Поиск повторяющихся фрагментов исходного кода при автоматическом рефакторинге

Н. Г. Зельцер
nzeltser@ispras.ru

Аннотация. В статье рассмотрена возможность совмещения автоматического рефакторинга с обнаружением повторяющихся фрагментов исходного кода для программ на языках C/C++. Предложена классификация программных клонов с точки зрения дальнейшего применения к ним автоматического рефакторинга. Для каждого выделенного типа клонов описан способ их поиска. Приведены недостатки существующих инструментов и показано, что предложенные методы работают корректно в рассмотренных ситуациях. Подход, описанный в статье, реализован в рамках инструмента Klocwork inSight.

Ключевые слова: рефакторинг, дубликаты кода.

1. Введение

Наличие повторяющихся фрагментов исходного кода влечет несколько проблем:

- неоправданное увеличение количества строк, а значит, ухудшение читаемости программы;
- ухудшение дизайна программы;
- усложнение поддержки программной системы;
- повышение затрат на исправление ошибок в программе, если ошибка найдена в повторяющемся коде, поскольку необходимо вносить изменения во все копии повторяющегося кода.

Для улучшения дизайна программы большую популярность приобрел рефакторинг “Выделение метода” [1](с. 110-116). Если выделенный пользователем фрагмент кода имеет дубликаты, то стоит попробовать применить тот же рефакторинг также и к ним. Таким образом, совмещение рефакторинга “Выделение метода” с поиском клонов позволяет существенно улучшить структуру программы.

Компилятор Klocwork inSight для языков C и C++ на одном из этапов работы представляет исходный код программы в виде дерева разбора на

базе дерева абстрактного синтаксиса (AST) с необходимой семантической информацией для каждого узла. Учитывая возможности компилятора по сохранению информации об оригинальной разметке исходного кода (индентации, макросах и комментариях), то возникает возможность осуществить поиск клонов с одной стороны наиболее точно и, в то же время, независимо от индентации, макросов и комментариев, сосредоточившись только на структуре кода в виде дерева разбора.

2. Существующие решения

Как правило, большинство существующих инструментов поиска клонов для программ на языках C/C++ обладают рядом недостатков и дают неточные или мало полезные с точки зрения дальнейшего рефакторинга результаты. Это происходит потому, что поиск зачастую осуществляется на уровне исходного текста программы, практически без учета структуры и семантики кода. Не учитываются типы переменных, функций и их аргументов. Нет корректной поддержки кода, содержащего макро-вызовы.

В качестве примеров, приведем существующие инструменты CloneDR [2] и CCFinder [3]. Инструмент CloneDR позволяет находить группы повторяющихся фрагментов кода в данной пользователем директории проекта, не задавая какой-либо конкретный участок кода. Как было сказано выше, к результатам работы, вообще говоря, не применим один и тот же рефакторинг “Выделение метода”. Например, следующие два фрагмента будут выданы инструментом CloneDR в качестве клонов:

```
int i; double d;
...
if (flags[i].compare(“string1”) == 0) {  // Фрагмент 1
    if (i + 1 < flags.size()) {
      //do work ...
    }
}
...
if (flags[i].compare(“string2”) == 0) { // Фрагмент 2
    if (i + 1 < flags.size()) {
      //do work ...
    }
}
...
```

```
if (flags[i].compare(“string3”) == 0) { // Фрагмент 3
    if (d + 1 < flags.size()) {
      //do work ...
    }
}
```

CloneDR выдает параметризованный шаблон, которому соответствует найденная группа клонов. При этом, если пользователь желает выполнить рефакторинг “Выделение метода” для найденных фрагментов, выбор списка параметров остается за программистом: CloneDR ничего об этом не сообщает. В данном случае шаблон выглядит так:

```
if (flags[i].compare(“$variable1”) == 0) {
    if ($variable2 + 1 < flags.size()) {
      //do work ...
    }
}
```

Очевидно, что было бы неверным заменить фрагменты 1 и 3 на вызов одной и той же функции, поскольку переменные i и d - разных типов. В данном случае клонами являются только фрагменты 1 и 2, и для них стоит выделить следующую функцию:

```
bool extracted_function(const vector<string>, const char
*e, int i) {
    if (flags[i].compare(e) == 0) {
        if (i + 1 < flags.size()) {
            //do work
        }
        return true;
    }
    return false;
}
```

Подход к поиску программных клонов, реализованный в инструменте Klocwork, учитывает вышеописанные факторы и в результате приводит к верной замене повторяющихся фрагментов на вызовы выделенной функции.

