САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра Системного Программирования

Иванов Всеволод Юрьевич

Семантический поиск в программном коде

Дипломная работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

_____
подпись

Научный руководитель:
к. ф.-м. н. Барашев Д. В.

_____
подпись

Рецензент:
Бреслав А. А.

_____
подпись

Санкт-Петербург
2013

SAINT-PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics Faculty

Software Engineering Chair

Ivanov Vsevolod

Concept Location in Source Code

Graduation Thesis

Admitted for defence.
Head of the chair:
professor Andrey Terekhov

_____
signature

Scientific supervisor:
PhD Dmitry Barashev

_____
signature

Reviewer:
Andrey Breslav

_____
signature

Saint-Petersburg
2013

Оглавление

Введение

Процесс поддержки и улучшения программного обеспечения состоит из повторяющихся задач, из которых самая частая – изменение. Она обычно вызвана *запросом на изменение*, который определяет, что именно необходимо поменять в системе. Примером такого запроса может быть фраза: «Добавить возможность платежей по кредитной карте в систему продаж». Реализация этого требования обычно начинается с поиска в коде мест, связанных с понятиями из запроса на изменение, такими как «платеж», «кредитная карта» и т.д.

Авторами запросов на изменение обычно бывают пользователи системы. Пользователи системы продаж знают понятия предметной области, такие как «клиент», «счет» и т.д., но ничего не знают о реализациях таких понятий в системе. Задачей поиска в программном коде является нахождение тех компонентов, которые реализуют соответствующие понятия. Надо отметить, что запросы на изменение могут быть сформулированы не только в терминах предметной области, но и в терминах реализации, таких как «сервер», «список», «стек» и т.д.

Исходными данными к задаче поиска в программном коде являются требования, сформулированные на естественном языке. Результатом решения такой задачи является набор компонентов системы, которые реализуют эти требования. В идеальном случае существующие провязки внешней документации (архитектуры и требований) и исходного кода предоставляют все необходимые данные для поиска. Однако, на практике внешняя документация устаревает быстрее, чем код, а иногда и вовсе отсутствует. Тогда и возникает необходимость в применении специальных техник поиска.

Если программист хорошо понимает имеющийся исходный код, за-

дача поиска обычно достаточно проста. Однако, она все равно может оказаться нетривиальной для больших систем с высокой степенью связности. Сложность вызвана, в том числе, тем, что входные и выходные данные у задачи поиска принадлежат разным уровням абстракции: входные данные выражены на естественном языке, а выходными данными являются файлы на языке программирования. Для того, чтобы осуществить преобразование с одного уровня абстракции на другой, необходимы дополнительные знания о:

1. предметной области,
2. техниках программирования,
3. идиомах,
4. алгоритмах,
5. структурах данных,
6. архитектуре программ.

Поиск в исходном коде является интуитивным и неформальным процессом, когда программисты хорошо разбираются в системе. В случае же, когда опыт и интуиция не могут дать нужного ответа, программисты широко используют *синтаксический поиск*, основанный на родстве имен переменных и понятий из предметной области. Например, в процессе поиска данных о «платеже», программист ищет имена переменных, такие как «payment», «payBill» и т.д. Когда найден нужный идентификатор, разработчик изучает близлежащий код и определяет, является ли это место искомым или случайным совпадением.

Одним из самых широко известных инструментов для синтаксического поиска является утилита *grep* из Unix. Несмотря на частое использование этой техники, у нее есть известные проблемы. Она базируется на соответствии между понятием предметной области и идентификатором в коде, и не выдает подходящих результатов в случае, если понятие выражено в коде неявно или когда автор кода использовал синонимы для имен переменных. Также продуктивность метода снижают омонимы.

Чуть более сложной разновидностью синтаксического поиска является поиск, использующий *регулярные выражения*. Он позволяет находить элементы проекта, содержащие не только точные вхождения слова, но и вхождения его словоформ, а также позволяет строить сложные поисковые запросы.

Однако, синтаксический поиск не в состоянии обнаружить элементы проекта, которые связаны с ключевыми словами косвенно. Например, если вы хотите найти те файлы в проекте, которые имеют отношение к шрифтам, вполне вероятно, что место, где шрифты загружаются из настроек, не будет найдено, так как нужный метод может называться просто “Load()”.

Существует альтернативный способ поиска, названный *семантическим*. Он позволяет решать такие задачи, так как ищет в исходном коде области, связанные с данной не только на уровне синтаксиса.

Однако, известные инструментальные средства, которые поддерживают семантический поиск, осуществляют его, используя только исходный код проекта.

Создано много различных инструментальных средств, которые помогают в разработке программного обеспечения: баг-трекеры, системы

контроля версий и т.д. Большинство из них хранит о проекте какую-то дополнительную информацию и историю изменений. Назовем эту информацию *метаданными проекта*.

Некоторые зависимости между компонентами системы не могут быть выявлены непосредственно из кода даже с использованием семантического поиска. Также есть некоторые элементы проекта, которые изменяются крайне часто и неявным образом влияют на все остальные элементы. При работе над найденной ошибкой или поставленной задачей было бы полезно также уметь заранее находить такие компоненты и проверять их на необходимость внесения изменений.

