Санкт-Петербургский Государственный Университет

Математико-механический факультет

Кафедра системного программирования

Кроссъязыковый рефакторинг «Изменение сигнатуры метода» для IDE IntelliJ IDEA

Курсовая работа студента 445 группы
Медведева Максима Юрьевича

Научный руководитель П. А. Громов
Ст. разработчик компании JetBrains

Санкт-Петербург
2010

Оглавление

Введение

Одним из главных средств разработки программного обеспечения в настоящее время являются интегрированные среды разработки (Integrated Development Environment, IDE). IDE – это набор средств, как правило включающий в себя текстовый редактор, компилятор, отладчик и т. д. Сейчас существует множество IDE практически для всех популярных языков программирования, включая Java, C/C++, C#. Крупнейшими IDE являются Visual Studio, Eclipse, Netbeans, IntelliJ IDEA.

В качестве главных задач интегрированных средств разработки можно выделить повышение производительности программиста, упрощение создания, редактирования и отладки кодов программ, выполнение рутинной работы. Это достигается с помощью всевозможных подсказок, инспекций кода, автодополнения, интеграцией в среду разработки вспомогательных средств, в частности, системы контроля версий, системы отслеживания ошибок и т.п. Отдельным пунктом стоят рефакторинги.

Рефакторинг – это изменение внутренней структуры программы, не затрагивающее ее функциональности, имеющее целью повышение ясности и наглядности исходного кода [5]. Один из простейших примеров – это переименование переменной (метода, класса…). До появления автоматизированных средств рефакторинга, приходилось вручную искать все использования переименовываемой переменной и заменять в них имя на новое, в то время как сейчас достаточно ввести новее имя в диалоге и нажать кнопку «OK». Таким образом, автоматизированный рефакторинг освобождает нас от ручной правки кода.

Одним из самых удобных языков для автоматизированного рефакторинга стал Java. Это обусловлено тем, что язык статически типизирован, имеет довольно простую структуру и в явном виде указывает практически всю информацию о методе, классе, переменной и т. д. Это создало условия для появления интегрированных сред разработки для Java с мощными средствами рефакторинга. Одной из первых таких сред была IntelliJ IDEA [4].

В настоящее время IntelliJ IDEA поддерживает достаточно большое количество рефакторингов [2], основными из которых являются:

- Переименование и перемещение
- Изменение сигнатуры метода
- Добавление переменной, параметра, константы, поля
- Выделение метода, класса, интерфейса
- Встраивание переменной, параметра, метода, класса.

Со временем появились новые языки, компилируемые в Java byte code и исполняемые на Java Virtual Machine. К таким языкам относятся Scala, Groovy, Clojure, Jython, JRuby. Эти языки легко интегрируются с Java и между собой. Во многих современных проектах одновременно используют один из них вместе с Java или, что встречается довольно редко, сразу несколько из них.

Такое положение вещей поднимает вопрос о кроссъязыковых рефакторингах. Кроссъязыковый рефакторинг – это рефакторинг, которому подвергается код сразу на нескольких языках. Например, переименование метода, который определен в Java, а используется в Scala.

Особенность кроссъязыковых рефакторингов состоит в том, что в разных языках используются различные конструкции для одних и тех же понятий, в которых есть возможности, не поддерживаемые другими языками. Например, параметры метода со значением по умолчанию есть в Groovy, но отсутствуют в Java.

Цель моей курсовой работы – написать базовую поддержку кроссъязыкового рефакторинга «Изменение сигнатуры метода» и с ее помощью реализовать рефакторинг для языков Java и Groovy в среде IntelliJ IDEA.

Рефакторинг «Изменение сигнатуры метода» позволяет изменять заголовочную часть метода, автоматически корректируя все необходимые части кода. В Java и Groovy заголовок метода состоит из:

- модификатора доступа (private, package local, protected, public)
- типа возвращаемого значения
- названия метода
- списка параметров
    - тип параметра
    - имя параметра
    - значение по умолчанию (для Groovy)
- списка бросаемых исключений

Реализация моей курсовой работы значительно повышает ценность рефакторинга «Изменение сигнатуры метода» для программистов, использующих одновременно и Java и Groovy. В дальнейшем, появляется возможность реализовать рефакторинг для других интегрируемых языков, в том числе не связанных с Java.

