Prying Eyes: A Policy-Based Observer (II)

Andrei Alexandrescu

February 19, 2006

During this year's Software Development West conference (`http://sdexpo.com`), I have had the honor to personally meet Alexander Stepanov, the creator of the STL. The Russian mathematician and Soviet dissident has produced a strong impression on at least three people—David Abrahams, Eric Niebler, and myself.

The three of us have had the chance to enjoy a chat with Stepanov over a glass of wine following his conference talk. Both in his talk and in the private discussion, Stepanov's message came across loud and clear: the essence of programming is mathematics, and any good programmer should master the mathematics that forms the basis of computing, which is just highschool-level algebra and (perhaps surprising to the unwary) geometry. Speaking of people who have the inclination to "do something," Stepanov said: "If you really want to do something in programming and can afford it, go back to school. It's obvious. I mean, if you wanted to play piano, would you have to ask if you need to buy a keyboard?" That, of course, was like (piano) music to the ears of someone—such as, just as an example, yours truly—who had decided to quit a well-paid job to attend grad school. (It was also great that Adobe paid for the wine.) Stepanov mentioned two books he'd recommend to any programmer, information that I'll dutifully pass further (although I do reckon the guru's message is pale when passed through an intermediary): George Chrystal's Algebra [2] and Euclid's Elements [3].

Assuming you have now come back from your nearest bookstore and could set the two freshly-acquired books aside for a minute, let's continue with analyzing the Observer design pattern [4] and generic implementations thereof. To recap the first article dedicated to Observer [1], let's walk again through the main points already discussed:

- The actors in an observation framework are: Subject, Event, and Observer. Observer objects watch Events originated by Subjects.
- The basic components are a subscription service and a distribution service. The subscription service maintains a registry that identifies which Observers are interested in which Events of which Subjects. The distribution service carries event notifications from the Subject to its Observers.
- The very palpable danger of mutual interactions between calls to registration, deregistration, and updating makes the pattern implementation challenging.

1 Mailcontainer

There's been some good traffic in regarding the Observer pattern. Umesh Sirsiwal writes:

> I have used a simple but very effective solution to the observer problem. As you mentioned, in a simplistic world, `Subject` will store `Observer` in a list and inform `Observer` when the event takes place. As you pointed out, this simple solution does not work for `Observers`. Remembering iterator at `Attach` and `Detach` works in simple solutions but does not scale if there is a

> possibility of multiple traversal over the list for different types of events.
>
> The solution I use is to insert a dummy element in the middle of list that was iterated point to this element while making the callback. This guarantees that the iterator is always pointing to a sane list element when the callback returns. This scales well, because you can use multiple iterators for multiple events happening at the same time. You just need a flag showing that this element as dummy element. If one wants to place callbacks to `Observer` because of a large number of `Subject`s (for example distribution of routes), the program can give up CPU and come back—and the iterator is still pointing to a sane location.
>
> As you mentioned, list is not the most efficient container for large number of observers. In that case, I maintain 2 different containers. Map for fast lookup, and list for signaling the observer. Map itself just contains pointer to the list element.

That's an interesting design, reminding of the linear search algorithm that first plants a sentinel at the end of the searched sequence; just like that algorithm, it has the disadvantage that it needs to mutate the structure being iterated.

A solution I have myself used in the past that bears some similarity with Umesh's is to use obliteration instead of removal. That is, `Subject` stores a vector of pointers to `Observer`, and whenever a deletion occurs, instead of actually removing the slot from the vector, `Detach` would simply write the null pointer in the corresponding slot. Of course, this doesn't help insertions that much.

