Project 2

- We will provide you with one good implementation and 3 bad implementations of the Observer pattern
- You need to write test cases
- We have another 3 good implementations and 20 bad implementations waiting in the wings
- Your test cases need to discriminate the good and bad implementations
Observer pattern

• Some object has state that changes (the Subject or the Observable)
  – update readings on a instrument
  – button clicked on a GUI
  – you have mail!
  – stock price goes outside of a range

• Lots of clients that might be interested in those changes (the Observers)
  – some of which will be written after the code for the subject is finalized
Loose coupling (plan for change)

- The only thing the subject knows about the observer is that it implements the observer interface
- We can add new observers at any time (during runtime)
- We don’t need to modify the subject to add new types of observers
- We can reuse subjects/observers independently
- Changes to subject or observer will not affect the other
Observer Pattern

`Subject`
- Attach(Observer)
- Detach(Observer)
- Notify()

`ConcreteSubject`
- GetState()
- SetState()
- subjectState

`Observer`
- Update()

`ConcreteObserver`
- Update()
- observerState

for all o in observers {
  o->Update()
}

observerState = subject->GetState()
Use of Observer Pattern

```
CONCRETE_SUBJECT

CONCRETE_OBSERVER

ANOTHER_CONCRETE_OBSERVER

setState()

notify()

update()

getState()

update()

getState()
```
Observer Pattern Consequences

• Low coupling between subject and observers
• Subject unaware of dependents
• Support for broadcasting
• Dynamic addition and removal of observers
Implementation issues

• Storing list of observers
  – Typically in subject

• Observing multiple subjects
  – Typically add parameters to update()

• Who triggers update?
  – State-setting operations of subject
    • Possibly too many updates
  – Client
    • Error-prone if an observer forgets to send notification message
Using an anonymous inner class to find out with Subject is updating you.

class ManyObserver {
    // method I want called
    void update(Subject s) { ... }

    void listenTo(final Subject s) {
        s.addObserver(
            new Observer() {
                public void update() {
                    update(s);
                }
            }));
    }
}
Implementation issues (cont’d)

• Possibility of dangling references when subject is deleted
  – Easier in garbage-collected languages
  – Subject notifies observers before dying

• How much information should subject send with update() messages?
  – Push model: Subject sends all information that observers may require
    • May couple subject with observers (by forcing a given observer interface)
  – Pull model: Subject sends no information
    • Can be inefficient
 Implementation issues (cont’d)

• Registering observers for certain events only
  – Use notion of an aspect in subject
  – Observer registers for one or more aspects

• Is Observable reentrant?
  – what if, while an Observer is being notified, the Observer asked to be unsubscribed?