САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра системного программирования

ИНТЕРАКТИВНАЯ ЯЗЫКОНЕЗАВИСИМАЯ СИСТЕМА ПОИСКА ШАБЛОНОВ И ДУБЛИКАТОВ В ПРОГРАММНОМ КОДЕ

Дипломная работа студента 545 группы
Куделевского Евгения Валерьевича

Научный руководитель	……	к.ф.-м.н. М. А. Мосиенко
Рецензент	……	к.ф-м.н, доцент Д. Ю. Булычев
“Допустить к защите” заведующий кафедрой,	……	д.ф.-м.н., профессор А. Н. Терехов

Санкт-Петербург
2011

SAINT PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics faculty

Software Engineering Chair

INTERACTIVE PROGRAMMING LANGUAGE INDEPENDENT SYSTEM FOR DETECTION OF PATTERNS AND DUPLICATES IN A SOURCE CODE

Graduate paper by 545 group's student
Kudelevskiy Evgeny Valeryevich

Scientific advisor	……	M. A. Mossienko
Reviewer	……	Ass. Professor D. Yu. Bulychev
"Approved by" Head of Department,	……	Professor A. N. Terekhov

Saint Petersburg
2011

Оглавление

Введение

Проблема поиска дубликатов [2] – одна из важнейших задач программной инженерии. Нахождение и удаление повторяющегося кода позволяет значительно упростить сопровождение продукта и его дальнейшую разработку. Кроме того, данная задача имеет и другие применения, например извлечение знаний об устройстве программной системы.

Чтобы формализовать задачу поиска дубликатов, необходимо определить эквивалентность на множестве фрагментов кода. Один из самых простых случаев такой эквивалентности – идентичность на уровне лексического представления без учета пробелов и комментариев. Следует заметить, что больший интерес представляет поиск именно «нечетких» дубликатов, то есть фрагментов, схожих по структуре, но различающихся в деталях, например в именах переменных или литералах.

На сегодняшний день существует большое количество средств поиска дубликатов. Большинство из них ориентировано на конкретные языки программирования и не предоставляет простого механизма расширения. Средства поиска имеются для многих широко используемых языков, таких как C [6, 14], Java [6, 16], PHP [15] и т.п. Следует заметить, что поисковая система может быть реализована на основе IDE, которая имеет в своем составе лексические и синтаксические анализаторы многих языков программирования. В этом случае возможность поиска в коде на любом из них была бы крайне полезной. С другой стороны, важно, чтобы поиск мог учитывать простейшие семантические особенности конкретного языка. Ниже приведено несколько примеров таких особенностей:

- Приоритеты операций. Например, выражения *a && b || c && d* можно считать эквивалентным выражению *(a && b) || (c && d)*, но не выражению *a && (b || c) && d*.

- Порядок следования элементов. В некоторых случаях порядок следования синтаксических единиц не имеет значения, например в случае списка интерфейсов, которые реализует Java класс, или в случае списка HTML/XML атрибутов.

С задачей поиска дубликатов связана другая, более общая - задача поиска по шаблону в программном коде. Поиск может быть как точным, когда ищутся фрагменты кода, полностью совпадающие с заданным шаблоном, так и неточным, когда поиск ведется по каким-либо критериям, приблизительно описывающим искомый фрагмент. Критерии могут быть различными, начиная от регулярных выражений и заканчивая скриптами-предикатами.

Некоторые применения поиска по шаблону:

- Извлечение знаний из кода (reverse engineering) [10]
- Инспекция кода (например, поиск нежелательных фрагментов «антипаттернов»)
- В некоторых случаях поиск является первой частью более сложного процесса изменения кода, например замены всех фрагментов, соответствующих заданному шаблону, на другой шаблон.

Поиск дубликатов в IntelliJ IDEA

Среда разработки IntelliJ IDEA имеет в своем составе инструмент поиска дубликатов в программном коде. На момент начала выполнения данной дипломной работы в нем имелась поддержка языков Java и CSS. В случае Java под дубликатами понимаются фрагменты, идентичные по структуре и, возможно, различающиеся в литералах, идентификаторах и т.п. Поисковый инструмент имеет следующие возможности для настройки:

- Анонимизация некоторых лексических единиц. Пользователь может выбрать, различать ли при поиске имена переменных, названия типов и

т.п.

- Размер дубликатов. В процессе поиска для каждой единицы языка, присутствующей в коде, вычисляется функция стоимости. При этом стоимость сущности-предка всегда не меньше суммы стоимостей вложенных в нее сущностей. Такая функция не зависит от расположения элементов и отражает только синтаксическую сложность кода. Пользователь может задать минимальную стоимость дубликатов, которые требуется найти.

- Анонимизация выражений. Возможно настроить поиск так, чтобы он не различал между собой выражения стоимости меньше заданной.

Плагин Structural Search & Replace

В составе IntelliJ IDEA также имеется средство интерактивного поиска по шаблону Structural Search & Replace [8]. Шаблон в нем описывается на языке программирования, в котором предполагается производить поиск, расширенном дополнительными конструкциями – шаблонными переменными. Шаблонные переменные – это параметры, вместо которых могут подставляться различные сущности языка, такие как выражения, идентификаторы и т.п. При этом на значения переменных и на весь фрагмент в целом могут накладываться дополнительные ограничения. Также каждый найденный фрагмент возможно заменить на другой шаблон, в описании которого могут присутствовать те же переменные, что и в искомом шаблоне. Изначально инструмент позволял производить поиск в коде на языках Java и XML. Также в 2010 году была добавлена поддержка языка JavaScript [17].

Цели и задачи

Цель данной дипломной работы – разработка и внедрение программного инструмента поиска, удовлетворяющего следующим критериям:

1. Языконезависимость. Ядро поиска должно быть не привязано к конкретному языку программирования, а объем работ, необходимых для поддерки нового языка в системе должен быть минимизирован.
2. Расширяемость. Должна иметься возможность учета при поиске особенностей конкретного языка программирования.

