Подход для проведения рефакторинга «Выделение функции» в инструменте Klocwork Insight

Н.Л. Луговской
lugovskoy@ispras.ru

Аннотация. В статье рассматривается подход для проведения рефакторинга исходного кода на языках Си/Си++, реализованный в инструменте Klocwork Insight. Приводится подробное описание подхода на примере рефакторинга «Выделение функции». Разбираются способы обработки различных языковых конструкций при проведении рефакторинга, показывается, как структурные изменения в синтаксическом дереве отображаются в изменения исходного кода программы. На основе описанного подхода для проведения рефакторинга предлагается выделить методы для реализации произвольных изменений в программе, выходящих за рамки широко используемых рефакторингов. В конце статьи проводится сравнение с существующими инструментами для проведения рефакторинга.

Ключевые слова. рефакторинг; выделение функции; трансформация кода; статический анализ

1. Введение

Задача раннего выявления ошибки в дизайне программы является важной темой для исследования, так как стоимость подобных ошибок впоследствии может оказаться достаточно высокой. Ряд решений для раннего обнаружения ошибок предлагают автоматические средства статического и динамического анализа кода [1]. Однако, они мало помогают в исправлении ошибок модели программы. Одним из самых распространенных методов улучшения модели программы является рефакторинг программного кода.

Рефакторинг — это процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы [2]. Как правило, он применяется для изменения дизайна программы с целью упрощения исходного кода, улучшения его понимания разработчиком и подготовки к добавлению новой функциональности. Более того, рефакторинг может применяться на ранних стадиях процесса построения программы. Важно отметить, что использование рефакторинга увеличивает возможности программиста по «расширению» программы, что, в свою

очередь, облегчает использование таких приемов проектирования, как паттерны проектирования.

Klocwork Insight - это инструмент статического анализа кода. Он обладает гибкой системой встраивания в сборку программного продукта, использует собственные синтаксический и семантический анализаторы и поддерживает множество расширений языков Си и Си++. Использование собственных анализаторов исходного кода дает большую свободу при разработке инструмента для проведения рефакторинга.

При описании нашего подхода и для сравнения с другими решениями мы будем использовать один из самых популярных и распространенных рефакторингов, а именно, «Выделение функции».

2. Рефакторинг «Выделение функции»

В основе рефакторинга лежит последовательность небольших эквивалентных (то есть сохраняющих поведение программы) преобразований. Поскольку каждое из преобразований, как правило, незначительно влияет на изменение исходного кода программы, то программисту легче проследить за их корректностью, и в то же время вся последовательность преобразований может привести к существенной перестройке программы и улучшению её дизайна.

В известной книге «Рефакторинг. Улучшение существующего кода» [2] описываются такие классы рефакторингов, как «Составление методов», «Перемещение функций между объектами», «Организация данных», «Упрощение условных выражений», «Упрощение вызовов методов», «Решение задач обобщения». Стоит заметить, что количество различных видов рефакторинга постоянно растет, самый полный список рефакторингов можно найти на web-сайте [3]. В данной работе мы будем рассматривать один из самых популярных рефакторингов – «Выделение функции», который относится к классу рефакторингов «Составление методов».

Проводя рефакторинг «Выделение функции», программист создает новую функцию из выделенного фрагмента кода, который заменяется вызовом этой функции. Такая трансформация позволяет быстро и аккуратно реорганизовать исходный код для лучшей поддержки и читаемости [4]. Более короткое описание рефакторинга «Выделение функции» звучит так: «*У вас есть фрагмент кода, который может быть сгруппирован. Выделите этот фрагмент в функцию, чье имя объяснит ее смысл*» [5]. Применение рефакторинга «Выделение функции», очевидно, имеет положительный эффект для исходного кода программы. Например, возможность повторного использования выделенной функции уменьшает повторения одного и того же фрагмента кода; улучшается документация, т.к. имя выделенной функции объясняет ее смысл; более того, т.к. новые функции относительно малы и, как

правило, реализуют какую-то законченную функциональность, то программисту легче ими оперировать.

3. Существующие решения

Для удобной и быстрой, а главное, корректной разработки программных систем автоматические средства трансформации кода играют немаловажную роль. Они позволяют производить сложные структурные изменения исходного кода с минимальным участием программиста. При использовании такого автоматического инструмента уменьшается риск ошибки, что положительно влияет на качество кода.

Наибольшее развитие получили средства автоматического рефакторинга для языков Java и C#. Для языков Си/Си++ подобные средства развиты в гораздо меньшей степени. Это обусловлено сложностью и низкоуровненностью языков Си/Си++, например, присутствием таких конструкций, как оператор goto, глобальные переменные, указатели, адресная арифметика, ненадежное приведение типов, списки аргументов переменной длины и, особенно, использованием макросов. Отдельно стоит упомянуть, что в языках Си/Си++ используются заголовочные файлы и раздельная компиляция.

На данный момент существует небольшое число инструментов для проведения рефакторинга Си/Си++ кода. Их можно поделить на две группы: инструменты, реализованные как подключаемые модули для программной среды Microsoft Visual Studio, и встроенные компоненты сред разработки приложений. К первой группе относятся следующие инструменты:

- CodeRush - это программный инструмент, разработанный компанией DevExpres, который включает в себя такие возможности, как анализ и поиск ошибок в коде, усовершенствованный метод выделения языковых конструкций, генерация кода, средства визуализации, модуль для проведения рефакторинга Refactor! Pro и т.д. Инструмент поддерживает довольно большое количество рефакторингов, например, «Переименование функции», «Выделение функции», «Встраивание функции» [6].
- Visual Assist X - это продукт компании Whole Tomato, который включает в себя такие возможности, как рефакторинг кода, автоматическое дополнение языковых конструкций, добавляет новые виды подсветки, обладает улучшенной навигацией по сравнению со встроенной в программную среду Visual Studio, связывающей языковые сущности. Инструмент поддерживает около 12-ти рефакторингов, среди которых есть, например, «Переименование», «Выделение функции», «Инкапсуляция поля» [7].

Ко второй группе относятся следующие программные среды:

- Eclipse CDT - это свободная интегрированная среда разработки программ на языках C/C++, базирующаяся на платформе Eclipse.