Надо отметить, что в проектах с развитой инфраструктурой все инструментальные средства и метаданные связаны друг с другом. Например, это выражается в том, что в комментариях к фиксации изменений в системе управления версиями пишется номер задачи в баг-трекере. Подобная интеграция позволяет рассчитывать, что поиск по проекту с метаданными окажется более эффективным, чем просто поиск по исходному коду.

1. Обзор

1.1. Инструменты для просмотра кода

Существует целый набор инструментов, предоставляющих богатые возможности просмотра и изучения имеющегося исходного кода:

1. Atlassian FishEye [1],
2. OpenGrok [17],
3. ViewVC [5],
4. Trac [24],
5. LXR Cross-Referencer [27].

Большинство из них представляют собой веб-приложение, в настройках которого указываются адрес репозитория с исходным кодом проекта и, возможно, другие ресурсы проекта, например, баг-трекер. Инструмент при старте осуществляет индексирование проекта, а затем в удобной форме представляет код в виде веб-страниц. Обычно такие инструменты имеют следующие возможности:

1. синтаксический поиск по коду,
2. поиск только по именам переменных,
3. поиск только в определенной папке,
4. поиск по истории,
5. поддержка сложных поисковых запросов (включающих AND и OR),

6. инкрементальное обновление индекса,

7. cross-reference (возможность быстрой навигации между файлами по гиперссылкам).

Такие инструменты используются большинством крупных open-source проектов, часто они даже доступны в Интернете, например, Android XRef [3]. Многие из средств для просмотра исходного кода построены поверх какой-нибудь готовой библиотеки для поиска, например, Apache Lucene [4].

1.2. JRipples

JRipples [10] реализует семантический поиск при помощи технологии DepIR. Она объединяет технику поиска зависимостей и синтаксический поиск способом, похожим на обычное поведение программистов, которые чередуют поиск и просмотр информации.

Реализация синтаксического поиска в этом продукте основана на библиотеке Lucene, применяемой также в данной работе.

Техника поиска зависимостей основана на пересечении зависимостей программы способом, похожим на поиск в глубину, однако, поиск выполняется не компьютером, а программистами. Статический поиск зависимостей обычно начинается с класса, который содержит точку входа в программу: функцию main() или init(). Программисты проходят по графу зависимостей, принимая решения, которые направляют поиск.

Например, в языке Java, класс A зависит от класса B, если:

1. в классе A есть ссылка на класс B:

    1) в поле данных,

2) в локальной переменной,

3) в аргументе метода,

4) в преобразовании типов;

2. класс A наследуется от класса B;

3. класс A реализует интерфейс из класса B.

Если текущий класс D не удовлетворяет поисковому запросу, программист должен указать, какой из зависимых классов для D удовлетворяет ему. Если ни один из зависимых классов не подойдет, это означает, что на предыдущем шаге было принято неверное решение, и поиск возвращается на один уровень вверх, а затем выбирается другой зависимый класс. Такой алгоритм повторяется до тех пор, пока не будет найдет подходящий класс.

Далее описывается стандартный процесс поиска с использованием технологии DepIR. Для удобства мы будем описывать эту технику в применении к методам, но все будет аналогично, если применять ее к классам.

Первым шагом является составление поискового запроса. Затем библиотека, осуществляющая синтаксический поиск, предоставляет результаты запроса, отсортированные по убыванию релевантности. Потом программист просматривает полученный список. Если искомый класс найден, то процедура поиска завершается. Если текущий найденный класс совсем не связан с искомой задачей, то программист переходит к следующему результату. Если текущий класс не точно соответствует запросу, однако связан с предметной областью, то программист переходит к исследованию графа зависимостей при помощи техники, описанной выше. При этом найденный класс рассматривается как точка

входа в технику поиска зависимостей. В процессе изучения кода результатов поиска разработчик может увидеть новые слова, связанные с предметной областью, переформулировать запрос и начать процесс заново.

Программисты исследуют граф зависимостей, изучая компоненты по соседству с выбранной точкой входа. На каждом шаге просматривается текущий выбранный компонент. Если становится понятно, что он не связан с искомой задачей, поиск продолжается по последующим компонентам или возвращается к предыдущему шагу. Если программисты решают переформулировать запрос и пересчитать релевантность результатов в процессе изучения какого-либо метода, порядок просмотра следующих компонентов также будет обновлен. В любой момент поиска через зависимости разработчик может его прервать и вернуться к просмотру результатов синтаксического поиска.

Процесс переформулирования поисковых запросов и исследования графа зависимостей продолжается до тех пор, пока не будет найден искомый результат. Дополнительно реализована функция: если программист отмечает в системе, что какой-либо метод не отвечает критериям поискового запроса, то этот метод удаляется из поиска и игнорируется до его завершения. Это позволяет гарантировать, что метод не будет просматриваться дважды, а также позволяет избегать циклов в графе зависимостей.

