., docent D.V. Luciv

Consultant:
senior lecturer A.P. Kozlov

Reviewer:
Senior Software Engineer at «Azul Systems, LLC» Y.A. Nesterenko

Saint Petersburg
2023

Оглавление

1. Введение

Проект OpenJDK представляет собой свободную эталонную реализацию платформы Java SE с открытым исходным кодом. Основная разработка проекта ведется для "текущей" версии [1] JDK, но дополнительно требуется поддерживать и предыдущие версии платформы для стабильности и безопасности проектов, их использующих. Для этой цели существуют так называемые Update Releases - выпуски обновлений, в которых исправляются ошибки и закрываются уязвимости предшествующих версий OpenJDK.

Но существует проблема: несмотря на хорошо выстроенный и документированный процесс разработки этих выпусков, в нем еще сохраняются части, которые нужно делать вручную [9].

Например, некоторые исправления из основного репозитория переносятся в стабильные релизы "в лоб": те же изменения делаются в тех же файлах. Лучше всего это работает с последними версиями JDK, но чем старше поддерживаемый выпуск, тем больше его код будет расходиться с основным репозиторием. Соответственно, растёт вероятность, что исправление потребует больше ручной работы и тестирования, нахождения зависимости переносимого изменения от какого-либо из предыдущих, не интегрированных в целевую версию. Автоматизация этих действий позволила бы ускорить добавление исправлений и упростить процесс внесения изменений для сообщества разработчиков.

Таким образом, данная работа посвящена разработке методов и инструментов для автоматизации частей процесса разработки выпусков обновлений проекта OpenJDK, в частности, процесса переноса исправлений из текущей версии в предшествующие.

[1] https://github.com/openjdk/jdk

2. Постановка задачи

Целью работы является ускорение и упрощение переноса исправлений в выпуски обновлений OpenJDK с помощью автоматизации ручных часто повторяемых действий. Для достижения цели были выделены следующие задачи.

1. Провести обзор процесса разработки проекта OpenJDK Updates и выделить области для автоматизации.

2. Разработать методы и инструменты, которые будут использоваться для автоматизации процесса разработки.

3. Провести тестирование полученных инструментов.

3. Обзор

3.1. Цикл выпуска OpenJDK

Процесс разработки и внедрения новой функциональности в OpenJDK является непрерывным и ведется в главном репозитории проекта [2], однако существует процесс для выпуска новых стабильных версий, которые являются ответвлениями (forks) основного репозитория.

Этот процесс состоит из следующих этапов [1]:

1. `Start of the release` - разработка и внедрение функциональности следующего релиза (мажорной версии JDK) в основной репозиторий OpenJDK в течение 6 месяцев;

2. `Ramp Down Phase 1` - из основного репозитория ответвляется стабилизационный, в котором далее идёт нахождение и исправление ошибок и стабилизация выпуска. Все большие изменения для будущего выпуска должны быть внесены в основной репозиторий до этого момента. Параллельно в основном репозитории начинается цикл подготовки релиза, следующего за данным;

3. `All Test Run` (ATR) - длится 6 недель и начинается одновременно с RDP1. В этот период все проводятся все запланированные для выпуска тесты;

4. `Ramp Down Phase 2` (RDP2) - в этой фазе исправляются только самые критичные ошибки, подготавливается предвыпускная версия;

5. `Release candidate` (RC) - стабильная сборка выпускается в качестве предвыпускной версии. Эта сборка дополнительно тестируется и, если ошибок не выявлено, она же используется для стабильного выпуска. Если ошибки выявлены, они исправляются, и новая сборка становится предвыпускной версией;

[2] https://github.com/openjdk/jdk

6. `General Availability` (GA) - конец цикла, сборка становится официальной и доступной публично.

3.2. Цикл разработки выпусков обновлений

Когда выпуск входит в стадию RDP2, для него должен быть создан репозитории обновлений `jdk{version}u` и `jdk{version}u-dev`, в котором с этого момента будет вестись разработка для устранения проблем и уязвимостей данной версии на протяжении периода поддержки выпуска. Основная часть вносимых в этот репозиторий изменений - перенос исправлений из основного репозитория и более новых мажорных версий OpenJDK. Некоторая новая функциональность может попадать в выпуски обновлений, но редко, и описание этого процесса мы опустим.

