The Observer Pattern

Python Patterns

Object-oriented design patterns work differently in Python than other languages, because of Python's very different feature set.

Observer pattern

Defines a "one to many" relationship among objects.

- One central object, called **the observable**, watches for events.
- Another set of objects, the **observers**, ask the observable to tell them when that event happens.

PubSub

There’s another name for this: "Pub-Sub".

- One central object, called **the publisher**, watches for events.
- Another set of objects, the **subscribers**, ask the publisher to tell them when that event happens.

To me, that’s a better name. So in working with the observer pattern, we’ll speak of "publishers" and "subscribers".

Let’s start with the simple observer pattern.

Subscriber

In the simplest form, each subscriber has a method named `update`, which takes a message.

```python
class Subscriber:
    def __init__(self, name):
        self.name = name
    def update(self, message):
        print('{} got message "{}"'.format(self.name, message))
```

The publisher invokes that update method.

Registration

The subscriber must tell the publisher it wants to get messages. So the publisher object has a `register` method.

```
class Publisher:
    def __init__(self):
        self.subscribers = set()
    def register(self, who):
        self.subscribers.add(who)
    def unregister(self, who):
        self.subscribers.discard(who)
```

Sending Messages

When an event happens, you have the publisher send the message to all subscribers using a `dispatch` method.

```
class Publisher:
    def __init__(self):
        self.subscribers = set()
    def register(self, who):
        self.subscribers.add(who)
    def unregister(self, who):
        self.subscribers.discard(who)
    def dispatch(self, message):
        for subscriber in self.subscribers:
            subscriber.update(message)
```

Using in Code

```
pub = Publisher()

bob = Subscriber('Bob')
alice = Subscriber('Alice')
john = Subscriber('John')

pub.register(bob)
pub.register(alice)
pub.register(john)

pub.dispatch("It's lunchtime!")
pub.unregister(john)
pub.dispatch("Time for dinner")
```

Output

```
# from last slide:
pub.dispatch("It's lunchtime!")
pub.unregister(john)
pub.dispatch("Time for dinner")
```

```
John got message "It's lunchtime!"
Bob got message "It's lunchtime!"
Alice got message "It's lunchtime!"
Bob got message "Time for dinner"
Alice got message "Time for dinner"
```

Other forms

This is the simplest form of the observer pattern in Python.

Advantage: Very little code. Easy to set up.

Disadvantage: Inflexible. Subscribers must be of classes implementing an `update` method.

Also: simplistic. Publisher notifies on just one kind of event.

If we go more complex, what does that buy us?

Alt Callback

In Python, *everything* is an object. Even methods.

So subscriber can register a method other than `update`.

```python
# This subscriber uses the standard "update"
class SubscriberOne:
    def __init__(self, name):
        self.name = name
    def update(self, message):
        print('{} got message "{}"'.format(self.name, message))
# This one wants to use "receive"
class SubscriberTwo:
    def __init__(self, name):
        self.name = name
    def receive(self, message):
        print('{} got message "{}"'.format(self.name, message))
```

Alt Callback: Publisher

```
class Publisher:
    def __init__(self):
        self.subscribers = dict()
    def register(self, who, callback=None):
        if callback is None:
            callback = who.update
        self.subscribers[who] = callback
    def dispatch(self, message):
        for callback in self.subscribers.values():
            callback(message)
    def unregister(self, who):
        del self.subscribers[who]
```

Using

```
pub = Publisher()
bob = SubscriberOne('Bob')
alice = SubscriberTwo('Alice')
john = SubscriberOne('John')

pub.register(bob)
pub.register(alice, alice.receive)
pub.register(john, john.update)

pub.dispatch("It's lunchtime!")
pub.unregister(john)
pub.dispatch("Time for dinner")
```

Output

```
# from last slide:
pub.dispatch("It's lunchtime!")
pub.unregister(john)
pub.dispatch("Time for dinner")
```

```
Alice got message "It's lunchtime!"
John got message "It's lunchtime!"
Bob got message "It's lunchtime!"
Alice got message "Time for dinner"
Bob got message "Time for dinner"
```

Channels

The publishers so far only do "all or nothing" notification.

What about one publisher that can watch several event types? How could we implement this?

For this, let’s use the regular "update" subscriber:

```
class Subscriber:
    def __init__(self, name):
        self.name = name
    def update(self, message):
        print('{} got message "{}"'.format(self.name, message))
```

Publisher: channels

```
class Publisher:
    def __init__(self, channels):
        # Create an empty subscribers dict
        # for every channel
        self.channels = { channel : dict()
                          for channel in channels }
    def register(self, channel, who, callback=None):
        if callback is None:
            callback = who.update
        subscribers = self.channels[channel]
        subscribers[who] = callback
```

Publisher: channels

```
    def dispatch(self, channel, message):
        subscribers = self.channels[channel]
        for callback in subscribers.values():
            callback(message)
```

Publisher: channels

```
pub = Publisher(['lunch', 'dinner'])
bob = Subscriber('Bob')
alice = Subscriber('Alice')
john = Subscriber('John')

pub.register("lunch", bob)
pub.register("dinner", alice)
pub.register("lunch", john)
pub.register("dinner", john)

pub.dispatch("lunch", "It's lunchtime!")
pub.dispatch("dinner", "Dinner is served")
```

Publisher: channels

```
# from last slide:
pub.dispatch("lunch", "It's lunchtime!")
pub.dispatch("dinner", "Dinner is served")
```

```
Bob got message "It's lunchtime!"
John got message "It's lunchtime!"
Alice got message "Dinner is served"
John got message "Dinner is served"
```

Lab: Observer Pattern/PubSub

Let's do a more self-directed lab. You're going to use the observer pattern to implement a program called `filewatch.py`.

Instructions: `patterns/filewatch-lab.txt`

- In labs/py3 for 3.x; labs/py2 for 2.7
- First follow the instructions to write `filewatch.py`
- When you are done, give a thumbs up…
- … and then follow the further instructions for `filewatch_extra.py`