1. Обзор средств и подходов

Мою задачу можно разделить на несколько независимых частей:

1. Инициализация рефакторинга: проверка выполнимости рефакторинга в данном контексте, поиск метода к которому будет применен рефакторинг.
2. Установка параметров рефакторинга (в разных языках могут быть разные параметры).
3. Поиск ошибок, возникающих при применении рефакторинга. Уведомление пользователя о них.
4. Применение рефакторинга к коду.

Рассмотрим каждую из частей по отдельности.

1.1. Инициализация рефакторинга

Инициализация рефакторинга происходит выбором соответствующего пункта меню “Refactor” или нажатием горячей комбинации клавиш. При этом сам рефакторинг определяет, может ли он быть применен к текущему элементу. Например, если текущим элементом является редактор, то поиск происходит среди конструкций кода, находящихся под кареткой. Если же была выделена вкладка “Structure”, то рефакторинг пытается выбрать элемент, выделенный в этой вкладке.

Возможны два подхода к поиску редактируемого элемента:

- явно спросить пользователя, какой именно из элементов он хочет подвергнуть рефакторингу
- попытаться выяснить этот элемент самостоятельно

Первый подход прост и позволяет точно выяснить, что именно хотел пользователь, но требует от пользователя больше действий.

Очевидно, что в большинстве случаев выяснить, к какому именно элементу необходимо применить рефакторинг, довольно просто. Например, в лис. 1 можно определить, что рефакторинг нужно применить к методу `foo`, проверив, что каретка указывает на идентификатор, играющий роль названия метода.

```
public void fo<caret>o(String s) {

}
```

Листинг 1

Поэтому второй способ выбора элемента более дружелюбен для пользователя.

1.2 Установка параметров рефакторинга

В случае Java и Groovy рефакторингу нужно указать:

- модификатор доступа
- тип возвращаемого значения
- название метода
- список параметров
    - тип параметра
    - название параметра
    - инициализатор (для Groovy)
- список бросаемых исключений

Также для каждого нового параметра надо задать значение по умолчанию, которое будет подставлено во все вызовы метода в качестве аргумента для этого параметра.

Сейчас в качестве основных альтернатив рассматривают три варианта:

1. Диалог с полным набором всех параметров, которые необходимо указать
2. Список диалогов с постепенным указанием всех параметров (мастер)
3. редактирование метода внутри текстового редактора без специального диалога

Первый вариант наиболее прост и прямолинеен. Основным его достоинством является то, что мы видим сразу всё, что мы должны отредактировать и нам не нужно переключаться между разными диалогами. Это же – и основной недостаток. Из-за большого количества элементов управления пользователь теряется и начинает думать, что все гораздо сложнее, чем на самом деле.

Второй вариант убирает недостаток перегруженности диалогов, но добавляет необходимость возвращаться на предыдущие диалоги, если в какой-то момент пользователь решит что-то поменять.

Наиболее вероятное разбиение по диалогам может быть такое:

1. Модификатор доступа, тип возвращаемого значения и название метода
2. Список параметров со всеми их атрибутами
3. Список бросаемых исключений

Далее может следовать сводная таблица всех вносимых изменений с кнопкой подтверждения.

В качестве третьего варианта может выступать набор так называемых корректоров. Корректор (quick-fix) – это небольшой рефакторинг, обычно вызываемый прямо из редактора

и не требующий большой настройки. Обычно корректоры предназначены для быстрого устранения ошибок в коде.

```
public String foo(int a) {…}

…
int x = foo(2);
```

Листинг 2

Также существуют корректоры, позволяющие удалять лишние параметры, менять типы параметров, переименовывать метод, менять область видимости метода.