1.3. Статья «An Information Retrieval Approach to Concept Location in Source Code»

В данной работе [8] предлагается техника для поиска в исходном коде, основанная на методе Latent Semantic Indexing (LSI) в условиях

отсутствия внешней документации.

В разработанном авторами статьи прототипе имеется 2 возможности поиска: пользователь может сам формулировать запросы к системе или использовать автоматическую генерацию запросов. Второй вариант может быть удобен в случае, когда пользователь недостаточно хорошо знаком с кодом. Тогда у него есть возможность сформулировать запрос на естественном языке, а система дополнит его идентификаторами из кода, которые связаны с исходными словами.

Разработанный инструмент поддерживает языки C, C++ и Java.

Также в статье проводится сравнение применяемой техники поиска, использующей LSI, с другими техниками, в том числе с поиском через зависимости и синтаксическим поиском. По сравнению с поиском через зависимости предлагаемый метод дает схожие результаты, но не требует предварительного знания системы. А по сравнению с синтаксическим поиском этот метод дает существенно лучшие результаты, даже при использовании регулярных выражений.

Однако, данный метод основывается только на исходном коде и не использует другие возможности для поиска, например, информацию из системы контроля версий.

2. Постановка задачи

2.1. Причины

С увеличением размера системы, численности команды и сроков разработки задача поиска в програмном коде приобретает все большую актуальность по следующим причинам:

1. С ростом количества написанного программного кода растет количество связей между частями системы, и изменение одного фрагмента должно повлечь за собой изменения все большего числа компонентов системы. Поэтому важно уметь находить фрагменты кода, связанные с данным.

2. Автор конкретного фрагмента кода в состоянии исправить в нем ошибки за значительно более короткий срок, чем другие разработчики. При достаточно большом размере команды, работающей над проектом, задача поиска автора кода, который вызвал ошибку, является нетривиальной.

2.2. Исходные данные и ожидаемые результаты

В данной работе рассматривается задача поиска в программном коде, подразумевающая следующие входные данные:

1. исходный код проекта;

2. система контроля версий для этого исходного кода;

3. баг-трекер проекта;

4. провязка задач из баг-трекера и коммитов в системе контроля версий;

5. непосредственно поисковый запрос, сформулированный на естественном языке.

В результате работы системы ожидаются следующие результаты:

1. список файлов, которые соответствуют поисковому запросу, отсортированный по их релевантности;

2. список разработчиков, которые внесли наибольший вклад в написание кода, связанного с поисковым запросом, отсортированный по их относительному вкладу.

2.3. Анализируемый язык программирования

Наиболее распространены 2 методологии разработки программного обеспечения: структурная и объектно-ориентированная. В структурной модели основой для написания кода является динамическая проекция системы. В такой модели система рассматривается как процесс преобразования данных, имеющий внешние источники данных и стоки для результатов. Для ее построения требуется преобразование требований к системе с переносом акцентов с субъектов на процессы. Мышление человека является объектно-ориентированным, поскольку только объекты могут непосредственно наблюдаться, а процессы могут наблюдаться только как изменение состояния объектов (Декарт). Объектно-ориентированное описание системы акцентирует внимание на субъектах, что приближает его к описанию на естественном языке. Любое действие (сказуемое) имеет явного или неявного субъекта (действующее лицо). Таким образом, объекты являются семантически гораздо более целостным понятием, так как они соответствуют предметам из реального мира.

Исходя из вышесказанного, при семантическом поиске по проектам, написанным на объектно-ориентированных языках, мы можем получить более точные результаты. В данной работе в качестве целевого был выбран язык Java, так как в отличие от других объектно-ориентированных языков, например от C#, в Java каждый файл почти всегда содержит ровно один класс (интерфейс, перечисление), если не считать вложенных. Необходимо также отметить, что в C# еще используются partial-классы, которые делают необходимым в процессе индексирования объединять некоторые файлы. Все это дает возможность в Java в качестве документов при индексировании использовать файлы.

2.4. Используемые метаданные

Из перечисленных выше возможных метаданных в большинстве существующих проектов можно использовать только данные системы контроля версий и баг-трекер. В данной работе в качестве целевой VCS был выбран Git, как один из самых распространенных. К дополнительным аргументам в сторону использования данной системы можно отнести ее распределенность, то есть в локальной копии каждого разработчика хранится вся история изменений. Это позволяет осуществлять запросы к системе локально, без обращений к внешнему серверу, что ускоряет процесс в несколько раз.

3. Предлагаемое решение

3.1. Краткое описание алгоритма поиска

Для решения задачи поиска в программном коде будет применяться семантический поиск. Существует достаточно много готовых реализаций семантического поиска по произвольному тексту; в данной работе [9] выбрана одна из них, затем произведена ее адаптация к программному коду. Потом результаты запроса к используемой библиотеке обрабатываются с помощью разработанной математической модели, при этом учитывается информация из метаданных проекта: системы контроля версий и баг-трекера.

3.2. Семантический поиск

Модели словесного пространства занимают ключевое место в исследованиях, связанных с семантическим поиском. Одной из самых известных таких моделей является Latent Semantic Analysis [7] [25].