Benjamin Kaufmann writes:

> I recently had to implement a little type safe observer-system [...] and I had exactly the problems you mention in your article, i.e (1) how to handle attaching/detaching while an update is active? (2) how to handle intervening notifications (a notify is triggered while an update is active)?
>
> I decided to implement an event-centric system with an event-channel class as its central element. An event channel dispatches events of one specific type and provides an interface for attaching/detaching handlers (observers). It distinguishes four different dispatching strategies—LIFO-Order, FIFO-Order, Holding, Discarding—which are unfortunatly hard-coded as an enum. It uses deferred attaching/detaching if an event is currently active, i.e. detaching for example only marks the observer as detached but erasing is done only after the current event is completly handled (thus keeping iterators valid).
>
> Although my implementation is good enough for our current needs, I'm really curious how an C++ Expert is going to solve those problems. Even if this means that I have to start over once I finished reading your second article.

Hey, hey, wait a minute. Could someone please take that huge rock off my back? Nothing short of walking on water could impress such an astute readership, sigh. Let's not forget that Benjamin is the one who's ported Loki to Microsoft Visual C++ 6 (`http://fara.cs.uni-potsdam.de/~kaufmann/?page=lokiport`), something that I thought is impossible for the longest time. So let me set the record straight away: don't expect this article to solve all of the observation-related issues ever in one fell swoop. But, with luck, you'll see a few ideas that you might find interesting.

2 Time-Decoupled Observation

The essential member functions of `Subject`, as outlined in the previous installment, are `**bool** Attach(Observer*)`, `**bool** Detach(Observer*)`, and `**void** NotifyAll()`. An obvious improvement that will help us notify observers of a specific event is to define `**void** NotifyAll(Event)` instead. But more interestingly, let's focus on the argument passed to `Attach` and `Detach`. Why necessarily *a pointer* to the `Observer`?

Asking seemingly silly questions about types in interfaces is worthwhile when talking genericese. In the generic world there's no gratuitous commitment to type, and therefore it's worthwhile asking, what *concept* does that `Observer*` (or `Observer&` if you wish) really stand for? Not only that question is interesting, but its answer reveals an entire new dimension of the Obsever design space.

And the answer is, the `*` in `Observer*` stands for "identifier," "moniker," "handle," even "sobriquet" if you allow my being snobbish. When attaching, an `Observer` object tells the `Subject`: "Here's my business card (the pointer), keep it and call me (invoke `Subject::Update`) when something interesting (an event) happens."

The interesting part is that, while a pointer (or a reference) is the simplest incarnation of the concept of a sobriquet, pardon, identifier, there could be many others. For example, imagine the following situations:

- *Impedance adaptation.* In a database application, the observers could be procedures identified and invoked by name (a string). In this case the functionality provided by the `Observer` object itself is limited to forwarding to a call into the database API. Then, it would be good to avoid keeping a bunch of `Observer` objects just to satisfy the observation protocol.

- *"Just-In-Time" Observation.* In an application using a larger-scale object model (such as CORBA, COM, or Mozilla's own platform-independent COM), it's quite common for applications to register Observers by their identifier (such as the Globally Unique Identifier (GUID)), without the actual `Observer` objects even being in existence at the moment of attachment. For example, an application might associate an observer with a certain system or file system event by using its GUID; when that event happens, the infrastructure (which has stored the GUID) will pass that GUID to an object factory, obtain a true pointer to an `Observer`, and immediately invoke `Update` against it.

So a good design would abstractize the notion of `ObserverID` and perhaps use `Observer*` as an overridable default.

3 Event-Oriented Observation

So far, we've dealt with a subject-oriented approach: each `Observer` would attach to a `Subject`, and would receive notification whenever `Subject` finds fit.

A more refined (and potentially more efficient) design variant is to introduce per-event subscription into the mix. In such a design, attachment and detachment would be made on a per-event basis, i.e. `Subject` would expose `Attach(ObserverID, Event)` and `Attach(ObserverID, Event)`. In addition, `Detach(ObserverID)` might still be there with the semantics of detaching an observer from all of the events it was listening to.

The choice of data structures used for storing `Observers` and `Events` inside the `Subject` depends on the targeted performance and behavior during mutual calls to `Attach`, `Detach`, and `NotifyAll`. A straightforward design would prescribe a multimap keyed by `Event` and storing `ObserverIDs`.

