

Simplifying C++ with Notifiers

by Dave Pomerantz

For the last seven years, I've been using notifiers to simplify software designs and I've come to believe they're an essential part of object-oriented programming. Notifiers, also called events or messages, are used to pass information anonymously between objects. They're integral to Java, Taligent¹, and Smalltalk, but they're curiously absent from C++.

Notifiers connect objects indirectly, replacing pointers and direct function calls. Because they're anonymous, notifiers reveal nothing about the implementation, interface, or even the existence of connected objects, leaving them independent of one another. By reducing dependencies they reduce complexity. Most important, notifiers are easy to understand and to use, so they continue to be used effectively as new people replace the original developers.

In this article I'll show how notifiers can work in C++ and demonstrate them with a simple multi-threaded application. I'll take the application through three releases to show what happens as objects and notifiers are added. Lastly, I'll compare notifiers with pointers, with COM events, and with Windows messages.

Although my own notifier implementation is a C++ Windows DLL, the source code could easily be ported to any platform and could be written in any object-oriented language. If you're a fan of design patterns, you'll recognize notifiers as an example of the *Observer* pattern, with some important differences which are explained further on.

¹ An application framework developed as a joint effort of Apple and IBM.

How notifiers work

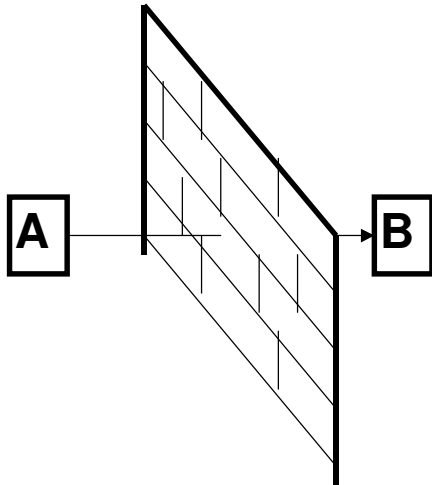


Figure 1: Good fences make good neighbors.

Notifiers are the minimum interface necessary for communication. In Figure 1, for A to send a notifier to B, it shouldn't need to include B's header or even know if B exists. That way, A and B depend only on the notifier interface, not on each other.

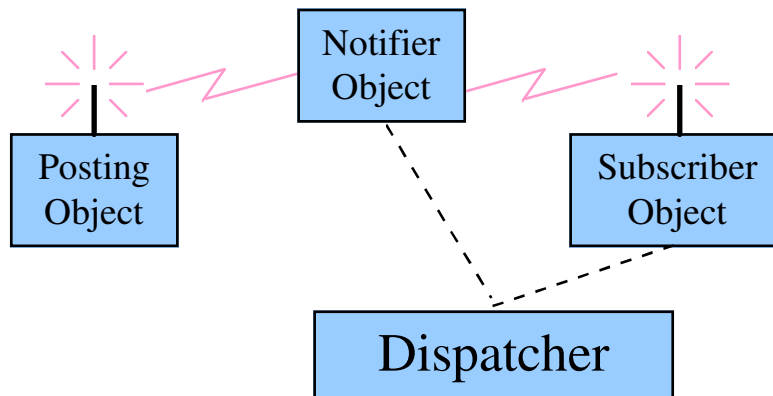


Figure 2: Four parts of the notifier subsystem

The notifier system has four pieces:

- a publisher, A
- a subscriber, B
- a notifier
- a mechanism to dispatch notifiers and register subscribers

As shown in Figure 2, the publisher creates a notifier and tells the dispatcher to place it in the queue. Subsequently, the dispatcher calls the subscriber, passing it the notifier.

The dispatcher has the hardest job, maintaining the lists of notifiers and subscribers, but this is a module that can be written once and forgotten. The other three objects have minimal code.

At first it may seem excessive to communicate by creating and posting a notifier instead of making a direct function call. If you already have a pointer to an object, calling a method of the object is faster and easier. But if you don't have that pointer to the object, notifiers are easier. That's why even the newest members of a team find themselves using notifiers to connect objects instead of using pointers.

Further on, there's a more detailed comparison of pointers and notifiers.

The Publisher

The publisher tells the system about a change. For example, a data module reports that a database value has changed. A real-time module reports a data acquisition event. A user-preferences module reports a font change.

