应用 AOP 技术实现 Observer 模式

郭丽丽,王清心,胡建华,丁家满
(昆明理工大学 信息工程与自动化学院,云南 昆明 650051)

摘 要:Observer 模式被人们广泛应用(特别是在 GUI 应用程序中),并构成了 MVC 架构的关键部分。它处理复杂的问题,而在解决这类问题方面表现得相对较好。但面向对象的 Observer 模式实现方案中,要求先侵入系统中现有的类,然后才能支持该模式,带来了以下问题:难以理解、可重用性差、后期维护很复杂、代码不容易组合等。基于 AOP 技术,提出了一种可重用的 Observer 模式的方案,并通过与面向对象实现方案的比较,得出了 AOP 技术实现方案具有模块松散耦合、可重用性高等优点的结论。

关键词:AOP;Observer Pattern;AspectJ
中图分类号:TP311 **文献标识码**:A **文章编号**:1673-629X(2007)06-0070-04

Implementation of Observer Pattern with AOP

GUO Li-li, WANG Qing-xin, HU Jian-hua, DING Jia-man
(Faculty of Information Engineering and Automation, Kunming University of Science and Technology, Kunming 650051, China)

Abstract: Observer pattern is the most common design pattern (especially in GUI application), and is the key part of the MVC model. However, there are some inherent flaws in the implementation of observer pattern with OOP, such as invade the class of the present system, then support this pattern, hard to understand, bad reusable, complex maintenance, not easy to composite. A reusable method to implement observer pattern based on AOP technology was presented. Compared with the object-oriented method, the advantages of AOP were addressed.

Key words: AOP; observer pattern; AspectJ

1 Observer 模式及其面向对象实现方案

1.1 Observer 模式及其面向对象实现方案

Observer 模式是由 GOF 在文献[1]中给出的一种设计模式。此模式的意图是:定义对象之间的一对多依赖关系,因此,当一个对象的状态发生改变时,其所有依赖项都会得到通知,并自动更新。此模式采取的方法是:为对象所扮演的每一个角色指定一个角色对象,并透明地、动态地将角色对象加载到这个对象上,从而实现了角色与对象动态组合。

1.1.1 Observer 模式的结构

Observer 模式的总体结构如图 1 所示[1],它由 4 种参与者组成:

(1)Subject 接口:目标知道它的观察者,可以有任意多个观察者观察同一个目标,提供注册和删除观察者对象的接口。

(2)Observer 接口:为那些在目标发生改变时需获得通知的对象定义一个更新接口。

(3)ConcreteSubject 类:将有关状态存入 ConcreteObserver 对象,当它的状态发生改变时,向它的各个观察者发出通知。

(4)ConcreteObserver 类:维护一个指向 ConcreteSubject 对象的引用,存储有关状态,这些状态应与目标的状态保持一致。实现 Observer 的更新接口以示自身状态和目标的状态保持一致。

1.1.2 协作关系

当 ConcreteSubject 发生任何可能导致其观察者与其本身状态不一致的改变时,它将通知它的各个观察者。在得到一个具体目标的改变通知后,ConcreteObserver 对象可向目标对象查询信息,ConcreteObserver 使用这些信息以使它的状态与目标对象的状态一致。

1.1.3 模式特性

Observer 模式的关键对象是目标(subject)和观察

收稿日期:2006-08-19
基金项目:联合国计划发展署 UNDP403 项目资助
作者简介:郭丽丽(1981-),女,江苏盐城人,硕士研究生,研究方向为 Web 技术与数据库;王清心,教授,硕士生导师,主要研究方向为 Web 及数据库技术。

者(observer),一个目标可以有任意数目的观察者依赖于它。一旦目标发生某些事件(例如状态发生变化),所有的观察者都将得到通知。随后观察者可以主动查询目标的状态,以保持系统各个部分状态的一致性。

