Observers and Adapters

Two simple patterns

Observe What?

- It’s common for code to care about the state of other objects
- Usually you want to execute specific logic when that state changes
- Adding this logic to the objects being modified introduces a maintenance debt

Inverted Responsibility

When class A needs to be aware of the state of class B, it shouldn't be B's responsibility to update A.

```
public class Stock {


        private String id;
        private BigDecimal price;
        private Application application;


        public Stock(String id, BigDecimal price, Application application) {
                if(id == null || id.trim().length() == 0){
                        throw new IllegalArgumentException("Empty/null identifiers not allowed.");
                }
                this.id = id;
                this.application = application;
                setPrice(price);
        }
```

Bad Solution: Stock is now responsible for telling Application when Stock has changes. What happens when 2 classes care about the state of Stock? 5? 10?

What needs to happen

Observer - exposes logic that gets executed when state changes

Observable - exposes way for Observers to register themselves, triggers all registered Observers' exposed logic when state changes

Java Implementation

Observer: interface with a update() method where you put the logic you want to execute when changes are made to the Observable class.

The Observer is responsible for adding itself to the list of Observers of the Observable.

Java Implementation

Observable: interface declares addObserver (Observer obs) method to allow Observers to register themselves.

Observable must call setChanged() and notifyObservers() when state changes. This call the update() of all registered Observers.

Adapter

Often times you'll have objects or data that contains the same information in incompatible formats.

This is usually a result of them being developed for separate systems, or one instance containing more information than the other.

I’m Afraid of Change

The Adapter pattern exists for when changing the behavior of those core Classes that have been around in your codebase for the last two decades isn’t an option.

Find the steps needed to turn a class/datum of instance A into a class/datum of instance B and implement them in your adapter: class C.

Example

```
public interface StockObserver {

        public void onUpdate(Stock stock);

}
```

Class A

```
public interface StockObserver2 {

        public void onUpdate(Stock before, Stock after);

}
```

Class B

```
public class StockObserverAdapter implements StockObserver {

        Stock lastSeen;
        StockObserver2 observer;

        public StockObserverAdapter(StockObserver2 observer){
                this.observer = observer;
        }


        @Override
        public void onUpdate(Stock stock) {
                observer.onUpdate(lastSeen, stock);
                lastSeen = new Stock(stock.getId(), stock.getPrice());
        }


}
```

Steps to turn B into A