
Class::Observable

This chapter will concern a CPAN module called `Class::Observable`, written by Chris Winters. The module is a *mixin class*, which means that it is intended to extend the capabilities of whatever class inherits it, possibly in a completely different direction than the rest of that class's functionality.

Suppose that the class `Alarm` inherits from `Class::Observable`. It then inherits two important methods (and a few others of less importance.) These two are `add_observer` and `notify_observers`. When one calls `->add_observer` on an `Alarm` object, one desigates some other object, the *observer*, as observing the `Alarm` object. Later on, when one calls `->notify_observers` on the `Alarm` object, the observers previously added with `->add_observer` are notified.

There are essentially three types of observers. An observer can be an object, in which case notification consists of calling the `update` method on the observers. An observer can be a class name, in which case the notification is to call the `update` method on the class. And as a special convenience, and observer can be a subroutine, in which case notification consists simply of invoking the subroutine.

So, supposing that `Alarm` inherits from `Class::Observable`:

```
my $alarm = Alarm->new();
my $guard  = Guard->new();

$alarm->add_observer($guard);   # Inherited method

...

$alarm->notify_observers(...);    # Inherited method
# Calls $guard->update(...);
```

This module was intended to emulate a similar feature in Java, and many puzzles and criticisms are

answered, at least for our purposes, by reference to the Java spec. For example, why don't the method names match: why does `notify_observers` result in `update` calls? Why not `notify` calls? [1] Or, if they are going to be `update` calls, why isn't the method that kicks them off `update_observers`? [2] In this case, the answer is "because those are the names in the Java API". The Java API designers may have their own reasons for the name mismatch, or they might just have screwed up, but we don't have to consider it any further.

I originally decided to review this module because of lines 21--29:

```
21      foreach my $observer ( @observers ) {
24          my $num_items = scalar @{ $O{ $item } };
25          $O{ $item }->[ $num_items ] = $observer;
26          if ( ref( $observer ) ) {
27              weaken( $O{ $item }->[ $num_items ] );
28          }
29      }
```

This, of course, is an **array length variable**, which we've seen before. The expression `scalar @{ $O{ $item } }` is complicated enough that it would make sense to set up a new temporary variable to hold its value, except that that value is only used to insert a new element into the array, and then to select it again. Perl's `$array[-1]` notation suffices here:

```
foreach my $observer ( @observers ) {
    push @{ $O{ $item } }, $observer;
    if ( ref( $observer ) ) {
        weaken( $O{ $item }->[ -1 ] );
    }
}
```

`Class::Observavble` comes with a test suite, and at this point we should rerun the tests to make sure they still pass. They do.

Global Variables

The other feature of the code that attreacted my attention right away was this pair of declarations, right at the top:

```
11  my %O = ();
12  my %P = ();
```

These, of course, are global variables. (Some might ask if the `my` declarations don't make them lexical variables, rather than global variables. This may be technically true, but it doesn't matter. The variables are declared right at the top of the file, and so are available to every bit of code in the entire module. To rename the variables, the maintenance programmer must examine every single method. A review of the reasons for avoiding global variables in Section 1.2 will confirm that these variables are globals, despite their declaration with `my`.

`%O` is the central data structure of the entire module: it is the data structure that records the observers of each observed thing. `%P` is much less important; it's a cache of information calculated about the `@ISA` arrays of the packages served by the module. Since `%P` is simpler, we'll deal with it first.

%P

%P, in fact, has no need to be global; it is used in only one function, _obs_get_parent_observers(), and serves to cache the information calculated by that function. We should move its declaration closer to the place it is used:

```
{
  my %P;
  sub _obs_get_parent_observers {
    ...
  }
}
```

The maintenance programmer looking at this can see at a glance that %P is the private business of not shared with _obs_get_parent_observers(), not shared with any other function, and so can be tinkered with safely, if that is needed.

People sometimes comment that such variables should be declared at the top of the module, where they will be immediately visible. But this is exactly the wrong suggestion. Unless the maintenance programmer is going to look at _obs_get_parent_observers(), it is better if they *don't* see %P, since it is irrelevant to their job. The good advice here is **limit scopes**. Variables should, in general, be declared with the smallest scope possible, because then you don't see them unless you need to, and when you do see them, you can decide immediately if they are important to the part of the program you need to change.

Problems with %O

In constrast to %P, %O truly is global. It is used in half of the module's methods, and most of the important ones. Each observed entity is recorded as a key in the hash; the corresponding value is an array of the observers of that entity. Unfortunately, this technique has some severe problems in Perl.

