Санкт-Петербургский государственный университет

Кафедра компьютерных технологий и систем

Стрельчёнок Георгий Вадимович

Выпускная квалификационная работа бакалавра

Ступенчатый метод проверки исходного кода программы на плагиат

Направление 010300

Фундаментальные информатика и информационные технологии

Научный руководитель,

кандидат физ.-мат. наук,

доцент Лепихин Т. А.

Санкт-Петербург

2016

Оглавление

Введение

Ввиду массовой доступности информации, с относительно недавнего времени научные и образовательные сообщества стали активно развивать методы выявления заимствования одних идей другими, т. е. обратили свой взор к решению задачи проверки на плагиат, как в научных статьях, так и в выпускных и прочих работах. Особенно остро эта проблема стоит среди ученических и студенческих работ в различных образовательных учреждениях. А ведь именно в этих местах закладывается «фундамент» знаний и то, насколько он будет прочным, напрямую зависит от самостоятельности при выполнении учеником различных работ. В настоящее время существует достаточно много систем проверки текста на наличие плагиата, но практически все они ищут совпадения в обычных текстах. Все алгоритмы таких систем основаны на посимвольном сравнении. Обратим же внимание еще на одну проблему этой области – вопрос заимствования программного кода. Но как же выявлять заимствования кода? Эта задача усложняется возможностью перестановки блоков кода, переименованием переменных и функций, добавлением комментариев и ещё большим количеством изменений, которые можно делать, не влияя при этом на конечный результат работы программы.

Случаи заимствования в исходном коде приложений встречаются так же часто как в обычных текстах, статьях, рефератах и прочих письменных работах. Но при проверке заимствований частей программ необходимо применять совсем другие инструменты – отличные от тех, которые используются при выявлении плагиата в текстовых блоках.

Плагиат в программном коде можно определить как копирование исходного кода или отдельных его частей без внесения каких-либо изменений или с незначительной его обработкой. Следует немного детальнее рассмотреть какие варианты изменения исходного кода существуют.

Выделяют четыре типа дублирующих фрагментов кода [1]:

1 тип – два фрагмента кода являются полностью идентичными, могут изменяться комментарии, пробельные символы и отступы.

2 тип – структурно/синтаксически два фрагмента кода идентичны, за исключением имён идентификаторов, строковых литералов, типов переменных. Так же допускаются все изменения из пункта 1.

3 тип – один фрагмент кода был получен путём копирования другого фрагмента кода, но последующим добавлением и/или удалением операторов языка. При этом сохраняются все изменения свойственные типу 2.

4 тип – два фрагмента синтаксически реализованы по-разному, но выполняют одинаковые действия.

Подведя итог, можно сказать, что только 1 и 2 тип относится к незначительным изменениям. Третий тип распознать сложнее, обычно к этому типу можно отнести участки кода, которые специально пытались замаскировать, внося дополнительный «шум». Четвёртый тип больше свойственен промышленному коду и практически никогда не встречается в студенческих работах в силу сложности своей реализации. Обычно, для того чтобы реализовать участок кода иначе с точки зрения синтаксиса в нем необходимо хорошо разобраться. Поэтому этот тип видоизменения кода не обязателен для выявления.

Если студент, который заимствовал код, изменил названия переменных, констант, процедур, классов и др., будем считать, что работа полностью «списана», так как это не требует выдающихся знаний языка программирования и является плагиатом. Но если студентами реализован одинаковый метод решения поставленной задачи, это вовсе не означает, что работа является плагиатом, ведь, прежде всего, оценивается знание предмета, а не оригинальность решения.

Исходя из сказанного, система выявления плагиата не может быть полностью автоматической. Система должна замыкаться на преподавателя – лицо принимающее решение (ЛПР). Но в тоже время она должна максимально облегчить задачу поиска дубликатов кода, автоматизируя

процесс проверки.

Постановка задачи

Рассматриваемая в работе задача состоит в разработке и реализации гибкой системы проверки исходного кода на наличие плагиата, включающей в себя несколько этапов анализа текста для наиболее точного выявления факта копирования программ. Система должна удовлетворять следующим требованиям:

1. Комплексный анализ программ
    a. анализ комментариев;
    b. анализ характеристик программы;
    c. анализ исходного кода несколькими алгоритмами.
2. Универсальное представление исходного кода.
3. Возможность работы по базе данных (опционально).
4. Осуществление проверки «один-к-одному» и «один-ко-многим».
5. Интуитивно понятный интерфейс.
6. Лёгкая переносимость на различные платформы.

Исходя из представленных требований, был выбран язык разработки Java. Такое решение автоматически позволит разрабатывать кроссплатформенную версию программы. Кроме того, будет использован Java DataBase Connectivity – платформенно-независимый промышленный стандарт взаимодействия Java-приложений с различными СУБД, что позволит работать с различными базами данных без привязки к коду. По умолчанию программа будет работать с открытой кроссплатформенной системой управления базой данных – H2, которая будет работать во встраиваемом режиме и использовать файловую систему для хранения данных.

Одним из ключевых требований является комплексный анализ программы. Что подразумевается под этим термином? Во-первых, проверка исходного кода будет осуществляться в несколько этапов. Во-вторых, каждый из этапов будет в свою очередь состоять из ступеней, которые необходимо

проделать для удовлетворительного результата работы. Общие принципы функционирования системы можно представить в виде схемыError: Reference s o u r c e n o t f o u n d.

