Announcing
Announcements

Shawn M Moore
@sartak

Announcements!
Announcements is a design pattern. BEWARE!
package Pattern::Announcements;

use vars q「@ISA」;

@ISA = qq【Pattern::Observer】;
Let’s say we’re modeling a trial. We want an object for the defendant, and an object for each of the jurors. When the defendant pleads innocent or guilty, we have to tell the jurors about it. We don’t want to just hardcode a list of jurors, or even store the jurors in an attribute on the defendant object, because that’s tight coupling. After all, the jurors are supposed to have no relationship with the defendant. Instead, we can have the jurors subscribe to the defendant’s plea event, which leads to a nice abstraction barrier.
Let’s say we’re modeling a trial. We want an object for the defendant, and an object for each of the jurors. When the defendant pleas innocent or guilty, we have to tell the jurors about it. We don’t want to just hardcode a list of jurors, or even store the jurors in an attribute on the defendant object, because that’s tight coupling. After all, the jurors are supposed to have no relationship with the defendant. Instead, we can have the jurors subscribe to the defendant’s plea event, which leads to a nice abstraction barrier.
Let’s say we’re modeling a trial. We want an object for the defendant, and an object for each of the jurors. When the defendant pleas innocent or guilty, we have to tell the jurors about it. We don’t want to just hardcode a list of jurors, or even store the jurors in an attribute on the defendant object, because that’s tight coupling. After all, the jurors are supposed to have no relationship with the defendant. Instead, we can have the jurors subscribe to the defendant’s plea event, which leads to a nice abstraction barrier.
Let’s say we’re modeling a trial. We want an object for the defendant, and an object for each of the jurors. When the defendant pleads innocent or guilty, we have to tell the jurors about it. We don’t want to just hardcode a list of jurors, or even store the jurors in an attribute on the defendant object, because that’s tight coupling. After all, the jurors are supposed to have no relationship with the defendant. Instead, we can have the jurors subscribe to the defendant’s plea event, which leads to a nice abstraction barrier.
Let’s say we’re modeling a trial. We want an object for the defendant, and an object for each of the jurors. When the defendant pleads innocent or guilty, we have to tell the jurors about it. We don’t want to just hardcode a list of jurors, or even store the jurors in an attribute on the defendant object, because that’s tight coupling. After all, the jurors are supposed to have no relationship with the defendant. Instead, we can have the jurors subscribe to the defendant’s plea event, which leads to a nice abstraction barrier.
Let’s say we’re modeling a trial. We want an object for the defendant, and an object for each of the jurors. When the defendant pleads innocent or guilty, we have to tell the jurors about it. We don’t want to just hardcode a list of jurors, or even store the jurors in an attribute on the defendant object, because that’s tight coupling. After all, the jurors are supposed to have no relationship with the defendant. Instead, we can have the jurors subscribe to the defendant’s plea event, which leads to a nice abstraction barrier.
Let’s say we’re modeling a trial. We want an object for the defendant, and an object for each of the jurors. When the defendant pleads innocent or guilty, we have to tell the jurors about it. We don’t want to just hardcode a list of jurors, or even store the jurors in an attribute on the defendant object, because that’s tight coupling. After all, the jurors are supposed to have no relationship with the defendant. Instead, we can have the jurors subscribe to the defendant’s plea event, which leads to a nice abstraction barrier.
Let’s say we’re modeling a trial. We want an object for the defendant, and an object for each of the jurors. When the defendant pleas innocent or guilty, we have to tell the jurors about it. We don’t want to just hardcode a list of jurors, or even store the jurors in an attribute on the defendant object, because that’s tight coupling. After all, the jurors are supposed to have no relationship with the defendant. Instead, we can have the jurors subscribe to the defendant’s plea event, which leads to a nice abstraction barrier.
Let’s say we’re modeling a trial. We want an object for the defendant, and an object for each of the jurors. When the defendant pleads innocent or guilty, we have to tell the jurors about it. We don’t want to just hardcode a list of jurors, or even store the jurors in an attribute on the defendant object, because that’s tight coupling. After all, the jurors are supposed to have no relationship with the defendant. Instead, we can have the jurors subscribe to the defendant’s plea event, which leads to a nice abstraction barrier.
Announcer/Publisher
In the observer pattern, when an event happens, the publisher tells each of the subscribers about it. The publisher might call a method on each of the subscriber objects passing in the event name as a string. The salient point about Observer is that it doesn’t hardcode the list of subscribers; it’s dynamic.
