Generic⟨Programming⟩

Prying Eyes: A Policy-Based Observer (I)

Andrei Alexandrescu

February 19, 2006

This article discusses implementations of the Observer pattern (as described by the Gang of Four [3]. The approach we'll take is similar to that of Modern C++ Design [1]: first discuss the pattern itself, identifying its key points of customization—the places in which there is a design decision to be made. Then, we will abstract the thusly discovered points of customization into policies, and we'll assemble a generic Observer framework from those policies.

1 Mailcontainer

The three μIdeas in Generic<Programming>'s February installment [2] have gotten a lot of attention and good feedback. As the Hula Hoop inventers might have said, you never know what people are gonna like. First, Dan Nuffer pointed out:

> I just finished reading your column in the February issue of CUJ, and thought I'd mention a way to detect unchecked return codes at compile time using a gcc extension: `__attribute__((warn_unused_result))`
>
> Compiling this code:
>
> ```
> int FallibleFunction()
>   __attribute__((warn_unused_result));
> int foo() {
>   FallibleFunction();
> }
> ```
>
> outputs this warning:
>
> warning: ignoring return value of `int FallibleFunction()`, declared with attribute `warn_unused_result`
>
> The warning could be changed to an error by specifying the `-Werror` command line option.

That's pretty useful for gcc users, and those aiming for portability, Dan points out, could use `#define` to use `warn_unused_result` attribute under gcc, and fall back to runtime errors on other platforms.

David Brownell sent me his `ErrorChecker` code that my μIdea#1 unwittingly plagiarizes, and Michael Borghart nicely points out a couple of typos (redundant braces) that I'm sure you also noticed if you ever tried to copy the code in a file. Moreover, Michael points out that the code below has a problem:

```
IgnoreError(FallibleFunction());
```

(where `IgnoreError` is a type and `FallibleFunction` is the name of a function returning `ErrorCode<int>`). The intent was to construct an anonymous temporary of type `IgnoreError` from the expression `FallibleFunction()`. But the compiler yields the impenetrable error message: "`IgnoreError FallibleFunction()`: overloaded function differs only by return type from `ErrorCode<T> FallibleFunction()`". Whoa, whoa. What is going on over there?

Whenever there's trouble, the French say, *cherchez la femme*. To paraphrase that for C++, whenever there's trouble, *cherchez la fastidieuse analyse du C++*, which is my attempt at rendering in French Scott Meyers' famous "beware of C++'s most vexing parse" [4]. Indeed, `IgnoreError(FallibleFunction());` is parsed as `IgnoreError FallibleFunction();` which is nothing but the declaration of a function.

A nice solution would be to define `IgnoreError` as a function and not as a type:

```
template <class T>
void IgnoreError(const ErrorCode<T>& code) {
  // silently shut off the error, if any
  code.read_ = true;
}
```

which requires making `IgnoreError<T>` a friend of `ErrorCode<T>`.

Finally, Attila Feher points out that `ErrorCode<T>::operator T()` should be made `const`, to obey to the "least mutability" principle. He also suggests to make `ErrorCode`'s constructor explicit, something that I chose to disagree with for ease-of-use reasons.

Today's discussion was motivated by an email from David Blume, who has written a very nice study (see `http://observer.dlma.com`) on generic implementations of Observer, study that opens with a bold question:

> What would happen if Andrei Alexandrescu, the author of Modern C++ Design, and Martin Fowler, the author of Refactoring, were required to refactor and generalize the Observer pattern sample code from the book Design Patterns? I imagine we'd end up with a policy-based observer pattern.

The study is interesting and touches on a number of interesting issues, which we're likely to get into in a future installment. The focus of the article you are now reading is to recap the Observer pattern and to discuss a few important tradeoffs and caveats in implementing the pattern that are often understated or even forgotten.

2 Observer: Generalities

Honest, Observer was on the tentative table of contents for Modern C++ Design. The pattern is beautiful, immensely useful, and fits the policy-based design methodology like a glove. It's also a glove with many fingers so to say, because, as we'll see, there are many useful ways to "cut" the pattern, with various degrees of orthogonality.

Let's recap the basics of Observer, as summarized by the GoF:

> Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Central to Observer is the long-distance dispatch from a concrete "subject" and an unbounded number of concrete "observers" via the abstraction provided by their respective base classes `Subject` and `Observer`. A barebones incarnation of the Observer pattern is shown below.

```
class Observer {
public:
  virtual void Update() = 0;
};

class Subject {
public:
  virtual bool Attach(Observer*);
  virtual bool Detach(Observer*);
  virtual void NotifyAll();
};

class BareboneSubject : public Subject {
public:
  virtual bool Attach(Observer* pObs) {
    if (find(data_.begin(),
        data_.end(), pObs) != data_.end()) {
      return false;
    }
    data_.push_back(pObs);
    return true;
  }
  virtual bool Detach(Observer* pObs) {
    const Container::iterator i =
      find(data_.begin(),
        data_.end(), pObs);
    if (i == data_.end()) return false;
    data_.erase(i);
    return true;
  }
  virtual void NotifyAll() {
    for (Container::iterator i = data_.begin();
        i != data_.end(); ++i) {
      (*i)->Update();
    }
  }
private:
    typedef vector<Observer*> Container;
    Container data_;
};
```

The `Subject` offers the `Attach` and `Detach` services. Concrete observers (derived from `Observer`) can use these services to register themselves as listeners for whatever events `Subject` would later signal. Whenever it feels like it, the `Subject` can invoke `NotifyAll` and send a burst of calls to all of the registered `Observers`.