Identity Theft

The first of these is the *identity theft problem*. The key in %O for an observed object is not the object itself. Perl hash keys can only be strings, so the key is the stringized version of the object. Returning to the code we saw earlier:

```
25          $O{ $item }->[ $num_items ] = $observer;

27              weaken( $O{ $item }->[ $num_items ] );
```

Here $item is the object itself, but when it is used as a hash key, it turns into something like Alarm=HASH(0x436c1d). The 0x436c1d part is the memory address at which the object resides, and since only one object can reside at a particular address at a time, this address identifies the object uniquely; it's impossible for another object to have the same address. But the key phrase in the previous sentence is "at a time"; if the obect is destroyed, there is no guarantee that another object will not be created later at the same address, and so have the same hash key. If the module looks up this new object in the %O hash, it will see the observer list for the old, dead object:

```
my $g = Guard->new("Fred");
my $a1 = Alarm->new;
```

```
$a1->add_observer($g);
$a1->notify_observers("I like pie!");   # Okay

undef $a1;
my $a2 = Alarm->new;
$a2->notify_observers("I like pie!");
```

Here we create an `Alarm` object, `$a1` and add an observer for it. Then we destroy `$a` and create a new `Alarm` object, `$a2`. `$a2` is completely fresh, and should have no observers, but the `$a2->notify_observers` call erroneously notifies Guard `$g`, because the module confuses `$a1` and `$a2`.

This is hard to fix, and the blame lies squarely on the Perl language. [3] In some contexts, an appropriate solution would be to add a `DESTROY` method on `$a1` that would remove its entry from `%O` when `$a1` was destroyed. In this case, the natural place to put this functionality is in the `Class::Observable::DESTROY()` method. But becuase Perl's `DESTROY()` semantics are unreliable, this won't work. Specifically, the problem is that `Alarm` might already have its own `DESTROY()` method, and if it does, this method will be called instead of the inherited `Class::Observable::DESTROY()`. `Class::Observable` could, conceivably, replace `Alarm::Destroy()` with a wrapper function that removes entries from `%O` and then transfers control to the real `Alarm::Destroy()`, if there was one, but this seems awfully risky; you never know what stringe thing might go wrong when you try to tamper with a subroutine in another module this way. Still, we should keep it in mind as a possible solution.

Weak References

Another, related problem is that observer objects are not properly garbage-collected. Suppose `$guard` is observing `$alarm`. Later, `$alarm` is destroyed. We would like `$guard` to be destroyed also, but it isn't, because it is referred to from the global `%O` hash. In fact, it is never destroyed.

The module tries to solve this problem by weakening the reference to `$guard` from `%O`. A weak reference is like a regular reference, except in the way it affects the garbage collection of the thing to which it refers. Unlike a regular reference, a weak reference does not contribute to the reference count of its referent. So it is possible that the reference count will drop to zero and trigger the destruction of the referent while there are still outstanding weak references in the program. What happens to these weak references at the time the thing to which they refer is destroyed? They are magically set to `undef`.

Using weak references in this case solves the problem that the observers are never destroyed, at the cost of introducing a more serious problem: now some observers are destroyed too soon:

```
my $alarm = Alarm->new();
$alarm->add_observer(Guard->new);
$alarm->notify_observers("I like pie!");
```

Here there is a reference to the anonymous guard object. Another reference is installed in the `%O` hash, and this new reference is weakened. When execution of the second statement completes, the only regular reference to the guard object is destroyed,and this triggers garbage collection of the guard object. The reference in the `%O` hash is set to `undef`.

When `notify_observers()` is called, the module looks at the observer list for `$alarm` and finds an undefined value where it expected to find an observer object, resulting in a fatal error:

```
Failed to send observation from 'Alarm=HASH(0x8113f74)' to '':
```

```
Can't call method "update" on an undefined value at
lib/Class/Observable.pm line 95.
```

The fatal error could easily be avoided, by putting a check for undefined items into `notify_observers()`, but that isn't the point. The point is that the alarm is trying to notify its observers, and the guard object can't be notified, because it has been prematurely destroyed.

This kind of problem is typical of weak references. They are useful only in a very limited set of situations. Whenever you see one used, a warning bell should go off in your head, and you should immediately check to see if the use is vulnerable to a problem analogous to this one.