Для достижения цели дипломной работы были поставлены следующие задачи:

1. Изучить научные публикации, связанные с темой работы.
2. Изучить механизмы работы существующих инструментов поиска шаблонов и дубликатов, являющихся частью IntelliJ IDEA.
3. Расширить возможности среды разработки IntelliJ IDEA, которые она предоставляет для поиска шаблонов и дубликатов, а именно реализовать механизм поиска, не привязанный к конкретному языку программирования, позволяющий легко внедрять поддержку нового языка, поддерживаемого IDE.
4. Разработать API для добавления в систему поддержки нового языка.
5. Реализовать в системе поддержку многих языков программирования.
6. Протестировать полученный инструмент на исходном коде реальных программных проектов.

1. Обзор существующих средств и подходов

Существует 3 основных подхода к решению задачи поиска дубликатов: текстовый, лексический и синтаксический.

1.1. Текстовый подход

Текстовый подход использует для поиска текстовое представление исходного кода. Впервые этот подход применил J. Johnson в 1993 году [7]. В его методе сначала вычислялась хэш-функция для всех фрагментов кода, содержащих фиксированное количество строк, а затем сравнивались фрагменты с одинаковыми хэш-кодами. Эта техника применялась циклически для разного числа строк.

1.2. Лексический подход

Первой этот подход применила B. Baker [1]. В ее системе исходный код программы разбивался на лексемы, а затем лексемы, представляющие собой идентификаторы и литералы, заменялись специальными лексемами-параметрами. После этого по полученной последовательности строилось суффиксное дерево.

Суффиксное дерево – это представление строки в виде дерева, где каждый суффикс соответствует некоторому пути от корня к листу. При этом если 2 суффикса имеют общий префикс, то пути, соответствующие им, имеют общую дугу. Суффиксное дерево требует для хранения линейное от длины строки количество памяти. Также существуют алгоритмы его построения за линейное время [3, 4]. Пример дерева показан на рис. 1.

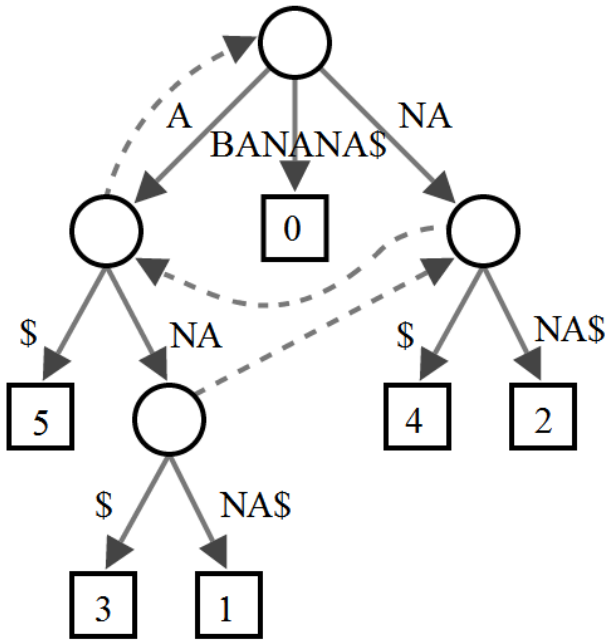

Рис. 1 Суффиксное дерево для строки BANANA. Символ $ обозначает конец строки, метки на ребрах – подстрок, метка на каждом листе – позиция суффикса, соответствующего пути от корня к этому листу.

Чтобы найти дубликаты с помощью суффиксного дерева, нужно взять все его внутренние узлы и найти все листья соответствующих поддеревьев. Например, поддерево, порожденное узлом, соответствующим подстроке ANA, имеет 2 листа с метками 1 и 3. Это означает, что в исходной строке эта подстрока встречается 2 раза: в позициях 1 и 3. Таким образом, когда дерево уже построено, дубликаты могут быть найдены за линейное время. Недостаток такой структуры – это большой объем занимаемой памяти. Для построения дерева за линейное время требуется хранить таблицу «узел · символ → исходящее ребро», поэтому объем памяти пропорционален z · n, где z – размер алфавита, а n – размер строки. В случае, когда алфавитом является множество всевозможных лексем языка, встречающихся в программе, z может быть довольно большим.

Существуют методы поиска дубликатов, использующие структуру, являющуюся легковесной альтернативой суффиксному дереву – суффиксный массив [5]. Суффиксный массив – это массив суффиксов строки,

отсортированный в лексикографическом порядке. Пример суффиксного массива показан в таб. 1.

Суффикс	Индекс в строке
$	6
A$	5
ANA$	3
ANANA$	1
BANANA$	0
NA$	4
NANA$	2

Таб. 1 Суффиксный массив для строки “BANANA”. Символ $ обозначает конец строки.

Когда суффиксный массив построен, с его помощью можно находить дубликаты за линейное время. При этом он требует меньше памяти, чем суффиксное дерево: каждый суффикс можно хранить как его позицию в строке, а количество суффиксов совпадает с размером строки. Известны алгоритмы построения массива, как за время $O(n \cdot \log n)$ [13], так и за линейное время [9].

1.3. Синтаксический подход

Синтаксический подход заключается в поиске схожих поддеревьев в абстрактном синтаксическом дереве программы. Впервые этот подход применил I. D. Baxter в системе CloneDr [6]. В ней для каждого поддерева вычислялась хэш-функция, а поддеревья с одинаковыми хэш-кодами затем сравнивались друг с другом. Позже такой метод был применен и в других системах [12].

Главное преимущество синтаксического подхода – гибкость. На уровне синтаксического дерева мы располагаем всей информацией о синтаксисе программы, можем различать языковые конструкции и учитывать их особенности. Ниже приведено несколько примеров, когда это может быть

полезно:

1. Алгоритм поиска может игнорировать некоторые промежуточные узлы, наличие или отсутствие которых не изменяет семантику программы:

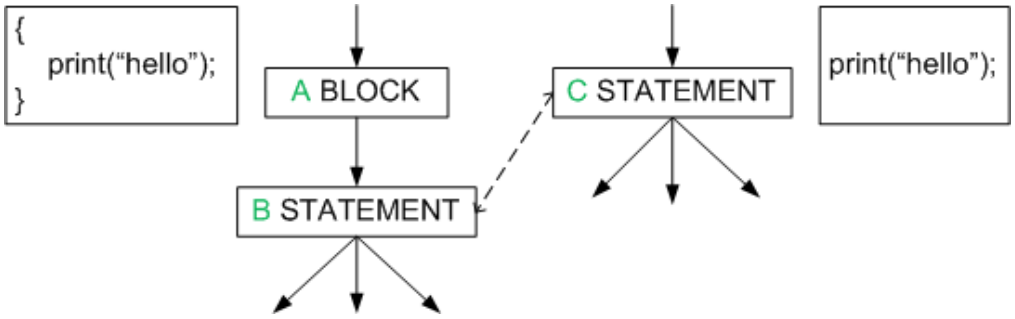

Рис. 2 Если поддеревья B и C – эквивалентны, деревья A и C также можно считать эквивалентными

2. В зависимости от типа узла, алгоритм может учитывать или не учитывать порядок расположения его потомков. Например, эту возможность можно применить для поиска дубликатов в XML коде, когда порядок расположения атрибутов не важен.