Недостаток корректоров состоит в том, что они ограничивают список изменений только теми, которые меняют код на корректный. Например, в листинге 2 нельзя поменять тип возвращаемого значения метода на `double`. Но это же и их достоинство. Не нужно еще раз писать `int` в диалоге, уже написав его в объявлении переменной. К тому же, в большинстве случаев требуется именно корректировка метода.

1.3. Поиск ошибок, возникающих при применении рефакторинга. Уведомление пользователя о них

Естественно, что пользователь может указать такие параметры рефакторинга, которые нарушают корректность кода.

```
class Foo {
     public void foo(String s) {…}
     public void bar(String s) {…}
}
```

Листинг 3

Например, если метод bar переименовать в foo, то мы получим два метода с одинаковыми названиями и списком параметров, что не разрешено в Java. Следовательно, перед выполнением рефакторинга нужно постараться найти все возможные последствия и предупредить о них пользователя.

Самый простой способ найти ошибки – это провести рефакторинг и запустить измененные и зависимые от них файлы на компиляцию. Найденные ошибки можно показать пользователю. Этот способ достаточно долгий, так как рефакторинг производится полностью

и количество файлов для компиляции ничем не ограничено. Зато он гарантированно находит все ошибки компиляции.

Второй способ – это программно проверять все известные коллизии, к которым может привести рефакторинг. Это сложнее, но работает несоизмеримо быстрее. Таким образом, преимуществом данного варианта является скорость. К тому же, он дает возможность находить ошибки, которые не проявляются на этапе компиляции.

1.4. Применение рефакторинга к коду

Общая схема работы рефакторинга такая:

1. Ищем все ссылки на метод, включая вызовы, ссылки в javadoc, в xml и т.п.
2. Получаем подтверждение пользователя, что все найденные ссылки корректны (в случае, если мы в этом не уверены)
3. Производим обработку всех ссылок и самого метода

2. Описание решения

2.1. Общая архитектура

- `ChangeSignatureAction` – обработчик, с которого начинается инициализация рефакторинга. С помощью `ChangeSignatureHandler` находит целевой элемент и запускает диалог или сразу рефакторинг.
- `ChangeSignatureHandler` – интерфейс, который должны реализовывать обработчики для конкретных языков. Имеет метод `findTargetMember`, возвращающий элемент, к которому применяется рефакторинг. Второй метод `invoke` используется для запуска рефакторинга в конкретном языке.
- `ChangeSignatureDialog` – языкозависимый диалог, в котором настраивается рефакторинг. Вызывается из `ChangeSignatureHandler`. После нажатия «ОК» происходит вызов языкозависимого `ChangeSignatureProcessor`.
- `ChangeInfo` – все настройки рефакторинга инкапсулируются в объекте имплементирующем данный интерфейс.
- `JavaChangeInfo extends ChangeInfo` – интерфейс, который должны наследовать все объекты с данными о рефакторинге в языках, совместимых с Java
- `ChangeSignatureProcessorBase` – базовая реализация процессора, выполняющего рефакторинг. Все языкозависимые процессоры должны наследоваться от него.
- `ChangeSignatureUsageProcessor` – интерфейс, который должны реализовать языкозависимые обработчики ссылок и использований метода. Занимается:
  - поиском использований целевого метода в конкретном языке, которые записывает в объекты `UsageInfo`
  - проверкой, требуется ли от пользователя подтверждение корректности всех использований
  - поиском ошибок, возникнувших вследствие применения рефакторинга
  - обработкой целевого элемента, если он определен в конкретном языке
  - обработкой всех использований метода в конкретном языке
- `UsageInfo` – объект, инкапсулирующий данные о ссылке на целевой элемент.

2.2. Поиск метода

Поиск метода производится в классе, реализующим `ChangeSignatureHandler`. Для каждого языка, поддерживающего рефакторинг, должен быть реализован свой обработчик.

