Санкт-Петербургский государственный университет

На правах рукописи

Луцив Дмитрий Вадимович

ПОИСК НЕТОЧНЫХ ПОВТОРОВ В ДОКУМЕНТАЦИИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Специальность 05.13.11
«Математическое и программное обеспечение
вычислительных машин, комплексов и компьютерных сетей»

Диссертация на соискание учёной степени
кандидата физико-математических наук

Научный руководитель:
доктор технических наук, доцент,
профессор кафедры системного программирования
КОЗНОВ Дмитрий Владимирович

Санкт-Петербург
2018

Содержание

Введение

Актуальность темы исследования. Документация современных программных проектов имеет значительные объёмы, сложную структуру и длительный жизненный цикл. При разработке и сопровождении документации в неё вносится большое количество точечных изменений, активно используется раскопирование (copy/paste). Часто документация разрабатывается коллективом авторов, состав которого со временем меняется. Все это приводит к большому количеству ошибок и рассогласованностей, а также к нарушению единого стиля, то есть к деградации качества документации. В качестве примера можно указать руководство программиста для ядра ОС Linux (Linux Kernel Documentation): проблемы качества этой документации в последнее время активно обсуждаются в Linux-сообществе [51, 108]. Одним из известных способов поддержки объёмной документации в актуальном и целостном состоянии является стандартизация и повторное использование. Стандартизация означает создание и использование шаблонов и соглашений. Повторное использование документации подразумевает выделение переиспользуемых свойств программного обеспечения (тестов, требований, функциональности модулей, методов и пр.) и унификацию соответствующих фрагментов документации. Это, в свою очередь, облегчает сопровождение документации, так как изменения вносятся один раз в определение текстового фрагмента и далее автоматически попадают во все соответствующие места текста.

Следует отметить важность неточных повторов, представляющих собой текстовые фрагменты, незначительно отличающиеся друг от друга. К появлению неточных повторов приводит широко используемый при разработке документации приём copy/paste: вначале фрагмент текста копируется, затем каждая копия как-то по-своему изменяется. Как указывали Э. Юргенс (E. Juergens) [89] и М. Омазиз (M. Oumaziz) [112], учёт неточных повторов может предоставить новые возможности для стандартизации и повторного использования документации, в том числе параметрического повторного

использования, как это предлагали К. Романовский и Д. Кознов [11, 21], М. Хори (M. Horie) [76], М. Носал (M. Nosál) и Я. Порубан (J. Porubän) [111], М. Омазиз [112].

Как при стандартизации существующей документации, так и при наладке повторного использования, важным оказывается поиск повторяющихся текстовых фрагментов. Такой поиск востребован также при формальной обработке документации, например, в процессе выделения формальных требований в техническом задании для последующей генерации тестов [7]. При этом, если для поиска точных повторов могут быть использованы различные существующие подходы, например, методы поиска клонов в программном обеспечении (ПО) [76, 89, 110, 111, 145, 150], то для поиска неточных повторов требуются специальные алгоритмы и методы. На настоящий момент такие подходы отсутствуют.

Таким образом, поиск неточных повторов в документации ПО является актуальной научно-практической задачей, решение которой способно существенно повысить качество сложной документации ПО путём предоставления различных сервисов по анализу и сопровождению документации. Также требуют изучения вопросы использования алгоритмов поиска неточных повторов в процессе реструктуризации и улучшения документации ПО.

Степень разработанности темы работы. Существуют работы, предлагающие методы разработки документации с применением повторного использования: М. Хори [76], М. Носал и Я. Порубан [111], К. Романовский и Д. Кознов [11, 21], М. Омазиз [112]. Следует также отметить модульный способ организации документации в технологии DITA [55]. Работы в этой области концентрируются вокруг форматов и языков разработки документации, разделяющих внутреннее и финальное представления документации: DITA [55], DocBook [147], JavaDoc [84], TeX [92], ReStructuredText [104], Markdown [96] и др. Такие форматы активно используются на практике, поскольку позволяют решать задачу единого представления документации (single source) и

автоматически порождать по этому представлению различные выходные документы, а также ввиду того, что соответствующие технологии обеспечивают надёжность при вёрстке больших документов. Методы повторного использования являются одним из основных источников требований к исследованиям в области автоматизированного поиска неточных повторов в документации ПО.

Существует значительное количество эмпирических исследований, посвящённых повторам в различной программной документации: Э. Юргенс [89] исследует повторы в спецификациях требований, М. Омазиз [112] — в API-документации, М. Носал и Я. Парубан [111] анализируют внутреннюю документацию (internal documentation). А. Вингквист (A. Wingkvist) с коллегами исследовали повторы в документации ПО с точки зрения управления дальнейшей разработкой документации [150]. Однако во всех этих исследованиях рассматривались лишь точные повторы, хотя и отмечалась необходимость работы с неточными. Требуется также отметить отсутствие в литературе формализации понятия «неточный повтор»: общая концепция повторно используемого контента ПО была разработана ещё в конце 1990-х П. Бассеттом (P. Bassett) [36, 37], но с тех пор были предприняты лишь незначительные дальнейшие шаги [110, 111], и на данный момент формальное определение неточных повторов отсутствует. Кроме того, осталась в стороне задача унификации большого класса документации программного обеспечения — так называемой справочной (reference) документации (различные справочники, руководства пользователей, API-документация и т.д.). Данный вид документации ввёл в 2011 году Д. Парнас (D. Parnas) [114].

Существующие методы поиска повторов в документации основываются преимущественно на технике поиска клонов в ПО [76, 89, 110, 111, 145, 150]: с этой целью используются готовые клон-детекторы, слегка адаптированные для работы с документацией. Следует отметить, что данный подход не позволяет эффективно искать неточные повторы в документации ПО, поскольку использует поиск по синтаксическому дереву программы. Кроме того, такой

подход приводит к большому количеству ложноположительных срабатываний (false positives), а также к низкому качеству найденных повторов (одна из основных проблем здесь — игнорирование структуры документа). С. Вагнер (S. Wagner) и Д. Фернандес (D. Fernández) [145] пишут о преимуществах методов обработки естественных языков (natural language processing) при формальной обработке текстовой информации программных проектов, но не приводят готовых методов для поиска повторов. Также следует отметить, что до сих пор остались не исследованными вопросы семантического анализа повторов в документации ПО.

Итак, необходима формальная модель нечетких повторов, ориентированная на разработку соответствующих алгоритмов, а также эффективные подходы для поиска неточных повторов в документации ПО, которые обеспечили бы учет семантической информации, а также были бы максимально свободны от ложноположительных срабатываний. Также требуется детально исследовать семантику повторов документации различного вида. Наконец, необходимо создание методов использования найденных повторов при улучшении документации и их стыковка с подходами к повторному использованию документации.

Объектом исследования диссертационной работы являются алгоритмы поиска точных и неточных повторов в документации ПО, подходы к сопровождению и улучшению документации, модели разработки программного обеспечения, языки разметки (markup languages), технологии и программные средства для разработки документации.

Целью данной работы является создание алгоритмов и методов для поиска неточных повторов в документации ПО для повышения её качества в процессе сопровождения. Для достижения этой цели был сформулирован ряд следующих **задач**.

1. Формализовать понятие неточного повтора в документации ПО и разработать эффективные алгоритмы поиска таких повторов.
2. Создать метод улучшения документации на основе поиска повторов.

3. Выполнить программную реализацию алгоритмов и методик, апробировать результаты исследования на реальной документации ПО.

Постановка цели и задач исследования соответствует следующим пунктам паспорта специальности 05.13.11: модели, методы и алгоритмы проектирования и анализа программ и программных систем, их эквивалентных преобразований, верификации и тестирования (пункт 1); оценка качества, стандартизация и сопровождение программных систем (пункт 10).

Методология и методы исследования. Методология исследования базируется на идеях и подходах программной инженерии, нацеленных на разработку эффективных методов создания ПО и документации.

В работе использована концепция архетипа и дельт, а также концепция настраиваемых фреймов для повторного использования вариативного контента (P. Bassett) [36]. В качестве базового средства поиска повторов был применен метод поиска клонов в ПО [35, 130], использована структура данных под названием интервальное дерево (interval tree) [42, 116] для быстрого поиска пересекающихся целочисленных интервалов, а также редакционное расстояние между строками для вычисления степени сходства тестовых фрагментов. Применена известная метафора тепловой карты (heat map) [136] для визуализации информации о найденных повторах. В качестве средств программной реализации использовались языки Python и Java.

Положения, выносимые на защиту.

1. Предложена формальная модель неточных повторов в программной документации, разработан алгоритм поиска неточных повторов в документации ПО на основе компоновки точных повторов, найденных с помощью метода поиска точных клонов ПО. Доказана корректность алгоритма.
2. Создана методика интерактивного поиска неточных повторов, позволяющая учитывать заданную экспертом семантику повторов. Создан алгоритм поиска по образцу, доказана полнота данного алгоритма.

3. Создан метод улучшения документации на основе неточных повторов, включая автоматизированный рефакторинг документации в формате DocBook.

Научная новизна представленных результатов заключается в следующем.

1. Предложенная формальная модель неточных повторов является новой, делает акцент на синтаксической структуре повторов, уточняя и формализуя концепцию повторно используемого контента, предложенную П. Бассеттом [36]. М. Носал и Я. Порубан дают определение неточного повтора [111], но оно является неформальным и оставляет в стороне синтаксическую структуру повтора. В работах Э. Юргенса с коллегами [89] и М. Омазиза с коллегами [112] лишь говорится о необходимости поиска неточных повторов, но не предлагается соответствующих формальных моделей.
2. Предложенные алгоритмы поиска неточных повторов в документации ПО новы, основываются на формальной модели повторов, что позволяет формально доказывать их свойства. В свою очередь, существующие алгоритмы поиска неточных клонов ПО не могут быть применены для решения данной задачи, так как они производят поиск по синтаксическому дереву программы, а тексты на естественных языках не могут быть эффективно проанализированы таким способом. Алгоритмы из области информационного поиска, решая аналогичную задачу, созданы для другой модели использования, решая такие вопросы как ранжирование результатов при представлении больших выдач, решение проблемы производительности на больших коллекциях документов, определение сходства/различия документов целиком (а не поиск повторяющихся фрагментов) и т.д.
3. Предложенный метод улучшения документации на основе неточных повторов является новым. Выдвинута идея использовать повторы для унификации документации и не считать их плагиатом или ненужной избыточностью. Существующие инженерные подходы к использованию

повторов ограничиваются повторным использованием (М. Носал и Я. Порубан [110, 111], М. Омазиз [112]) и верхнеуровневым анализом качества документации (А. Вингквист [150]).

Теоретическая и практическая значимость работы. Полученные результаты обобщают исследования в области поиска повторов в документации ПО, впервые предлагая формальные определения и алгоритмы. При этом использованы современные методы поиска, такие как поиск клонов в ПО.

Практическая значимость работы заключается в создании метода улучшения документации на основе неточных повторов, а также в реализации предложенных алгоритмов в рамках программного инструмента Duplicate Finder (исходные коды и примеры выложены в свободный доступ [60]).

Достоверность результатов работы подтверждается формальными доказательствами базовых свойств предложенных алгоритмов, а также инженерными экспериментами на документах реальных программных продуктов.

Результаты исследования были доложены на следующих научных конференциях и семинарах: 10th International Andrei Ershov Memorial Conference on Perspectives of System Informatics (PSI, 26 августа 2015 года, Казань), Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE, 31 марта 2017 года, Иннополис), 5th International Conference on Actual Problems of System and Software Engineering (APSSE, 15 ноября 2017 года, Москва), семинар в ИПМ им. М.В. Келдыша РАН (05 декабря 2017 года, Москва), семинар в СПбГТУ (08 декабря 2017 года, Санкт-Петербург).

Дополнительной апробацией результатов является поддержка исследований, представленных в диссертации, грантом РФФИ №16-01-00304 «Управление повторами при разработке и сопровождении документации программного обеспечения».

Публикации по теме диссертации. Все результаты диссертации опубликованы в 8-ми печатных работах, из них 5 зарегистрированы в РИНЦ, 3 статьи изданы в журналах из «Перечня российских рецензируемых научных журналов, в

которых должны быть опубликованы основные научные результаты диссертаций на соискание учёных степеней доктора и кандидата наук», 3 статьи опубликованы в изданиях, входящих в базы цитирования Scopus и Web of Science.

Работы [16, 19, 17, 18, 90, 91, 101] написаны в соавторстве. Личный вклад автора в данных публикациях заключается в следующем. В работе [19] автор участвовал в разработке алгоритма поиска неточных повторов на основе N-грамм, а также в экспериментах, соавторы создали и реализовали алгоритм. В работах [16, 17] автор разработал и реализовал алгоритм поиска неточных повторов на основе поиска клонов, соавторы участвовали в создании идеи алгоритма, разработке процесса и настройки инструмента поиска клонов. В работах [90, 101] автору принадлежит идея доработки алгоритма поиска неточных повторов на основе клонов, реализация этих доработок. Соавторы занимались экспериментами. В работе [91] автор разработал методику применения поиска неточных повторов для улучшения документации, соавторы предложили идею этой методики, сделали обзор литературы и эксперименты. В работе [18] автору принадлежат идеи визуальных метафор, предложенных для отображения результатов иерархического сравнения документов, соавторы предложили концепцию создания целевых сервисов на базе поиска повторов для разработки и сопровождения офисной документации.

Объем и структура работы. Диссертация состоит из введения, пяти глав, заключения и приложения. Полный объем диссертации — 122 страницы текста с 16 рисунками и 5 таблицами. Список литературы содержит 153 наименования.

Глава 1. Обзор

В этом разделе рассматривается документация ПО, описывается задача поиска повторов в документации ПО, обсуждаются существующие средства поиска и методы использования найденных повторов при разработке документации ПО. Также описываются методы поиска повторов в других областях — в информационном поиске, при анализе текстов на естественных языках, при поиске клонов в ПО. Рассматриваются существующие языки и средства разметки электронных документов, которые применяются при разработке документации ПО. Приводятся примеры документации ПО, имеющей различный объём и использующей различные форматы. Наконец, описываются подходы, технологии и модели, используемые в диссертационной работе: редакционное расстояние, интервальные деревья, технологии DocBook и DocLine, средства Clone Miner и Pandoc. На основании обзора делаются выводы и обосновываются методы исследований, применённые в диссертации.

1.1. Документация программного обеспечения

Документация программного обеспечения является очень важным аспектом разработки компьютерных систем и одновременно одним из самых дискуссионных. Ещё в 1970 году одним из первых о важности документации ПО писал В. Ройс (W. Royce) [127] в своей знаменитой статье, посвящённую водопадной модели. Он считал, что для того, чтобы разработать программное обеспечение на сумму 5 миллионов долларов, должно быть написано не менее 1000 страниц документации (при этом он имел в виду проектную, а не пользовательскую, документацию). С тех пор дискуссия вокруг необходимого количества документации не прекращается. Ф. Брукс (F. Brooks) в своей знаменитой книге «Мифический человекомесяц» [2] отмечает, что на начало 70-х годов программисты так и не научились создавать хорошую документацию. Проблемы в отечественных программных проектах, связанные с документацией, отмечались многими исследователями в 90-х и 2000-х годах, например, А.Н. Тереховым [25], А.А. Шалыто [27], В.В. Липаевым [15]. В 2001 году был опубликован Agile-манифест

(авторы — К. Бек (K. Beck), А. Кокбурн (A. Cockburn), М. Фаулер (M. Fowler) и другие известные специалисты в сфере разработки ПО) [28]. Один из принципов манифеста гласил, что работающий продукт важнее исчерпывающей документации. В том же 2001 году в [141] указывалось, что никто не знает, какие виды документации необходимы и полезны для разработчиков и помогают понимаю системы, а также кто и когда должен её создавать. В 2003 году неоднозначное отношение к документации ПО отмечают Т. Лезбридж (T. Lethbridge) с коллегами [95], делая выводы на основании опроса разработчиков в различных компаниях. Наконец, в 2011 году Д. Парнас отмечает, что ситуация с документацией в программных проектах по-прежнему остаётся нерешённой [114].

Не вдаваясь глубоко в детали данной дискуссии следует отметить, что ситуация с документацией для открытых проектов (open source projects) на настоящий момент очевидно неблагополучна. Одним из самых ярких примеров является проект Linux Kernel; проблемы с его документацией активно обсуждаются сообществом [51, 108]. Также многие исследователи и практики отмечают проблемы в данном вопросе для различных видов проектов. Наконец, опыт автора по участию в различных проектах — сопровождение систем в режиме аутсорсинга, созданных сторонними командами, создание требований по доработке существующих систем, реинжиниринг систем, функционирующих на устаревших платформах, и т.д. — явно свидетельствует о наличии проблем с документацией. При этом участники проектов в целом, как правило, имеют необходимую профессиональную подготовку благодаря тому, что обучению в области программирования уделяется должное внимание при разработке и реализации современных образовательных стандартов [5, 24], но применение полученных в академической среде знаний на практике бывает затруднительным и не всегда оказывается успешным. Таким образом, необходимы дополнительные исследования, а также создание новых методов и подходов в области работы с документацией ПО.

Рассмотрим различные виды документации ПО согласно различным классификациям, имеющимся в литературе. Известный методолог программной инженерии И. Соммервил (I. Sommerville) выделил следующие виды документации:

документацию процесса и документацию продукта [133]. В первый вид документации он включил различные описания процесса разработки ПО — планы проекта, отчёты, используемые в проекте стандарты. Остановимся подробнее на втором виде документации — документации продукта, — который, следуя И. Соммервилу, состоит из пользовательской документации (user documentation) и документации системы (system documentation).

Пользовательская документация включает в себя описание предоставляемых системой сервисов, а также руководство по её установке, руководство по началу работы (getting started with the system), руководство пользователя (то есть описание функциональных возможностей системы — user manual, user reference), руководство по администрированию (system administration guide).

Документация системы включает в себя спецификацию требований, спецификацию архитектуры, описание функциональности и интерфейсов компонент, листинги кода (возможно, включая комментарии), документы по тестированию (validation documents) и руководство по сопровождению (maintenance guide).

Известный специалист по разработке документации Т. Баркер (T. Barker) [38] разделяет документацию ПО на разработческую и пользовательскую. Первый вид включает в себя внешнюю документацию, ориентированную на пользователей и заказчика, и внутреннюю документацию, описывающую статус проекта, включающую отчёты об изменениях, а также тесты, ревью и пр. Перечислим виды внутренней документации, отмечаемые Т. Баркером:

- спецификация проекта (project specification);
- план проекта (project management plan);
- документация кода (internal code documentation);
- отчёты по тестированию (test/usability reports).

Т. Лезбридж с коллегами, делая обзор ситуации с документацией ПО на основе опроса разработчиков, выделяют следующие виды документации [95]:

- спецификация требований (requirements);
- спецификация системы (specifications);

- архитектура (architectural documents);
- детальная архитектура (detailed design);
- низкоуровневая архитектура (low level design);
- документация по тестированию (testing/quality documents).

Интересный взгляд на документацию ПО предлагает Д. Парнас [114]: он разделяет документацию на описательную (narrative) и справочную (reference). Описательная документация предназначена для прочтения целиком, её читатели обычно мало знакомы с предметом и нуждаются во вводной информации и общих сведениях. Написание такой документации является, по мнению Д. Парнаса, искусством. Справочная документация предназначена для выборочного чтения, и, как правило, читатель хорошо знаком с предметной областью и не нуждается во вводной информации. В случае с этой документацией возможны технологи и инженерные подходы. Продолжая идею Д. Парнаса, следует отметить, что справочная документация, как правило, описывает наборы однотипных сущностей — элементы пользовательского интерфейса (руководства пользователей), типы и структуры данных, функции, прерывания (API документация и руководства программистов), конструкции языков программирования (руководства по языкам программирования). Реже такая документация описывает процессы, например, процесс использования драйвера: его инициализация, запуск и т.д. Следует отметить, что на практике реальная документация часто бывает смешанной, то есть может содержать описательную и справочные компоненты, при это строгость и формальность описания справочных компонент документации может быть различной.

Многие исследователи выделяют так называемую API-документацию [57, 76, 110, 111, 112, 128, 139]. Она описывает программные интерфейсы библиотек и модулей: функции, классы, типы данных и константы, и т.д. API-документация является справочной, и, как правило, её структура отражает структуру

документируемого программного обеспечения. Часто эта документация автоматически генерируется по программному коду системы такими инструментами как Javadoc [84], консолидируя комментарии в коде.

Принимая во внимание приведённые выше классификации, автор диссертационного исследования выделил следующие виды документации ПО, делая акцент на доступности соответствующих документов (последнее необходимо, поскольку в исследовании требовались для анализа документы реальных проектов).

- Спецификация требований — присутствует во всех классификациях; автор использовал данный вид документации из коммерческих проектов, в которых он принимал участие.
- Спецификация архитектуры — у И. Соммервилла и Т. Лезбриджа этот вид документации упомянут явно, у Т. Баркера информация об архитектуре системы может присутствовать в спецификации проекта и документации кода.
- Руководство пользователя — присутствует у И. Соммервилла и Т. Баркера, отсутствует у Т. Лезбриджа; последнее объясняется тем, что Лезбридж рассматривал документацию разработчиков. В целом пользовательская документация традиционно стоит особняком от документации иного вида — имеются специальные методы по её разработке [62, 94, 77, 148], и, как правило, её создают не программисты, а специальные технические писатели. Следует отметить, что данная документация легко доступна в целях исследований — как для коммерческих проектов (она обычно не составляет, в отличие от иных видов документации, коммерческой тайны), так и для открытых проектов. Отметим особый вид пользовательской документации – руководства по применению систем и отдельных утилит для Unix/Linux пользователей. Такие пользователи обычно работают в интерфейсе командной строки, являясь, фактически, программистами, поэтому соответствующая пользовательская документация сильно похожа на руководство для

программистов. Тем не менее, автор диссертационной работы включил такие документы в группу «Руководство пользователя».

- Руководство программиста — вид документации, который присутствует у И. Соммервилла как руководство по сопровождению, у Т. Баркера частично как документация кода, частично как руководство по сопровождению (внешняя документация); у Т. Лезбриджа этот вид документации может быть скомбинирован из видов описания архитектуры (однако в явном виде отсутствует). Данный вид документации важен для данного исследования, поскольку имеется большое количество открытых проектов, документация которых также открыта и доступна для анализа в рамках различных исследований. При этом в данных проектах при составлении документации особенный упор делается на описание уже созданного кода с целью облегчить открытую (open source) разработку.
- Руководство по администрированию — данный вид документации присутствует у И. Соммервилла и у Т. Баркером в пользовательской документации и отсутствует у Т. Лезбриджа. Этот вид документации был включён в классификацию в виду доступности таких документов у открытых продуктов.
- API-документация — отсутствует в явном виде у И. Соммервилла, Т. Баркера и Т. Лезбриджа; может быть отнесена к документации программного кода или к описанию низкоуровневой архитектуре, так как описывает детали кода (классы, атрибуты, функции и т.д.); также может быть отнесена к пользовательской документации, т.к. предназначена для расширения функциональности системы теми, кто использует систему. API-документация включена автором диссертационной работы в классификацию в виду широкого распространения в открытых и коммерческих продуктах, а также активных исследований проблем и методов разработки и сопровождения такой документации.

- Справочник по языку — у И. Соммервилла присутствует в документации процесса, отсутствует у Т. Баркера и Т. Лезбриджа. Для данного исследования справочники по языку интересны в виду доступности, а также ввиду широкого распространения предметно-ориентированных языков (Domain Specific Languages, DSLs) [58, 125] и соответствующих средств разработки и использования — следовательно, и соответствующих документов оказывается много.

1.2. О повторах в документации программного обеспечения

При разработке документации широко используется техника copy/paste. Функциональность ПО содержит значительные повторы, поэтому при составлении документов, при описании очередной сущности (класса, функции, атрибута и т.д.) часто копируется уже существующее описание, которое исправляется и дополняется в соответствии с новым контекстом. Часто таких копирований происходит много, они по-разному редактируются, и в результате в документации накапливается значительное количество повторов, которые оказываются неточными, являясь различными вариациями одной и той же информации. Кроме того, весьма часто раскопируются отдельные фрагменты текста, которые не связанны с общей функциональностью ПО — одинаковые приглашения к рассмотрению примеров, похожие замечания, предупреждения, общие вводные фразы в начале различных разделов и т.д.

Разные исследователи по-разному относятся к повторам в документации. В работах [89, 121] повторы рассматриваются как признак избыточности, и авторы видят в них причину снижения качества документации. Хотя в [89] указывается о том, что с одной стороны, повторы затрудняют сопровождение и чтение, но с другой стороны требуются дополнительные исследования влияния избыточности документации на скорость восприятия, так как, возможно, что во многих случаях эта избыточность наоборот, повышает понятность текста. В [11, 20, 76, 110, 111, 112] повторы в документации связываются с повторами в коде ПО и

активно обсуждается вопрос о повторном использовании документации аналогично повторному использованию кода. В [150] повторы в документации ПО используются для анализа качества документации ПО — их наличие считается признаком плохого качества документов. Итак, разное отношение исследователей к повторам в документации в значительной степени обусловлено разным характером документации, которую анализируют авторы перечисленных работ.

Таким образом, справочная документация может и должна содержать большое количество повторов в целях максимальной унификации, а также из тех соображений, что подобное должно быть описано одинаково, с минимальным использованием синонимов, минимальным наличием «параллельных мест» в тексте.

Рассмотрим подробнее работы, посвящённые анализу повторов в документации ПО.

М. Омазиз [112] анализирует API-документацию нескольких известных открытых (open source) программных проектов, созданную с применением Javadoc [84]. То есть в этом случае документация генерируется на основе комментариев. Поиск повторов авторы выполняют с помощью средства собственного средства поиска повторов в API-документации DocReuse [59], построенного использующего в качестве средства поиска программных клонов библиотеку Colibri-Java [49]. В работе представлены результаты анализа встроенной документации следующих известных открытых проектов: Apache Commons Collections, Apache Commons IO, Google-GSON, Guava, JUnit, Mockito, SLF4J. Авторы разделяют найденные повторы на две группы: в первую группу входят фрагменты, получившиеся путём copy/paste, во вторую группу входят повторы, появившиеся в документе в том случае, когда описания сходной функциональность совпали. Помимо этого, они предлагают классификацию повторов на основе того, как связаны между собой фрагменты программного кода, которым соответствуют текстовые повторы одной группы. Выделяются следующие виды обусловленных связями в исходном коде повторов в документации: в 60% случаев часть документации вызываемого метода копируется в документацию вызывающего; в 28% случаев при наследовании часть документации метода наследника

копируется из документации метода предка; в 8% случаев раскопированный исходный код одинаково документирован; наконец, в 3% случаев частично повторяется документация похожих методов. Их подход не поддерживает работу с неточными повторами, но авторы признают, что такие повторы часто встречаются и важны на практике.

М. Хори с коллегами [76] предлагают добавить возможности повторного использования в API-документации (Javadoc). Они предлагают инструмент CommentWeaver, позволяющий пользоваться возможностями повторного использования в Javadoc для Java и AspectJ. Согласно результатам апробации, проведённой с использованием исходных кодов стандартной библиотеки Java, Eclipse и Javaassist, от 4 до 20% документации Javadoc повторяется. Применение CommentWeaver сокращает объём документации приблизительно на 10%. Неточные повторы в работе также не рассматриваются.

М. Носал и Я. Парубан [111] расширяют подход из работы [76] неточными повторами. Они дают следующее определение неточных повторов, называя последние Documentation Phrases: множество фрагментов документа, которые имеют одинаковую семантику или описывают одно и то же свойство, принадлежащее документируемым элементам программного обеспечения. Таким образом, авторы сопоставляют группу повторов определённому свойству программного обеспечения (более точно, исходному коду ПО). Предлагаемый авторами инструмент поддерживает добавление стандартных Documentation Phrases в документацию Javadoc на основании *аннотаций*[1] в исходном коде на Java. Но вопрос поиска Documentation Phrases работа не затрагивает. Определение неточных повторов неформально.

[1] Аннотация в Java — это способ указывать дополнительную метаинформацию, относящуюся к сущностям программы, таким, как классы, методы и т.д. [87]. Аннотации также поддерживаются многими другими современными языками программирования, например, Microsoft C# и Python 3.

Следующая работа М. Носала и Я. Парубана [110] посвящена поиску и анализу повторов в документации в формате Javadoc с использованием модифицированного статического анализатора кода PMD [115]. В работе анализируются исходные коды и Javadoc-комментарии 5 проектов с открытым исходным кодом, в том числе Java Framework версии 8. Результаты анализа показывают большое количество найденных повторов в документации, причём фрагменты, повторяющиеся многократно, встречаются реже, чем повторяющиеся небольшое количество раз.

