观察者模式

维基百科，自由的百科全书

本条目**需要补充更多来源**。（*2013年3月21日*）
请协助添加多方面可靠来源以改善这篇条目，无法查证的内容可能会被提出异议而移除。

观察者模式是软件设计模式的一种。在此种模式中，一个目标对象管理所有相依于它的观察者对象，并且在它本身的状态改变时主动发出通知。这通常透过呼叫各观察者所提供的方法来实现。此种模式通常被用来实时事件处理系统。

目录

结构

参与类别

参与本模式的各类别列出如下。成员函式以模拟的方式列出。

抽象目标类别

此抽象类别提供一个界面让观察者进行添附与解附作业。此类别内有个不公开的观察者串炼，并透过下列函式(方法)进行作业

- 添附(Attach)：新增观察者到串炼内，以追踪目标对象的变化。
- 解附(Detach)：将已经存在的观察者从串炼中移除。
- 通知(Notify)：利用观察者所提供的更新函式来通知此目标已经产生变化。

添附函式包涵了一个观察者对象参数。也许是观察者类别的虚拟函式(即更新函式)，或是在非面向对象的设定中所使用的函式指标(更广泛来讲，函式子或是函式对象)。

目标类别

此类别提供了观察者欲追踪的状态。也利用其源类别(例如前述的抽象目标类别)所提供的方法,来通知所有的观察者其状态已经更新。此类别拥有以下函式

- 取得状态(GetState)：回传该目标对象的状态。

抽象观察者界面

抽象观察者类别是一个必须被实做的抽象类别。这个类别定义了所有观察者都拥有的更新用界面，此界面是用来接收目标类别所发出的更新通知。此类别含有以下函式

- 更新(Update)：会被实做的一个抽象(虚拟)函式。

观察者类别

这个类别含有指向目标类别的参考(reference)，以接收来自目标类别的更新状态。此类别含有以下函式

- 更新(Update)：是前述抽象函式的实做。当这个函式被目标对象呼叫时，观察者对象将会呼叫目标对象的取得状态函式，来其所拥有的更新目标对象资讯。

每个观察者类别都要实做它自己的更新函式，以应对状态更新的情形。

当目标对象改变时，会通过呼叫它自己的通知函式来将通知送给每一个观察者对象，这个通知函式则会去呼叫已经添附在串炼内的观察者更新函式。通知与更新函式可能会有一些参数，好指明是目前目标对象内的何种改变。这么作将可增进观察者的效率(只更新那些改变部分的状态)。

用途

- 当抽象个体有两个互相依赖的层面时。封装这些层面在单独的对象内将可允许程序员单独地去变更与重复使用这些对象，而不会产生两者之间交互的问题。
- 当其中一个对象的变更会影响其他对象，却又不知道多少对象必须被同时变更时。
- 当对象应该有能力通知其他对象，又不应该知道其他对象的实做细节时。

观察者模式通常与 MVC 范式有关系。在 MVC 中，观察者模式被用来降低 model 与 view 的耦合程度。一般而言， model 的改变会触发通知其他身为观察者的 model 。而这些 model 实际上是 view 。 Java Swing 就是个范例，示意了 model 预期会透过 PropertyChangeNotification 架构以送出改变的通知给其他 view 。 Model 类别是 Java bean 类别的一员，并拥有与上述目标类别同样的行为。 View 类别则系结了一些 GUI 中的可视元素，并拥有与上述观察者类别同样的行为。当应用程序在执行时。使用者将因 view 做出相应的更新而看见 model 所产生的变更。

示例

Python

```
class AbstractSubject(object):
    def register(self, listener):
        raise NotImplementedError("Must subclass me")

    def deregister(self, listener):
        raise NotImplementedError("Must subclass me")

    def notify_listeners(self, event):
        raise NotImplementedError("Must subclass me")
 class Listener(object):
    def __init__(self, name, subject):
        self.name = name
        subject.register(self)

    def notify(self, event):
        print self.name, "received event", event
 class Subject(AbstractSubject):
    def __init__(self):
        self.listeners = []
        self.data = None

    def getUserAction(self):
        self.data = raw_input('Enter something to do:')
        return self.data

    # Implement abstract Class AbstractSubject

    def register(self, listener):
        self.listeners.append(listener)

    def deregister(self, listener):
        self.listeners.remove(listener)

    def notify_listeners(self, event):
        for listener in self.listeners:
            listener.notify(event)

 if __name__=="__main__":
    # make a subject object to spy on
    subject = Subject()

    # register two listeners to monitor it.
    listenerA = Listener("<listener A>", subject)
    listenerB = Listener("<listener B>", subject)
    # simulated event
    subject.notify_listeners ("<event 1>")
    # outputs:
    #     <listener A> received event <event 1>
    #     <listener B> received event <event 1>

    action = subject.getUserAction()
    subject.notify_listeners(action)
    #Enter something to do:hello
    # outputs:
    #     <listener A> received event hello
    #     <listener B> received event hello
```