3. Возможно учитывать опциональный синтаксис. Простой пример: в некоторых языках при объявлении переменной не обязательно явно указывать ее тип. Алгоритм может не различать объявления переменных с типом и без.

Стоит отметить, что описанная выше функциональность может быть реализована и с использованием лексического подхода, но для этого требуется построение некоторого нормализованного лексического представления кода на основе синтаксического дерева.

1.4. Сравнение подходов

Недостаток текстового подхода – зависимость от пробелов и переводов строк. С другой стороны, его достоинство в том, что он не требует ни лексического, ни синтаксического анализа кода, то есть полностью независим от языка, а также

имеет наибольшую производительность.

Лексический подход, имеет высокую скорость работы, но при его использовании сложно учитывать особенности конкретного языка. Также при использовании лексического метода в чистом виде не учитываются границы синтаксических конструкций, например, может допускаться такой фрагмент:

```
    final int factor = 30;
    return p * factor;
}
return p + 1;
```

В 2006 году R. Koschke предложил метод, лишенный этого недостатка. Он предложил строить лексическую последовательность не при помощи лексического анализатора, а из синтаксического дерева, добавляя в нее специальные скобки, обозначающие границы синтаксических единиц. Применяя этот метод, также можно учитывать особенности конкретного языка при помощи нормализации лексического представления. С другой стороны, при таком подходе сильно возрастает размер лексической последовательности и увеличивается время ее построения.

Синтаксический подход, по сравнению с лексическим, имеет более низкую скорость работы, так как требуется полностью обходить дерево. Также при хэшировании блока операторов, требуется вычислить хэш-коды всех его подблоков, так как они могут являться дубликатами, а всего подблоков – порядка m^2, где m – число операторов в блоке. С другой стороны, синтаксический подход, как уже говорилось, позволяет легко учитывать различные особенности языка.

1.5. Связь с задачей поиска по шаблону

2 последних подхода могут быть применены также и к решению задачи поиска по шаблону, описанному с использованием шаблонных переменных. При

использовании лексического подхода, по шаблону может быть сгенерирована последовательность лексем. Тогда задача сводится к поиску подстроки в строке, в которой символами являются лексемы. В этом случае в качестве значений шаблонных переменных могут выступать отдельные лексемы или их последовательности. При использовании синтаксического подхода происходит разбор исходной программы и шаблона, а затем возникает задача поиска поддерева (или последовательности поддеревьев) в синтаксическом дереве. Здесь значениями шаблонных переменных будут являться поддеревья синтаксического дерева.

1.6. Различные средства поиска в программном коде

Существуют различные средства автоматизированного поиска в тексте. Самое распространенное из них – утилита GREP [18], с помощью которой можно находить фрагменты, соответствующие заданному регулярному выражению. Следует заметить, что использование подобных инструментов для поиска в программном коде имеет ряд недостатков. Во-первых, иногда регулярное выражение оказывается очень сложным, и при его составлении легко ошибиться. Во-вторых, в некоторых случаях вообще невозможно описать шаблон с помощью регулярного выражения. Например, пусть мы хотим найти последовательность двух подряд идущих операторов for и while с произвольными условиями, находящиеся на одном и том же уровне вложенности. При этом между ними могут присутствовать другие операторы. Такой шаблон невозможно описать с помощью регулярных выражений, так как они ничего не знают о синтаксисе программы и, в частности, об уровнях вложенности.

Этих недостатков лишен подход, используемый в утилите Structural Search & Replace, которая уже упоминалась во введении. Идея такого способа описания шаблонов не нова, хоть и не имеет широкого распространения. Подобный

подход использовался для реализации инструмента поиска еще в 1994 году [11].

2. Описание решения

2.1. Поиск дубликатов

Программное средство поиска дубликатов, разработанное в рамках данной дипломной работы, поддерживает все возможности настройки, которые имелись ранее для языка Java (рис. 3), а именно:

- Анонимизация имен переменных, методов, названий типов и т.п.
- Минимальный размер дубликатов.
- Анонимизация синтаксических конструкций стоимости меньше заданной.

Кроме того, поисковой инструмент обладает поддержкой многих языков программирования, а также имеется простой механизм добавления нового языка, который подробно описан в разделе 4. На данный момент поиск дубликатов можно производить в коде на языках Groovy, PHP, JavaScript и HTML.