In the observer pattern, when an event happens, the publisher tells each of the subscribers about it. The publisher might call a method on each of the subscriber objects passing in the event name as a string. The salient point about Observer is that it doesn’t hardcode the list of subscribers; it’s dynamic.
In the observer pattern, when an event happens, the publisher tells each of the subscribers about it. The publisher might call a method on each of the subscriber objects passing in the event name as a string. The salient point about Observer is that it doesn’t hardcode the list of subscribers; it’s dynamic.
Instead, with announcements, when an event worth noting occurs, the announcer creates an announcement OBJECT. Then passes it to each of the subscribers. But the pattern looks more like this, because the same object is passed to each of the subscribers. And then when it’s done being handed out, the Announcer still has that object in the end. And when the announcer is done with it, it’s garbage collected. Since we’re done with this slide, it’s garbage collected. Oh, thought I missed one? No ma’am.
Instead, with announcements, when an event worth noting occurs, the announcer creates an announcement OBJECT. Then passes it to each of the subscribers. But the pattern looks more like this, because the same object is passed to each of the subscribers. And then when it’s done being handed out, the Announcer still has that object in the end. And when the announcer is done with it, it’s garbage collected. Since we’re done with this slide, it’s garbage collected. Oh, thought I missed one? No ma’am.
Instead, with announcements, when an event worth noting occurs, the announcer creates an announcement OBJECT. Then passes it to each of the subscribers. But the pattern looks more like this, because the same object is passed to each of the subscribers. And then when it’s done being handed out, the Announcer still has that object in the end. And when the announcer is done with it, it’s garbage collected. Since we’re done with this slide, it’s garbage collected. Oh, thought I missed one? No ma’am.
Instead, with announcements, when an event worth noting occurs, the announcer creates an announcement OBJECT. Then passes it to each of the subscribers. But the pattern looks more like this, because the same object is passed to each of the subscribers. And then when it’s done being handed out, the Announcer still has that object in the end. And when the announcer is done with it, it’s garbage collected. Since we’re done with this slide, it’s garbage collected. Oh, thought I missed one? No ma’am.
Instead, with announcements, when an event worth noting occurs, the announcer creates an announcement OBJECT. Then passes it to each of the subscribers. But the pattern looks more like this, because the same object is passed to each of the subscribers. And then when it’s done being handed out, the Announcer still has that object in the end. And when the announcer is done with it, it’s garbage collected. Since we’re done with this slide, it’s garbage collected. Oh, thought I missed one? No ma’am.
Instead, with announcements, when an event worth noting occurs, the announcer creates an announcement OBJECT. Then passes it to each of the subscribers. But the pattern looks more like this, because the same object is passed to each of the subscribers. And then when it’s done being handed out, the Announcer still has that object in the end. And when the announcer is done with it, it’s garbage collected. Since we’re done with this slide, it’s garbage collected. Oh, thought I missed one? No ma’am.
Instead, with announcements, when an event worth noting occurs, the announcer creates an announcement OBJECT. Then passes it to each of the subscribers. But the pattern looks more like this, because the same object is passed to each of the subscribers. And then when it’s done being handed out, the Announcer still has that object in the end. And when the announcer is done with it, it’s garbage collected. Since we’re done with this slide, it’s garbage collected. Oh, thought I missed one? No ma’am.
Instead, with announcements, when an event worth noting occurs, the announcer creates an announcement OBJECT. Then passes it to each of the subscribers. But the pattern looks more like this, because the same object is passed to each of the subscribers. And then when it’s done being handed out, the Announcer still has that object in the end. And when the announcer is done with it, it’s garbage collected. Since we’re done with this slide, it’s garbage collected. Oh, thought I missed one? No ma’am.
Instead, with announcements, when an event worth noting occurs, the announcer creates an announcement OBJECT. Then passes it to each of the subscribers. But the pattern looks more like this, because the same object is passed to each of the subscribers. And then when it’s done being handed out, the Announcer still has that object in the end. And when the announcer is done with it, it’s garbage collected. Since we’re done with this slide, it’s garbage collected. Oh, thought I missed one? No ma’am.
Announcements are not a new idea; they are constantly reinvented. They’re not particularly brilliant either, they’re a pretty straightforward extension of a common pattern. Furthermore, they’re so not-brilliant that hopefully they stick in your mind so that in 1, or 8 years you’ll remember this talk and have no trouble busting these bad boys out. Even if you don’t actually remember the name “Announcements” you’ll be alright.
Announcements