  Поддерживает довольно небольшое количество рефакторингов, среди которых «Выделение функции», «Переименование», «Выделение переменной» [8]. Среда разработки предоставляет интерфейс для написания подключаемых модулей, в котором присутствуют методы для проведения трансформации исходного кода [9].
- XCode - программа для разработки приложений под OS X и iOS, разработанная компанией Apple. Включает в себя среду разработки, документацию и измененную версию набора компиляторов GNU GCC и Clang. Для языка Си поддерживается два рефакторинга: «Переименование» и «Выделение функции» [10].

Существующие инструменты для проведения автоматического рефакторинга не гарантируют, что результат рефакторинга будет синтаксически корректен и сохранится первоначальное поведение программы. Как правило, они лишь проверяют несколько предусловий, чтобы отсечь случаи, когда проведение автоматического рефакторинга невозможно или требует слишком сложного анализа [11]. При этом окончательная проверка корректности результата ложится на программиста, что значительно снижает пользу от использования автоматического средства. Стоит ли упоминать о том, что существует человеческий фактор, который в этой ситуации также может привести к ошибке в программе. Описанный принцип не дает возможности использовать существующие инструменты для проведения автоматического рефакторинга в качестве повседневного инструмента программиста для «улучшения» программного кода.

4. Принцип для проведения рефакторинга, реализованный в Klocwork Insight

Так как ни одно из существующих решений не может использоваться как надежный инструмент для проведения рефакторинга, то был предложен другой принцип для проведения рефакторинга Си/Си++ кода, основным требованием к которому является сохранение корректности кода и его поведения после применения рефакторинга. Если инструмент не может гарантировать корректность результата проводимого рефакторинга, то он либо совсем отказывается его проводить, либо сообщает некую подсказку пользователю о потенциальной ошибке.

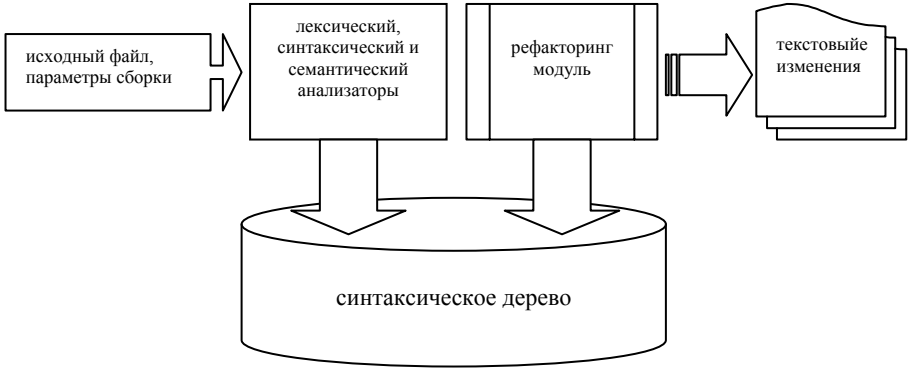
Придерживаясь такого принципа, нам удалось итеративно расширить рамки применимости нашего инструмента. В итоге, метод для проведения рефакторинга, реализованный в Klocwork Insight, поддерживает подавляющее количество языковых конструкций, сложные виды выделения, использует ряд эвристик для улучшения результата рефакторинга. В инструменте для проведения рефакторинга используется база знаний о языковых элементах и их текстовом представлении полученная напрямую от компилятора. Такая глубокая связка компилятора и инструмента для рефакторинга дает последнему возможность проводить рефакторинг предельно точно, вплоть до

сохранения пользовательской индентации и комментариев [12]. И, как будет видно из сравнительного анализа существующих инструментов для рефакторинга в главе 8, метод для проведения рефакторинга, реализованный в Klocwork Insight, во многом превосходит существующие аналоги.

4.1. Стадии процесса рефакторинга

Процесс рефакторинга исходного файла состоит из нескольких стадий. На первой стадии исходный файл с информацией, описывающей компиляцию этого файла (директории заголовочных файлов, определения макросов, специальные опции), подается на вход компилятору. В процессе компиляции последовательно запускаются лексический, синтаксический и семантический анализаторы. В результате работы анализаторов строится синтаксическое дерево исходного файла, которое передается в рефакторинг модуль. Далее, рефакторинг модуль анализирует внутреннее представление исходного файла и создает набор текстовых изменений для трансформации исходного кода. Таким образом, применяя правила из этого набора к исходному файлу, можно изменить исходный код соответственно заданному рефакторингу.

Схематически последовательность стадий процесса рефакторинга исходного файла можно изобразить схемой:



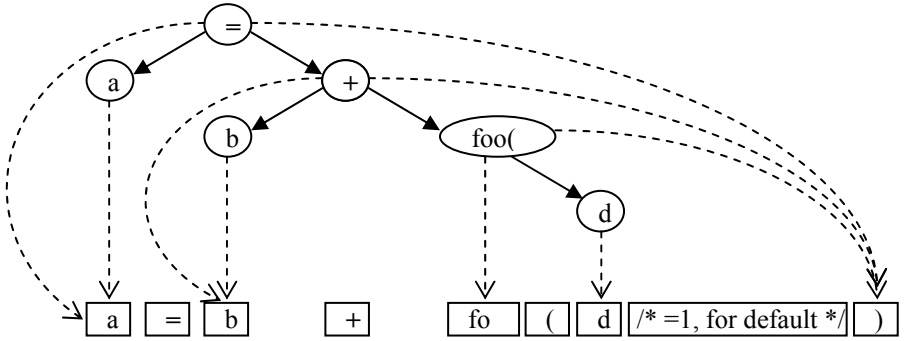
В инструменте Klocwork Insight присутствуют подключаемые модули для сред программирования Microsoft Visual Studio и Eclipse, что позволяет запускать все стадии рефакторинга автоматически. Единственное интерактивное общение с пользователем возможно только для уточнения параметров рефакторинга, например, имя новой переменной или функции.

4.2. Препроцессирование, лексический и семантический анализ

Итак, на вход компилятору подается исходный файл. На первой стадии анализа - в лексическом анализаторе, исходный файл разбивается на лексеммы-токены. Каждый токен - это одиночная атомарная частица языка, например, ключевое слово или идентификатор. Следующей стадией анализа

является препроцессирование. На этой стадии раскрываются макросы и выполняется условная компиляция. Далее, происходит синтаксический анализ последовательности токенов. На этой стадии определяется синтаксическая структура программы. И, как результат, из токенов строится синтаксическое дерево программы. Для построения синтаксического дерева используются правила, описанные в грамматике языка. Завершающей стадией компиляции является семантический анализ. На этой стадии семантический анализатор добавляет семантическую информацию к узлам синтаксического дерева. Семантический анализатор выполняет такие задачи, как проверка типов, связывание определения и использования переменных и функций, вычисление областей видимости. После стадии семантического анализа построенное синтаксическое дерево используется для анализа и проведения изменений, соответствующих выбранному рефакторингу.