Obviously, if there's no `Event`-based discrimination, then all of that is just a waste of infrastructure. How, then, to devise a design that allows for optional support for events? A simple solution that we'll use in this article is to define a `NullEvent` empty class as in:

```
class NullEvent {};
```

and then allow `Subject` implementers to specialize their implementations on `NullEvent` and other event event types.

4 A Hierarchy of Observers

But the question remains, how to go about customizing the Observer pattern in the zillion ways that we discussed so far? We should be able to configure the following design dimensions:

- The storage used, together with the attachment and detachment algorithms, should be customizable featuring various tradeoffs.

- Per-event subscription versus per-subject subscription should be customizable, too.

- Even for a given storage, the notification procedure can vary widely (and wildly!) including various techniques to handle attachments and detachments during notifications, notifications during notifications, and other such mutually recursive calls.

- Observer identification (by default through a pointer) should be customizable so as to support just-in-time notifications that construct the `Observer` object on the fly.

My early designs have all attempted to separate the subscription mechanism (the actual container plus `Attach` and `Detach`) from the notification mechanism (`NotifyAll`). There was quite some appeal to such an approach, backed up by numerous metaphors rooted in the real world. For example, imagine a newspaper distribution framework. The subscription service would keep recipient addresses and all, whereas the actual distribution service would be distinct, featuring the postal system, couriers, pigeons, owls, and whatnot. In such a framework, there would be several subscription systems (based for example on `vector`, `list`, and `map`) and on top of those, various transport mechanisms that take care of iterating the containers and delivering updates to `Observer` objects. That is all nice and dandy, in keep with real-world metaphors, and therefore likely to engender solutions similar to those existing in that metaphorical space. So, for the longest time, my Observer designs invariably gravitated around the formula:

```
template <
  class Event,
  class SubscriptionPolicy,
  class NotificationPolicy,
  class BrokeragePolicy
>
class Subject;
```

where `SubscriptionPolicy` would implement `Attach` and `Detach`, `NotificationPolicy` would take care of `NotifyAll` by iterating over the `SubscriptionPolicy`, and, finally, `BrokeragePolicy` would implement the ID-to-`Observer` mapping that we talked about in the previous section (by default, there would be a `PointerBrokerage` that uses pointers as IDs).

The appeal of such a scheme made it particularly hard to see its fatal flaw: there is too little orthogonality (independence) between the subscription and distribution policies to warrant decomposition across the subscription-distribution line. The iteration process in `NotifyAll` is way too dependent on calls to `Attach` and `Detach`, and therefore the notification policy would need to intercept `Attach` and `Detach` and perform things like saving iterators, throwing exceptions, backing off, and the such. Once a policy needs to to stick its nose into another policy's business, out the window are things like orthogonality, debugging policies independently, freely combining them to achieve mighty design richness, and ease of implementing them.

So something that I'd take pride in was to tear the commitment to the cute subscription-notification view away from my heart. Metaphors are great, but only when they resist beyond the watercooler conversation level. You must have a reasonable level of similarity so you can work with a metaphor. In our case, yes, transporting a newspaper is different from maintaining subscriptions to it. However, in the real world you don't have customers subscribing and unsubscribing other customers as soon as the newspaper arrives (calls to `Subject::Attach`/ `Subject::Detach` during `Observer::Update`); subscribers don't give new, confusing directions to the mailman on where to go with the other newspapers; they don't change the newspapers being delivered to other customers (active observation); they don't take enjoyment in punching the mailman (iterator invalidation); and so on.

So, could one build a policy-based design following the newspaper subscription paradigm? Sure. It would just be a very, very limited design. (On the other hand, you must admit that, if the real world were built as a replica to our designs, it would be a very fun and interesting place—except for postal workers.)