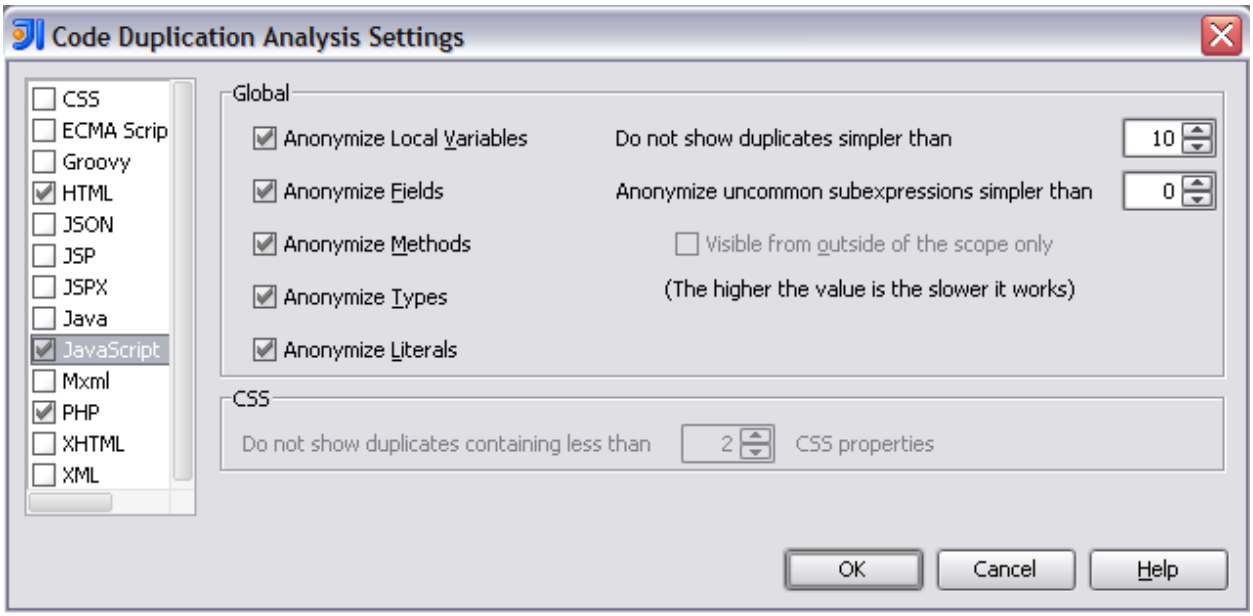

Рис. 3 Интерфейс инструмента поиска дубликатов в IntelliJ IDEA

2.2. Поиск по шаблону

Инструмент поиска по шаблону, разработанный в рамках данной дипломной работы на основе существующего средства Structural Search & Replace, поддерживает все функции, которые имелись ранее для языков Java, JavaScript и XML. Он предоставляет следующие возможности:

- Ввод запроса происходит на языке программирования, в котором предполагается производить поиск, при этом можно использовать шаблонные переменные.
- В поле ввода запроса обеспечивается подсветка синтаксиса (рис. 4).
- Как на значения шаблонных переменных, так и на фрагмент в целом, можно накладывать ограничения в виде регулярных выражений и скриптов-предикатов, описанных на языке Groovy.
- Интерактивность. Пользователь может просматривать текущие результаты, не дожидаясь завершения процесса поиска.
- Имеется поддержка мультипеременных - переменных, в качестве значений которых может выступать не одна, а несколько подряд идущих синтаксических единиц одного типа (например, параметров функции, как в шаблоне "`function f($params$) {}`"). При этом пользователь может указать минимальное и максимальное количество допустимых вхождений. Между вхождениями могут находиться разделители.
- Возможность замены на другой шаблон.
- Поддержка многих языков. На данный момент поиск по шаблону можно производить в коде на языках Groovy, PHP и CSS.
- Инструмент имеет простой механизм добавления поддержки новых языков, который подробно описан в разделе 4.

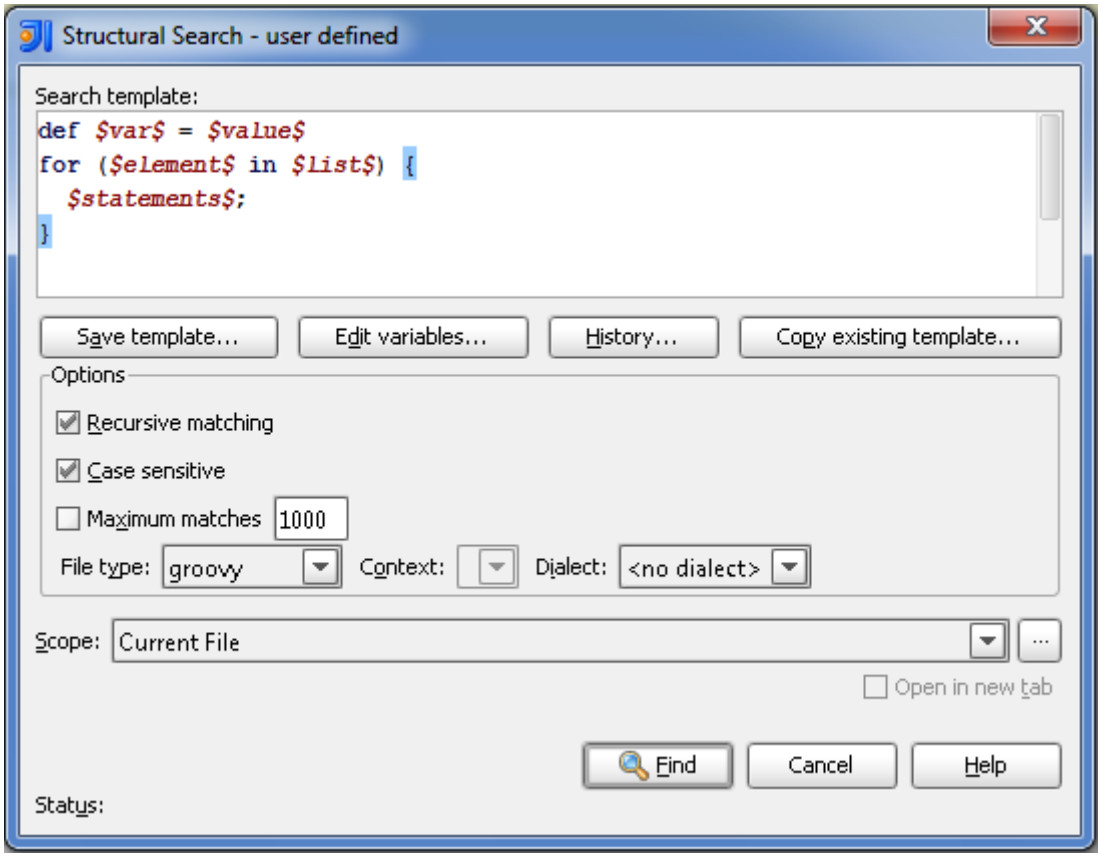

Рис. 4 Интерфейс инструмента Structural Search & Replace. Шаблон для поиска в Groovy-коде

3. Реализация

3.1. Поиск дубликатов

Алгоритм работы поиска дубликатов основан на синтаксическом подходе, поскольку с его помощью проще учитывать особенности конкретного языка программирования. Сначала происходит полный обход дерева программы, вычисление хэш-функции и функции стоимости для каждого поддererева, а затем поддеревья с одинаковыми хэш-кодами и стоимостью больше минимальной сравниваются.