C#

C#和其他使用[.NET Framework](#) 的语言一般无需使用接口和类实现典型的观察者模式，但是这里依然给一个例子。

```
using System;using System.Collections;
namespace Wikipedia.Patterns.Strategy{
        // IObserver --> interface for the observer
        public interface IObserver
        {
                // called by the subject to update the observer of any
change
                // The method parameters can be modified to fit certain
criteria
                void Update(string message);
        }

        public class Subject
        {
                // use array list implementation for collection of
observers
                private ArrayList observers;

                // constructor
                public Subject()
                {
                        observers = new ArrayList();
                }

                public void Register(IObserver observer)
                {
                        // if list does not contain observer, add
                        if (!observers.Contains(observer))
                        {
                                observers.Add(observer);
                        }
                }

                public void Deregister(IObserver observer)
                {
                        // if observer is in the list, remove
                        if (observers.Contains(observer))
                        {
                                observers.Remove(observer);
                        }
                }

                public void Notify(string message)
                {
                        // call update method for every observer
                        foreach (IObserver observer in observers)
                        {
                                observer.Update(message);
                        }
                }
        }

        // Observer1 --> Implements the IObserver
        public class Observer1 : IObserver
        {
                public void Update(string message)
                {
                        Console.WriteLine("Observer1:" + message);
                }
        }

        // Observer2 --> Implements the IObserver
        public class Observer2 : IObserver
        {
                public void Update(string message)
                {
                        Console.WriteLine("Observer2:" + message);
                }
        }
```

```
    // Test class
    public class ObserverTester
    {           [STAThread]
        public static void Main()
        {
            Subject mySubject = new Subject();
            IObserver myObserver1 = new Observer1();
            IObserver myObserver2 = new Observer2();

            // register observers
            mySubject.Register(myObserver1);
            mySubject.Register(myObserver2);

            mySubject.Notify("message 1");
            mySubject.Notify("message 2");
        }
    }}
```