The publisher can post the notifier or send it. Posting is asynchronous: the notifier is queued and the publisher continues to execute without waiting for the notifier to be dispatched. Sending is synchronous: the notifier is dispatched to all its subscribers before the publisher continues.

The Notifier

The notifier encapsulates information about a change and contains its own copy of the data. This is important because a notifier stays in the dispatch queue after posting, so it may outlive the publisher and the original data. For example, a real-time notifier might contain sensor data; a user-preferences notifier might contain fonts and colors.

The Subscriber

The subscriber is any object that inherits from the `CSubscriber` mixin class shown in [Listing 2](#). Subscribers are kept in a list maintained by the dispatcher and are removed from the list automatically upon destruction.

Notifiers are dispatched to subscribers by calling a `CSubscriber` virtual method. For this reason, each notifier class has a corresponding method in the `CSubscriber` class. A notifier class that reports changes in temperature has a corresponding *OnTemperatureNotifier* method in the `CSubscriber` class. To receive that notifier, a subscriber would:

- subscribe to the temperature notifier
- override *OnTemperatureNotifier*

The Dispatcher

The dispatcher maintains a list of all pending notifiers and all active subscribers. It receives notifiers from the publisher, sends the notifiers to the appropriate subscribers, then destroys the notifiers. Only the subscriber and notifier base classes are aware of the dispatcher.

In the Windows version, I triggered the dispatcher periodically with a timer message, so it always runs in the application thread.

An example

I used notifiers as the basis for a simple Windows application that looks like a household thermometer. To show how notifiers help during the evolution of a product, I've developed the example through several 'customer releases'.

Release 1: A simple thermometer

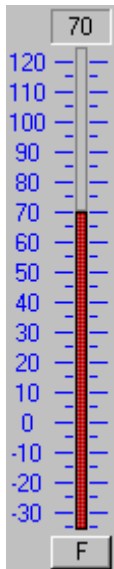


Figure 3. Release 1 of thermometer example: an analog meter, a text display of degrees, and a Celsius/Fahrenheit button.

The product begins its life-cycle as the thermometer shown in Figure 3. The MFC AppWizard generated the shell of the application and to this I added a few objects:

- A temperature sensor running in its own thread.
- A text degrees window at the top to display the temperature digitally.
- A thermometer window containing a scale and a bar of mercury.
- A 'Celsius' button to toggle between Fahrenheit and Celsius.

Then the notifiers:

- A temperature notifier sent by the sensor when the temperature changes. The text display and the graphical thermometer both respond to it.
- A units notifier sent by the Celsius button to toggle between Fahrenheit and Celsius.

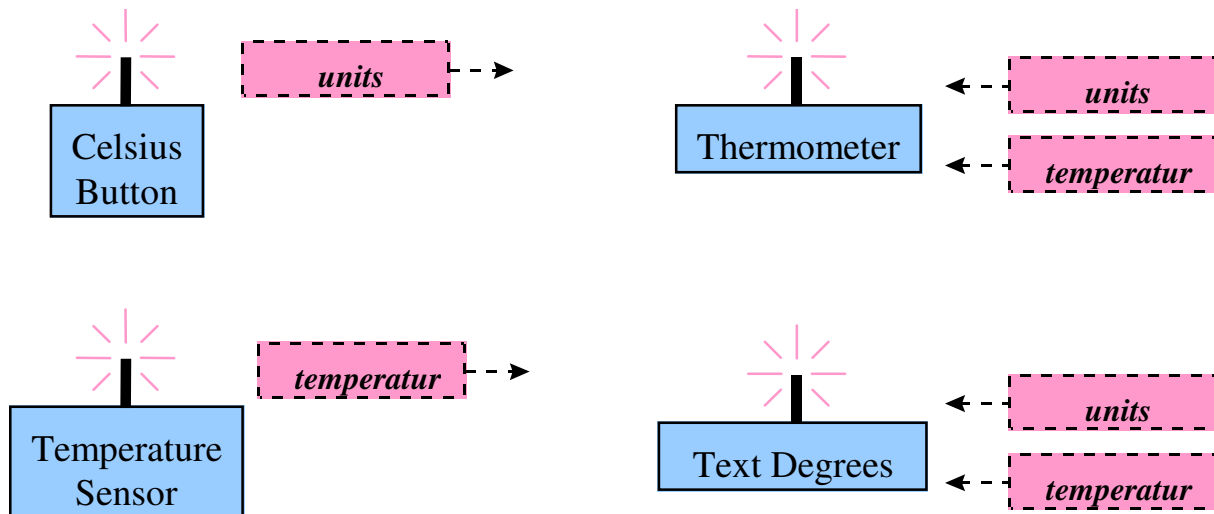


Figure 4: Relationships of objects and notifiers in Release 1. Pink boxes shows the notifiers, with arrows indicating if they are sent or received.