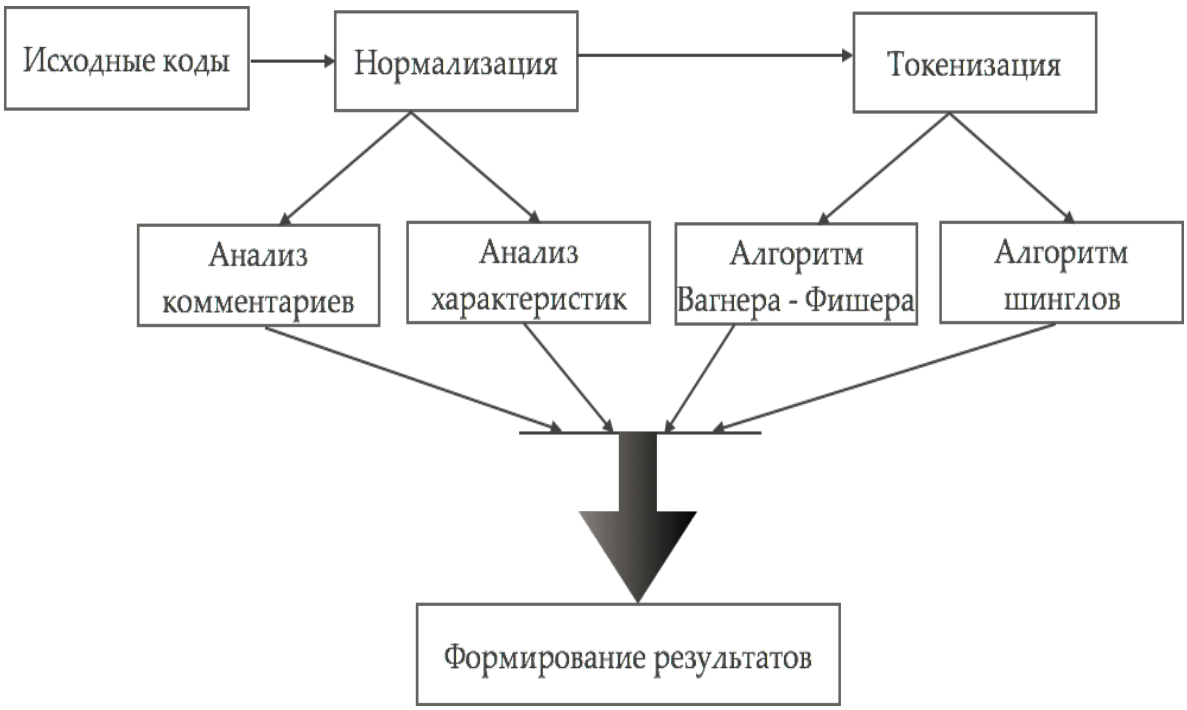


Рисунок 1. Этапы работы программы

Глава 1. Типы совпадений исходного кода

1.1. Совпадения типа I

Первый тип совпадений можно представить как полностью скопированный участок кода и, по факту, является полной копией оригинала. Тем не менее могут быть некоторые вариации в пробельных символах (пробелы, переносы строк, символы табуляции и т. д.), комментариях и/или разметке. Программы типа 1 широко известны как точные клоны. Рассмотрим следующий фрагмент кода:

```
if (a >= b) {
    c = d + b; // Comment1
    d = d + 1;}
else
    c = d - a; //Comment2
```

И сравним его с точной копией, представленной ниже:

```
if (a>=b) {
         // Comment1'
c=d+b;
        d=d+1;}
else  // Comment2'
c=d-a;
```

Мы видим, что эти два фрагмента являются текстуально похожими (построчно) после удаления пробелов и комментариев. Тем не менее, даже после удаления комментариев и пробелов, следующий фрагмент кода не похож на два предыдущих при построчном сравнении начиная с символа "{" и заканчивая "}". Но при этом, этот фрагмент также относится к типу I и является точной копией двух других.

```
if (a>=b)
{ // Comment1''
        c=d+b;
        d=d+1;
```

```
        }
else  // Comment2''
        c=d-a;
```

Типичные построчные методы обнаружения не смогут обнаружить такие участки идентичного кода.

1.2. Совпадения типа II

Второй тип совпадений представляет собой фрагмент кода, который совпадет с оригинальным за исключением различных имен идентификаторов определенных пользователем (имен переменных, констант, классов, методов и т. д.), типов, разметки, комментариев. Зарезервированные слова и синтаксическая структура остается такой же, как в оригинале. Рассмотрим следующий фрагмент кода:

```
if (a>= b) {
c = d + b; // Comment1
d = d + 1;} else
        c = d - a; //Comment2
```

Его клоном второго типа может быть фрагмент кода представленный ниже:

```
if (m >= n)
{ // Comment1'
    y = x + n;
    x = x + 5; //Comment3
    }
else
    y = x - m; //Comment2'
```

Мы видим, что два фрагмента кода изменены по своей форме, именам переменных и присваиваемым значениям. Однако, синтаксическая структура по-прежнему одинаковая в обоих сегментах.

1.3. Совпадения типа III

В этом типе заимствований, скопированный фрагмент дополнительно модифицирован с помощью изменения, добавления и/или удаления некоторого участка кода. Рассмотрим оригинальный фрагмент:

```
if (a >= b) {
        c = d + b; // Comment1
        d = d + 1;}
else
        c = d - a; //Comment2
```

Теперь, если добавить выражение e = 1, получим:

```
if (a >= b) {
        c = d + b; // Comment1
        e = 1; // Новое выражение
d = d + 1; }
else
c = d - a; //Comment2
```

Этот, скопированный фрагмент с добавленным выражением будет относиться к третьему типу. Другой пример клона третьего типа представлен ниже :

Фрагмент 1:

```
publicintgetSoLinger() throws SocketException{
    Object o = impl.getOption(SocketOptions.SO_LINGER);
    if (o instanceof Integer) {
        return((Integer) o).intValue();
    }
    else return -1;
}
```

Фрагмент 2:

```
public synchronized intgetSoTimeout()
throwsSocketException{
    Object o = impl.getOption(SocketOptions.SO_TIMEOUT);
```

```
    if (o instanceof Integer) {
        return((Integer) o).intValue();
    }
    else return -0;
}
```

Для более наглядного представления изменений можно сформировать таблицу, которая будет отображать различия между двумя этими участками кода:

Фрагмент 1	Фрагмент 2	Статус
<public>	<public>	Точная копия
ε	<synchronized>	Вставка
<int>	<int>	Точная копия
<getSoLinger>	<getSoTimeout>	Замена
<()throws... SocketOptions .>	<() throws... SocketOptions .>	Точная копия
<SO LINGER>	<SO TIMEOUT>	Замена
<) ; ...elsereturn>	<) ; ...elsereturn>	Точная копия
<-1 >	<-0>	Замена
<;}>	< ;} >	Точная копия

Из таблицы видно, что один участок получен из другого путем полного копирования и замены имен некоторых идентификаторов и литералов, а также вставкой дополнительного ключевого слова. Поэтому эти изменения можно отнести к третьему типу. Если не учитывать добавления дополнительного «шума», то копируемый фрагмент может быть отнесен к клонам типа II.

1.4. Совпадения типа IV

Этот тип копий является результатом семантического сходства между двумя или более фрагментами кода. В этом типе клонов, клонированный фрагмент не обязательно скопирован с оригинала. Два фрагмента кода могут быть разработаны двумя разными программистами, реализовывать одинаковую логику и аналогичные по своей функциональности алгоритмы.

Рассмотрим следующий фрагмент кода, где конечное значение переменной 'J' является факториалом значения переменной VALUE.

Фрагмент 1:

```
int i, j=1;
for (i=1; i<=VALUE; i++)
j=j*i;
```

Теперь рассмотрим еще один фрагмент, который является рекурсивной функцией вычисления факториала аргумента n.

Фрагмент 2:

```
int factorial(int n) {
       if (n == 0) return 1;
else        return n * factorial(n-1);
}
```

С семантической точки зрения это разные участки кода, но они одинаковы по своей функциональности и относятся к типу IV.

Глава 2. Представление исходного кода программ

Существует большое количество способов представления и дальнейшего разбора исходных кодов программ. В основном все они состоят из фазы преобразования и фазы сравнения. На первом этапе исходный текст преобразуется во внутренний формат, который позволяет использовать более эффективный алгоритм сравнения. В следующей фазе происходит непосредственный поиск совпадений и вывод некоторой количественной меры, характеризующей их количество. Поэтому целесообразно классифицировать методы обнаружения в соответствии с их внутренним форматом представления текста. Распространенными методами являются следующие [2]:

- String-based – является распространенным подходом, используемым в информатике. Применительно к проблеме обнаружения плагиата, документы сравниваются дословно с возможным перекрытием текста. Многочисленные методы были предложены для решения этой задачи, некоторые из которых были адаптированы именно для обнаружения плагиата.
- Token-based – похож на String-based подход, но избавлен от многих его недостатков. Код программы в первую очередь преобразуется в набор токенов с использованием «лексера». А затем применяется один из алгоритмов строкового сравнения для поиска совпадающих фрагментов кода.
- Tree-based – метод, в котором фаза преобразования состоит из построения абстрактного синтаксического дерева [3], которое представляет собой внутреннюю структуру программы в соответствии с некоторой контекстно-свободной грамматикой. Затем полученные деревья сравниваются между собой.
- PDG-based (PDG – ProgramDependencyGraph) – программа представляется как ориентированный граф, вершинами которого будут: точка входа, множество точек выхода, команды передачи

управления. С каждой вершиной ассоциируется структура, хранящая адрес этого узла, и адрес перехода из этого узла (если он существует). Дугам этого графа соответствуют участки программы, состоящие из команд обработки данных. Затем полученные графы для каждого файла с исходным кодом сравниваются между собой.