Основной идеей, лежащей в основе моделей словесных пространств, является создание линейных пространств высокой размерности на основе статистики частоты использования слов; в них слова будут представлены контекстными векторами. При этом близость направлений векторов отвечает за семантическую схожесть. Это утверждение объясняется гипотезой, что слова с похожими значениями чаще всего встречаются в похожих контекстах. Согласно этой гипотезе, если мы находим два слова, которые постоянно используются в одинаковых контекстах, то мы считаем, что эти слова обозначают одно и то же. Необходимо отметить, что данная гипотеза не требует, чтобы слова встречались вместе, достаточно, чтобы эти слова встречались с одинаковыми другими сло-

вами. Эта гипотеза была подтверждена многими экспериментами [20] [2].

В стандартной модели словесного пространства векторное пространство высокой размерности создается путем сбора данных в *матрицу совместной встречаемости* F, в которой каждая строка F_w представляет уникальное слово, а каждый столбец F_c соотвествует контексту c (обычно это набор из нескольких слов или документ). В случае алгоритма LSI мы имеем матрицу, основанную на документах.

Основная идея матрицы встречаемости заключается в том, что строки F_w эффективно образовывают векторы в пространстве высокой размерности, где компонентами этих векторов становятся нормализованные частоты встречаемости. При этом размерность пространства определяется количеством столбцов в матрице, которое равно количеству контекстов (например количеству документов в исходном наборе данных). Мы называем векторы контекстными, потому что они представляют контексты, в которых встречались слова. Фактически, контекстные векторы - это профили встречаемости слов, поэтому мы можем определить схожесть между словами в терминах схожести векторов, например как косинус угла между ними.

Есть несколько причин, почему модели словесного пространства доказывают свою эффективность и наглядность во всем большем количестве исследований:

1. Векторные пространства математически хорошо определены и изучены. Мы знаем, как себя ведут векторные пространства, и имеем целый набор алгебраических инструментов для работы с ними.

2. Методологии словесных пространств делают семантику исчислимой, они позволяют нам определить семантическую похожесть в

математических терминах, и предлагают способ ею управлять.

3. Такие модели предлагают чисто описательный подход к семантическому моделированию и не требуют предварительных знаний из области лингвистики и семантики, а просто оперируют имеющимися данными.

4. Геометрическая метафора выглядит интуитивно понятной.

Однако, известно некоторое количество проблем, связанных с моделями словесных пространств, в основном, с их эффективностью и масштабируемостью. Такие проблемы становятся особенно заметными при попытке применения этих моделей к задачам реального мира и большим объемам данных. Основой этих проблем является высокая размерность контекстных векторов, которая линейно зависит от размера данных. Если мы используем встречаемость, основанную на документах, то размерность пространства равна количеству документов в исходном наборе; если основанную на словах – то размеру словаря, который обычно даже больше. Это означает, что любые операции с матрицей встречаемости становятся очень долгими при возрастании исходного объема данных.

Другой важной проблемой матрицы встречаемости является то, что большинство клеток в ней будут иметь нулевое значение. Это объясняется тем, что большая часть слов может встречаться только в очень ограниченном наборе контекстов (также этот феномен называют законом Зипфа [26]). В типичной матрице встречаемости больше 99% клеток - нулевые.

Для того, чтобы решить проблемы, связанные с высокой размерностью, большая часть известных и успешных моделей, таких как LSA,

используют статистические *методы уменьшения размерности*. Стандартный LSA использует *Singular Value Decomposition* (SVD), который представляет собой технику факторизации матрицы. Эта техника может быть использована, чтобы разложить и аппроксимировать матрицу встречаемости таким образом, чтобы конечная матрица имела гораздо меньше столбцов - обычно несколько сотен – и была гораздо плотнее. Надо отметить, что SVD не является единственным способом достичь такого результата, однако остальные часто используемые способы (*принципиальный компонентный анализ* и *независимый компонентный анализ*) используют тот же принцип: сначала данные собираются в большую матрицу встречаемости, а затем она преобразовывается в меньшую и более плотную. Несмотря на то, что эти методы математически хорошо изучены и опробованы, есть несколько причин, почему стоит избегать использования таких техник:

1. Методы уменьшения размерности, такие как SVD, вычислительно достаточно дорогие, что приводит к большому потреблению памяти и долгому времени исполнения. Для многих реальных задач, основанных на больших объемах данных, не удается дождаться результата выполнения алгоритма.

2. Метод уменьшения размерности обычно представляет собой однократную операцию, что означает, что весь процесс сбора матрицы встречаемости и дальнейшего ее преобразования должен быть выполнен с самого начала каждый раз при изменении данных. Невозможность добавления новых данных в модель является серьезным недостатком, так как многим задачам требуется возможность легко обновлять модель.

3. И, самое важное, такие техники не могут избежать построения первоначальной огромной матрицы встречаемости. С этим связано две проблемы. Во-первых, размер этой матрицы делает ее вычислительно сложной, поэтому этот шаг уже требует существенных затрат ресурсов в серьезных задачах. Во-вторых, какие-либо результаты работы с моделью могут быть получены только после завершения первоначального процесса сбора данных и преобразования матрицы, что может потребовать существенного времени. Было бы хорошо иметь возможность анализировать модель еще в процессе сбора данных.