Традиционно компиляторы не сохраняют последовательность токенов [13]. Она удаляется в момент построения синтаксического дерева. Однако, без этой последовательности невозможно однозначно восстановить исходный текст. Если пытаться строить исходный код программы по синтаксическому дереву, то неизбежна потеря «стиля» исходного кода программы. Более того, т.к. комментарии представляют из себя токены, но не имеют соответствующих узлов, то при генерации исходного кода программы по синтаксическому дереву комментарии будут утеряны. Вопрос о генерации исходного кода программы очень важен для рефакторинга, т.к. изменения на уровне синтаксического дерева необходимо преобразовывать в изменения на уровне исходного кода. Поэтому, чтобы сохранить оригинальное текстовое представление программы, синтаксический анализатор не удаляет токены, а связывает их с узлами синтаксического дерева. При таком подходе сохраняются и комментарии, и исходный стиль программного кода. Например, синтаксическое дерево и токен-последовательность для выражения «*a = b + foo(d /* =1, for default*/)*» будет выглядеть следующим образом:



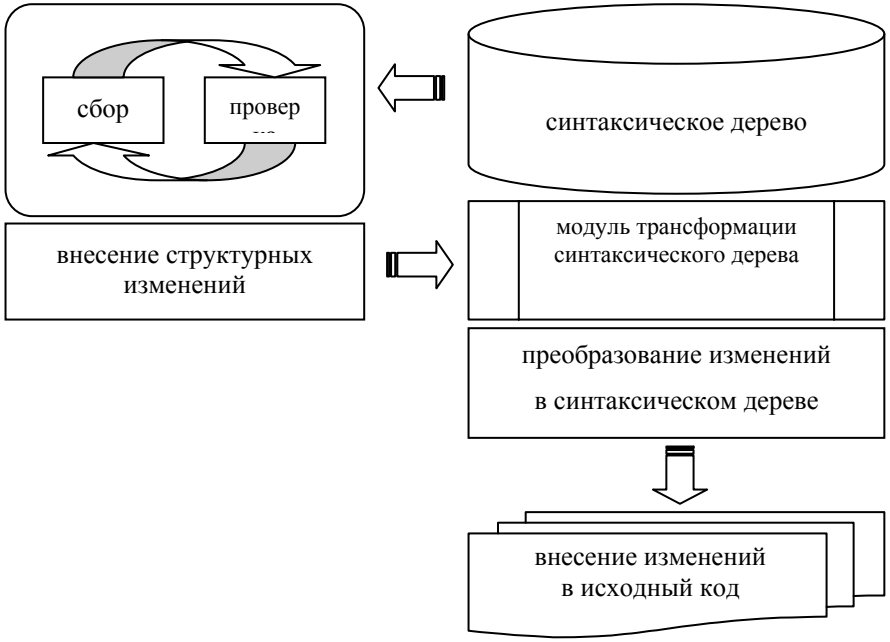
Можно добавить, что некоторые существующие анализаторы сохраняют также пробелы и переводы строк в виде токенов [14]. Однако, эта информация избыточна, т.к. ее можно получить из позиций обычных токенов.

4.3. Рефакторинг-модуль

Основным компонентом, где, собственно, и происходит анализ и построение изменений для выбранного рефакторинга, является Рефакторинг-модуль (далее РМ). Он анализирует синтаксическое дерево и, если дерево было построено корректно и без ошибок, создает набор текстовых изменений для трансформации исходного кода, необходимых для проведения рефакторинга. РМ также запрашивает у пользователя информацию о виде рефакторинга, его параметры. Например, для рефакторинга «Выделение функции» запрашивается имя новой функции. Работа РМ состоит из нескольких последовательных стадий:

1. поиск данных в выделенном фрагменте и всей программе;
2. проверка на корректность выделенного фрагмента кода, а также локальной, для выделенного фрагмента, области;
3. внесение структурных изменений в синтаксическое дерево;
4. преобразование изменений в синтаксическом дереве на уровень исходного кода.

Эту последовательность стадий удобно представить в виде схемы:



Первые три стадии работы РМ являются уникальными для каждого отдельного вида рефакторинга. Поиск и анализ данных, необходимых для проведения рефакторинга, происходит одновременно с проверкой на их корректность. Такими данными являются, например, используемые переменные, вызовы функций, декларации и т.д. Собрав необходимую «базу»

данных о программе и выделенном фрагменте, РМ проводит структурные изменения в синтаксическом дереве программы, которые соответствуют выбранному рефакторингу. На последней стадии сканируется синтаксическое дерево и для измененных частей создается текстовое представление для внесения изменений в исходный код. Эти текстовые представления могут быть приведены, например, к *нормальному формату* утилиты diff [15] для автоматического внесения изменений в файл.

5. Процесс рефакторинга «Выделение функции»

Процесс рефакторинга «Выделение функции» начинается со сбора и анализа данных о выделенном фрагменте. Главным образом, РМ собирает информацию обо всех переменных, находящихся внутри выделенного фрагмента. Например, необходимо определить следующие характеристики: тип переменной, входит ли в выделенный фрагмент ее декларация, является ли она аргументом функции, область видимости переменной, ее использование до и после выделенного фрагмента, различные виды использования (чтение, запись, перезапись, взятие адреса и т.д.).

Переменные, найденные в выделенном фрагменте кода, являются, по сути, основными данными для построения новой функции. Фактически, зная перечисленную информацию о выделенных переменных, можно уже определить сигнатуру новой функции. Далее достаточно поместить выделенный фрагмент кода внутрь тела новой функции и изменить в нем способ доступа к переменным. Такой подход является базовым при проведении рефакторинга «Выделение функции». Поддержку новых языковых конструкций, которые требуют дополнительного анализа, можно реализовывать постепенно, так как они не отменяют, а лишь добавляют соответствующие корректировки в базовый подход.

5.1. Анализ переменных в выделенном фрагменте кода

Чтобы собрать информацию о переменных внутри выделенного фрагмента кода, достаточно совершить обход синтаксического поддерева, соответствующего этому фрагменту. В рефакторинге «Выделение функции» заданный фрагмент кода всегда находится внутри функции, поэтому РМ обходит только синтаксическое поддерево, соответствующее этой функции.