Для языка Java неплохим решением является клон-детектор, реализованный как часть среды разработки Intellij IDEA. Пользователь выделяет фрагмент кода для рефакторинга, и для него автоматически

ищутся клоны. Здесь учитывается семантика кода: типы переменных, сигнатуры функций т.д. Также плюсом является то, что фрагменты кода, отличающиеся лишь некоторыми входящими в них арифметическими выражениями, выдаются как клоны, что придает гибкость процессу рефакторинга.

Однако, данный инструмент также имеет серьезные недостатки. Во-первых, отсутствует возможность поиска клонов вне текущей функции. Во-вторых, фрагменты кода, отличающиеся лишь входящими в них числовыми и строковыми константами, не обнаруживаются как клоны. Клон-детектор, реализованный в проекте Klocwork, описанных недостатков не имеет.

Не стоит также забывать, что анализ программ на языке Java с точки зрения поиска клонов и рефакторинга в общем случае проще, чем аналогичные действия для программ на языках C/C++ в силу присутствия в последних таких понятий, как, например, указатели и макросы.

3. Типы программных клонов

Итак, с точки зрения дальнейшего рефакторинга целесообразно выделить следующие типы клонов.

3.1 Точные клоны

Два фрагмента исходного кода называются точными клонами, если они синтаксически и семантически совпадают с точностью до имен идентификаторов.

Ниже приводится пример таких клонов.

Заданный фрагмент:

```
while(1){
int fd = 4;
    int e = a + b;
    printf("%f:%f\n", a, b);
    a++; b++;
    /* комментарий*/
    send_data(fd, &e);
    d = (double*)receive_data(fd);
    sum += *d *(e);
}
```

Точный клон:

```
while(1){
    int fd = 4;
    int e = x + y;
    printf("%f:%f\n", x, y);
    x++;
    y++;
    send_data(fd, &e);
    d = (double*)receive_data(fd);
    sum += *d *(e);
}
```

Важно отметить, что точные клоны могут отличаться форматированием текста, наличием или отсутствием комментариев. Очевидно, что для данных фрагментов исходного кода можно выделить метод с тремя параметрами.

В качестве тривиального подтипа можно выделить *идентичные* клоны: точные клоны, в которых также совпадают имена соответствующих идентификаторов. Этот подтип клонов полезен при рефакторинге “Введение переменной” [1](с. 124-127). Например:

```
if (b*b + 2*d + 3 > 0) { // Пользователь выделяет условное выражение
   a = b*b + 2*d + 3;  // Идентичный клон
}
w = q*q + 2*r + 3;      // Не идентичный клон
```

После проведения рефакторинга код будет выглядеть следующим образом:

```
int new_variable = b*b + 2*d + 3;
if (new_variable > 0) {
    a = new_variable;
}
w = q*q + 2*r + 3;
```

Ясно, что второй клон не стоит заменять на `new_variable`, в связи с использованием отличающихся переменных

Кроме того, интерес могут представлять фрагменты, идентичные по структуре и семантике, но отличающиеся используемыми в них константами. Иными словами, это фрагменты, являющиеся точными

клонами с точностью до констант. Отнесем им также к этому типу клонов.

3.2 Клоны 2-го типа

Фрагмент исходного кода называется клоном 2-го типа заданного фрагмента кода, если они совпадают с точностью до того, что для каждой переменной типа T, входящей в заданный фрагмент, на соответствующем месте в фрагменте-клоне находится либо переменная типа T, либо произвольное выражение типа T.

Здесь можно провести аналогию между переменной и wildcard-символом в шаблоне, на месте которого может стоять произвольное сложное выражение.