3.1.1. Анонимизация

Если используется анонимизация имен переменных, типов и т.п., хэш-коды соответствующих синтаксических конструкций принимаются равными нулю, а алгоритм сравнения считает узлы с одинаковыми ролями (например, имя переменной) эквивалентными. Аналогичная стратегия применяется в случае, когда используется анонимизация простых конструкций, стоимость которых меньше указанной пользователем. Следует заметить, что для работы алгоритма необходим механизм, для каждого языка умеющий определять роль узла. Этот механизм будет описан в разделе 4.

3.1.2. Подблоки

Описанный выше метод может находить дубликаты, соответствующие поддеревьям синтаксического дерева, но во многих случаях дубликаты соответствуют последовательностям из нескольких поддеревьев. Например, рассмотрим следующий фрагмент кода:

```
function f($a, $b) {
  if ($a < 1 || $b < 1) {
    return false;
  }
  $a = adjust($a);
  $b = adjust($b);
  if ($a < 1 || $b < 1) {
    return false;
  }
```

```
  return true;
}

function g() {
  $a = read();
  $b = read();
  if ($a < 1 || $b < 1) {
    return false;
  }
  $a = adjust($a);
  $b = adjust($b);
  if ($a < 1 || $b < 1) {
    return false;
  }
  return $a < max && $b < max;
}
```

Дубликаты выделены жирным шрифтом. Понятно, что выделенные фрагменты соответствуют последовательностям поддеревьев, корни которых имеют тип STATEMENT. Чтобы иметь возможность находить такие дубликаты, нужно хэшировать не только поддеревья, но и их последовательности. Следует отметить, что сложность такой операции для одного узла – $O(m^2)$, где m – количество потомков, поэтому будем производить такое хэширование только для отдельных типов узлов, представляющих собой блоки кода. Как и в случае с анонимизацией, для каждого языка нужно уметь определять, является ли узел блоком кода. Подробнее об этом рассказывается в разделе 4.

3.2. Поиск по шаблону

Инструмент поиска по шаблону, созданный в рамках данной дипломной работы, разработан на основе существующего плагина Structural Search & Replace (SSR). Алгоритм работы поиска также основан на синтаксическом подходе. Сначала происходит разбор шаблона и строится дерево, которое имеет вид, показанный на рис. 5.

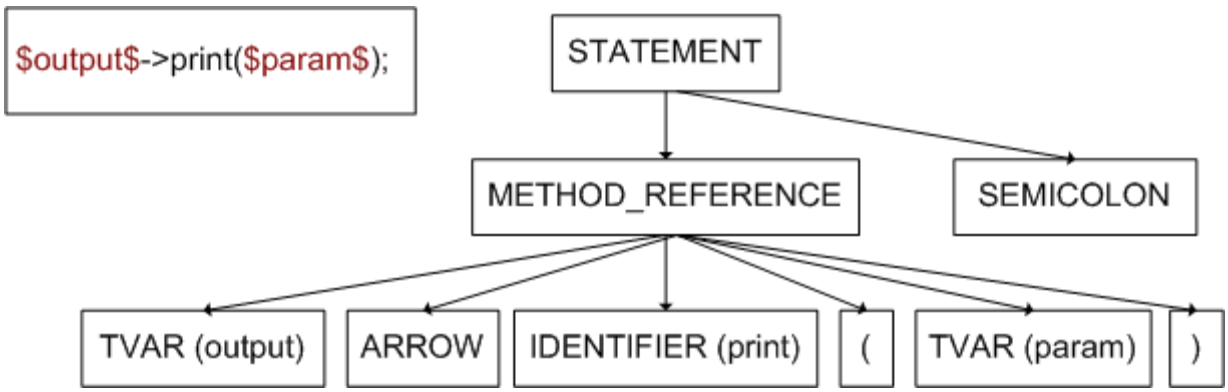

Рис. 5 Синтаксическое дерево для шаблона “`$output$->print($param$)`”

Будем считать, что узлы, соответствующие шаблонным переменным, имеют некоторый тип **TVAR**. Способ построения такого дерева подробно описан в разделе 3.2.2 «Синтаксический разбор шаблона». После того, как дерево построено, происходит обход дерева программы в глубину и поиск всех поддеревьев, соответствующих дереву шаблона.

3.2.1. Алгоритм сопоставления

Далее будет изложен упрощенный алгоритм сопоставления последовательности узлов дерева с шаблонной последовательностью. Алгоритм определяет, соответствует ли заданная последовательность шаблонной и, если так, сопоставляет каждой шаблонной переменной ее значение.

1. Узлы сравниваются последовательно, при этом игнорируются пробелы и комментарии.

2. Если один из узлов имеет тип TVAR, то вычисляется имя шаблонной переменной и второй узел запоминается в качестве значения переменной с этим именем. Предварительно проверяется, удовлетворяет ли данный узел ограничениям, которые накладываются на значение переменной. Если у переменной с данным именем уже существует значение, то происходит сравнение нового значения со старым.

3. Когда оба узла имеют типы отличные от TVAR, происходит сравнение

их типов, а затем алгоритм рекурсивно запускается для последовательностей их непосредственных потомков.

3.2.2. Синтаксический разбор шаблона

Шаблон в SSR не является обычным фрагментом кода из-за присутствия в нем шаблонных переменных. Чтобы произвести синтаксический разбор шаблона, он сначала преобразуется в обычный код, который затем передается синтаксическому анализатору. Для этого выполняется замена всех шаблонных переменных на идентификаторы языка программирования, которые получаются путем добавления некоторого уникального префикса к именам переменных. Уникальность префикса необходима для того, чтобы полученные идентификаторы не встречались в коде программы, тогда можно будет легко отличить шаблонную переменную от обычного идентификатора.