The object relationships are shown in Figure 4. The objects are unaware of each other, depending only on the notifiers represented by the pink boxes. Arrows show whether the notifiers are sent or received.

When the temperature changes, the sensor posts a notifier:

```
PostTemperatureNotifier(m_temperature);
```

The thermometer class receives the notifier by subscribing to it during initialization and overriding the *OnTemperatureNotifier* method inherited from the *CSubscriber* base class.

```
HRESULT CThermometer::Init(void)
{
    ... other initialization ...
    Subscribe(TEMPERATURE_NOTIFIER);
    return result;
}

void CThermometer::OnTemperatureNotifier(CTemperatureNotifier *pNotifier)
{
    if ((GetSafeHwnd() != 0) &&
        (m_temperature != pNotifier->m_temperature))
    {
        m_temperature = pNotifier->m_temperature;
        RedrawThermometer();
    }
}
```

The notifier itself is a simple object with one data member, shown in [Listing 1](#). The header includes two inline convenience functions, one for posting the notifier and one for sending it.

Without notifiers, the temperature sensor and the thermometer would be forced to exchange pointers. The thermometer would call the sensor to get the temperature during initialization, and the sensor would call the thermometer whenever the temperature changed. They would also need to tell each other when they were deleted, so neither was left with a dangling pointer.

Even so, notifiers are overkill in this very simple example, but let's see what happens as the product continues through its life cycle.

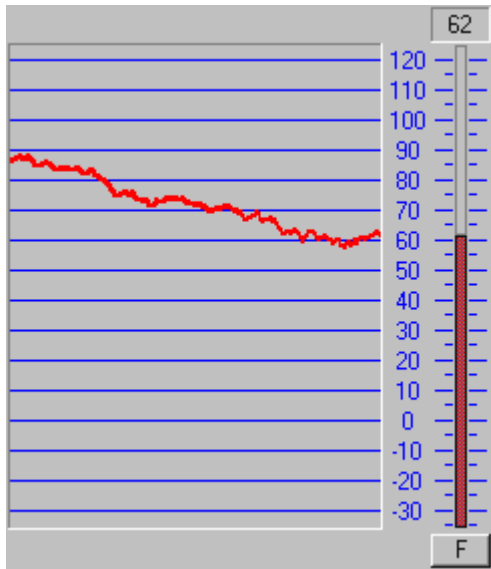
Release 2: Add a temperature graph

Figure 5: Release 2 adds a graph to track temperature.

Figure 5 shows the addition of a graph to track temperature over time. The graph object responds to changes in temperature and toggling of Celsius/Fahrenheit. Since these notifiers already exist, all I had to do was write the code for the graph and modify the parent window to instantiate it. None of the other objects changed at all. In fact, none of the other objects are aware of the graph, and vice versa.

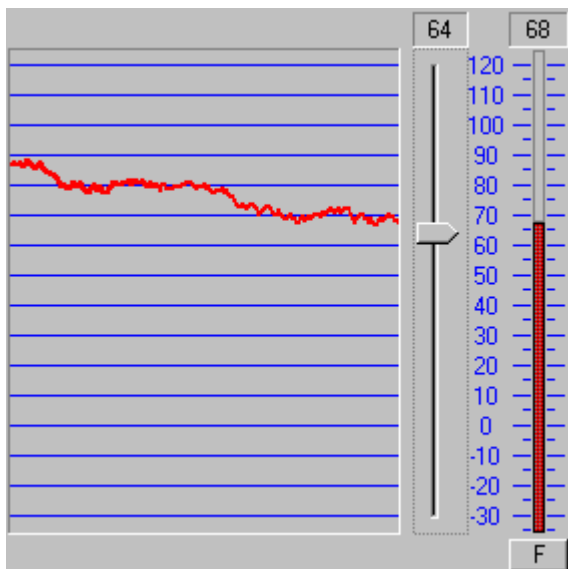
Release 3: Add a thermostat setpoint

Figure 6: Release 3 adds a thermostat setpoint.

