Рефакторинг в рамках программного проекта

С. В. Сыромятников (syrom@ispras.ru), И. Е. Бронштейн (ibronstein@ispras.ru), Н. Л. Луговской (lugovskoy@ispras.ru)

Аннотация. Рефакторинг является одной из самых популярных и «успешных» техник улучшения исходного кода. Он является неотъемлемой частью гибкой методологии разработки. Однако, до сих пор наблюдается недостаток в существовании «качественных» средств проведения автоматического рефакторинга исходного кода на языках C/C++. В данной статье рассматривается один из подходов к разработке инструмента для проведения такого рефакторинга. Стоит отметить, что возможность проведения рефакторинга только на одной единице компиляции является существенным ограничением любого создаваемого инструмента. Поэтому важной особенностью данной статьи является подробное описание перехода от схемы проведения рефакторинга на одной единице компиляции к схеме проведения рефакторинга в рамках всего проекта. Кроме того, особое внимание в статье отводится рефакторингу «Переименование», так как это один из самых распространенных рефакторингов, проводимых в рамках всего проекта.

Ключевые слова. рефакторинг; переименование; глобальная область видимости; статический анализ

1. Введение

Рефакторинг - это процесс изменения внутренней структуры программы, не влияющий на её видимое поведение и имеющий целью облегчить понимание её работы [1]. Он позволяет упростить исходный код и сделать его более понятным для разработчиков. Кроме того, подобное изменение программы способно облегчить её дальнейшую модификацию и расширение. Существует достаточно большое количество программных средств, позволяющих автоматизировать процесс рефакторинга. В данной статье рассматриваются решения, реализованные в инструменте *Klocwork Insight* [2], разработанном в ИСП РАН в рамках контракта с компанией Klocwork Inc.

Выделяют разные виды автоматического рефакторинга, поддерживаемые, например, программными инструментами *Eclipse CDT*, *CodeRush* [3,4]. Для нас представляет интерес тот факт, что рефакторинг бывает *локальным*, то есть осуществляемым в рамках одной единицы компиляции, и *глобальным*, то есть затрагивающим несколько таких единиц, и потому осуществляемым в рамках всего проекта (здесь и далее под *единицей компиляции* мы имеем в

виду указываемый в команде компиляции исходный файл вместе с подключаемыми в него заголовочными файлами). Так, переименование локальных переменных является исключительно локальным рефакторингом. С другой стороны, для глобальных переменных и глобальных функций именно глобальное переименование является единственно правильным. Распространённой практикой является объявление функций (а в случае C++ - и классов) в заголовочных файлах и подключение этих файлов в несколько единиц компиляции. Очевидно, что локальное переименование таких функций, классов, методов классов, вероятнее всего сделает код некорректным.

В данной статье мы рассмотрим возможное устройство инструмента для проведения глобального рефакторинга «Переименование», опишем проблемы, возникающие в процессе создания такого инструмента, и способы их решения.

2. Локальный рефакторинг

Автоматический рефакторинг для языков C/C++ в инструменте *Klocwork Insight* основан на принципах сохранения синтаксической корректности кода и неизменности поведения программы после преобразования. Глубокая интеграция компилятора и инструмента для проведения рефакторинга позволяет проводить изменение исходного кода максимально точно, вплоть до сохранения пользовательской индентации и комментариев.

Общая схема локального рефакторинга в *Klocwork Insight*, описана ранее [5], и схематически выглядит так:

1. Проводится лексический, синтаксический и семантический анализ исходного кода. В результате строится синтаксическое дерево, некоторые узлы которого снабжены семантической информацией.
2. Выделяется подмножество синтаксического дерева, требующее преобразования. Для этого пользователь тем или иным способом (при помощи графического интерфейса или при помощи опций командной строки) указывает фрагмент программного текста, и по этому фрагменту определяются соответствующие ему узлы синтаксического дерева.
3. Автоматически определяются другие места в синтаксическом дереве или непосредственно в коде, которые должны или могут быть преобразованы. Так, при выделении фрагмента кода в новую функцию бывает полезно найти другие фрагменты, аналогичные выделенному, и предложить пользователю заменить их на вызов новой функции. Эту задачу традиционно называют *выявлением дубликатов*. Более подробно алгоритм выявления дубликатов описан в статье [6]. При переименовании переменной или функции задача преобразования состоит в том, чтобы выявить все вхождения данной переменной в исходном коде, отличив их от других одноимённых переменных и

функций. Эту задачу мы будем называть проблемой *отождествления идентификаторов*. Отметим, что в случае локального переименования она решается за счёт использования семантической информации, поскольку в задачи семантического анализатора как раз входит различение разных одноимённых идентификаторов и связывание разных вхождений одного идентификатора между собой.