Цикл выпуска стабильных версий несколько отличается от выпуска текущих версий: разработка идет в течение двух-трёх месяцев в репозитории `jdk{version}u-dev`, затем переходит в стадию Ramp Down с заморозкой кода в `jdk{version}u`, после чего в репозитории для разработки начинается подготовка к следующему выпуску, а спустя примерно месяц тестирования выпускается стабильная версия из сборки репозитория `jdk{version}u`.

3.3. Перенос исправлений

Процесс создания и принятия перенесённых из основного репозитория исправлений в поддерживаемые стабильные версии JDK происходит следующим образом [4]:

1. создаётся репорт на багтрекере проекта [3] в разделе JDK для актуальной версии Java, и его исправление обсуждается в электронной рассылке `jdk-updates-dev`;

2. если исправление актуально для какой-либо из более старых поддерживаемых версий JDK, на репорт ставится ярлык `jdk{version}u-fix-request`;

[3] https://bugs.openjdk.org/

3. автоматически, с помощью бота, заводится отдельный репорт для нужной версии JDK и ставится в раздел «backports» оригинального репорта;

4. создаётся Pull Request в репозитории `jdk{version}u-dev` с кодом исправления. Название PR должно включать в себя номер оригинального репорта;

5. если исправление одобрено сопроводителем проекта и на оригинальный репорт поставлен ярлык `jdk{version}u-fix-yes`, оно интегрируется в ветку master, откуда позже попадает в репозиторий `jdk{version}u` и стабильный выпуск соответствующей версии.

Остановимся отдельно на процессе создания и тестирования изменений в репозитории разработки выпуска обновлений. Для разработчика, не имеющего прямого доступа на запись в репозитории OpenJDK этот процесс в самом простом варианте включает следующие шаги:

1. создать собственное ответвление от репозитория `jdk{version}u-dev`;

2. создать в этом ответвлении ветку для внесения изменений;

3. используя либо встроенные команды git, либо инструменты командной строки проекта SKARA [4], перенести изменение из основного репозитория;

4. внести дополнительные изменения, если требуется;

5. сделать коммит с сообщением «Backport <SHA-хэш оригинального коммита> »;

6. протестировать изменения. Этот шаг включает:

    - `Tier 1` - набор тестов для поверхностного тестирования всех основных частей JDK;

[4] https://wiki.openjdk.org/display/SKARA/CLI+Tools#CLITools-Installing

- выбранные вручную наборы тестов, проверяющих части системы, в которые вносятся изменения;
- новые регрессионные тесты, добавленные в изменении;
- `Tier 2` (при наличии ресурсов) - расширение Tier 1, включающее дополнительно наборы более глубоких и нестабильных тестов;

7. вручную или через инструменты SKARA создать Pull Request из своего репозитория в `jdk{version}u-dev`;
8. по умолчанию, описание PR будет автоматически отправлено в рассылку jdk-updates-dev, но его так же можно отправить в другие рассылки с помощью комментария с текстом «/label», оставленного в обсуждении.

Если исправление для корректной работы требует изменений, внесённых позже, или зависит от какого-либо предыдущего, то весь вышеописанный процесс нужно сначала проделать для первого изменения, а затем повторить для зависимого, но базируясь не на основной ветке репозитория, а на ветке с внесенным первым изменением. Более того, такие зависимости требуется искать вручную с помощью багтрекера проекта, либо методом проб и ошибок.

3.4. Аналоги

3.4.1. Linux

Проектов с открытым исходным кодом с похожими проблемами, особенно настолько больших и важных, как OpenJDK, существует не так много. Это, в свою очередь, означает, что найти прямые аналоги для нашей работы может быть крайне сложно. Самым известным и очевидным сравнением становится ядро Linux: большой проект с множеством внутренних правил и специальных используемых инструментов. Так же у OpenJDK и Linux схож общий процесс выпуска и поддерж-

ки стабильных версий программного обеспечения, что даёт нам больше точек для сравнения.

Судя по словам ключевых разработчиков Linux [2], среди многочисленных ботов, используемых для автоматизации в проекте ядра Linux существует и такой, который (по крайней мере частично) выполняет поставленную нами задачу: применить изменение и, в случае неудачи, найти возможные предыдущие изменения, от которых зависит данное. При дальнейшем изучении стало ясно, под ботом здесь скорее всего подразумевается проект одного из разработчиков, отвечающих за поддержку стабильных выпусков ядра. По сути, он является набором файлов, где каждый файл назван по хэшу SHA коммита и содержит все коммиты-зависимости, которые нужно переносить группой для чистого переноса данного коммита. Эти файлы генерируются с помощью утилиты git bisect, но как именно - неизвестно, так как скрипта для генерации этих файлов в публичном доступе найти не удалось.