Release 3 changes the thermometer to a thermostat by adding a setpoint, shown in Figure 6. The slider changes the setpoint, which is reflected in a text display above it. The graph tracks the

setpoint as well as the temperature, and the sensor simulates climate control by adjusting the temperature to the setpoint.

To implement this release, I had to:

- add the two new windows and the setpoint notifier
- change the graph to respond to setpoint changes
- change the temperature sensor to simulate changes as if a furnace was responding to the setpoint

The number of objects increased and the interaction between those objects increased even more, but only one new notifier was added, as shown in Figure 7.

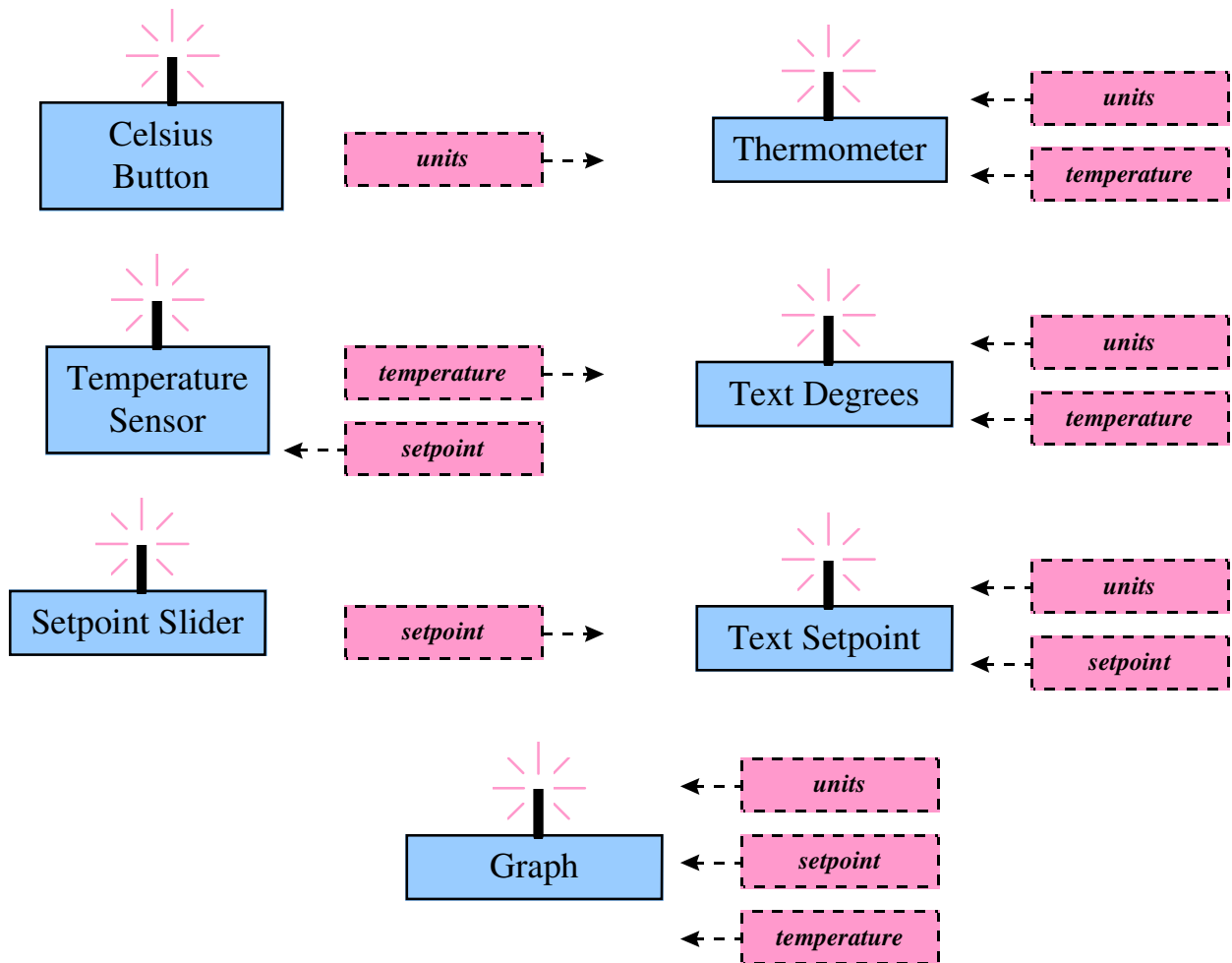


Figure 7: Object relationships with a graph and setpoint.

