Observer Pattern

CS342 -- Fall 2016

Design Problem

- Given a WeatherData object with the following methods
  - getTemperature() // gets the temp data member
  - getHumidity() // gets the humidity data member
  - getPressure() // gets the pressure data member
  - measurementsChanged()
    - this method is called by some other object whenever the data changes
- 3 display elements use the new data for display
  - current conditions, forecast, statistics
  - Assume the WeatherData class has access to these data members:
    - currentConditionsDisplay, forecastDisplay, statisticsDisplay

WeatherData class (attempt 1)

Write code for the measurementsChanged() method:

```
class WeatherData
     def intialize()
          @conditionsDisplay = DisplayConditions.new()
          @statisticsDisplay = DisplayStats.new()
          @forecastDisplay = DisplayForecast.new()
     end

     def measurementsChanged()
          #ADD CODE HERE
          #get the latest values and call the three displays
     end
     def getTemperature() #pulls info from some hardware
     def getPressure()
     def getHumidity()
end
```

WeatherData class (attempt 1)

```
class WeatherData
    # ...
    def measurementsChanged()
        temp =  self.getTemperature()
        humidity = self.getHumidity()
        pressure = self.getPressure()
        @currentConditionsDisplay.updateConds(temp, humidity, pressure)
        @statisticsDisplay.updateStats(temp, humidity, pressure);
        @forecastDisplay.updateForecast(temp, humidity, pressure);
    end
    # ...
end
```

WeatherData class (attempt 2)

```
# improved with a common interface
class WeatherData
    # …
    def measurementsChanged()
        temp =  self.getTemperature()
        humidity = self.getHumidity()
        pressure = self.getPressure()
        @currentConditionsDisplay.update(temp, humidity, pressure)
        @statisticsDisplay.update(temp, humidity, pressure)
        @forecastDisplay.update(temp, humidity, pressure)
    end
    # …
end
#Now, what if 5 new display devices need to be added.
#What parts of the code need to change?
```

WeatherData class: rank the design

- Cons
  - For every new display element we need to alter code.
  - We have no way to add (or remove) display elements at run-time.
  - We haven’t separated out the part that changes.
- Pros:
  - The display elements implement a common interface
    - as a result the same update(...) method is being called

Listeners

- When the button is clicked on a webpage, a specific function that is set to listen on an element fires when an event, the button being pressed, occurs
  - and prints “Button was clicked”, for example
- The object that listens for the event is an **observer**

Observer Pattern (1)

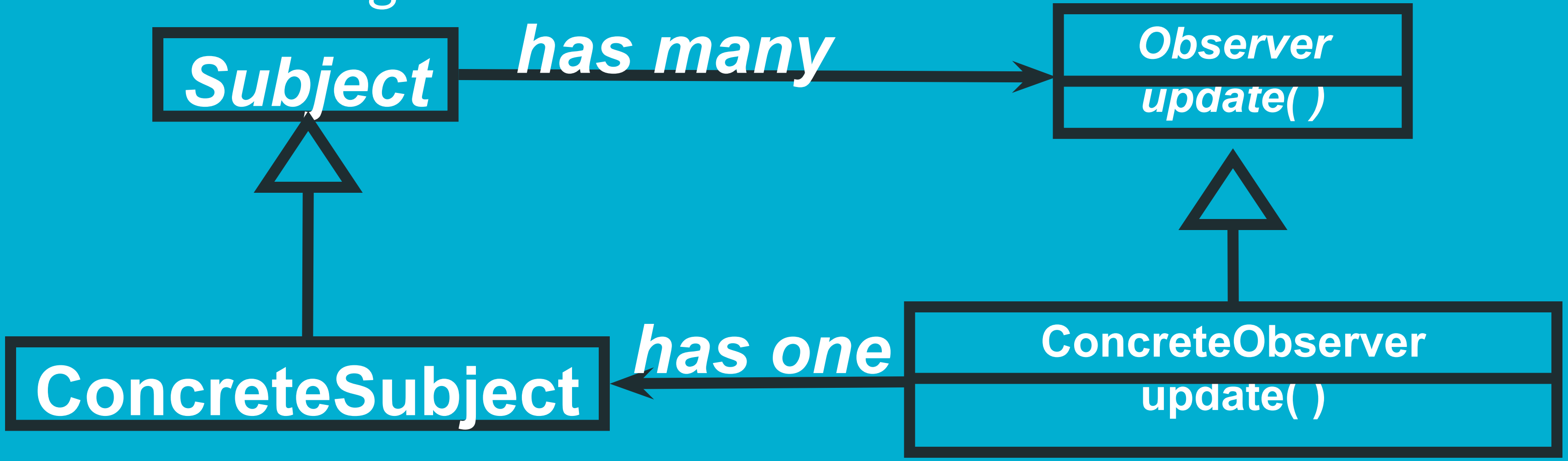
- Several objects depend on a single common object with state that will change as the program runs
    - Note that the design needs to scale well as we increase in number of state changes and observers
- Each dependent object needs to know when the state changes without tight coupling of objects
    - **Tight coupling** occurs when one object depends on concrete, not abstract behaviour (black boxed), of another object