Э. Юргенс [89] исследует точные повторов в спецификациях требований, используя для этой цели программный пакет Clone Detective [88]. Ищутся повторы длиной от 20 слов. Авторы проанализировали 28 документов из различных проектов и классифицировали найденные повторы следующим образом: описание пошагового взаимодействия пользователя с системой, перекрёстные ссылки, спецификация графических интерфейсов, пояснение к нефункциональному требованию, описание программного интерфейса, пре/постусловие и инвариант в программе, описание настроек, функциональное требование, информация о применяемых технологиях, уточнение к требованию. Помимо самих требований, в работе анализируется их влияние на программный код. Авторами выделяются три ситуации:

- программный код содержит повторное использование, при этом повторы в требованиях не приводят к повторам в коде; например, в требованиях неоднократно упоминается, что при возникновении ошибки информация о ней записывается в журнал; соответственно, в коде имеется одна процедура записи в журнал, которая многократно вызывается из разных контекстов;
- дублирование требований соответствует дублированию программного кода; упомянутая выше функциональность о записи в журнал при возникновении ошибки реализована как раскопированный код;

- одинаковая функциональность, описанная при помощи одинаковых требований, реализуется многократно; упомянутая выше функциональность о записи в журнал при возникновении ошибки реализована несколько раз поразному.

Процесс поиска повторов в документации выглядел так. Для минимизации субъективности анализа повторов работа с каждым документом проводилась парой исследователей, распределение документов по парам производилось случайным образом. В начале анализа каждая пара описывала шаблоны для замеченных ложноположительных повторов в анализируемо документе, адресованные используемому клон-детектору. Это делалось до тех пор, пока клон-детектор переставал выдавать *ложноположительные срабатывания* — повторы, которые не представляют практического интереса. После этого запускался клон-детектор и далее его выдача анализировалась «вручную». Авторы приводят следующую классификацию ложноположительных срабатываний:

- повторы в метаданных документов — фамилии авторов, дата публикации документации и пр.;
- повторы в автоматически генерируемых разделах документации — в оглавлении, в колонтитулах и т.д.;
- «TODO»-заметки;
- стандартные названия разделов, общие для разных документов документации.

Также в работе оценивается влияние повторов на скорость чтения, а также на эффективность сопровождения требований. Авторы делают вывод о том, что повторяются в среднем 13,6% текста (отношение размера всех повторов в документе к размеру документа термин clone coverage — *покрытие клонами)*. При этом они сравнивают объём анализируемой документации с гипотетической документацией, содержащей такой же текст, но без повторов, и показывают, что в среднем, благодаря повторам, документация увеличена на 13,5% (термин blow-

up — *раздувание*). Авторы также отмечают важность неточных повторов, однако в своём исследовании их не учитывают.

С. Вагнер и Д. Фернандес [145] рассматривают задачу разработки и сопровождения различной текстовой информации программных проектов: исходного кода, документации, текстовых артефактов конфигурационного управления, заметок в системе учёта ошибок (bug tracking system) и т.д. В работе рассматриваются различные техники поиска повторов, основанные на анализе текстов на естественных языках и на поиске программных клонов. При проведении экспериментов авторы используют для анализа документации инструмент поиска программных клонов из состава анализатора программного обеспечения [50]. Также обсуждаются вопросы субъективности при «ручном» анализе и классификации найденных в документации повторов.

А. Вингквист с коллегами [150] также исследуют точные повторы в документации различных, не только программных, проектов, делая на основании количества найденных повторов выводы об уникальности и об избыточности документации. Для анализа они используют средство поиска клонов, входящее в состав пакета VizzAnalyzer [99].

А. Раго (A. Rago) с коллегами [121] ищет неточные повторы в текстовых описаниях случаев использования (use cases, см. Д.В. Кознов [9] и М. Фаулер [66]). Используются методы анализа текстов на естественных языках. Следует отметить, что данный подход ориентирован на очень простые документы, состоящие из последовательностей пронумерованных действий. Применимость подхода к более сложным документам неочевидна. Заметим, что авторы согласны с [89] в части отношения к повторам: по их мнению, повторы могут затруднять использование и сопровождение документации, а также оказывать негативное влияние на реализованный в соответствии с этими спецификациями программный код.

1.3. Методы и средства поиска текстовых повторов

Выше были рассмотрены подходы к поиску повторов в документации ПО, включая связи с программным кодом, средства повторного использования, разные виды документации и другие реалии разработки программного обеспечения. В этом разделе рассмотрим различные методы и средства обнаружения повторов в текстах — как в контексте других областей, так и в смысле низкоуровневых средств поиска, которые могут быть использованы в нашей задаче.

Методы и средства анализа текстов на естественных языках

Некоторые техники и средства анализа и обработки текста на естественных языках могут быть использованы при поиске повторов в документации.

Популярной является модель N-грамм [46] — разновидность статистической модели языка, которая рассматривает текст как множество перекрывающихся фрагментов одинаковой длины, называемых N-граммами. Модель позволяет, например, с помощью меры Жаккара [82], делать выводы о синтаксической близости текстов на основе того, насколько похожи множества их N-грамм.

Тематическое моделирование (topic modelling) [12, 138] позволяет разбивать коллекцию документов или фрагментов текстов на классы, относящиеся к различной тематике. Подробнее можно анализировать текст с использованием векторных смысловых представлений слов при помощи таких систем, как Google Word2Vec [105] или Stanford GloVe [137]. Большое количество алгоритмов анализа текстов на естественных языках реализовано в программных библиотеках NLTK [44] и Texterra [26]. Детальный анализ текста на естественном языке, включая морфологический анализ слов и синтаксический анализ предложений, может быть выполнен при помощи более мощных анализаторов, таких как SyntaxNet [140], SpaCy [75]. Сравнительный анализ нескольких таких систем представлен в [48].

В некоторых исследованиях, посвящённых анализу повторов в документации ПО, применяются средства анализа текстов на естественных языках. В работе [121] эти средства применяются для анализа случаев использования. В [145]

описывается поиск повторов в текстах программных проектов с применением N-грамм, а также средств для разметки частей речи и тематического моделирования. В работах [89, 145] стандартные алгоритмы обработки текстов на естественных языках использовались для нормализации текста — приведения форм слов к основным формам (падежам, числам, временам и т.д.)[2].

Информационный поиск

Поиску повторов в документах посвящён ряд работ в рамках тематики информационного поиска. В отличие от исследований, описанных в разделе 1.2, эти работы не относятся к программной инженерии и не рассматривают документацию ПО. Тем не менее, применяемые в них приёмы и техники анализа текстовых данных на предмет повторов интересны с точки зрения темы данной работы.

В [149] рассматриваются подходы к дедупликации[3] цифровых коллекций с использованием перцептивного хеширования[4] на основе алгоритма SimHash [47], а также и при помощи метода N-грамм. В работе анализируется контент электронных библиотек CiteSeerX и ArXiv, которые содержат сотни тысяч научных статей. Статистический анализ повторов, найденных с применением обоих подходов, доказывает их эффективность в задачах поиска повторов (точнее, повторяющихся документов).

---

[2] Использование предварительной нормализации перед поиском повторов формально позволяет говорить о поиске неточных повторов — в одну группу повторов попадают текстовые фрагменты, которые не являются идентичными. Но отличия в текстовых повторах, очевидно, нельзя сводить только к различиям на уровне форм отдельных слов или пунктуации.

[3] Дедупликация (deduplication) является разновидностью сжатия данных, которая производит замену повторяющихся фрагментов ссылками на единственный экземпляр этих данных. Этот метод активно применяется в работах, посвящённых систематизации цифровых коллекций и организации систем хранения данных [102, 131]

[4] Перцептивное хеширование (perceptual hashing) — способ построения цифрового отпечатка данных, при котором сходным данным соответствуют близкие отпечатки [107].

Работа [122] предлагает механизм обнаружения повторяющихся фрагментов Веб-страниц с целью дедупликации их контента и оптимизации кеширования[5]. Современные технологии Веб-разработки поддерживают повторное использование, но «ручное» выделение повторно используемых фрагментов Веб-страниц трудоёмко и повышает риск появления ошибок. Предложенные авторами методы повторного использования и обнаружения повторно используемых фрагментов призваны решать эти проблемы. Проведённая апробация показывает хорошие результаты для крупных информационных ресурсов, таких, как веб-сайты BBC, IBM и SlashDot. Следует отметить, что данная задача решается на больших коллекциях документов и имеет очень большую трудоёмкость (как правило, для её решения требуется вычислительный кластер).

Интересна задача поиска в наборе документов исходного документа, из которого фрагменты текста были скопированы в остальные [30]. В данной работе предлагается алгоритм поиска в коллекции документов исходных фрагментов раскопированного текста, позволяющий найти «оригинальный» документ, из которого, вероятно, они были заимствованы. В работе [40] рассматривается задача интеллектуального плагиата — перефразирования оригинального содержимого — и предлагается следующая классификация способов перефразирования: использование лексического изменения, изменение в построении предложений и изменения в модальности. Одним из результатов проведённого исследования является специальный корпус перефразирования, который может использоваться при обнаружении содержательного плагиата, когда чужие идеи выражаются «своими словами».

В [151] рассматривается задача поиска неточно повторяющихся фрагментов текста в нескольких больших коллекциях документов, содержащих научные ста-

[5] На сервере кеширование, как правило, применяется для предотвращения повторной генерации одних и тех же страниц, на клиенте — для того, чтобы не загружать одни и те же страницы многократно.

тьи, записи в блогах, произвольные веб-страницы (десятки миллионов документов на английском и китайском языках общим объёмом в сотни Гб). В работе предлагается алгоритм, который, пользуясь N-граммами, находит близкие по мере Жаккара [82] предложения в различных документах, а затем решает в отношении найденных предложений задачу выравнивания последовательностей[6]. Обе операции требуют вычислений значительного объёма. Авторы ставят эксперименты на вычислительном кластере, применяя подход MapReduce [56], и уделяют отдельное внимание производительности предлагаемого решения. Анализируя полученные результаты, авторы делают вывод об эффективности предложенного ими алгоритма.

Работа [67] посвящена анализу повторяющихся фрагментов в веб-страницах. В ней показывается, что 40-50% Веб-контента приходится на потенциальные шаблоны, причём, согласно оценкам авторов, эта доля на момент публикации работы росла со скоростью 6-8% в год. Авторы предлагают рандомизированный алгоритм[7] выявления текстовых шаблонов в коллекциях веб-страниц и делают вывод о том, что бóльшая часть Интернет-трафика приходится на стандартные данные, а не на уникальное содержимое Веб-страниц. При этом они отмечают, что источником бóльшей части ссылок между страницами также являются имеющиеся в них повторы.

Помимо того, что работы, рассмотренные в данном разделе, интересны с точки зрения поиска повторов в текстовых данных, очевидна общая с многими работами из раздела 1.2 направленность на нахождение повторов в текстовых данных. Основные цели перечисленных в данном разделе работ — оптимизация ра-

[6] Задача выравнивания последовательностей (sequence alignment) заключается в нахождении наибольшей общей подпоследовательности двух или, для, случая множественного выравнивания (Multiple Sequence Alignment), большего количества строк. Данная задача актуальна для биоинформатики и информатики [86].

[7] Рандомизированный алгоритм — процедура, в которой один или несколько шагов основаны на случайном выборе правила [3, 4].

боты программного обеспечения, повышение сопровождаемости коллекций документов, обнаружение и анализ плагиата. Кроме того, специфика предметных областей информационного поиска и программной инженерии различна: информационный поиск работает с большими (до сотен Гб) коллекциями данных, для чего часто требуется значительные вычислительные ресурсы. При поиске информации приоритетной является не полнота, а скорость работы и правильное ранжирование результатов.

Поиск клонов в программном обеспечении

В большинстве работ по поиску и анализу повторов в документации ПО [76, 89, 110, 111, 145, 150] используются средства поиска повторов в коде ПО (software clone detection tools). Рассмотрим вопрос поиска программных клонов подробнее.

Несмотря на богатый набор средств повторного использования, предоставляемый современными языками и системами программирования, проблема нахождения повторов (клонов) в исходных кодах программного обеспечения является актуальной. В попытках найти её решение многими исследователями создавались средства обнаружения программных клонов, подробный систематизирующий обзор которых доступен в [124].

Многие средства поиска повторов в исходных кодах анализируют синтаксические деревья программ. Поскольку синтаксическое дерево программы является результатом работы синтаксического анализатора, подобные инструменты работают с лишь программами, написанными на конкретных языках программирования, поддерживаемых этими инструментами. Примерами таких средств являются Deckard [85] и ClemanX [106]. Также предлагают находить программные клоны при помощи анализа синтаксических деревьев М.Х. Ахин и В.М. Ицыксон [1] и Н.Г. Зельцер [6]. Интересное решение предлагают Б. Бейгель (B. Beigel) и Д. Диль (D. Diehl) [43]. В их работе описывается фреймворк JCCD, предназначенный для написания и настройки поиска клонов при помощи синтаксических

деревьев. Но грамматику исходного языка при этом может задавать пользователь.

С точки зрения данной диссертационной работы интересен поиск повторов в текстах на естественных языках. При помощи анализа синтаксических деревьев, очевидно[8], его нельзя реализовать столь же просто, как для исходных кодов ПО. На практике это подтверждается тем, что многие упомянутые средства анализа текстов на естественных языках [32, 75, 105, 137] используют для синтаксического и семантического анализа нейронные сети, а не построенные по формальным грамматикам анализаторы.

Тем не менее, поиск повторов в текстах на естественных языках можно осуществлять при помощи универсальных средств поиска программных клонов, таких как Simian [130] и Clone Miner [35]. Универсальные средства поиска клонов рассматривают исходный код программ как обычный «плоский» текст, при этом выполняя базовый анализ текста с разбиением на токены (слова), но не выполняя синтаксического анализа. В дальнейшем будем понимать под *токеном* подстроку документа, ограниченную слева и справа неотображаемыми символами (пробелами, переводами строк, табуляцией и т.д.)[9]. Так, например, каждая из строк «FM Registers» и «Primary key» состоит из двух токенов. В отличие от определяющих синтаксис грамматик, лексические правила различных языков программирования часто бывают похожими друг на друга, что позволяет подобным инструментам не зависеть от входного языка. Общие лексические правила просты и у естественных языков: для разбиения текстов на токены достаточно распознавать алфавит языка, пробелы и знаки препинания. Simian пригоден для по-

[8] Типичные компьютерные языки относятся к классу контекстно-свободных. Для этого класса существуют простые и эффективные средства синтаксического анализа. Но естественные языки не являются контекстно-свободными [129], и простых средств однозначного синтаксического анализа для них не создано.

[9] Фактически, речь идёт об отдельных словах текста, поэтому часто вместо термина «токен» будем использовать также термин «слово».

иска повторов в любых текстовых файлах. При анализе исходных текстов программ, написанных на поддерживаемых им языках программирования, инструмент позволяет использовать дополнительные настройки наподобие «распознавать идентификаторы и игнорировать их регистр» и т.д. Более подробное описание Clone Miner будет приведено ниже.

1.4. Средства разметки электронных документов

В этом разделе рассматриваются известные языки и программные средства, предназначенные для разметки текстов. В обзоре будет сделан акцент на возможности повторного использования текстов.

В 1964 году Дж. Зальцером (J. Saltzer) была создана система RUNOFF [34]. Язык разметки системы RUNOFF позволял снабжать текст документа простыми директивами форматирования (жирный, подчёркнутый текст и т.д.). Система RUNOFF также брала на себя работу по выравниванию текста и генерации колонтитулов. В настоящее время семейство языков разметки ROFF (troff, nroff, groff) [68], основанное на языке RUNOFF и поддерживаемое системой Unix Man Pages [103], используется в сообществах Unix и Linux. Система Unix Man Pages реализует форматирование для вывода на терминал, принтер и для экспорта в виде Веб-страниц. Система поддерживает концепцию Single Source (единый источник): из одного исходного представления генерируются различные конечные форматы. RUNOFF/Unix Man Pages не предоставляет возможностей повторного использования.

Широко известен созданный Ч. Гольдфарбом (C. Goldfarb) язык SGML [71, 81]. На базе SGML был создан язык XML, а также языки разработки промышленной документации — DocBook [147] и DITA [55]. DocBook и DITA поддерживают Single Source: в качестве выходных форматов используются PDF, Eclipse Help, HTML и др. DocBook и DITA поддерживают также смысловую (семантическую) разметку. Технический писатель может оперировать такими понятиями как «раздел второго уровня», «листинг программы», «выделенный текст». Внешний вид документа определяется генератором выходного формата и

настраивается отдельно от самого документа. Такой подход называется WYSIWYM (What You See Is What You Mean), в противоположность подходу WYSIWYG (What You See Is What You Get), предлагаемому текстовыми процессорами офисных пакетов, например, Microsoft Word.

DITA предоставляет возможности повторного использования. Для этого описание каждой темы выделяется в отдельный модуль и может быть использовано повторно в различных контекстах с применением условного включения. В DocBook базовые возможности повторного использования реализованы при помощи механизма XInclude[10]; дополнительные возможности повторного использования предоставляются сторонними средствами, например, системой DocLine [11].

Для оформления научных текстов популярным является язык разметки TeX, созданный Д. Кнутом (D. Knuth) [92]. TeX позволяет добиваться типографского качества вёрстки с использованием персональных компьютеров. Базовые возможности языка TeX ориентированы на управление деталями внешнего вида текста. Система LaTeX [93], созданная Л. Лампортом (L. Lamport) и являющаяся дальнейшим развитием TeX, делает акцент на возможностях семантической разметки текста, предлагает для разработки документов классы и стилевые пакеты. TeX и LaTeX поддерживают повторное использование при помощи макроопределений и конструкций условного включения. Например, при использовании этих инструментов для написания диссертации авторы часто вставляют часть автореферата также и во введение основного текста диссертации. Но при этом требуется дополнительно отобразить ссылки на соответствующую литературу, которые, как правило, отсутствуют в автореферате в виду ограничений на его объём. Для этого повторно используемая часть автореферата оформляется отдельно и содержит в себе условные включения этих ссылок. Далее в каждом из двух текстов, в которые общая часть включается с помощью директивы «\input»,

[10] Стандартный механизм включения содержимого одного XML-файла на место тега директивы «<include>» в другом.

определяется своё значение переменной условного включения, и ссылки появляются или нет в тексте в зависимости от значения этой переменной. Аналогичные способ используется и при написании разных версий одной и той же научной статьи.

В настоящее время активно развиваются легковесные языки разметки, которые проще в освоении, чем LaTeX, DocBook, DITA. В качестве примеров таких языков можно привести языки форматирования Вики-страниц WikiWiki [53], MediaWiki [39], а также универсальные языки разметки ReStructuredText [104], AsciiDoc [22], Markdown [96]. В последнее время некоторые проекты по разработке ПО, ранее использовавшие DocBook, отказываются от него в пользу таких языков [51, 108, 152]. Форматирование текстов с помощью легковесных языках осуществляется посредством отступов, пропусков строк и простейшей псевдографики, например, символов «*» и «-» для маркированных списков, «подчёркивания» заголовков при помощи символа «=» и т.д. Многие легковесные языки также поддерживают Single Source с генерацией различных конечных представлений (Microsoft Word, HTML, LaTeX, PDF, EPub и т.д.) при помощи таких систем как Pandoc [113] и GitBook [69].

Повторное использование поддерживается Wiki-языками при помощи специальных директив, позволяющих включать в Wiki-страницу отдельно описанные фрагменты текста. Универсальные легковесные языки не предоставляют стандартных возможностей повторного использования.

Известным примером системы генерации документации ПО по исходным текстам является Javadoc [84]. Для осуществления генерации требуется наличие специальных комментариев в Java-коде. Эти комментарии могут содержать директивы для обозначения различных конструкций языка Java — параметров функций, констант и т.д. Основной выходной формат для Javadoc — HTML, но существует возможность подключать внешние модули для генерации RTF и некоторых других форматов. Javadoc не предоставляет возможностей повторного использования. Аналогичные средства генерации документации имеются на

платформах Microsoft .NET и Python, а также во многих других современных средах программирования. Следует отметить, что существуют подобные средства, которые обеспечивают работу с различными языками программирования. Например, система Doxygen [61] поддерживает C/C++, D, Fortran, Perl, Tcl и выходные форматы HTML, Microsoft HELP, RTF, LaTeX, ROFF. Doxygen ограниченно поддерживает повторное использование, позволяя при помощи специальных директив включать в выходное представление документации примеры исходного кода.

Сводная информация об описанных выше языках разметки текстов представлена в табл. 1.4.1.

Табл. 1.4.1. Свойства языков разметки текстов

Название	**Год появления**	**Single Source**	**WYSIWYM**	**Повторное использование**
RUNOFF, *ROFF	≥1964	Ограничено	Нет	Нет
SGML	1974	Нет	Нет	Нет
DocBook	1994	Да	Да	Ограничено
DITA	2001	Да	Да	Да
DRL	2008	Да	Да	Да
TeX	1978	Нет	Нет	Да
LaTeX	1985	Нет	Да	Да
Wiki (различные)	≥1995	Нет	Да	Ограничено
Легковесные языки общего назначения	2002	Да	Да	Нет
Javadoc	1995	Ограничено	Нет	Нет
Doxygen	1997	Да	Нет	Ограничено

Следует отметить, что современные средства разработки документации ПО (языки и программные средства) поддерживают повторное использование

крайне незначительно [112], и сама эта практика не является пока общеупотребимой. В этом направлении требуются дополнительные усилия исследователей, разработчиков языков и инструментов.

1.5. О размере документации ПО

Важным вопросом при выборе методов и средств поиска является оценка данных, на которых происходит поиск. В связи с этим было выбрано 19 документов ПО из различных проектов: 12 документов были взяты из известных открытых проектов (open source projects), 7 — из коммерческих проектов. Полный список документов представлен в табл. 1.5.1.

Табл. 1.5.1. Документация ПО, выбранная для экспериментов

№	Документ	Тип документа	Доступ
1	DocBook definitive guide	Справочник по языку	http://docbook.org/
2	Subversion Book	Рук-во по администрированию	https://subversion.apache.org/
3	Zend Framework V1 guide	Рук-во программиста	https://github.com/zendframework/zf1/tree/master/documentation
4	Linux Kernel Documentation	Рук-во программиста	https://github.com/torvalds/linux/tree/master/kernel
5	GNU Core Utils Manual	Рук-во пользователя	https://www.gnu.org/software/coreutils/coreutils.html
6	Postgre SQL Manual	Справочник по языку	https://www.postgresql.org
7	The GIMP user manual	Рук-во пользователя	https://www.gimp.org/
8	Blender reference manual	Рук-во пользователя	https://www.blender.org/
9	LibLDAP Manual	API-документация	http://www.openldap.org/
10	Eclipse SWT Reference	API-документация	http://git.eclipse.org/
11	Python Requests	API-документация	http://python-requests.org/
12	Qt Quick Reference	Рук-во программиста	http://wiki.qt.io/Developer_Guides
13	Документ 1	Рук-во пользователя	-
14	Документ 2	Рук-во пользователя	-
15	Документ 3	Спецификация требований	-
16	Документ 4	Спецификация архитектуры	-
17	Документ 5	Спецификация требований	-
18	Документ 6	Спецификация архитектуры	-
19	Документ 7	Спецификация требований	-

Говорить об объёме документации ПО сложно потому, что она разрабатывается и существует в различных форматах, состоит из многочисленных файлов. Был выбран следующий способ оценки объёма документов. Файлы, относящиеся

к одному документу, были объединены в один, и после этого при помощи утилиты Pandoc [113] получившийся файл конвертировался в «плоский» текст — неразмеченный текст в кодировке UTF-8. Таким образом, из этих итоговых документов была устранена вся разметка, картинки и прочая дополнительная информация. Размер документа определялся как размер такого файла. Средний размер одного документа из табл. 1.5.1 составил 757 Кб, максимальный размер — 2512 Кб. Для того, чтобы нагляднее представить, насколько велики эти документы, автор диссертационного исследования конвертировал те же исходные файлы утилитой Pandoc в формат Microsoft Word (шрифт Times New Roman, кегль 11, одинарный интервал, страница A4). Как можно увидеть в табл. 1.5.2, размер нескольких документов оказался близок к 1000 страницам, размер один документа превысил 1000 страниц. Автор диссертационного исследования склонен считать, что в области разработки ПО не встречается существенно бóльших документов. В связи с этим можно положить допустимый верхний предел размеров одного документа в 3 Мб «плоского» текста. В основном, предложенный в работе Duplicate Finder работает именно с «плоским» текстом. Имеются также возможности для работы с DocBook-документами, которые, как следует из табл. 1.5.2, могут быть ненамного больше и тоже укладываются в ограничение 3 Мб. Можно предположить, что остальные существующие документы будут близки по размерам к тем, которые были выбраны, так как были рассмотрены достаточно крупные проекты[11].

Далее в этом исследовании для экспериментов будет использована именно эта выборка документов.

[11] Следует отметить, что в целом документация по проекту может быть объёмнее, однако в данном исследовании не рассматривается вся документация проекта, а только отдельные документы, которые, однако, могут быть представлены набором исходных файлов.

Табл. 1.5.2. Объём выбранной документации ПО

№	Документ	Исходная документация				Формат MS Word		«Плоский» текст, Кб
		Формат	Размер, Кб	Кол-во файлов	Ср. размер файла, Кб	Размер, Кб	Кол-во страниц	
1	DocBook definitive guide	DocBook	704	22	32	182	232	332
2	Subversion Book	DocBook	1854	26	71	515	483	1294
3	Zend Framework V1 guide	DocBook	2995	186	16	686	895	1473
4	Linux Kernel Documentation	DocBook	918	33	28	269	296	599
5	GNU Core Utils Manual	TROFF	1803	169	11	1055	937	2104
6	Postgre SQL Manual	TROFF	1056	169	6	582	493	915
7	The GIMP user manual	DocBook	3160	566	6	606	739	1545
8	Blender reference manual	ReStructuredText	3198	1161	3	1225	1383	2512
9	LibLDAP Manual	TROFF	1104	183	6	501	609	1076
10	Eclipse SWT Reference	Java	4321	215	20	1168	89	1449
11	Python Requests	Python	1122	83	14	54	49	36
12	Qt Quick Reference	EPub	342	1	342	2241	110	170
13	Документ 1	Microsoft Word	7343	1	7343	7343	107	176
14	Документ 2	Microsoft Word	2505	1	2505	2505	58	100
15	Документ 3	Microsoft Word	164	1	164	164	20	97
16	Документ 4	Microsoft Word	1787	4	447	1787	105	241
17	Документ 5	Microsoft Word	1381	1	1381	1381	22	43
18	Документ 6	Microsoft Word	817	1	817	817	20	50
19	Документ 7	Microsoft Word	768	1	768	768	63	167
	Среднее							**757**

1.6. Используемые в диссертации методы, модели и технологии

В данном разделе описываются методы и технологии, использованные в данной диссертации.

Технология DocLine и рефакторинг документации

Система DocLine и язык DRL [11, 21] предназначены для планового повторного использования документации на DocBook. DocLine/DRL позволяет применить подход адаптивного повторного использования, предложенный С. Ерзабеком (S. Jarzabek) и П. Бассеттом [83], к разработке документации ПО.

Язык DRL расширяет набор тегов DocBook новыми тегами, которые позволяют определять повторно используемые фрагменты текста (информационные элементы или словари). Документация в формате DRL обрабатывается препроцессором, который раскрывает ссылки и выдаёт документ в формате DocBook. Дальнейшая обработка DocBook-документации может выполняться стандартными для DocBook средствами с генерацией PDF, HTML, RTF и т.д.