Таким образом, можно сделать вывод, что инструменты, решающие схожие с нашей задачи у разработчиков ядра есть, но они не документированы, не интегрированы в систему автоматизированного тестирования и используются только несколькими людьми для экономии времени на проверку переносимых изменений.[8]

3.4.2. Git-deps

С другой стороны, есть инструменты, выполняющие какую-то часть работы. Например, к ним относится утилита командной строки git-deps [5] - с помощью истории изменений в конкретном патче (используя git blame) она выстраивает цепочку коммитов, которые изменяли те же строки файлов, что и данный на вход патч.

С одной стороны, это действительно даёт нам цепочку коммитов, которые нужно применить для чистого накладывания нашего патча, с другой - результаты получаются слишком обобщёнными, часто представляя бóльшее количество зависимостей, чем нам бы хотелось, часть

[5]https://github.com/aspiers/git-deps

из которых не обязательна для чистого накладывания конкретного патча, к чему мы и стремимся. Так же инструмент будет упускать логические зависимости, которые невозможно получить только из истории изменений.[3]

4. Разработка инструментов

4.1. Архитектура

Разработанный инструментарий состоит из нескольких скриптов, написанных на Python и Bash, которые логически объединяются в три частично пересекающиеся группы: первая отвечает за анализ изменений в коде и построение чисто применяемой цепочки зависимостей коммитов; вторая - это расширение первой, позволяет нам использовать дополнительные инструменты и эвристики для возможного исправления конфликтов слияния и предоставления информации пользователю, а дальше эта информация и некоторый пользовательский ввод могут быть использованы третьей группой для окончательного наложения желаемых изменений.

Для удобства использования, в каждой группе есть скрипт, который является "входной точкой" для пользователя и запускает все остальные части группы по мере необходимости. Для взаимодействия с пользователем и предоставления ему собранных инструментами данных был выбран формат Yaml как один из наиболее простых для восприятия форматов как для человека, так и для машины.

Основными источниками данных для разработки и анализа применяемых эвристик являются система контроля версий Git, используемая во всех подпроектах OpenJDK, документация OpenJDK, в частности всё, что относится к формализации процесса разработки, система отслеживания ошибок Jira и её открытый REST API и сервис для командной разработки Github, его REST API и система предложения изменений (Pull Requests) и некоторые комментарии, которые оставляют специальные боты в этих системах.

4.2. Методы автоматизации

В ходе экспериментов и разработки было выделено несколько паттернов и эвристик, опираясь на которые можно выполнять поставленные перед нами задачи.

1. Самый простой случай не чистого внесения изменения - это конфликт слияния в строках авторского права, а конкретно - несовпадение года обновления файла. Такое изменение считается чистым по установленным правилам проекта OpenJDK Updates, но исправлять конфликт всё равно нужно вручную. Наш инструмент автоматически с помощью регулярного выражения найдет и исправит такую ситуацию без вмешательства пользователя.

2. Конфликты иногда бывают в комментариях к коду, что не влияет на сборку и исполнение проекта. Наши инструменты дают пользователю возможность применить патч игнорируя конфликтующие строки с комментариями, что может быть полезно, например, для быстрого тестирования гипотезы по конкретному патчу. Большим исключением из этого правила являются комментарии в файлах с тестами: в них могут указываться особые параметры для запуска теста, и такие конфликты мы игнорировать не будем.

3. С комментариями в тестах связан еще один применяемый нами алгоритм: во многих тестах есть отдельная строка, обозначенная `@bug`, в которой указаны идентификаторы ошибок, которые в той или иной мере проверяются данным тестом. В этой строке возникают конфликты, которые зачастую решаются простым добавлением идентификатора переносимого изменения в конец строки.

4. Применение файлов патчей в git реализовано достаточно консервативно, предположительно чтобы было как можно меньше шансов на то, что результирующий код будет некорректным, а пользователь об этом не узнает. Поэтому возникают ситуации, когда изменение можно перенести чисто, но git по-умолчанию с этой задачей не справляется. Git ориентируется на несколько строк контекста для определения положения конкретного изменения. Если этот контекст не совпадает в файле патча и в файле, на который этот патч накладывается, то возникает конфликт слияния.

