The Observer Pattern

Event-driven design will set you free
BDD-style Testing

This is feature testing, not unit testing
(yes, you have to do it this way on the project)
Example: Game of Life

**Given** a 5 by 5 game
**When** I toggle the cell at (2, 3)
**Then** the grid should look like

```
.....
.....
....
.X..
.....
```

**When** I toggle the cell at (2, 4)
**Then** the grid should look like

```
.....
.....
....
.X.. 
.X.. 
```

**When** I toggle the cell at (2, 3)
**Then** the grid should look like

```
.....
.....
....
.X..
.X.. 
```

**Asides:**
- Gives actual inputs and outputs
- Progressive when-then clauses
- Follow BDD scenario’s structure and content when forming a testcase
- Each “Given”, “When”, and “Then” will have its own method
  - **Given**: initialization
  - **When**: action to be tested (do the action)
  - **Then**: check that action made intended change
- Make the test sound like the scenario
// Given a $width by $height game
void gameIsRunning(int width, int height) {
  game = new Game(width, height);
  renderer = new StringRenderer();
  game.registerObserver(renderer);
}

// When I toggle the cell at ($column, $row);
void iToggleTheCellAt(int column, int row) {
  game.toggleCellAt(column, row);
}

// Then the grid should look like $grid
void theGridShouldLookLike(String grid) {
  assertThat(renderer.asString(),
      equalTo(grid));
}
Example: Game of Life, III

Scenario 1: ...

Given a 5 by 5 game
When I toggle the cell at (2, 3) Then the grid should look like

.....
.....
.X..
.....
When I toggle the cell at (2, 4) Then the grid should look like

.....
.....
.X..
.X..
When I toggle the cell at (2, 3) Then the grid should look like

.....
.....
.X..

I've asserted an alphabetic ordering of the tests (and named them thusly) so that the progression holds up.

Note how the test cases look and sound like the scenario.

More test cases can be written with same base methods.

True BDD writes test cases before the code. (I don't.)
Observer pattern

Event-driven programming, call-backs, listening, and all that
What’s an event?
- It’s a kind of broadcast message that can be “heard” by any object that chooses to listen
- A typical event message from an object is “I changed!”
- This enables objects who depend on that object to update themselves
- E.g., update the map’s center when location changes; GPS’s event message is, in essence, “I changed!”

Android depends heavily on events
- Every sensor on phone sends events
- Like above: GPS periodically “announces” changes in location

User interfaces run almost entirely on events
- Every mouse motion, click, etc.,

But wait a second, Java doesn’t have events!
Simulating Events with **Callbacks**

- Classic software technique (OS, games, OOD)
- If the map control wants to receive events from location manager, it “registers” with it
  - *conceptually*: `locationMgr.register(mapControl)`
- Later, when location changes, `locationMgr` “calls back” `mapControl` (and all others who registered):
  - *conceptually (in locationMgr)*:
    ```java
    foreach (LocListener L : Listeners) { L.changed(this); }
    ```
- This is how we implement events with methods
- Note: *many details omitted, but forthcoming*
// Acquire reference to the Location Manager
LocationManager locationManager = (LocationManager)
this.getSystemService(Context.LOCATION_SERVICE);

// Create a callback that handles location updates
LocationListener locationListener = new LocationListener()
{
  public void onLocationChanged(Location location) {
    // Called when a new location is found by the network location provider.
    makeUseOfNewLocation(location);
  }
}

// Register the callback with Location Manager to receive location updates
locationManager.requestLocationUpdates(
  LocationManager.NETWORK_PROVIDER, 0, 0, locationListener);
Why go to all this trouble? Knows About

- We say component A **knows about** component B if A references an interface defined by B

- In example below, **WeatherData knows about** Weather Station and Display One, Two, & Three
What’s wrong with this design?

A. update() for each Display should take only the parameters it needs

B. Will have to edit WeatherData to add a new Display

C. Will have to edit WeatherData to add a new weather measurement

Because WeatherData knows about each individual display, adding a display means that WeatherData has to be updated. Unfortunately, we anticipate adding more displays (e.g., more users)
We have a knows-about problem

- Because `WeatherData` knows about the individual displays, adding a new display requires changing `WeatherData` as well.

- Violates our design goal of localizing change.
  - (Adding features shouldn’t affect *internals* of a class.)

```java
public void measurementsChanged() {
    float temp = getTemperature();
    float humidity = getHumidity();
    float pressure = getPressure();

    currentConditionsDisplay.update(temp, humidity, pressure);
    statisticsDisplay.update(temp, humidity, pressure);
    forecastDisplay.update(temp, humidity, pressure);
    forecastDisplay.update(temp, humidity, pressure);
```
public class WeatherData {

    // instance variable declarations

    public void measurementsChanged() {

        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();
    }

}  

- It’s design patterns, it’s gotta involve interfaces

currentConditionsDisplay.update(temp, humidity, pressure);
statisticsDisplay.update(temp, humidity, pressure);
forecastDisplay.update(temp, humidity, pressure);

- But have to handle 0, 1, 2, 3... displays

Grab the most recent measurements by calling the WeatherData’s getter methods (already implemented).