Среди всех найденных переменных автоматически выделяются переменные, которые только читаются и только записываются. Их мы будет называть соответственно входными и выходными параметрами новой функции. Если выходная переменная всего одна, то она будет возвращаемым параметром. Остальные будут передаваться в функцию согласно виду использования. Например, входная целочисленная переменная должна передаваться в новую функцию по значению, а выходная по указателю или ссылке.

5.1.1 Использование переменной вне выделенного фрагмента

Хотя анализ выделенного фрагмента точно определяет, является переменная входной или выходной, этого бывает недостаточно, чтобы правильно определить новую функцию. Например:

```
int main() {
    int a = 0;
    if (bar())
        a = 1;
    return a;
}
```

При анализе выделенного фрагмента переменная «*a*» будет помечена как выходной параметр. Более того, так как это единственная выходная переменная, то она становится возвращаемым параметром. В результате может быть проведен ошибочный рефакторинг:

```
int extracted_function() {
    int a;
    if (bar())
        a = 1;
    return a;
}
int main() {
    int a = 0;
    a = extracted_function();
    return a;
}
```

Очевидно, что если «*bar() == 0*», то в переменную «*a*» попадут неинициализированные данные, что приведет к неправильной работе программы. Если же мы знаем об инициализации переменной перед выделенным фрагментом, то это поможет определить новую функцию правильно. В этом случае рефакторинг можно провести, например, так:

```
void extracted_function(int *a) {
    if (bar())
        *a = 1;
}
int main() {
    int a = 0;
    extracted_function(&a);
    return a;
}
```


Если внутри выделенного фрагмента содержится определение переменной, то необходимо знать об использовании переменной вне выделенного фрагмента. Без этой информации невозможно определить, стоит ли выносить определение переменной внутрь новой функции.

5.1.2 Обработка переменной-массива

Для выделенной переменной, являющейся массивом, необходимо, во-первых, корректно формировать ее тип, т.е. при передаче в качестве параметра двух и более размерного массива в ее типе указывать все размеры, за исключением главного. Во-вторых, необходимо отслеживать использование оператора «*sizeof*», так как это может привести к ошибке, например:

```
int main () {
    char buf[16];
    printf(«bufsize is %u», sizeof(buf));
    return 0;
}
```

В этом примере, новая функция может иметь вид:

```
void extracted_function(char buf[]) {
    printf(«bufsize is %u», sizeof(buf));
}
```

Но, по сути, это будет изменением поведения программы потому, что «*sizeof(char[])*» не равно «*sizeof(char[16])*». Поэтому новая функция должна иметь следующий вид:

```
void extracted_function(char buf[16]) {
    printf(«bufsize is %u», sizeof(buf));
}
```

5.1.3 Оператор взятия адреса

Если в выделенном фрагменте кода присутствует оператор взятия адреса от выделенной переменной, то она обязательно должна передаваться как аргумент в новую функцию, даже если она не используется за пределами выделенного фрагмента. Рассмотрим пример:

```
int main() {
    int *ptr;
    int port = 8080;
    ptr = &port;
    set_socket_port(*ptr);
    return 1;
}
```

Как видно из примера, переменная «*port*» не используется вне выделенного фрагмента, поэтому в результате рефакторинга «Выделение функции» она может быть вынесена в новую функцию следующим образом:

```
int *extracted_method() {
    int *ptr;
    int port = 8080;
    ptr = &port;
    return ptr;
}
int main() {
    int *ptr;
    ptr = extracted_method();
    set_socket_port(*ptr);
    return 1;
}
```

Что, очевидно, приведет к ошибке, т.к. указатель «*ptr*» будет указывать на данные, находящиеся в освобожденной части стека.

5.1.4 Зависимость от локальных директив и типов

В выделенном фрагменте кода могут присутствовать узлы синтаксического дерева, зависящие от локальных объявлений типов и using-директив. Локальными здесь названы директивы и объявления, определенные внутри функции. В этом случае рефакторинг «Выделение функции» может привести к ошибке, если выделенный фрагмента кода содержит зависимости от локальных объявлений и директив, которые не входят в этот фрагмент. Для того, чтобы провести корректный рефакторинг «Выделение функции» необходимо отслеживать такие зависимости и добавлять используемые объявления типов и using-директивы в начало новой функции. Рассмотрим пример:

```
string foo(vector<string> &v)
{
    typedef string MyString;
    for (int i = 0; i < 10; i++) {
        MyString s = get_string();
        v.push_back(s);
    }
    return v[2];
}
```

В этом примере рефакторинг «Выделение функции» нужно делать следующим образом:


```
void extracted_method(vector<string> &v)
{
    typedef string MyString;
    for (int i = 0; i < 10; i++) {
        MyString s = get_string();
        v.push_back(s);
    }
}
```

С другой стороны, если выделенный фрагмент кода содержит локальное объявление типа или using-директиву, то, перемещая их в новую функцию, можно нарушить зависимости, идущие к ним из-за пределов выделенного фрагмента. Поэтому, локальные объявления типов и using-директивы находящиеся внутри выделенного фрагмента кода не выносятся, а только копируются в новую функцию.

5.2. Построение функции

При построении новой функции первым делом определяется ее сигнатура, причем входные переменные передаются по значению, а выходные - через указатель или ссылку (для языка Си++). Сложные типы данных также передаются по указателю или ссылке. Определение новой функции помещается перед функцией, в которой производится рефакторинг. Таким образом, все внешние зависимости остаются ненарушенными (например, зависимость от определений типов и using-директив). Построение тела новой функции происходит в два этапа. На первом этапе синтаксическое поддерево, соответствующее выделенному фрагменту кода, копируется и помещается в тело новой функции. На втором этапе изменяются способы доступа к переменным в созданной функции. Если у функции есть возвращаемая переменная, то для нее создаются определение и return-узел для возвращения ее значения.