Рассмотрим пример повторного использования фрагментов текста средствами DocLine. Пусть в тексте документа имеется два схожих фрагмента, представленные на листинге 1.6.1 (здесь жирным выделены отличия этих фрагментов):

…
When module instance receives refresh_news call, it updates its data from **RSS and Atom** feeds it is **configured to listen** to and pushes new aticles to the main storage.
…
When module instance receives refresh_news call, it updates its data from **Twitter** feeds it is **subscribed** to and pushes new articles to the main storage.
…

Листинг 1.6.1. Два сходных фрагмента документации

Единое описание этих фрагментов при помощи языка DRL представлено на листинге 1.6.2.

```
<infelement id="refresh_news">When module instance receives rfresh_news
call, it updates its data from
<nest id="SourceType" /> feeds it is <nest id="SubscrType" /> to and pushes
new articles to the main storage.</infelement>
```

Листинг 1.6.2. Единая спецификация двух сходных фрагментов

В этом описании в тех местах, где оба фрагмента различаются, вставлены формальные параметры. При использовании этого описания надо заменить формальные параметры на фактические, как это показано на листинге 1.6.3.

```
<infelemref infelemid="refresh_news">
 <replace-nest nestid="SourceType">RSS and Atom</replace-nest>
 <replace-nest nestid="SubscrType">configured to listen</replace-nest>
</infelemref>

…

<infelemref infelemid="refresh_news">
 <replace-nest nestid="SourceType">Twitter</replace-nest>
 <replace-nest nestid="SubscrType">subscribed</replace-nest>
</infelemref>
```

Листинг 1.6.3. Повторное использование сходных фрагментов

Д. Кознов и К. Романовский [8] описывают процесс разработки документации семейства программных продуктов с применением *рефакторинга*[12]. Для документации семейств программных продуктов (software product lines) [13] они определяют рефакторинг как выделение общих активов из документации продукта и соответствующее изменение этой документации так, чтобы её фактиче-

[12] Рефакторинг программного обеспечения определяется М. Фаулером и К. Беком, как процесс изменения программной системы, выполняемый таким образом, что внешнее поведение системы не изменяется, но при этом улучшается её внутренняя структура [65].

ское содержание осталось неизменным. Понятие рефакторинга важно для данного диссертационного исследования, поскольку позволяет найти формальное применение найденным в документации повторам. В рамках технологии DocLine был реализован первый прототип рефакторинга на основе автоматически найденных повторах [23].

Инструмент Clone Miner

Clone Miner [35] является инструментом поиска клонов в исходном коде программного обеспечения, основанным на токенах, а не на анализе синтаксического дерева программы, подобно другим клон-детекторам. Инструмент реализован на языке C++ и использует основанный на суффиксных массивах [31] алгоритм поиска повторяющихся фрагментов текста.

Clone Miner предоставляет удобный интерфейс командной строки, данные об обнаруженных точных повторах выдаются в текстовом файле. Инструмент поддерживает анализ русскоязычных текстов. Эти свойства позволяют легко интегрировать Clone Miner с другими программными средствами и эффективно использовать его в исследовательских целях.

Утилита Pandoc

Pandoc [113] является инструментом для конвертации документов из одних форматов в другие. Утилита поддерживает 26 входных форматов и 47 выходных. Большинство поддерживаемых Pandoc форматов является легковесными языками разметки, но также он поддерживает форматы ROFF, HTML, DocBook, LaTeX, Microsoft Word, OpenOffice/LibreOffice Writer, EPub и т.д. Форматирование результатов конвертации может настраиваться с помощью пользовательских шаблонов. Кроме основного инструмента — конвертора — в программный пакет также входит генератор списков литературы, сходный с BibTeX и BibLaTeX.

Утилита Pandoc предоставляет интерфейс командной строки, позволяющий задавать большое количество опций конвертации, благодаря чему её удобно интегрировать с другим программным обеспечением.

Редакционное расстояние

При работе с неточными текстовыми повторами актуальна задача определения степени совпадения повторяющегося текста. Существуют различные определения степени близости (схожести) текстовых строк. Одним из них является *редакционное расстояние* между двумя строками, определяемое как минимальное количество операций редактирования, позволяющих получить одну строку из другой [72]. Существуют различные определения редакционного расстояния, отличающиеся набором операций редактирования. Расстояние Левенштейна [14] использует следующие операции: вставка символа, удаление символа замена одного символа на другой. Расстояние Дамерау-Левенштейна [54] в дополнение к предыдущим трём операциям использует операцию взаимного перемещения двух символов строки. Расстояние Хемминга [74] определяется для двух строк одинаковой длины с применением единственной операции — замены одного символа на другой. Расстояние по наибольшей общей подпоследовательности символов [41, 72, 97] использует операции вставки и удаления символа и, согласно [41], может быть вычислено следующей простой формулой:

$$\mathrm{d}(s_1, s_2) = |s_1| + |s_2| - 2|\mathrm{LCS}(s_1, s_2)|,$$

где $|s_j|$ — длина строки в символах, а $|\mathrm{LCS}(s_1, s_2)|$ — длина наибольшей подпоследовательности символов, встречающаяся в обеих строках.

Если применять редакционное расстояние при поиске неточных повторов сотен символов, то потребуется вычислять его десятки и даже сотни тысяч раз. Рассмотрим сложность вычисления различных видов редакционных расстояний. Сложность вычисления расстояний Левенштейна, Дамерау-Левенштейна, а также расстояния по наибольшей общей подпоследовательности символов для строк s_1и s_2 при помощи известных алгоритмов [41, 143, 144] оценивается как $\mathcal{O}(|s_1| \times |s_2|)$ в худшем случае. Для расстояний Левенштейна и по наибольшей

общей подпоследовательности символов также показано, что более производительные алгоритмы в принципе не могут быть реализованы [29, 33][13]. Сложность вычисления расстояния Хеммнига оценивается как $\mathcal{O}(|s_1|) = \mathcal{O}(|s_2|)$, однако расстояние Хемминга не позволяет работать со строками разной длины и не рассматривает операции вставки и удаления, что существенно для задач данного исследования. На основе экспериментов автора исследования с разными расстояниями оказалось, что данная сложность существенно влияет на быстродействие алгоритмов поиска неточных повторов и требует оптимизации. Следует минимизировать количество и длины строк, расстояние между которыми требуется вычислять, запоминать уже вычисленные значения расстояния, при возможности — вычислять их, опираясь на значения, уже вычисленные для других строк. В дальнейшем в данном исследовании использовано расстояния по наибольшей общей подпоследовательности символов в силу удобства при выполнении доказательств.

В [72] показано, что редакционное расстояние по наибольшей общей подпоследовательности $d(s_1, s_2)$ обладает метрическими свойствами: **(1)** $\mathrm{d}(s, s) = 0$, **(2)** $\forall s_1 \neq s_2 : d(s_1, s_2) = d(s_2, s_1) > 0$, и, наконец, **(3)** $\mathrm{d}(s_1, s_3) \leq \mathrm{d}(s_1, s_2) + \mathrm{d}(s_2, s_3)$.

Алгоритмы вычисления редакционного расстояния реализованы в различных программных библиотеках. Например, доступны библиотеки для языка Python, позволяющие вычислять расстояния Левенштейна [119] и Дамерау-Левенштейна [117]. Для C и C++ реализована библиотека вычисления расстояния по наибольшей общей подпоследовательности [134]. В данной работе для вычисления редакционного расстояния используется алгоритм нахождения наибольшей общей подпоследовательности [123], реализованный в библиотеке difflib [118]. Данная библиотека входит в стандартную поставку Python, а её части, критичные

[13] При допущении, что верна гипотеза об экспоненциальном времени, утверждающая, что задача выполнимости булевых формул не может быть решена за субэкспоненциальное время в худшем случае [79].

с точки зрения производительности, реализованы на языке C, что обеспечивает высокую скорость работы.

Интервальное дерево

Интервальное дерево (interval tree) — это структура данных, которая позволяет для некоторого интервала числовой оси быстро находить в некотором наборе интервалов те, которые с ним пересекаются. Первоначально интервальные деревья были созданы для решения задач вычислительной геометрии [42, 63, 116].

Интервальное строится дерево следующим образом. Пусть у нас имеется набор из n интервалов в натуральных числах, b_1 — минимум, e_n — максимум концов всех интервалов, а m — это середина интервала $[b_1, e_n]$. Интервалы разделяются на три группы: целиком расположенные слева от m, полностью расположенные справа от m и содержащие m. Текущий узел интервального дерева хранит последнюю группу интервалов, а также ссылки на левый и правый дочерние узлы, которые хранят интервалы слева и справа от m соответственно. Для каждого дочернего узла процедура построения повторяется. Детальное описание алгоритмов построения и редактирования интервального дерева можно найти в [42, 116]. Там же описан алгоритм поиска по интервальному дереву. Его средняя вычислительная сложность оценивается как $O(k + \log n)$, где n — это количество интервалов в наборе, k — количество интервалов, которые пересекаются с данным.

В данном исследовании используется программная реализация интервального дерева, выполненная Х.-Л. Халбертом (C.-L. Halbert) на языке Python [73].

1.7. Выводы

На основании сделанного обзора сделаем следующие выводы.

1. Проблема разработки и сопровождения документации ПО является актуальной, соответственно, востребованы новые исследование, создание новых методов и средств.

2. Задача поиска и анализа повторов в документации ПО является актуальной, о чем свидетельствует ряд исследований на протяжении последних десяти лет [76, 89, 111, 110, 112, 121, 145, 150]. В то же время данные работы не образуют явно выделенную область исследований, ограничиваясь case-studies для различных видов документов, отсутствует единая терминология, нет перекрёстных цитирований. Наконец, хоть в некоторых работах и указывается на важность поиска и анализа неточных повторов [89, 111, 112], однако реальных шагов в этом направлении до сих пор не было сделано.
3. Задача поиска неточных повторов в текстах не является новой. Данная задача рассматривается и решается в различных областях — в информационном поиске, в рамках обнаружения клонов в ПО, при поиске плагиата и т.д. Тем не менее следует отметить, что затруднительно применение имеющихся в этих областях методов «as is» к задаче описка неточных повторов в документации ПО из-за специфики вышеупомянутых областей (например, работа с большими объёмами данных в информационном поиске, анализ дерева разбора программы в алгоритмах поиска клонов), а также в силу следующих особенностей документации ПО: небольшие (до 3 Мб) документы, необходимость принимать во внимание структуру документа (включая синтаксическую корректность XML-фрагментов для последующего повторного использования), осмысленность (meaningfulness) повторов, гарантия полноты выдачи при поиске.
4. Тем не менее, необходимо использовать готовые базовые средства поиска, чтобы облегчить решение задачи поиска повторов в документации и получить возможность сосредоточиться на вопросах применения найденных повторов при разработке и сопровождения документации ПО. В качестве таких базовых средств, вслед за многими другими исследованиями повторов в документации ПО [76, 89, 110, 111, 145, 150], предлагается использовать методы поиска клонов в ПО, основанные на анализе текста (т.н. token-based подход). В частности, выбран алгоритм и соответствующий программный инструмент Clone Miner [35].

5. На данный момент существует несколько способов применения найденных повторов в документации ПО — это повторное использование [23, 76, 111, 112] и верхнеуровневый анализ качества документации [89, 150]. В то же время очевидно, что повторы могут использоваться в формировании шаблонов и рекомендаций для разработки документации, в частности, справочной документации, способствуя единообразию, унификации и целостности текстов. В данном направлении в настоящий момент отсутствуют соответствующие исследования.

Глава 2. Алгоритм компоновки неточных повторов

2.1. Модель неточных повторов

Задачей данного раздела является дать формальное определение неточного повтора. Рассмотрим документ D как последовательность символов. Любой символ документа D имеет координату, соответствующую смещению символа от начала документа, и эта координата является числом из интервала $[1, \text{length}(D)]$, где $\text{length}(D)$ — это количество символов в D.

Определение 2.1.1. Определим ***текстовый фрагмент*** документа как вхождение в документ некоторой строки символов. Таким образом, каждому текстовому фрагменту документа D соответствует целочисленный интервал $[b, e]$, где b — это координата первого символа фрагмента, e — координата последнего. Будем обозначать как $g \in D$ тот факт, что у нас имеется некоторый текстовый фрагмент g документа D.

Обозначим множество всех текстовых фрагментов документа D как D^*, множество всех целочисленных интервалов, входящих в интервал $[1, \text{length}(D)]$ как I^D, множество всех возможных символьных строк длины не больше $\text{length}(D)$ как S^D.

Введём следующие обозначения.

- $[g]: D^* \to I^D$ — функция, которая по текстовому фрагменту g выдает его интервал.
- $\text{b}(g)$: $\text{D}^* \to [1, \text{length}(\text{D})]$ — функция, которая по текстовому фрагменту g выдаёт начало его интервала.
- $\text{e}(g)$: $\text{D}^* \to [1, \text{length}(\text{D})]$ — функция, которая по текстовому фрагменту g выдаёт конец его интервала.
- $\text{str}(g): D^* \to S^D$ — функция, которая по текстовому фрагменту g выдает его текст.
- $\overline{I}: I^D \to D^*$ — функция, которая по интервалу I выдает соответствующий ему текстовый фрагмент.

- $||[b,e]||: I^D \to [0, \text{length}(D)]$ — функция, которая по интервалу $[g] = [b,e]$ выдает его длину, вычисляемую как $||[g]|| = e - b + 1$. Для сокращения записи вместо $||[g]||$ будем писать $|g|$.
- Для любых $g^1, g^2 \in D$ их пересечение $g^1 \cap g^2$ будем рассматривать как пересечение соответствующих интервалов $[g^1] \cap [g^2]$, под $g^1 \subset g^2$ будем подразумевать $[g^1] \subset [g^2]$, под $g^1 \subseteq g^2$ — $[g^1] \subseteq [g^2]$, под $g^1 \backslash g^2$— разность $[g^1] \setminus [g^2]$.
- Определим двухместный предикат Before на множестве $D^* \times D^*$, который является истинным для двух текстовых фрагментов g^1, g^2 документа D тогда и только тогда, когда $\text{e}(g^1) < \text{b}(g^2)$.

Определение 2.1.2. Пусть G является набором текстовых фрагментов документа D таких, что $\forall g^1, g^2 \in G \ \ ((\text{str}(g^1) = \text{str}(g^2)) \wedge (g^1 \cap g^2 = \emptyset))$. Такие фрагменты назовём ***точными повторами***, а G — ***группой точных повторов*** или ***точной группой***. Посредством $\#G$ будем обозначать количество элементов в G.

Определение 2.1.3. Для упорядоченного набора точных групп $G_1, \dots, G_N$ будем говорить, что этот набор образует ***вариативную группу*** $\langle G_1, \dots G_N \rangle$, если выполняются следующие условия.

1. $\#G_1 = \cdots = \#G_N$.
2. Текстовые фрагменты, имеющие в разных группах одни и те же порядковые номера, следуют в исходном тексте в одном и том же порядке:

   $\forall g_i^k \in G_i \ \forall g_j^k \in G_j \ ((i < j) \Leftrightarrow \text{Before}(g_i^k, g_j^k))$, и

   $\forall\ k\ \in \{1, \dots, N-1\}\ \text{Before}(g_N^k, g_1^{k+1})$.

При этом будем писать, что для любой группа G_k из этого набора справедливо $G_k \in VG$.

Замечание 2.1.1. Из условия 2 определения 2.1.3 следует, что $\forall g_i^k \in G_i, \forall g_j^k \in G_j \ \ (i \neq j \Rightarrow g_i^k \cap g_j^k = \emptyset)$.

Замечание 2.1.2. Если $VG = \langle G_1, \dots G_N \rangle$ и $VG' = \langle G'_1, \dots G'_M \rangle$ являются вариативными группами, то $\langle VG, VG' \rangle = \langle G_1, \dots G_N, G'_1, \dots G'_M \rangle$ также является вариативной группой в том случае, если она удовлетворяет определению 2.1.3.

Определение 2.1.4. ***Расстояние между текстовыми фрагментами.*** Для любых $g^1, g^2 \in D$ определим расстояние следующим образом:

$$\mathrm{dist}(g^1, g^2) = \begin{cases} 0, & g^1 \cap g^2 \neq \emptyset, \\ \mathrm{b}(g^2) - \mathrm{e}(g^1) + 1, & \mathrm{Before}(g^1, g^2), \\ \mathrm{b}(g^1) - \mathrm{e}(g^2) + 1, & \mathrm{Before}(g^2, g^1). \end{cases} \quad (2.1.1)$$

Определение 2.1.5. ***Расстояние между точными группами*** G_1 и G_2 при условии, что $\#G_1 = \#G_2$ определим так:

$$\mathrm{dist}(G_1, G_2) = \max_{k \in \{1,\dots,\#G_1\}} \mathrm{dist}(g_1^k, g_2^k) \quad (2.1.2)$$

Определение 2.1.6. ***Расстояние между вариативными группами*** VG_1 и VG_2, при условии, что существуют $G_1 \in VG_1, G_2 \in VG_2$ такие, что $\#G_1 = \#G_2$, определим следующим образом:

$$\mathrm{dist}(VG_1, VG_2) = \max_{G_1 \in VG_1, G_2 \in VG_2} \mathrm{dist}(G_1, G_2) \quad (2.1.3)$$

Определение 2.1.7. ***Длина точной группы*** G определяется следующим образом: $\mathrm{length}(G) = \sum_{k=1}^{\#G} \left| g^k \right|$ где $g^k \in G$.

Определение 2.1.8. ***Длина вариативной группы*** $VG = \langle G_1, \dots, G_N \rangle$ определяется следующим образом:

$$\mathrm{length}(VG) = \sum_{i=1}^{N} \mathrm{length}(G_i) \quad (2.1.4)$$

Определение 2.1.9. ***Группа неточных повторов*** — это вариативная группа $VG = \langle G_1, \dots, G_N \rangle$ такая, что $\forall k \in \{1, \dots, \#G_1\}$ справедливо следующее:

$$\sum_{i=1}^{N-1} \mathrm{dist}\left(g_i^k, g_{i+1}^k\right) \leq 0{,}15 * \sum_{i=1}^{N} |\, g_i^k|. \quad (2.1.5)$$

Данное определение является формализацией понятия неточного повтора в [37], где утверждается, что вариативная часть неточных повторов одной и той

же информации (*delta*) не должна превышать 15% их неизменной части (*архетипа, archetype*).

Замечание 2.1.3. Точная группа может быть рассмотрена как группа неточных повторов, сформированных единственной группой $\langle G \rangle$.

Определение 2.1.10. Рассмотрим группу неточных повторов $VG = \langle G_1, G_2 \rangle$, где G_1 и G_2 — точные группы. Будем считать, что VG содержит единственную ***точку расширения***, а фрагменты текста, расположенные в интервалах $[\mathrm{e}(g_1^k) + 1, \mathrm{b}(g_2^k) - 1]$, назовём ***значениями точки расширения***. В общем случае группа неточных повторов $\langle G_1, \dots, G_N \rangle$ имеет $N - 1$ точек расширения.

Данное выше определение неточных повторов хорошо подходит к ситуации, когда неточные повторы строятся из точных. Однако могут использоваться и другие подходы к поиску. Кроме того, данное определение не предполагает вариаций текста на краю текстового фрагмента (левом и/или правом) – вариация (то есть точка расширения — см. определение 2.1.10) обязательно располагается между двумя фрагментами архетипа. Наконец, в этом определении зафиксировано отношение размера вариации к размеру архетипа. Для того, чтобы снять все эти ограничения, предложено следующее определение.

Определение 2.1.11. ***Группа неточных повторов с мерой близости k.*** Пусть у нас имеется набор текстовых фрагментов $fr_1, \dots, fr_M$ документа D. Будем называть этот набор группой неточных повторов с мерой близости k (далее — группой неточных повторов), если выполнены следующие условия:

1. $\forall\, i \;\; \mathrm{Before}\,(fr_i, fr_{i+1})$
2. Существует упорядоченный набор строк $I_1, \dots, I_N$ такой, что имеется вхождение этого набора в каждый текстовый фрагмент, то есть $\forall j \in \{1, \dots, M\} \forall i \in \{1, \dots, N\} \;\; I_i \subset \mathrm{str}(fr_j) \wedge \forall\, i'\, i'' \in \{1, \dots, N\} \;\; \left(i' < i'' \Rightarrow \mathrm{Before}\left(I_{i'}^j, I_{i''}^j\right)\right)$, где I_i^j — вхождение I_i в fr_j.
3. Число k таково, что $\frac{1}{\sqrt{3}} < k \leq 1$, и для него выполнено следующее условие:

$$\forall j \in \{1, \dots, M\} \quad \frac{\sum_{i=1}^{N} |I_i|}{|fr_j|} \geq k \qquad (2.1.1)$$

Набор строк $\langle I_1, \dots, I_N \rangle$ назовём *архетипом* данной группы неточных повторов и будем обозначать A.

Замечание 2.1.1. Важно отметить, что нас не интересуют фрагменты с небольшой долей похожести (то есть когда k близко к 0) — такие повторы, в основном, оказываются случайными, то есть, попадая в одну группу, они не имеют общей семантики. Исходя из наших экспериментов, целесообразно рассматривать случай, когда $k > 1/2$. Значение $\frac{1}{\sqrt{3}}$ ненамного больше, чем ½ и удобно для последующих доказательств[14].

Замечание 2.1.2. Очевидно, что при $k > 1/2$ не может быть больше одного вхождения архетипа в каждый текстовый фрагмент.

Определение 2.1.12. Пусть у нас имеется группа неточных повторов с архетипом $\langle I_1, \dots, I_N \rangle$. Тогда будем говорить, что данная группа имеет, как минимум, $\mathrm{N} - 1$ ***точек расширения***, которые являются формальными параметрами группы и располагаются в «разрывах» архетипа. На место каждой точки расширения можно вставить вариативный текст — ***значение точки расширения***. Кроме того, группа неточных повторов может иметь одну или две дополнительные точки расширения, расположенные в начале/конце фрагмента и соседствующие с архетипом только с одной стороны (в этом заключается отличие данного определения от определения 2.1.11).

Приведём пример группы неточных повторов. На листинге 2.1.1 представлен шаблон для описания фрагмента функциональности графического редактора по созданию новой диаграммы, которому соответствует описания пяти однотипных редакторов (данный пример взят из руководства пользователя промышленной

[14] Значение 15% в определении 2.1.9 группы неточных повторов соответствует $k = 0{,}87$ и попадает в интервал $\frac{1}{\sqrt{3}} < k \leq 1$.

системы визуального моделирования). Функциональность по созданию новой диаграммы у всех редакторов одинаковая с точностью до названий диаграмм и папок. В местах различий в данном шаблоне находятся точки расширения, представленные открывающими/закрывающими треугольниками с номером точки расширения.

> В папке Объекты нужно выбрать бизнес-решение, которое предполагается моделировать. Для него правой кнопкой мыши нужно вызвать всплывающее меню, в нем выбрать пункт Детализации, как показано на рис. **◂1◂ ▸1▸.** Далее в окне Свойства в разделе Детализации необходимо нажать на кнопку Новый, как показано на рис. **◂2◂ ▸2▸.** После этого в появившемся окне Мастер детализации необходимо выбрать тип модели **◂3◂ ▸3▸** бизнес-решения, как показано на рис. **◂4◂ ▸4▸**, нажать кнопку Далее. После этого, в появившемся окне следует выбрать папку для сохранения **◂5◂ ▸5▸** бизнес-решения. Внимание! В качестве папки для сохранения нужно выбрать папку Модели, в ней папку вашего бизнес-решения, далее **◂6◂ ▸6▸**, а в ней папку **◂7◂ ▸7▸.**

Листинг 2.1.1. Описание функциональности по созданию диаграмм

На листинге 2.1.2 представлено описание процесса создания новой диаграммы определённого типа (диаграммы бизнес-решения). Это описание создано на базе шаблона из листинга 2.1.1, в точки расширения которого вставлены соответствующие значения, выделенные жирным. Таким образом, Листинг 2.1.1 задаёт группу неточных повторов с семью точками расширения, количество элементов в этой группе — 5, мера близости этой группы равна 0,83.

> В папке Объекты нужно выбрать бизнес-решение, которое предполагается моделировать. Для него правой кнопкой мыши нужно вызвать всплывающее меню, в нем выбрать пункт Детализации, как показано на рис. **7**. Далее в окне Свойства в разделе Детализации необходимо нажать на кнопку Новый, как показано на рис. **8**. После этого в появившемся окне Мастер детализации необходимо выбрать тип модели **диаграмма** бизнес-решения, как показано на рис. **9**, нажать кнопку Далее. После этого, в появившемся окне следует выбрать папку для сохранения **диаграммы бизнес-решения**. Внимание! В качестве папки для сохранения нужно выбрать папку Модели, в ней папку вашего бизнес-решения, далее **Функционально-компонентная архитектура**, а в ней папку **Архитектура бизнес-решения**.

Листинг 2.1.2. Пример описания создания диаграммы
для конкретного графического редактора

Утверждение 2.1.1. Пусть у нас имеется группа неточных повторов в смысле определения 2.1.9. Тогда она будет также группой неточных повторов в смысле определения 2.1.11.

Доказательство этого утверждения очевидно. Обратное утверждение не выполняется, поскольку, неточная группа повторов в смысле определения 2.1.11 может начинаться и или заканчиваться вариативным текстом, а в смысле определения 2.1.9 — нет, кроме того, константа k может быть слишком мала, чтобы соответствовать константе 15% из определения 2.1.9.

2.2. Описание алгоритма и доказательство его корректности

Дадим несколько дополнительных определений.

Определение 2.2.1. Рассмотрим две группы неточных повторов $G = \langle G_1, \dots, G_n \rangle$ и $G' = \langle {G'}_1, \dots, {G'}_m \rangle$. Предположим, что они могут образовать вариативную группу $\langle G_1, \dots, G_n, {G'}_1, \dots, {G'}_m \rangle$ или $\langle G_1, \dots, G_n, {G'}_1, \dots, {G'}_m \rangle$, которая также является группой неточных повторов. В этом случае будем называть G и G' ***соседними группами***.

Определение 2.2.2. ***Соседние повторы*** — это текстовые фрагменты, принадлежащие соседним группам.

Следуя замечанию 2.1.3, определение 2.2.2 применимо как к точным, так и к неточным повторам.

Далее представлен алгоритм, который ищет по множеству точных групп документа D (множество $SetG$) множество групп неточных повторов (множество $SetVG$). Алгоритм использует интервальное дерево [42, 116]. Множество $SetG$ создаётся с помощью Clone Miner. Основная идея алгоритма — находить и соединять соседние группы из $SetG$ и добавлять получающиеся группы неточных повторов в $SetVG$. Спецификация алгоритма представлена на листинге 2.2.1.

```
/* SetG — Входные данные
/* SetVG — Результат
1  SetVG ← ∅
2  Initiate()
3  repeat
4    SetNew ← ∅
5    for each G ∈ SetG ∪ SetVG
6      SetCand ← NearBy(G)
7      if SetCand ≠ ∅
8        G′ ← GetClosest(G, SetCand)
9        Remove(G, G′) /* Удаление G и G′ из SetG и SetVG
10       if Before(G, G′)
11         SetNew ← SetNew ∪ {⟨G, G′⟩}
12       else
13         SetNew ← SetNew ∪ {⟨G′, G⟩}
14       Join(SetVG, SetNew)
15 until SetNew = ∅
16 SetVG ← SetVG ∪ SetG
```

Листинг 2.2.1. Алгоритм компоновки неточных повторов

При помощи функции Initiate() строится исходное интервальное дерево для множества $SetG$ (строка 2). Основаная часть алгоритма является циклом, в котором производится выбор новых групп неточных повторов (строки 3–15). Цикл повторяется, пока на очередном шаге множество новых найденных групп неточных повторов $SetNew$ не окажется пустым (строка 15). Внутри этого цикла алгоритм перебирает все группы из $SetG \cup SetVG$, и для каждой из них функция NearBy возвращает множество близкорасположенных к ней групп $SetCand$ (строки 5, 6), которые затем используются для конструирования нтточных групп. Ниже эта функция описана детально. Также доказывается её корректность, то есть тот факт, что она действительно возвращает группы, соседние с G. Затем из $SetCand$ выбирается ближайшая к G группа G' (строка 8) и создаётся вариативная группы $\langle G, G' \rangle$ или $\langle G', G \rangle$, которая добавляется в $SetNew$ (строки 10–14). Поскольку G' и G объединяются, и, таким образом, прекращают самостоятельное

существование, они удаляются из множеств $SetG$ и $SetVG$ при помощи функции Remove (строка 9). Затем функция Join добавляет $SetNew$ к $SetVG$ (строка 14). Функции Remove и Join также выполняют некоторые вспомогательные действия, описанные ниже. В самом конце множество $SetVG$ объединяется с множеством $SetG$ и представляется в виде итога работы алгоритма. Данное объединение выполняется для того, чтобы итоговый результат содержал не только группы неточных повторов, но также и те точные группы, которые не были использованы для комбинации неточных повторов (строка 16). Опишем функции, которые используются в алгоритме.

Функция Initiate() строит интервальное дерево по точным повторам, найденным Clone Miner (детали построения интервального дерева можно найти в [42, 116]). Мы строим интервальное дерево из расширенных интервалов, которые получаются из точных следующим образом. Мы расширяем на 15% интервалы, соответствующие точным повторам: если $[b, e]$ является исходным интервалом, то расширенный интервал выглядит так: $[b - 0{,}15 * (e - b + 1), e + 0{,}15 * (e - b + 1)]$. Расширенный интервал, соответствующий точному повтору g, обозначим $\updownarrow g$. В узле интервального дерева сохраняется также ссылка на группу точных повторов, которой он принадлежит.