Пример:

Заданный фрагмент:

```
while (1) {
    int e = a; // переменная-wildcard
    printf("data = %d, counter = %d", e, c);
    c++;
    send_data(fd, &e);
    d = (double*)receive_data(fd); // fd - переменная
    sum += *d *(e);
}
```

Клон 2-го типа:

```
while (1) {
  int e = x + (x + 2)/bar(q); // сложное выражение на
месте переменной
  printf("data = %d, counter = %d", e, r);
  r++;
  send_data(fd, &e);
  z = (double*)receive_data(fd + 4); // сложное выражение
на месте переменной
  sum += *z *(e);
}
```

Этот тип программных клонов представляет интерес с точки зрения рефакторинга по той причине, что для таких фрагментов можно выделить один и тот же метод всего лишь четырьмя параметрами:

```
void extracted_function(double a, int *c, int fd, double
*sum){
  double *d;
  while(1){
    int e = a;
    printf("data = %d, counter = %d", e, *c);
    (*c)++;
    send_data(fd, &e);
    d = (double*)receive_data(fd);
    *sum += *d *(e);
  }
}
```

Тогда заданный фрагмент кода следует заменить на вызов выделенного метода со следующими фактическими параметрами:

```
extracted_function(a, &c, fd, &sum);
```

Фрагмент-клон необходимо заменить такой строчкой:

```
extracted_function(x + (x + 2)/bar(q), &r, fd + 4, &sum);
```

3.3 Клоны 3-го типа

Если в заданном фрагменте кода наряду с переменными рассматривать так же сложные выражения как wildcard-символы, то можно ввести еще один тип клонов.

Фрагмент исходного кода называется клоном 3-го типа заданного фрагмента кода, если они совпадают с точностью до того, что для каждой переменной v и каждого выражения expr типа T, входящего в заданный фрагмент, на соответствующем месте в клоне находится либо переменная типа T, либо произвольное выражение типа T.

Иными словами, клон 3-го типа - это такой фрагмент кода, который отличается от оригинала только тем, что на местах переменных или некоторых выражений в нем могут стоять переменные либо произвольные выражения того же типа.

Пример:

Заданный фрагмент:

```
while(1){
    int e = a — 28 * foo(1.0); //выражение - wildcard
    printf("data = %d, counter = %d", e, c);
    c++;
```

```
    send_data(fd, &e);
    d = (double*)receive_data(fd + 4); //выражение -
wildcard
    sum += *d *(e);
}
```

Клон 3-го типа:

```
while(1){
    int e = (x + 2)/bar(q - t) + x; //выражение на месте
wildcard-выражения
    printf("data = %d, counter = %d", e, r);
    r++;
    send_data(fd, &e);
    z = (double*)receive_data(fd); //переменная на месте
wildcard-выражения
    sum += *z *(e);
}
```

Для приведенных выше фрагментов исходного кода естественно выделить тот же метод, что в примере с клонами 2-го типа. Тогда вызовы выделенного метода будут выглядеть, как:

```
extracted_function(a – 28 * foo(1.0), &c, fd + 4, &sum);
extracted_function((x + 2)/bar(q - t) + x, &r, fd, &sum);
```

4. Подход для поиска программных клонов, реализованный в Klocwork Insight.

В инструменте Klocwork Insight реализована возможность автоматического рефакторинга заданного пользователем фрагмента кода. Среди поддержанных рефакторингов такие, как “Выделение метода”, “Введение переменной” [1] (с. 124-127).

4.1 Представление AST

На вход модулю поиска повторяющихся фрагментов кода поступает AST программы, AST выделенного фрагмента кода (либо AST ближайшего объемлющего узла, если выделена часть выражения) и тип клонов: точные или идентичные. На выходе модуля - список клонов с их границами в исходном коде, а также некоторая вспомогательная информация. Также поддержан поиск клонов 2-го и 3-го типов, однако на данный момент для них не проведена интеграция с модулем

автоматического рефакторинга. На рисунке ниже схематически изображена работа инструмента.

