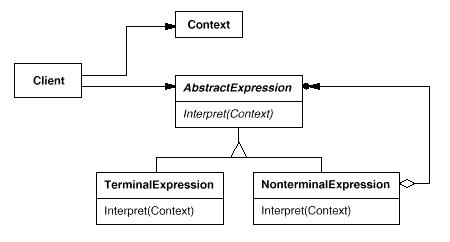
设计模式——解释器模式(Interpreter)

http://simg.sinajs.cn/blog7style/images/common/sg_trans.gif (2011-06-19 01:09:02)

如果一种特定类型的问题发生的频率足够高，那么就值得将该问题的各个实例表述为一个简单语言中的句子。

目的：按照自己定义的组合规则集合来组合可执行对象  
一、组成角色  
1、抽象表达式角色：声明一个抽象的解释操作，这个接口为所有具体表达式角色（抽象语法树中的节点）都要实现的。  
抽象语法树的每一个节点都代表一个语句，而在每个节点上都可以执行解释方法。这个解释方法的执行就代表这个语句被解释。由于每一个语句都代表这个语句被解释。由于每一个语句都代表一个常见的问题的实例，因此每一个节点上的解释操作都代表对一个问题实例的解答。  
2、终结符表达式角色：具体表达式。  
(1)实现与文法中的终结符相关联的解释操作  
(2)而且句子中的每个终结符需要该类的一个实例与之对应  
3、非终结符表达式角色：具体表达式。  
(1)文法中的每条规则R::=R1R2…Rn都需要一个非终结符表带式角色  
(2)对于从R1到Rn的每个符号都维护一个抽象表达式角色的实例变量  
(3)实现解释操作，解释一般要递归地调用表示从R1到Rn的那些对象的解释操作  
4、上下文（环境）角色：包含解释器之外的一些全局信息。  
5、客户角色：  
(1)构建（或者被给定）表示该文法定义的语言中的一个特定的句子的抽象语法树  
(2)调用解释操作  
二、类图

[](http://photo.blog.sina.com.cn/showpic.html#blogid=636415010100v9s2&url=http://s2.sinaimg.cn/orignal/636415014a5ff971ffc81)  
  
三、实现  
1、抽象表达式接口Expression  
package com.makocn.javapatterns.interpreter;  
public interface Expression {  
 public int interpret(Context con);  
}  
2、终结符表达式Constan和  
package com.makocn.javapatterns.interpreter;  
public class Constant implements Expression {  
 private int i;  
 public Constant(int i) {  
  this.i = i;  
 }  
 public int interpret(Context con) {  
  return i;  
 }  
}  
package com.makocn.javapatterns.interpreter;  
public class Variable implements Expression {  
 public int interpret(Context con) {  
  // this为调用interpret方法的Variable对象  
  return con.LookupValue(this);  
 }  
}  
3、非终结符表达式Add、Subtract、Multiply、Division  
package com.makocn.javapatterns.interpreter;  
public class Add implements Expression {  
 private Expression left, right;  
 public Add(Expression left, Expression right) {  
  this.left = left;  
  this.right = right;  
 }  
 public int interpret(Context con) {  
  try {  
   return left.interpret(con) / right.interpret(con);  
  } catch (ArithmeticException ae) {  
   System.out.println("被除数为0！");  
   return -11111;  
  }  
 }  
}  
package com.makocn.javapatterns.interpreter;  
public class Subtract implements Expression {  
 private Expression left, right;  
 public Subtract(Expression left, Expression right) {  
  this.left = left;  
  this.right = right;  
 }  
 public int interpret(Context con) {  
  return left.interpret(con) + right.interpret(con);  
 }  
}  
package com.makocn.javapatterns.interpreter;  
public class Multiply implements Expression {  
 private Expression left, right;  
 public Multiply(Expression left, Expression right) {  
  this.left = left;  
  this.right = right;  
 }  
 public int interpret(Context con) {  
  return left.interpret(con) - right.interpret(con);  
 }  
}  
package com.makocn.javapatterns.interpreter;  
public class Division implements Expression {  
 private Expression left, right;  
 public Division(Expression left, Expression right) {  
  this.left = left;  
  this.right = right;  
 }  
 public int interpret(Context con) {  
  return left.interpret(con) \* right.interpret(con);  
 }  
}  
4、上下文环境类Context  
package com.makocn.javapatterns.interpreter;  
import java.util.HashMap;  
import java.util.Map;  
public class Context {  
 private Map valueMap = new HashMap();  
 public void addValue(Variable x, int y) {  
  Integer yi = new Integer(y);  
  valueMap.put(x, yi);  
 }  
 public int LookupValue(Variable x) {  
  int i = ((Integer) valueMap.get(x)).intValue();  
  return i;  
 }  
}  
5、测试类  
package com.makocn.javapatterns.interpreter;  
public class InterpreterClient {  
 public static void main(String[] args) {  
  Context con = new Context();  
  // 设置变量、常量  
  Variable a = new Variable();  
  Variable b = new Variable();  
  Constant c = new Constant(2);  
  // 为变量赋值  
  con.addValue(a, 5);  
  con.addValue(b, 7);  
  // 运算，对句子的结构由我们自己来分析，构造;计算 (a\*b)/(a-b+2)  
  Expression ex = new Division(new Multiply(a, b), new Add(new Subtract(a, b), c));  
  System.out.println("运算结果为：" + ex.interpret(con));  
 }  
}  
解释器模式提供简单的方式来执行语法，且容易修改或扩展。一般系统中很多类使用相似语法，使用一个解释器来代替为每一个规则实现一个解释器。而且在解释器中不同的规则是由不同的类来实现的，这样使得添加一个新的语法规则变得简单。  
但是解释器模式对于复杂文法难以维护。每一个规则要对应一个处理类，且这些类还要递归调用抽象表达式角色，维护相对较复杂。  
四、与其他模式区别  
1、与Command命令模式区别  
(1)解释器模式的意图在于使开发人员可以组合可执行对象，这些对象来自于对某公共操作提供各种解释的类层次结构。而命令模式的意图仅仅是一个对象中请求。  
2)第二，解释器模式能否作为命令使用？当然可以，可以说它是命令模式的子模式。个体应该使用哪种设计模式，取决于你的意图。如果是期望创建组合可执行对象的工具集就可采用解释器模式，如果是采用请求封装在对象中就可采用命令模式。  
2、与Composite(组合模式)区别  
Interpreter模式也类似于组合模式，Composite模式通常会为单个对象和群组对象定义一个公共接口。不过，Composite模式并不要求支持不方式组织的结构，尽管该模式可以支持这些结构。而解释器模式通常会涉及不同类开的组合结构（所以说interpreter通常处于Composite模式之上）。