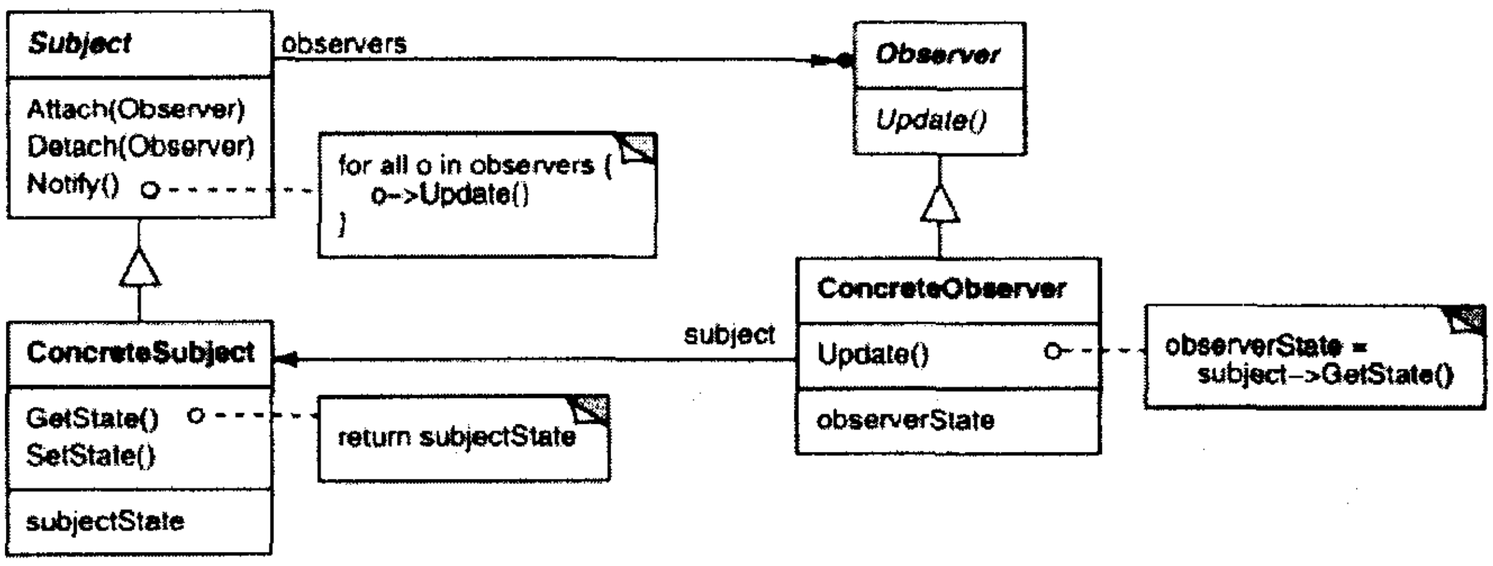

图 1　Observer 模式的总体结构

1.2　Observer 模式的面向对象实现方案的缺点

Observer 模式[2]的面向对象实现方案存在着一系列的问题:观察者(Observer)模式是一个重量级的模式,该模式对系统当前和潜在的冲击如下:

(1)混合与散乱:在 OO 编程中,模块化的自然单位是类,而横切关注点就是跨多个类的关注点。如一个设计模式行为要分解到三个不同的类,那么就说这个设计模式横切了这些类。横切会造成代码散布和代码混乱。这种散布和混乱造成很难对系统进行推断。

(2)可重用性差:模式应用程序分布在多个类中。将状态更改通知依赖对象的最容易的方法是直接调用它们。但是,对象之间的直接协作需要让它们的类建立相互的依赖性。例如,如果模型对象调用视图对象以便将更改通知它,则模型类现在也会依赖于视图类。两个对象之间的这种直接耦合(也称为紧耦合)降低了类的可重用性。要重用这个模式,必须从头开始重新实现它。

(3)维护:不管属于哪种情况的修改,都只在调用上下文环境得到保证的情况下才调用 notify()。这些修改可能会给已经饱受围困的模式参与者再添加了一些负担和复杂性。因为变化将影响多个文件,所以在重新设计的过程中就可能出现 bug。

(4)组合:需要考虑的另一个情况就是不同观察者的观察。因为 subject 必须为每种类型的 Observer 触发不同的事件。面对着另一种 Observer 类型,Java 语言的实现就会崩溃。

2　AOP 编程技术

传统的面向对象方法将软件的业务逻辑作为软件设计的主线,业务逻辑是软件系统的核心关注点。但除此之外,软件还包括系统级的关注点,如日志、事务完整性、授权、安全性及性能等。这些系统级的关注点,往往混合与散乱于根据核心关注点划分组成软件的结构单元中。这类混合与散乱于多个模块的关注点,被称为横切(crosscutting)关注点。横切关注点使得系统难以设计、理解、实现和演化。

AOP(面向方面编程)是一种能有效地管理、组织、分离横切关注点的程序设计方法学,是对 OOP 的一种补充。AOP 引入了一种新的模块单元——方面(aspect)来实现横切关注点。在构造系统时,再将实现横切关注点的方面与实现核心关注点的模块单元集成起来。这种集成过程被称为织入(weaving),由 AOP 程序的编译器完成。AOP 将横切关注点模块化,使得开发人员可以按照松散耦合的方式构造系统,也便于更好地维护和扩充软件系统。

