Отражение при проектировании образа посетителя (Visitor pattern)

Реализуйте посетителей на Java, используя reflection.

Tips 'N Tricks

Джереми Блоссер

Образ посетителя(Visitor pattern) часто используется, чтобы отделить структуру коллекции объектов от действий, выполненных над этой коллекцией. Например, можно отделить логику синтаксического анализа в компиляторе от логики генерации объектного кода. Сохраняя это разделение, вы можете легко использовать различные генераторы объектного кода. Более того, другие утилиты типа *lint* могут использовать логику синтаксического анализа без передачи логики генерации объектного кода. К сожалению, добавление нового типы объекта к коллекции часто требует изменения классов visitor, которые уже были написаны. Эта статья представляет более гибкий подход к разработке посетителя (visitor) в языке Java, используя отражение. *(1600 слов)*

Коллекции `(Collections)` широко используются в объектно-ориентированном программировании и часто вызывают вопросы, связанные с кодом . Например,

" Как вы исполняете определенную операцию над коллекцией различных объектов? "

Один из подходов состоит в том, чтобы выполнить итерации по каждому элементу в коллекции и затем проделать что-то специфическое над каждым элементом, основываясь на его типе/классе. Это может быть довольно рискованным, особенно, если вы не знаете, какие объекты находятся в коллекции. Если вы захотели распечатать элементы в коллекции, вы могли бы написать метод подобно этому:

```
public void messyPrintCollection(Collection collection) {
 Iterator iterator = collection.iterator()
 while (iterator.hasNext())
```

```
 System.out.println(iterator.next().toString())
}
```

Это выглядит достаточно просто. вы только вызываете метод `Object.toString()` и распечатываете объект, правильно? Что если, например, вы имеете вектор хэш-таблицы? Тогда вещи становиться более сложными. вы должны проверить тип объекта, взятого из коллекции:

```
public void messyPrintCollection(Collection collection) {
 Iterator iterator = collection.iterator()
 while (iterator.hasNext()) {
 Object o = iterator.next();
 if (o instanceof Collection)
 messyPrintCollection((Collection)o);
 else
 System.out.println(o.toString());
 }
}
```

Хорошо, теперь вы можете обрабатывать и вложенные коллекции, но что делать относительно других объектов, которые не возвращают тип `String`, в котором вы нуждаетесь ? Что, если вы хотите добавить кавычки вокруг объектов типа `String` и добавлять букву f после объектов типа `Float`? Код становится все более сложным:

```
public void messyPrintCollection(Collection collection) {
 Iterator iterator = collection.iterator()
 while (iterator.hasNext()) {
 Object o = iterator.next();
 if (o instanceof Collection)
 messyPrintCollection((Collection)o);
 else if (o instanceof String)
 System.out.println("'"+o.toString()+"'");
 else if (o instanceof Float)
 System.out.println(o.toString()+"f");
 else
 System.out.println(o.toString());
 }
}
```

Ясно, что такие приемы могут становиться довольно быстро запутанными. вы же не хотите писать часть кода с огромным списком инструкций типа `if-else`! Как избежать этого? Образ посетителя(Visitor) появляется, чтобы спасти Вас.

Для реализации образа посетителя(Visitor) вы создаете интерфейс (interface) `Visitor` для посетителя (visitor), и интерфейс `Visitable` для коллекции, которая будет посещаться. Тогда вы имеете конкретный класс, который реализует интерфейсы `Visitable` и `Visitor`. Два интерфейса выглядят следующим образом:

```
public interface Visitor {
 public void visitCollection(Collection collection);
 public void visitString(String string);
 public void visitFloat(Float float);
}

public interface Visitable {
 public void accept(Visitor visitor);
}
```

Для конкретного типа `String`, вы могли бы записать:

```
public class VisitableString implements Visitable {
 private String value;
 public VisitableString(String string) {
 value = string;
 }
 public void accept(Visitor visitor) {
 visitor.visitString(this);
 }
}
```

В методе `accept()`, вы вызываете нужный метод посетителя (Visitor) для типа `this`:

```
Visitor.visitString(this);
```

Это позволяет вам реализовать конкретный образец посетителя (Visitor) следующим образом:

```
public class PrintVisitor implements Visitor {

 public void visitCollection(Collection collection) {
 Iterator iterator = collection.iterator()
 while (iterator.hasNext()) {
 Object o = iterator.next();
 if (o instanceof Visitable)
 ((Visitable)o).accept(this);
 }
```

```
 public void visitString(String string) {
 System.out.println("'"+string+"'");
 }


 public void visitFloat(Float float) {
 System.out.println(float.toString()+"f");
 }
}
```

Затем реализовав классы `VisitableFloat` и `VisitableCollection` таким образом, что каждый из них вызывает соответствующие методы посетителя (visitor), вы получает тот же самый результат как и в `if-else` методе `messyPrintCollection`, но с более ясным кодом. В методе `visitCollection()`, вы вызываете метод `Visitable.accept(this)`, который в свою очередь вызывает нужный метод посетителя(visitor). Это называется двойной отправкой (double-dispatch); `Visitor` вызывает метод в классе `Vistable`, который передает вызов обратно в `Visitor`.