4. Осуществляется требуемое преобразование выделенных узлов синтаксического дерева. При этом от пользователя может потребоваться дополнительная информация, которую он предоставляет заранее (при запуске инструмента) или вводит в интерактивном режиме. Так, при выделении новой функции требуется задать её имя.

5. По измененным фрагментам дерева формируются соответствующие текстовые представления. Поскольку окончательное изменение исходного кода, по меньшей мере, требует одобрения пользователя, подобное представление генерируется в так называемом *нормальном формате* утилиты *diff* [7], и дальнейшее изменение исходного кода не представляет труда. Высокая точность генерации изменённого кода обуславливается тем, что в узлах синтаксического дерева, строящегося в инструменте Klocwork Insight, сохраняются ссылки на соответствующие им лексемы и позиции этих лексем. Более того, для узлов, построенных по исходному коду, полученному в результате макроподстановки, хранится текстовый вид изначального макровызова. Означенные лексемы и макровызовы по возможности используются при генерации кода [5].

3. Подход к проведению глобального рефакторинга

Задача проведения рефакторинга в рамках всего проекта, как правило, сводится к задаче проведения соответствующего локального рефакторинга. Последовательно осуществляя требуемое локальное преобразование для всех единиц компиляции, возможно осуществление глобального рефакторинга. В частности, можно проводить глобальное переименование, если удается корректно отождествлять одноимённые идентификаторы. Отметим, однако, что последовательное применение локального преобразования ко всем единицам компиляции требует очень больших временных затрат. В то же время, скажем, что при переименовании идентификатора в конкретной единице компиляции может случиться так, что вхождения интересующей нас функции или переменной отсутствуют. Поэтому встаёт задача минимизации числа команд локального рефакторинга, которые запускаются при проведении рефакторинга глобального.

3.1. Отождествление идентификаторов при глобальном переименовании

В соответствии со спецификациями языков C и C++, некоторые идентификаторы в программе могут быть доступны за пределами своей единицы компиляции [8]. Про такие идентификаторы говорят, что они обладают *внешним связыванием*, и, собственно говоря, именно их переименование должно осуществляться в рамках целого проекта. При этом из спецификаций следует, что два идентификатора, имеющих внешнее связывание, обозначают один и тот же объект, если:

1. их имена совпадают;
2. они являются членами одного и того же пространства имён или одного и того же класса;
3. в случае, если они соответствуют функциям, то эти функции являются одинаковыми с точки зрения перегрузки.

Практически же это означает, что для отождествления идентификаторов при глобальном переименовании достаточно сравнивать их полные квалифицированные имена и (в случае функций) сигнатуры. Под сигнатурой понимается текстовая запись типов всех параметров функции. В этой записи «раскрываются» типы, определённые с помощью конструкции *typedef*, что соответствует семантике перегрузки функций в C++.

3.2. Задача минимизации числа команд локального рефакторинга

Для случая глобального переименования эта задача может быть решена за счёт использования некоторой базы знаний, в которой будет храниться информация, какие идентификаторы в каких исходных файлах встречаются. Подобную базу знаний мы будем называть *индексом*. Далее мы подробно рассмотрим, как он может быть организован.

3.2.1 Индексация

Как было показано в предыдущем разделе, для однозначного задания идентификатора при глобальном переименовании требуется его полное квалифицированное имя и (в случае функций) сигнатура. Это значит, что индекс должен хранить соответствие между полными квалифицированными именами идентификаторов (а в случае функций еще и сигнатур) и файлами, в которых идентификаторы встречаются. Следовательно, построение индекса требует полного синтаксического и семантического анализа всего проекта, что для больших проектов может занимать десятки минут и даже часы.