- Metrics-based – этот метод основан на оценке различных метрик программы. В качестве таковых могут использоваться количество вызовов функций, количество используемых переменных, количество циклов и т. д. Далее две программы сравниваются по соответствующим метрикам.
- Hybrid-based – гибридные подходы основаны на сочетание нескольких из выше перечисленных подходов и берут сильные стороны каждого из них для достижения максимального результата.

2.1.Text-based метод

Исходный код программы не подвергается никаким изменениям и представляет собой символьную строку. Метод не учитывает структуру программы и ее синтаксические особенности, поэтому переименование функций и переменных или несущественные изменения в коде могут стать серьезным препятствием для правильной работы этого метода. К плюсам такого метода можно отнести достаточно простую реализацию и возможность сравнения не только программного кода, но и обычного текста – комментариев и других незначимых частей программы. Кроме того, с текстом легко работать и перед анализом можно провести некоторую его нормализацию и удалить несущественные символы и/или слова. Так же следует отметить что данный метод не имеет привязки к языку программирования, а следовательно хорошо подходит для первой ступени алгоритма проверки.

2.2. Metrics-based метод

Исходный код программы также не подвергается никаким изменениям. Но в отличие от Text-based метода простое переименование функций и/или переменных, а так же незначительные манипуляции с исходным кодом не повлияют на работу этого метода. В качестве отправной точки для выявления плагиата выбираются некоторые количественные характеристики программы. Это могут быть количество циклов, переменных, условий или их совокупность. Поскольку в достаточно внушительном количестве языков программирования используются одинаковые ключевые слова для задания циклов/условий, то этот метод так же достаточно универсален и может работать практически независимо от языка программы. Минусом этого метода является достаточно частые ложные срабатывания. Особенно заметно это на небольших программах, так как количество основных ключевых конструкций в них отличается незначительно.

2.3. Token-based

В этом подходе весь текст преобразуется в последовательность лексем, которая затем сканируется для поиска дубликатов. При таком подходе сохраняются все существенные и пропускаются все поверхностные детали кода. Все участи исходного текста, которые легко поддаются изменению – имена переменных, функций, классов, комментарии будут игнорироваться.

Возьмем для примера две программы:

1) Program 1.

```
int sum(int a, int b){
    return a+b;
}
int main()
{
    int a = 5;
    int b = 12;
```

```
    cout << sum(a,b) << endl;
    return 0;
}
```

2) Program 2.

```
int summator(int var1, int var2){
    return var1+var2;
}

int main()
{
    int v1 = 3;
    int v2 = 11;
    cout << summator(v1,v2) << endl;
    return 0;
}
```

Очевидно, что обе программы делают одно и тоже – выводят сумму двух чисел. Незначительной обработкой код одной из них может быть получен из другой, достаточно переименовать переменные, функцию и подставить другие тестовые значения переменных. Благодаря процедуре токенизации, удастся не принимать во внимание эти аспекты и выявить похожие участки.

Разбиение программы на токены в общем виде можно представить так:

1) Разбиваем программу на наборы операторов. Затем каждому оператору, принадлежащему определенному набору, приписываем символьный или цифровой код, назначенный заранее.
2) Из полученных кодов, строим строку, сохраняя при этом порядок следования как в исходной коде.
3) Применяем один из алгоритмов сравнения для простых текстов

Благодаря такому подходу, мы автоматически игнорируем разделительные символы, названия функций и переменных (классов, объектов и так далее), предотвращаем влияние мелких изменений исходного

кода программы.

Минус такого подхода – это зависимость процесса токенизации от конкретного языка программирования. Одно из решений этой проблемы – использование нескольких различных «лексеров» для разных классов языков.

Глава 3. Реализация программной части

Рассмотрев основные типы совпадений и возможные представления исходного кода, мы можем преступить к поэтапной разработке программного комплекса, который будет выполнять требуемые задачи. Прежде всего, мы должны конвертировать наш исходный код в форму удобную для анализа и дальнейшего сравнения. Перед этим необходимо произвести некоторую нормализацию для удаления несущественных частей кода.

3.1. Нормализация

Нормализация – это процесс преобразования всех исходных данных, таким образом, чтобы убрать все несущественные части. Она включает:

- перевод всех символов в нижний регистр,
- замену всех символов табуляции на пробел,
- удаление символов перевода строк,
- замену множественных пробельных символов одним.

После проведения этого процесса исходный текст готов к дальнейшей обработке.

3.2.Токенизация

Лексема (token)– минимальная единица языка, имеющая самостоятельный смысл. Существуют следующие виды лексем

- имена (идентификаторы);
- ключевые слова;
- знаки операций;
- разделители;
- литералы (константы).

Процесс токенизации будет основываться на регулярных выражениях. В программе он выступает отдельным блоком и может работать независимо от других частей программы. На вход мы получаем текстовое представление исходного кода программы. На выходе – массив, каждый элемент которого будет являться токеном.

Класс токена имеет следующий вид:

```
public class Token {
    public TokenType type;
    public String data;

    public Token(TokenType type, String data) {
        this.type = type;
        this.data = data;
    }
}
```

Он содержит в себе поля type типа TokenType и data типа String. Поле data содержит найденный токен (лексему). TokenType представляет собой перечисление в котором описаны паттерны для различных составляющих языка. Каждый паттерн описывается регулярным выражением.

1) Ключевые слова:

   KEYWORD("asm|else|new|this|auto|enum|operator|throw|explicit|private|true|break|export|protected|try|case|extern|public|typedef|catch|false|register|typeid|reinterpret_cast|typename|class|return|union|const|friend|unsigned|const_cast|goto|signed|using|continue|if|sizeof|virtual|default|inline|static|void|delete|static_cast|volatile|struct|wchar_t|mutable|switch|dynamic_cast|namespace|template")
2) Циклы:

   CYCLE("for|do|while")
3) Типы переменных:

   VARIABLETYPE("bool|char|float|short|int|long|double")
4) Одиночные операторы:

   OPERATOR1("([\\+]|[-]|[*]|[\\/]|[=]|[%]|[>]|[<]|[!]|[~]|[&]|[|]|[/^])")
5) Двойные операторы:

   OPERATOR2("(\\^=|[\\|]=|%=|&=|/=|*=|\\-=|\\+=|>>|<<|([|][|])|&&|<=|>=|!=|[=]{2}|]|[\\-]{2}|[\\+]{2})")
6) Тройные операторы:

   OPERATOR3(">>=|<<=)")
7) Константы и числовые данные:

   NUMBER("-?[0-9]+[.]?[0-9]*")

8) Имена идентификаторов:

IDENTIFICATOR("\\w+")

3.3 W-shingling

Шингл – это небольшой, состоящий из нескольких слов, фрагмент текста, обработанный по специальной методике для анализа

Алгоритм шинглов (w-shingling) представляет собой обработку входных данных с использованием набора уникальных шинглов (N-грамм, смежных подпоследовательностей токенов в документе), которые могут быть использованы для оценки сходства двух документов [4]. Символ N обозначает количество токенов в каждом наборе.