This is the profound advantage of notifiers and this is why I believe they're an essential tool of the object-oriented developer. Notifiers constrain the design of an object, limiting its complexity in a fundamental way, without limiting its operational capability.

Pointers vs. Notifiers

If I'd built the example program using pointers to connect objects, I could have gotten the same results and the code would be faster and more efficient, but the object relationships would be more complex, as shown in Figure 8.

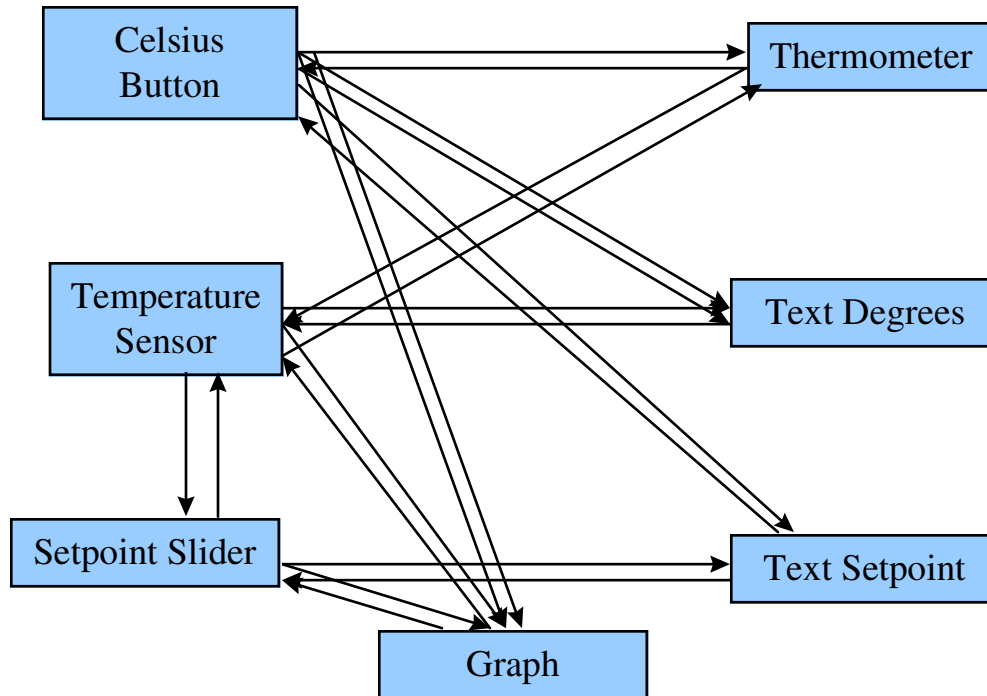


Figure 8. Pointers are faster but more complex than notifiers.

The complexity comes from these requirements:

- View objects request data when they initialize, and data objects tell view objects when they change, so each needs a pointer to the other.
- Each object has to tell its clients when it's destroyed, so pointers are not left dangling.
- Each pointer is associated with a full header file.
- Clients make direct calls to an object's interface, which means an object can't be changed without considering all its clients.
- Additional objects can cause exponential growth in object relationships.

Complex pointer exchanges aren't necessary with small non-interactive systems, so with those I might not bother with notifiers. On the other hand, I wouldn't consider writing a user-interface of any size without notifiers. User-interfaces tend to have many objects, each of which needs to know some aspect of the system state. Further, user-interfaces are hardest hit by changing requirements so they have the greatest need for design flexibility.

Requesting Information

One problem with notifiers is getting information on demand. Let's say the user opens a new graph to track the setpoint. The graph needs the current setpoint, but won't receive a notifier until the setpoint changes. Since the graph isn't directly connected to the setpoint control, it can't call it for the current value.