Overencapsulation

Sidebar: Overencapsulation

The `Class::Observable` module has another nice example of overencapsulation. It contains a `set_debug()` method, documented as follows:

> `SET_DEBUG( $bool )`
>
> Turn debugging on or off. If set the built-in implementation of `observer_log()` will issue a warn at appropriate times during the process.

Now, what does this function actually do? Let's look at the code:

```
my ( $DEBUG );
sub DEBUG     { return $DEBUG; }
sub SET_DEBUG { $DEBUG = $_[0] }
```

So what we have here is a functional interface to the setting and reading of a scalar variable. What would have been wrong with just advertising the scalar variable, like this:

> To enable (disable) debugging, assign a true (false) value to `$Class::Observable::DEBUG`.

This is simpler, and there don't appear to be any drawbacks. (It's possible, although unlikely, that the debugging facility might evolve to be so complicated that it required a functional interface. In that unlikely event, the module could preserve backward compatibility by tying the `$DEBUG` variable.) [4]

A third problem with the use of `%O` here is that it is *overencapsulated*. Because `%O` is a lexical variable, it is inaccessible outside of the module file. Usually this is what we want. But in this case there is an unforunate side effect: it is nearly impossible to subclass `Class::Observable` in a useful way. For example, suppose I'd like to define `Class::Observable::ExtraTricky`, which inherits all of the methods of `Class::Observable`, but overrides the `add_observer()` method to do something else also. I'm stuck, because my `add_observer()` method will need to insert new entries into `%O`, and `%O` is inaccessible. The best my new method can do is to call the base class `add_observable()` to modify `%O` for it; if it needs to do anything different from the way `add_observable()` does, to read or modify `%O` itself, it is out of luck.

Probably the simplest way to fix this is to remove the `my` qualifier from the declaration of `%O`, making it a package variable. (What we really want here is a "protected" variable, available in derived classes of `Class::Observable` and not elsewhere. Perl has no such feature, and it wouldn't be very Perl-like to have one.) Why is `%O` lexical anyway? I think it's because protection and encapsulation are so important and so nearly always the right thing to do that we get in the habit of doing them automatically. This is a good habit, but, like all habits exercised without judgement, sometimes leads us into error.

Fixing `%O`

Let’s step back from the many problems with `%O` and consider the problem it is there to solve. Associated with some entity *E* is the list of V’s observers. Where should this list be stored?

I have a theory about the answer to such questions. I’m not sure yet that the theory is a good one, but it seems to have worked well so far. The theory is this: There is a right place to store information about an object, and a wrong place. The right place to store information about an object is **in the object**. The wrong place is **anywhere else**.

Why might this theory be correct? To answer that, consider the question "What is an object?" An object is a data structure whose purpose is to record in one place all the pertinent information about some entity. If you need to record some pertinent information about that entity, such as a list of observers of the entity, you should store it in the data structure that is in the program for that exact purpose, namely, the object that represents that entity.

This theory says that `Class::Observable` should try to store the list of observers of each object *E* in the data structure of *E* itself. For a first cut, this would mean that code like this:

```
push @{ $O{ $item } }, $observer;
```

would turn into something like this:

```
push @{ $self->{observers} }, $observer;
```

This approach solves the garbage collection problems: the observed object holds the reference to the observer object, which is just the relationship we need. The observer will hang around as long as the observer, and, if there are no other references to it, no longer.

But this approach immediately raises some other problems, which is why I think the author did not do it in the first place. The most obvious problem is: What hash key do you use? In the "first cut" example above, I used `observers`, but this doesn’t always work, because the object might already be using that key for something else.

I investigated several solutions to this problem; as I mentioned in Chapter 1, I don’t always (or usually) get the right answer on the first try, and this time was no exception. I think the failures are instructive, so I’ll show the things that didn’t work well.

Escape hatches

My first idea was that the module should simply choose an unlikely hash key, and then provide an escape hatch in case the hash key was unsuitable. The key I selected was `Class::Observable::_observers`. The key begins with the name of the class that uses it; if other modules also allocate keys in the object, and if they follow this convention, we don’t have to worry that our keys will collide with theirs.