Функция Remove удаляет группы из множеств, а все их интервалы — из интервального дерева. Реализованный в этой функции алгоритм удаления из интервального дерева описан в [42, 116].

Функция Join, в дополнение к описанным выше действиям, добавляет в интервальное дерево интервалы для вновь созданной группы неточных повторов $G = \langle G_1, \dots, G_N \rangle$. При этом использован описанный в [42, 116] стандартный алгоритм вставки. Расширенные интервалы, добавляемые в дерево, для каждого неточного повтора $g = (g_1^k, \dots, g_N^k)$ выглядит следующим образом: $[\mathrm{b}(g_1^k) - x^k, \mathrm{e}(g_N^k) + x^k]$, где $x^k = 0{,}15 * \sum_{i=1}^{N} |g_i^k| - \sum_{i=1}^{N-1} \mathrm{dist}(g_i^k, g_{i+1}^k)$. Расширенный интервал, созданный на основе неточного повтора g, также будем обозначать $\updownarrow g$.

Функция NearBy для некотрой группы G (своего аргумента) выбирает соседние с ней группы. В связи с этим для каждого текстового фрагмента из G ищется набор интервалов, пересекающихся с интервалом этого текстового фрагмента. При этом те текстовые фрагменты, которые соответствуют этим интервалам, оказываются соседними по отношению к данному, то есть для них выполнено условие (2.1.5). Поиск выполняется при помощи алгоритма поиска по интервальному дереву [42, 116]. Сначала строится множество групп GL_1, являющихся претендентами на то, чтобы быть группами, соседними с G:

$$GL_1 = \{G' | (G' \in SetG \cup SetVG) \wedge \exists g \in G,\ g' \in G' :\ \updownarrow g \cap \updownarrow g' \neq \emptyset. \quad (2.2.2)$$

То есть множество GL_1 сосоит из тех групп, в которых хотя бы один повтор находится близко к какому-нибудь повтору из G. Затем из этих групп выбираются только те, которые могут составить вариативную группу с G; эти группы помещаются в множество GL_2:

$$GL_2 = \{G' | G' \in GL_1 \wedge (\langle G, G' \rangle \text{или } \langle G', G \rangle \text{ — вариативнаягруппа})\}. \quad (2.2.3)$$

Наконец, строится множество GL_3 (результат работы функции $NearBy$), куда помещаются только те группы из GL_2, в которых все элементы близки к соответствующим элементам G:

$$GL_3 = \{G' | G' \in GL_2 \ \wedge \forall k \in \{1, \dots, \#G\} :\ \updownarrow g^k \cap \updownarrow g'^k \neq \emptyset\} \quad (2.2.4)$$

Теорема 2.2.1. Предложенный алгоритм обнаруживает группы неточных повторов, которые соответствуют определению 2.1.9.

Доказательство. По построению функции NearBy легко показать, что для группы G — своего аргумента — она возвращает множество соседних с ней групп (см. определение 2.2.1). То есть каждая из этих групп может вместе с G сформировать неточную группу, и далее алгоритм выбирает из этого множества ближайшую к G группу и конструирует новую неточную группу. Корректность всех промежуточных множеств, а также других используемых функций напрямую следует из способа их построения. ■

2.3. Эксперименты и оптимизации

Были проведены эксперименты с документами из табл. 1.5.2. Эксперименты проводились на компьютере с процессором Intel Core i7 2600 (тактовая частота 3,4 ГГц, кэш 8 Мб) и 16 Гб ОЗУ.

На рассмотренных документах алгоритм работал в среднем 73 секунды, и лишь для одного крупного документа объёмом более 2 Мб время его работы оказалось больше 5 минут. Эти результаты оказываются приемлемыми для практического применения.

С другой стороны, было установлено, что алгоритм выдаёт очень много ложноположительных срабатываний (около 85%). При этом основная проблема алгоритма заключается в том, что он не учитывает семантику повторов — например, даже включая в повторы осмысленный текст, алгоритм может добавлять к нему посторонние текстовые фрагменты. В связи с тем, что данное ограничение является принципиальным (у автора диссертационной работы имеется скепсис относительно возможности адекватного решения данной проблемы полностью автоматическим путём), было принято решение использовать предложенный алгоритм для первичного анализа документации на предмет наличия и состава неточных повторов с тем, чтобы принять организационные решения по дальнейшему сопровождению и доработке этой документации. Но даже при таком ограничении на использование алгоритм должен удовлетворять дополнительному условию — обеспечить выдачу, обозримую для анализа человеком. «Обозримость для человека» является субъективным критерием, но выдачи для одного документа в несколько тысяч групп повторов, среди которых не более 15% являются существенными, делают алгоритм неприменимым на практике. В данном случае мы готовы жертвовать полнотой (пусть алгоритм чего-то не найдёт) с тем, чтобы результаты его работы были обозримы. После анализа результатов работы алгоритма на документах из табл. 1.5.2 были сформулированы следующие численные эвристики:

- величина выдачи должна составлять не более нескольких сотен групп повторов для описательных документов и
- не более двух тысяч для справочных документов.

Для того, чтобы выполнить это требование, были сформулированы следующие правила оптимизации.

1. Исключить из выдачи точные группы, повторы которых пересекаются с повторами неточных групп. Если мы имеем две пересекающиеся точные группы, то надо исключить группу с наименьшей суммарной длиной повторов.
2. Исключить из выдачи точные группы, включающие по два текстовых фрагмента при условии, что эти текстовые фрагменты содержат не более 6 слов.

Первое правило достаточно очевидно, рассмотрим второе правило. Оно мотивировано тем, что бóльшая часть выдачи алгоритма приходится именно на группы из двух элементов с небольшой длиной текстовых фрагментов. При этом большинство таких групп попадает в один из следующих случаев:

- группа состоит из коротких фрагментов текста, нарушающих структуру документа: конец одного предложения и начало следующего (например, «is used. Fill in the»), несколько последних ячеек таблиц и фрагмент текста, непосредственно следующего за таблицей, подпись к рисунку (или заключительную часть подписи) и фрагмент текста сразу после подписи и т.д.;
- группа включает речевые обороты следующего вида: «doesn't need to be», «you want to use the»;
- группа состоит из специальных терминов, например, «ALSA ver.0.5.x», «USB 2.0 high speed»; такие группы были бы осмысленны, если бы содержали больше элементов.

Исключение таких групп из выдачи алгоритма сделает её существенно меньше, в то время как содержательной информации в таких группах немного, и она полезна лишь при детальном анализе повторов.

Предложенные правила оптимизации были реализованы. В табл. 2.3.1 показано количество групп в выдаче алгоритма до и после оптимизации.

Табл. 2.3.1. Результаты оптимизации

№	Документ	Групп в выдаче	
		До оптимизации	После оптимизации
1	DocBook definitive guide	1500	382
2	Subversion Book	4219	1054
3	Zend Framework V1 guide	6058	1796
4	Linux Kernel Documentation	1293	393
5	GNU Core Utils Manual	4683	1725
6	Postgre SQL Manual	2438	959
7	The GIMP user manual	7800	2026
8	Blender reference manual	6587	2221
9	LibLDAP Manual	152	71
10	Eclipse SWT Reference	4539	1770
11	Python Requests	59	33
12	Qt Quick Reference	282	96
13	Документ 1	408	144
14	Документ 2	208	73
15	Документ 3	117	41
16	Документ 4	394	172
17	Документ 5	15	7
18	Документ 6	77	27
19	Документ 7	145	43
В среднем		**1471**	**686**

Реализованные оптимизации увеличили время работы алгоритма в среднем на 2,5 секунды, то есть оставили его приемлемым.

Несмотря на то, что представленный алгоритм обладает приемлемыми характеристиками, он игнорирует семантику выделяемых повторов, что оказывается важным для практических применений.

Глава 3. Методика интерактивного поиска неточных повторов

В данной главе излагается методика интерактивного поиска, которая предложена для решения вопроса о поиске семантически осмысленных неточных повторов. Основная идея методики заключается в решении данного вопроса с помощью интерактивности, то есть на основе взаимодействия с пользователем. Таким образом, при анализе документации пользователю предлагается принять решение о том, какая информация является семантически значимой: пользователь формирует образец для поиска, и специальный алгоритм находит все неточные повторы данного образца в документе. Далее пользователь редактирует полученную выдачу, выбирает новый образец и так далее.

3.1. Описание методики

Схема методики представлена на рис. 3.1.1.

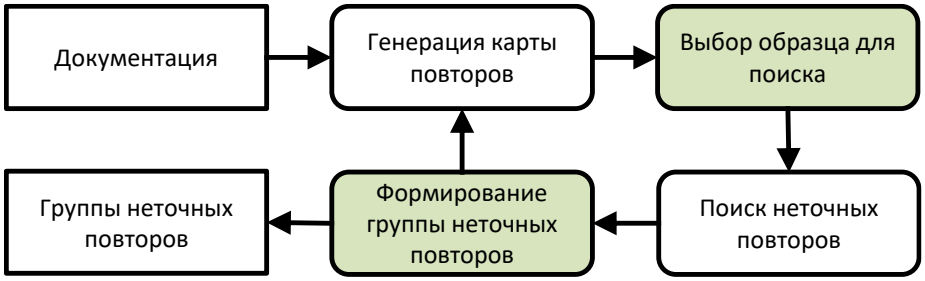

Рис. 3.1.1. Методика интерактивного поиска неточных повторов

На первом шаге («Генерация карты повторов») для выбранного документа строится карта повторов. Для этого с помощью Clone Miner в документе ищутся все точные повторы длиной не менее пяти токенов[15]. Затем для каждого токена в документе вычисляется максимальная мощность группы повторов, в которую он входит (для токена, не входящего ни в одну точную группу, элементы которой

[15] Опытным путём было установлено, что минимальное значение длины точного повтора в пять токенов как правило позволяет достичь наибольшей наглядности карты повторов.

состоят не менее чем из пяти токенов, это значение оказывается равным нулю). Далее, каждой группе повторов G_j ставится в соответствие трехкомпонентный вектор: $Color_j = \frac{\#G_j}{M} * \text{Red} + \left(1 - \frac{\#G_j}{M}\right) * \text{White}$, где $M = \max_{\forall i} \#G_i$ — максимальная мощность групп точных повторов (не менее чем с пятью токенами), $\text{Red} = [1, 0, 0]$, $\text{White} = [1, 1, 1]$. В цветовой модели RGB[16] вектора Red и White соответствуют красному и белому цвету. Известно, что с помощью выпуклой линейной комбинации для пары векторов, соответствующих паре цветов в модели RGB, строится смешанный цвет [64]. Таким образом, меняя коэффициенты выпуклой комбинации от 0 ($\#G_j = 0$) до 1 ($\#G_j = M$), получаем набор векторов, соответствующий палитре оттенков красного цвета от белого до чистого красного $[1, 0, 0]$. Соответственно, группы с большой мощностью получают более красный цвет, с меньшей — менее красный. В соответствии с этим каждому токену в документе присваивается цвет. Таким образом, весь документ оказывается «раскрашенным» в красный цвет разной насыщенности в зависимости от того, насколько часто те или иные фрагменты повторяются. Описанный способ визуализации называется *тепловой картой* (heat map) [136]. Пример тепловой карты для документа PostgreSQL Manual из табл. 1.5.1 представлен на рис. 3.1.2.

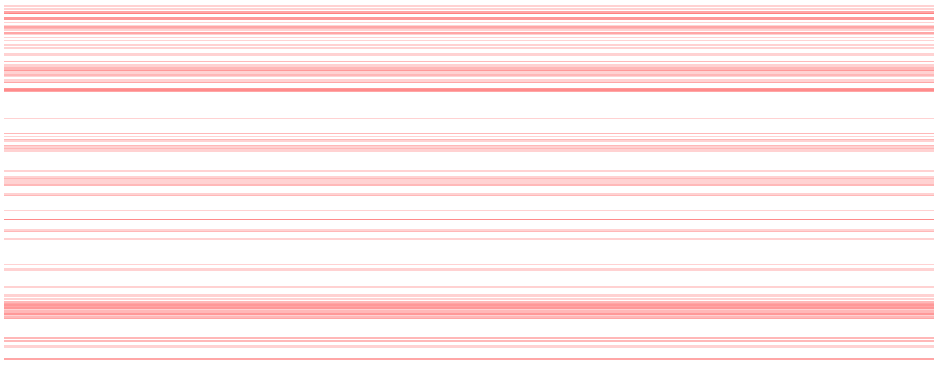

Рис. 3.1.2. Пример тепловой карты для документа PostgreSQL Manual

[16] RGB — аддитивная цветовая модель, используемая при выводе изображений на экраны современных электронных устройств [64]. Единица в соответствующей компоненте означает максимальную интенсивность красного, зелёного или синего цвета, ноль — минимальной.

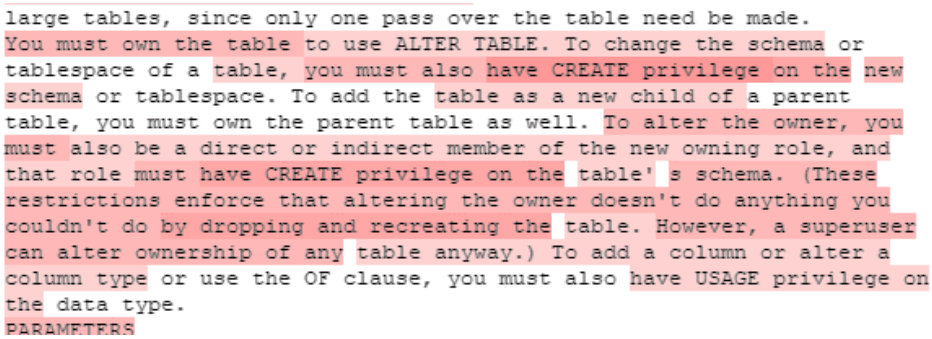

Рис. 3.1.3. Карта повторов в тексте документа

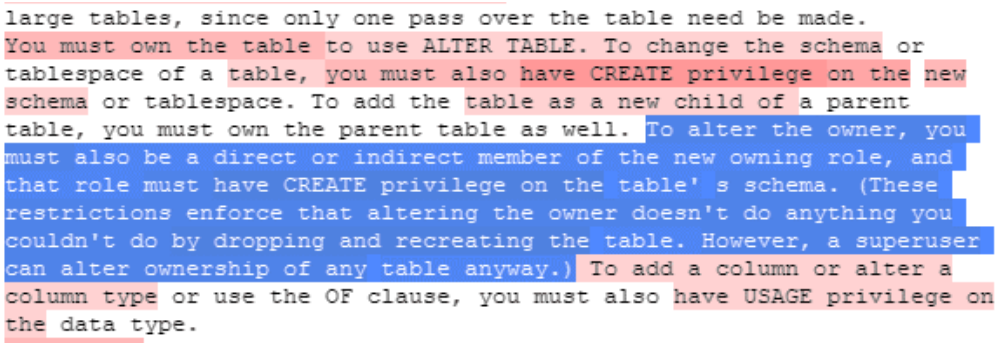

Рис. 3.1.4. Пример выбранного пользователем фрагмента для поиска

На втором шаге («Выбор образца для поиска») пользователь находит на тепловой карте (визуально, вручную) самое насыщенное повторами место (оно оказывается самым красным) и увеличивает его, получая *карту повторов*. На этой карте, в отличие от тепловой карты, уже имеется текст. Для самой нижней ярко красной области с рис. 3.1.2 получаем текст, изображённый на рис. 3.1.3. Затем пользователь выбирает на карте повторов некоторый фрагмент текста в качестве образца для поиска, как показано на рис. 3.1.4.

Следует отметить, что выбранный фрагмент обладает семантической замкнутостью, то есть полностью описывает самостоятельный факт, связанный с СУБД PostgreSQL: для того, чтобы пользователь мог сменить владельца таблицы базы данных, ему требуется обладать специфическими правами или быть администратором. Подобным образом пользователь может оформить в качестве образца для поиска целостное описание некоторой программной сущности, атрибута, свойства, шага в процессе и т.д., причём включить в образец те фрагменты такого

описания, которые имеют белый цвет на карте повторов. Эти «белые пятна» могут быть вариациями данного образца — в неточных повторах образца им может соответствовать другой текст, который был изменён при copy/paste данного фрагмента при разработке документа.

Далее пользователь запускает для выделенного фрагмента (рис. 3.1.4) алгоритм поиска по образцу (рис. 3.1.1, «Поиск неточных повторов»). Результатом работы алгоритма является набор текстовых фрагментов, похожих на выделенный образец с выбранной мерой близости (число от $\frac{1}{\sqrt{3}}$ до 1). После этого пользователь переходит к следующему шагу.

На этом шаге («Формирование группы неточных повторов») пользователь редактирует полученную выборку. Прежде всего, он удаляет из неё те элементы, которые похожи на образец только синтаксически, но не семантически. Такая ситуация часто происходит, если взято k достаточно далёким от 1. Кроме этого пользователь может корректировать границы фрагментов.

Так, например, на основании образца, выделенного на рис. 3.1.4 синим цветом, наш алгоритм нашёл неточный повтор, представленный на листинге 3.1.1. (полный список повторов данного образца представлен в Приложении):

> alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the materialized view's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the materialized view. However, a superuser can alter ownership of any view anyway.)

Листинг 3.1.1. Неточный повтор, найденный по образцу

Пользователь расширил начало этого повтора, получив текстовый фрагмент, представленный на листинге 3.1.2.

> To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the materialized view's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the materialized view. However, a superuser can alter ownership of any view anyway.)

Листинг 3.1.2. Откорректированный пользователем неточный повтор

Аналогично можно сдвинуть и правую границу фрагмента. Полные результаты поиска по образцу и результаты их коррекции пользователем приведены в Приложении. Всего в документе PostgreSQL Manual для данного образца содержатся 13 неточных повторов. Они отличаются тем, что в них описываются привилегии, необходимые для манипуляции с различными сущностями схемы базы данных: таблицами, представлениями, хранимыми функциями и т.д. Так фрагмент текста на листинге 3.1.1 отличается от образца с рис. 3.1.4 тем, что в нём токен «table» заменён на токен «view», то есть последний фрагмент описывает те же самые требования к правам, и содержит такие же оговорки, что и в предыдущем фрагменте, но уже применительно к смене владельца представления, а не таблицы.

3.2. Алгоритм поиска по образцу

В этом разделе описывается созданный алгоритм поиска по образцу, применяемый в представленной методике. Дадим в начале несколько определений.

Определение 3.2.1. ***Неточные повторы текстового фрагмента.*** Пусть у нас имеется некоторый текстовый фрагмент p документа D, то есть $p \in D$. Пусть имеется также $fr \in D$. Будем говорить, что fr является неточным повтором p с точностью k, если fr и p вместе образуют группу неточных повторов с точностью k в смысле определения 2.1.11. Будем говорить о нескольких неточных повторах фрагмента p с точностью k, если все эти текстовые фрагменты вместе с p образовывают группу неточных повторов с точностью k. Будем говорить, что группа G является группой неточных повторов некоторого фрагмента p с точностью k, если $p \in G$ и G является группой неточных повторов с точностью k.

Определение 3.2.2. Будем далее в этой работе понимать под ***редакционным расстоянием*** между двумя строками минимальное суммарное количество вставок и удалений символов, позволяющее преобразовать первую строку во вторую [41, 97]. Пусть S_1 и S_2 являются некоторыми строками символов. Будем обозначать редакционное расстояние между ними как $\mathrm{d}_{di}(S_1, S_2)$ (индекс di обозначает операции удаления и вставки — *delete* и *insert*). Редакционное расстояние для

текстовых фрагментов $fr_1, fr_2 \in D$ определим как $\mathrm{d}_{di}(fr_1, fr_2) = \mathrm{d}_{di}(\mathrm{str}(fr_1), \mathrm{str}(fr_2))$.

Нам требуется создать алгоритм, который ищет в документе D неточные повторы некоторого выделенного текстового фрагмента p (далее — образца). При этом важно, чтобы были найдены все неточные повторы p с некоторой точностью. Допустимо, чтобы алгоритм выдавал дополнительно некоторое количество ложноположительных срабатываний. Предполагается, что пользователь может легко скорректировать результат, поскольку в документации программного обеспечения группы неточных повторов относительно невелики: исходя из экспериментов, выполненных автором диссертационного исследования на реальных промышленных документах, количество элементов в семантически осмысленных группах обычно не превосходит 10 и редко превосходит 50. То есть общее количество элементов, выдаваемое пользователю для анализа, исчисляется десятками, а не сотнями и тысячами.

Спецификация алгоритма приведена на листинге 3.2.1. На вход алгоритм получает документ D, текстовый фрагмент этого документа p — образец для поиска, константу k — меру близости неточных повторов: $\frac{1}{\sqrt{3}} < k \leq 1$. Результатом алгоритма является множество текстовых фрагментов R, которое содержит неточные повторы фрагмента p в документе D.

```
/* D, p, k — Входные данные
/* R — Результат
1   W_1 ← ∅                          /* начало фазы 1 (сканирование)
2   for ∀ w_1: w_1 ∈ D ∧ |w_1| = L_w
3       if d_di(w_1, p) ≤ k_di
4           add w_1 to W_1
5   W_2 ← ∅                          /* начало фазы 2 («усушка»)
6   for w ∈ W_1
7     w'_2 ← w
8     for l ∈ I
9       for ∀w_2: w_2 ⊆ w ∧ |w_2| = l
10          if Compare(w_2, w'_2, p)
11              w'_2 ← w_2
12          add w'_2 to W_2
13  W_3 ← Unique(W_2)                /* начало фазы 3 (фильтрация)
14  for w_3 ∈ W_3
15    if ∃w'_3 ∈ W_3: w ⊂ w'_3
16        remove w_3 from W_3
17  R ← W_3
```

Листинг 3.2.1. Алгоритм поиска по образцу

Алгоритм разбит на следующие фазы. На **фазе 1 (сканирование)** текст документа D сканируется окном w длины $L_w = \frac{|p|}{k}$ с шагом в один символ[17]; текстовый фрагмент, соответствующий текущему положению окна, сравнивается с образцом p по редакционному расстоянию, и если они близки, то этот фрагмент сохраняется в множестве W_1. На **фазе 2 («усушка»)** производится поиск наибольший фрагмент текста внутри каждого элемента из множества W_1, максимально похожего на образец p. Таким образом происходит «усушка» элементов из W_1, то есть у них уменьшается длина. Это целесообразно, поскольку окно имеет максимально возможный размер, который может иметь неточный повтор фрагмента

[17] Здесь и далее не выполняется округление до целого, поскольку все рассуждения и доказательства могут быть легко повторены с учётом этого замечания, а явное обозначение округления загромождает формулы.

p (см. утв. 3.3.1), то есть все элементы множества W_1 имеют максимально возможный размер. Результатом работы этой фазы является множество W_2. На **фазе 3 (фильтрация)** из множества W_2 удаляются одинаковые элементы (они появляются на предыдущей фазе в силу того, что в W_1 могут попадать текстовые фрагменты, отличающиеся друг от друга на сдвиг окна в несколько символов), а также удаляются элементы, полностью входящие в другие элементы W_2. Итогом этой фазы является множество W_3, которое оказывается результатом действия алгоритма, то есть множеством R.

Опишем теперь функции, использованные в представленном на листинге 3.2.1 алгоритме. Функция Compare используется на фазе 2 для того, чтобы выяснить, какой из двух фрагментов текста является максимально близким к образцу p в смысле редакционного расстояния; если оба текстовых фрагмента имеют равное редакционное расстояние от образца, то они сравниваются по количеству символов (то есть нас интересуют максимально большие повторы):

$$\text{Compare}\,(w_1, w_2, p) = \begin{cases} \text{true}, & \text{если } \mathrm{d}_{di}(w_1, p) < \mathrm{d}_{di}(w_2, p) \\ \text{false}, & \text{если } \mathrm{d}_{di}(w_1, p) > \mathrm{d}_{di}(w_2, p) \\ |w_1| > |w_2|, & \text{если } \mathrm{d}_{di}(w_1, p) = \mathrm{d}_{di}(w_2, p). \end{cases} \quad (3.2.1)$$

Функция Unique получает набор текстовых фрагментов и делает так, чтобы каждый фрагмент входил в набор ровно один раз.

Опишем фазы алгоритма детально.

Фаза 1 (сканирование). При обходе документа фрагмент текста w, соответствующий текущему положению окна, добавляется в множество W_1, если для него справедливо следующее утверждение:

$$\mathrm{d}_{di}(p, w) \leq |p| \left(\frac{1}{k} + 1\right)(1 - k^2) \quad (3.2.2)$$

Введём следующее обозначение: $k_{di} = |p| \left(\frac{1}{k} + 1\right)(1 - k^2)$. Выбор длины окна и значения k_{di} будут обоснованы ниже.

Фаза 2 («усушка»). Интервал, по которому производится «усушка» для $w_k \in W_1$, определен следующим образом: $I = \left\{k|p|, \dots, \frac{|p|}{k}\right\}$. Перебор элементов происходит так. Вначале рассматриваются все фрагменты, содержащиеся в w_k и имеющие длину $k|p|$: «прижатый» к началу w_k, отстоящий от начала w_k на один символ и т.д. до тех пор, пока такие текстовые фрагменты будут содержаться в w_k. Затем увеличиваем длину рассматриваемых фрагментов на единицу и повторяем процедуру. Действуем так до тех пор, пока длина рассматриваемых фрагментов не достигнет $\frac{|p|}{k}$. Из получившегося множества выбирается тот фрагмент w'_k, для которого расстояние $\mathrm{d}_{di}\left(\mathrm{str}(w'_k), \mathrm{str}(p)\right)$ будет минимальным, а при наличии нескольких таких — обладающий наибольшей длиной (функция Compare). В итоге для каждого элемента $w_k \in W_1$ находим один элемент w'_k, которым заменяем w_k в множестве W_1. Таким образом строится множество W_2, которое и является результатом работы фазы 2.

Фаза 3 (фильтрация). На этом шаге происходит фильтрация множества W_2, Это делается следующим образом.

1. $\forall w_2^i, w_2^j \in W_2$, если $\left[w_2^i\right] = \left[w_2^j\right]$, то из W_2 удаляется один из них, то есть в W_2 остаются только уникальные элементы.
2. Из W_2 удаляются все фрагменты текста, отрезки которых находятся внутри отрезков каких-либо других фрагментов, то есть $\forall w_2^j \in W_2 \ \exists w_2^i \in W_2$, такой, что $w_2^j \subset w_2^i$.

Результатом фазы 3 является множество найденных фрагментов текста документа W_3. Результатом работы всего алгоритма является множество R, которое совпадает с множеством W_3.

3.3. Доказательство полноты алгоритма

Сформулируем теперь критерий полноты для предложенного алгоритма.

Критерий полноты для нашего алгоритма определим следующим образом. Пусть для произвольного документа D, для любого его текстового фрагмента p,

а также для R — некоторой выдачи алгоритма и для любой группы неточных повторов G фрагмента p с точностью k (см. опр. 2.1.11) истинно следующее условие:

$$\forall g \in G\ \exists w \in R\colon |g \cap w| \geq \frac{|p|}{2}\left(3k - \frac{1}{k}\right) \qquad (3.3.3)$$

Функцию $\frac{|p|}{2}\left(3k - \frac{1}{k}\right)$ будем обозначать как $O_{min}(k)$. Смысл данного критерия заключается в том, что для любого фрагмента документа D, являющегося неточным повтором образца p, множество R будет содержать текстовый фрагмент, который существенно пересекает данный неточный повтор. Таким образом, этот неточный повтор оказывается «покрыт» выдачей, и пользователь, просматривая результаты работы алгоритма, сможет, при желании выполнить редактирование границ соответствующего элемента выдачи с тем, чтобы данный повтор был включён в результирующую выдачу полностью. Степень существенности этого пересечения задаётся с помощью $O_{min}\ (k)$. Функция $O_{min}(\frac{1}{\sqrt{3}}) = 0$, при бóльших k $O_{min}(k) > 0$, так как эта функция возрастает по k (её производная равна $\frac{|p|}{2}\left(3 + \frac{1}{k^2}\right)$ и, очевидно, положительна при всех допустимых $\frac{1}{\sqrt{3}} < k \leq 1$). На практике наилучшие результаты достигаются для $k \geq 0{,}77$: при таких k $O_{min}(k) > \frac{|p|}{2}$, то есть все элементы R «цепляют» неточные повторы, минимум, на половину длины образца. Отметим, что данная оценка пессимистична: результаты экспериментов показывают бóльшее пересечение выдачи алгоритма и неточных повторов в документе.