5.3. Декларации переменных

Если выделенный фрагмент кода содержит декларации переменных, то возникают ограничения, которые необходимо учитывать при проведении рефакторинга «Выделение функции». В простом случае, когда переменная не используется за пределами выделенного фрагмента, достаточно вынести ее декларацию в новую функцию. Если же она используется за пределами выделенного фрагмента, то необходимо проводить дополнительный анализ. Рассмотрим пример:

```
class A {
public:
    int v;
    A(int i) { v = i; }
```

```
private:
    A(const A &a) { v = a.v; }
};
int foo() {
    A a(10);
    return a.v;
}
```

Выделенный фрагмент кода может быть изменен в результате рефакторинга следующим образом:

```
A extracted_function() {
    A a(10);
    return a;
}
int foo() {
    A a = extracted_function();
    return a.v;
}
```

Если обратить внимание на определение класса «*A*», то можно заметить, что конструктор копирования объявлен в секции «*private*». С другой стороны, он будет вызван в функции «*foo()*» при инициализации переменной «a», что приведет к ошибке компиляции.

Для проведения корректного рефакторинга «Выделение функции» на фрагменте кода, в котором присутствуют декларации переменных, необходимо выделить декларации с инициализаторами и для каждой из них выбрать такой метод инициализации переменной, который будет использоваться в новой функции и не будет приводить к ошибке. В тех или иных случаях необходимо проверить конструктор по умолчанию, операторы присваивания, использование ссылок. Декларации переменных, у которых нет инициализатора, но которые присутствуют в выделенном фрагменте, не влияют на вид новой функции.

Для языка Си (стандарта C89) существует правило, по которому декларации переменных обязаны стоять вначале открытого блока кода. В этом случае, если выделенный фрагмент содержит только декларации переменных, причем за ним следуют другие декларации, то необходимо проверить используются ли выделенные переменные внутри инициализаторов деклараций, которые следуют за выделенным фрагментом. Например, для следующего выделенного фрагмента кода нельзя провести рефакторинг «Выделение функции», т.к. в языке Си нельзя вставить вызов функции перед декларацией переменной «*c*».


```
int main()
{
    int a = 1;
    int b = 2;
    int c = a++ + b;

    return a + b + c;
}
```

Кроме этого, необходимо учитывать количество выделенных деклараций, т.к. это влияет на вид новой функции. Например, если выделена только одна декларация переменной, то при проведении рефакторинга «Выделение функции» достаточно заменить инициализатор декларации на вызов новой функции. При выделении фрагмента кода, в котором присутствуют две и более декларации, логичнее создавать функцию для инициализации переменных, не привязанную к какой-либо из деклараций.

5.4. Узлы выхода

Для языков Си/Си++ узлами выхода являются операторы «*return*», «*continue*» и «*break*», причем можно заметить, что два последних оператора стоит учитывать только, если выделенный фрагмент находится внутри цикла. В зависимости от количества узлов «выхода», их вида и места расположения должны применяться различные стратегии для проведения рефакторинга «Выделение функции». Рассмотрим пример:

```
int main(int argc, char *argv[])
{
    int i;
    if (argc < 2)
        return -1;
    i = argc;
    …
    return 0;
}
```

Очевидно, что этот выделенный фрагмент нельзя заменить только вызовом новой функции. Для того чтобы сохранить поведение программы необходимо провести замену выделенного фрагмента так, чтобы сохранилось условие вызова и возвращаемое значение оператора «*return*». В приведенном примере можно заменить выделенный фрагмент кода условным оператором с вызовом новой функции в качестве условия вызова оператора «*return*»:

```
int main(int argc, char *argv[])
{
    int i;
    if (extracted_method(argc, &i))
        return -1;
    …
    return 0;
}
```

Выделенный фрагмент кода может быть сложнее, например, могут присутствовать несколько узлов выхода с разными возвращаемыми значениями. В таком случае можно завести новую переменную, через которую функция будет возвращать значение, например:

```
int foo(char *ptr)
{
    int ret;
    ret = bar(ptr);
    if (ret == -1)
        return 128;
    if (ret == -2)
        return 129;
    ...
}
```

В этом примере рефакторинг «Выделение функции» можно провести следующим образом:

```
int extracted_function(int *ret, char *p, int *o)
{
    *ret = bar(p);
    if (*ret == -1) {
        *o = 128;
        return 1;
    }
    if (*ret == -2) {
        *o = 129;
        return 1;
    }
    return 0;
}
int foo(char *ptr)
{
    int ret;
```



```
    int o;
    if (extracted_func(&ret, ptr, &o))
        return o;
    ...
}
```

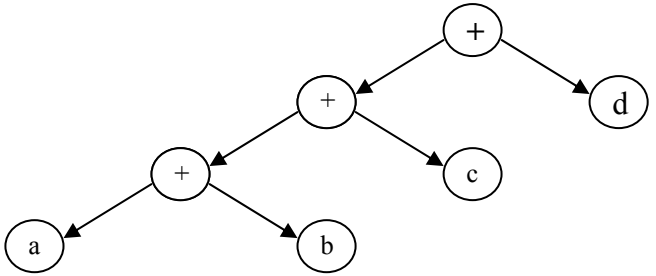
Если в выделенном фрагменте кода одновременно присутствуют различные узлы выхода, то предпочтительнее не проводить рефакторинг. Т.к. результат, полученный при применении рефакторинга, может оказаться более запутанным, нежели исходный фрагмент кода, что, очевидно, противоречит цели рефакторинга.

5.5. Проверка выделенного выражения

Если выделенный фрагмент кода является частью другого выражения, например, «*d - a + b*», то необходимо проверить, что проведение рефакторинга «Выделение функции» не нарушит приоритет операций, а, следовательно, не изменит поведение программы.

Рассмотрим пример: «*a = b && c == d + e*». Очевидно, что после проведения рефакторинга «Выделение функции», а именно вынесения выражения «*b && c*» в новую функцию, будет нарушен приоритет операций, и, как следствие, будет изменено поведение программы.

Частично эту проблему решает синтаксический анализатор, т.к. для выделенного фрагмента не будет существовать узла в синтаксическом дереве. Можно сказать, что существование соответствующего узла в синтаксическом дереве является достаточным условием для подтверждения корректности выделенного фрагмента. Однако, поиск необходимого условия осложняется тем, что синтаксическое дерево может не содержать соответствующего узла для корректно выделенного фрагмента. Например, рассмотрим выражение «*a + b + c + d*», для которого будет построено следующее синтаксическое дерево:



В этом дереве нет узла, соответствующего выражению «*b + c*», но для проведения рефакторинга «Выделение функции» это выражение корректно.

Более того, можно указать похожее синтаксическое дерево, где выделение выражения «*b + c*» уже не будет корректно: «*a * b + c + d*».