Итак, мы имеем 2 текста и нам нужно предположить, являются ли они почти дубликатами. Реализация алгоритма подразумевает несколько этапов:

- Нормализация текстов;
- Разбиение текста на шинглы;
- Нахождение контрольных сумм;
- Поиск одинаковых подпоследовательностей.

Например, высказывание «Кто владеет информацией, тот владеет миром» после процесса нормализации может быть представлено в виде набора лексем:

(кто, владеет, информацией, тот, владеет, миром)

Тогда множество всех смежных последовательностей токенов (3-грамм) будет выглядеть так:

{(кто, владеет, информацией), (владеет, информацией, тот), (информацией, тот, владеет), (тот, владеет, миром)}

Теперь необходимо для каждого шингла найти его контрольную сумму. Для этого будет использован метод hashCode() стандартной библиотеки Java, который возвращает хэш-код для строкового объекта и вычисляется следующим образом:

*s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1], где s[i] – i-й символ строки*

длины n. A ^ – возведение в степень. (Хэш-значение пустой строки равно нулю.)

После получения хеш значения для каждого шингла получим набор

(-1220431594, -1991703130, 176721574, 458820733)

Ту же операцию проводим со вторым текстом, с которым будет происходить сравнение и получаем для него аналогичный набор хеш значений. После этого результирующее значение, выражающее процент схожести двух текстов можно будет найти по формуле:

$$r(A,B)=\frac{|S(A)\cap S(B)|}{|S(A)\cup S(B)|}*100$$ *, где |A| – размер множества A*

Таким образом, применив данный алгоритм для последовательности токенов исходных кодов программ, мы сможем выяснить процент их сходства.

3.4. Алгоритм Вагнера — Фишера

Алгоритм Вагнера – Фишера позволяет вычислить кратчайшее расстояние Левенштейна (англ. Levenshtein distance), которое отражает сходство между двумя строками [5]. Похожесть исходной и целевой строки измеряется как количество замен, вставки и удаления, необходимых для преобразования одной строки в другую. Расстояние Левенштейна (LD) увеличивается в зависимости от количества преобразований, необходимых для превращения одной строки в другую.

Если исходная строка – «тип», а целевая – «топ», то LD = 1. Это означает, что нам необходимо сделать одну замену, для того чтобы преобразовать одну строку в другую.

Реализация алгоритма включает в себя несколько шагов:

1) Устанавливаем переменную n равную длине исходной строки s.
2) Устанавливаем переменную m равную длине целевой строки t.
3) Если n = 0, возвращаем m и выходим.

   Если m = 0, возвращаем nи выходим .
4) Инициализируем первую строку от 0 до n.

Инициализируем первый столбец от 0 до m.

5) В двойном цикле по переменным n и m начинаем производить сравнения.
6) Если s[i] равно t[j], стоимость операции равна 0.

   Если s[i] не равно t[j], стоимость операции равна 1.
7) Установить ячейку матрицы (расстояние Левенштейна) d[i,j] в значение минимальное из:
   a) Ячейки сверху + 1: d[i-1,j] + 1
   b) Ячейки слева + 1: d[i, j-1] + 1
   c) Ячейки слева и сверху по диагонали + стоимость операции из п. 6: d[i-1, j-1] + cost.
8) После итераций шагов 3-6, расстояние Левенштейна будет в последней ячейке: d[m,n].

Рассмотрим работу алгоритма на примере. Предположим мы хотим посчитать расстояние Левенштейна для строк «корова» и «солома».

Первым шагом расположим этим слова по горизонтали и вертикали в таблицу. При этому не важно какое из них будет в строке, а какое в колонке. Нам понадобится одна дополнительная строка и один столбец. Пронумеруем строку от 0..n и столбец от 0..m. Таблица к текущему моменту будет выглядеть так:

		К	О	Р	О	В	А
	0	1	2	3	4	5	6
С	1						
О	2						
Л	3						
О	4						
М	5						
А	6						

Таблица 1. После первоначальной инициализации

Теперь приступим к подсчету. Начнем с колонки «К» сверху вниз, затем «О» и т.д. Нам необходимо сравнить символ в колонке с символом в строке. Если они совпадают, то мы просто кладем в текущую ячейку значение из ячейки (i-1, j-1). Если они не совпадают, тогда нужно выбрать минимум из трех значений (ячейки сверху, ячейки слева и ячейки сверху слева по диагонали), увеличенных на единицу.

Символ «К» слова «КОРОВА» и символ «С» слова «СОЛОМА» не совпадают, поэтому берем минимум из ячейки выше (1), сверху слева (0) и слева (1) от текущей ячейки. Минимум равен 0, увеличиваем на 1 и кладем в текущую ячейку. Продолжаем заполнять таблицу для остальных символов по этому алгоритму и получим в последней ячейке значение расстояния Левенштейна равное 3.

		К	О	Р	О	В	А
	0	1	2	3	4	5	6
С	1	1	2	3	4	5	6
О	2	2	1	2	3	4	5
Л	3	3	2	2	3	4	5
О	4	4	3	3	2	3	4
М	5	5	4	4	3	3	4
А	6	6	5	5	4	4	**3**

Таблица 2. После сравнения всех символов

Мы можем применить этот алгоритм для вычисления сходства исходных кодов программ. Сначала проводим процесс нормализации исходного текста, затем получаем строку из лексем одной и второй программы в процессе токенизации. И вызываем функцию нахождения расстояния Левенштейна для этих строк. Получим численное значение, выражающее разницу (LD = Diff) между двумя строками. Тогда значение выражающее процент схожести двух программ можно вычислить как:

$$Plagiarized\ Value = \left\{1 - \frac{Diff}{Max(SS, TS)}\right\} * 100, где$$

SS = длина исходной строки

TS = длина целевой строки

Если значение Plagiarized Value больше порогового значения, то мы предполагаем, что программы являются плагиатом.