Теперь перейдём к доказательству полноты предложенного алгоритма. Сначала докажем ряд вспомогательных утверждений.

Утверждение 3.3.1. Пусть G — группа неточных повторов фрагмента p с точностью k. Тогда для $\forall\ g_1, g_2 \in G$ справедливо $k \leq \frac{|g_1|}{|g_2|} \leq \frac{1}{k}$.

Доказательство. Пусть A — архетип группы G, тогда в силу (2.1.1) имеем:

$$k|g_1| \leq |A|$$
$$k|g_2| \leq |A|.$$

Так как $A \subset \text{str}\,(g)_1$ и $A \subset \text{str}\,(g_2)$, имеем:

$$k|g_1| \leq |A| \leq |g_1|$$
$$k|g_2| \leq |A| \leq |g_2|.$$

Следовательно, справедливо следующее:

$$k|\mathrm{g}_1| \leq |g_2| \tag{3.3.4}$$

$$k|\mathrm{g}_2| \leq |g_1|. \tag{3.3.5}$$

Разделив (3.3.4) на $|g_2|$, а (3.3.5) — на $k|g_2|$, получаем:

$$k\,\frac{|g_1|}{|g_2|} \leq 1 \quad \Rightarrow \quad \frac{|g_1|}{|g_2|} \leq \frac{1}{k} \quad \Rightarrow \quad k \leq \frac{|g_1|}{|g_2|} \leq \frac{1}{k}$$
$$1 \leq \frac{|g_1|}{k|g_2|} \qquad\qquad k \leq \frac{|g_1|}{|g_2|}$$

■

Утверждение 3.3.2. Пусть G — группа неточных повторов фрагмента p с точностью k. Тогда $\forall\ g \in G$ справедливо следующее: $\mathrm{d}_{di}(g,p) \leq (1-k^2)|p|$.

Доказательство. Поскольку p и g принадлежат одной группе неточных повторов, то они имеют один и тот же архетип и могут быть представлены следующим образом:

$$p = v_0^p I_1 v_1^p I_2 \ldots v_{N-1}^p I_N v_N^p,$$

$$g = v_0^g I_1 v_1^g I_2 \ldots v_{N-1}^g I_N v_N^g,$$

где $I_1, I_2 \ldots, I_N$ является архетипом группы G, $v_0^p, v_1^p, \ldots, v_N^p$ — вариативной частью p, $v_0^g, v_1^g, \ldots, v_N^g$ — вариативной частью g. Введем следующие обозначения: $v^p = v_0^p v_1^p \ldots v_N^p$, $v^g = v_0^g v_1^g \ldots v_N^g$, $I = I_1 I_2 \ldots I_N$. Тогда в силу (2.1.1) имеем:

$$\frac{|\mathrm{I}|}{|\mathrm{p}|} \geq k \ \Rightarrow\ |p| - |v^p| \geq |p|k \ \Rightarrow\ |p| - |p|k \geq |v^p| \ \Rightarrow\ |p|(1-k) \geq v^p.$$

Аналогично: $|g|(1-k) \geq v^g$. При этом g из p можно получить заменой v_i^p на v_i^g, то есть $\mathrm{d}_{di}(g,p) \leq |v^g| + |v^p| \leq (1-k)(|p|+|g|)$. В силу утверждения 3.3.1 имеем $|g| \leq k|p|$. Тогда $\mathrm{d}_{di}(g,p) \leq (1-k)(1+k)|p| = (1-k^2)|p|$. ■

Утверждение 3.3.3. Пусть G — группа неточных повторов фрагмента p с точностью k, множество W_1 является результатом первой фазы алгоритма, а $fr \in D$ — произвольный текстовый фрагмент. Тогда критерий полноты выполнен для множества W_1, если его взять в качестве множества R.

Доказательство. Для d_{di} справедливо неравенство треугольника: $\mathrm{d}_{di}(fr,p) \leq \mathrm{d}_{di}(fr,g) + \mathrm{d}_{di}(g,\ p)$. Согласно утверждению 3.3.2, $\mathrm{d}_{di}(g,p) \leq |p|(1-k^2)$. Также известно, что $g \subseteq fr$. Следовательно, поскольку g можно получить из fr отбрасывая всех символов, которые принадлежат $fr \backslash g$, справедливо следующее утверждение: $\mathrm{d}_{di}(fr,g) \leq |fr| - |g|$. Но так как $|fr| = \frac{|p|}{k}$ и, согласно утверждению 3.3.1, $|g| \geq k|p|$, то справедливо следующее: $|fr| - |g| \leq \frac{1}{k}|p| - k|p|$. Следовательно, $\mathrm{d}_{di}(fr,p) \leq |p|\left(\frac{1}{k} - k + 1 - k^2\right) = |p|\left(1+\frac{1}{k}\right)(1-k^2)$. Очевидно, что при сканировании документа на фазе 1 найдётся такое положение окна, что fr попадает в окно. Тогда в силу (3.2.2) $fr \in W_1$. Таким образом, справедливо следующее:

$$\forall g \in G\colon \left(|fr| = \frac{|p|}{k}, g \subseteq fr\right) \Rightarrow fr \in W_1.$$

Поскольку для любого неточного повтора найдётся элемент из W_1, который не просто пересекает этот повтор, а целиком его содержит, то критерий 1 выполнен для множества W_1, если его взять в качестве множества R. ■

Утверждение 3.3.4. Пусть G — группа неточных повторов фрагмента p с точностью k, а множество W_2 — это результат второй фазы алгоритма. Тогда справедливо следующее утверждение:

$$\forall g \in G\ \exists w_2^g \in W_2\colon \left| w_2^g \cap g \right| \geq O_{min}.$$

Доказательство. На фазе 2 алгоритма каждый элемент $w_1 \in W_1$ заменяется на новый. Таким образом, общее количество элементов в множествах W_1 и W_2 одно

и то же. Согласно утверждению 3.3.3, для каждого $g \in G$ можно найти подходящий $w_1^g \in W_1$ такой, что $g \subseteq w_1^g$. Докажем, что получающийся в результате фазы 2 текстовый фрагмент w_2^g, соответствующий w_1^g, удовлетворяет формулировке утверждения.

По построению множества W_1 имеем: $\left|w_1^g\right| = \frac{1}{k}|p|$. В соответствии с утверждением 3.3.1 $k|p| \le |g| \le \frac{1}{k}|p|$. Поскольку по утверждению 3.3.3 в W_1 попадают все фрагменты длины $\frac{1}{k}|p|$, содержащие g, можно выбрать фрагмент w_1^g так, чтобы интервал $[g]$ был расположен по центру интервала $\left[w_1^g\right]$. Далее, в рамках этой ситуации рассмотрим случай, когда по результатам фазы 2 в качестве w_2^j был выбран фрагмент текста, «прижатый» к правому краю w_1^g: $\mathrm{e}\left(w_2^j\right) = \mathrm{e}\left(w_1^g\right)$.

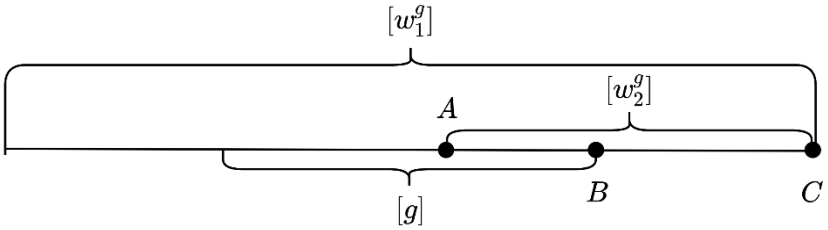

Рис. 3.3.1. Расположение фрагментов текста w_1^g, w_2^g, g,
когда w_2^g «прижат к правому краю» w_1^g.

Обозначим $\mathrm{b}\left(w_2^g\right)$ и $\mathrm{e}\left(w_2^g\right)$ как A и C соответственно. Согласно описанию фазы 2 имеем: $|AC| = \left|w_2^g\right| \ge k|p|$. Точка C также совпадает с $\mathrm{e}\left(w_1^g\right)$. Точку $\mathrm{e}(g)$ обозначим как B. Тогда между границами g и w_1^g остаётся $|BC| = \frac{1}{2}\left(\left|w_1^g\right| - |g|\right)$. Покажем теперь, что $B \in AC$. Действительно, $|BC| = \frac{1}{2}\left(\left|w_1^g\right| - |g|\right) \le \frac{1}{2}\left(\frac{|p|}{k} - k|p|\right) = \frac{1}{2}|p|\left(\frac{1}{k} - k\right)$. Это неравенство справедливо согласно описанию фазы 1, в силу которого $\left|w_1^g\right| = \frac{|p|}{k}$, а также утверждения 3.3.1 ($|g| \ge k|p|$). Поскольку также справедливо, что $|AC| \ge k|p|$, то справедливо и следующее:

$$|AC| - |BC| \ge |p|\left(k - \frac{1}{2}\left(\frac{1}{k} - k\right)\right) = \frac{|p|}{2}\left(3k - \frac{1}{k}\right) \qquad (3.3.6)$$

Легко показать, что поскольку $\frac{1}{\sqrt{3}} < k \leq 1$, то правая часть (3.3.6) положительна и, следовательно, $|AC| - |BC| > 0$. Таким образом, точка B лежит внутри отрезка AC. Теперь можно оценить $|AB|$ следующим образом:

$$|AB| = \left| w_2^g \cap g \right| = |AC| - |BC| \geq \frac{|p|}{2}\left(3k - \frac{1}{k}\right) = O_{min}.$$

Аналогично рассматривается ситуация, когда $\mathrm{b}\left(w_2^j\right) = \mathrm{b}\left(w_1^g\right)$, то есть когда w_2^j «прижат» к левому краю окна w_1^g.

Очевидно, что когда w_2^g «не прижат» к краям окна, оценки $|w_2^g|$ остаются прежними, но поскольку w_2^g смещается ближе к центру w_1^g, то $\left| w_2^g \cap g \right|$ может разве лишь увеличиться, и в этом случае также оказывается справедливым утверждение $\left| w_2^g \cap g \right| \geq O_{min}$. ■

Утверждение 3.3.5. Пусть G — группа неточных повторов фрагмента p с точностью k и дано множество W_3 — результат работы третьей фазы алгоритма. Тогда справедливо следующее утверждение:

$$\forall g \in G \; \exists w_3^g \in W_3 : \left| g \cap w_3^g \right| \geq O_{min}.$$

Таким образом, утверждается, что результаты третьей фазы алгоритма дают оценку для пересечения с неточными повторами в тексте не хуже, чем результаты второй фазы алгоритма.

Доказательство. Первый шаг третьей фазы алгоритма очевидно не ухудшает эту оценку, т.к. он лишь устраняет дубликаты.

Второй шаг третьей фазы алгоритма, встретив $w_2', w_2'' \in W_2 : w_2' \subset w_2''$, оставляет из них лишь w_2''. Но если $w_2' \subset w_2'$, то очевидно, что $\forall g \in G \; |g \cap w_2''| < |g \cap w_2''|$. Следовательно, второй шаг третьей фазы, убрав из результата w_2', также не ухудшит оценку. ■

Теперь докажем полноту предложенного алгоритма.

Теорема 3.3.1. Для любого текстового фрагмента $p \in D$, для любого k: $\frac{1}{\sqrt{3}} < k \leq 1$, для любой соответствующей выдачи алгоритма R и для любой G группы неточных повторов фрагмента p с точностью k выполняется критерий полноты.

Доказательство. Фаза 1 алгоритма формирует набор фрагментов W_1, в отношении которого в силу утверждения 3.3.3 следует, что оно удовлетворяет критерию полноты. Фаза 2 поэлементно оптимизирует W_1, строя на основе его элементов W_2. При этом утверждение 3.3.4 доказывает, что для каждого $g \in G$ среди элементов W_2 найдутся те, которые удовлетворяют критерию полноты. Фаза 3 отбрасывает избыточные и неоптимальные элементы W_2. При этом утверждение 3.3.5 доказывает, что оставшихся в W_3 элементов также достаточно, чтобы удовлетворить критерию полноты. Поскольку $R = W_3$, то результат работы всего алгоритма удовлетворяет критерию. ■

3.4. Эксперименты и оптимизации

Для предложенного алгоритма был проведён ряд экспериментов на документах ПО из табл. 1.5.1. Исследовались время работы алгоритма и величина выдачи. В последнем случае интересовало количество ложных срабатываний.

Эксперименты проводились на компьютере с центральным процессором Intel Core i7 2600 (тактовая частота 3,4 ГГц, кэш 8 Мб) и 16 Гб ОЗУ. Для определения времени работы эксперименты проводились для образцов длиной от 20 до 100 с шагом в 10 символов. Для сопоставления, фрагменты из Приложения (колонка «Найдено») имеют длину в среднем 338 символов, а длина максимального фрагмента составила 348 символов. Эксперименты проводились при значении меры близости $k = 0{,}87$ (по утв. 2.1.1 соответствует 15% в определении 2.1.9). В качестве образца для поиска в каждом документе выбирались фрагменты текста, повторяющиеся наибольшее количество раз. Выбор подходящего фрагмента производился каждый раз «вручную» с использованием тепловой карты и алгоритма компоновки. Здесь автор исследования исходил из того соображения, что чем большее число раз образец встречается в документе среди образцов той же длины, тем больше времени будет работать алгоритм. Тот факт, что в данном эксперименте изменяемым параметром является длина образца, а мера близости зафиксирована, объясняется тем, что, исходя из предварительных эксперимен-

тов, время работы алгоритма сильно варьируется в зависимости от размера образца и этот размер может значительно меняться, в то время как k меняется в незначительном диапазоне и поэтому не оказывает значительного влияния на производительность.

В результате экспериментов было установлено следующее. Как показано на рис. 3.4.1, при изменении длины образца от 20 до 100 быстро растёт и достигает в среднем 7 минут. При этом для самого большого документа (Blender, 2,5 Мб) время работы алгоритма при $|p| = 100$ составило более получаса. С учётом того, что поиск по образцу для одного документа может выполняться десятки раз, и этот поиск происходит в интерактивном режиме, такая скорость работы алгоритма является неприемлемой на практике. Соответственно, эксперименты с большей длиной образца давали результат в 10 минут и более, и в тексте диссертационной работы они не представлены в детальном виде.

Для оценки длины выдачи использовался описанный выше эксперимент. Анализ документации показал, что на практике большинство осмысленных групп неточных повторов образовано двумя текстовыми фрагментами, реже — десятью, ещё реже — несколькими десятками, в отдельных случаях — несколькими сотнями. Тем не менее, для различных документов и искомых образцов алгоритм часто выдавал в 5–10 раз больше повторов, чем фактически содержалось в документе. Для документов из табл. 1.5.2 максимальный размер выдачи в рамках проводимых экспериментов составил 13056 элементов. Это была самая большая выдача на рассмотренных в ходе эксперимента примерах. При этом ложноположительных срабатываний оказалось 12044. Большинство ложноположительных срабатываний, выдаваемых данным алгоритмом, являются фрагментами текста, смещёнными относительно друг друга на один или несколько символов. Это связано с тем, что во время первой фазы алгоритма для каждого вхождения образца фрагменты текста начинают попадать в W_1 не позже, чем двигающееся по тексту окно полностью включит в себя это вхождение, и продолжают попадать в W_1 по крайней мере до тех пор, пока окно не сдвигается с него (см. доказательство утв. 3.3.3). При этом окно существенно больше любого вхождения (то есть оно

является максимально большим текстовым фрагментом, который может попасть с образцов в одну группу неточных повторов). Итак, выдача алгоритма является формально верной (соответствует критерию полноты), но её размер не позволяет пользоваться алгоритмом на практике.

В результате проведённых экспериментов были сформулированы следующие дополнительные требования к алгоритму.

1. Время работы алгоритма не должно превышать нескольких минут для документов, размер которых не превышает 3 Мб.
2. Алгоритм должен обеспечивать обозримую выдачу за счёт существенного уменьшения ложноположительных срабатываний.

Для того, чтобы удовлетворить этим требованиям, были предложены и реализованы следующие оптимизации.

Оптимизация 1 применяется на фазе 1 (сканирование) и позволяет минимизировать количество вычислений d_{di}, тем самым существенно понижая время работы алгоритма. Идея оптимизации взята из известного алгоритма поиска точных вхождений образца в строку, предложенного Р. Бойером (R. Boyer) и Дж. Муром (J. Moore) [45]. Как и в алгоритме Бойера-Мура, мы, убедившись, что данное положение окна не подходит для выдачи, используем результаты проверки для того чтобы определить, на сколько символов можно переместить окно, не рискуя пропустить значимый результат. То есть, если во время сканирования для положения окна w справедливо $\mathrm{d}_{di}(w,p) > k_{di} + 1$, то будем сдвигать окно на $(\mathrm{d}_{di}(w,p) - k_{di})/2$ символов вправо, в противном случае сдвигаем на один символ.

Оптимизация 2 применяется на фазе 2 («усушка») и также позволяет минимизировать количество вычислений d_{di}. Подход аналогичен используемому в предыдущей оптимизации. Во время «усушки» текстового фрагмента w_1 при каждом положении окна w_2' обновляем (при необходимости) наименьшее значение $\mathrm{d}_{di}(p, w_2')$ (обозначим его d_{min}). Если для данного положения окна w_2' справедливо $\mathrm{d}_{di}(p, w_2') > d_{min} + 1$, сдвинем далее окно не на один, а на

$(\mathrm{d}_{di}(p, w_2') - d_{min})/2$ символов вправо, в противном случае сдвинем окно на один символ вправо. Значение d_{min} стирается в начале нового цикла со следующим фиксированным значением длины сканирующего окна.

Оптимизация 3 применяется на фазе 3 (фильтрация) и позволяет уменьшить мощность W_3. Делается следующее. Множество W_3 разбивается на максимальные, транзитивно замкнутые по пересечению, подмножества. Для каждого такого подмножества выбирается фрагмент w_3 с наименьшим значением $\mathrm{d}_{di}(w_3, p)$, а в случае наличия нескольких таких — фрагмент с наибольшей длиной, то есть наилучший из перекрывающихся определяется с использованием определённой выше функции Compare (3.2.1). Остальные фрагменты из множества W_3 удаляются.

Оптимизация 4 применяется на фазе 3 и производит расширение всех текстовых фрагментов W_3 до целых слов. Другими словами, начало и конец текстового фрагмента могут нарушать границу слова, и в этом случае границы текстового фрагмента соответственно «раздвигаются» с тем, чтобы включить в себя эти слова целиком.

После реализации оптимизаций алгоритм был проверен на тех же тестах, что и до оптимизации. Результаты представлены на рис. 3.4.1 и 3.4.2.

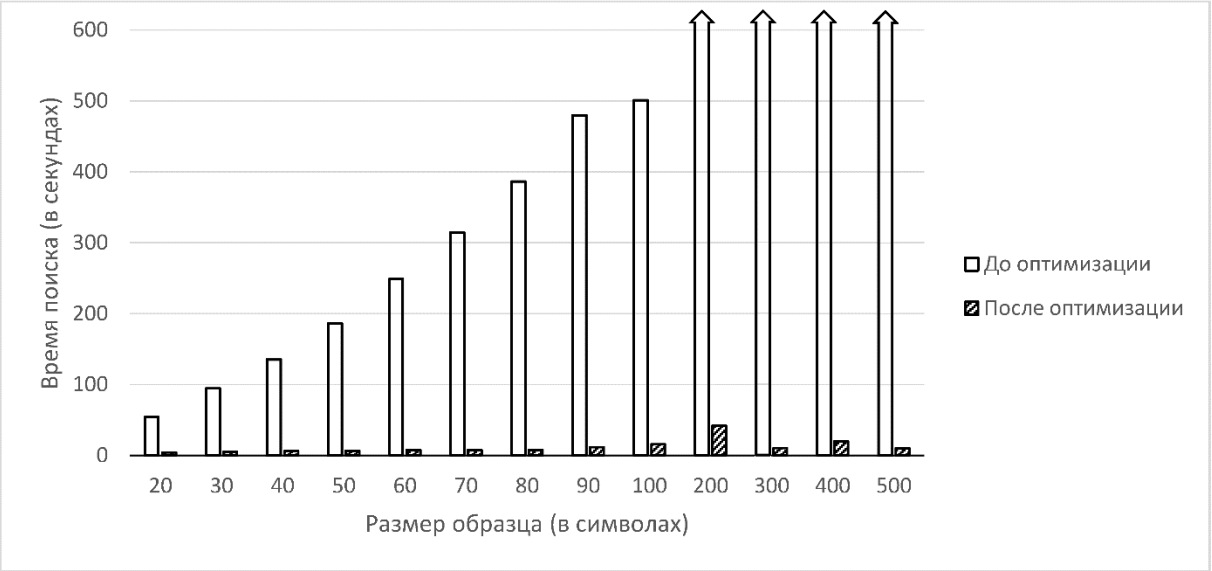

Рис. 3.4.1. Результаты оптимизации алгоритма поиска по образцу: время поиска в среднем по документам

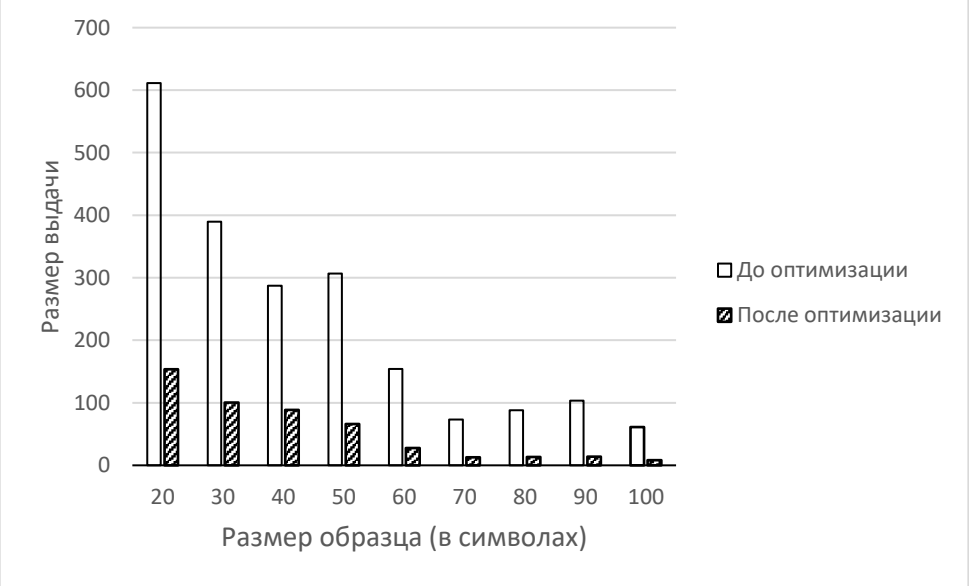

Рис. 3.4.2. Результаты оптимизации алгоритма поиска по образцу: размер выдачи в среднем по документам

Оптимизация сократила время поиска в среднем до 14 секунд, то есть в 31,75 раз. Максимальное время поиска составило две минуты[18]. Отметим, что при по-

[18] Дальнейшее ускорение программной реализации алгоритма возможно при помощи распараллеливания фаз 1 и 2.

иске по образцам размером более 300 символов время работы алгоритма не увеличивалось, т.к. настолько длинные фрагменты текста не повторялись многократно. Средний размер выдачи до оптимизации составлял 231, а после оптимизации стал равен 44, то есть сократился в 5,25 раз. Для образца, по которому алгоритм без оптимизации выдавал 13056 повторов (это пример обсуждался выше), после оптимизации алгоритм выдаёт 1012 повторов. В ходе экспериментов были получены следующие результаты (ниже приводятся данные анализа совокупной выдачи по всем тестам для алгоритма с оптимизациями).

1. У 60,8% выдачи присутствует по 1–2 «лишних» токенов в начале или в конце. У 0,1% выдачи в случае отличий повторов от образца близко к началу или концу повторов, начало и конец были потеряны. Эти проблемы легко решаются пользователем при редактировании результатов работы алгоритма.
2. При поиске по коротким образцам (3–4 токена) в отдельных случаях найденные повторы встречались в контексте, отличающемся от контекста исходного образца. Такие фрагменты можно назвать сематическими ложными срабатываниями, но наша методика не претендует на автоматический отсев таких случаев: осмысленность повторов достигается точностью выбора образца, а также возможностью пользователя удалить неосмысленные фрагменты из выдачи «вручную».
3. Ни для одного фактически присутствующего повтора не выдавалось по нескольку вариантов, все результаты перекрывались с повторами более (в среднем существенно более), чем на $3/4$ длины образца (при $O_{min}(0{,}87) \approx 0{,}73|p|$, то есть результаты оказались лучше теоретической оценки).

Таким образом, после применения оптимизации алгоритм удовлетворил требованиям, сформулированным выше, как по времени поиска, так и по размеру/составу выдачи. Итак, можно сделать вывод о готовности алгоритма и методики для практического использования.

Покажем теперь, что представленные выше оптимизации сохраняют полноту. Вначале докажем два свойства редакционного расстояния d_{di}.

Утверждение 3.4.1. Если s_1, s_2, a, b — строки, причём $|a| = |b| = 1$ (строки из одного символа), то $|\mathrm{d}_{di}(s_1, a \circ s_2) - \mathrm{d}_{di}(s_1, s_2 \circ b)| \leq 2$.[19]

Таким образом, применительно к алгоритму, если сдвинуть окно w на один символ, то для нового положения окна w' изменение редакционного расстояния не превысит 2: $|\mathrm{d}_{di}(p, w) - \mathrm{d}_{di}(p, w')| \leq 2$.

Доказательство. Для произвольной строки s очевидно, что $\mathrm{d}_{di}(s, a \circ s) = \mathrm{d}_{di}(s, s \circ b) = 1$, т.к. добавление одного символа в начало или в конец одной из строк соответствует всего одной операции редактирования.

Покажем теперь, что для любых строк s_1 и s_2 справедливо $|\mathrm{d}_{di}(s_1, a \circ s_2) - \mathrm{d}_{di}(s_1, s_2)| \leq 1$. Поскольку расстояние d_{di} обладает метрическими свойствами, для него выполняется неравенство треугольника, то есть $\mathrm{d}_{di}(s_1, a \circ s_2) \leq \mathrm{d}_{di}(s_1, s_2) + \mathrm{d}_{di}(s_2, a \circ s_2) = \mathrm{d}_{di}(s_1, s_2) + 1$. Следовательно, $\mathrm{d}_{di}(s_1, a \circ s_2) - \mathrm{d}_{di}(s_1, s_2) \leq 1$. Также неравенство треугольника для d_{di} позволяет утверждать $\mathrm{d}_{di}(s_1, a \circ s_2) - \mathrm{d}_{di}(s_1, s_2) \leq \big(\mathrm{d}_{di}(s_1, s_2) + \mathrm{d}_{di}(s_2, a \circ s_2)\big) - \mathrm{d}_{di}(s_1, s_2) = \mathrm{d}_{di}(s_2, a \circ s_2) = 1$, то есть $|\mathrm{d}_{di}(s_1, a \circ s_2) - \mathrm{d}_{di}(s_1, s_2)| \leq 1$. Аналогично $|\mathrm{d}_{di}(s_1, s_2) - \mathrm{d}_{di}(s_1, s_2 \circ b)| \leq 1$.

Поскольку редакционное расстояние — целое неотрицательное число, то для разности его значений также выполняется неравенство треугольника, следовательно: $|\mathrm{d}_{di}(s_1, a \circ s_2) - \mathrm{d}_{di}(s_1, s_2 \circ b)| \leq |\mathrm{d}_{di}(s_1, a \circ s_2) - \mathrm{d}_{di}(s_1, s_2)| + |\mathrm{d}_{di}(s_1, s_2) - \mathrm{d}_{di}(s_1, s_2 \circ b)| \leq 1 + 1 = 2$. ■

Утверждение 3.4.2. Пусть имеется два текстовых фрагмента $w_1, w_2 \in D$. Если при этом $|\mathrm{b}(w_1) - \mathrm{b}(w_2)| < \delta$ и $|w_1| = |w_2|$, то для любой строки s справедливо: $|\mathrm{d}_{di}(s, w_1) - \mathrm{d}_{di}(s, w_2)| < 2\delta$.

Доказательство. Пусть $\mathrm{b}(w_1) < \mathrm{b}(w_2)$. Будем сдвигать w_1 вправо посимвольно, пока он не совпадёт с w_2. Пусть у нас имеется два соседних положения w_1: w^j и w^{j+1}, то есть $\mathrm{b}(w^j) + 1 = \mathrm{b}(w^{j+1})$. Тогда очевидно, что существуют строки a и b: $|a| = |b| = 1$ и $\mathrm{str}(w^j) \circ a = b \circ \mathrm{str}(w^{j+1})$. Тогда, согласно

[19] Здесь и далее при помощи $s_1 \circ s_2$ будем обозначать результат конкатенации строк s_1 и s_2.