Как альтернативу моделям, похожим на LSA, которые сначала создают большую матрицу встречаемости, а затем производят над ней преобразования для уменьшения размерности, существует инкрементальная модель словесного пространства, называемая *Random Indexing*. Основная ее идея заключается в том, чтобы аккумулировать контекстные векторы, основываясь на встречаемости слов в контекстах.

Техника Random Indexing представляет собой двухшаговую операцию:

1. Каждому контексту (документу или слову) в исходных данных присваивается уникальный случайно сгенерированный индексный вектор. Эти векторы разреженные, имеют высокую размерность (d) порядка нескольких сотен или тысяч и состоят из небольшого количества случайно распределенных +1 и -1, а остальные компоненты вектора нулевые.

2. Контекстные векторы создаются в процессе сканирования текста. Каждый раз, когда слово встречается в каком-либо контексте,

индексный вектор этого контекста добавляется к контекстному вектору слова. Таким образом, слова представлены d-размерными контекстными векторами, которые являются суммой векторов контекстов.

Надо отметить, что эта методология отличается от традиционной способом создания контекстных векторов. Здесь контекстные векторы имеют размерность d, которая меньше, чем количество контекстов c. В случае $d == c$ эти векторы просто представляли бы собой нулевые с единственной единицей в позиции, равной номеру контекста, и были бы все ортогональны друг другу. Однако, если $d < c$, то эти векторы почти ортогональны, поэтому матрица F' в этом методе является аппроксимацией исходной матрицы F. Этот же результат достигается применением методик, похожих на SVD, однако в Random Indexing сразу строится приближенная матрица, без построения исходной.

Эта методика основана на [6], где продемонстрировано, что существует гораздо больше почти ортогональных векторов в пространстве высокой размерности, чем действительно ортогональных. Это означает, что мы можем аппроксимировать ортогональности просто выбором случайных направлений в таком пространстве. Почти ортогональность случайных векторов послужила ключом к созданию целого набора методов, таких как Random Projection [15] и Random Indexing. Математическое обоснование этих методов более полно представлено в статье [21].

По сравнению с другими моделями словесного пространства Random Indexing имеет следующие преимущества:

1. Это - инкрементный метод, что означает, что контекстные векторы могут быть использованы для измерения схожести, как только

была проиндексирована даже небольшая часть исходных данных.

2. Размерность векторов d является параметром, то есть размерность устанавливается один раз и больше никогда не увеличивается. Это является существенным улучшением, так как при больших массивах данных размерность могла приводить к резкому росту вычислительной сложности.

3. Уменьшение размерности в данной технике получается автоматически, что позволяет избежать дорогостоящих операций преобразования больших матриц и существенно ускоряет процесс индексирования.

Random Indexing была эмпирически проверена на большом количестве экспериментов. Например, в [13] приводится описание применения этой методики для решения части поиска синонимов в экзамене TOEFL. В этой части испытуемому предлагается найти синоним к данному слову из 4 возможных вариантов. Процент верно найденных синонимов составил около 64,5-67, что сравнимо с результатами LSA (64,4%), а также с результатами иностранных абитуриентов на вступительных экзаменах в колледжи США. Путем некоторых оптимизаций можно даже добиться результата в 72%. Описание некоторых других экспериментов можно найти тут: [14] [22] [16].

3.3. Математическая модель

Для каждого найденного документа его итоговый ранг вычисляется как сумма исходного ранга, который был получен из библиотеки SemanticVectors, и количества его упоминаний в коммитах, умноженного на 0.01. Число 0.01 было выбрано эмпирически, оно объясняется тем,

что при работе над одной задачей количество коммитов крайне редко превышает сотни. Полученные числа нормализуются, чтобы попадать в диапазон от 0 до 1.

Основой для этой модели является рейтинг результатов по итогам поиска с использованием алгоритма Random Indexing. В дальнейшем этот результат корректируется, чтобы повысить релевантность тех результатов, которые часто изменялись, а также включить в результаты поиска файлы, которые изменялись вместе с уже найденными. Наглядное представление описанной выше схемы представлено в (1).

$$targetRank = normalize(sourceRank + commitsCounts * 0.01)$$

при этом:

$targetRank$	–	конечный ранг документа
$normalize$	–	нормализация полученных значений так, чтобы они попадали в интервал от 0 до 1
$sourceRank$	–	исходный ранг документа в результатах из SemanticVectors
$commitsCounts$	–	количество упоминаний документа в коммитах

(1)

Кроме данных из системы контроля версий также используются данные из баг-трекера. Фактически, новых данных о коде в нем уже не содержится, однако там есть информация о том, что некоторые коммиты связаны друг с другом, так как производились в рамках решения одной и той же задачи. Определить такие коммиты можно, так как обычно в сообщении к ним указывается номер задачи. Мы будем счи-

тать такие коммиты за один.