В Java и Groovy метод будет найден, если каретка находится в одном из положений:

- на заголовке метода,
- в списке параметров метода,
- на любой ссылке на метод,
- в списке аргументов вызова метода, если там нет другого вызова метода,
- в javadoc на ссылке на метод.

```
class X {
     public void f<1>oo(X x,<2> int i){…}
     /**
      * @see fo<3>o(X, int)
      */
     public X bar() {…}
     public static void main(String[] args) {
          fo<4>o(b<5>ar(), <6>2);
     }
}
```

Листинг 4

Например, в листинге 4 в качестве целевого метода будет выбран метод `foo`, если каретка будет находиться в позиции 1, 2, 3, 4 и 5. В позиции 5 будет выбран метод `bar`, так как ссылка на него находится “ближе” к каретке в дереве разбора, чем ссылка на `foo`.

В произвольном файле метод будет найден, если каретка находится на ссылке на метод.

2.3. Диалог

В общем случае диалог с настройками не обязателен. Если он отсутствует, то `ChangeSignatureHandler` сразу вызывает `ChangeSignatureProcessor`, и процесс переходит к поиску ссылок, пропуская стадию поиска ошибок. Ее можно пропустить, потому что в ней происходит поиск ошибок пользователя, а если есть непреодолимые коллизии, то рефакторинг просто не запускается.

Диалог для Groovy выглядит так:

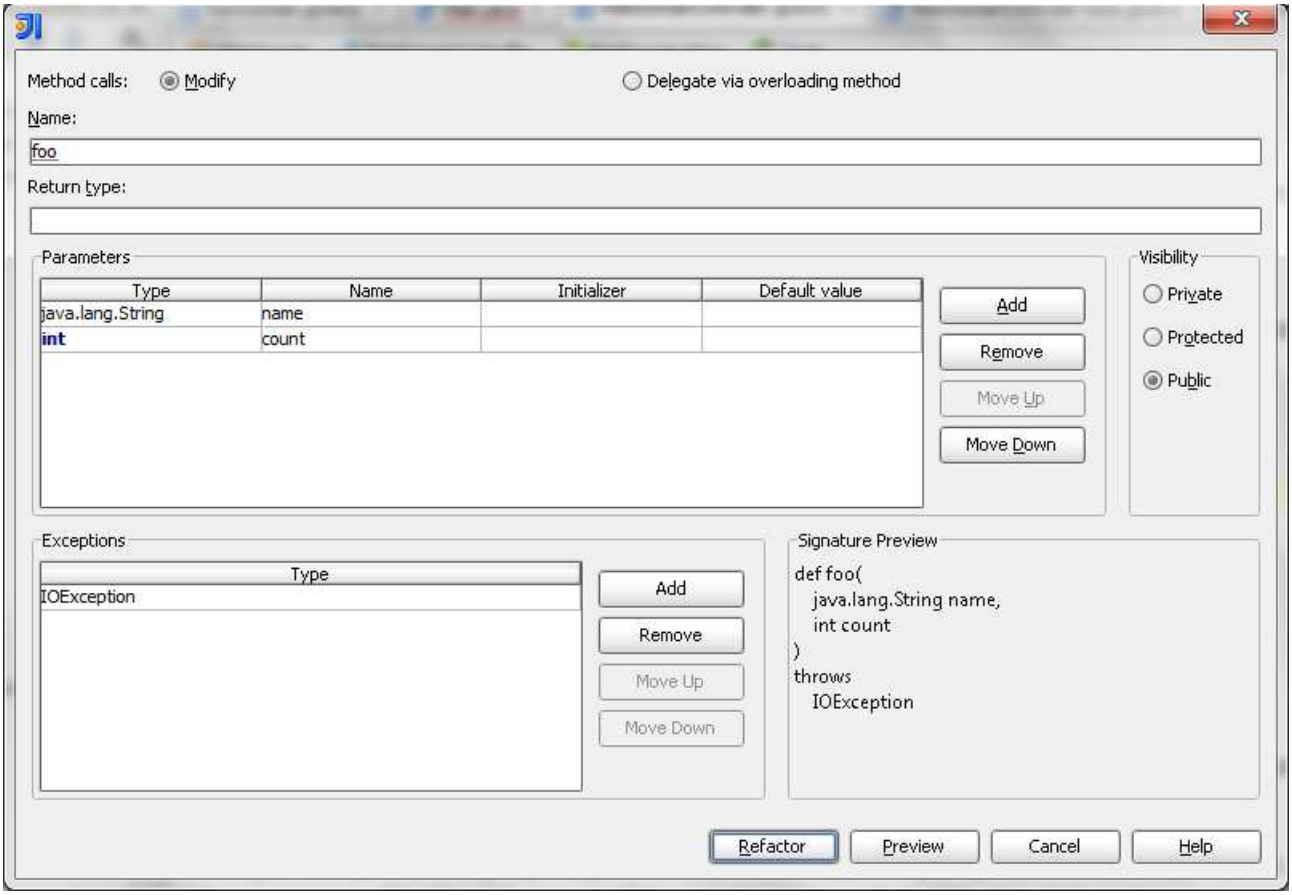

Рис. 1

Диалог разбит на несколько смысловых частей:

- панель Modify/Delegate
- панель Name/Return type
- панель Visibility позволяет установить модификатор доступа
- панель Parameters состоит из таблицы, в которой перечисляются параметры с указанием типа, названия, инициализатора и значения по умолчанию. Также есть четыре кнопки, позволяющие добавлять, удалять и перемещать параметры по списку.
- панель Exceptions состоит из таблицы, в которой перечисляются все исключения, которые бросает метод, и четырех кнопок для добавления, удаления и изменения порядка исключений.
- панель Signature Preview показывает, как будет выглядеть метод после запуска рефакторинга

- панель Refactor/Preview/Cancel/Help
    - Refactor инициализирует рефакторинг на выполнение и запускает поиск ошибок

- Preview показывает все элементы, которые будут изменены в процессе рефакторинга (сам метод, переопределяющие его методы, супер-методы и т.п.)
- Cancel отменяет рефакторинг
- Help запускает справку

В диалоге можно не указывать тип возвращаемого значения и типы параметров, так как Groovy не требует явно задавать типы. Именем метода может быть идентификатор или произвольная строка в кавычках.

2.4. Поиск ошибок

Поиск ошибок делится на две части: первичную проверку введенных пользователем данных на общую корректность и более глубокую проверку данных на нарушение корректности кода.

В рамках первой части проверяются следующие утверждения:

- название метода должно удовлетворять правилам языка на названия методов
- названия параметров должны удовлетворять правилам языка на называние параметров и не должны повторяться
- тип возвращаемого значения и типы параметров либо должны быть не указаны, либо должны удовлетворять правилам языка на “ссылки на тип”
- Все типы, указанные в таблице Exceptions должны быть унаследованы от `Throwable`

То, что выполнены все вышестоящие утверждения, не гарантирует корректность кода после выполнения рефакторинга. Например, переименовав параметр `s` в `name` в листинге 5, мы получим две переменные `name` в одном методе.

```
public String foo(String s) {
    String name = getName(s);
    return s + name;
}
```

Листинг 5

Для разрешения таких коллизий проводится второй этап поиска ошибок. Проверяются следующие случаи:

- Понижение уровня доступа метода (например, с public на protected) делает метод недоступным из некоторых мест, где метод используется. Проверка этого условия проводится одновременно с поиском всех ссылок на метод.

- В классе уже есть метод с такой же сигнатурой, как у целевого метода после применения рефакторинга. Для проверки этого условия используются собранные данные о сигнатурах методов класса модулей поддержки Java и Groovy IntelliJ IDEA.
- Измененный метод переопределяет метод супер-класса. Этот случай невозможно проверить с помощью компиляции, так как формально он разрешен, но может привести к непредсказуемым последствиям. Для этого так же используется механизм определения супер-методов модулей поддержки Java и Groovy IntelliJ IDEA.
- Новое название параметра (или новый параметр) перекрывает переменную, определенную внутри тела метода. Проверка этого условия проводится обходом дерева разбора тела метода и сбором информации о названиях всех определенных там переменных.