На выручку нам приходит утилита patch [6], у которой есть параметр fuzz. Он позволяет игнорировать заданное количество строк контекста и, соответственно избегать некоторых конфликтов. С другой стороны, этот параметр может сделать так, что изменение наложится не на нужные нам строки, поэтому данный метод используется только при конфликте слияния, и только на строках, где этот конфликт появился.

5. По аналогии с опцией fuzz утилиты patch мы можем проанализировать другие возникающие конфликты и численно оценить (от 0 до 100 процентов), насколько большие изменения они вносят, используя расстояние Левенштейна [5]: таким образом мы можем выделить наиболее весомые (с точки зрения модификации текста) изменения и обратить внимание пользователя на них, если наши инструменты не смогли автоматически конфликт разрешить.

6. Еще одно применения расстояния Левенштейна заключается в следующем: если изменяемые строки совпадают или почти совпадают в файле, на который накладывается изменение, и в версии файла, которая была "родительской" по отношению к взятому из основного репозитория коммиту, но конфликт всё равно возникает, мы можем с уверенностью разрешить конфликт в пользу данного на вход патча, так как он, очевидно, вносит одно и то же изменение в кодовые базы, но возможно с сильным отличием в нумерации строк.

7. Так же, часто конфликты возникают из-за того, что какого-либо файла не существует в версии OpenJDK, на которую мы переносим изменение, так как коммит, создающий его, не был перенесен в эту версию. Такой конфликт в проекте принято разрешать удалением изменения несуществующего файла из патча, а не переносом создающего коммита, поэтому наши инструменты по-умолчанию поступают аналогично.

[6] https://www.man7.org/linux/man-pages/man1/patch.1.html

8. Если ни одна из описанных выше эвристик не позволяет нам разрешить конфликт слияния, то далее мы пытаемся найти в истории системы контроля версий такие изменения, перенеся которые мы сможем избавиться от конфликтов. Другими словами, мы ищем такую цепочку патчей-зависимостей, которая в совокупности даст нам чисто применить данный на вход коммит.

4.3. Реализация инструментов

Так как мы во многом полагаемся на git и его инструменты командной строки, для реализации наиболее простых методов был выбран Bash, а для более сложных операций используется Python 3.9. На момент разработки и тестирования наиболее актуальной стабильной версией OpenJDK являлась JDK 17, поэтому именно на ней мы проводили разработку и тестирование, но все описанные методы и инструменты с тривиальными поправками переносимы на любые другие поддерживаемые версии.

Как мы отмечали ранее, разработанные скрипты делятся на три группы, которые далее мы опишем подробнее.

4.3.1. Поиск зависимостей

Основной задачей первой группы является поиск чисто переносимых зависимостей для данного на вход идентификатора изменения. Общий алгоритм работы показан на Рис. 1. В первую очередь скрипт попробует найти соответствующий идентификатору коммит в основном репозитории и применить его напрямую на кодовую базу репозитория обновления. В случае возникновения конфликта в ходе этой операции, мы пытаемся найти зависимости, которые помогут нам наложить данный патч чисто. Начинаем мы этот процесс с разбора конфликтующих строк (вывода git diff), чтобы затем с помощью git blame узнать, какой коммит (или коммиты) последним менял эту часть кода. В данном случае Git blame запускается части коммитов основного репозитория, которая начинается с последнего совпадающего по хэшу SHA-1 в обо-

