**The Publisher-Subscriber Pattern**

Publishers are also called senders, observables, subjects, broadcasters, and notifiers. Subscribers are also called receivers, listeners, observers and callbacks.

**Problem**

Various monitors need to be notified when a device changes state, but the number and types of monitors can vary dynamically. We want to avoid polling, and we don't want to make assumptions in the device code about the numbers and types of monitors.

**Solution**

The device should maintain a list of pointers to registered monitors. Monitors can be different, but each must implement the same interface, which includes an update() function that the device will call when it changes state. The list of monitor pointers can be managed by a reusable **Publisher** base class. An abstract **Subscriber** base class defines the interface monitors must implement.

**Structure**

A screenshot of a computer screen

Description automatically generated

Subscriber is an abstract class because update() is a pure virtual function that will be implemented differently in different derived classes. Alternatively, we could have stereotyped Subscriber as an interface.

**Behavior**

Assume two monitors subscribe to a device. When the device changes state, monitor1.f() and monitor2.g() must be called:

A computer screen with white rectangles

Description automatically generated

**Discussion**

Pattern catalog entries don't usually include implementations, because they are meant to be language independent. It is the programmer's job to provide the implementation. Some patterns are so useful, they are provided in well known libraries. For example, publisher and subscriber classes are provided in the Java utility package, where they are called Observable and Observer. Specializations of publisher and subscriber classes are also provided in the Microsoft Foundation Class library, where they are called CDocument and CView, and IBM's Visual Age library, where they are called Notifier and Subscriber. Unfortunately, the standard C++ library doesn't include Publisher and Subscriber, so we'll have to implement them ourselves. There are many approaches, we follow one loosely based on the Java implementation.

**pubsub.h**

We place the publisher and subscriber class declarations in a header file called pubsub.h. Here is the structure of the file:

/\*  
�\* File:����������� pop\util\pubsub.h  
�\* Programmer:����� Pearce  
�\* Copyright (c):�� 2000, all rights reserved.  
�\*/  
#ifndef PUBSUB\_H  
#define PUBSUB\_H  
#include <list>� // for our subscriber list  
using namespace std;  
class Publisher; // forward reference  
  
class Subscriber { ... };  
class Publisher { ... };  
  
#endif

In addition to a do-nothing virtual destructor, the subscriber interface contains a pure virtual update function. To allow for the possibility that a subscriber might subscribe to multiple publishers, we pass a pointer to the notifying publisher. To allow messages to be sent from the publisher, we provide an optional generic "message" pointer:

class Subscriber  
{  
public:  
� virtual ~Subscriber() {}  
� virtual void update(Publisher\* who, void\* what = 0) = 0;  
};

A publisher maintains a list of pointers to subscribers. Monitors implementing the subscriber interface can add themselves to the list using subscribe(), and remove themselves using unsubscribe(). Devices can call the update() function of each registered monitor by calling notify(). A client may need to change the state of a device without notifying the monitors. This can be done by first setting the notifyEnabled flag to false.

class Publisher  
{  
public:  
�� Publisher() { notifyEnabled = true; }  
�� virtual ~Publisher() {}  
�� void subscribe(Subscriber\* s) { subscribers.push\_back(s); }  
�� void unsubscribe(Subscriber\* s) { subscribers.remove(s); }  
�� void notify(void\* what = 0, Subscriber \*s = 0);  
�� void setNotifyEnabled(bool flag) { notifyEnabled = flag; }  
�� bool getNotifyEnabled() const { return notifyEnabled; }  
private:  
�� list<Subscriber\*> subscribers;  
�� bool notifyEnabled;  
};

**pubsub.cpp**

Including the implementation of notify() in the Publisher class declaration would make it inline, but some compilers won't allow inline functions containing iterative statements. Therefore, we move the implementation to a file called pubsub.cpp. We must not forget to include pubsub.h in this file:

/\*  
�\* File:����������� pop\util\pubsub.cpp  
�\* Programmer:����� Pearce  
�\* Copyright (c):�� 2000, all rights reserved.  
�\*/  
#include "pubsub.h"

Here is our implementation of notify(). Notice that notifyEnabled is automatically set to true at the end of the function. This protects against clients who may forget to set the flag back to true after setting it to false. The notify function has two parameters. The first parameter is an optional, generic message. The second parameter points to a subscriber that is not to be notified. This feature is useful when a subscriber wants to send a message to its fellow subscribers through the publisher. In this case the sender calls the notify function passing a message and its address. Every subscriber receives the message except the sender. Recall that both parameters have the null pointer as a default argument.

void Publisher::notify(void\* what, Subscriber\* s)  
{  
�� if (notifyEnabled)  
�� {  
����� list<Subscriber\*>::iterator p;  
����� for(p = subscribers.begin(); p != subscribers.end(); p++)  
�������� if (\*p != s) (\*p)->update(this, what);  
�� }  
�� notifyEnabled = true;  
}