Важно заметить, что в некоторых случаях нельзя использовать один и тот же префикс для всех шаблонных переменных. Например, в языке PHP имя переменной должно начинаться с символа ‘$’, а имя метода наоборот не может начинаться с этого символа и, более того, вообще не может содержать этот символ, так как его присутствие допускается только в начале идентификатора. В таком случае, если, например, использовать префикс “$_____” для разбора шаблона “`function $name$() {}`”, то разбор произойдет некорректно. С другой стороны, если для разбора шаблона “`$var_name$->run();`” использовать префикс, который не начинается с ‘$’, то синтаксический анализатор языка PHP посчитает идентификатор константой, а поскольку константы не могут иметь методов, при разборе также возникнет синтаксическая ошибка. Эта проблема решается следующим образом: для каждого языка задан некоторый набор префиксов, а для каждой переменной происходит выбор подходящего. Последовательность префиксов считается подходящей, если в процессе синтаксического анализа не возникает ошибок.

Помимо наличия шаблонных переменных, существует другая проблема –

необходимость задания контекста разбора. Рассмотрим шаблон, представляющий собой вызов функции: “`func($param$);`”. В случае языка Java, разбор этого шаблона должен происходить в контексте тела метода. Для решения этой проблемы для каждого языка программирования в системе поиска задан контекст, в котором должен происходить разбор. Например, для языка Java строка контекста выглядит следующим образом:

```
class C {
    void f() {
        $PLACE_HOLDER$
    }
}
```

А для языка PHP:

```
<?php $PLACE_HOLDER$
```

Пример того, как происходит разбор шаблона, показан на рис.6.

Итак, после разбора будет построено синтаксическое дерево, но оно не будет иметь узлов типа TVAR, поскольку синтаксический анализатор языка ничего не знает о шаблонных переменных. Узлами типа TVAR будем считать все поддеревья, соответствующие элементу кода, который начинается с одного из допустимых префиксов, а оставшаяся его часть является Java-идентификатором. Этот идентификатор будем считать именем шаблонной переменной. Пример показан на рис.7.

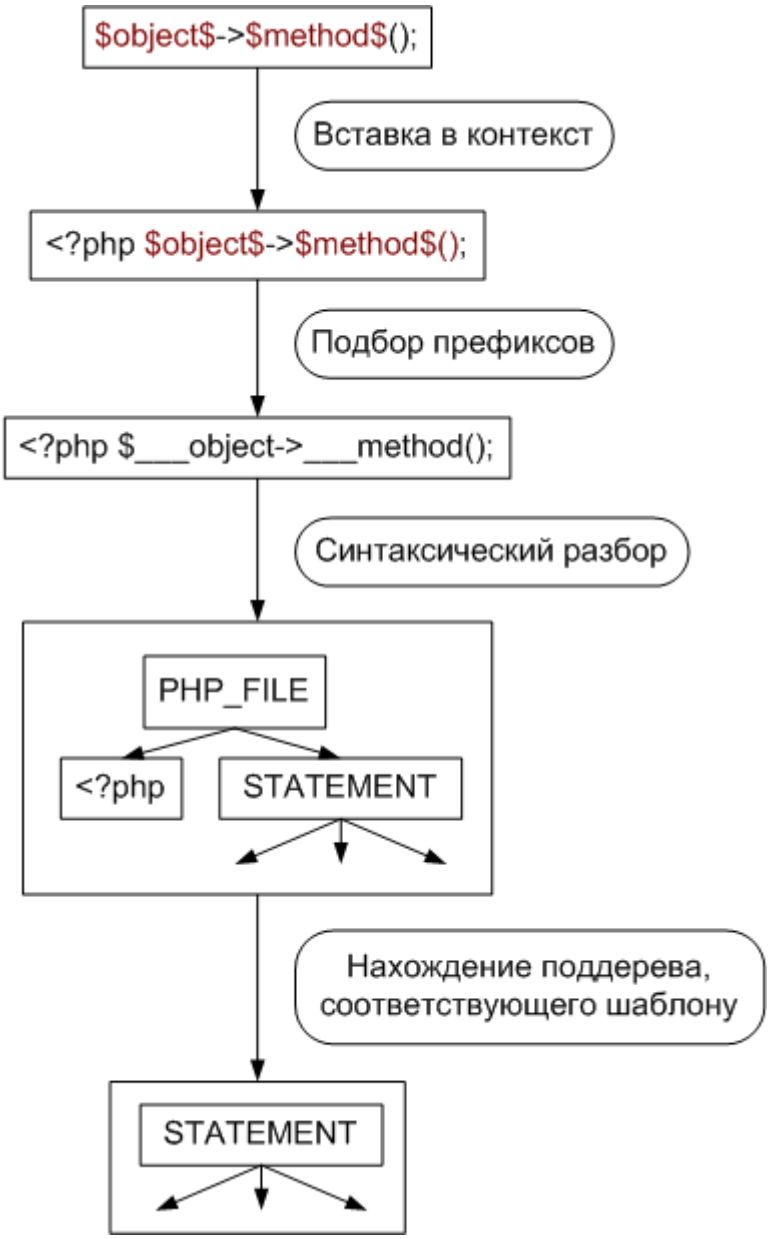

Рис. 6 Разбор шаблона $object$->$method$() для языка PHP.