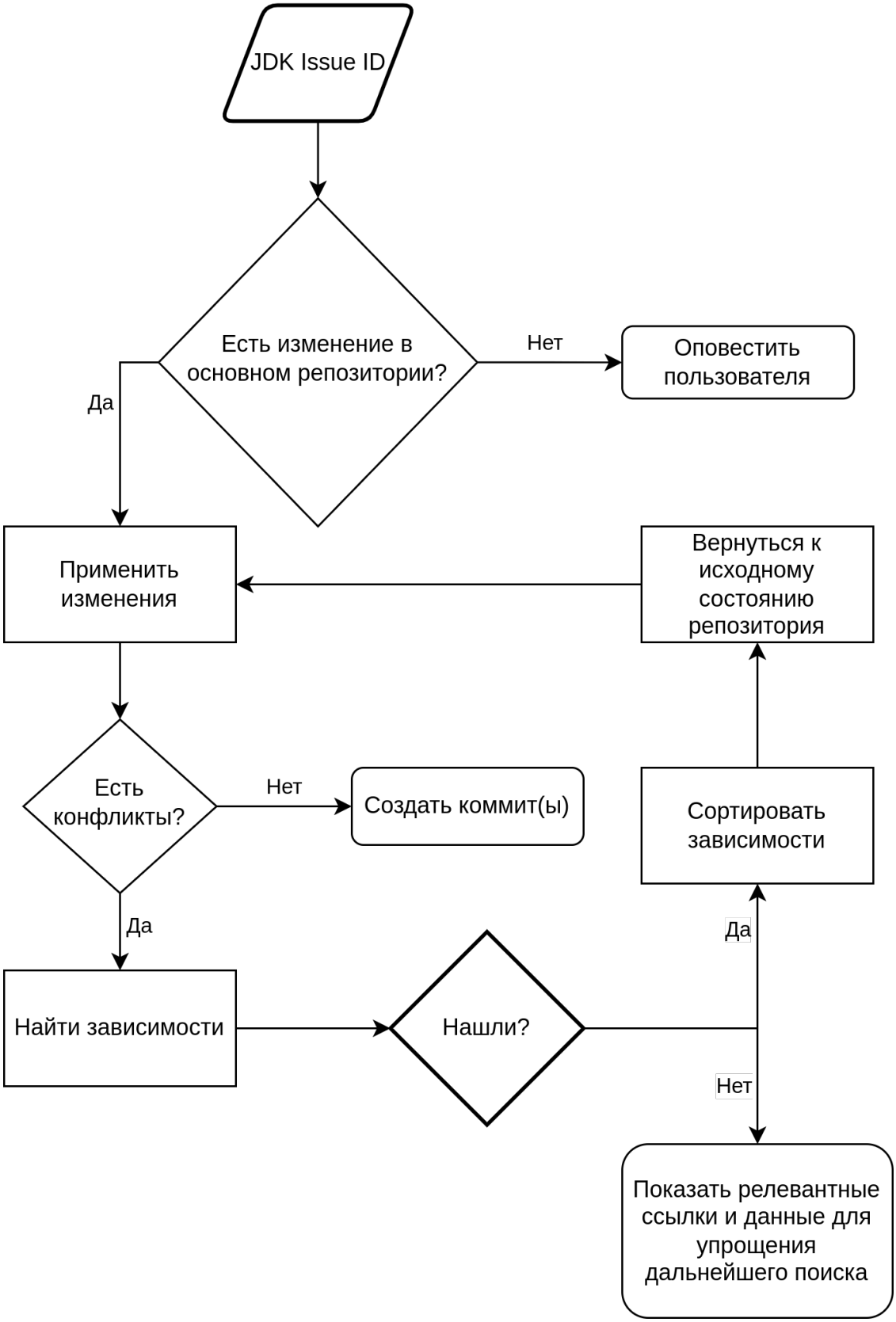

Рис. 1: Общий алгоритм работы первой группы инструментов

их репозиториях изменения. Затем все найденные таким образом коммиты проверяются на репозитории обновления: есть вероятность, что они уже были перенесены в него, и мы можем исключить их из списка возможных зависимостей. Если нашлось более одного возможного изменения-зависимости, то мы берем их все и сортируем таким образом, чтобы они переносились в том же порядке, в каком они вносились в основной репозиторий. Так как изменения с конфликтом только в годах авторского права считаются в проекте чистыми [6], мы так же будем считать их чистыми и автоматически исправлять возникающий конфликт при работе первой группы скриптов. Стоит сказать, что есть еще один способ поиска зависимостей изменения: через мета-данные соответствующего Pull Request в Github. В данном случае мы пользуемся тем фактом, что целевой веткой Pull Request должна быть ветка, от которой задача зависит [7]. Однако это работает только в том случае, когда зависимое изменение было предложено до того, как второе изменение было принято, что является довольно редкой ситуацией, однако позволяет нам иногда находить зависимости, которые не нашлись бы через систему контроля версий.