Обо всех найденных ошибках сообщается в специальном диалоге, который предлагает либо отменить, либо продолжить выполнение рефакторинга.

2.5. Поиск ссылок

Поиск ссылок производится языкозависимыми хендлерами, имплементирующими `ChangeSignatureUsageHandler`, с помощью метода `findUsages(…)`. Каждый из них должен найти все ссылки на целевой метод в своем языке и вернуть их, обернутыми в виде массива `UsageInfo[]`. При этом, ссылки разных типов желательно оборачивать в объекты разных классов, расширяющих `UsageInfo` и хранящих дополнительную информацию, помогающую при обработке этих ссылок.

Класс `UnresolvableUsageInfo`, расширяющий `UsageInfo`, служит для поиска ошибок, которые удобнее искать вместе со всеми ссылками на метод. При окончании поиска все `UnresolvableUsageInfo` извлекаются из найденных ссылок и показываются вместе с остальными найденными ошибками в соответствующем диалоге.

Интерфейс `PossiblyIncorrectUsage` предназначен для ссылок, которые с некоторой вероятностью могут вести к целевому методу. Например, ссылка под кареткой в листинге 6 может вести к методу `method`, но из кода это явно не следует.

```
interface Foo {
     def createInstance();
     def method(String s, int x);
}
def foo = FooFactory.createFoo().createInstance();
foo.met<caret>hod("abc", 2);
```

Листинг 6

Если среди найденных `UsageInfo[]` будут присутствовать объекты `PossiblyIncorrectUsage`, то пользователю предложат проверить найденные ссылки на корректность и удалить лишние.

Для Java и Groovy выделены следующие типы `UsageInfo`:

- `ChangeSignatureParameterUsageInfo` инкапсулирует ссылки на параметры, которые надо переименовать в теле метода.
- `MethodCallUsageInfo` хранит ссылки на вызовы метода в Java-коде. Дополнительно сохраняет информацию о подставленных параметрах типа. Например, в листинге 7, в вызове под кареткой параметру `T` соответствует `String`, а параметру `S` – `Double`.

```
class Foo<T> {
     public <S> bar(T t, S s) {…}
     public static void main(String[] args) {
          Foo<String> foo = new Foo<String>();
          foo.b<caret>ar("abc", 2.5);
     }
}
```

Листинг 7

- `OverriderUsageInfo` хранит информацию о методах, переопределяющих целевой метод в потомках. Переопределяющие методы тоже необходимо привести к той же сигнатуре, что и целевой метод с учетом подставленных параметров типа.
- `GrMethodCallUsageInfo` инкапсулирует информацию о вызове метода в Groovy-коде. Хранит в себе информацию о том, какой аргумент какому параметру соответствует, и информацию о подставленных параметрах типа.

```
def foo(Map map, String s, int x = 5,
        double y, int z = 6, String… strs) {
    …
}
foo(a:1, “s”, 4, 6.7, “a”, b:2, “b”, “c”);
```

Листинг 8

В листинге 8 вызову параметру `map` соответствуют аргументы `a:1` и `b:2`; параметру `s` – “s”; `x` – 4; `y` – 6.7; `z` пропущен; `strs` – “a”, “b”, “c”.

2.6. Выполнение рефакторинга

Выполнение рефакторинга можно разбить на три части: обработка ссылок до обработки метода, обработка метода и обработка ссылок после обработки метода.

При обработки каждой ссылки сначала выбирается подходящий `ChangeSignatureUsageProcessor`, который дальше и обрабатывает ссылку. Обработка метода также выполняется первым подошедшим `ChangeSignatureUsageProcessor`.

2.6.1. Переименование

Переименование метода происходит заменой элемента дерева разбора метода, отвечающего за заголовок, на новый – созданный из строки, указанной в диалоге. Во всех ссылках также заменяется этот элемент. Во всех переопределенных методах тоже заменяется имя.