С другой стороны, за гораздо меньшее время, с привлечением одного лишь препроцессора, может быть построен так называемый *предварительный индекс*, хранящий информацию о соответствии неквалифицированных имён

идентификаторов и содержащих их файлов. Очевидно, что он избыточен, однако во многих случаях позволяет существенно уменьшить количество команд запуска локального переименования, которые необходимо выполнить для осуществления переименования глобального. По мере того, как для тех или иных файлов будет проводиться анализ (например, при автоматическом рефакторинге), предварительная индексная информация будет заменяться точной.

Отметим важный технический момент, касающийся реализации подобного индекса. На первый взгляд, напрашивается решение, при котором в индексе будут храниться соответствия вида *<имя основного исходного файла, идентификатор>*, где *имя исходного файла* - это имя файла, который соответствует единице компиляции. Поскольку назначение индекса в том, чтобы определять, для каких единиц компиляции надо проводить рефакторинг, то нас не интересует, содержится ли идентификатор непосредственно в исходном файле или в каком-то из подключаемых заголовочных. Однако практика показывает, что такое решение неэффективно. Заголовочный файл может содержать большое число идентификаторов и подключаться (непосредственно или транзитивно) в большое число исходных файлов. Информация обо всех идентификаторах из данного заголовочного файла будет продублирована для каждого из подключающих его файлов исходных. Более того, если идентификатор присутствует лишь в некотором заголовочном файле и не встречается в исходных файлах, то в случае его переименования придётся переанализировать все исходные файлы, подключающие данный заголовочный файл, что, очевидно, нерационально. Поэтому более предпочтительным оказывается подход, при котором идентификаторам ставятся в соответствие файлы, которые физически их содержат, и вводится дополнительная таблица, хранящая информацию о том, какие заголовочные файлы подключаются в той или иной единице компиляции. Такую таблицу мы будем называть *таблицей файловых зависимостей*.

При работе с индексом следует придерживаться принципа *инкрементальности*. Если какие-то файлы в проекте были изменены, индекс перестраивается только для тех единиц компиляции, которые этим изменением затронуты. Если изменению подверглись заголовочные файлы, соответствующие им единицы компиляции могут быть определены по таблице файловых зависимостей. Между сеансами работы с проектом построенный индекс сохраняется на диске. Если с момента построения индекса для некоторого файла в него были внесены изменения, то это будет выявлено путем сравнения дат, и индекс с таблицей зависимостей будут инкрементально перестроены.

3.2.2 Алгоритм поиска минимального числа команд локального рефакторинга

При построении множества подлежащих рефакторингу единиц компиляции следует уделять внимание тому, чтобы это множество не было неоправданно велико. В системе *Klocwork Insight* используется следующий алгоритм минимизации этого множества:

1. Все исходные файлы, непосредственно содержащие данный идентификатор, добавляются в множество S^B.
2. Для каждого заголовочного файла h_i (где i принимает значения 1, 2, ..., n), непосредственно содержащего данный идентификатор, вычисляется множество S^i исходных файлов, подключающих заголовочный файл h_i. Очевидно, что для проведения рефакторинга в файле h_i достаточно провести рефакторинг на единице компиляции соответствующей любому исходному файлу из S^i. Если пересечение S^i и S^B непусто, то заголовочный файл h_i исключается из дальнейшего рассмотрения.
3. Строится граф G, причем:
   - 3.1 в граф добавляются вершины vh_i, соответствующие найденным заголовочным файлам h_i;
   - 3.2 для каждого найденного заголовочного файла h_i в граф добавляются узлы vs_i соответствующие исходным файлам из S^i;
   - 3.3 между узлами vh_i и vs_i проставляются дуги, соответствующие подключению исходным файлом s_i заголовочного файла h_i.
4. В графе G ищется вершина vs_n с наибольшим количеством дуг.
5. Исходный файл s_n, соответствующий вершине vs_n добавляется в множество S^I.
6. Из графа G удаляются все vh_j, связанные дугами с vs_n.
7. Из графа G удаляется вершина vs_i.
8. Если в графе G остались вершины vh_i, то переходим к пункту 4.
9. Объединение множеств S^B и S^I даст минимальное множество единиц компиляции, для которых необходимо запустить локальный рефакторинг