3 Attaching and Detaching

Generic<Programming> readers have an eye that's highly trained to identify sources of variability and potential customization. Thus, definitely the linear operations in `Subject::Attach` and `Subject::Detach` must have appeared to you as obvious as the plot of a Hollywood flick after five minutes.

In a system where there are few observers and/or there aren't many attachments and detachments, a linear search will do great. If, on the contrary, there are many observers that are frequently attach to and detach from subjects, then `Subject::Attach` and `Subject::Detach` can become a bottleneck. In the latter case, choosing an associative container (tree or hashtable) keyed on `Observer*` would be a better choice.

The meta-design solution is simple: encapsulate the `Container`, the obvious setter of the performance of the subscription process, and `Subject::Attach` and `Subject::Detach` into a separate policy. This is a textbook application of separating concerns in a design, and we sure won't have any problem to encapsulate the subscription API into a little policy.

4 "If it looks too simple and elegant to be true..."

It all looks so nice and clean, it's almost a pity to spoil the pleasure by asking the following simple question: what if `Attach` or `Detach` are called from within `Update`?

The GoF description of the pattern doesn't discuss this aspect, and I believe that's an important omission. The problem of interleaved registrations with notifications is a very legitimate one because it is backed up by a number of real-world situations:

- Say the `Subject` is a stock market stream, initially wired to a ticker-tape `Observer`. When some stock varies by a large percentage, the ticker-tape opens a window that tracks that stock's price. The window itself would be an `Observer`. Hence, a new `Observer` is being `Attach`ed to its `Subject` while the `Subject` notifies another `Observer`.

- The said windows should close when the market closes. And guess what, the "market close" comes as a notification from the `Subject`—so we're in the situation of calling `Detach` from within `Update`.

The implementation described above uses a `vector` for storage, and a straight iteration for event broadcasting the calls to `Observer::Update`. So the loop in `Subject::NotifyAll` calls `Observer::Update`, which calls `Subject::Attach`, which changes the `vector` in the middle of the iteration!

To illustrate the subtlety of the problem, let's switch from a `std::vector` to a `std::list`. Let's pull `Subject::NotifyAll`'s code again:

```
void Subject::NotifyAll() {
  for (Container::iterator i = data_.begin();
      i != data_.end(); ++i) {
    (*i)->Update();
  }
}
```

The `std::list` is more stable to changes during iteration: when removing an element from the list, only the iterators referring to that element are invalidated. The code above remains buggy, however: after `(*i)->Update()` returns, the next iteration calls `++i` on an invalid iterator. So a better version would increment the iterator before calling `Update`:

```
void Subject::NotifyAll() {
  for (Container::iterator i = data_.begin();
      i != data_.end(); ) {
    (*i++)->Update();
  }
}
```

This version would increment `i` and return a copy of the old `i`, which will be dereferenced and used to call `Update`. Of course, the code must be properly documented because otherwise the maintainer will sure think "hey, the original author didn't know that postincrement is actually slower than preincrement... heh, let me fix this real quick."

But of course the code is not fixed yet; what if some `Observer`'s `Update` function ends up detaching *another* `Observer`, which happens to be next in the list? We're back to square one, and the worst part is that scuh restrictions are very hard to enforce effectively.

A much more solid version is to make a copy of the container before updating anything:

```
void Subject::NotifyAll() {
  Container d(data_); // make a copy
  for (Container::iterator i = d.begin();
      i != d.end(); ++i) {
    (*i)->Update();
  }
}
```

That's rock-solid, but has juuust a little wrinkle. You could actually label one as "not a bug, but a feature"—if you detach an `Observer` object from within another `Observer`'s `Update`, then the detached `Observer` might still receive a last event even after having been detached. That's because the detached `Observer` is still present in the the snapshot of the container. Ouch. That makes seemingly innocuous code such as:

```
void TickerTape::Update() {
  Subject * pS = GetSubject();
  Observer * p = GetCurrentGraph();
  // Untie p from the world
  pS->Detach(p);
  // p is unlinked, we can delete it
  // or... can we?
  delete p; // KA-BOOM!!!
}
```

crash in flames. Again, this is the sort of problems that is hardest to protect against: the constraints are hard to document and hard to enforce efficiently.

A solution that is solid as well as efficient is to store the iterator as a member in the `BareboneSubject` class, and then make sure that the `Attach` and `Detach` functions update it properly.