4.3.2. Дополнительный анализ конфликтов

Вторая группа инструментов является расширением первой: здесь мы добавляем больше возможностей по анализу возникающих конфликтов. Как говорилось ранее, одним из способов избавиться от конфликта слияния является применение утилиты patch с относительно высоким показателем параметра fuzz. Именно этот подход мы и будем использовать: генерируем из данного на вход коммита файл с изменениям, а затем итеративно применяем его с помощью утилиты patch, каждый раз повышая параметр fuzz. Таким образом, мы применяем все не конфликтующие части патча наиболее безопасным способом (при низком значении fuzz), но при этом есть вероятность, что конфликт разрешится при более высоких значениях параметра. Так же мы вводим свой собственный аналогичный параметр: hunk-fuzz. Он позволяет нам представить численным значением процент, на который конфликтующие

строки разнятся между собой. Для его вычисления мы используем расстояние Левенштейна, которое вычисляется после разбора строк конфликта, которые создал git. С помощью этой характеристики мы можем дать понять пользователю, насколько лего или сложно будет разрешить конкретный конфликт слияния. Все собранные данные в итоге помещаются в специальный Yaml-файл, в котором пользователь может посмотреть интересующие его характеристики, а так же исправить данные инструментами рекомендации относительно того, стоит применять конкретное конфликтующее изменение или нет.

4.3.3. Частичное применение изменений

После того, как пользователь принял решения о том, какие части патча применять, а какие игнорировать, созданный второй группой скриптов Yaml-файл подаётся на вход третьей группе, которая согласно этому файлу применяет указанные пользователем изменения.

5. Тестирование

В ходе поиска по историческим данным проекта для тестирования прототипа было выбрано несколько задач, которые приведены в таблице 5. Наши инструменты успешно прошли тестирование на данных задачах, устранив конфликты слияния.

Идентификатор задачи	**Описание**	**Результат работы скриптов**
JDK-8301170	переносится чисто, но присутствует конфликт в авторском праве	Изменение применяется успешно без ручного вмешательства, включая строку с авторским правом
JDK-8274527	конфликты слияния в нескольких файлах, но переносится чисто если до этого применить изменения из задач (в данном порядке) JDK-8268276, JDK-8269404, и JDK-8273459	Успешно найдена цепочка зависимостей, в итоге все изменения применяются чисто
JDK-8297590	конфликт слияния, так как изменяемый файл был создан в JDK-8296967	Зависимость успешно определена, все изменения применяются чисто

Заключение

В ходе данной работы были получены следующие результаты.

- Проведен обзор процесса разработки OpenJDK.
  - Рассмотрен подпроект OpenJDK Updates, с которым связана большая часть работы.
  - Выделены области, требующие автоматизации.
  - Рассмотрены аналоги нашей системы.
- Реализованы методы для автоматизации нахождения цепочек зависимостей и решения конфликтов слияния.
  - Инструменты для нахождения чисто применяемой цепочки зависимостей.
  - Инструменты для нахождения цепочки зависимостей с дополнительными изменениями.
  - Инструменты, использующие собранные данные и пользовательский ввод.
- Проведено тестирование на специально собранных данных.

Список литературы

[1] JDK Enhancement Proposal 3. — 2022. — URL: https://openjdk.org/jeps/3 (дата обращения: 2022-12-21).

[2] Maintaining stable stability. — 2020. — URL: https://lwn.net/Articles/825536/ (дата обращения: 2023-04-16).

[3] Textual vs. semantic (in)dependence. — 2019. — URL: https://github.com/aspiers/git-deps/blob/8cafb5cb1f181a6f220f7ce7f93be711ec160a9b/README.md#textual-vs-semantic-independence (дата обращения: 2023-05-02).

[4] Документация OpenJDK Updates. — 2022. — URL: https://wiki.openjdk.org/display/JDKUpdates/JDK+17u (дата обращения: 2022-12-21).

[5] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов // Доклады Академий Наук СССР,. — 1965. — . — Vol. 10. — P. 707.

[6] Обсуждение в проекте SKARA. — 2022. — URL: https://bugs.openjdk.org/browse/SKARA-827 (дата обращения: 2022-12-21).

[7] Обсуждение в рассылке OpenJDK. — 2021. — URL: https://mail.openjdk.org/pipermail/jdk-dev/2021-March/005232.html (дата обращения: 2023-02-12).

[8] Рассылка проекта ядра Linux. — 2020. — URL: https://lore.kernel.org/lkml/Y%2F79Tfn5kFIItUDD@sol.localdomain/ (дата обращения: 2023-04-16).

[9] Руководство по переносу исправления. — 2022. — URL: https://wiki.openjdk.org/display/JDKUpdates/How+to+contribute+or+backport+a+fix (дата обращения: 2022-12-21).