3.5. Анализ характеристик программы

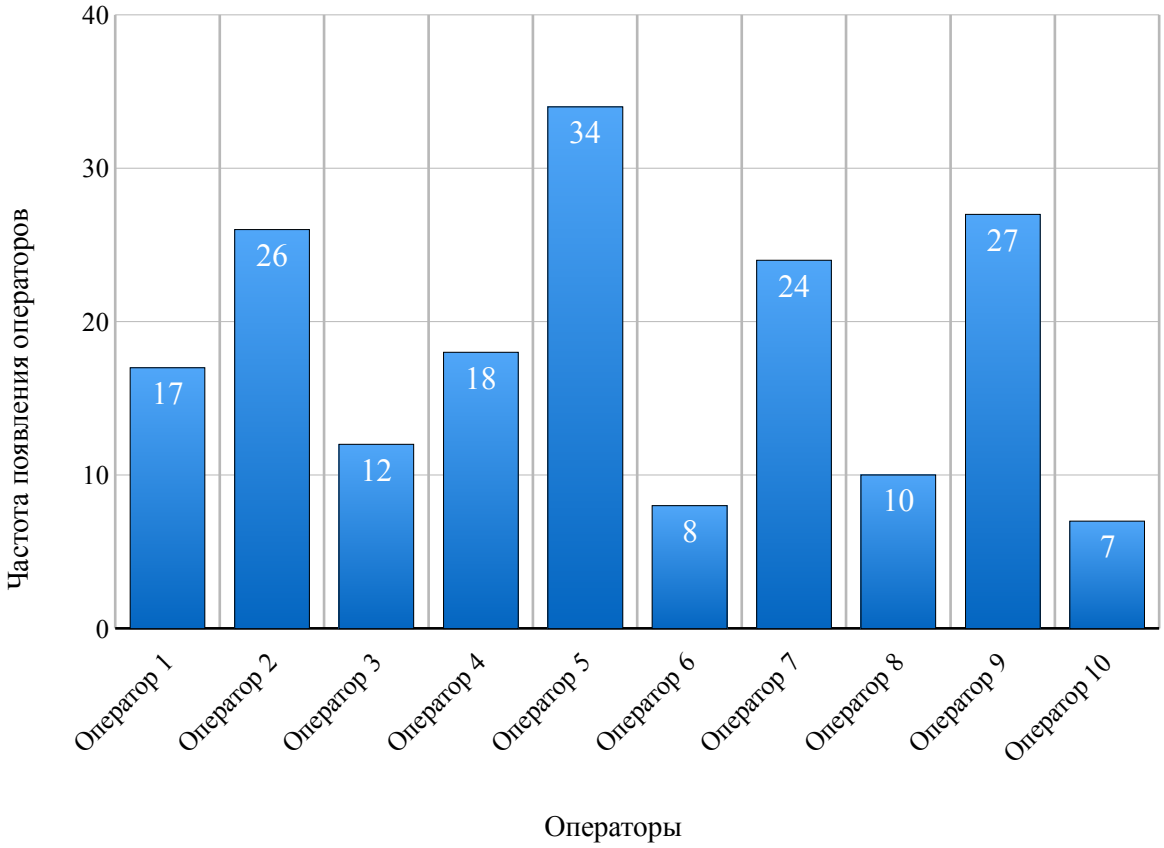
Любая программа имеет определенную структуру данных, которая может быть выявлена и использована в качестве одной из характеристик

совпадения программного кода. Применительно к доказательству факта заимствования, эта характеристика должна слабо меняться в случае модификации программы или включения фрагментов одной программы в другую. Для этой цели лучше подойдет частота появления операторов и взаимная корреляция двух программ, поскольку их модификация требует глубокого понимания логики функционирования программы и является очень трудоемким процессом.

3.5.1. Частота появления операторов

Одним из способов анализа программы является исследование частоты появления и размещения операторов в теле программы [6]. Частоту можно определить как количество появлений конкретного оператора, деленное на количество появлений всех операторов. Для наглядности процесса можно построить диаграмму. На оси абсцисс отложим номера различных операторов, а на оси ординат – частоты появления этих операторов в процентах.

При незначительном изменении кода могут быть удалены или добавлены новые операторы, но, скорее всего, общие изменения будут незначительны. Конечно высокий процент совпадений двух различных исходных кодов не всегда говорит о наличии плагиата, но служит стабильным индикатором для дальнейшей проверки.

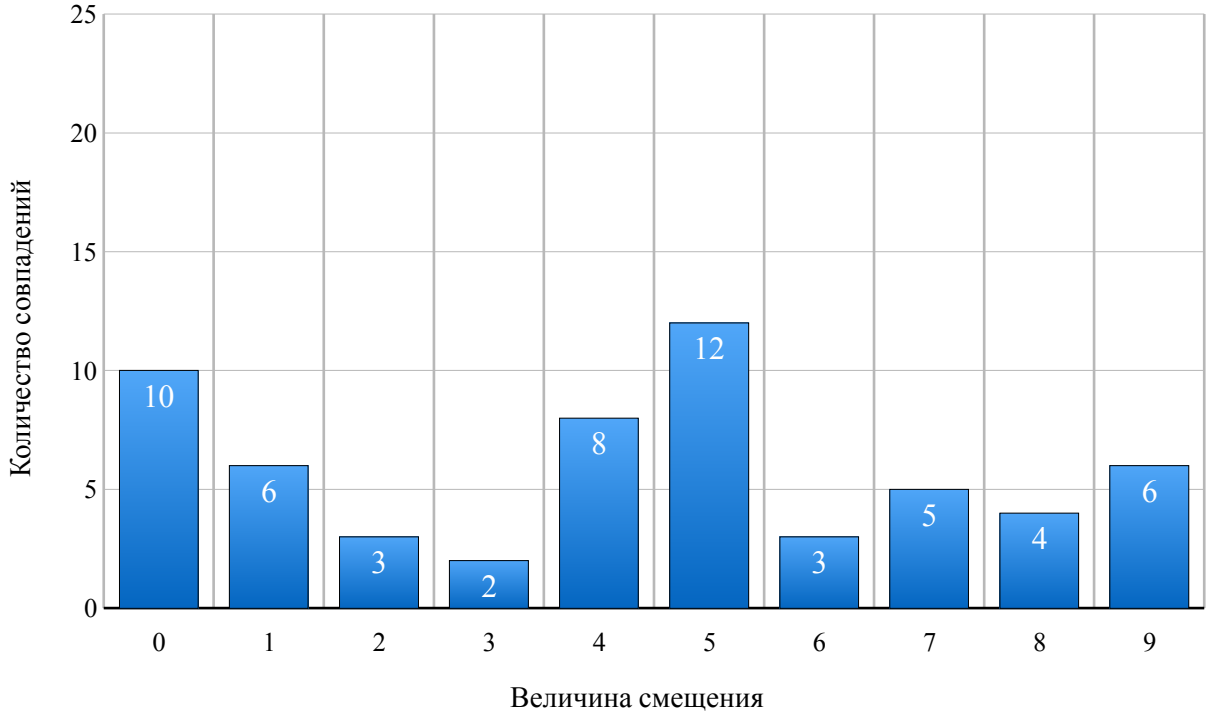


3.5.2. Взаимная корреляция двух программ

Продолжим рассматривать количественные характеристики программы. Будем анализировать размещение операторов в теле программы. Для этого нам потребуется сравнение двух последовательностей операторов и нахождение их взаимной корреляции [7].

Первый оператор одной последовательности сравниваем с первым оператором другой последовательности, второй – со вторым, третий – с третьим и так до тех пор, пока не будут перебраны все операторы. Если длины последовательностей различны, сравниваем по длине более короткой. Если выявлено совпадение – увеличиваем счетчик на единицу. Затем запоминаем значение счетчика и сдвигаем более короткую последовательность так, чтобы её первый оператор сравнивался со вторым другой последовательности, второй – с третьим, и так далее. И снова увеличиваем значение счетчика, если найдено совпадение. Эту процедуру мы делаем столько раз, сколько операторов в более длинной последовательности. Результат корреляции двух программ можно также представить в виде

диаграммы. На оси абсцисс отложим величину смещения одной последовательности относительно другой, а на оси ординат – количество совпадений операторов при таком смещении.