Поиск необходимого условия для подтверждения корректности выделенного фрагмента кода описывается в виде алгоритма:

1. найти наименьшую вершину синтаксического дерева, являющуюся родителем для всех узлов из выделенного фрагмента кода;
2. пройти путь снизу-вверх от самого левого и правого узлов выделенного фрагмента до найденной вершины;
3. если при прохождении пути приоритет операций не понижается, значит выделение корректно.

Стоит отметить, что для большего удобства при проведении рефакторинга «Выделение функции» существуют инструменты, которые помогают бороться с некорректным выделением, подсказывая пользователю начало и конец выделенного выражения [16]. Однако, если для выделенного выражения не существует соответствующего узла в синтаксическом дереве, как было показано выше, то эти инструменты ведут себя некорректно.

6. Модуль трансформации синтаксического дерева

Рассмотрим, как можно распечатать синтаксическое дерево в виде исходного кода:

- обходом по дереву (например, вглубь) и распечаткой заранее заготовленных текстовых шаблонов для каждого конкретного узла
- проходом по токен-последовательности верхнего узла дерева

Эти два способа сильно разнятся между собой. Первый из них базируется на структуре дерева, но не имеет информации об его текстовом представлении. Второй базируется на точном текстовом представлении дерева, но не имеет информации об его структуре. Рассмотрим пример:

```
#define START_NUM 0x5
for (i = START_NUM /* start num of … */; i < len;
i++) {
    a = str[i]; // str is …
    if (a) // fatal
        exit(0);
}
```

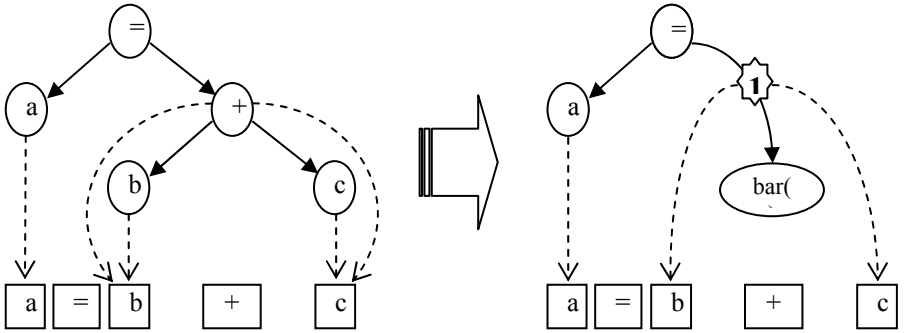
В результате распечатки синтаксического дерева первым способом произойдет потеря ряда важной информации, и результат будет примерно следующим: «*for(i=0x5;i<len;i++){a=str[i];if(a)exit(0);}*». Очевидно, что такой стиль исходного кода является неприемлемым. С другой стороны, если в результате рефакторинга меняется структура синтаксического дерева, то

распечатка синтаксического дерева проходом по токен-последовательности верхнего узла не будет отражать эти изменения.

Для решения задачи распечатки синтаксического дерева с максимально возможным сохранением пользовательской индентации был создан механизм, который отображает изменения в структуре синтаксического дерева на уровню его токен-последовательности. Этот механизм реализован в модуле трансформации синтаксического дерева. Он предоставляет простой интерфейс для работы с деревом (удалить/заменить/добавить узел) и позволяет сохранять по возможности оригинальный стиль исходного кода. Все изменения, которые вносятся в синтаксическое дерево, сохраняются и учитываются при распечатке узла обходом по токен-последовательностей.

6.1. Принцип работы с деревом

Синтаксическое дерево обладает единой связной токен-последовательностью, и каждый узел дерева имеет ссылки на свои начальный и конечный токены. Кроме того, каждый узел обладает точно определенными ребрами, которые описаны в спецификации синтаксического дерева. Можно заметить, что изменение узла в синтаксическом дереве - это изменение поддерева, находящегося на неком ребре, поэтому, чтобы сохранить информацию об изменении в синтаксическом дереве, ребро, ведущее к измененному узлу, помечается специальной меткой. В этой метке указывается начальный и конечный токены старого узла. Сам узел заменяется на новый. Этого достаточно, чтобы правильно определить позицию для внесения изменений в исходный код и найти токены, которые соответствуют измененным узлам синтаксического дерева. Рассмотрим пример, где в выражении «*a = b + c*» происходит замена «*b + c*» на вызов функции «*bar()*».



В этом примере ребро, идущее к узлу «*b + c*», помечается меткой «*1*». Для этой метки сохраняются токены «*b*» и «*c*», как начальный и конечный токены старого узла.

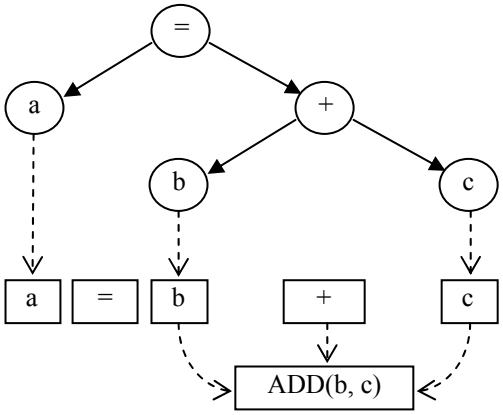
Можно отметить ряд положительных моментов в описанном подходе. Например, синтаксическое дерево изменяет свою структуру согласно трансформациям, которые проводит пользователь. Это дает возможность для

внесения изменений в уже измененные части дерева и дальнейшего структурного анализа. Сохраняется целостность токен-последовательности синтаксического дерева и, как следствие, сохраняется максимальное количество пользовательской индентации. Более того, узлы, которые были заменены другими узлами, не меняют свою токен-подпоследовательность.

Отдельно стоит упомянуть, что при распечатке синтаксического дерева в виде исходного кода важно сохранять макро-вызовы. Для того, чтобы информация о макросах присутствовала в синтаксическом дереве, существует дополнительное отображение из токенов, представляющих раскрытый макрос, в макро-вызов. Это дает возможность определить, какие токены были частью раскрытого макроса. Рассмотрим пример:

```
#define ADD(x, y) x + y
a = ADD(b, c);
```

Для этого примера будет построено следующее синтаксическое дерево:



В этом примере токены, находящиеся внутри раскрытого макроса, имеют ссылку на текстовое представление макро-вызова.