• Not new

Announcements are not a new idea; they are constantly reinvented. They’re not particularly brilliant either, they’re a pretty straightforward extension of a common pattern. Furthermore, they’re so not-brilliant that hopefully they stick in your mind so that in 1, or 8 years you’ll remember this talk and have no trouble busting these bad boys out. Even if you don’t actually remember the name “Announcements” you’ll be alright.
Announcements

- Not new
- Not brilliant

Announcements are not a new idea; they are constantly reinvented. They’re not particularly brilliant either, they’re a pretty straightforward extension of a common pattern. Furthermore, they’re so not-brilliant that hopefully they stick in your mind so that in 1, or 8 years you’ll remember this talk and have no trouble busting these bad boys out. Even if you don’t actually remember the name “Announcements” you’ll be alright.
Announcements

• Not new
• Not brilliant
• Obvious

Announcements are not a new idea; they are constantly reinvented. They’re not particularly brilliant either, they’re a pretty straightforward extension of a common pattern. Furthermore, they’re so not-brilliant that hopefully they stick in your mind so that in 1, or 8 years you’ll remember this talk and have no trouble busting these bad boys out. Even if you don’t actually remember the name “Announcements” you’ll be alright.
Announcements

- Not new
- Not brilliant
- Obvious
- *Life-changing*

Announcements are not a new idea; they are constantly reinvented. They’re not particularly brilliant either, they’re a pretty straightforward extension of a common pattern. Furthermore, they’re so not–brilliant that hopefully they stick in your mind so that in 1, or 8 years you’ll remember this talk and have no trouble busting these bad boys out. Even if you don’t actually remember the name “Announcements” you’ll be alright.
Announcements

An announcement is a notification about an event. Each announcement is an object. And of course, objects can have attributes. And methods. Which means announcements can have attributes and methods. I’ll let that sink in for a moment.
Announcements

• Event notification

An announcement is a notification about an event. Each announcement is an object. And of course, objects can have attributes. And methods. Which means announcements can have attributes and methods. I’ll let that sink in for a moment.
Announcements

• Event notification
• An object

An announcement is a notification about an event. Each announcement is an object. And of course, objects can have attributes. And methods. Which means announcements can have attributes and methods. I’ll let that sink in for a moment.
An announcement is a notification about an event. Each announcement is an object. And of course, objects can have attributes. And methods. Which means announcements can have attributes and methods. I’ll let that sink in for a moment.
Announcements

• Event notification
• An object
  • with attributes
• and methods

An announcement is a notification about an event. Each announcement is an object. And of course, objects can have attributes. And methods. Which means announcements can have attributes and methods. I’ll let that sink in for a moment.
If you want to announce something you consume the Announcing role. That’s the only real Moosiness in this framework. When you consume this role you get these two methods.
Announcing

with ‘Announcements::Announcing’;

• announce

If you want to announce something you consume the Announcing role. That’s the only real Moosiness in this framework. When you consume this role you get these two methods.
Announcing

with ‘Announcements::Announcing’;

• announce
• add_subscription

If you want to announce something you consume the Announcing role. That’s the only real Moosiness in this framework. When you consume this role you get these two methods.
Here’s what it looks like to subscribe to an announcement.
Since an announcement is an object, part of subscribing means you choose a class name to filter only the announcements you care about.
You subscribe to announcements from a particular object.

```perl
$defendant->add_subscription(
  when => 'Plead',
  do   => sub {
    my $plead = shift;
    $juror->deliberate($plead);
  },
);
```
In the reaction code, you get the announcement object.
Announcing

```php
$announcement = Plead::Innocent->new;
$defendant->announce($announcement);
```

To actually announce something you create the announcement object then call the announce method passing it.

Then after the announcement object has been passed around to all of the subscribers, the announcer gets it back in the end. Which means you can examine attributes that subscribers set, for example.
$announcement = Plead::Innocent->new;
$defendant->announce($announcement);

if ($announcement->is_convicted) {
    die;
}

To actually announce something you create the announcement object then call the announce method passing it.

