Proxy Pattern Motivation

- **Goal:**
  - Prevent an object from being accessed directly by its clients
- **Solution:**
  - Use an additional object, called a proxy
  - Clients access protected object only through proxy
  - Proxy keeps track of status and/or location of protected object

Uses of the Proxy Pattern

- **Virtual proxy:** impose a lazy creation semantics, to avoid expensive object creations when strictly unnecessary.
- **Monitor proxy:** impose security constraints on the original object, say by making some public fields inaccessible.
- **Remote proxy:** hide the fact that an object resides on a remote location.

Template Method pattern

- **Problem**
  - You’re building a reusable class
  - You have a general approach to solving a problem,
  - But each subclass will do things differently
- **Solution**
  - Invariant parts of an algorithm in parent class
  - Encapsulate variant parts in template methods
  - Subclasses override template methods
  - At runtime template method invokes subclass ops
Example: JUnit

- JUnit uses template methods pattern for run()
  ```java
  package junit.framework;
  public class TestCase {
      public void run() {
          setUp(); runTest(); tearDown()
      }
  }
  ```
  - In the class example, we subclass TestCase and override setUp() and tearDown()

Observer pattern

- Concrete subject
  - Actual object of interest. Notifies observers when something happens.
- Concrete observers
  - Actual objects interested in subject. Will perform appropriate task when notified that something interesting has happened.

Project 1 Event Notification (almost an observer)

- Abstract subject: Dispatcher
  - Maintains a list of other EventProcessors that are interested in the events it receives
  - Notifies objects when its process() method is called
- Abstract observer: EventProcessor
  - Observers are notified via their process() method.
  - Interesting twist: a Dispatcher is also an observer!
Concrete subject: Timestamper (say)  
- Notifies observers by calling super.process() whenever it receives an event. Passes information of interest to each observer via the observer’s process() call.

Concrete observer: EventProcessor subclasses  
- Act on events that they receive from process().

Design Decisions
- Dispatcher is not an abstract class  
  - You can create a dispatcher just to forward events, you don’t have to subclass it (though it’s useful to do so)
- Concrete Observers receive the events of interest as arguments to their process() method  
  - Could also have that method call back to the concrete subject to acquire the event
- Subjects and Observers share the same superclass  
  - Notify and are notified using the same method: process(). Notification happens when event received.
- Could do things differently …

Consequences
- Low coupling between subject and observers  
  - Subject indifferent to its dependents; can add or remove them at runtime
- Support for broadcasting
- Unexpected updates  
  - Subject not tied to computations by 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

Implementation Issues (cont’d)

- How much information should subject send with update() messages?
  - Push model (event notification): subject sends all information that observers may require
  - Pull model: Subject sends no information
- Registering observers for certain events only
  - Have filter objects, like Project 1
  - Build notion of filter into Abstract subject; clients include filters when attaching

Decorator Pattern

- Motivation
  - Want to add responsibilities/capabilities to individual objects, not to an entire class.
  - Inheritance requires a compile time choice of parent class.
- Solution
  - Enclose the component in another object that adds the responsibility/capability
    - The enclosing object is called a decorator

Decorator Pattern: Example

Decorator Pattern: Features

- A decorator conforms to the interface of the component it decorates
  - so that its presence is transparent to the component’s clients.
- A decorator forwards requests to its encapsulated component and may perform additional actions before or after forwarding.
- Can nest decorators recursively, allowing unlimited added responsibilities.
- Can add/remove responsibilities dynamically.

Structure
Decorator Pattern Analysis

• Advantages
  – fewer classes than with static inheritance
  – dynamic addition/removal of decorators
  – keeps root classes simple

• Disadvantages
  – proliferation of run-time instances
  – abstract Decorator must provide common interface

• Tradeoffs:
  – useful when components are lightweight

Example 1: Java I/O

```java
FileReader frdr = new FileReader(filename);
LineNumberReader lrdr = new LineNumberReader(frdr);
String line;
while ((line = lrdr.readLine()) != null) {
    System.out.println(lrdr.getLineNumber() + ":\t" + line);
}
```

Example 2: Project 1 EventProcessors

```
EventProcessor
  process(Event e)

Printer
  process(Event e)
  Print(e);

Dispatcher
  process(Event e)
  e = mystuff(e)
  super.process(e)

TimeStamp
  process(Event e)
      ...

Filter
  process(Event e)
      ...
```

Each concrete subclass adds its own special behavior

State pattern

• Problem
  – An object is always in one of several known states
  – The state an object is in determines the behavior of several methods

• Solution
  – Could use if/case statements in each method
  – Better: use dynamic dispatch

State pattern Approach

• Encode different states as objects with the same superclass
• To change state, change the state object
• Methods delegate to state object
Example – Finite State Machine

class FSM {
    State state;
    public FSM(State s) { state = s; }  
    public void move(char c) { state = state.move(c); }  
    public boolean accept() { return state.accept(); }  
}

public interface State {
    State move(char c);  
    boolean accept();  
}

FSM Example – cont.

class State1 implements State {
    static State1 instance = new State1();  
    private State1() {}  
    public State move(char c) {
        switch (c) {
            case 'a': return State2.instance;  
            case 'b': return State1.instance;  
            default: throw new IllegalArgumentException();  
        }
    }
    public boolean accept() {return false;}  
}

class State2 implements State {
    static State2 instance = new State2();  
    private State2() {}  
    public State move(char c) {
        switch (c) {
            case 'a': return State1.instance;  
            case 'b': return State1.instance;  
            default: throw new IllegalArgumentException();  
        }
    }
    public boolean accept() {return true;}  
}

Instance of State Pattern

State pattern notes

- Can use singletons for instances of each state class  
  – State objects don’t encapsulate (mutable) state, so can be shared  
- Easy to add new states  
  – New states can extend the base class, or  
  – New states can extend other states  
  * Override only selected functions  

Structure of State pattern