утв. 3.4.1, $\left|\mathrm{d}_{di}(s, w^j) - \mathrm{d}_{di}(s, w^{j+1})\right| \leq 2$. В итоге мы можем утверждать, что по неравенству треугольника для разности целых чисел справедливо $|\mathrm{d}_{di}(s, w_1) - \mathrm{d}_{di}(s, w_2)| \leq \sum_{j=1}^{\mathrm{b}(w_2)-\mathrm{b}(w_1)}\left|\mathrm{d}_{di}(s, w^j) - \mathrm{d}_{di}(s, w^{j+1})\right| < 2\delta$, т.к. $|\mathrm{b}(w_1) - \mathrm{b}(w_2)| < \delta$.

Аналогично утверждение доказывается для случая $\mathrm{b}(w_1) > \mathrm{b}(w_2)$.

В случае $\mathrm{b}(w_1) = \mathrm{b}(w_2)$ фрагменты совпадают, и справедливость утверждения очевидна. ■

Теорема 3.4.1. В результате применения оптимизаций 1, 2, 4 сохраняется свойство полноты алгоритма.

Доказательство. Рассмотрим оптимизацию 1. На первой фазе (сканирование) мы имеем положение окна w, для которого в случае $\mathrm{d}_{di}(w, p) > k_{di} + 1$ следующее положение сканирующего окна вычисляется как смещение вправо на $\delta = (\mathrm{d}_{di}(w, p) - k_{di})/2$ символов.

Покажем, что для $\forall w' : |w'| = |w| \wedge \mathrm{b}(w) < \mathrm{b}(w') < \mathrm{b}(w) + \delta$ справедливо $\mathrm{d}_{di}(w', p) > k_{di}$, то есть по описанию фазы 1 w' нельзя поместить в W_1. Из этого утверждения будет следовать, что при скачке на δ символов вправо мы не потеряем ни одного элемента W_1.

Действительно, $\mathrm{b}(w) < \mathrm{b}(w') < \mathrm{b}(w) + \delta \Rightarrow |\mathrm{b}(w) - \mathrm{b}(w')| < \delta$. Тогда, согласно утверждению 3.4.2, справедливо $\mathrm{d}_{di}(p, w) - \mathrm{d}_{di}(p, w') < 2\delta$. Но поскольку $\delta = (\mathrm{d}_{di}(w, p) - k_{di})/2$, то выполняется $\mathrm{d}_{di}(p, w) - \mathrm{d}_{di}(p, w') < \mathrm{d}_{di}(p, w) - k_{di}$. Следовательно, $\mathrm{d}_{di}(w', p) > k_{di}$, т.е. по построению фазы 1 фрагмент w' не может быть элементом W_1.

Доказательство полноты для оптимизации 2 аналогично доказательству для оптимизации 1.

Рассмотрим оптимизацию 4. Ее использование лишь расширяет фрагменты в финальной выдаче, и это действие не может повлиять на полноту результата. ■

Замечание 3.4.1. Возможны ситуации, когда оптимизация 3 нарушает критерий полноты. Например, если текст документа — «abc abc abc abc», образец —

«abc abc» то при любом k в результате оптимизации 3 алгоритм выдаст единственное вхождение образца в текст, в то время как образец входит в текст документа дважды (согласно определению 2.1.11 неточные повторы не пересекаются). Тем не менее, результаты апробации показывают, что потеря значимых элементов в выдачах алгоритма на практике происходит только в случаях наличия близко расположенных повторов, при этом даже в этих случаях это происходит очень редко.

Глава 4. Метод улучшения документации на основе неточных повторов

В этой главе описывается метод улучшения документации на основе неточных повторов. Данный метод позволяет работать с документацией любого характера, но наибольшего эффекта позволяет достичь при работе со справочной документацией (API-документами, руководствами пользователя и т.д.). Документы этого вида помогают уже знакомому с предметом читателю установить конкретные факты и не предназначены для чтения целиком [114]. Описывая наборы однотипных сущностей, таких как функции, классы, элементы графического интерфейса и т.д., справочная документация должна быть как можно более однородной. Вследствие этого она неизбежно включает в себя много повторов. На практике технический писатель обычно имеет дело с документами, которые требуют улучшений в смысле однородности. Для этого ему(ей) приходится выявлять неточные повторы, анализировать их и вносить в документацию правки, повышая её ясность и чёткость, целостность и однозначность.

4.1. Описание метода

Поскольку само ПО часто меняется (появляется новая функциональность, изменяется интерфейс пользователя, исправляются ошибки и т.д.), то его документация также требует соответствующих обновлений. В то же время, документация требует изменений, не связанных с изменениями программного обеспечения: необходимо исправлять ошибки, реструктурировать документацию и приводить её к единообразному виду, изменять её в связи с внедрением новых инструментов и технологий работы с документацией и т.д. В качестве отрицательного примера можно привести документацию ядра ОС Linux, которую в данный момент активно улучшают и конвертируют в новые форматы [51]. Ниже мы определим метод улучшения документации на основе неточных повторов и покажем, каким образом он может быть применён на практике.

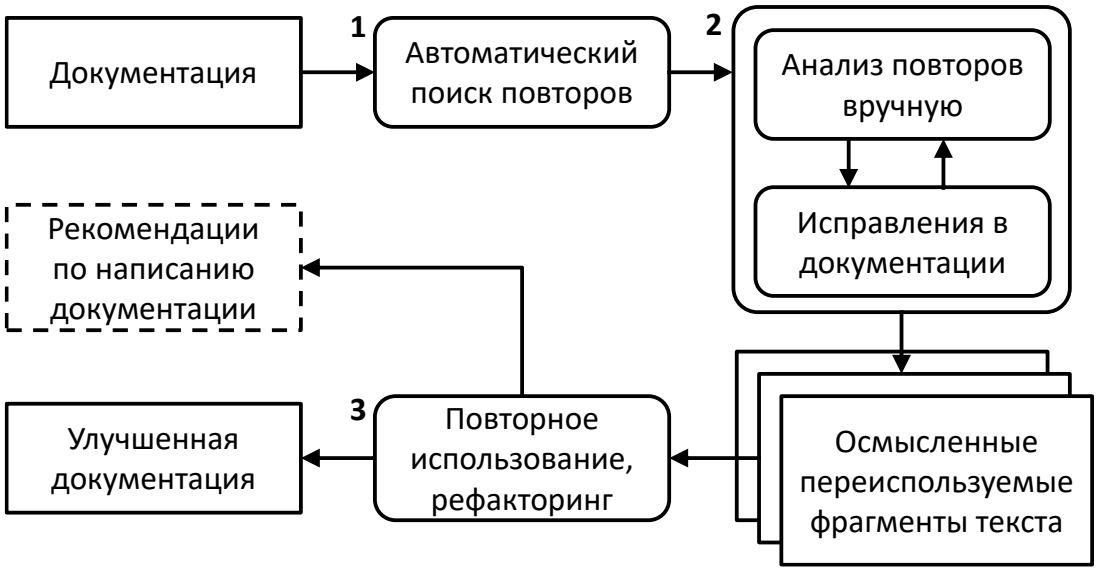

Рис. 4.1.1. Схема метода улучшения документации

Определение 4.1.1. ***Управление повторами*** в документации — это процесс обнаружения и анализа повторов с целью исправления ошибок и достижения единообразия документации, в ходе которого могут применяться техники повторного использования.

Предлагаемый метод улучшения документации основывается на управлении повторами и состоит из несколько фаз — см. рис. 4.1.1.

Первая фаза — автоматическое обнаружение повторов, как точных, так и неточных. Здесь применяются подходы и техники, описанные в предыдущих главах.

Вторая фаза — «ручной» анализ повторов и внесение изменений в документацию. С одной стороны, автоматизированный поиск повторов выдаёт значительную долю ложноположительных несодержательных срабатываний [89]. С другой стороны, даже обнаруженные содержательные повторы зачастую не являются корректными: они нарушают структуру документа и не обладают смысловой замкнутостью. Для того, чтобы решить эти проблемы, к анализу в интерактивном режиме привлекается пользователь. С другой стороны, повторы, обнаруженные автоматически, должны быть проанализированы вручную для того, чтобы внести в документацию правки, исправляющие ошибки и повышающие её

единообразие. Найденные повторы содержат много информации, полезной для решения этих задач. После того, как документация исправляется и унифицируется, повторный поиск обнаруживает ещё большее количество повторов. На выходе второй фазы мы имеем набор осмысленных групп повторов.

Третья фаза — использование найденных повторов. Прежде всего, это автоматизированный рефакторинг для документации в формате DocBook, переиспользование текста на основе найденных повторов. Следует отметить, что использование найденных групп неточных повторов в рамках организации формального переиспользования текста возможно только сторонними средствами в том случае, если документация представлена в других форматах, которые поддерживают данную возможность [76, 110, 111].

Главным результатом третьей фазы является улучшенная документация. Дополнительным результатом третьей фазы могут быть рекомендации по написанию документации и шаблоны фрагментов документации, созданные на базе найденных документации повторов. Эти рекомендации и шаблоны важны для дальнейшей разработки документации: располагая ими, технический писатель, описывая те или иные сущности, может не сверяться с существующими описаниями, а сразу получать информацию о том, каким образом следует структурировать новый текст. На схеме метода эти рекомендации обозначены пунктиром, так как их выработка и автоматизация работы с ними выходят за рамки данной диссертационной работы.

Опишем подробнее, каким образом предложенный метод может быть использован для повышения качества документации. Для этого мы используем ряд показателей качества документации, предложенных в [153]. Рассмотрим эти показатели.

Структура документа (information organization, document structure). Этот показатель характеризует то, насколько хорошо организована представленная в документе информация, на сколько у документа чёткая структура. Этот показатель является высоким, когда структура чёткая, а информация организована таким образом, что читателю легко воспринимать документ.

Оформление (format, writing style). Этот показатель характеризует качество форматирования документа и общую стилистику. На него влияет единообразное использование поясняющих иллюстраций, качество вёрстки и т.д. Стилистика и вёрстка документа являются важными для читателя: хорошо оформленный текст легко читать и воспринимать. Также важно и наличие легко воспринимаемых иллюстраций (диаграмм, графиков), которые упрощают понимание информации, изложенной в документе.

Грамотность (Spelling and grammar) — отсутствие орфографических и грамматических ошибок. Показатель грамотности также важен, т.к. большое количество грамматических и орфографических ошибок затрудняют восприятие текста, снижают мотивацию читателя.

Перечисленные показатели могут быть значительно улучшены при помощи управления повторами.

Первый показатель (структура документа) улучшается следующим образом. Справочная документация описывает большое количество однотипных сущностей, поэтому для описания их свойств используются единообразные разделы и порядок информации в этих разделах. Следует отметить, что данная структура не может быть зафиксирована только оглавлением. Более того, эта структура включает в себя некоторое количество общего текста. Например, описание каждого класса может начинаться с одной и той же следующей фразы: «Данный класс предназначен для …». Описание параметров функции может иметь следующий вид: «Функция имеет … параметров. Первый параметр означает … Второй параметр означает …». В тех случаях, когда описание однотипных сущностей не стандартизовано таким образом, структура документации обычно нуждается в переработке. В итоге, в документации мы получаем большое количество неточных повторов. Эти повторы могут быть могут быть задействованы при организации повторного использования.

Второй показатель (оформление) улучшается путём унификации специфичного для документации языка и стиля: подписей к иллюстрациям, приглашений к рассмотрению примеров кода, информации об авторах, перекрёстных ссылок и

т.д. — все это должно быть единообразным. Также производится унификация форматирования (например, выделение терминов полужирным шрифтом или курсивом), т.к. несообразности в форматировании обычно хорошо видны при анализе неточных повторов.

Третий показатель (грамотность) улучшается, когда технический писатель анализирует различия неточных повторов: орфографические, грамматические и пунктуационные ошибки часто попадают в вариации или же замечаются при внимательном чтении различных фрагментов текста в ходе анализа. Отметим, что не только грамотность, но и другие ошибки попадают в вариации неточных повторов. По мере их исправления растёт уровень повторного использования.

Многие другие показатели, перечисленные в [153], такие, как точность, согласованность (единообразие и целостность), корректность, читаемость и т.д. также косвенно улучшаются в результате управления повторами.

4.2. Автоматизированный рефакторинг DocBook-документации

Опишем автоматизированный рефакторинг документации в формате DocBook, осуществляемый на основе найденных повторов.

Повторы, автоматически найденные в DocBook-документах (первая фаза на схеме метода, рис. 4.1.1), можно использовать в качестве исходных данных для автоматического рефакторинга (третья фаза метода), поскольку они являются группами сходных текстовых фрагментов, в явном виде содержащих архетип и дельту. Вследствие последнего факта группу неточных повторов можно представить средствами языка DRL в виде одного текстового шаблона с набором точек расширения и соответствующими значениями.

Среди автоматически найденных групп повторов пользователь может выбрать группы, заинтересовавшие его в смысле переиспользования. Важно, чтобы повторно используемые фрагменты текста были осмысленными, являясь частью описания функции, прерывания, типа данных и т.д. Выбрав одну из найденных групп, пользователь может создать на её основе переиспользуемый фрагмент текста — информационный элемент или словарь [11, 21]. При этом применяется

подход к рефакторингу, описанный в [23]. На основе точной группы можно создать информационный элемент или словарь, на основе неточной — только информационный элемент.

При рефакторинге выполняется следующая последовательность действий.

1. Создаётся описание переиспользуемой сущности (информационного элемента или словаря), как это показано на листинге 1.6.2.
2. Далее, все повторы выбранной группы заменяются ссылками на это описание, как это показано на листинге 1.6.3. В случае неточных повторов каждая ссылка параметризуются соответствующими значениями точек расширения (см. определение 2.1.10).
3. Координаты вхождений в документ повторов всех оставшихся групп пересчитываются в соответствии с внесёнными в текст изменениями, чтобы инструмент мог корректно осуществлять навигацию по исходному тексту документа.

Текстовые фрагменты, входящие в группу повторов, к которой применяется рефакторинг, могут не быть корректными с точки зрения синтаксиса DocBook. Например, в листинге 4.2.1 повтором является выделенный ***полужирным курсивом*** фрагмент — очевидно, что он нарушает DocBook-разметку, так как содержит лишь закрывающий тег «</emphasis>» и не содержит открывающего.

<emphasis>INI-file ***reader</emphasis> example illustrates a basic use of loading configuration data from a file. In this example there are configuration data for both a production system and for a staging system.*** Because the staging system configuration data are very similar to those for production, the staging section inherits from the production section.

Листинг 4.2.1. Повтор, нарушающий разметку DocBook

DocBook/DRL-документ после рефакторинга должен оставаться корректным, но, как мы видим в примере выше, разметка документа может быть нарушена при рефакторинге. Для этого при рефакторинге выполняется балансировка нарушенной разметки — добавляются соответствующие открывающие и закрывающие теги как в выделенном фрагменте, так и в исходных местах его вхождения.

Пример результата балансировки тегов в определении и применении повторно используемого элемента (в нотации DRL) представлен на листинге 4.2.2; добавленные при балансировке теги выделены *курсивом*.

```
<infelement id="reader_example">
<emphasis>reader</emphasis> example illustrates a basic use of loading configuration data from a file. In this example there are configuration data for both a production system and for a staging system.
</infelement>

……

<emphasis>INI-file</emphasis><infelemref infelemid = "reader_example"/> Because the staging system configuration data are very similar to those for production, the staging section inherits from the production section.
```

Листинг 4.2.2. Пример сбалансированной после рефакторинга разметки

Отметим, что полноценная реализация такой балансировки весьма сложна. Так, например, если мы лишний раз откроем и закроем тег «<para>» (задаёт параграф), то в результирующем документе DocBook получим два параграфа вместо одного, что изменит структуру DocBook-документа и его внешний вид. В силу последнего данное действие в этом случае не будет рефакторингом.

В некоторых случаях можно довольно просто сбалансировать разметку, исключив из повтора открывающий/закрывающий тег, подобный «<para>», если повтор начинается/закрывается этим тегом. Но в случае, когда повторы содержат такие теги в середине, применить такую простую балансировку тегов не представляется возможным. Задача балансировки разметки в общем случае не решена в данной диссертационной работе.

4.3. Пример использования метода

Рассмотрим применение метода управления повторами на примере улучшения руководства пользователя для одной индустриальной системы. В документе описывается среда моделирования систем крупного предприятия [10]. Моделирование осуществляется при помощи десяти графических редакторов, которые позволяют редактировать модели различных видов. При анализе повторов данного

документа было установлено, что функциональность пяти редакторов, практически, идентична (различаются лишь названия моделей и мелкие детали). Тем не менее, различия в соответствующих фрагментах текста были существенными. После унификации этих фрагментов в среднем была достигнута их схожесть 82% (среднее отношение длины архетипа в символах к длине всего описания).

Опишем подробнее улучшение показателей качества документации во время управления повторами.

1. **Структура документа.** При создании описания данных редакторов были раскопированы три фрагмента текста. Затем структура этих фрагментов была существенно переработана разными техническими писателями. В похожих фрагментах были по-разному упорядочены некоторые параграфы. Наблюдался беспорядок и с иллюстрациями: для некоторых фрагментов соответствующие иллюстрации отсутствовали. Также в некоторых повторах отсутствовали некоторые предложения и последовательности предложений, которые должны были присутствовать во всех повторах группы. Обычно это были замечания, предупреждающие пользователя о возможных типичных ошибках при работе с ПО. Эти дефекты были обнаружены при управлении повторами и устранены путём внесения в документ 25 исправлений.
2. **Форматирование.** Были проанализированы различия в описании одинаковых действий и сущностей. По результатам анализа эти описания были унифицированы. Типичный пример различия в описании действия пользователя — «Затем нажмите "Далее"» и «Затем нажмите кнопку "Далее"». Также были устранены различия в использовании выделения текста полужирным шрифтом и курсивом в повторяющихся фрагментах текста. Всего по найденным различиям было сделано 11 исправлений.
3. **Грамотность.** Были исправлены орфографические и пунктуационные ошибки. Всего было внесено 15 исправлений.

Документ, на базе которого построен этот пример, хоть и носит преимущественно справочный характер, всё же не настолько формализован, как, например, типичная API-документация. В нём содержится много развёрнутых описаний

действий пользователя. Соответственно, многие найденные повторы были достаточно большими (несколько предложений). Объём документа в формате «плоского» текста составляет 180 Кб. Количество правок (51), внесённых в документ при управлении повторами, позволяет говорить об эффективности предложенного метода.

Глава 5. Апробация результатов: инструмент Duplicate Finder

Описанные в предыдущих главах алгоритмы, методика и метод реализованы в инструменте Duplicate Finder [60]. Данная глава посвящена описанию этого инструмента.

5.1. Обзор функциональности

Duplicate Finder реализует функциональность для работы с повторами при сопровождении документации. Рассмотрим его возможности.

1. **Поиск повторов.** Инструмент позволяет автоматически находить как точные, так и неточные повторы в отдельном документе. Из форматов документов он поддерживает неформатированный текст и DocBook [147] или DRL [11, 21]. В качестве входных поддерживаются также и другие форматы, но документы в других форматах конвертируются в DocBook или текст при помощи утилиты Pandoc [113]. Если документация состоит из нескольких файлов, они объединяются в один. Поиск повторов осуществляется в двух режимах — автоматическом и интерактивном.
2. **Визуализация повторов.** Инструмент поддерживает навигацию по найденным повторам (рис. 5.1.1). Он отображает список групп (не)точных повторов, их общую часть (архетип) и специфичные для каждого повтора части. Для работы с повторами каждый повтор можно посмотреть в контексте окружающего его текста.
3. **Управление повторами.** Инструмент позволяет редактировать найденные повторы путём изменения их границ. Пользователь может счесть какие-либо из повторов осмысленными и утвердить их, то есть отметить и вывести из дальнейшего рассмотрения (рис. 5.1.2). Таким образом, окончательное решение по вопросу, действительно ли найденный кандидат в повторы имеет значение, как повтор, принимается человеком.

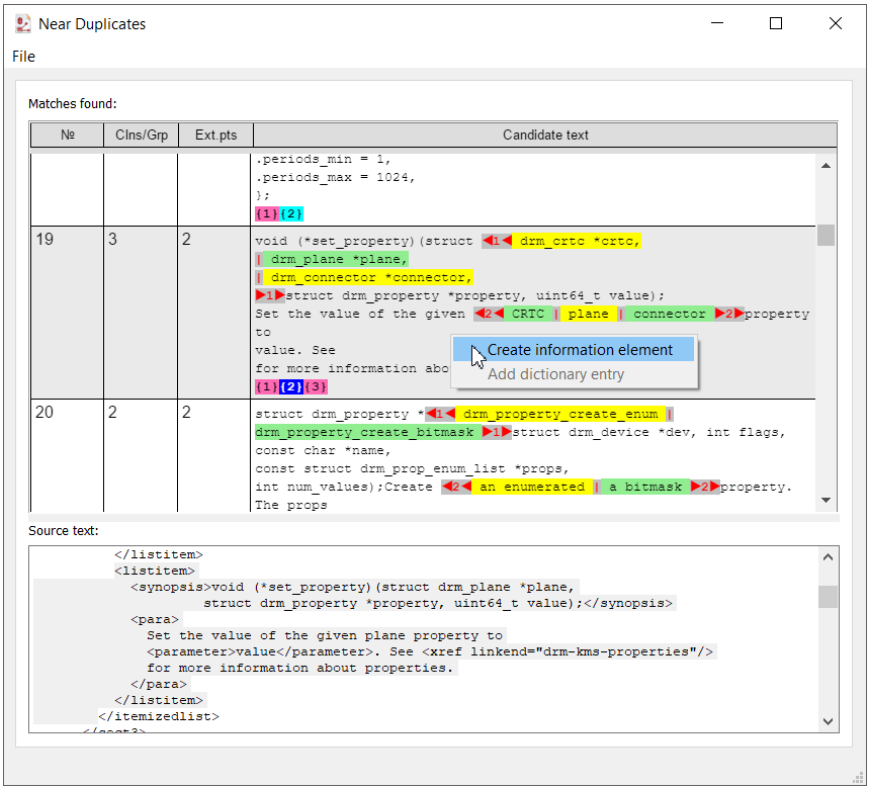

Рис. 5.1.1. Представление результатов автоматического поиска неточных повторов (браузер неточных повторов)

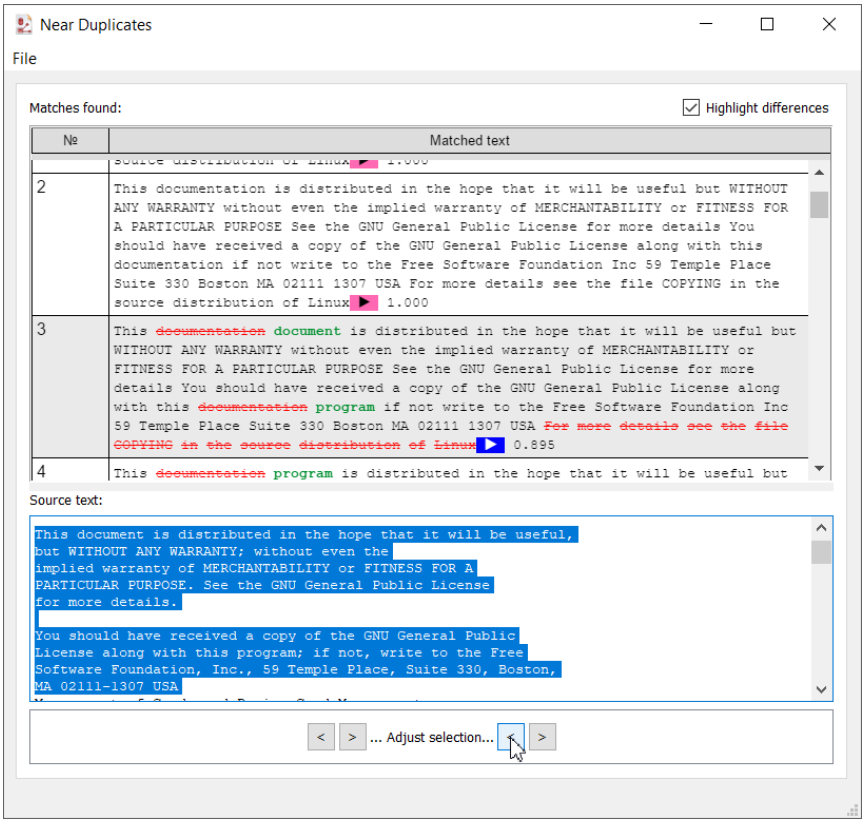

Рис. 5.1.2. Редактирование границ найденных неточных повторов

Также Duplicate Finder поддерживает рефакторинг документов в форматах DocBook и DRL: пользователь может отметить группу неточных повторов, как источник для автоматического создания шаблона. У получившегося шаблона будут формальные параметры, соответствующие точкам расширения (определение 2.1.10) группы неточных повторов. В результате рефакторинга шаблон выделяется в виде новой повторно используемой конструкции DRL (информационного элемента или словаря), а все неточные повторы выбранной группы заменяются на ссылки на шаблон, параметризованные фактическими параметрами, взятыми из текста до рефакторинга. При рефакторинге инструмент сохраняет корректность разметки, обеспечивая, таким образом, корректность документа относительно XML-схемы DocBook.

5.2. Функциональная архитектура

На рис. 5.2.1 представлена функциональная архитектура Duplicate Finder: прямоугольниками представлены блоки функциональности инструмента, стрелки задают основной сценарий использования.

Компонента **Выбор режима работы, настройки** реализует диалоговое окно, которое появляется при запуске Duplicate Finder и позволяет пользователю выбрать режим работы — автоматический (см. рис 5.2.2) или интерактивный (см. рис 5.2.3). Автоматический режим предназначен для предварительного анализа повторов в документе: пользователь имеет возможность быстро определить, сколь много повторов в документации и какие они. Для этого используется алгоритм компоновки неточных повторов, описанный в главе 2. Интерактивный режим предназначен для поиска осмысленных неточных повторов в документе с помощью методики, представленной в главе 3. Кроме того, данная компонента конвертирует различные форматы в «плоский» текст, используя утилиту Pandoc [113], которая должна быть предварительно установлена на компьютере. Исключение делается для формата DocBook, который обрабатывается инструментов

Duplicate Finder «as is». Также в диалоговом окне можно выбрать следующие дополнительные настройки инструмента. В автоматическом режиме (см. рис. 5.2.2) пользователь может:

- изменить минимальную длину в токенах точных повторов, фиксируемых Clone Miner (по умолчанию эта длина равна 1);
- задать действия при обнаружении в найденных повторах повреждённой DocBook-разметки (обработать группу как есть, проигнорировать группу, или автоматически сократить границы повторов, исключая из них синтаксически некорректные конструкции);
- изменить минимально допустимую длину архетипа неточных групп (в токенах, по умолчанию 5);
- оптимизировать выдачу алгоритма компоновки неточных повторов (по умолчанию включено).

В интерактивном режиме пользователь может задать следующие настройки (см. рис. 5.2.3):

- изменить минимальную длину повторов, используемых при построении карты повторов (в токенах, по умолчанию 5);
- изменить максимальную длину повторов, используемых при построении карты повторов (по умолчанию не ограничена);
- изменить минимальную мощность точных групп, повторы которых используются для построения тепловой карты (по умолчанию 2, то есть все группы).

Поиск точных повторов выполняется при помощи инструмента поиска программных клонов Clone Miner [35]. Эти повторы используются в обоих режимах работы инструмента: в автоматическом режиме на их основе строятся (компонуются) неточные повторы, в интерактивном режиме с их помощью строится карта повторов и тепловая карта документа.

Компонента **Компоновка неточных повторов** реализует соответствующий алгоритм.