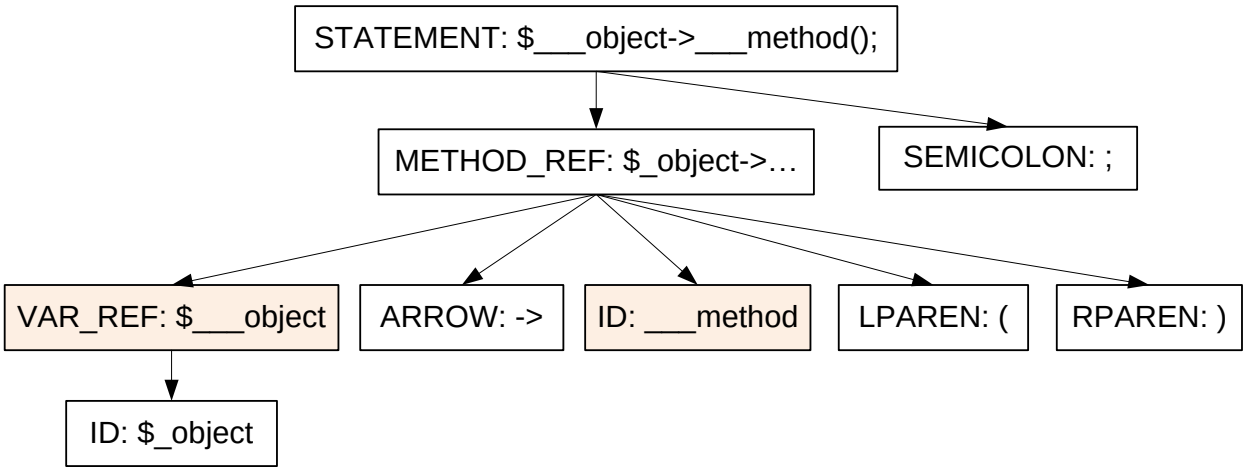

Рис. 7 Дерево, полученное в результате разбора шаблона $object$->$method$(). Для каждого узла через двоеточие указан фрагмент текста программы, которому он соответствует. Выделены узлы типа TVAR.

3.2.3. Мультипеременные

Как уже говорилось, в поисковой системе имеется поддержка мультипеременных, то есть переменных, которые обозначают не одну синтаксическую единицу, а последовательность синтаксических единиц одного типа. При этом пользователь может указать минимальное и максимальное количество допустимых вхождений. Если в процессе работы алгоритма сопоставления на шаге 2 встретилась мультипеременная, то дальше он работает по следующей схеме:

1. Происходит итерация по узлам исходной последовательности. При этом пропускаются пробелы, комментарии и разделители, допустимые между вхождениями. Для каждого узла проверяются ограничения на значения переменной, и, если он им соответствует, узел добавляется в список вхождений.

2. Пусть на какой-то итерации количество вхождений в списке достигло минимального. После этого возникает неоднозначность: с одной стороны мы уже можем взять в качестве значения мультипеременной полученную последовательность, но с другой стороны, если максимальное число вхождений для переменной больше минимального, в

последовательность можно включить еще больше узлов. Будем проверять все варианты поочередно, то есть после добавления очередного узла в список вхождений будем запускать алгоритм сопоставления для оставшихся подпоследовательностей. Если в какой-то момент сопоставление прошло успешно – принимаем полученную последовательность в качестве значения переменной.

4. Возможности расширения

Разработанная система поиска шаблонов и дубликатов имеет простые механизмы для расширения и поддержки новых языков. Ниже будет описан предоставляемый API.

4.1. Поиск дубликатов

Для поддержки функции поиска дубликатов для нового языка программирования необходимо создать класс, наследующий DuplicatesProfileBase. Используя этот механизм расширения, можно указать, какие узлы синтаксического дерева являются литералами, а какие – именами переменных, что сделает возможной их анонимизацию. Также с помощью него можно указать, какие узлы являются блоками кода, чтобы поисковая система рассматривала подблоки в качестве потенциальных дубликатов. Ниже в качестве примера приведен исходный код класса GroovyDuplicatesProfile, добавляющего поддержку языка Groovy:

```
public class GroovyDuplicatesProfile
             extends DuplicatesProfileBase {

  public boolean isMyLanguage(Language language) {
    return language == GroovyLanguage.INSTANCE;
  }

  public Role getRole(@NotNull PsiElement e) {
    PsiElement p = e.getParent();

    if (p instanceof GrVariable &&
        ((GrVariable)p).getNameElement() == e) {
      return Role.VARIABLE_NAME;
    }
    else if (p instanceof GrMethod &&
             ((GrMethod)p).getNameElement() == e) {
      return Role.FUNCTION_NAME;
    }
    else if (p instanceof GrBlockStatement) {
      return Role.CODE_BLOCK;
    }

    return null;
  }
```

```
  public int getNodeCost(PsiElement e) {
    if (e instanceof GrStatement) {
      return 2;
    }
    return 0;
  }

  public TokenSet getLiterals() {
    return GrTokenSets.LITERALS;
  }
}
```

4.2. Поиск по шаблону

Для поддержки функции Structural Search & Replace для нового языка программирования необходимо создать класс, наследующий StructuralSearchProfileBase и реализовать несколько методов. В качестве примера приведен исходный код класса PhpStructuralSearchProfile, добавляющего поддержку языка PHP:

```
public class PhpStructuralSearchProfile
       extends StructuralSearchProfileBase {

  protected String[] getVarPrefixes() {
    return new String[]{"aaaaaaaaa", "$______"};
  }

  protected LanguageFileType getFileType() {
    return PhpFileType.INSTANCE;
  }

  public String getContext(@NotNull String pattern) {
    return "<?php $$PATTERN_PLACEHOLDER$$";
  }

  protected TokenSet getVariableDelimiters() {
    return TokenSet.create(PhpTokenTypes.opCOMMA,
                           PhpTokenTypes.opSEMICOLON)
  }
}
```

Метод **getVarPrefixes** возвращает массив префиксов для шаблонных переменных.

Метод **getFileType** возвращает тип файла, соответствующий языку, в котором

предполагается производить поиск.

Метод **getContext** возвращает строку контекста.

Метод **getVariableDelimiters** возвращает множество лексем, которые могут выступать в качестве разделителей значений мультипеременной. В случае PHP это ‘,’ и ‘;’.

4.3. Описание эквивалентности

Для описания эквивалентности синтаксических конструкций какого-либо языка программирования можно использовать базовый класс EquivalenceDescriptorProvider. Использование этого механизма необязательно. По-умолчанию для поиска шаблонов и дубликатов используется некоторая стандартная эквивалентность, но, как уже говорилось в разделе 1.3, в некоторых случаях эквивалентность может принимать во внимание особенности конкретного языка. С помощью этого механизма для узла синтаксического дерева можно определить некий дескриптор, описывающий характеристики его потомков. Например, можно указать, что порядок следования потомков не имеет значения для данного элемента (как в случае со списком HTML атрибутов), либо что какие-то элементы следует пропустить.