The escape hatch is that the key name will not be hardcoded into the module. Instead of:

```
push @{ $self->{Class::Observable::_observers} }, $observer;
```

we will always use:

```
push @{ $self->{$self->_observable_key} }, $observer;
```

where `Class::Observable::_observable_key` is simply:

```
sub _observable_key { "Class::Observable::_observers" }
```

If some class, say `Foo`, wants to inherit from `Class::Observable`, but finds that the key `"Class::Observable::_observers"` is unsuitable for some reason, it can simply override the `_observable_key` method to return a key that is suitable. [5]

But this solution doesn't go far enough. What if the watched object is not a hash at all? I initially found myself making excuses for the failure of my solution to solve the problem in this case. "Yeah," I wrote in my talk about this program, "how often does *that* happen?" Then I added, "Then you shouldn't use the module." In other words, I was willing to have the module fail in that case.

A problem I ignored at at the time was that even if the watched object is a hash, there may not be any safe choice of key; some hash-based objects need every possible key to do their jobs. [6]

Observed classes

Another problem that arises with the "just put it in the object" philosophy of where to put the observer list is that observed entities need not be objects; they can be classes. With the old `%0` strategy, this was no problem. The class name makes a good key into the `%0` hash to hold the observers of a class. But how to use "put it in the object" in conjunction with something that isn't an object? I tried having the module do the analogous thing, which is to put the observer list for a class into a class variable in that class. This worked adequately, but complicated the code, because now there are two different code paths, one for when `$self` is an object, and another for when it is a class.

When `$self` was an object, its observer list was stored in `$self->{$self->_observable_key}`. When `$self` was a class, its observer list was stored in an array whose name was produced by by the overridable method

```
sub _observable_classvar {
  my $class = shift;
  return "$class\::Class_Observable_observers";
}
```

Use interfaces to abstract different implementations

The natural thing to do at this point is to put the same interface onto both cases, so that the rest of the code can treat them the same way:

```
sub direct_observers {
  my $self = shift;
  if (ref($self)) {              # Get object member
    return $self->{$self->_observable_key} ||= [];
  } else {
    no strict 'refs';
```

```
    return \@{$self->_observable_classvar};  # Get package variable
  }
}
```

Where the original code had

```
push @{$O{$self}}, ...
```

the new code would have:

```
push @{$self->direct_observers}, ...;
```

Use methods to decouple implementation decisions

At this point I finally got the right idea. The `direct_observers()` method abstracts away the decisions about where the observer lists are stored so that nobody else needs to worry about it. But it remains tightly coupled to these two decisions itself. Why not decouple them?

```
sub direct_observers {
  my $self = shift;
  return ref($self) ? $self->direct_observers_object
                    : $self->direct_observers_class;
```

The special knowledge about storage methods is then encapsulated into the `direct_observers_object()` and `direct_observers_class()` methods:

```
sub direct_observers_object {
  my $self = shift;
  return $self->{"Class::Observable::_observers"} ||= [];
}

sub direct_observers_class {
  my $class = shift;
  no strict 'refs';
  return \@{"$class::Class_Observable_observers"};
}
```

Now let's reconsider the question of what to do when the watched object is not a hash. Suddenly, there is an easy answer: override the `direct_observers_object` method to do this:

```
# Suitable for use with Some::Class::That's::an::Array
sub direct_observers_object {
  my $self = shift;
  return $self->[37] ||= [];
```

or whatever is appropriate for the particular class in question.

With this design, the user of the module can even recover the original implementation, if they like:

```
sub direct_observers_object {
  my $self = shift;
  return $O{$self} ||= [];
}
```

[1] This is because the name `notify` is already used for something else.

[2] Here I have no explanation.

[3] It is also worth pointing out that the problem is caused in the first place by the weakness of Perl's hashes: keys can only be strings, not objects. Languages such as Common Lisp and (dare I say it?) Python have hashes whose keys can be arbitrary objects, and wouldn't suffer from this problem at all.

[4] Actually, there is a drawback: someone is sure to send the author silly complaints about encapsulation issues. My `Text::Template` module has an interface like this; you can recover the most recent template error by examining `$Text::Template::ERROR`. I periodically get worried email about this, asking "Isn't this violating the interface abstraction?" (answer: no) or "What if the variable name or implementation changes in the future?" (answer: it's part of the published interface, so it won't) or "Wouldn't it be safer to provide a function to be the interface?" (answer: no) or "But what if you decide to change it someday?" (answer: then you're doomed. But using a functional interface would not solve this problem.)

[5] The method can be overridden either in `Foo` itself or in some subclass of `Class::Observable` that the author of `Foo` creates especially for the purpose.

[6] For example, a typical implementation of a tied hash class uses an object that represents the tied hash. Keys in the tied hash map directly to keys in the object.