3.4. Программная реализация

3.4.1. Используемые библиотеки

3.4.1.1. Индексирование и поиск. Существует несколько реализаций описанных выше технологий семантического поиска. Некоторые полностью независимы (jLSI), а некоторые базируются на других фреймворках для поиска (S-Space Package). В качестве основы было решено выбрать библиотеку Apache Lucene, так как у нее есть следующие премущества.

1. Масштабируемая и высокоскоростная индексация:
    1) свыше 150 Гб/час на современном оборудовании;
    2) требуется небольшой объем оперативной памяти;
    3) размер индекса составляет 20-30% от исходных данных.
2. Мощный, точный и эффективный поисковый алгоритм:
    1) ранжированный поиск — лучшие результаты показываются первыми;
    2) множество мощных типов запросов: запрос фразы, wildcard запросы, поиск интервалов и т. д.;
    3) возможность одновременного поиска и обновления индекса.
3. Кроссплатформенное решение:
    1) исходный код полностью написан на Java;
    2) наличие портов на другие языки программирования.

3.4.1.2. Семантический поиск. Существует много реализаций семантического поиска, в том числе технологии Random Indexing, для библиотеки Lucene:

1. jLSI [19],
2. S-Space Package [28],
3. SemanticVectors [23].

В данной работе была выбрана библиотека SemanticVectors по следующим причинам:

1. реализовано несколько алгоритмов семантического поиска, в том числе Latent Semantic Indexing и Random Indexing;
2. полная и понятная документация;
3. проект активно развивается и используется.

3.4.1.3. Система контроля версий. Работа с системой Git осуществляется с помощью библиотеки JGit [18] по следующим причинам:

1. по утверждению разработчиков, эта библиотека реализует всю функциональность Git;
2. библиотека написана на Java и имеет открытый исходный код;
3. работа с Git через стандартные системные средства не представляется достаточно удобной, так как предполагает необходимость разбирать и анализировать их текстовый вывод, а также требует запуска большого количества внешних процессов, что может сказаться на производительности;

4. библиотека используется в качестве стандартного средства для интеграции Git в Eclipse и имеет активных пользователей и разработчиков.

3.4.2. Индексирование

Предлагаемое решение состоит из двух частей: индексирования и собственно поиска. Индексирование выполняется в две стадии.

3.4.2.1. Lucene. Исходный набор файлов, содержащих программный код, индексируется при помощи библиотеки Apache Lucene.

На этом этапе необходимо отфильтровать те файлы, которые не должны входить в область поиска (картинки, служебная информация из системы контроля версий, различные скрипты и т.д.). Существует возможность настройки параметров, по которым производится фильтрация; по умолчанию оставляются только файлы с расширением .java, которые не расположены в служебной папке VCS.

Далее необходимо превратить текст на языке программирования в текст, по которому можно осуществлять осмысленный поиск. Этот процесс разделяется на 2 задачи:

1. Игнорировать в программном коде зарезервированные слова языка, так как они часто встречаются и не несут смысловой нагрузки для поиска. Также имеет смысл исключить из поиска самые часто встречающиеся служебные слова из английского языка.

2. Основная семантическая информация в коде будет содержаться в комментариях и именах переменных, классов и других элементов. Комментарии фактически являются предложениями на естественном языке, поэтому остаются в неизменном виде. А имена пере-

менных требуется преобразовывать. Данная работа поддерживает 2 соглашения об именовании переменных: слова, разделенные разными регистрами, и слова, отделенные нижним подчеркиванием.

Для решения этих задач необходимо было модифицировать стандартный парсер библиотеки Lucene. Так как код в современных языках программирования достаточно похож на английский язык, смысла реализовывать анализатор заново нет. Таким образом, над стандартным парсером библиотеки Lucene сделана некоторая надстройка, которая преобразовывает некоторые элементы из исходного кода в несколько слов на естественном языке в процессе анализа.

3.4.2.2. SemanticVectors. Файлы, полученные в результате индексирования Lucene, подаются на вход библиотеке SemanticVectors, которая проводит анализ методом Random Indexing и строит файлы контекстных векторов.

3.4.3. Поиск

В разработанной системе поиск состоит из нескольких этапов:

1. Исходный запрос подается на вход библиотеке SemanticVectors, которая формирует список найденных файлов с их первоначальным рангом релевантности.

2. Необходимо для каждого найденного файла определить коммиты, в которых производилось его изменение. Это производится при помощи операции *git blame*, которая выводит для каждой строки файла хэш коммита, в котором она последний раз модифицировалась. Такой путь более предпочтителен, чем использование опера-

ции *git log*, которая выводит абсолютно все коммиты, затрагивающие файл, даже если все изменения из этих коммитов были уже затерты последующими модификациями. Таким образом, на выходе данного этапа мы получаем список коммитов, которые имели отношение к задаче.

3. Далее коммиты, которые относятся к одной задаче, объединяются. Связанность коммита с задачей определяется по ее номеру в баг-трекере, который содержится в сообщении к коммиту. Номер извлекается из сообщения при помощи регулярных выражений.