Браузер неточных повторов (см. пример на рис. 5.1.1) представляет информацию о найденных группах неточных повторов в режиме автоматического поиска. Строка в таблице соответствует одной группе. В первой колонке содержится номер группы, во второй — количество элементов (повторов) в группе, в третьей — количество точек расширения. Наконец, в последней колонке отображаются повторы группы. Для этого предложена следующая метафора визуализации. Точки расширения ограничены двойными треугольниками, между которыми указан номер точки. Внутри этой области разными цветами указываются различные варианты значений данной очки расширения для разных повторов в группе. Чтобы эти значения не сливались, они отображаются разными цветами. Внизу под таблицей с группами находится окно с исходным документом, представленном в текстовом формате или в формате Docbook (последнее используется для случая, когда исходный документ находится в формате Docbook). В этом окне можно увидеть каждый повтор, содержащийся в группе, в том виде, как он входит в исходный документ — это вхождение выделено синим цветом. Также можно посмотреть на тот контекст, в котором находится данное вхождение. Навигация по повторам группы производится с помощью цифр, находящихся в правом нижнем углу ячейки с повторами. Эти цифры отображаются то фиолетовым, то лиловым цветом с тем, чтобы выделяться и не сливаться. При нажатии мышью на каждую из них происходит переключения окна исходного документа на соответствующий повтор.

В браузере неточных повторов можно вызвать функцию экспорта информации в **HTML-отчёт.** Вся функциональность браузера по навигации по множеству групп повторов, найденных для данного документа, будет доступна в Интернет-браузере. В отчёт будет также включён и «плоский» текст самого документа.

Данная функциональность удобна для анализа повторов без использования Duplicate Finder, только средствами Интернет-браузера. Поддерживаются браузеры FireFox и Chrome. Пример отчёта представлен на рис. 5.2.5.

Компонента **Средства рефакторинга для Docbook** позволяет выполнять рефакторинг Docbook-документов. С этой целью необходимо в браузере неточных повторов указать соответствующую группу и выбрать из контекстного меню одну из двух возможностей — создать вариативный повторно используемый шаблон (в терминах DRL — информационный элемент) или словарь.

Компонента **Браузер карты повторов и тепловая карта** реализует интерактивный режим поиска неточных повторов. Эта компонента содержит три разных окна: тепловую карту документа, карту повторов и список найденных групп повторов (см. рис. 5.2.4). Тепловая карта отображает «плотность» повторов в разных частях документов, при этом документ представлен в данном окне целиком. Кроме красно-белой гаммы в этом окне также отображаются (голубым цветом) те участки документа, которые уже вошли в группы повторов, созданные на предыдущих шагах. При нажатии мышкой в какую-то часть тепловой карты в следующем окне (в карте повторов) отображается окрестность документа вокруг места, которое отмечено на тепловой карте. Эта окрестность показывается в виде текста, тоже размеченного цветом, как и в тепловой карте. Наконец, третье окно отображает список групп, найденный в рамках данного режима работы инструмента на предыдущих итерациях. Данное окно аналогично браузеру неточных повторов и впоследствии будет заменено этим браузером, когда будет реализован поиск архетипа и дельты в неточных повторах, найденных с помощью алгоритма поиска по образцу.

Инструмент может работать на операционных системах Windows и Linux.

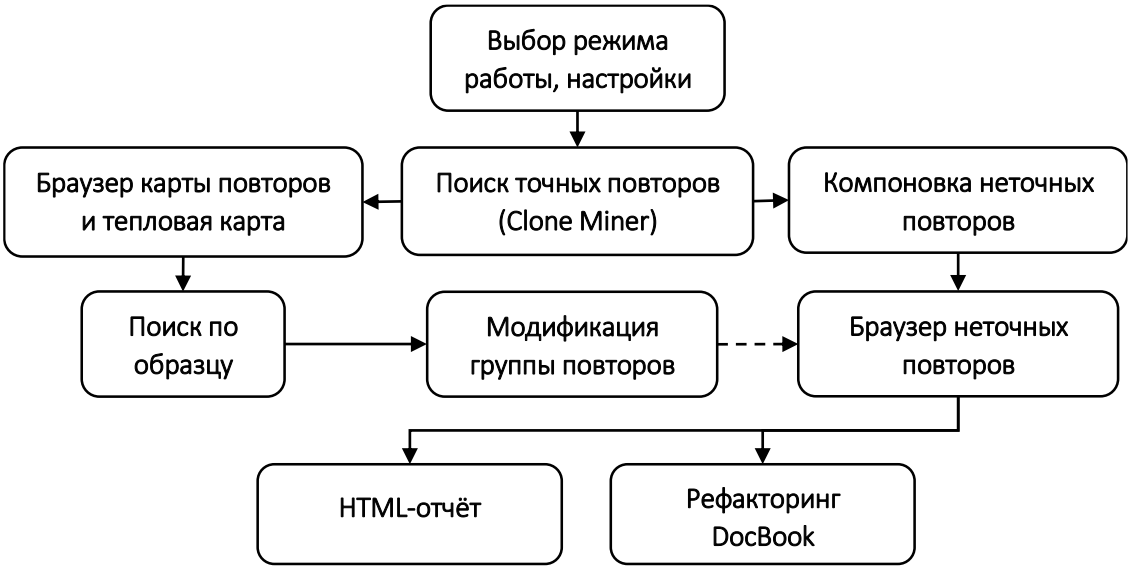

Рис. 5.2.1. Функциональная архитектура Duplicate Finder

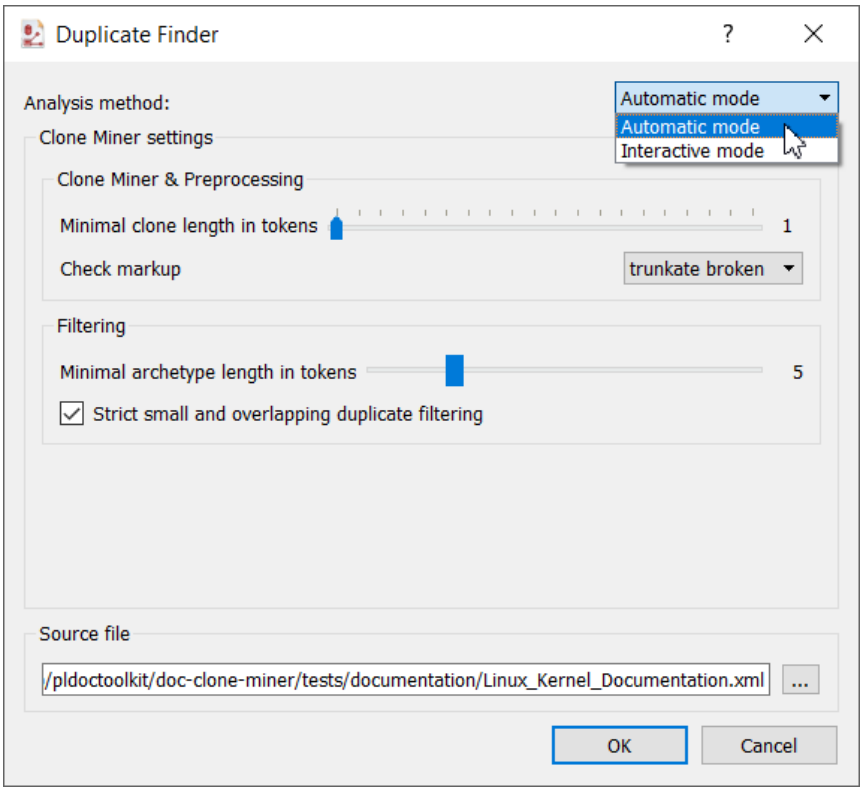

Рис. 5.2.2. Настройки режима автоматического поиска

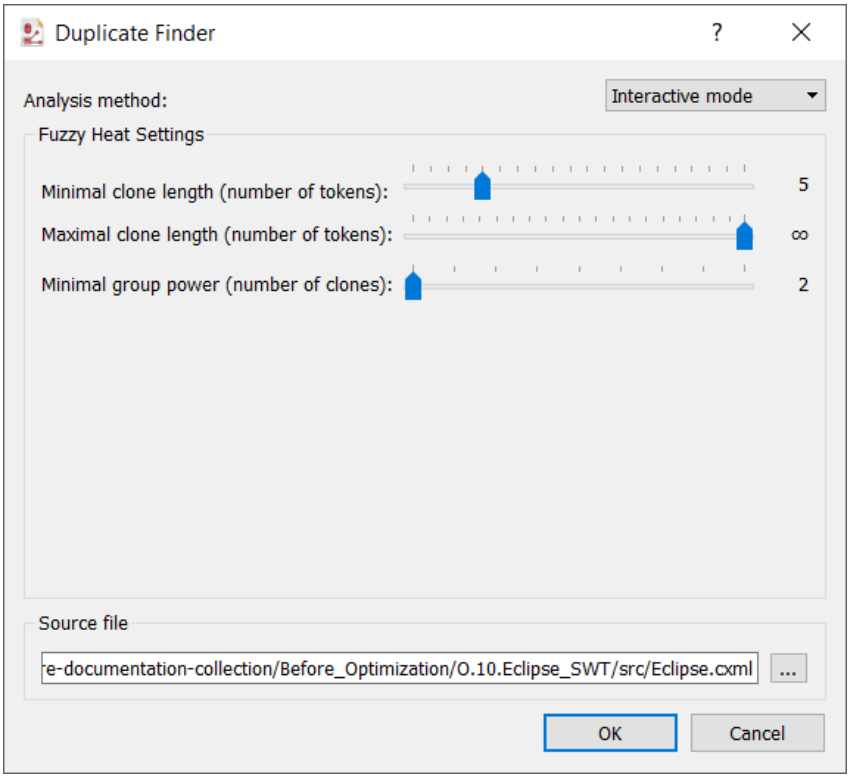

Рис. 5.2.3. Настройки интерактивного режима

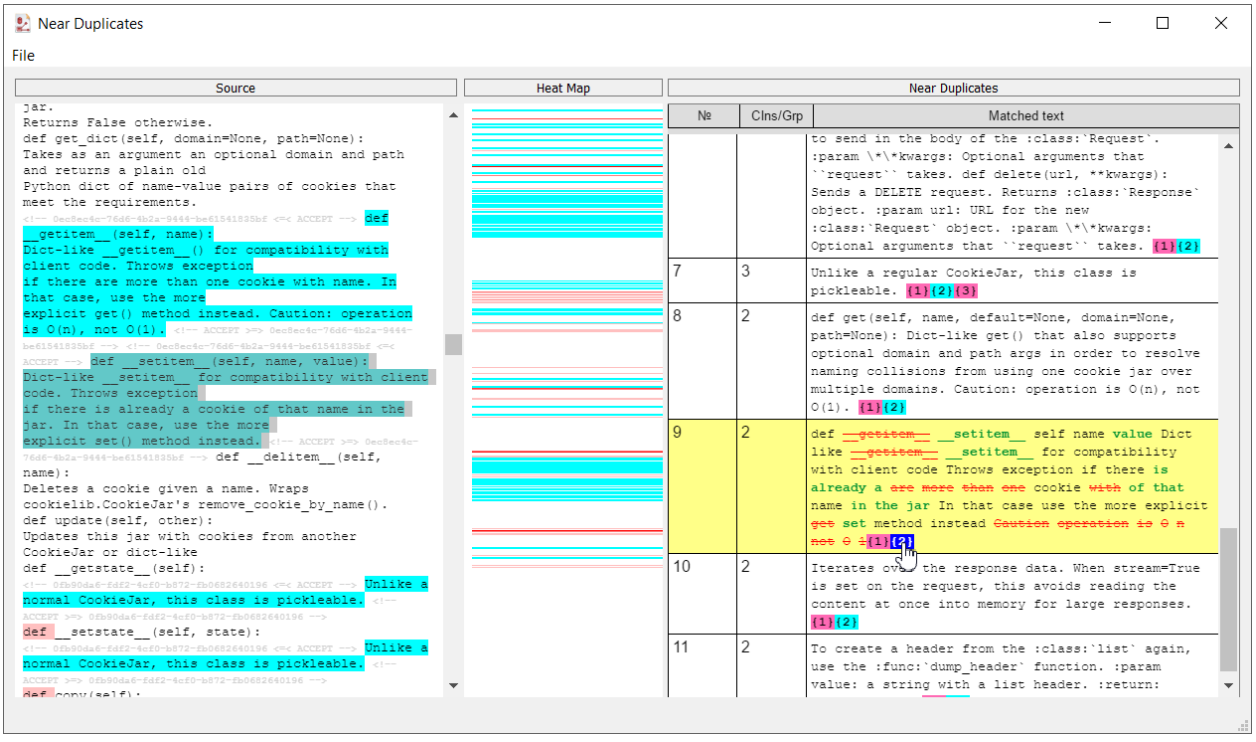

Рис. 5.2.4. Браузер карты повторов и тепловая карта

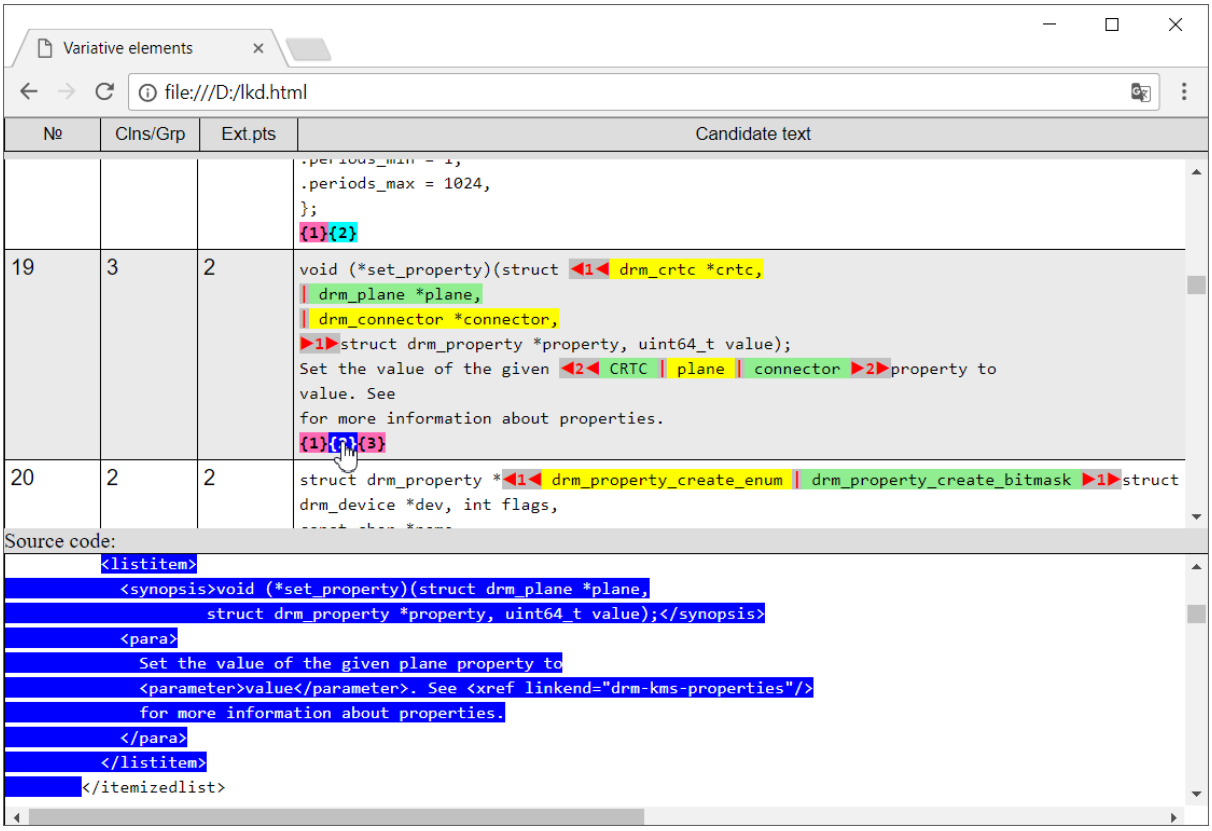

Рис. 5.2.5. HTML-отчет

5.3. Программная архитектура

Инструмент Duplicate Finder [60] реализован на языке Python. Этот язык популярен в научном сообществе при решении задач анализа и обработки текста, аналитических задач и задач машинного обучения. К достоинствам языка можно отнести наличие его реализаций для различных операционных систем и аппаратных платформ, множество готовых программных библиотек, реализующих разнообразные алгоритмы. Также программы на Python хорошо читаются, сам язык лёгок в освоении. Два последних свойства важны при вовлечении молодых исследователей в научную работу, а также при обмене результатами с коллегами из других исследовательских групп. Основными недостатками языка являются низкая производительность его интерпретатора и трудности с распараллеливанием программ. При решении исследовательских задач эти недостатки не являются существенными: при обработке больших данных критические участки кода можно вынести в библиотеки на компилируемых языках, например, на языке C.

Программная архитектура Duplicate Finder представлена на рис. 5.3.1. Прямоугольник являются программными модулями и в основном соответствуют исходным файлам на Python. Стрелками показаны вызовы функциональности одних модулей из других. Цветом выделены программные модули, не являющиеся частями Duplicate Finder.

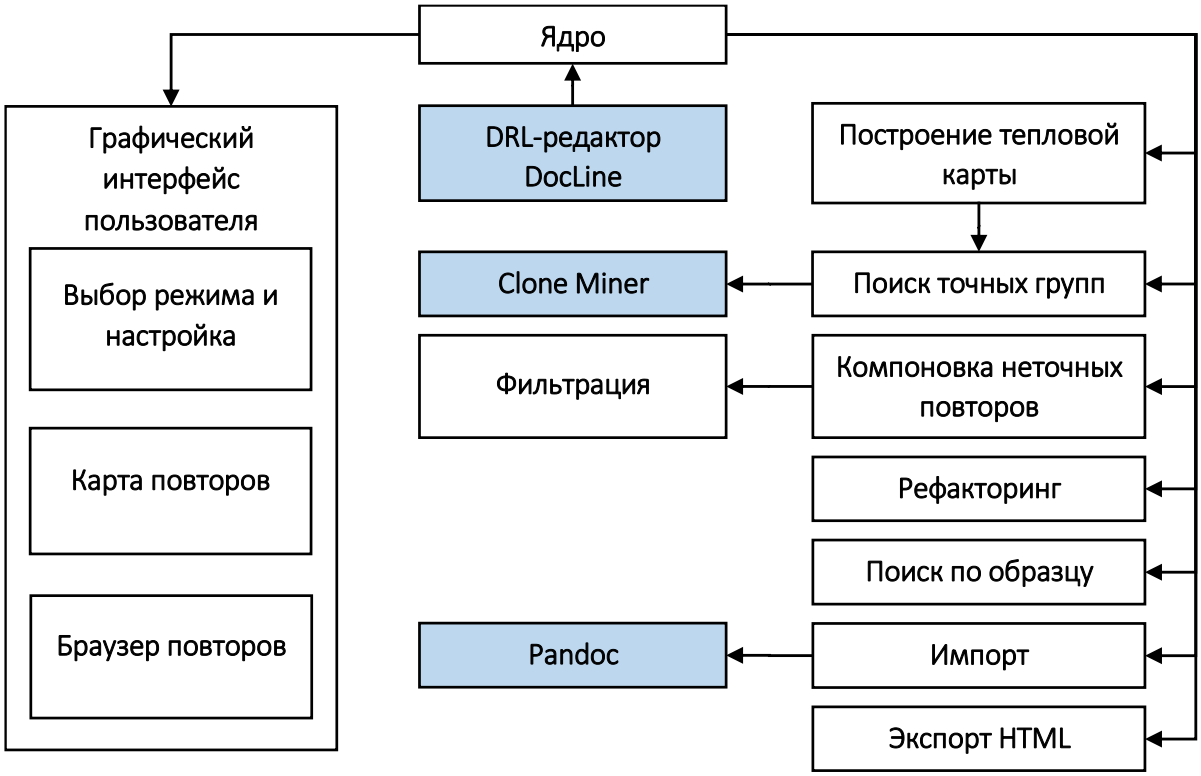

Рис. 5.3.1. Программная архитектура Duplicate Finder

Кратко опишем программные модули инструмента.

- Модуль **Ядро** обеспечивает передачу управления и данных между прочими модулями.
- Модуль **DRL-редактор DocLine** может быть использован для запуска Duplicate Finder на текущем DocBook-документе из среды DocLine [11, 21]. Основной режим работы Duplicate Finder не требует интеграции с пакетом DocLine.
- Модуль **Построение тепловой карты** генерирует цветовую разметку документа и строит тепловую карту.

- Модуль **Поиск точных групп** выполняет автоматический поиск точных групп, используя инструмент Clone Miner [35]. Clone Miner поставляется вместе с Duplicate Finder.
- Модуль **Компоновка неточных повторов** реализует алгоритм автоматического поиска неточных повторов.
- Модуль **Фильтрация** оптимизирует выдачу алгоритма компоновки неточных повторов.
- Модуль **Рефакторинг** реализует автоматизированный рефакторинг документации в формате DocBook.
- Модуль **Поиск по образцу** выполняет поиск неточных повторов по выбранному образцу, реализуя алгоритм поиск по образцу.
- Модуль **Импорт** выполняет конвертацию документов из различных входных форматов в «плоский» текст. Конвертация осуществляется при помощи внешней утилиты Pandoc [113], инсталлируемой отдельно.
- Модуль **Экспорт HTML** позволяет экспортировать текущее состояние браузера неточных повторов в автономный HTML-файл для последующего просмотра в Интернет-браузере.
- Модуль **Графический интерфейс пользователя** обеспечивает выбор режима работы инструмента, выбор параметров и предоставляет пользовательский интерфейс на протяжении работы инструмента. Он реализован на базе библиотеки PyQt [120], позволяющей использовать графический интерфейс фреймворка Qt в программах на языке Python.

Заключение

Подведем итоги данного диссертационного исследования.

1. Предложена формальная модель неточных повторов в программной документации, разработан алгоритм поиска неточных повторов в документации ПО на основе компоновки точных повторов, найденных с помощью метода поиска точных клонов ПО. Доказана корректность алгоритма.
2. Создана методика интерактивного поиска неточных повторов, позволяющая учитывать заданную экспертом семантику повторов. Создан алгоритм поиска по образцу, доказана полнота данного алгоритма.
3. Создан метод улучшения документации ПО на основе неточных повторов, включая автоматизированный рефакторинг документации в формате DocBook.

Предлагаются следующие **рекомендации для использования полученных результатов** в промышленности, образовании и научных исследованиях. Предложенные алгоритмы могут быть реализованы в составе более сложных целевых сервисов по разработке промышленной документации ПО. Модульная архитектура реализации предложенных алгоритмов допускает замену отдельных элементов (алгоритма вычисления редакционного расстояния и др.), а также эффективное распараллеливание. Предложенная в работе методика может быть уточнена в соответствии с особенностями процесса разработки документации в конкретной компании и эффективно применена при сопровождении больших пакетов долгоживущей документации, а также для выработки корпоративного стандарта документации. Предложенные средства разработки могут быть применены на практике — как непосредственно, так и в составе более сложных средств разработки и поддержки документации ПО.

Выделен ряд **перспектив дальнейшей разработки представленной в работе тематики**: создание алгоритмов автоматического поиска неточных повторов,

учитывающих структуру документа, автоматизированное извлечение из документации иерархии объектов, которые документация описывает (с использованием машинного обучения); классификация повторов в зависимости от типа документации, дальнейшее совершенствование инструментов поиска неточных повторов.

Список литературы

1. Ахин, М.Х. Слайсинг над деревьями: метод обнаружения разорванных и переплетенных клонов в исходном коде программного обеспечения / М.Х. Ахин, В.М. Ицыксон // Моделирование и анализ информационных систем. — 2012. — 19 (6). — С. 69–78.
2. Брукс, Ф. Мифический человеко-месяц или как создаются программные системы / Ф. Брукс. — СПб.: Символ-плюс, 1999. — 304 с.
3. Граничин, О.Н. Введение в методы стохастической оптимизации и оценивания. Учеб. пособие / О.Н. Граничин. — СПб.: Издательство С.-Петербургского университета, 2003. — 131 с.
4. Граничин, О.Н. Рандомизированные алгоритмы в задачах обработки данных и принятия решений / О.Н. Граничин // Системное программирование. — 2011. — 6 (1). — С. 141–162.
5. Дробинцев, П.Д. Индустриальные технологии разработки программного обеспечения. Учебное пособие / П.Д. Дробинцев, О.В. Александрова, А.Н. Карпов. — СПб.: Изд-во СПбГТУ, 2016. — 76 с.
6. Зельцер, Н.Г. Поиск повторяющихся фрагментов исходного кода при автоматическом рефакторинге / Н.Г. Зельцер // Труды института системного программирования РАН. — 2013. — Т. 25. — С. 39–50.
7. Иванников, В.П. Использование контрактных спецификаций для представления требований и функционального тестирования моделей аппаратуры / В.П. Иванников, А.С. Камкин, А.С. Косачев, В.В. Кулямин, А.К. Петренко // Программирование – 2007. – 33 (5). – С. 47–61.
8. Кознов, Д.В. Автоматизированный рефакторинг документации семейств программных продуктов / Д.В. Кознов, К.Ю. Романовский // Системное программирование. — 2009. — 4 (1). — С. 128–150.
9. Кознов, Д.В. Основы визуального моделирования / Д.В. Кознов. — М: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2007. — 248 с.

10. Кознов, Д.В. Особенности проектов в области разработки корпоративной архитектуры предприятий / Д.В. Кознов, М.Ю. Арзуманян, Ю.В. Орлов, М.А. Деревянко, К.Ю. Романовский, А.А. Сидорина // Бизнес-информатика. — 2015. — 4 (34). — С. 15–23.
11. Кознов, Д.В. DocLine: метод разработки документации семейства программных продуктов / Д.В. Кознов, К.Ю. Романовский // Программирование. — 2008. — 34 (4). — С. 1–13.
12. Коршунов, А. В. Тематическое моделирование текстов на естественном языке / А.В. Коршунов, А.Г. Гомзин // Труды института системного программирования РАН. — 2012. — Т. 23. — С. 215–242.
13. Лаврищева, Е.М. Моделирование семейств программных систем / Е.М. Лаврищева, А.К. Петренко // Труды института системного программирования РАН. — 2016. — 28 (6). — С. 49–64.
14. Левенштейн, В.И. Двоичные коды с исправлением выпадений, вставок и замещений символов / В.И. Левенштейн // Доклады АН СССР. — 1965. — 163 (4). — С. 845–848.
15. Липаев, В.В. Документирование сложных программных средств / В.В. Липаев. — М.: СИНТЕГ, 2005. — 216 с.
16. Луцив, Д.В. Задача поиска нечётких повторов при организации повторного использования документации / Д.В. Луцив, Д.В. Кознов, Х.А. Басит, А.Н. Терехов // Программирование. — 2016. — 42 (4). — С. 39–49.
17. Луцив, Д.В. Метод поиска повторяющихся фрагментов текста в технической документации / Д.В. Луцив, Д.В. Кознов, Х.А. Басит, О.Е. Ли, М.Н. Смирнов, К.Ю. Романовский // Научно-технический вестник информационных технологий, механики и оптики. — 2014. — 4 (92). — С. 106–114.
18. Луцив, Д.В. Иерархический алгоритм DIFF при работе со сложными документами / Д.В. Луцив, Д.В. Кознов, В.С. Андреев // Системное программирование. — 2012. — 7(1). — С. 57–68.
19. Луцив, Д.В. Обнаружение неточно повторяющегося текста в документации программного обеспечения / Л.Д. Кантеев, Ю.О. Костюков, Д.В. Луцив,

Д.В. Кознов, М.Н. Смирнов // Труды института системного программирования РАН. — 2017. — № 4. — С. 303–314.

20. Романовский, К.Ю. Метод повторного использования документации семейств программных продуктов / К.Ю. Романовский. Диссертация на соискание научной степени кандидата физико-математических наук. — Санкт-Петербургский государственный университет. — 2010. — 111 с.
21. Романовский, К.Ю. Язык DRL для проектирования и разработки документации семейств программных продуктов / К.Ю. Романовский, Д.В. Кознов // Вестник Санкт-Петербургского университета. Прикладная математика. Информатика. Процессы управления. — 2007. — № 4. — С. 110–122.
22. Руководство по AsciiDoc [Электронный ресурс]. — URL: http://asciidoc.org/userguide.html (дата обращения: 10.01.2018).
23. Смирнов, М.Н. Поиск клонов при рефакторинге технической документации / М.Н. Смирнов, Д.В. Кознов, М.А. Смаржевский, А.В. Шутак // Компьютерные инструменты в образовании. — 2012. — № 4. — С. 30–40.
24. Терехов, А.А. Computing Curricula: software engineering и российское образование / А.А. Терехов, А.Н. Терехов // Открытые системы — 2006. — № 8 — С. 61–66.
25. Терехов, А.Н. Технология программирования встроенных систем реального времени / А.Н. Терехов. Диссертация на соискание учёной степени доктора физико-математических наук. — ВЦ СО АН СССР. — 1991. — 44 с.
26. Турдаков, Д.Ю. Texterra: инфраструктура для анализа текстов / Д.Ю. Турдаков, Н.А. Астраханцев, Я.Р. Недумов, А.В. Коршунов // Труды института системного программирования РАН — 2014. — 26 (1). — С. 421–438.
27. Шалыто, А.А. Новая инициатива в программировании. Движение за открытую проектную документацию / А.А. Шалыто // Информационно-управляющие системы. — 2003. — № 4. — С. 52–56.
28. Agile-манифест разработки программного обеспечения. [Электронный ресурс]. — URL: http://agilemanifesto.org/iso/ru/manifesto.html (дата обращения: 10.01.2018).