Then after the announcement object has been passed around to all of the subscribers, the announcer gets it back in the end. Which means you can examine attributes that subscribers set, for example.
In the observer pattern you don’t GET two way communication. In announcements you do.
That’s why Announcements are “communicate across the object network” and not “a framework for event notification”
This communication which is the heart of why announcements are cool enables some interesting new patterns. I’ll show a quick example of each of these.
A company class might have a president attribute. This is a straightforward writer accessor. Let’s jazz it up with announcements.
We can announce when we’re about to change the president.
Here’s another cool thing you can do. Allow ChangePresident to be vetoed!
package ChangePresident;
use Moose;
extends 'Announcements::Announcement';

has ['old_value', 'new_value'] => (  
is => 'ro',
);

has is_vetoed => (  
is => 'rw',  
isa => 'Bool',
);

sub veto { shift->is_vetoed(1) }

Just add an is_vetoed attribute.
$bluth_company->add_subscription(
    when => 'ChangePresident',
    do => sub {
        my $change = shift;
        if ($change->new_value eq 'Jobe') {
            $change->is_veto;
        }
    },
);

Now that we have the is_vetoed attribute on the ChangePresident announcement, we can set it in a subscriber and it’ll filter back up to the announcer, thus blocking the change.
package ChangePresident;
use Moose;
extends 'Announcements::Announcement';

has ['old_value', 'new_value'] => ( 
    is => 'ro',
);

has is_vetoed => ( 
    is => 'rw',
    isa => 'Bool',
);

sub veto { shift->is_vetoed(1) }

This veto code could be a role for announcements that you put on CPAN as, I dunno...
AnnouncementsX::Vetoable

MooseX::VetoableAnnouncement

(please don’t use the MooseX namespace for extensions to Announcements!!)
This may seem crazy and wild but I bet most of you have seen the announcement veto pattern before.
It fits the same pattern as what we’ve already seen.
The really cool thing is that this announcement crosses language barriers. The browser, probably written in C++, constructs an announcement that can be passed into javascript code. Then the javascript code can “veto” the bubbling up that the browser’s C++ code will do.
This communication which is the heart of why announcements are cool enables some interesting new patterns. I’ll show a quick example of each of these.
has score => (  
  is => 'rw',  
  isa => 'Int',  
  default => 0,  
);

sub vote_up {  
  my $self = shift;  
  $self->score($self->score + 1);  
}

sub vote_down {  
  my $self = shift;  
  $self->score($self->score - 1);  
}

You can get as complicated as you like with voting. Here we maintain a score and we can reject the change if the score is not positive.
It doesn’t have to be voting on one thing either. You can have a list of items and have the subscribers select which items to operate on. This is one of the ways we used Announcements in my NetHack bot – when we saw items on the ground we asked the AI components which items to pick up using code like this.
Three-Phase Commit

- about to change
- will change
- changed

The “announcements” hello world seems to be the three-phase commit. When you want to change something, you send an announcement stating your intent. That can be vetoed. If that passes, you send another announcement saying you’re definitely about to change the value, but it hasn’t happened yet. Then you change the value and send one final announcement saying you just did that.
You might wonder why there are two before announcements. That’s because the first announcement you send out can be vetoed. If a subscriber vetoes that announcement then subscribers that need to hook into a more secure “this is definitely going to happen” announcement are being lied to. And of course, after the change already happens, it’s way too late to veto it. I’d show some code here but you already saw the interesting vetoable announcement.
You can subscribe to all of the events, since they all inherit from Announcements::Announcement, generated by a particular object and dump them for debugging purposes. That’s yet another feature you typically don’t get with observer pattern libraries.
The subscription class has a method called “matches” that determines whether a particular announcement matches the subscription. The default “matches” uses DOES which in Perl 5.10 just uses “isa”, but Moose extends the DOES method to know about roles. This means you can subscribe to announcements by role name instead of by class, even in the default framework.
In your subscription subclass you could have any extra logic you want in the matcher method. This enables some pretty crazy flexibility. There's even more flexibility here because there's a SubscriptionRegistry object, but I don't even know how to usefully extend that one.
Announcements are an idea that were formalized by Vassili Bykov for Cincom Smalltalk. Piers Cawley (who is here today) brought them to Ruby and put the idea in my head. And now I put the on CPAN as Announcements. Hooray for namespace squatting.
Announcements

- originally from Smalltalk (Vassili Bykov)
Announcements

- originally from Smalltalk (Vassili Bykov)
- via Ruby (Piers Cawley)

Announcements are an idea that were formalized by Vassili Bykov for Cincom Smalltalk. Piers Cawley (who is here today) brought them to Ruby and put the idea in my head. And now I put the on CPAN as Announcements. Hooray for namespace squatting.
Announcements

- originally from Smalltalk (Vassili Bykov)
- via Ruby (Piers Cawley)
- Announcements.pm on CPAN (me)

Announcements are an idea that were formalized by Vassili Bykov for Cincom Smalltalk. Piers Cawley (who is here today) brought them to Ruby and put the idea in my head. And now I put the on CPAN as Announcements. Hooray for namespace squatting.
Thank you very much!