I solved this problem by creating a special notifier just for requesting other notifiers. If a subscribing object needs an update, it posts a *RequestNotifier*, like this:

```
RequestNotifier(SETPOINT_NOTIFIER);
```

The slider responds by posting the setpoint notifier. The setpoint doesn't know who sent the request and the graph doesn't know who sent the notifier. If I decide later to eliminate the slider and have a different object maintain the setpoint, the graph won't change at all.

This technique has another important use. It's especially tricky to initialize a system containing mutually-dependent objects. Adding new objects to the startup sequence can break old code.

You can use used notifiers to decouple objects during startup:

- in the constructor for each object, initialize all data to safe values. The object can run with these values, although it won't do anything useful
- each object posts a *RequestNotifier* for each type of data it needs
- after all objects are created, the *RequestNotifiers* are dispatched and the publishers respond by posting their current values.
- the subscribers receive their first updates and begin normal operation

Since objects don't exchange pointers, startup and shutdown aren't sensitive to the order of object creation and deletion.

Notifiers and the Observer design pattern

Notifiers are similar to the Observer pattern described by Gamma et al. in *Design Patterns*. The Observer pattern defines two abstractions, a subject (publisher) and an observer (subscriber). Observers hold pointers to subjects. When a subject changes state, it triggers a notification which is sent to its observers. The observers use their pointer to the subject to call for an update.

The advantages of the observer pattern:

- There is no overhead of a separate notifier object.
- Notifications are much faster.

The disadvantages:

- Observers have a pointer to the subject, which become invalid if the subject is deleted.
- Observers are dependent on the subject's interface.

I use the notifier object to abstract the changed information, which becomes the only element shared between the publisher and the subscriber. I've found this abstraction to be the greatest advantage of using notifiers.

Frequently Asked Questions

Why not use COM events and connection points?

One problem is that COM connections are direct. A client creates a sink which holds a pointer to a connectable object. The client can't subscribe unless the connectable object exists. The connectable object can't be deleted until the client releases it. In contrast, notifiers keep subscribers and publishers separate, connected only to the dispatcher. This independence makes implementing both objects simpler.

COM is a powerful mechanism for communicating with outside objects. Notifiers can help by distributing external COM events to internal C++ objects. For example, on a Font change event sent to an ActiveX control, a notifier could propagate the new font to all the views in the control.

Why not use Windows messages instead of inventing this new mechanism?

Unfortunately, only a window can receive a Windows message. In addition, using a dispatcher mechanism allows the messages to be objects, to carry their own data, and to be deleted after dispatching.

Why not have the dispatcher in its own thread?

If the dispatcher runs in its own thread, subscribers have to lock the data referenced in their *OnNotifier* methods, and they have to lock it everywhere else it's referenced.

It's better to run the dispatcher in the main application thread. If a notifier will trigger a long-running operation, that one operation can be isolated in its own thread, which bounds the synchronization problem.

As the system grows, do notifier classes proliferate?

Not necessarily. A notifier class defines a set of related data associated with a set of subscribers. If I need to send a new kind of update, I look at the existing notifier classes to see if one exists with a similar purpose targeted to the same subscribers. Table 1 shows how data might be assigned to notifier classes in a factory automation system. I've found that even large systems can get by with a small working set of notifiers.

Purpose	Data	Publishers	Subscribers
user-preferences	fonts, colors, application options	property events, options dialogs	all UI objects
thermostat data	temperature, setpoint	temperature sensors, and controls	control logic, thermometer displays
flow data	pressure, flow rate, valve settings	flow sensors and controls	control logic, valve displays

Table 1: Assigning related data to notifier classes

What about performance?

Notifiers are slower than pointers. If a notifier is posted, it must be allocated and queued, and each subscriber to that notifier must be called. If a notifier is sent, there is no allocation or queuing, but the subscribers are still called.

In addition, the presence of the *OnNotifier* functions in the v-table of each subscriber increases the memory footprint proportional to the number of notifier classes and the number of different subscriber subclasses.

When used appropriately, however, notifiers have little impact on performance. In commercial systems with nearly a hundred notifiers (too many!) and dozens of subscribers, the dispatcher has never been a bottleneck. Other problems, like database access time, were much bigger culprits.

Conclusions

Pointers are a wide, fast pipe for carrying a complex payload of information. They're hard to debug, unrestricted in use and resistant to change. Despite their disadvantages, they remain the best approach for naturally complex or performance-sensitive connections.