3.6. Анализ комментариев

Одним из самых простых, но в тоже время действенных способов выявления плагиата является анализ комментариев к программам. Для выделения комментариев (как простых, так и многострочных) из исходного кода программы применим функционал регулярных выражений. После получения строки комментариев можно воспользоваться методом Шинглов для анализа их сходства.

Отсутствие совпадений не является гарантом уникальности программ, так как комментарии поддаются легкой модификации или полному удалению. Однако, достаточно высокий процент их сходства практически точно свидетельствует о наличие плагиата в программах.

3.7. Пример работы программы

Чтобы продемонстрировать работу разработанного приложения для поиска совпадений воспользуемся реализацией простейшего алгоритма – сортировкой пузырьком, который будет модифицирован так, как будто кто-либо пытается скрыть «следы» заимствования.

Исходный код данной программы представлен ниже.

```
#include <iostream>
#include <cstdlib>
using namespace std;

// sort array function
void bubble(int* arr, int size)
{
    for (int i = size - 1; i >= 0; i--)
    {
        for (int j = 0; j < i; j++)
        {
            // if one is bigger than another - swap elements
            if (arr[j] > arr[j + 1])
            {
                int tmp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = tmp;
            }
        }
    }
}

int main()
{

    double arr[]  = {3, 2, 1, 5, 4 , 22, 31, 0, -3, -5, 1, 3};
    int size = 12;
```

```
    bubble(arr, size);

    // print sorted array
    for (int i = 0; i < size; ++i) {
        cout << arr[i] << " ";
    }

    return 0;
}
```

А модифицированный вариант имеет следующие отличия:

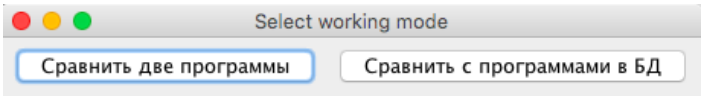
- изменено имя функции сортировки
- оставлен только один комментарий из всех
- изменены имена входных аргументов
- один из циклов for обновлен на while
- тип входного массива – double, вместо int
- все индексы внутри циклов переименованы
- модифицирован оператор сравнения

Итоговый вариант этой программы представлен ниже (полужирным помечены места, в которые вносились изменения).

```
#include <iostream>
#include <cstdlib>
using namespace std;

void bubbleSort(double* a, int n)
{
    int index1 = n-1;
    while (index1 >= 0)
    {
        for (int index2 = 0; index2 < index1; index2++)
        {
            if (a[index2 + 1] < a[index2])
            {
                int temp = a[index2];
```

```
                a[index2] = a[index2 + 1];
                a[index2 + 1] = temp;
            }
        }
        index1--;
    }
}


int main()
{

    double a[]  = {3, 2, 1, 5, 4 , 22, 31, 0, -3, -5, 1, 3};
    int n = 12;

    bubbleSort(a, n);

    // print sorted array
    for (int k = 0; k < n; ++k) {
        cout << a[k] << " ";
    }

    return 0;
}
```

Теперь можно посмотреть на результат работы приложения. Для этого запускаем его и выбираем режим работы для сравнения двух программ.



Затем вставляем исходные тексты и нажимаем сравнить в верхнем баре. Во время своей работы программа пройдет через все стадии, описанные в предыдущих разделах, а именно:

1. нормализация (весь текст будет переведен в нижний регистр и убраны незначимые символы)

2. токенизация

   Внутренне (токенизированное) представление исходных кодов из указанного примера будет следующее:

- для первого кода:

  IOIOIOIOKKIKIVOIVICVIOIONIONIOCVIONIOIIOKIIOOIIONO VIOIIOIIOOIIONOIIONOOIVIVIOONNNNNNNNONONNNVION IIICVIONIOIOIIOIIOOKN

- для второго кода:

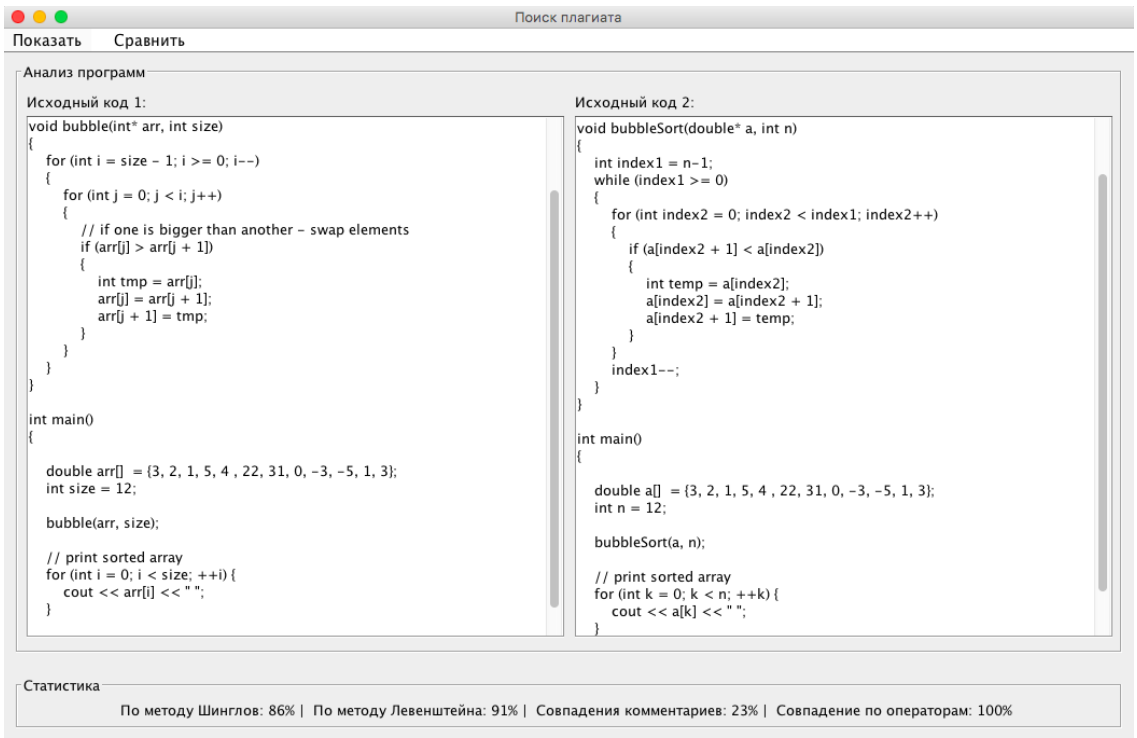
  IOIOIOIOKKIKIVOIVIVIOIONCIONCVIONIOIIOKIIONOOIIOVI OIIOIIOOIIONOIIONOOIIOVIVIOONNNNNNNNONONNNVION IIICVIONIOIOIIOIIOOKN

3. выделены и проанализированы комментарии
   a. // array sort function,

      // if one bigger another - swap elements

      // print sorted array
   b. // print sorted array
4. Последовательно сработают алгоритм подсчета расстояния Левенштейна и алгоритм Шинглов.
5. Посчитаны метрики