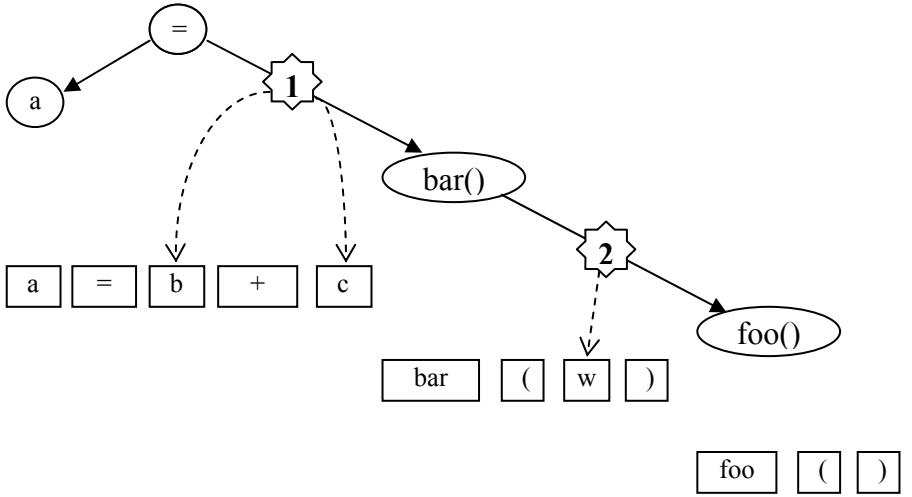
Стоит заметить, что в рассматриваемом рефакторинге «Выделение функции» выделенный фрагмент либо содержит макро-вызов целиком, либо не содержит вовсе. Более того, нет необходимости рассматривать случаи, когда макро-вызов задает неполную синтаксическую конструкцию, т.к. при выделении кода выполняется проверка на корректность выделенного фрагмента, которая, в частности, включает проверку на синтаксическую целостность.

6.2. Создание набора текстовых изменений для исходного кода

Результатом работы РМ являются изменения в исходном коде. Для этого на последней стадии РМ изменения в синтаксическом дереве отображаются в изменения исходного кода. Найти измененные узлы синтаксического дерева

является простой задачей, так как ведущие к ним ребра имеют специальную метку. Чтобы определить место, где в исходном коде начинается и заканчивается измененный узел, достаточно посмотреть на начальный и конечный токены, сохраненные для метки, т. к. каждый токен хранит свою позицию в тексте. Далее происходит распечатка измененного узла. Распечатывается узел проходом по токен-последовательности, причем токены, соответствующие измененным частям узла, пропускаются, а вместо них распечатывается новый узел (это узел, которым заменили старый узел в процессе трансформации синтаксического дерева). Процесс распечатки узла естественным образом реализуется с применением рекурсии, т.к. каждый новый узел распечатывается по тому же принципу, как и его прародитель. Более того, глубина изменений в узле может быть произвольной.

Рассмотрим пример, где в выражении «*a = b + c*» сначала «*b + c*» заменяется на вызов функции «*bar(w)*», а потом переменная «*w*» заменяется на вызов функции «*foo()*». В результате будет построено следующее синтаксическое дерево:



При замене выражения «*b + c*» на вызов функции «*bar(w)*» ребро, ведущее к «*b + c*» помечается меткой «*1*». Для метки сохраняются начальный и конечный токены выражения, а именно, токены «*b*» и «*c*». При замене переменной «*w*» на вызов функции «*foo()*» ребро, ведущее к «*w*», помечается меткой «*2*». Для нее сохраняются начальный и конечный токены узла, которые в данном случае совпадают, т.к. переменной соответствует только один токен - «*w*». Рассмотрим, как изменения на уровне синтаксического дерева отобразятся в изменения на уровне исходного кода. Обходом верхнего узла «в глубину» ищутся измененные ребра, причем обходчик не заходит внутрь таких ребер. Результатом поиска будет ребро, помеченное меткой «*1*». Используя сохраненные токены метки «*1*», можно определить позицию в

исходном коде, куда нужно вставить измененную часть дерева. После этого происходит распечатка измененной части дерева, а именно узла, к которому ведет ребро, помеченное меткой «*1*». Для текстовой распечатки используется токен-последовательность узла, причем для рассматриваемого узла такой последовательностью будет: «*bar*», «*(*«, «*w*», «*)*». Чтобы определить, какие токены в токен-последовательности соответствуют измененным частям узла, опять проводится обход в глубину с поиском измененных ребер. Таким ребром является ребро, помеченное меткой «*2*». Оно ссылается на токен «*w*», который соответствует измененному узлу. Т.к. все токены, соответствующие измененным узлам, найдены, то начинается распечатка токен-последовательности узла. Распечатываются токены «*bar*» и «*(*». Далее, т.к. токен «*w*» соответствует измененному узлу, распечатывается узел «*foo()*». После этого распечатывается последний токен «*)*». Таким образом, выражение «*a + b*» в исходном коде заменяется на выражение «*bar(foo())*».

7. Обработка ошибок компиляции

В процессе разработки программисту может понадобиться провести рефакторинг неполного, а, следовательно, синтаксически некорректного кода. Это, очевидно, дает ему определенную свободу, т.к. позволяет улучшать структуру программы еще в самом начале процесса ее написания. В этот момент программа может содержать незаконченные синтаксические конструкции. Однако, для успешного проведения рефакторинга необходимо, чтобы программа была не только синтаксически, но и семантически корректна. С другой стороны, если рефакторинг проводится над одной оконченной частью программы, то неверно в процессе рефакторинга рапортовать об ошибке, ссылаясь на незаконченность или некорректность другой части. Например, странно отказываться проводить рефакторинг «Выделение функции» внутри некой выбранной функции, если другая независимая функция программы еще не дописана. Чтобы определить, можно ли провести рефакторинг на некорректном синтаксическом дереве, необходимо проанализировать некую локальную область выделенного фрагмента кода. Т.е. проверить на синтаксическую и семантическую корректность только те узлы дерева, которые подвергнутся изменениям в процессе рефакторинга. Таким образом, подобный подход, с одной стороны, подскажет пользователю, что выделяемый для рефакторинга фрагмент кода еще не готов и должен быть дописан, с другой стороны, проведет рефакторинг без учета неоконченности программы.

8. Сравнительное тестирование существующих решений

Для объективного сравнения существующих решений, которые реализуют рефакторинг «Выделение функции», были опрошены 5 программистов.

Каждому из них было дано задание выбрать около 7 тестовых примеров различной сложности, содержащих разнообразные языковые конструкции. Одно из основных условий - это выбор примеров из реальных проектов, а не создание синтетических наборов для тестирования.