2.6.2. Изменение области видимости

Изменение области видимости происходит добавлением нового модификатора доступа к методу и удалением старого, если он был. В Groovy, если метод стал public, то ему не добавляется никакого модификатора доступа. Также в синтаксисе Groovy нет модификатора доступа package local. Он реализован с помощью аннотации @PackageLocal. Во всех переопределенных методах происходят те же изменения.

2.6.3. Изменение типа возвращаемого значения

В методах, определенных в Java-коде, тип возвращаемого значения заменяется на новый. Если рефакторинг был запущен из диалога для Groovy и тип не был указан, то подставляется `java.lang.Object`.

В методах, определенных в Groovy, если тип не был указан и метод не имеет модификатора доступа (то есть является public), то добавляется ключевое слово def.

В переопределенных методах тип не заменяется, если он расширяет тип, указанный для целевого метода. Например, если в листинге 9 у метода `foo` изменить тип возвращаемого значения на `B`, то в классе `Inheritor` тип метода `foo` не изменится, так как он имеет тип `C`, который расширяет тип `B`. В итоге получится код как на листинге 10.

```
class A {…}
class B extends A {…}
class C extends B {…}
class  Base {
     public A foo() {…}
}
class  Inheritor  extends  Base
{
     public C foo() {…}
}
```

Листинг 9

```
class A {…}
class B extends A {…}
class C extends B {…}
class  Base {
     public B foo() {…}
}
class Inheritor extends Base {
     public C foo() {…}
}
```

Листинг 10

2.6.4. Изменение списка параметров и обработка вызовов метода

Список параметров заменяется на список, указанный пользователем в диалоге. Если для параметра не указан тип, а метод определен в Java-коде, то подставляется `java.lang.Object`.

В общем случае аргументы для удаленных параметров удаляются, для добавленных – создаются новые аргументы из указанных в столбце «Default value», для перемещенных – перемещаются на нужное место.

2.6.4.1. Конструкторы

Рассмотрим пример:

```
public class Base {
     public Base() {…}
}

public class Inheritor extends Base {
     public Inheritor(String s) {
     }
}
```

Листинг 11

При добавлении параметров к конструктору `Base()`, необходимо добавить вызов `super(…)` ко всем конструкторам прямых наследников класса `Base`, у которых первый вызов не является `super(…)` или `this(…)`, так как они неявно вызывают конструктор без параметров своего супер-класса, то есть `Base`.

То, что получится при добавлении одного параметра к конструктору `Base()` показано в листинге 12.

```
public class Base {
     public Base(int i) {…}
}

public class Inheritor extends Base {
     public Inheritor(String s) {
          super(0);
     }
}
```

Листинг 12

Также возможен случай, что у класса-наследника нет ни одного конструктора. Тогда у него будет создан конструктор без параметров с вызовом `super(…)` внутри. Ясно, что это возможно, потому что у целевого конструктора изначально параметров не было вообще, следовательно, все параметры – новые, и у каждого из них есть некоторое значение в столбце «Default value».

2.6.4.2. Массив переменной длины в качестве последнего параметра

Рассмотрим пример на Groovy:

```
class X<K> {
     def <V> get(String s, Map<K, V>… maps) {…}
}
…
def x = new X<String>()
def map = x.<Double>get("s", map1, map2, map3)
```

Листинг 13

Поменяем параметры `s` и `maps` местами, заменив тип параметра `maps` на `Map<K, V>[]`. В таком случае необходимо стоящие отдельно аргументы map1, map2 и map3 объединить в один аргумент-массив, при этом правильно расставив параметры типа.

```
class X<K> {
     def <V> get(String s, Map<K, V>… maps) {…}
}
…
def x = new X<String>()
def map = x.<Double>get([map1, map2, map3] as
     Map<String, Double>[], "s")
```

Листинг 14

При этом используется информация из GrMethodCallUsageInfo о соответствии аргументов параметрам и о параметрах типа.

Аналогичным образом обрабатываются вызовы таких методов в Java.

Обратная операция помещения массива в конец списка параметров тоже поддержана. В таком случае рефакторинг пытается «раскрыть» массив, переданный в качестве аргумента, в список аргументов соответствующего типа.