The new design I came up with is hierarchical: there is only one policy model (template if you allow overloading the word). Those policies build not in parallel with one another, but *on top of* one another, in a hierarchical fashion. Each policy implements `Attach`, `Detach`, and `NotifyAll` in addition to a few helpers that foster inter-policy communication, as we'll see below. A policy could either implement these three functions from first principles, or decorate (yes, that's the Decorator design pattern [4] used at compile time!) another policy with the same interface.

Before showing an example, let's show the `Subject` archetypal policy. In the code below, "`@`" acts as a placeholder for some code written by the policy implementer.

```
template <@arguments@>
class Subject {
public:
  typedef @ Event;
  typedef @ Observer;
  typedef @ ObserverID;
  enum {
    attachKillsAllIters = @,
    detachKillsCurrentIter = @,
    detachKillsAllIters = @
  };
  Observer* ID2Observer(ObserverID id);
  bool Attach(ObserverID, Event);
  bool Detach(ObserverID, Event);
  bool Detach(ObserverID);
  void NotifyAll(Event);
protected:
  typedef @ iterator;
  iterator begin(Event);
  iterator end(Event);
};
```

Of course, this is a syntactic interface that doesn't need to be respected verbatim. For example, `Observer` could be a class and not a **`typedef`**. The interface's protected part is provided to allow derived `Subjects` to hook new functionality.

The **`enum`** values can be zero or nonzero, depending on the capability level implemented by the `Subject`. If `attachKillsAllIters` is nonzero, that means a call to `Attach` invalidates any iterators. Similarly, if `detachKillsAllIters` is nonzero, that means a call to `Detach` invalidates any iterators. As a refinement, if `detachKillsAllIters` is zero and `detachKillsAllIters` is nonzero, that means a call to `Detach` only invalidates iterators pointing to the element being detached, but not others (this is the case for all node-based containers such as `list` or `map`). As a corrolary, a `Subject` that has all zeros in the **`enum`** values is rock solid (and perhaps not too efficient).

The idea is to build piecemeal functionality as `Subject` policy implementations that build on top of other `Subject` policy implementations. For example, let's turn back and implement the simplest Observer design—the one that we've shown in the previous installment of Generic⟨Programming⟩. We first define a `BaseSubject` class that's only templated on the event type, and has no protected interface because it provides no iteration. `BaseSubject` serves as an abstract root on which to build functionality.

```
template <class E>
class BaseSubject {
public:
  typedef E Event;
  struct Observer {
    virtual void Update(Event) = 0;
  };
  typedef Observer* ObserverID;
  enum {
    attachKillsAllIters = 1,
    detachKillsCurrentIter = 1,
    detachKillsAllIters = 1
  };
  virtual bool Attach(ObserverID,
    Event) = 0;
  virtual bool Detach(ObserverID,
    Event) = 0;
  virtual void Detach(ObserverID) = 0;
  virtual void NotifyAll(Event) = 0;
  virtual ~BaseSubject() {}
  Observer* ID2Observer(ObserverID id) {
    return id;
  }
};
```

Now let's implement `BareboneSubject` on top of `BaseSubject`:

```
template <class E>
```

```
class BareboneSubject : BaseSubject<E> {
public:
  typedef typename
    BaseSubject<E>::Event Event;
  typedef typename
    BaseSubject<E>::Observer Observer;
  typedef typename
    BaseSubject<E>::ObserverID ObserverID;
  bool Attach(ObserverID id, Event e) {
    value_type v = make_pair(e, id);
    if (find(data_.begin(),
        data_.end(), v)
        != data_.end()) {
      return false;
    }
    data_.push_back(v);
    return true;
  }
  virtual bool Detach(ObserverID id,
      Event e) {
    const value_type v = make_pair(e, id);
    const iterator i =
      find(data_.begin(),
      data_.end(), v);
    if (i == data_.end()) return false;
    data_.erase(i);
    return true;
  }
  virtual void Detach(ObserverID id) {
    for (iterator i = data_.begin();
        i != data_.end(); ) {
      if (i->second != id) ++i;
      else i = data_.erase(i);
    }
  }
  virtual void NotifyAll(Event e) {
    for (iterator i = data_.begin();
        i != data_.end(); ++i) {
      if (i->first != e) continue;
      (i->second)->Update(e);
    }
  }
private:
  typedef pair<Event, ObserverID>
    value_type;
  typedef vector<value_type> container;
  container data_;
protected:
  typedef typename container::iterator
    iterator;
  iterator begin(Event);
  iterator end(Event);
};
```