4. Заключение

Автоматический рефакторинг, реализованный в системе *Klocwork Insight*, обладает высокой точностью и позволяет сохранять пользовательский стиль в преобразованных фрагментах кода. Однако, до недавнего времени существенным его недостатком была возможность применения только в

рамках одной единицы компиляции, в то время как для ряда рефакторингов возможность глобального преобразования является очень важной. В данной статье мы рассмотрели некоторые проблемы, возникающие при реализации глобального рефакторинга «Переименование» на основе соответствующего локального рефакторинга, и описали пути их решения. В настоящее время авторы продолжают активные исследования в данной области. Изучаются возможности поддержки глобальных преобразований для других видов рефакторинга, исследуются возможности повышения точности преобразований, в частности, за счёт более корректной обработки препроцессорных директив.

Список литературы

[1] Мартин Фаулер. Рефакторинг. Улучшение существующего кода.
[2] http://www.klocwork.com/products/insight/refactoring
[3] http://www.eclipse.org/cdt
[4] https://www.devexpress.com/Products/CodeRush
[5] Н. Л. Луговской. Подход для проведения рефакторинга «Выделение функции» в инструменте Klocwork Insight. Сборник трудов Института системного программирования РАН. Под ред. акад. РАН Иванникова В. П. Т. 23. М., ИСП РАН, 2012
[6] Н. Г. Зельцер. Поиск повторяющихся фрагментов исходного кода при автоматическом рефакторинге. Сборник трудов Института системного программирования РАН. Под ред. акад. РАН Иванникова В. П. т. 25 М., ИСП РАН, 2013
[7] http://www.opennet.ru/docs/RUS/diff/diff-3.html
[8] Working Draft, Standard for Programming Language C++, http://www.open-std.org/Jtc1/sc22/wg21/docs/papers/2011/n3242.pdf

Refactoring on the whole project

N. L. Lugovskoy, S. V. Syromyatnikov, I. E. Bronstein
lugovskoy@ispras.ru, syrom@ispras.ru, ibronstein@ispras.ru
ISP RAS, Moscow, Russia

Abstract. Refactoring is one of the most popular and successful techniques for improving source code. It is an integral part of agile development methods. However, C/C++ developers still lack effective tools for source code automatic refactoring. It is obviously a serious limitation when refactoring is applicable only to a single translation unit. In many cases applying refactoring to the only source file with its headers may cause errors when linking the whole software project. This article describes in details how whole project (global) refactoring can be implemented on basis of an existing single unit refactoring tool. Special attention is paid here to refactoring «Rename» as it is one of the most widely used transformations applicable to the whole project. Two important problems specific for global renaming are highlighted. First, the article describes a way of matching two different identifiers used in different translation units. Second, a problem of minimizing the number of local renamings required for the given global renaming is also discussed. It is displayed that using a database containing information about identifiers used in a project and positions of those identifiers in project sources can significantly increase the speed of global renaming.

Keywords: refactoring; rename; global scope; static analysis

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.klocwork.com/products/insight/refactoring
[3]. http://www.eclipse.org/cdt
[4]. https://www.devexpress.com/Products/CodeRush
[5]. N. L. Lugovskoj. Podkhod dlya provedeniya refaktoringa «Vydelenie funktsii» v instrumente Klocwork Insight [“Extract Function” Refactoring in Klocwork Insight Toolkit]. Trudy ISP RAN [The Proceedings of ISP RAS]. 2012, vol. 23, pp. 107-132 (in Russian).
[6]. N. G. Zeltser. Poisk povtoryayushhikhsya fragmentov iskhodnogo koda pri avtomaticheskom refaktoringe [Automatic clone detection for refactoring]. Trudy ISP RAN [The Proceedings of ISP RAS]. 2013, vol. 25, pp. 39-50 (in Russian).
[7]. http://www.opennet.ru/docs/RUS/diff/diff-3.html
[8]. Working Draft, Standard for Programming Language C++, http://www.open-std.org/Jtc1/sc22/wg21/docs/papers/2011/n3242.pdf