Observer Pattern (2)

- Defines a one-to-many dependency between objects
  - when one object changes state all its dependents are notified and updated automatically
- Subject
  - The object that changes state and needs to notify others is called the **subject**.
- Observer
  - The Observer interface must be implemented by those objects wishing to be informed of the changes in the subject

Structure

- The subject has the responsibility to notify all the observers when a change occurs



Examples of observer pattern

- Newspaper publishes newspapers
    - users subscribe whenever a new edition is published
    - users can un-subscribe
- More examples?
    - chat messages
    - anyone use feedly?
- Publishers + subscribers = Observer Pattern

Observer Pattern: definition

- Defines a one-to-many dependency between objects so that when one object changes, all of its dependents are notified and updated automatically

Subject and Observer

- Abstract Class Subject
  - What is the subject in our WeatherData example?
    - ★ WeatherData class
  - What methods should a subject have?
    - ★ notifyObservers(), registerObserver(), removeObeserver()
- Interface Observer
  - What is the Observer in our WeatherData example?
    - ★ Displays
  - What methods should they have?
    - ★ update()

Observer Pattern Base Interface

```
class Subject
     def intialize() #abstract
     def registerObserver(observer)
     def removeObserver(observer)
     def notifyObservers() #parameter?
end

class Observer
     def update(temp, humidity, …)
     #can you make the parameters generic? what if the observable parameters change?
end

class DisplayElement
     def display()
end
```

WeatherData class design

- Should it be a subclass of an observable (Subject) interface?
  - It should have methods from the Subject interface (along with measurementChanged, setMeasurements, etc)
- What data members should it have?
  - It should have a data member to store the list of observers
- What should the constructor do?
  - The constructor should initialize the list of observers (array list to null);

WeatherData Subject

Write code for the initialize() and measurementsChanged() method:

```
class WeatherData
    def intialize()
        @observers[]
    end
    def measurementsChanged()
        @observers.each do |o|
            o.update(self)
        end
    end
    ...
end
```

Subject Runtime Observers

Write code for the initialize() and measurementsChanged() method:

```
class WeatherData
    ...
    def registerObserver(obs)
        @observers << obs
    end
    def deregisterObserver(obs)
        @observers.delete_if{|o| o == obs}
     end
    ...
end
```

ForecastDisplay Observer

- Which interface(s) should it implement?
    - The observer here also needs to implement the observer interface of update()
- What should be passed to it in the constructor?
    - In the constructor, pass it a reference to the Subject, so the observer can register/un-register later on.

ForecastDisplay Observer

Write code for the initialize(), update(), and display() method:

```
class ForecastDisplay
    def initialize(observable)
        @subject = observable
    end
    def update(subject)
        #update stuff
    end
    def display()
        #display stuff
    end
end
```

Observer: Push Interface

- The subject calls update whenever there is a change and passes the data on
- CONS:
  - The observers have no control over when the data is sent to them
  - Also, the observers cannot choose the data that is sent to them
    - what if an observer is interested in just the temperature data?
- PROS:
  - Observer is notified immediately upon changes

Observer: Pull Interface

- It is possible to change the interface to be “pull” based
- Do not need to change the update() call at the subject.
- Observer decides when and what it wants by calling update and passing the subject
    - Observers can pull the information whenever they like
        - example: every 15 minutes to conserve network data and power
- Observers can pull a subset of the information, if they are not interested in all state changes in the subject

Implementing Pull Notifications

```
class Observer
    …
    def pull()
        @poll = Thread.new {
            self.update(@subject.only_needed_info)
            #sleep for 15 minutes
        }
    end
...
end
```

Observable Relationship

- What kind of relationship occurs between the observer and the subject?
  - The subject is not an observer or vice-versa, so inheritance is out
  - Because Ruby is single inheritance, if we define the subject as a subclass of the Observer, we cannot define it as anything else
    - Solution : modules
- The subject HAS-A observer
  - This means we need to look at composition

Ruby has built-in Observer Pattern

- Observable is so standard, Ruby has library module
  - just use: ‘include Observable’ in your class
  - http://docs.ruby-lang.org/en/2.2.0/Observable.html
- Has a call to update all observers
  - notify_observers(*args) #*args means variable number of arguments
- Has a changed flag
  - The changed flag is useful so the subject can control when the notify_observers() should be called. It can wait for the changes to be specific (for example only when the temperature changes, will it notify).

Modules

- Modules are like chunks of readymade code you can use in your classes
  - Observable is a module not a class
  - you can define your own module, and they are defined very similar to classes
- Modules are categories of behaviour rather than models of objects
  - Modules are perfect for HAS-A relationships
- Modules cause problems when you call super()
  - HOMEWORK: What’s this problem?

Classwork:
News Preferences

Classwork: Spreadsheet

Classwork:
Game Designer

Classwork:
University Admissions