AspectJ 是 AOP 在 Java 语言上的一种实现,是最成熟与最具代表性 AOP 语言工具。AspectJ 语言描述了两种性质的织入规则:静态横切与动态横切。静态横切与动态横切通过 AspectJ 所提供的语法规则,对方面的织入方式进行描述。它们之间的区别是:动态横切是将一个新的行为织入到程序的执行中,它会改变系统的行为。例如可以通过声明被织入的执行点、织入的行为与织入时序,使得织入行为可以在指定的执行点被执行之前、之后或之中执行。静态横切是将对静态结构(如类、接口与方面)的修改织入到系统中。静态横切主要用途是用于支持动态横切的实现。例如为一个类或接口引入动态横切行为会用到新的成员与函数。

3　基于 AOP 的 Observer 模式的实现方案

3.1　实现方案

基于 AspectJ[3],实现了下面可重用的类。同时安排好了模式的结构,然后由具体的方面定义模式应用到这个系统的方式。模块都是很实际的,而且从模式视角来看,模块也很完整。如下代码所示,为 ObserverPatterm 方面:

```
package lily. aspectPatterns. patternLibrary;
import java. util. WeakHashMap;
import java. util. List;
import java. util. LinkedList;
import java. util. Iterator;
public abstract aspect ObserverPattern{
/* *
 * This interface is used by extending aspects to say what types can be Subjects. It models the Subject role.
```

```
*/
protected interface Subject { }
/**
* This interface is used by extending aspects to say what types can be Observers. It models the Observer role.
*/
protected interface Observer { }
/**
* Stores the mapping between Subjects and Observers. For each Subject, a LinkedList is of its Observers is stored.
*/
private WeakHashMap perSubjectObservers;
/**
* Returns a Collection of the Observers of a particular subject. Used internally.
* @param subject the subject for which to return the Observers
*. @return a Collection of s's Observers
*/
protected List getObservers(Subject subject) {
if (perSubjectObservers == null) {
perSubjectObservers = new WeakHashMap();
}
List observers = (List)perSubjectObservers.get(subject);
if ( observers == null ) {
observers = new LinkedList();
perSubjectObservers.put(subject, observers);
}
return observers;
}
/**
* Adds an Observer to a Subject. This is the equivalent of attach(), but is a method on the pattern aspect, not the Subject.
* @param s the Subject to attach a new Observer to
* @param o the new Observer to attach
*/
public void addObserver(Subject subject, Observer observer) {
getObservers(subject).add(observer);
}
/**
* Removes an observer from a Subject. This is the equivalent of detach(), but is a method on the pattern aspect, not the Subject.
* @param s the Subject to remove the Observerfrom
* @param o the Observer to remove
*/
public void removeObserver(Subject subject, Observer observer) {
getObservers(subject).remove(observer);
}
/**
* The join points after which to do the update.
* It replaces the normally scattered calls to notify(). To be concretized by sub-aspects.
*/
protected abstract pointcut subjectChange(Subject s);
/**
* Calls updateObserver(..) after a change of interest to
* update each Observer.
* @param subject the Subject on which the change occured
*/
after(Subject subject): subjectChange(subject) {
Iterator iter = getObservers(subject).iterator();
while ( iter.hasNext() ) {
updateObserver(subject, ((Observer)iter.next()));
}
}
/**
* Defines how each Observer is to be updated when a change to a Subject occurs. To be concretized by sub-aspects.
* @param subject the Subject on which a change of interest occured
* @param observer the Observer to be notifed of the change
*/
protected abstract void updateObserver(Subject subject, Observer observer);
}
```

3.2 比较分析

经比较可以得出,AOP 的实现方案可以有效地解决 OO 实现方案中存在的问题[4],并具有以下优点:

(1)易于理解性:从参与者的视角来看,AOP 版本的观察者更简单,因为它不需要基本类有什么特殊的东西。从参与者那里消除了这种依赖性,从主要抽象角度来说,让它们更容易理解,同时也使它们更加灵活、更具重用性。例如,它们可以重新用在其他不要求观察者(Observer)模式的系统中。它们也可以参与到其他模式中,而不用担心大量的模式代码会淹没类的核心职责。

从模式的视角来看,AOP 版本还是更简单一些。AspectJ 实现不要求把模式片断汇集在头脑中,而是提供了一个 Observerpattern 方面,安排好了模式的结构。然后由具体的方面定义模式应用到这个系统的方式。

通知"事件"的生成在 AspectJ 实现中获得很好的模块化。在传统实现中,这些调用存在于几个不同的位置。不存在能说明为什么某个操作会触发事件,或者其他哪个操作也会这么做的直接线索。在 AOP 模式中,命名切入点沟通了操作的公共属性(标题的用法)。切入点规范指出了模式影响的其他连接点。

(2)可重用性:Observerpattern 方面的重用潜力非常高。方面提供了重用横切代码的能力,这些代码以前纠缠在实现的细节中,根本不是独立存在的。

(3)可维护性:这样做让系统的可扩展性增强,要针对某类扩展,只需要编辑 subjectChange() 切入点即可。要修改一个类,工作量比较小,特别是因为它只影响一个文件。而且,它还让新策略的意图变得很清楚。其他修改,例如添加新 Subjects 或主角变化操作,都只需像这样对方面进行简单的修改即可,不需要协调多个文件的修改。