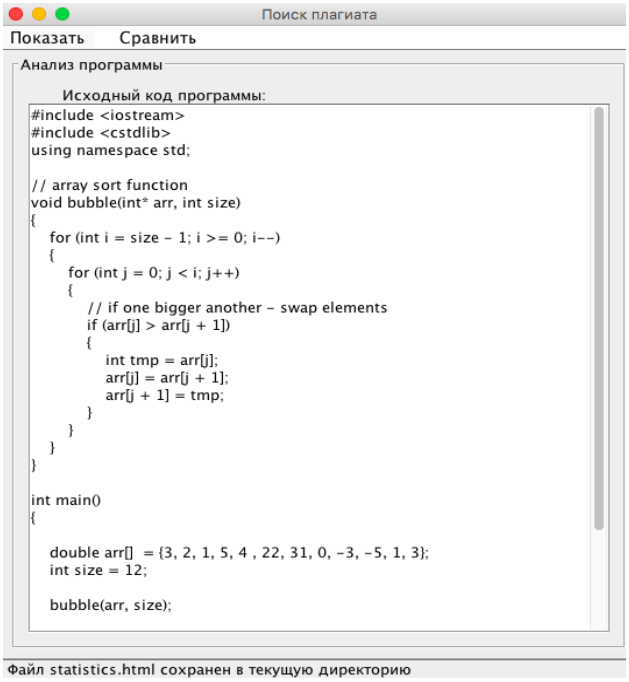
В итоге получаем следующие показатели.



Три показателя из четырех дали высокие результаты, что подтверждает

аналогичность этих исходных кодов и доказывает высокую эффективность работы приложения по обнаружению заимствований даже в случае сильной модификации кода. Совпадения по комментариям не были выявлены, так как в изменённом варианте практически все они были удалены.

Если требуется произвести сравнение программного кода с уже имеющимися в базе исходниками, достаточно выбрать режим работы для сравнения с программами в базе данных. И произвести сравнение в этом



режиме.

Тогда результат будет сохранен в формате HTML и может быть просмотрен любым браузером.

Статистика по анализу программ:

bubbleSort1	
По методу Шинглов	66%
По методу Левенштейна	90%
Совпадения комментариев	71%
Совпадение по операторам	75%

bubbleSort2	
По методу Шинглов	100%
По методу Левенштейна	100%
Совпадения комментариев	71%
Совпадение по операторам	87%

В настройках программы также возможно указать пороговое значение процентного совпадения исходных текстов, ниже которого результаты не

будут заноситься в отчет.

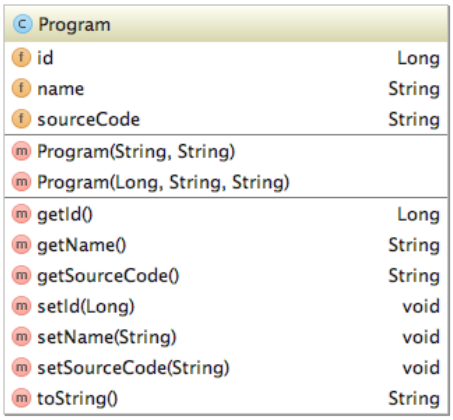
Глава 4. Структура программного кода

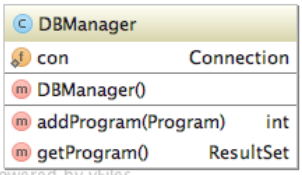
Программа состоит из четырех основных пакетов.

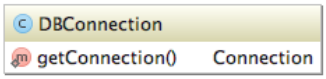
1. database – содержит классы для работы с базой данных
2. gui – содержит три класса, представляющих различные фреймы для разных режим работы программы
3. methods – включает в себя реализации методов проверки кода на наличие заимствований
4. util – в данном пакете расположен класс, включающий в себя функционал для нормализации исходного кода и пакет, содержащий классы для токенизаци исходного текста.

4.1. Пакет database

Пакет database содержит классы DBManager и DBConnection, а также класс Program, представляющий собой POJO - простой Java-объект, хранящий структуру сохраняемой в базу данных прогаммы (id, name, source code) . DBConnection содержит единственный статический метод getConnection(), возвращающий соединение с базой данных. DBManager содержит методы DAO (Data Access Object) слоя для работы с хранилищем.

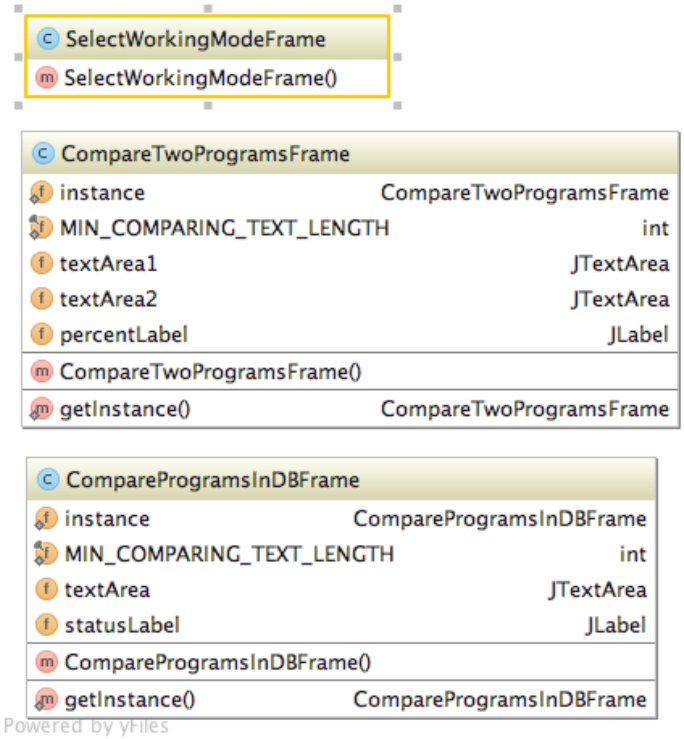






4.2. Пакет gui

Данный пакет включает в себя классы, которые отвечают за графический пользовательский интерфейс. Первый из них – окно выбора режима работы программы. Второй – окно сравнения двух программ между собой. Третий – окно для сравнения исходного кода программы с исходниками в базе данных. Все классы реализованы по паттерну синглтон, для отсутствия возможности создания нескольких экземпляров одного класса.



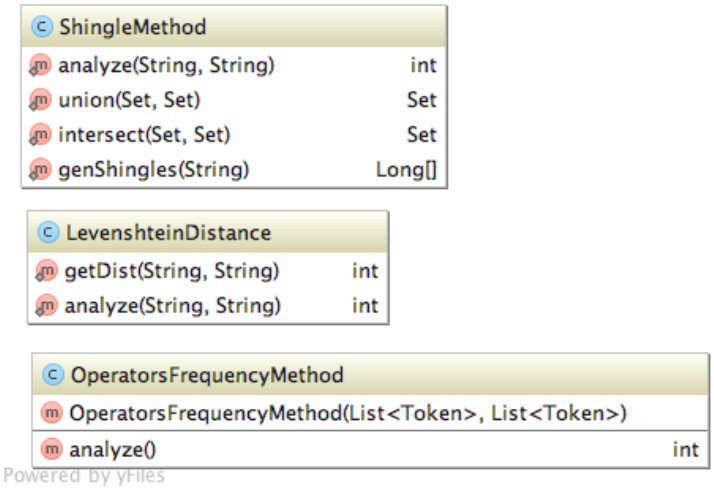
4.3. Пакет methods

Пакет methods содержит статический класс ShingleMethod, который содержит несколько методов:

- private методы union и intersect, находящие, соответственно, объединение и пересечение двух множеств и необходимые для реализаци алгоритма w-shingling
- private Long[] genShingles(String) – метод, который принимает на вход текст и возвращает массив, состоящий из вычисленных хеш-кодов N-грамм.
- public analyze – принимает две последовательности токенов или необработанных текстов и выдает результат сравнения в процентном соотношении.