Хотя вы избежали применения инструкции `if-else`, реализовав посетителя (visitor), вы все еще использовали много дополнительного кода. вы были должны завернуть (wrap) Ваши первоначальные объекты типа `String` и `Float` в объекты, реализующие интерфейс `Visitable`. Несмотря на неудобства, это обычно не является проблемой поскольку коллекции, которые вы обычно посещаете, могут быть созданы для того, чтобы содержать только объекты, которые реализуют интерфейс `Visitable`.

Однако, это требует дополнительной работы. Что может произойти, когда вы добавляете новый тип `Visitable`, скажем `VisitableInteger`? В этом один из главных недостатков образа `Visitor`. Если вы хотите добавить новый объект `Visitable`, вы должны изменить интерфейс `Visitor`, а затем реализовать данный метод в каждом из Ваших реализованных классов `Visitor`. Вы могли бы использовать абстрактный класс `Visitor` с заданными по умолчанию no--op функциями вместо интерфейса. Это было бы подобно классам `Adapter` в Java GUIs. Проблема с этим подходом заключается в том, что вам прийдется использовать свое единичное наследование, которое часто хочется сохранить для чего-нибудь другого,

например, расширения `StringWriter`. Это также ограничивает вашу возможность посещать объекты `Visitable`.

К счастью, язык Java позволяет вам сконструировать образ посетителя (Visitor) намного более гибким, так что вы можете добавлять объекты типа `Visitable` по желанию. Как? Ответ: используя отражение (reflection). В интерфейс `ReflectiveVisitor` вам необходимо добавить один метод:

```
public interface ReflectiveVisitor {
 public void visit(Object o);
}
```

Хорошо, на самом деле достаточно просто. Интерфейс `Visitable` может оставаться без изменений, а я начну изменения в исходном коде. Сейчас я буду реализовывать класс `PrintVisitor` с использованием отражения (reflection):

```
public class PrintVisitor implements ReflectiveVisitor {
 public void visitCollection(Collection collection)
 { ... без изменений ... }
 public void visitString(String string)
 { ... без изменений ... }
 public void visitFloat(Float float)
 { ... без изменений ... }


 public void default(Object o) {
 System.out.println(o.toString());
 }

 public void visit(Object o) {
// Class.getName () возвращает также информацию о пакете.
// this удаляет информацию о пакете полученную вами, оставляя
// Только имя класса
String methodName = o.getClass () .getName ();
MethodName = "visit" +
methodName.substring(methodName.lastIndexOf('.')+1);
// Теперь мы попробуем вызвать метод visit<methodName>
try {
// Получить метод visitFoo(Foo foo)
Method m = getClass().getMethod(methodName,
 new Class[] { o.getClass() });
```

```
 // Попытка вызвать visitFoo(Foo foo)
m.invoke(this, new Object[] { o });
 } catch (NoSuchMethodException e) {
 // Нет метода, делайте заданное по умолчанию выполнение
default(o);
 }
 }
}
```

Теперь вы не нуждаетесь в оберточном классе (wrapper class) `Visitable`. Вы можете вызывать метод `visit()`, а он передаст выполнение нужному методу. Одним хорошим аспектом этого является то, что метод `visit()` может передавать выполнение, если находит нужным. При этом не обязательно использовать отражение(reflection) — можно использовать полностью отличный механизм.

С новым классом `PrintVisitor` вы имеете методы для типов `Collections`, `Strings`, и `Floats`, и тогда вы захватываете все необрабатываемые типы в инструкции `catch`.

Вы расширяете метод `visit()` так, чтобы вы могли бы пробовать обрабатывать также все суперклассы. Сначала, вы добавите новый метод, имеющий название `getMethod(Class c)` и возвращающий вызывающий метод, который в свою очередь ищет метод соответствия для всех суперклассов класса `Class c`, а затем все интерфейсы для `Class c`.

```
protected Method getMethod(Class c) {
 Class newc = c;
 Method m = null;
 // Пробовать суперклассы
 while (m == null && newc != Object.class) {
 String method = newc.getName();
 method = "visit" + method.substring(method.lastIndexOf('.') + 1);
 try {
 m = getClass().getMethod(method, new Class[] {newc});
 } catch (NoSuchMethodException e) {
 newc = newc.getSuperclass();
 }
 }

 // Пробовать интерфейсы. В случае необходимости, вы
```

```
// Можете сортировать их сначала, чтобы определить интерфейс 'visitable'
// В случае, если объект реализуется больше одного раза.
if (newc == Object.class) {
 Class[] interfaces = c.getInterfaces();
 for (int i = 0; i < interfaces.length; i++) {
 String method = interfaces[i].getName();
 method = "visit" + method.substring(method.lastIndexOf('.') + 1);
 try {
 m = getClass().getMethod(method, new Class[] {interfaces[i]});
 } catch (NoSuchMethodException e) {}
 }
 }
 if (m == null) {
 try {
 m = thisclass.getMethod("visitObject", new Class[] {Object.class});
 } catch (Exception e) {
// Не может случаться!
}
 }
 return m;
}
```

Это выглядит сложным, но на самом деле это не так. В основном вы только ищете методы, основанные на названии(имени) класса, в котором вы прошли. Если не находите, вы пробуете его суперклассы. Тогда, если не находите среди них, вы пробуете любые интерфейсы. Наконец, вы можете только пробовать метод `visitObject()`, как значение по умолчанию.