There are even more dimensions on which that `Subject::NotifyAll` could be customized. To enumerate a few:

- *Threading.* What level of thread safety does the `Subject/Observer` duo offer? Things are not as simple as locking the `Subject` inside each public functions; indeed, with non-recursive locks, the system could easily deadlock by asking for the same lock twice due to recursive mutual calls among `Update`, `Attach`, and `Detach`. All of a sudden, the solution that copies the entire container and then serves notifications from the snapshot becomes much more attractive.

- *Exceptions.* The specification of should clarify whether or not functions overriding `Observer::Update` can throw exception, and what the behavior of `Subject::NotifyAll` should be (just let the exception go through? Ignore and finish the burst? Finish the burst but throw something later?...)

- *Iteration order.* A priority system could be very useful, especially as an attempt to control active observation, which is discussed below.

The takeaway of this section is that we better make the process of subscription and notification a separate, configurable part of the planned `Observer` design so that different implementations featuring different designs can plug into the larger architecture.

5 The Sins of Active Observation

The example above glossed over how an `Observer` could get a hold of the `Subject` that notified it, hiding the details under the magic function `GetSubject` in the example above; it is the time now to discuss how information is passed from the `Subject` to its `Observers`. This is an actual necessity stemming from the observation (no pun intended) that one `Observer` might be listening to several subjects, all of which will call the same `Update` function. It would be necessary, then, for the `Observer` to figure out who the caller was.

On the face of it, the solution is simple and obvious: just change `Observer::Update()` to `Observer::Update(Subject*)`. But a more general question arises: just how much information ought to be passed from the `Subject` to its `Observers` upon each burst of notifications? The Design Patterns book discusses the *push* and the *pull* model: under the push model the `Subject` gives its `Observers` comprehensive information about exactly what event happened (in the form of additional arguments to `Observer::Update`. On the contrary, in the pull model the `Observers` only receive minimal information. It is up to them, the `Observers`, to go back to the corresponding `Subject` and "interrogate" it about what happened. If your typical `Observers` track closely events that have a uniform representation (keystrokes, mouse events, stock name and price...), then a push model is most appropriate; if, on the contrary, each of the `Observers` is focused on a specific aspect of a `Subject` with complex events that are hard to represent uniformly, then the pull model is the design of choice.

Of course, as often in life, the right path is someplace in between these two extremes. As an example, imagine `Subject` is a complex image to which we attach `Observers` such as a drawing canvas, image statistics, a lens... In that case, the push model would advocate passing a lot of information about what kind of update happened, while the pull model would just have the drawing say "something happened" and let the clients ask about the details. A hybrid approach would have the image notify the clients that a specific rectangle (push information) has been updated, and the clients would have to get back to the image and pull the bits of that rectangle.

So far so good. But as soon as the idea of passing information from the `Subject` to its `Observers`, a strange feeling creeps in: how much of that information is actually handles to mutable data that would allow a reverse information flow—from the `Observers` to the `Subject`? To coin two terms, we distinguish between *passive* observation (the observer just inspects its subject during a notification cycle) and *active* observation (the observer might change its subject during a notification cycle).

Active observation introduces a severe long-distance coupling among various observers connected to the same subject: the notification order becomes a veritable food chain in which the first registered observer sees a "fresh," or better said, freshly changed subject, and the last observer sees a tarnished subject, the integral of whatever changes each of the previous observers have found fit to apply; thus, the behavior of the system becomes dependent on what observers are actually connected, and on the order they are notified. Now only try to imagine what happens when during an update some observer changes the subject by invoking some method, and the subject notifies again via a burst of `Suject::Update` calls, while the current updating cycle hasn't finished yet! This kind of inverted flow of control, in which the observers actually determine what the subject is doing, is worse than the proverbial tail wagging the dog: due to the long-distance coupling among observers, it's like the neighbor cat's tail is wagging the dog!

A good start towards a solution is to pass the subject as a pointer to `const`:

```
void Observer::Update(const Subject* p) {
  ... cannot change *p ...
}
```

Unfortunately, a good door lock doesn't help if the window's open: in a real program, the `Observers` might have aliases of the subjects that allow mutation. Such aliasing is not statically tractable (not in C++, at least) and hard to track even dynamically. So program correctness is left to two of the least reliable teammates: attention and discipline.

6 Conclusion

This article discusses some of the hidden challenges in defining good Observer designs. So far we've identified some valuable points of customization: the subscription mechanism (attaching and detaching), the distribution means (notifying all observers about a change in the subject). We've also found some dangers that are hard to protect against: mutual recursion between attachment/detachment and notification, as well as active observation.

The next installment of the column will discuss brokered observation, events and event filtering, event-centric vs. subject-centric observation, and will proceed with a policy-based implementation.

References

[1] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN 0-201-70431-5.

[2] Andrei Alexandrescu. Generic<programming>: Three μideas. *C/C++ Users Journal*, February 2005.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2.

[4] Scott Meyers. *Effective STL*. Addison-Wesley Longman Publishing Co., Inc., 2004.