29. Abboud, A. Tight Hardness Results for LCS and Other Sequence Similarity Measures / A. Abboud, A. Backurs, V. Vassilevska-Williams // Proceedings of the 2015 IEEE 56th Annual Symposium on Foundations of Computer Science. — 2015. — P. 59–78.
30. Abdel Hamid, O. Detecting the Origin of Text Segments Efficiently / O. Abdel Hamid, B. Behzadi, S. Christoph, M. Henzinger // Proceedings of the 18th International Conference on World Wide Web. — 2009. — P. 61–70.
31. Abouelhoda, M.I. Replacing Suffix Trees with Enhanced Suffix Arrays / M.I. Abouelhoda, S. Kurtz, E. Ohlebusch // Journal of Discrete Algorithms. — 2004. — 2 (1). — P. 53–86.
32. Andor, D. Globally Normalized Transition-Based Neural Networks [Electronic resource] / D. Andor, C. Alberti, D. Weiss, A. Severyn, A. Presta, K. Ganchev, S. Petrov, M. Collins // Cornell University Library. — 2016. URL: https://arxiv.org/abs/1603.06042 (online; accessed: 2018.01.20).
33. Backurs, A. Edit Distance Cannot Be Computed in Strongly Subquadratic Time (unless SETH is false) / A. Backurs, P. Indyk // Proceedings of 47th Annual ACM on Symposium on Theory of Computing. — 2015. — P. 51–58.
34. Barnes, L. RUNOFF: A Program for the Preparation of Documents [Electronic resource] / L. Barnes // Bitsavers' PDF Document Archive. — 1973. — URL: http://www.bitsavers.org/pdf/sds/9xx/940/ucbProjectGenie/mcjones/R-37_RUNOFF.pdf (online; accessed: 2018.01.20).
35. Basit, H.A. Efficient Token Based Clone Detection with Flexible Tokenization / H. A. Basit, S. Jarzabek // Proceedings of the 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers. — 2007. — P. 513–516.
36. Bassett, P.G. Framing Software Reuse: Lessons from the Real World / P.G. Bassett. — Prentice-Hall, Inc., 1997. — 384 p.

37. Bassett, P.G. The Theory and Practice of Adaptive Reuse / P.G. Bassett // Proceedings of SIGSOFT Software Engineering Notes. — 1997. — 22 (3). — P. 2–9.
38. Barker, T.T. Documentation for Software and IS Development / T.T. Barker // Encyclopedia of Information Systems. — Elsevier, 2003. — Vol. 1. — P. 683–693.
39. Barrett, D.J. MediaWiki: Wikipedia and Beyond / D.J. Barrett. O'Reilly Media, Inc., 2008. — 380 p.
40. Barrón-Cedeño, A. Plagiarism Meets Paraphrasing: Insights for the Next Generation in Automatic Plagiarism Detection / A. Barrón-Cedeño, M. Vila, M. Antònia Martí, P. Rosso // Computational Linguistics. — 2013. — 39 (4). — P. 917–947.
41. Bergroth, L. A survey of longest common subsequence algorithms / L. Bergroth, H. Hakonen, T. Raita // Proceedings of Seventh International Symposium on String Processing and Information Retrieval. —2000. — P. 39–48.
42. De Berg, M. Computational Geometry / M. De Berg, O. Cheong, M. Van Kreveld, M. Overmars. Springer Berlin Heidelberg, 2008. — P. 220–226.
43. Biegel, B. JCCD: a flexible and extensible API for implementing custom code clone detectors / B. Biegel, S. Diehl // Proceedings of the IEEE/ACM international conference on Automated software engineering. — 2010. — P. 167–168.
44. Bird, S. NLTK: the natural language toolkit / S. Bird // Proceedings of the COLING/ACL on Interactive presentation sessions. — 2006. — P. 69–72.
45. Boyer, R.S. A fast string searching algorithm / R.S. Boyer, J.S. Moore // Communications of the ACM. — 1977. — 20 (10). — P. 762–772.
46. Broder, A. On the Resemblance and Containment of Documents / A. Broder // Proceedings of the Compression and Complexity of Sequences. — 1997. — P. 21–29.
47. Charikar, M.S. Similarity estimation techniques from rounding algorithms / M.S. Charikar // Proceedings of the 34th annual ACM symposium on Theory of computing. — 2002. — P. 380–388.
48. Choi, J.D. It Depends: Dependency Parser Comparison Using A Web-based Evaluation Tool / J.D. Choi, J.R. Tetreault, A. Stent // Proceedings of the 53rd Annual

Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing. — 2015. — P. 387–396.

49. Colibri-Java implementation of Formal Concept Analysis in Java [Electronic resource]. — URL: https://github.com/cunkart/colibri-java (online; accessed: 2018.01.20).
50. ConQAT — a toolkit for rapid development and execution of software quality analyses [Electronic resource]. — URL: http://www.conqat.org/ (online; accessed: 2018.01.20).
51. Corbet, J. The present and future of formatted kernel documentation [Electronic resource] / J. Corbet // Linux Weekly News. — 2016. — URL: https://lwn.net/Articles/671496/ (online; accessed: 2018.01.20).
52. Cordy, J.R. The NiCad Clone Detector / J.R. Cordy, C.K. Roy // Proceedings of IEEE 19th International Conference on Program Comprehension. — 2011. — P. 219–220.
53. Cunningham, W. What is a Wiki [Electronic resource] / W. Cunningham. — 2002. — URL: http://www.wiki.org/wiki.cgi?WhatIsWiki (online; accessed: 2018.01.20).
54. Damerau, F.J. A technique for computer detection and correction of spelling errors / F.J. Damerau // Communications of ACM 7 (3). — 1964. — P. 171–176.
55. Darwin Information Typing Architecture (DITA) Version 1.2 Specification [Electronic resource]. — URL: http://docs.oasis-open.org/dita/v1.2/os/spec/DITA1.2-spec.html (online; accessed: 2018.01.20).
56. Dean, J. Mapreduce: Simplified data processing on large clusters / J. Dean, S. Ghemawat // Proceedings of OSDI 2004. — 2004. — P. 107–113.
57. Dekel, U. Improving API documentation usability with knowledge pushing / U. Dekel, J.D. Herbsleb // Proceeding ICSE '09 Proceedings of the 31st International Conference on Software Engineering. — 2009. — P. 320–330.
58. Van Deursen, A. Domain-specific languages: An annotated bibliography / A. Van Deursen, P. Klint, J. Visser // ACM Sigplan Notices. — 2000. — 35 (6). — P. 26–36.

59. DocReuse: Documentation duplication detector based on Colibri-Java library [Electronic resource]. — URL: https://github.com/docreuse/docreuse (online; accessed: 2018.01.20).
60. Duplicate Finder [Electronic resource]. — URL: http://www.math.spbu.ru/user/kromanovsky/docline/index.html (online; accessed: 2018.01.20).
61. Doxygen [Electronic resource]. — URL: http://www.stack.nl/~dimitri/doxygen/ (online; accessed: 2018.01.20).
62. Earle, R.H. User preferences of software documentation genres / R.H. Earle, M.A. Rosso, K.E. Alexander // Proceedings of the 33rd Annual International Conference on the Design of Communication. — 2015. — P. 46:1–46:10.
63. Edelsbrunner, H. Dynamic Data Structures for Orthogonal Intersection Queries / H. Edelsbrunner // Institut für Informationsverarbeitung, 1980. — Ch. 2.2. — P. 23–32.
64. Foley, J.D. Introduction to computer graphics / J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes. – Reading: Addison-Wesley, 1994.
65. Fowler, M. Refactoring: improving the design of existing code / M. Fowler, K. Beck – Addison-Wesley Professional, 1999. — 337 p.
66. Fowler, M. UML Distilled: A Brief Guide to the Standard Object Modeling Language / M. Fowler. — Addison-Wesley Professional, 2004. — 212 p.
67. Gibson, D. The Volume and Evolution of Web Page Templates / D. Gibson, K. Punera, A. Tomkins // Special Interest Tracks and Posters of the 14th International Conference on World Wide Web. — 2005. — P. 830–839.
68. Gilly, D. Unix in a nutshell / D. Gilly. O'Reilly Books, 1992. — 444 p.
69. GitBook publishing toolkit [Electronic resource]. — URL: https://www.gitbook.com/ (online; accessed: 2018.01.20).
70. Goldfarb, C.F. The Roots of SGML — A Personal Recollection [Electronic resource] / C.F. Goldfarb. — URL: http://www.sgmlsource.com/history/roots.htm (online; accessed: 2018.01.20).
71. Goldfarb, C.F. The SGML Handbook / C.F. Goldfarb. Clarendon Press, 1990. — 663 p.

72. Gusfield, D. Algorithms on Strings, Trees, and Sequences / D. Gusfield // Computer Science and Computational Biology. — Cambridge University Press, 1997. — 534 p.
73. Halbert, C.-L. A mutable, self-balancing interval tree [Electronic resource] / C.-L. Halbert. — URL: https://github.com/chaimleib/intervaltree (online; accessed: 2018.01.20).
74. Hamming, R.W. Error detecting and error correcting codes / R.W. Hamming // The Bell System Technical Journal. — 29 (2). — 1950. — P. 147–160.
75. Honnibal, M. An Improved Non-monotonic Transition System for Dependency Parsing / M. Honnibal, M. Johnson // Proceedings of Conference on Empirical Methods on Natural Language Processing. — 2015. — P. 1373–1378.
76. Horie, M. Tool Support for Crosscutting Concerns of API Documentation / M. Horie, S. Chiba // Proceedings of the 9th International Conference on Aspect-Oriented Software Development. — 2010. — P. 97–108.
77. Horton, W. Designing and writing online documentation (2nd ed.): hypermedia for self-supporting products / W. Horton. John Wiley & Sons, Inc., 1994. — 464 p.
78. Huang, T.-K. An Analysis of Socware Cascades in Online Social Networks / T.- K. Huang, Md. S. Rahman, H. V. Madhyastha et al. // Proceedings of the 22d International Conference on World Wide Web. — 2013. — P. 619–630.
79. Impagliazzo, R. On the Complexity of k-SAT / R. Impagliazzo, R. Paturi // Journal of Computer and System Sciences. — 62 (2). — 2001. — P. 367–375.
80. Indyk, P. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality / P. Indyk, R. Motwani // Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing. — 1998. — P. 604–613.
81. ISO 8879:1986 Information processing — Text and office systems — Standard Generalized Markup Language (SGML), Edition 1 [Electronic resource]. —URL: https://www.iso.org/standard/16387.html (online; accessed 2018-01-20).
82. Jaccard, P. Distribution de la flore alpine dans le Bassin des Dranses et dans quelques regions voisines [Fr.] / P. Jaccard // Bulletin de la Société Vaudoise des Sciences Naturelles. — 37 (140). —1901. — P. 241–272.

83. Jarzabek, S. XVCL: XML-based variant configuration language / S. Jarzabek, P. Bassett, H. Zhang, W. Zhang // Proceedings of the 25th International Conference on Software Engineering. —2003. — P. 810–811.
84. JDK 5.0 Javadoc Technology-related APIs & Developer Guides [Electronic resource] / Sun Microsystems. — 2011. — URL: https://docs.oracle.com/javase/1.5.0/docs/guide/javadoc (online; accessed: 2018-01-20).
85. Jiang, L. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones / L. Jiang, G. Misherghi, Z. Su, S. Glondu // Proceedings of the 29th International Conference on Software Engineering. — 2007. — P. 96–105.
86. Jones N.C. An Introduction to Bioinformatics Algorithms / N.C. Jones, P.A. Pevzner. — Cambridge, MA: MIT Press, 2004. — 435 p.
87. JSR 175: A Metadata Facility for the JavaTM Programming Language [Electronic resource]. — URL: https://jcp.org/en/jsr (online: accessed: 2018-01-20).
88. Juergens, E., Deissenboeck, F., Hummel, B. CloneDetective — A workbench for clone detection research / E. Juergens, F. Deissenboeck, B. Hummel // Proceedings of the 31st International Conference on Software Engineering. — 2009. — P. 603–606.
89. Juergens, E., Can clone detection support quality assessments of requirements specifications? / E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, J. Streit // Proceedings of ACM/IEEE 32nd International Conference on Software Engineering. — Vol. 2. — 2010. — P. 79–88.
90. Koznov, D.V. Clone Detection in Reuse of Software Technical Documentation / D.V. Koznov, D.V. Luciv, H.A. Basit, O.E. Lieh, M.N. Smirnov // Proceedings of International Andrei Ershov Memorial Conference on Perspectives of System Informatics. — 2015. — Lecture Notes in Computer Science. — Vol. 9609. — P. 170–185.
91. Koznov, D.V. Duplicate management in software documentation maintenance / D.V. Koznov, D.V. Luciv, G.A. Chernishev // Proceedings of the 5th International Conference on Actual Problems of System and Software Engineering

(APSSE 2017). CEUR Workshops proceedings. — Vol. 1989. — 2017. — P. 195–201.

92. Knuth, D.E. The TEXbook / D.E. Knuth. — Addison Wesley, 1989. — 483 p.
93. Lamport, L. LATEX: a document preparation system / L. Lamport. Addison-Wesley, 1986. — 242 p.
94. Laue, R. Anti-Patterns in End-User Documentation / R. Laue // Proceedings of the 22Nd European Conference on Pattern Languages of Programs. — 2017. — P. 20:1–20:11.
95. Lethbridge, T.C. How software engineers use documentation: the state of the practice / T.C. Lethbridge, J. Singer, A. Forward // IEEE Software. — 20 (6). — 2003. — P. 35–39.
96. Leonard, S. Guidance on Markdown: Design Philosophies, Stability Strategies and Select Registrations (RFC-7764) [Electronic resource] / S. Leonard. — IETF. — 2016. — URL: https://tools.ietf.org/html/rfc7764 (online; accessed: 2018-01-20).
97. Leskovec, J. Mining of Massive Datasets / J. Leskovec, A. Rajaraman, J.D. Ullman // Cambridge University Press, 2 edition, 2014. — 466 p.
98. Linux Kernel Documentation [Electronic resource]. — URL: https://github.com/torvalds/linux/tree/master/Documentation/DocBook/ (online; accessed 2013-12-10).
99. Löwe, W. Vizzanalyzer — a software comprehension framework / W. Löwe, M. Ericsson, J. Lundberg, T. Panas, N. Pettersson // Proceedings of the Third Conference on Software Engineering Research and Practice in Sweden, Lund University. — 2003. — P. 127–136.
100. Luciv, D.V. Detecting and Tracking Near Duplicates in Software Documentation / D.V. Luciv // Preliminary Proceedings of the 11th Spring/Summer Young Researchers' Colloquium on Software Engineering. — 2017. — P. 125—129.
101. Luciv, D.V. On Fuzzy Repetitions Detection in Documentation Reuse / D.V. Luciv, D.V. Koznov, A.N. Terekhov, H.A. Basit // Programming and Computer Software. — 2016. — 4 (42). — P. 216–224.

102. Mandagere, N. Demystifying data deduplication / N. Mandagere, P. Zhou, M.A. Smith, S. Uttamchandani // Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion. — 2008. — P. 12–17.

103. McIlroy, M.D. A Research Unix reader: annotated excerpts from the Programmer's Manual (Technical report) [Electronic resource] / M.D. McIlroy. — Bell Labs, 1986. — URL: http://www.cs.dartmouth.edu/~doug/reader.pdf (online; accessed: 2018-01-20).

104. Mertz, D. reStructuredText: A light, powerful document markup [Electronic resource]. IBM DeveloperWorks, 2003. — URL: https://www.ibm.com/developerworks/library/x-matters24/ (online; accessed: 2018-01-20).

105. Mikolov, T. Distributed representations of words and phrases and their compositionality / T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado, J. Dean // Proceedings of Advances in neural information processing systems. — 2013. — P. 3111–3119.

106. Nguyen, T.T. ClemanX: Incremental clone detection tool for evolving software / T.T. Nguyen, H.A. Nguyen, N.H. Pham, J.M. Al-Kofahi, T.N. Nguyen // Proceedings of the 31st International Conference on Software Engineering. 2009. — P. 437–438.

107. Niu, X. An overview of perceptual hashing / X. Niu X., Y. Jiao // Acta Electronica Sinica. — 2008. — 36 (7). — P. 1405–1411.

108. Nikula, J. Kernel documentation with Sphinx, part 1: how we got here [Electronic resource] / J. Nikula // Linux Weekly News. — 2016. — URL: https://lwn.net/Articles/692704/ (online; accessed: 2018-01-20).

109. Navarro, G. 2001. A guided tour to approximate string matching / G. Navarro // ACM Computing Surveys. — 2001. — 33 (1). — P. 31–88.

110. Nosál', M. Preliminary report on empirical study of repeated fragments in internal documentation / M. Nosál', J. Porubän // Proceedings of Federated Conference on Computer Science and Information Systems. — 2016. — P. 1573–1576.

111. Nosál’, M. Reusable software documentation with phrase annotations / M. Nosál’, J. Porubän // Central European Journal of Computer Science. — 2014. — 4 (4). — P. 242–258.

112. Oumaziz, M.A. et al. Documentation Reuse: Hot or Not? An Empirical Study // Proceedings of 16th International Conference on Software Reuse. — 2017. — P. 12–27.

113. MacFarlane, J. Pandoc: an universal document converter [Electronic resource] / J. MacFarlane. — URL: http://pandoc.org/ (online; accessed: 2018-01-20).

114. Parnas, D.L. Precise Documentation: The Key to Better Software / D.L. Parnas // The Future of Software Engineering. — 2011. — P. 125–148.

115. PMD Source Code Analyzer [Electronic resource]. — URL: https://pmd.github.io/ (online; accessed: 2018-01-20).

116. Preparata, F.P. Computational Geometry — An Introduction / F.P. Preparata, M.I. Shamos. Springer, 1985. — 390 p.

117. Damerau-Levenshtein (DL) edit distance algorithm for Python in Cython for high performance [Electronic resource]. — URL: https://github.com/gfairchild/pyxDamerauLevenshtein (online; accessed 2018-01-20).

118. Python DiffLib module [Electronic resource]. — URL: https://docs.python.org/3/library/difflib.html (online; accessed 2018-01-20).

119. [PyLevenshtein] Python-Levenshtein library: The Levenshtein Python C extension module contains functions for fast computation of Levenshtein distance and string similarity [Electronic resource]. — URL: https://github.com/ztane/python-Levenshtein/ (online; accessed: 2018-01-20).

120. PyQt Python v2 and v3 bindings for The Qt Company's Qt application framework [Electronic resource]. — URL: https://riverbankcomputing.com/software/pyqt/intro (online; accessed 2018-01-20).

121. Rago, A. Identifying duplicate functionality in textual use cases by aligning semantic actions / A. Rago, C. Marcos, J.A. Diaz-Pace // Software & Systems Modeling. — 2016. — 15 (2). — P. 579–603.

122. Ramaswamy, L. Automatic Detection of Fragments in Dynamically Generated Web Pages / L. Ramaswamy, A. Iyengar, L. Liu, F. Douglis // Proceedings of the 13th International Conference on World Wide Web. — 2004. — P. 443–454.

123. Ratcliff, J.W. Pattern Matching: The Gestalt Approach / J.W. Ratcliff, D.E. Metzener // Dr. Dobb's Journal. — 1988. — 13 (7). — P. 46–72.

124. Rattan, D. Software clone detection: A systematic review / D. Rattan, R. Bhatia, M. Singh // Information and Software Technology. — 2013. — 55 (7). — P. 1165–1199.

125. Rodrigues, I.P. Usability Evaluation of Domain-Specific Languages: A Systematic Literature Review / I.P. Rodrigues, M. de Borba Campos, A.F. Zorzo // Proceedings of Human-Computer Interaction. User Interface Design, Development and Multimodality. — 2017. — P. 522–534.

126. Romanovsky, K. Refactoring the Documentation of Software Product Lines / K. Romanovsky, D. Koznov, L. Minchin // Lecture Notes in Computer Science. — 2011. — Vol. 4980. — P. 158–170.

127. Royce, W.W. Management the development of large software systems / W.W. Royce // Proceedings of the 9th international conference on Software Engineering. — 1987. — P. 328–338.

128. Shi, L. An empirical study on evolution of API documentation / L. Shi, H. Zhong, T. Xie, M. Li // Lecture Notes in Computer Science. — 2011. — Vol. 6603. — P. 416–431.

129. Shieber, S.M. Evidence against the context freeness of natural language / S.M. Shieber // Linguistics and Philosophy. — 2012. — 8 (3). — P. 333–343.

130. Simian — Similarity Analyser [Electronic resource]. — URL: http://www.harukizaemon.com/simian/index.html (online; accessed: 2018-01-20).

131. Singh, J. Understanding Data Deduplication [Electronic resource] / J. Singh. — 2009. — URL: https://www.druva.com/blog/understanding-data-deduplication/ (online; accessed 2018-01-20).

132. Smyth, W. Computing Patterns in Strings / W. Smyth. — Addison-Wesley, 2003. — 423 p.

133. Sommerville, I. Software documentation, Report [Electronic resource] / I. Sommerville. Lancaster University, UK, 2001. — URL: http://www.literateprogramming.com/documentation.pdf (online; accessed: 2018-01-20).

134. Šošić, M. Edlib: a C/C ++ library for fast, exact sequence alignment using edit distance / M. Šošić, M. Šikić // Bioinformatics. — 2017. — 33 (9). — P. 1394–1395.

135. Sajnani, H. SourcererCC: Scaling Code Clone Detection to Big-code / H. Sajnani, V. Saini, J. Svajlenko et al. // Proceedings of the 38th International Conference on Software Engineering. — 2016. — P. 1157–1168.

136. Špakov, O. Visualization of eye gaze data using heat maps / O. Špakov, D. Miniotas // Electronics and Electrical Engineering. — 2007. — 2 (74). — P. 55–58.

137. Pennington, J. Glove: Global vectors for word representation / J. Pennington, R. Socher, C. Manning // Proceedings of the 2014 conference on empirical methods in natural language processing. — 2014. — P. 1532–1543.

138. Steyvers, M. Probabilistic Topic Models / M. Steyvers, T. Griffiths // Handbook of latent semantic analysis. — 2007. — 427 (7). — P. 424–440.

139. Subramanian, S. Live API documentation / S. Subramanian, L. Inozemtseva, R. Holmes // Proceedings of the 36th International Conference on Software Engineering. — 2014. — P. 643–652.

140. Petrov, S. Announcing syntaxnet: The world’s most accurate parser goes open source [Electronic resource] / S. Petrov // Google Research Blog. — 2016. — URL: https://research.googleblog.com/2016/05/announcing-syntaxnet-worlds-most.html (online; accessed: 2018-01-20).

141. Thomas, B. Documentation for software engineers: what is needed to aid system understanding? / B. Thomas, S. Tilley // Proceedings of the 19th annual international conference on Computer documentation. — 2001. — P. 235–236.

142. Vallés, E. Detection of Near-duplicate User Generated Contents: The SMS Spam Collection / E. Vallés, C. Rosso // Proceedings of the 3rd International Workshop on Search and Mining User-generated Contents. — 2011. — P. 27–34.

143. Wagner, R.A. An Extension of the String-to-String Correction Problem / R.A. Wagner, R. Lowrance // Journal of the ACM. — 1975. — 22 (2). — P. 177–183.

144. Wagner, R.A. The String-to-String Correction Problem / R.A. Wagner, M.J. Fischer // Journal of the ACM. — 1974. — 21 (1). — P. 168–173.

145. Wagner, S. Analyzing Text in Software Projects / S. Wagner, D. Méndez Fernández // The Art and Science of Analyzing Software Data. — Elsevier, 2015. — P. 39–72.

146. Wang, P. Local Similarity Search for Unstructured Text / P. Wang, C. Xiao, J. Qin, W. Wang, X. Zhang, Y. Ishikawa // Proceedings of International Conference on Management of Data. — 2016. — P. 1991–2005.

147. Walsh, N. DocBook 5: The Definitive Guide / N. Walsh. O'Reilly Media, 2010. — 552 p.

148. Weiss, E.H. How To Write Usable User Documentation / E. H. Weiss. 2nd edition, Phoenix: Oryx Press, 1991. — 267 p.

149. Williams, K. Near Duplicate Detection in an Academic Digital Library / K. Williams, C.L. Giles // Proceedings of the ACM Symposium on Document Engineering. — 2013. — P. 91–94.

150. Wingkvist, A. Analysis and visualization of information quality of technical documentation / A. Wingkvist, W. Löwe, M. Ericsson, R. Lincke // Proceedings of the 4th European Conference on Information Management and Evaluation. — 2010. — P. 388–396.

151. Zhang, Q. Efficient Partial-duplicate Detection Based on Sequence Matching / Qi Zhang, Yu Zhang, H. Yu, Xu. Huang // Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval. — 2010. — P. 675–682.

152. Zend Framework 3 Documentation [Electronic resource]. — URL: https://framework.zend.com/learn (online; accessed: 2018-01-20).
153. Zhi, J. Cost, benefits and quality of software development documentation: A systematic mapping / J. Zhi, V. Garousi-Yusifoğlu, B. Sun, G. Garousi, S. Shahnewaz, G. Ruhe // Journal of Systems and Software. — 2015. — Vol. 99 — P. 175–198.

Приложение. Пример группы неточных повторов

В данном приложении представлен пример группы повторов, найденной по образцу в интерактивном режиме. В качестве образца для поиска был взят один из повторов группы, выделенный на тепловой карте и карте повторов в документации СУБД PostgreSQL.

Образец представлен на листинге А.1:

> To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the table's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the table. However, a superuser can alter ownership of any table anyway.)

Листинг А.1. Образец описания прав, необходимых для смены владельца таблицы СУБД PostgreSQL

Поиск по данному образцу выдал 13 вхождений. Эти вхождения и результаты их коррекции представлены в табл. А.1.

Табл. А.1. Неточные повторы, найденные по образцу

№	Найдено	Отредактировано
1	To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the table's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the table. However, a superuser can alter ownership of any table anyway.)	To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the table's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the table. However, a superuser can alter ownership of any table anyway.)

№	Найдено	Отредактировано
2	To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the sequence's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the sequence. However, a superuser can alter ownership of any sequence anyway.)	To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the sequence's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the sequence. However, a superuser can alter ownership of any sequence anyway.)
3	To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the type's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the type. However, a superuser can alter ownership of any type anyway.)	To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the type's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the type. However, a superuser can alter ownership of any type anyway.)
4	To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the table's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the table. However, a superuser can alter ownership of any table anyway.)	To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the table's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the table. However, a superuser can alter ownership of any table anyway.)
5	alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the conversion's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the conversion. However, a superuser can alter ownership of any conversion anyway.)	To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the conversion's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the conversion. However, a superuser can alter ownership of any conversion anyway.)

№	Найдено	Отредактировано
6	To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the view's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the view. However, a superuser can alter ownership of any view anyway.)	To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the view's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the view. However, a superuser can alter ownership of any view anyway.)
7	To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the operator's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the operator. However, a superuser can alter ownership of any operator anyway.)	To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the operator's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the operator. However, a superuser can alter ownership of any operator anyway.)
8	alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the materialized view's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the materialized view. However, a superuser can alter ownership of any view anyway.)	To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the materialized view's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the materialized view. However, a superuser can alter ownership of any view anyway.)
9	To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the operator class's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the operator class. However, a superuser can alter ownership of any	To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the operator class's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the operator class. However, a superuser can alter ownership of any operator class anyway.)

№	Найдено	Отредактировано
10	To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the collation's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the collation. However, a superuser can alter ownership of any collation anyway.)	To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the collation's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the collation. However, a superuser can alter ownership of any collation anyway.)
11	alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the aggregate function's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the aggregate function. However, a superuser can alter ownership of any	To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the aggregate function's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the aggregate function. However, a superuser can alter ownership of any aggregate function anyway.)
12	alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the function's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the function. However, a superuser can alter ownership of any function anyway.)	To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the function's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the function. However, a superuser can alter ownership of any function anyway.)
13	To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the domain's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the domain. However, a superuser can alter ownership of any domain anyway.)	To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the domain's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the domain. However, a superuser can alter ownership of any domain anyway.)