Обратите внимание, что ради тех, кто знаком с традиционным образом посетителя (Visitor), я следовал теми же самыми соглашениями об именах для названий методов. Однако, некоторые из Вас, возможно заметили, что будет более эффективным назвать все методы "посещением" (visit) и позволять типу параметра быть переменным. Если вы проделаете это, удостоверьтесь, что вы заменяете имя основного метода `visit(Object o)` на подобное `dispatch(Object o)`. Иначе, вы не будете иметь заданного по умолчанию метода для возвращения, и вы должны будете приводить тип к типу `Object` всякий раз, когда вы вызываете метод `visit(Object o)`.

Теперь вы модифицируете метод `visit()`, чтобы воспользоваться преимуществом метода `getMethod()`:

```
public void visit(Object object) {
 try {
 Method method = getMethod(getClass(), object.getClass());
 method.invoke(this, new Object[] {object});
 } catch (Exception e) { }
}
```

Сейчас Ваш объект visitor намного усложнен. вы можете передавать любой произвольный объект и иметь некоторый метод, который обрабатывает этот обьект. Дополнительно, вы получаете выгоду из-за наличия заданного по умолчанию метода `visitObject(Object o)`, который может обработать что-нибудь из того, что вы не задаете. Немного работы и вы можете даже добавить метод для `visitNull()`.

Я сохранил интерфейс `Visitable` без изменений по определенной причине. Другое дополнительное преимущество традиционного образа посетителя (Visitor) — это то, что позволяет объектам `Visitable` управлять передвижением объектной структуры. Например, если вы имели объект `TreeNode`, который реализовал `Visitable`, вы могли иметь метод `accept()`, который перемещается по узлам слева направо:

```
public void accept(Visitor visitor) {
 visitor.visitTreeNode(this);
 visitor.visitTreeNode(leftsubtree);
 visitor.visitTreeNode(rightsubtree);
}
```

Так, что только с одной дополнительной модификацией к классу `Visitor` вы можете позволить Visitable-управляющую навигацию:

```
public void visit(Object object) throws Exception {
 Method method = getMethod(getClass(), object.getClass());
 method.invoke(this, new Object[] {object});
 if (object instanceof Visitable) {
 callAccept((Visitable) object);
 }
}

public void callAccept(Visitable visitable) {
 visitable.accept(this);_
}
```

Если вы реализовали структуру объекта Visitable, вы можете оставить метод `callAccept()` таким, как он есть и использовать Visitable-управляющую навигацию.

Если вы хотите управлять структурой в пределах посетителя, вы только переписываете метод `callAccept()` таким образом, что он не делает ничего.

Преимущество образа посетителя (Visitor) становится ощутимым при использовании нескольких различных посетителей на одной и той же коллекции объектов. Например, я имею интерпретатор, префиксную запись, постфиксную запись, `XML` программу записи и программу записи SQL, работающие над одной и той же коллекцией объектов. Я мог легко записывать префиксную запись или программу записи протокола SOAP(simple object access protocol) для той же самой коллекции объектов. Кроме того, такие программы записи могут хорошо работать с объектами, которых они не знают или по-моему выбору они могут генерировать исключение.

1. Выводы

Используя Java отражение (reflection), вы можете расширять образы проектирования Visitor, обеспечивая мощный способ работы на объектных структурах, создавая гибкость в добавлении новых типов Visitable в случае необходимости. Я надеюсь на то, что вы способны будете использовать этот образец в Вашем программировании.

2. Об авторе

Джереми Блоссер (Jeremy Blosser) программирует на языке Java пять лет, работая в различных компаниях по созданию программных продуктов. Сейчас он трудится в новой компании — Software Instruments. Вы можете посетить Веб страницу Джереми на http://www.blosser.org.

3. Ресурсы

- Patterns homepage: http://www.hillside.net/patterns/
- *Design Patterns: Elements of Reusable Object-Oriented Software,* Erich Gamma, et al. (Addison-Wesley, 1995): http://www.amazon.com/exec/obidos/ ASIN/0201633612/o/qid=963253562/sr=2-1/002-9334573-2800059
- *Patterns in Java, Volume 1,* Mark Grand (John Wiley & Sons, 1998):

http://www.amazon.com/exec/obidos/ ASIN/0471258393/o/qid=962224460/sr=2-1/104-2583450-5558345

- *Patterns in Java, Volume 2,* Mark Grand (John Wiley & Sons, 1999): http://www.amazon.com/exec/obidos/ ASIN/0471258415/qid=962224460/sr=1-4/104-2583450-5558345
- View all previous Java Tips and submit your own: http://www.javaworld.com/javatips/jw-javatips.index.html

View the original article at: http://www.javaworld.com/javaworld/ javatips/jw-javatip98.html