4. Далее для объединенных коммитов извлекаются списки файлов, которые были в них изменены. Потом вычисляются конечные рейтинги файлов по формуле, описанной выше, на основе исходных рейтингов и встречаемости в коммитах.

5. Для списка результатов выполняется операция *git blame*, чтобы узнать количество строк кода, написанного каждым разработчиком. Затем эти количества умножаются на рейтинг файлов, содержащих строки, и суммируются. Это позволяет получить на выходе алгоритма еще и список разработчиков, которые внесли наибольший вклад в данную задачу.

4. Апробация разработанной системы

Система, разработанная в рамках данной работы, была протестирована на двух крупных проектах с открытым исходным кодом: Kotlin [11] и MPS [12] от компании JetBrains. Репозитории обоих этих проектов организованы с помощью системы контроля версий Git, в каждом из них более 10000 коммитов; также оба проекта обладают баг-трекерами. Тестирование проводилось следующим образом: по каждому проекту эксперта просили написать несколько примеров запросов, ожидаемых результатов по ним, а также список разработчиков, которые внесли наибольший вклад в задачи, связанные с этими запросами. Далее в разработанной системе, а также в библиотеках Lucene и SemanticVectors, проводился поиск по запросам, которые предложил эксперт. Затем результаты поиска сравнивались с эталоном, а также эксперта просили дополнительно оценить их релевантность. Были получены следующие выводы:

1. Результаты разработанной системы заметно более полные и точные, чем результаты поиска только в библиотеке SemanticVectors. Это позволяет сделать вывод, что использование для поиска по проекту не только исходного кода, но и метаданных является перспективной идеей.

2. Полученные рейтинги файлов недостаточно точны. Более подробно это объясняется в разделе 6.

3. Приблизительно в половине предложенных запросов были найдены релевантные файлы с достаточно высоким рейтингом. Однако, существовали запросы, для которых результаты поиска в библиотеке Lucene, то есть обыкновенного синтаксического поиска, были

более точными. Так как данная работа изначальные результаты поиска берет из библиотеки SemanticVectors, а для этих запросов ее результаты тоже были неверными, можно сделать вывод, что для некоторых запросов в некоторых проектах нельзя основываться на результатах этой библиотеки. Данная библиотека реализует семантический поиск на основе моделей словесных пространств, которые предполагают, что слова, которые встречаются в одинаковых контекстах, похожи по смыслу. Вполне возможно, что это предположение является верным не для всех файлов с исходным кодом.

4. Необходимо учитывать тот факт, что разработанная система находит не только абсолютно релевантные запросу файлы, но и те файлы, которые могут потребовать изменений в контексте запроса. Под этот критерий с разной вероятностью могут попадать практически все файлы в проекте. Например, если есть какой-нибудь файл настроек, который перезаписывается в каждом коммите, то достаточно вероятно, что он потребует изменений в коммите, вызванным данной задачей, хотя непосредственно к запросу он отношения не имеет. Такая особенность результатов поиска может приводить в замешательство пользователя системы, так как количество результатов существенно больше, чем при синтаксическом поиске. Возможно, улучшение системы подсчета рейтингов поможет отсортировать результаты таким образом, чтобы релевантные файлы имели более высокий рейтинг, чем файлы, которые просто могут потребовать изменений.

5. Результаты

Разработана система, которая позволяет в проекте с достаточным объемом метаданных по поисковому запросу, сформулированному на естественном языке, указать:

1. файлы, которые соответствуют поисковому запросу;
2. файлы, которые скорее всего также потребуют изменений при модификации файлов из предыдущего пункта;
3. разработчиков, которые внесли наибольший вклад в код файлов, связанных с данной задачей.

6. Возможность развития

Основным направлением дальнейшего развития системы является улучшение математической модели для учета в рейтинге файла языковой конструкции, которая в нем встречается. В частности, если искомые слова из запроса встречаются в названии класса, то файл, содержащий этот класс, должен иметь более высокий рейтинг, чем файл, в котором слова из запроса встречаются в названии локальной переменной. Такой механизм предполагает улучшение процесса парсинга исходного кода, чтобы он воспринимался не как текст, а как синтаксическое дерево, при этом чем выше в этом дереве положение элемента, тем выше должен быть его рейтинг.

Текущая реализация системы дает достаточно хорошие результаты для поисковых запросов, которым не соответствуют напрямую какие-либо классы, однако, в случае запросов, на которые ожидается получить конкретный файл, рейтинг этого файла оказывается не самым высоким. Это позволяет утверждать, что описанное выше направление является крайне желательным.

Вторым направлением развития является улучшение механизма построения индексов к системе контроля версий. Даже несмотря на использование распределенных VCS, которые хранят всю информацию локально, время выполнения запросов к ним остается достаточно большим.

Так как система предполагает хранение индексов достаточно большого размера, более целесообразным видится создание веб-приложения на основе нее, чем интеграция в существующие IDE. Это позволит использовать систему не только разработчикам, но и, например, менеджерам, которые смогут найти программиста, которому целесообразно

поручить задачу.

Список литературы

[1] Atlassian. FishEye. — 2013. — URL: http://www.atlassian.com/software/fisheye/overview (online; accessed: 26.05.2013).