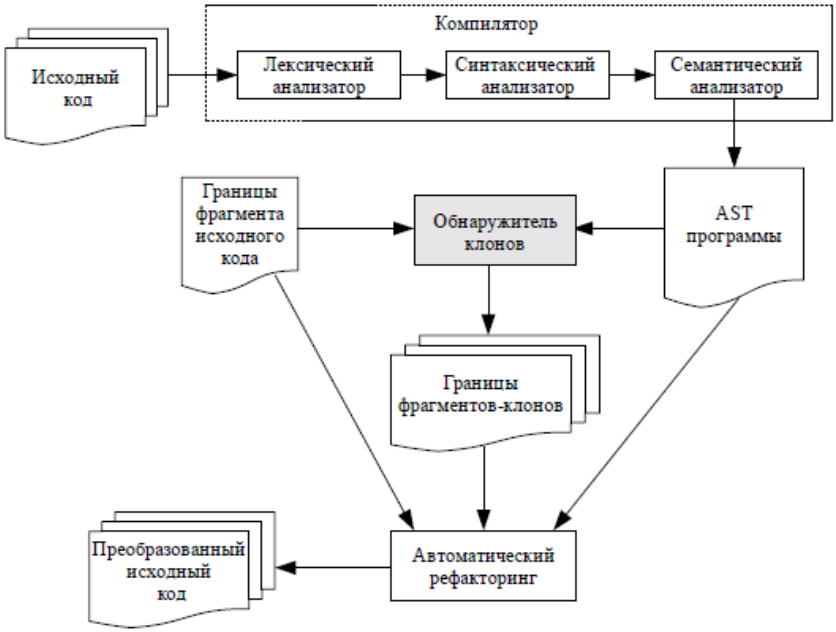

Рис. 1. Схема работы рефакторинга с поиском клонов в инструменте Klocwork

С целью поиска программных клонов, AST представляется в виде последовательности специально введенных *символов*. Для этого каждому типу узла сопоставляется числовой код и некоторая дополнительная информация, вид которой частично зависит от типа узла. Эта информация получается из семантической информации, хранимой в узле AST. Для подавляющего большинства типов узлов AST никакой дополнительной информации, кроме позиции узла в исходном коде, хранить не нужно. Интерес представляют лишь типы узлов, относящиеся к переменным и выражениям. Так, например, для узлов типа “Выражение” нужно хранить тип выражения; для узлов типа “Строковый литерал” - значение этого литерала; для узлов типа “Приведение типа” - тип исходного выражения и целевой тип. Всего для языка C++ можно выделить около 250-ти типов узлов AST.

4.2 Алгоритм поиска точных клонов

Задачу поиска точных программных клонов можно свести к задаче поиска подстроки в строке [4]. Алгоритм поиска таков:

1) Перевести AST выделенного фрагмента в последовательность символов;

2) Выполнить аналогичное действие для AST всей программы;
3) Найти во второй последовательности вхождения первой.

При этом нужно всего лишь задать отношение равенства для введенных нами выше *символов*. Например, если сравниваются два символа с одинаковым кодом, обозначающим узел-переменную, то нужно сравнить типы переменных.

4.3 Алгоритм поиска клонов 2-го типа

Поиск клонов второго типа можно проводить схожим способом. Немного меняется функция сравнения символов. В частности, необходимо сопоставлять символ, отвечающий узлу-переменной не только с символом с тем же кодом, но и с узлом, отвечающим узлу-выражению. При этом также обязательна проверка на совместимость типов переменной и выражения.

На рисунке изображен пример сопоставления строки-шаблона (сверху) с подстрокой основной строки (снизу). Числами обозначены коды символов. Символ с кодом 5 отвечает узлу AST - переменной, а символ с кодом 9 - узлу AST типа “Сложение”. Здесь мы сопоставили одному символу несколько символов - 9, 1, 6, 2, 8 - те, что отвечают поддереву AST с корнем-узлом типа “Сложение”.

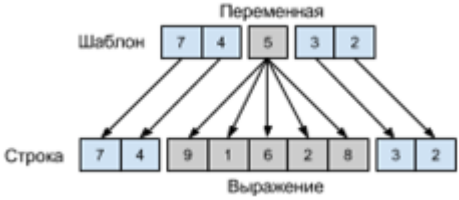

Рис. 2. Схема поиска клонов 2-го типа.

4.4 Алгоритм поиска клонов 3-го типа

Способ обнаружения таких клонов отличен от предыдущего, тем что в функцию сравнения символов добавляется симметричность. А именно: при обработке символа из строки-шаблона, отвечающего узлу AST-выражению стоит сопоставлять их не только с символом с тем же кодом, но и с с символом, отвечающим узлу AST-переменной. Также обязательна проверка на соответствие типов данных выражений. На рисунке обозначен пример соответствия строки-шаблона (сверху) с подстрокой строки, отвечающей всей программе.