C++

```
#include <list>#include <vector>#include <algorithm>#include
<iostream>using namespace std;
// The Abstract Observerclass ObserverBoardInterface{public:
    virtual void update(float a,float b,float c) = 0;};
// Abstract Interface for Displaysclass DisplayBoardInterface{public:
    virtual void show() = 0;};
// The Abstract Subjectclass WeatherDataInterface{public:
    virtual void registerob(ObserverBoardInterface* ob) = 0;
    virtual void removeob(ObserverBoardInterface* ob) = 0;
    virtual void notifyOb() = 0;};
// The Concrete Subjectclass ParaWeatherData: public
WeatherDataInterface{public:
    void SensorDataChange(float a,float b,float c)
    {
        m_humidity = a;
        m_temperature = b;
        m_pressure = c;
        notifyOb();
    }

    void registerob(ObserverBoardInterface* ob)
    {
        m_obs.push_back(ob);
    }

    void removeob(ObserverBoardInterface* ob)
    {
        m_obs.remove(ob);
    }protected:
    void notifyOb()
    {
        list<ObserverBoardInterface*>::iterator pos = m_obs.begin();
        while (pos != m_obs.end())
        {

((ObserverBoardInterface* )(*pos))->update(m_humidity,m_temperature,m
_pressure);
            (dynamic_cast<DisplayBoardInterface*>(*pos))->show();
            ++pos;
        }
    }
private:
    float       m_humidity;
    float       m_temperature;
    float       m_pressure;
    list<ObserverBoardInterface* > m_obs;};
// A Concrete Observerclass CurrentConditionBoard : public
ObserverBoardInterface, public DisplayBoardInterface{public:
    CurrentConditionBoard(WeatherDataInterface& a):m_data(a)
    {
        m_data.registerob(this);
    }
    void show()
    {
        cout<<"_____CurrentConditionBoard_____"<<endl;
        cout<<"humidity: "<<m_h<<endl;
        cout<<"temperature: "<<m_t<<endl;
        cout<<"pressure: "<<m_p<<endl;
        cout<<"_______________________________"<<endl;
    }

    void update(float h, float t, float p)
    {
        m_h = h;
        m_t = t;
        m_p = p;
    }
private:
    float m_h;
    float m_t;
    float m_p;
    WeatherDataInterface& m_data;};
// A Concrete Observerclass StatisticBoard : public
ObserverBoardInterface, public DisplayBoardInterface{public:
    StatisticBoard(WeatherDataInterface&
a):m_maxt(-1000),m_mint(1000),m_avet(0),m_count(0),m_data(a)
    {
        m_data.registerob(this);
    }

    void show()
    {
        cout<<"________StatisticBoard_________"<<endl;
        cout<<"lowest  temperature: "<<m_mint<<endl;
        cout<<"highest temperature: "<<m_maxt<<endl;
        cout<<"average temperature: "<<m_avet<<endl;
        cout<<"_______________________________"<<endl;
    }

    void update(float h, float t, float p)
    {
        ++m_count;
        if (t>m_maxt)
        {
            m_maxt = t;
        }
        if (t<m_mint)
        {
            m_mint = t;
        }
        m_avet = (m_avet * (m_count-1) + t)/m_count;
    }
private:
    float m_maxt;
    float  m_mint;
    float m_avet;
    int m_count;
    WeatherDataInterface& m_data;};

int main(int argc, char *argv[]){
    ParaWeatherData * wdata = new ParaWeatherData;
    CurrentConditionBoard* currentB = new 
CurrentConditionBoard(*wdata);
    StatisticBoard* statisticB = new StatisticBoard(*wdata);

    wdata->SensorDataChange(10.2, 28.2, 1001);
    wdata->SensorDataChange(12, 30.12, 1003);
    wdata->SensorDataChange(10.2, 26, 806);
    wdata->SensorDataChange(10.3, 35.9, 900);

    wdata->removeob(currentB);

    wdata->SensorDataChange(100, 40, 1900);

    delete statisticB;
    delete currentB;
    delete wdata;

    return 0;}
```

PHP

class STUDENT

```
<?php
class Student implements SplObserver{
    protected $tipo = "Student";
    private $nome;
    private $endereco;
    private $telefone;
    private $email;
    private $_classes = array();

    public function GET_tipo() {
        return $this->tipo;
    }

    public function GET_nome() {
        return $this->nome;
    }

    public function GET_email() {
        return $this->email;
    }

    public function GET_telefone() {
        return $this->nome;
    }

    function __construct($nome) {
        $this->nome = $nome;
    }

    public function update(SplSubject $object){
        $object->SET_log("Comes from ".$this->nome.": I'm a student of
".$object->GET_materia());
    }
}
?>
```

class TEACHER

```
<?php
class Teacher implements SplObserver{
    protected $tipo = "Teacher";
    private $nome;
    private $endereco;
    private $telefone;
    private $email;
    private $_classes = array();

    public function GET_tipo() {
        return $this->tipo;
    }

    public function GET_nome() {
        return $this->nome;
    }

    public function GET_email() {
        return $this->email;
    }

    public function GET_telefone() {
        return $this->nome;
    }
    function __construct($nome) {
        $this->nome = $nome;
    }

    public function update(SplSubject $object){
        $object->SET_log("Comes from ".$this->nome.": I teach in
".$object->GET_materia());
    }}
?>
```