[2] Charles W. Contextual correlates of meaning // Applied psycholinguistics. — Cambridge University Press, 2000. — Vol. 21.

[3] Chiossi Rodrigo. AndroidXRef. — 2013. — URL: http://androidxref.com/ (online; accessed: 26.05.2013).

[4] Foundation The Apache Software. Apache Lucene Core. — 2013. — URL: http://lucene.apache.org/core/ (online; accessed: 26.05.2013).

[5] Group The ViewCVS. ViewVC. — 2013. — URL: http://www.viewvc.org/ (online; accessed: 26.05.2013).

[6] Hecht-Nielsen R. Context vectors; general purpose approximate meaning representations self-organized from raw data // In Zurada, J. M.; Marks, R. J. II; Robinson, C. J.; Computational intelligence: imitating life. — IEEE Press, 1994.

[7] Indexing by Latent Semantic Analysis / S. Deerwester, S. Dumais, G. Furnas et al. // Journal of the society for information science, 41(6). — 1990. — Vol. 41, no. 6.

[8] An Information Retrieval Approach to Concept Location in Source Code / Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, Jonathan I. Maletic // Proceedings of the 11th Working Conference on Reverse Engineering. — WCRE '04. — Washington, DC, USA : IEEE Computer Society, 2004. — P. 214–223.

[9] Ivanov Vsevolod. SSearchM. — 2013. — URL: https://github.com/seva-ask/SSearchM (online; accessed: 26.05.2013).

[10] JRipples. DepIR. — 2013. — URL: http://jripples.sourceforge.net/jripples/manual/concepts/icconceptlocation.html (online; accessed: 26.05.2013).

[11] JetBrains. Kotlin. — 2013. — URL: http://kotlin.jetbrains.org/ (online; accessed: 30.05.2013).

[12] JetBrains. MPS. — 2013. — URL: http://www.jetbrains.com/mps/ (online; accessed: 30.05.2013).

[13] Kanerva P., Kristofersson J., Holst A. Random Indexing of text samples for Latent Semantic Analysis // Proceedings of the 22nd annual conference of the cognitive science society. — New Jersey : Erlbaum, 2000.

[14] Karlgren J., Sahlgren M. From words to understanding // In Uesaka, Y.; Kanerva, P.; Asoh, H.; Foundations of real-world intelligence. — Stanford: CSLI Publications, 2001.

[15] Latent Semantic Indexing: a probabilistic analysis / C. H. Papadimitriou, P. Raghavan, H. Tamaki, S. Vempala // Proceedings of the 17th ACM symposium on the principles of database systems. — ACM Press, 1998.

[16] M. Sahlgren., J. Karlgren. Automatic bilingual lexicon acquisition using Random Indexing of parallel corpora // Journal of Natural Language Engineering. — 2005. — June. — no. Special Issue on Parallel Texts.

[17] OpenGrok. OpenGrok. — 2013. — URL: http://opengrok.github.io/OpenGrok/ (online; accessed: 26.05.2013).

[18] Penguin King. Java GIT. — 2013. — URL: http://www.jgit.org/ (online; accessed: 26.05.2013).

[19] Project X-Media. java Latent Semantic Indexing. — 2013. — URL: http://hlt.fbk.eu/en/technology/jlsi (online; accessed: 26.05.2013).

[20] Rubenstein H., Goodenough J. Contextual correlates of synonymy // Communications of the ACM. — 1965. — Vol. 8, no. 10.

[21] Sahlgren M. An introduction to random indexing // In Methods and Applications of Semantic Indexing Workshop at the 7th International Conference on Terminology and Knowledge Engineering, TKE 2005, August 16, Copenhagen, Denmark. — 2005.

[22] Sahlgren M., Coster R. Using bag-of-concepts to improve the performance of support vector machines // Proceedings of COLING 2004. — Geneva, 2004.

[23] package was created as part of a project by the University of Pittsburgh Office of Technology Management The, has been developed, maintained by contributors from the University of Texas Queensland University of Technology the Austrian Research Institute for Artificial Intelligence Google Inc. et al. The Semantic Vectors Package. — 2013. — URL: https://code.google.com/p/semanticvectors/ (online; accessed: 26.05.2013).

[24] Software Edgewall. Trac. — 2013. — URL: http://trac.edgewall.org/ (online; accessed: 26.05.2013).

[25] Using Latent Semantic Analysis to improve access to textual information / S. Dumais, G. Furnas, T. Landauer, S. Deerwester // Proceedings of CHI'88. — New York : ACM, 1988.

[26] Zipf G. K. Human behavior and the principle of least effort. — Addison-Wesley, 1949.

[27] community LXR. The Linux Cross Referencer. — 2013. — URL: `http://lxr.linux.no/` (online; accessed: 26.05.2013).

[28] group at UCLA led by David Jurgens Natural Language Processing, Keith Stevens under the advisory of Dr. Michael Dyer. The S-Space Package. — 2013. — URL: `https://github.com/fozziethebeat/S-Space` (online; accessed: 26.05.2013).