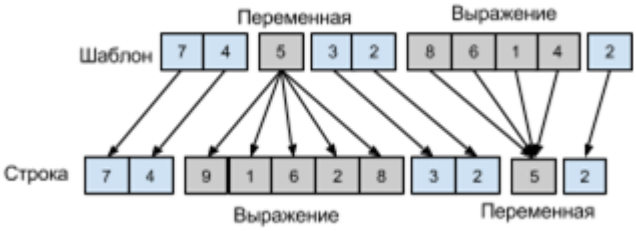

Рис. 3. Схема поиска клонов 3-го типа.

5. Примеры роботы обнаружителя программных клонов

Пример иллюстрирует работы инструмента в случае точных клонов, отличающихся лишь константами. Допустим, выделен фрагмент кода:

```
if(record.size() < 2) throw SyntaxError;
if(record.size() < 3) throw SyntaxError;
```

Тогда следующие фрагменты будут обнаружены как клоны:

```
if(record.size() < 10) throw SyntaxError;
if(record.size() < 5) throw SyntaxError;
…
if(record.size() < 3) throw SyntaxError;
if(record.size() < 4) throw SyntaxError;
```

В результате будет выделена новая фукнция `check_errors`, а приведенные выше участки кода - заменены на:

```
check_errors(2, 3);
check_errors(10, 5);
check_errors(3, 4);
```

Заключение

В статье предложен способ обнаружения повторяющегося кода с целью применения автоматического рефакторинга к найденным фрагментам. Выделено несколько типов клонов и предложены алгоритмы их поиска. Эффективность алгоритма подтверждена реализацией в инструменте Klocwork inSight. С точки зрения перспективных направлений развития данной темы можно выделить следующие.

1. Интеграция с модулем автоматического рефакторинга для случая клонов 2-го и 3-го типа.
2. Применение анализа потока данных в коде с целью более гибкого поиска клонов. Например, фрагменты могут отличаться порядком следования некоторых строк, но их перестановка не меняет функциональность программы. Кроме того, можно искать клоны, пренебрегая недосягаемым кодом или кодом, который не влияет на поведение программы.
3. Применение клон-детектора для других видов рефакторинга. Например, для рефакторинга “Замена условного оператора полиморфизмом”.

Список литературы:

[1]. M. Fowler., K. Beck, J. Brant, W. Opdyke, D. Roberts. Refactoring. Improve the design of exesting code. Addison-Wesley, 2001

[2]. http://www.semdesigns.com/products/clone/CCloneDR.html

[3]. http://www.ccfinder.net

[4]. Т.Кормен, Ч. Лейзерсон, Р.Риверст, К.Штайн Алгоритмы: построение и анализ.-М.: Вильямс, 2005.-1296 с.

Automatic clone detection for refactoring.

N. G. Zeltser
nzeltser@ispras.ru
ISP RAS, Moscow, Russia

Abstract. Software clones are repeating fragments of source code. Clones are considered harmful for many reasons and should be eliminated from the program. In this paper we study the possibility to combine automatic refactoring with the detection of repeating fragments in C/C++ source code. Classification of software clones is proposed in terms of their further use during automatic refactoring. Three types of clones are defined depending on the level of rigor: exact clones, clones withing the accuracy of concrete arithmetical expressions and clones of the third type – advanced version of the second type. For each clone type the method for detection is described. Proposed clone detection algorithm is based on program's abstract syntax tree (AST) analysis and, thus, it is accurate and does not have false positives. Existing clone detection tools are reviewed, their shortcomings are pointed out and it is shown that proposed method works correctly in considered situations. The approach described in this article has been implemented in the Klocwork inSight tool in refactorings "Extract method" and "Introduce variable". Finally, idea on improvement of the proposed approach are given: using of data flow analysis.

Keywords. refactoring, clone detection

References

[1]. M. Fowler., K. Beck, J. Brant, W. Opdyke, D. Roberts. Refactoring. Improve the design of exesting code. Addison-Wesley, 2001
[2]. http://www.semdesigns.com/products/clone/CCloneDR.html
[3]. http://www.ccfinder.net
[4]. T. Cormen, C. Leiserson, R. Rivest, K.Stein Introduction to Algorithms. MIT Press, 2002