Class SUBJECT

```
<?php
class Subject implements SplSubject {
    private $nome_materia;
    private $_observadores = array();
    private $_log = array();

    public function GET_materia() {
        return $this->nome_materia;
    }

    function SET_log($valor) {
        $this->_log[] = $valor ;
    }
    function GET_log() {
        return $this->_log;
    }

    function __construct($nome) {
        $this->nome_materia = $nome;
        $this->_log[] = " Subject $nome was included";
    }
    /* Adiciona um observador */
    public function attach(SplObserver $classes) {
        $this->_classes[] = $classes;
        $this->_log[] = " The ".$classes->GET_tipo()."
".$classes->GET_nome()." was included";
    }

    /* Remove um observador */
    public function detach(SplObserver $classes) {
        foreach ($this->_classes as $key => $obj) {
            if ($obj == $classes) {
                unset($this->_classes[$key]);
                $this->_log[] = " The ".$classes->GET_tipo()."
".$classes->GET_nome()." was removed";
            }
        }
    }

    /* Notifica os observadores */
    public function notify(){
        foreach ($this->_classes as $classes) {
            $classes->update($this);
        }
    }}?>
```

Application

```
<?phprequire_once("teacher.class.php");require_once("student.class.ph
p");require_once("subject.class.php");
$subject = new Subject("Math");$marcus = new Teacher("Marcus
Brasizza");$rafael = new Student("Rafael");$vinicius = new
Student("Vinicius");
// Include observers in the math
Subject$subject->attach($rafael);$subject->attach($vinicius);$subject
->attach($marcus);
$subject2 = new Subject("English");$renato = new
Teacher("Renato");$fabio = new Student("Fabio");$tiago = new
Student("tiago");
// Include observers in the english
Subject$subject2->attach($renato);$subject2->attach($vinicius);$subje
ct2->attach($fabio);$subject2->attach($tiago);
// Remove the instance "Rafael from subject"$subject->detach($rafael);
// Notify both subjects$subject->notify();$subject2->notify();
echo "First Subject <br />";echo
"<pre>";print_r($subject->GET_log());echo "</pre>";echo "<hr>";echo
"Second Subject <br />";echo "<pre>";print_r($subject2->GET_log());echo
"</pre>";?>
```

OUTPUT

First Subject

Array (

```
   [0] =>  Subject Math was included
   [1] =>  The Student Rafael was included
```

```
    [2] =>  The Student Vinicius was included
    [3] =>  The Teacher Marcus Brasizza was included
    [4] =>  The Student Rafael was removed
    [5] => Comes from Vinicius: I'm a student of Math
    [6] => Comes from Marcus Brasizza: I teach in Math

)
```

Second Subject

```
Array (

    [0] =>  Subject English was included
    [1] =>  The Teacher Renato was included
    [2] =>  The Student Vinicius was included
    [3] =>  The Student Fabio was included
    [4] =>  The Student tiago was included
    [5] => Comes from Renato: I teach in English
    [6] => Comes from Vinicius: I'm a student of English
    [7] => Comes from Fabio: I'm a student of English
    [8] => Comes from tiago: I'm a student of English

)
```

亦可参考 http://www.javaworld.com/javaworld/javaqa/2001-05/04-qa-0525-observer.html

[隐藏]

- 查
- 论
- 编

软件设计模式

Gang of Four	创建型	• 抽象工厂 • 生成器

设计模式

- 工厂方法
- 原型
- 单例

结构型

- 适配器
- 桥接
- Composite
- 修饰
- 外观
- 享元
- 代理

行为型

- 责任链
- 命令
- Interpreter
- 迭代器
- Mediator
- Memento
- 观察者
- State
- 策略
- 模板方法
- 访问者

并行模式

- Active object
- Balking
- Binding properties
- 双重检查锁定模式
- 异步方法调用
- Guarded suspension
- Join
- 锁
- 监视器
- Proactor
- 反应器
- Read write lock
- 调度
- 线程池
- Thread-local storage

架构模式	• Front controller • Interceptor • MVC • *n*-tier • Specification • 发布/订阅 • Naked objects • Service locator • Active Record • Identity map • Data access object • Data transfer object • ADR • 控制反转
其他模式	• Blackboard • Business delegate • Composite entity • 依赖注入 • Intercepting filter • 惰性载入 • 模拟对象 • 空对象 • 对象池 • Servant • Twin • Type tunnel

导航菜单

- 打印版本
- 固定链接
- 页面信息
- 维基数据项
- 引用本页
- 左侧跳顶连接

其他语言

- العربية
- Български
- Català
- Čeština
- Deutsch
- English
- Español
- Français
- Galego
- עברית
- Magyar
- Italiano
- 日本語
- 한국어
- Nederlands
- Polski
- Português
- Русский
- Svenska
-
- Українська

编辑链接