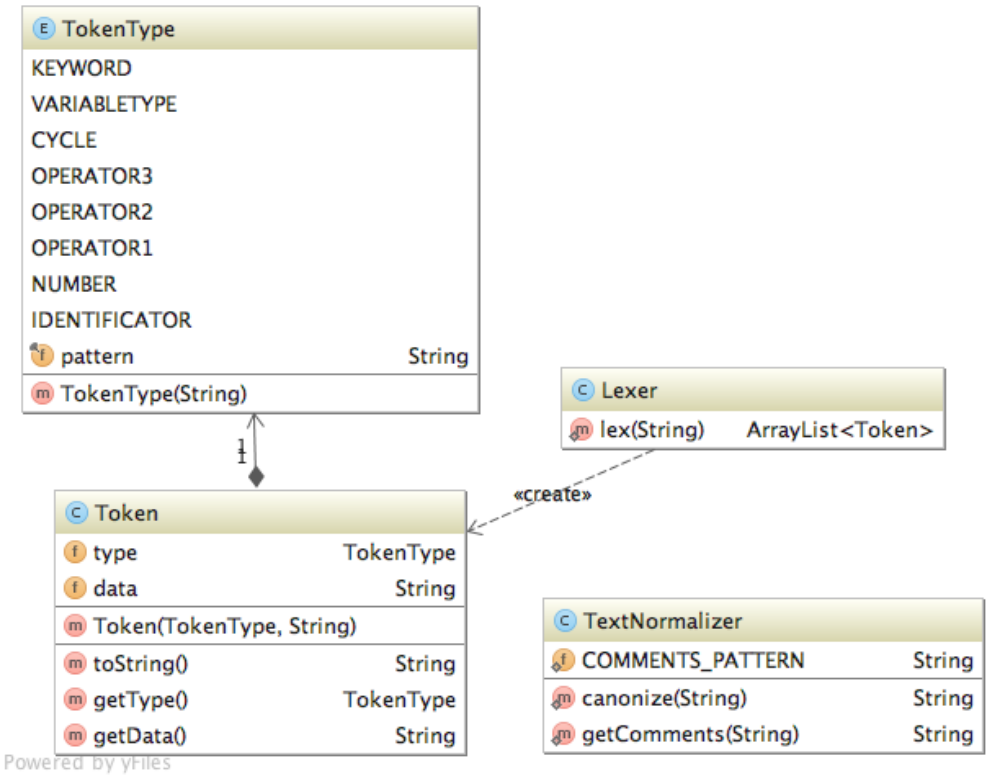
Класс LevenshteinDistance реализует алгоритм поиска расстояния Левенштейна и вычисления процентного сходства двух исходных кодов на его основе.

OperatorsFrequencyMethod – класс, который содержит один метод analyze(), анализирующий сходство на основе характеристик программ.



4.4. Пакет util

Данный пакет включает в себя перечисление TokenType, содержащее паттерны для различных типов языковых конструкций. Класс Token, представляющий из себя определенную единицу языка (поле data) и ее тип (поле type). Класс Lexer, в котором реализован единственный метод lex(String), возвращающий массив из токенов от входной строки. И вспомогательный класс TextNormalizer с методами canonize(String) для нормализации текста и getComments(String) для выделения комментариев. UML диаграмма данного пакета представлена ниже.


TokenType
KEYWORD
VARIABLETYPE
CYCLE
OPERATOR3
OPERATOR2
OPERATOR1
NUMBER
IDENTIFICATOR
pattern String
TokenType(String)
Lexer
lex(String) ArrayList<Token>
«create»
Token
type TokenType
data String
Token(TokenType, String)
toString() String
getType() TokenType
getData() String
TextNormalizer
COMMENTS_PATTERN String
canonize(String) String
getComments(String) String
Powered By yFiles

Заключение

В данной работе удалось достичь поставленных целей, а именно:

1. Выявить основные типы видоизменений исходного кода, которые позволяют скрыть факт заимствования.
2. Выделить основные преимущества и недостатки различных способов представления исходных кодов и способов их выявления
3. Собрать лучшие из них и реализовать программный продукт, который позволяет произвести комплексный анализ программ и выявлять заимствования в программном коде приложений
4. Разработанное приложение позволяет:
    a. Преобразовывать исходный код в универсальный формат, удобный для проведения анализа
    b. Производить исследование кода, используя расстояние Левенштейна и метод Шинглов
    c. Анализировать комментарии и различные характеристики – количество переменных, количество условных конструкций и циклов
    d. Осуществлять проверки «один-к-одному» и «один-ко-многим».
    e. Работать по базе данных и представлять результаты анализа в удобочитаемом формате с HTML разметкой.
    f. Работать на любой системе с установленным Java Runtime окружением

Одной из особенностей приложения является возможность проверки кода программы, даже если требуемый язык отсутствует в базе. Точность выявления при этом несколько снизится, но результат все равно будет удовлетворять заявленным требованиям и может значительно сократить время ручной проверки кода.

Программный продукт хорошо справляется с поставленной задачей, примененные алгоритмы достоверно работают при различных типах изменений исходных текстов.

Таким образом, данная программа может быть использована на практике в учебных заведениях и других организациях нуждающихся в подобном инструменте выявления заимствований.

Список литературы

1. Roy C. K. and Cordy J. R.. A survey on software clone detection research, Tech. Rep. 2007-541, School of Computing, Queen's University, Kingston, Ontario, Canada, 2007., pages 43-59.
2. Chanchal K. Roy, James R. Cordy, Rainer Koschke, Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. Science of Computer Programming Volume 74, Issue 7, 1 May 2009, pages 470–495.
3. А. В. Лаздин, О.Ф. Немолочнов. Метод построения графа функциональной программы для решения задач верификации и тестирования.
4. Mohand-Said Hacid, Zbigniew W Ras, Shusaku Tsumoto. Foundations of Intelligent Systems – 15th International Symposium ISMIS 2005, Saratoga Springs, NY, USA, May 25-28, 2005, Proceedings, pages 641-656.
5. Желудков А. В., Макаров Д. В., Фадеев П. В. Особенности алгоритмов нечёткого поиска, электронный научно-технический журнал "Инженерный вестник". Издатель ФГБОУ ВПО МГТУ им. Н.Э. Баумана, декабрь 2014, с. 501-510.
6. Merlo E., Detection of Plagiarism in University Projects Using Metrics-based Spectral Similarity, 2007, pages 3-10
7. Г.В. Стрельчёнок, Т.А. Лепихин, К.С. Лабзо Использование различных способов выявления плагиата исходных кодов в учебном процессе // Современные информационные технологии и ИТ-образование, 2015. — Т. 1, — № 11. — С. 211-214.