Some Design Patterns
March 4, 2003

Singleton objects

- Some classes have conceptually one instance
  - Many printers, but only one print spooler
  - One file system
  - One window manager
- Naïve: create many objects that represent the same conceptual instance
- Better: only create one object and reuse it
  - Encapsulate the code that manages the reuse

The Singleton solution

- Class is responsible for tracking its sole instance
  - Make constructor private
  - Provide static method/field to allow access to the only instance of the class
- Benefit:
  - Reuse implies better performance
  - Class encapsulates code to ensure reuse of the object; no need to burden client

Singleton pattern

- Class is responsible for tracking its sole instance
- Make constructor private
- Provide static method/field to allow access to the only instance of the class
- Benefit:
  - Reuse implies better performance
  - Class encapsulates code to ensure reuse of the object; no need to burden client

Implementing the Singleton method

- In Java, just define a final static field
  public class Singleton {
    private Singleton() {...}
    final private static Singleton instance = new Singleton();
    public Singleton getInstance() { return instance; }
  }
- Java semantics guarantee object is created immediately before first use

Generalizing Singleton: Typesafe Enum

- Problem:
  - Need a number of unique objects, not just one
  - Basically want a C-style enumerated type, but safe
- Solution:
  - Generalize the Singleton Pattern to keep track of multiple, unique objects (rather than just one)
Typesafe Enum Pattern

```
enum Enum {
    static Enum inst1;
    static Enum inst2;
    EnumOp ()
    data
}
```

Note: constructor is private

Typesafe Enum: Example

```
public class Suit {
    private final String name;
    private Suit(String name) { this.name = name; }
    public String toString()
    {
        return name;
    }
    public static final Suit CLUBS = new Suit("clubs");
    public static final Suit DIAMONDS = new Suit("diamonds");
    public static final Suit HEARTS = new Suit("hearts");
    public static final Suit SPADES = new Suit("spades");
}
```

Adapter Motivation

• Situation:
  – You have some code you want to use for a program
  – You can’t incorporate the code directly (e.g. you just have the .class file, say as part of a library)
  – The code does not have the interface you want
    • Different method names
    • More or fewer methods than you need
  • To use this code, you must adapt it to your situation

Adapter pattern

```
Client
  Request
  Target
    Request
  Adapter
    Request -> adapter
    adapter -> adapter
    adapter -> adapter
    adapter
```

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.
• 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
  ```java
class JUnitFramework.TestCase
```
  ```java
    run() {
      setUp(); runTest(); tearDown()
    }
  ```
  ```java
  ```
- In class example, subclass overrides runTest() and setUp()
Observer pattern

- Problem
  - dependent’s state must be consistent with master’s state
- Solution structure
  - define four kinds of objects:
    - abstract subject
      - maintain list of dependents, notifies them when master changes
    - abstract observer
      - define protocol for updating dependents
    - concrete subject
      - manage data for dependents, notifies them when master changes
    - concrete observers
      - get new subject state upon receiving update message

Use of Observer pattern

Observer Pattern (cont’d)

- Consequences
  - low coupling between subject and observers
    - subject unaware of dependents
  - support for broadcasting
    - dynamic addition and removal of observers
  - unexpected updates
    - no control by the subject on computations by observers

Observer pattern (cont’d)

- 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

Observer pattern (cont’d)

- Implementation issues (cont’d)
  - possibility of dangling references when subject is deleted
    - easier in garbage-collected languages
  - subject notifies observers before dying
  - possibility of premature notifications
    - typically, method in Subject subclass calls inherited method which does notification
    - solve by using Template method pattern
      - method in abstract class calls deferred methods, which is defined by concrete subclasses
Observer pattern (cont’d)

- Implementation issues (cont’d)
  - 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
  - registering observers for certain events only
    - use notion of an aspect in subject
    - Observer registers for one or more aspects

- Implementation issues (cont’d)
  - complex updates
    - use change managers
    - change manager keeps track of complex relations among (possibly) many subjects and their observers and encapsulates complex updates to observers

Implementation details

- Observing more than one subject.
  - It might make sense in some situations for an observer to depend on more than one subject. The subject can simply pass itself as a parameter in the Update operation, thereby letting the observer know which subject to examine.
  - Making sure Subject state is self-consistent before notification.

More implementation issues

- Implementations of the Observer pattern often have the subject broadcast additional information about the change.
  - At one extreme, the subject sends observers detailed information about the change, whether they want it or not. At the other extreme the subject sends nothing but the most minimal notification, and observers ask for details explicitly thereafter
  - You can extend the subject's registration interface to allow registering observers only for specific events of interest.

Examples

- The standard Java and JavaBean event model is an example of an observer pattern

State pattern

- Suppose an object is always in one of several known states
  - The state an object is in determines the behavior of several methods
  - Could use if/case statements in each method
  - Better solution: state pattern
State pattern

- Have a reference to a state object
  - Normally, state object doesn’t contain any fields
  - Change state: change state object
  - Methods delegate to state object

Structure of State pattern

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

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(); }
}

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; }
}