Once this rather boring scaffolding is in place, adding new functionality is quite easy. For example, let's define a policy implementation `ClosedNotify` that builds on top of some other policy (possibly `BareboneSubject`) a mechanism that rejects calls to `Attach` and `Detach` during `NotifyAll`:

```
template <class Subject>
class ClosedNotify : Subject {
public:
  ClosedNotify() : closed_(false) {
  }
  typedef typename
    Subject::Event Event;
  typedef typename
    Subject::Observer Observer;
  typedef typename
    Subject::ObserverID ObserverID;
  bool Attach(ObserverID id, Event e) {
    if (closed_)
      throw logic_error("");
    return Subject::Attach(id, e);
  }
  virtual bool Detach(ObserverID id,
      Event e) {
    if (closed_)
      throw logic_error("");
    return Subject::Detach(id, e);
  }
  virtual void Detach(ObserverID id) {
    if (closed_)
      throw logic_error("");
    return Subject::Detach(id);
  }
  virtual void NotifyAll(Event e) {
    closed_ = true;
    struct Local {
      ~Local() { *b_ = false; }
      bool * b_;
    } local = { &closed_ };
    Subject::NotifyAll(e);
  }
private:
  bool closed_;
};
```

(Yeah, I, too, like the **struct** Local inside NotifyAll that resets b_ quickly and easily.) The ClosedNotify policy builds a fail-proof notification strategy on top of any other Subject policy, no matter how that is implemented. And that's the beauty of it all: you grow whatever complex functionality by choosing the appropriate policy layers. In an application, you'd define for example:

```
typedef BaseSubject<int> MySubject;
typedef MySubject::Observer MyObserver;
...
typedef ClosedNotify<BareboneSubject<int> >
  MySubjectImpl;
```

Pretty cool. Other policies could flip some or all of the three **enum**s from **true** to **false**, guard against exceptions thrown from within Observer::Update, and so on.

Now, the email traffic following my first column on the subject suggest a high level of interest in the Observer pattern, and that many of you have built quite some interesting variations on it. Furthermore, I'm way over the word limit for this article and I want to keep Stepanov's advice because I believe it's important. So here's a challenge for you: send me some good policy implementations starting from the Subject model above. Let's see how robust the design is—or how it could be made better. The best, most insightful submissions will be featured in the next column's Mailcontainer section.

5 Conclusions

After concluding (definitely without exhausting the subject) the discussion on variations of the Observer design pattern, this article built a small framework that relies on a hierarchical, layered topology of policies, as opposed to the usual scattered one in which each policy is a separate template argument. The hierarchical design has the advantage that it deals much better with unorthogonalities. The disadvantage is that it puts extra burden on the client in that the design user must be careful how they stack policies, otherwise the framework might generate nonsensical or inefficient implementations.

I'll attach an Observer to my email Inbox carefully, eagerly waiting your submissions. Til next time, happy coding, and don't punch the mailman.

6 Acknowledgments

Many thanks are due to Eric Niebler, who provided helpful insights at a time where I didn't even know exactly what to ask.

References

[1] Andrei Alexandrescu. Generic⟨Programming⟩: Prying Eyes: A Policy-Based Observer (I). *C++ Users Journal*, April 2005.

[2] George Chrystal. *Algebra.* 7th edition. Chelsea Pub Co, 1980.

[3] Euclid, Dana Densmore, and T.L. Heath (Translator). *Elements.* Green Lion Press, 2002.

[4] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, Reading, MA, 1995.