(4)组合:在与应用到(某些)相同的参与者的第二个模式协作时,Java 语言实现的观察者也存在问题。要用 AspectJ 实现这个需求,只要再次用新的切入点和抽象方法定义扩展抽象方面即可。由于每个方面都管理自己的 Observer 列表和 Subject 列表,而且方面也定义了自己的 subjectChange 切入点,所以两个方面不会冲突。每个方面操作时实际上都独立于另一个方面。因为在 AspectJ 系统中,多个模式(甚至同一模式的多个实例)可以透明地组合在一起,所以 AspectJ 系统避免了来自"模式密度"的一些问题。这就能够更多地使用设计模式所拥有的最佳实践,而不用担心实现的重量会压跨代码。

4　结束语

利用 AOP 技术实现 Observer 模式表明,使用 AOP 可以更好处理模式中横切关注点。AOP 方案将业务逻辑的关键概念与模式行为相互分离,混合与散乱于代码中的不同逻辑接口相互分离,降低系统中业务逻辑与横切关注点的耦合性。利用 AOP 技术[5],可以更好地克服使用强类型的静态 OOP 语言引起的接口纠缠与设计困难的问题,使软件系统具有更好的可扩展与可维护性。

参考文献:

[1] Gamma E,Helm R,Johnson R,et al. 设计模式:可复用面向对象软件的基础[M]. 北京:机械工业出版社,2002:194-201.

[2] 透　明 .Observer 模式实战解析[J]. 程序员杂志,2004(4):95-96.

[3] Gradecki J D,Lesiecki N. 精通 AspectJ[M]. 王欣轩,吴东升译 . 北京:清华大学出版社,2005:69-115.

[4] Laddad R V. 使用 AspectJ 描述现实问题里的横切关注点[J]. 程序员杂志,2002(11):64-67.

[5] 张广红,陈　平. 关于 AOP 实现机制和应用的研究[J]. 计算机工程与设计,2003,24:14-16.

(上接第 69 页)

设计这样一个复杂的渐进过程。

2　结束语

提出了一个整合设计及绘图过程的笔式 CAD 系统设计思想,致力于将概念设计过程与详细设计过程整合在一起。从这个目的出发进行深入广泛的研究,阐述了目前可用于进行系统实现的原理及方法,以及系统实现所面临的挑战。主要从以下几点进行阐述:

1)手势设计的复杂性以及可扩展手势集的设计;

2)用以支持概念设计的 CAD 系统需要进行草图的概念识别;

3)笔式交互模糊性特点使得交互界面设计尤为困难。

参考文献:

[1] Gross M D, Yi-Luen Do E. Ambiguous Intentions:a Paper-like Interface for Creative Design[C]// ACM O-8979 1-798-7/96/11. UIST'96 Seattle. Washington, USA:[s. n.], 1996.

[2] Citrin W, Halbert D, Hewitt C, et al. Potentials and Limitations of Pen-Based Computers[M/OL]. New York, NY, USA: ACM Press, 1993. http://portal.acm.org/citation.cfm? id=171171.

[3] Hong J I, Landy J A. SATIN: A Toolkit for Informal Ink-based Applications[C]// ACM 1-58113-212-3/00/11, UIST'00. San Diego, CA, USA:[s. n.],2000.

[4] Suwa M, Tversky B. What Architects See in Their Sketches: Implications for Design Tools[C]// CHI '96 Companion. Vancouver, BC, Canada:[s. n.], 1996.

[5] 马翠霞. 支概念设计的手势描述和草图设计系统的研究[D]. 北京:中国科学院软件研究所,2003.

[6] Long A C,Jr James A L, Rowe L A, et al. Visual Similarity of Pen Gestures[C]// ACM 2000 1-58113-216-6/00/04. CHI'2000. The Hague, Amsterdam:[s. n.], 2000.

[7] Quill C L. A Gesture Design Tool for Pen-based User Interface[D]. Berkeley: University of California,2001.

[8] Forbus K D. Towards a Computational Model of Sketching[C]// IUI'01. Santa Fe, New Mexico, USA:[s. n.],2001.

[9] 宋保华,叶　军,于明玖,等. 笔输入草图的分层识别[J]. 计算机辅助设计与图形学学报,2004,16(6):753-754.

[10] Li Yang, Hinckley K, Guan Zhiwei, et al. Experimental Analysis of Mode Switching Techniques in Pen-based User Interfaces[C]// CHI 2005. Portland, Oregon, USA:[s. n.], 2005.