	количество пройденных тестов	*процент пройденных тестов*
Visual Assist X	4	11 %
Eclipse CDT	11	31 %
XCode	15	42 %
CodeRush	26	74 %
Klocwork Insight	31	87 %

Табл. 1.

Полученный в результате опроса набор примеров для тестирования существующих решений охватывает большое количество различных конструкций языка программирования. Например, в него входят тесты с проверками на конфликты имен, обработку типов, использование пространств имен, using-директив, сложные возвращаемые значения, обработку деклараций и узлов возврата. Каждое из существующих решений было протестировано с помощью примеров из тестового набора. Результат тестирования представлен в таблице 1.

Подробные результаты тестирования:

Visual Assist X

Правильно отрабатывает лишь на простейших примерах. Удивило отсутствие проверки выделяемого фрагмента (например, можно выделить часть комментария в новую функцию). Отсутствие правильной работы с типами, узлами возврата, декларациями переменных, анализа входных/выходных параметров постоянно приводит к неверным результатам рефакторинга.

Eclipse CDT

Инструмент правильно отрабатывает только на простейших примерах. Иногда сообщает о невозможности провести рефакторинг с указанием причины. Также присутствует проверка выделенного фрагмента, что уменьшает риск проведения рефакторинга на неправильно выделенном коде. Но поддержка ограниченного количества языковых конструкций и некорректная обработка типов и входных/выходных параметров приводит, либо к отказу проводить рефакторинг, либо к неправильным результатам.

XCode

Стоит отметить, что инструмент не поддерживает рефакторинг Си++ кода. Правильный рефакторинг возможен только на простейших примерах. Неправильно обрабатываются сложные типы данных, узлы выхода. Нет возможности выделить часть выражения, в результате в новую функцию выносится весь оператор. Не сохраняется пользовательская индентация.

CodeRush

Инструмент показал высокий уровень анализа кода. Из недостатков можно отметить недостаточный анализ связей между выделенным фрагментов и окружающими конструкциями (например, локальные объявления типов), использование конструкций языка Си++ в исходном коде на языке Си, обработку сложных возвращаемых конструкций (например, указатель на массив), ошибки при анализе макро вызовов.

Klocwork Insight

К недостаткам можно отнести отсутствие обработки частей кода, находящихся под условной компиляцией, не раскрытой в текущей конфигурации, избыточное вынесение переменных в параметры при использовании оператора «взять адрес» от этих переменных (при условии, что они используются только внутри выделенного фрагмента), отказ от рефакторинга, если необходимо изменить способ доступа к переменной, которая является аргументом макроса.

9. Перспективы развития

Рассматриваемый подход для проведения рефакторинга имеет несколько направлений для развития. Первое направление - это реализация новых видов рефакторинга. Для добавления нового рефакторинга достаточно описать правила преобразования узлов синтаксического дерева, а работу по конструированию изменений на уровне исходного кода выполнит модуль трансформации синтаксического дерева. Но стоит отметить, что процесс описания правил преобразования узлов синтаксического дерева достаточно трудоемок. Несмотря на наличие синтаксической и семантической информации о программе в виде синтаксического дерева, способ работы с

деревом нельзя назвать удобным. Например, трудоемкость написания функций, осуществляющих поиск по критерию в дереве, достаточно высока, кроме того, разнообразие типов, атрибутов и структур приводит к написанию сложных конструкций для реализации простых задач. Отсюда вытекает второе направление для развития - разработка «удобного» средства поиска и доступа к узлам и атрибутам синтаксического дерева. В качестве такого средства могут быть использованы методы работы с синтаксическим деревом, описанные в статье «Декларативный интерфейс поиска дефектов по синтаксическим деревьям: язык KAST» [17]. С их помощью, используя модуль трансформации синтаксического дерева, можно автоматически проводить почти любую трансформацию кода, например, переносить программу на новую архитектуру, или проводить сложные изменения в программе, выходящие за рамки широко используемых рефакторингов.

10. Заключение

Рефакторинг исходного кода является одной из самых распространенных и успешных техник для улучшения дизайна программы. Но, не смотря на то, что каждый из рефакторингов описывается довольно кратким и понятным образом, реализация инструмента для автоматического проведения рефакторинга является довольно сложной задачей. С другой стороны, даже несмотря на недостатки, существующие инструменты для проведения автоматического рефакторинга активно используются среди программистов, что подчеркивает актуальность выбранной темы. Приведенный подход для проведения рефакторинга «Выделение функции» содержит принципы и методы, с помощью которых результат рефакторинга обладает не только высокой точностью, но и сохраняется стиль исходного кода. Стоит отметить, что описанный подход для проведения рефакторинга был применен при реализации нескольких рефакторингов. В частности, были реализованы рефакторинги «Встраивание функции», «Введение переменной», «Встраивание переменной», которые также вошли в состав инструмента Klocwork Insight.

Список литературы

[1] В.О. Савицкий, Д.В. Сидоров. *Инкрементальный анализ исходного кода на языках C/C++*. Труды Института системного программирования РАН, том 22, 2012 г.
[2] Мартин Фаулер. *Рефакторинг. Улучшение существующего кода.*
[3] Martin Fowler. *Refactoring Home Page.* http://www.refactoring.com/
[4] http://msdn.microsoft.com/en-us/library/0s21cwxk.aspx
[5] http://www.refactoring.com/catalog/extractMethod.html
[6] http://www.devexpress.com/Subscriptions/DXperience/DXv2/index.xml
[7] http://www.wholetomato.com/products/featureRefactoring.asp
[8] http://www.eclipse.org/cdt
[9] Michael Ruegg. *Eclipse CDT refactoring overview and internals.*
[10] https://developer.apple.com/xcode

[11] Max Schaefer, Oege de Moor. *Specifying and implementing refactorings.*
[12] Zhiying (Vicky) Wang. *A Survey of Refactoring Tool Researches.*
[13] Peter Sommerlad. *Retaining comments when refactoring code.*
[14] Jeffrey Overbey, Ralph Johnson. *Generating Rewritable Abstract Syntax Trees.*
[15] http://www.opennet.ru/docs/RUS/diff/diff-3.html
[16] Emerson Murphy-Hill, Andrew Black. *Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method.*
[17] С.В. Сыромятников. *Декларативный интерфейс поиска дефектов по синтаксическим деревьям: язык KAST.* Труды Института системного программирования РАН, том 20, 2011 г.