Hey, these all kind of have the same interface!
currentConditionsDisplay.update(temp, humidity, pressure);
statisticsDisplay.update(temp, humidity, pressure);
forecastDisplay.update(temp, humidity, pressure);

for (weatherUpdateInterface : weatherUpdateInterfaces)
    weatherUpdateInterface.update(temp, humidity, pressure);

- Now fully dynamic – can add and delete any time, as many or few as we want
- But how does WeatherData get access to the weatherUpdateInterface objects to call?
Three interfaces (first two in the pattern)

Since update() is kind of specific, I'd call this WeatherObserver Interface for the thing being observed

public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}

public interface Observer {
    public void update(float temp, float humidity, float pressure);
}

public interface DisplayElement {
    public void display();
}

Both of these methods take an Observer as an argument; that is, the Observer to be registered or removed.

This method is called to notify all observers when the Subject’s state has changed.

The Observer interface is implemented by all observers, so they all have to implement the update() method. Here we’re following Mary and Sue’s lead and passing the measurements to the observers.

These are the state values the Observers get from the Subject when a weather measurement changes.

Not part of pattern, don't really need it

The DisplayElement interface just includes one method, display(), that we will call when the display element needs to be displayed.
Refactored Design

```java
public class WeatherData implements Subject {
    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            observers.get(i).update(i);
        }
    }

    public void measurementsChanged() {
        notifyObservers();
    }
}
```

```java
public class ForecastDisplay implements Observer, DisplayElement {
    private float currentPressure = 29.92f;
    private float lastPressure;
    private Subject weatherSubject;

    public ForecastDisplay(Subject weatherSubject) {
        this.weatherSubject = weatherSubject;
        weatherSubject.registerObserver(this);
    }

    public void update(float temp, float humidity, float pressure) {
        lastPressure = currentPressure;
        currentPressure = pressure;
        display();
    }

    public void display() {
        System.out.println("Current Pressure: "+ pressure);
    }
}
```

```java
public static void main(String[] args) {
    WeatherData weatherData = new WeatherData();
    StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
    ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

    weatherData.setMeasurements(80, 65, 30.4f);
    weatherData.setMeasurements(82, 70, 29.25f);
}
```
All our weather components implement the Observer interface. This gives the Subject a common interface to talk to when it comes time to update the observers.

There will be a few changes to make to the update() method in the concrete Observers, but basically it's the same idea... we have a common Observer interface, with an update() method that's called by the Subject.

WeatherData now implements the Subject interface.
Coupling in Observer

- **Coupling** is the property that when one thing changes, another thing may have to change (they are coupled).

- Want *loosely coupled* designs.

- Note that Observer is “coupled” to weather generically, and hence to WeatherData generally.

- But the designer of WeatherData chose the Observer interface.

- ForecastDisplay can’t be implemented until Observer has been worked out.

- We have another pattern later that lets ForecastDisplay be preexisting.

- Critical for using existing dials, etc.
// Acquire reference to the Location Manager
LocationManager locationManager = (LocationManager) 
this.getSystemService(Context.LOCATION_SERVICE);

// Create a listener that handles location updates
LocationListener locationListener = new LocationListener() {
    public void onLocationChanged(Location location) {
        // Called when a new location is found by the network location provider.
        makeUseOfNewLocation(location);
    }

    public void onStatusChanged(String provider, int status, Bundle extras) {} 
    public void onProviderEnabled(String provider) {} 
    public void onProviderDisabled(String provider) {} 
};

// Register the listener with the Location Manager to receive location updates
locationManager.requestLocationUpdates(
    LocationManager.NETWORK_PROVIDER, 0, 0, locationListener);

• Breaks the pattern on the subject side
  • There’s no Subject interface implemented by LocManager
  • Makes building new location subjects harder
  • But Android really only needs one, so why bother? (tradeoffs)
• Also, concrete LocListener doesn’t have ref to subject to unregister
  • Looks like creator of concrete LocListener has that reference
public class MainActivity extends Activity implements LocationListener {

    public void onLocationChanged(Location location) {
        // Called when new location is found by network location provider.
        makeUseOfNewLocation(location);
    }

    public void onStatusChanged(String provider, int status, Bundle extras) {}
    public void onProviderEnabled(String provider) {}
    public void onProviderDisabled(String provider) {}

    ...

    • Now the concrete LocationListener has a name
    • Now it’s the “outer” class that is LocationListener
      • Means it has reference to LocationManager to unregister itself
    • Still no Subject interface implemented by LocationManager
Take-Aways

- **Revisited lessons:**
  - Classes: *bad*; Interfaces: *good*
  - *Program to interfaces, not implementations*
  - Aggregate/compose, don’t subclass

- **New lessons:**
  - Advanced communication concepts can be emulated (*callbacks*: events with methods)
  - Observer pattern is an elegant, reusable solution
    - Use of interfaces ensures that (changeable) concrete classes only refer to (stable) interfaces
    - Gets right “knows about” relationship
    - Localization of change (low coupling)