2.6.4.3 java.util.Map в качестве первого параметра в Groovy

Если в качестве первого параметра метода в Groovy-коде стоит переменная с типом `java.util.Map`, то в вызове такого метода можно указывать сколько угодно именных аргументов вида key: value. Все они будут собраны в одну `Map` и переданы в качестве первого аргумента. При перемещении этого параметра с первой позиции все именные параметры собираются в один аргумент вида `[key: value, …, key: value]`.

2.6.5 Обработка ссылки в javadoc

В ссылке вида `@see method(String, double)` заменяется список типов и название на новые.

2.6.6. Делегирование

Иногда бывает удобно не менять целевой метод, а добавить новый, которому будет делегировать все действия старый:

```
public void foo(String s, String g) {
     System.out.println(s+g);
}
```

Листинг 15

После добавления параметра `String e` со значением по умолчанию `"a"` будет создан следующий код:

```
public void foo(String s, String g, String e) {
     System.out.println(s+g);
}
public void foo(String s, String g) { //старый
     foo(s, g, "a");                  //метод
}
```

Листинг 16

При этом будет проделаны следующие операции:

1. Будет создан новый метод с указанным рефакторингу списком параметров, именем, модификатором доступа и списком бросаемых исключений.
2. В новый метод будет перенесен весь код из старого.
3. В старый метод будет добавлен делегирующий вызов нового метода с передачей ему корректного списка аргументов.

2.6.7. Исключения

Если у метода не было ни одного исключения в сигнатуре, то все вызовы будут обернуты в блок try{…}. К блоку будут добавлены соответствующие блоки catch(…) для всех исключений, которые наследую класс Exception. Исключения, расширяющие Error, будут проигнорированы.

Если вызов уже обернут в блок try{…}, то блоки catch(…) будут добавлены к нему. При этом исключения, уже обработанные в других блоках catch(…), будут проигнорированы.

Заключение

В рамках курсовой работы был разработан кроссъязыковый рефакторинг «Изменение сигнатуры метода». Были поддержаны языки Java и Groovy. При этом была значительно переработана и переписана существовавшая поддержка рефакторинга для Java.

Были поддержаны практически все возможности, которые могут понадобиться в таком рефакторинге. Осталась совместимость со всеми рефакторингами, которые использовали «Изменение сигнатуры метода». Теперь они автоматически используют все возможности кроссъязыкового рефакторинга и обновляют код, написанный не только на Java, но и на Groovy. К таким рефакторингам относятся корректоры «Исправление списка параметров», «Исправление типа возвращаемого значения метода», «Удаление ненужного параметра», и т. п.

Результатом моей работы стало повышение производительности разработчиков, использующих Groovy, а также Groovy вместе с Java. Теперь они могут избежать рутинной работы по ручному изменению всех необходимых частей кода при изменении сигнатуры метода. Соответсвенно, значительно снизилась вероятность допустить при этом ошибку. Таким образом, были достигнуты все поставленные в рамках курсовой работы цели.

Возможные направления дальнейшей работы

- Поддержка других языков
  - Scala
  - Jython
  - JRuby
- Улучшение пользовательского интерфейса. Возможно, в будущем мы откажемся от диалога, и все действия будут производиться прямо в редакторе.
- Новые функциональные возможности рефакторинга
- Более совершенное предупреждение о возможных коллизиях
- Более корректный отбор ссылок, которые необходимо изменить
- Кроссъязыковая поддержка других рефакторингов

Список использованной литературы

1. Groovy language home page . – http://groovy.codehaus.org
2. IntelliJ IDEA help . – http://www.jetbrains.com/idea/webhelp/refactoring-source-code.html
3. J. Gosling, B. Joy, G. Steele, G. Bracha, “Method Declarations” . – //The Java Language Specification, 2005, p. 209-237. – : Addison-Wesley
4. M. Fauler, “Crossing Refactoring’s Rubicon” . – http://www.martinfowler.com/articles/refactoringRubicon.html
5. Refactoring . – http://en.wikipedia.org/wiki/Code_refactoring