Notifiers are narrow, specific units of information. They allow you to abstract the data flow of a system to promote independence between publishers and subscribers. Notifiers are easy to use because they don't use pointers, but they're slower and less direct, so they're more appropriate for loosely-coupled objects.

Used properly, notifiers make systems so much simpler that I wouldn't start a major project without them.

Bibliography

[Foote, Brian \(Advisor: Ralph Johnson\). "Designing to Facilitate Change with Object-Oriented Frameworks". Masters Thesis, 1988, Dept. of Computer Science, University of Illinois at Urbana-Champaign.](#)

[Burbeck, Steve. "Applications Programming in Smalltalk-80™: How to use Model-View-Controller \(MVC\)."](#)

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995.

Biography

Dave Pomerantz is the owner of On Target Software, a consulting firm in Marshfield, Massachusetts. The thermometer and notifier source code are available at <http://www.targetsoft.com>.

Listings

Listing 1. Temperature notifier class.

```
class CTemperatureNotifier : public CNotifier
{
public:
    CTemperatureNotifier(void) : CNotifier(TEMPERATURE_NOTIFIER)
        { m_temperature = 0.0; }

public:
    float m_temperature;    // temperature is in degrees celsius
};

// convenience function to send the notifier immediately
inline HRESULT SendTemperatureNotifier(float temperature)
{
    HRESULT hr = E_OUTOFMEMORY;
    CTemperatureNotifier *pNotifier = new CTemperatureNotifier();
    if (pNotifier)
    {
        pNotifier->m_temperature = temperature;
        pNotifier->Send();
    }
    return hr;
}

// convenience function to queue the notifier for dispatching
inline HRESULT PostTemperatureNotifier(float temperature,
    CNotifier::PRIORITY nPriority = CNotifier::PRIORITY_NORMAL)
{
    HRESULT hr = E_OUTOFMEMORY;
    CTemperatureNotifier *pNotifier = new CTemperatureNotifier();
    if (pNotifier)
    {
        pNotifier->m_temperature = temperature;
        pNotifier->Post(nPriority);
    }
    return hr;
}
```

Listing 2. Subscriber class definition

```
class CSubscriber
{
public:
    enum SUBSCRIBER_PRIORITY
    {
        PRIORITY_HIGH,
        PRIORITY_NORMAL,
        PRIORITY_LOW
    };

public:
    // Constructors and destructors
    CSubscriber(void);
    CSubscriber(const CSubscriber &src);
    virtual ~CSubscriber(void);
    CSubscriber &operator=(const CSubscriber &src);

    // Subscription Operations
    // GetSubscriberPriority: return relative priority of subscriber
    SUBSCRIBER_PRIORITY GetSubscriberPriority(void);

    // SetSubscriberPriority: move this subscriber ahead or behind other
    // subscribers in the list
    void SetSubscriberPriority(const SUBSCRIBER_PRIORITY nPriority);

    // IsSubscribed: return true if a subscriber will be sent this notifier
    bool IsSubscribed(UINT nSubscription);
    // Subscribe: add subscriber to the list that receives this notifier
    HRESULT Subscribe(UINT nSubscription);
    // Unsubscribe: no longer send this notifier type to the subscriber
    HRESULT Unsubscribe(UINT nSubscription);

    // Notifier response functions, called by dispatcher
    // called when the temperature changes
    virtual void OnTemperatureNotifier(CTemperatureNotifier *pNotifier);
    // called when temperature units change
    virtual void OnUnitsNotifier(CUnitsNotifier *pNotifier);
    // called when thermostat setpoint changes
    virtual void OnSetPointNotifier(CSetPointNotifier *pNotifier);
    // called when another subscriber wants a notifier
    virtual void OnRequestNotifier(CRequestNotifier *pNotifier);

protected: // Implementation
    // SetDispatcher: Called by CDispatcher
    static void SetDispatcher(CDispatcher *pDispatcher);
    // GetDispatcher: Get pointer to dispatcher
    static CDispatcher * GetDispatcher(void);

private: // Data members
    static CDispatcher *m_pDispatcher; // see SetDispatcher

    SUBSCRIBER_PRIORITY m_nSubscriberPriority;

    friend class CDispatcher;
};
```