5. Апробация

Программный инструмент, разработанный в рамках данной дипломной работы, был протестирован на исходном коде проекта MediaWiki [19], представляющего собой программный механизм для веб-сайтов, работающих по технологии «вики». Проект содержит код на языках PHP, JavaScript и HTML и XML. Объем исходного кода – 47 Мб.

5.1. Поиск дубликатов

Время работы поиска: 135 сек. Было найдено более 300 групп дубликатов, в каждой из которых находится от 2 до 11 фрагментов. Распределения дубликатов по языкам и по количеству строк кода приведены в таблице 2.

	Общее кол-во	> 40 строк	20-40 строк	10-20 строк	< 10 строк
Все языки	339	5	14	62	258
PHP	259	3	8	54	193
JavaScript	53	0	0	2	51
HTML	14	1	2	2	10
XML	13	1	4	4	4

Таб. 2 Распределение дубликатов по языкам и по количеству строк кода

5.2. Поиск по шаблону

Инструмент поиска был протестирован с использованием различных шаблонов на языке PHP. Процесс поиска занимал от 3 до 80 сек. в зависимости от сложности шаблона. Далее приведены результаты тестирования инструмента для нескольких шаблонов, которые являются примерами «антипаттернов»:

1. Вызов функции в заголовке цикла for. В следующем примере функция count будет вызываться перед каждой новой итерацией, что ухудшит производительность кода:

```
for ($i = 0; $i < count($array); $i++) { ... }
```

Описание шаблона для поиска таких фрагментов выглядит следующим образом:

```
for ($i$ = $value$; $i$ < $function$($params$); $inc$)
```

2. Обращение к элементу массива по индексу без использования кавычек. В случае, когда индекс не написан заглавными буквами (как в случае с константами), это является возможной ошибкой, как например `$array[user]` скорее всего следует заменить на `$array['user']`. Описание шаблона выглядит следующим образом: `$array$[$index$]`. При этом на переменную index накладывается ограничение в виде регулярного выражения [^$'"]\w*[a-z]\w*.

3. Функции, содержащие 7 и более аргументов. Шаблон описывается в виде `function $name$($params$)`. При этом минимальное число вхождений для мультипеременной params устанавливается равным 7.

Результаты тестирования приведены в таблице 3.

	Шаблон 1	Шаблон 2	Шаблон 3
Время работы поиска	7 сек.	76 сек.	12 сек.
Кол-во соответствий	23	0	3

Таб. 3 Показатели тестирования инструмента поиска на различных шаблонах

6. Заключение

В рамках данной дипломной работы на основе существующих средств IntelliJ IDEA был разработан и внедрен языконезависимый инструмент поиска шаблонов и дубликатов в программном коде, имеющий простой механизм расширения для добавления поддержки новых языков. Также разработанный инструмент позволяет учитывать семантические особенности конкретного языка программирования. С использованием созданного механизма расширения была добавлена поддержка многих существующих языков программирования и было проведено тестирование системы на исходном коде проекта MediaWiki. Дальнейшие развитие возможно в направлениях увеличения быстродействия поиска, поддержки более сложных средств описания шаблонов и интеграции со средствами генерации кода.

Список литературы

1. B. Baker, “On Finding Duplication and Near-Duplication in Large Software Systems”. - // 2nd Working Conference on Reverse Engineering, 1995, pp. 86-95. - : WCRE

2. Duplicate code [Электронный ресурс] URL: http://en.wikipedia.org/wiki/Duplicate_code

3. E. M. McCreight, “A Space-Economical Suffix Tree Construction Algorithm”. - // Journal of the ACM 23 (2), 1976, p.262-272. - : Association for Computing Machinery

4. E. Ukkonen, “On-line construction of suffix trees”. - // Algorithmica 14(3), 1995, p. 249-260. - : Springer

5. H. A. Basit, S. J. Puglisi, W. F. Smyth, A. Turpin, S. Jarzabek, “Efficient token based clone detection with flexible tokenization”. - // The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, 2007, p.513-516. - : Association for Computing Machinery

6. I. Baxter, A. Yahin, L. Moura and M. Anna, “Clone Detection Using Abstract Syntax Trees”. - // 14th International Conference on Software Maintenance, 1998, pp. 368-377. - : WCRE

7. J. Johnson, “Identifying Redundancy in Source Code Using Fingerprints”. - // 1993 Conference of the Centre for Advanced Studies on Collaborative Research, 1993, pp. 171–183. - : CASCON

8. Maxim Mossienko, “Structural Search and Replace”: What, Why and How-to [Электронный ресурс] URL: http://www.jetbrains.com/idea/docs/ssr.pdf

9. Pang Ko and Srinivas Aluru, "Space efficient linear time construction of suffix

arrays". - // Combinatorial Pattern Matching, 2003, p.203-210. - : LNCS 2676, Springer

10. Reverse Engineering [Электронный ресурс]
URL: http://en.wikipedia.org/wiki/Reverse_engineering

11. S. Paul and A. Prakash, "A Framework for Source Code Search Using Program Patterns," Software Engineering, vol. 20, 1994, pp. 463-475. - : IEEE

12. W. Evans, C. Fraser and M. Fei, “Clone Detection via Structural Abstraction”. - // Proceedings of the 14th Working, 2007 - : IEEE

13. Yuta Mori. DivSufSort, 2005. [Электронный ресурс]
URL: http://code.google.com/p/libdivsufsort/

14. Z. Li, S. Lu, S. Myagmar, and Y. Zhou, CP-Miner, “Finding Copy-Paste and Related Bugs in Large-ScaleSoftware Code”. - // IEEE Transactions on Software Engineering, 2006, pp. 176-192. - : IEEE

15. Инструмент PhpCPD [Электронный ресурс]
URL: https://github.com/sebastianbergmann/phpcpd/

16. Инструмент SimScan [Электронный ресурс] URL: http://www.blue-edge.bg/simscan/

17. Куделевский Е. В. “Поиск шаблонов в программном коде” [Электронный ресурс] URL: http://se.math.spbu.ru/SE/YearlyProjects/2010/445/Kudelevsky_report.pdf

18. Утилита GREP [Электронный ресурс]
URL: http://en.wikipedia.org/wiki/Grep

19. Фреймворк MediaWiki [Электронный ресурс]
URL: http://www.mediawiki.org