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
Implementing the Singleton method

• In Java, just define a final static field

```java
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

```
public class Enum {
    EnumOp ()
    enum EnumOp {
        Data
    }
}
```

Note: constructor is private

Typesafe Enum: Example

```java
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**

- **Clients needs a target that implements one interface**

**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: AbstractMap

- Map interface has 12 methods
  - can be a pain to implement all of them
- Can extend AbstractMap
  - provides implementations of most methods in terms of a few key methods
- Can implement read-only Map by just implementing `entrySet` method

Example: JUnit

- JUnit uses template methods pattern
  
  ```java
  JUnit.framework.TestCase.run() {
    setUp(); runTest(); tearDown()
  }
  ```
  - 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; notify them when master changes
    - abstract observer
      - define protocol for updating dependents
    - concrete subject
      - manage data for dependents; notify them when master changes
    - concrete observers
      - get new subject state upon receiving update message
Observer pattern

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 updater()
  - 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

Observer pattern (cont’d)

• 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
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

```java
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(); }
}
```

```java
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;}
}

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: 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: Example

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
  - Otherwise use Strategy

Example: Java I/O

```java
FileReader frdr = new FileReader(filename);
LineNumberReader lrdr = new LineNumberReader(frdr);
String line;
while ((line = lrdr.readLine()) != null) {
    System.out.print(lrdr.getLineNumber() + ":	" + line);
}
```
Lexi: Simple GUI-Based Editor

- Lexi is a WYSIWYG editor
  - supports documents with textual and graphical objects
  - scroll bars to select portions of the document
  - be easy to port to another platform
  - support multiple look-and-feel interfaces
- Highlights several OO design issues
- Case study of design patterns in the design of Lexi

Design Issues

- Representation and manipulation of document
- Formatting a document
- Adding scroll bars and borders to Lexi windows
- Support multiple look-and-feel standards
- Handle multiple windowing systems
- Support user operations
- Advanced features
  - spell-checking and hyphenation

Structure of a Lexi Document

- Goals:
  - store text and graphics in document
  - generate visual display
  - maintain info about location of display elements
- Caveats:
  - treat different objects uniformly
    - e.g., text, pictures, graphics
  - treat individual objects and groups of objects uniformly
    - e.g., characters and lines of text

Structure of a Lexi Document

- Use recursive composition for defining and handling complex objects
  - Abstract class Glyph for all displayed objects
  - Glyph responsibilities:
    - know how to draw itself
    - knows what space it occupies
    - knows its children and parent
  - Glyph instances can recursively compose other Glyph instances
Recursive Composition

User Display

Objects

Glyph Class Diagram

The Composite Pattern

- Motivation:
  - support recursive composition in such a way that a client need not know the difference between a single and a composite object (as with Glyphs)

- Applicability:
  - when dealing with hierarchically-organized objects (e.g., columns containing rows containing words ...)

Composite Pattern Structure
Composite Pattern Consequences

- Class hierarchy has both simple and composite objects
- Simplifies clients
- Aids extensibility
  - clients do not have to be modified
- Too general a pattern?
  - difficult to restrict functionality of concrete leaf subclasses

Formatting Lexi Documents: Strategy

- We know that documents are represented as Glyphs, but not how documents are constructed.
- Formatting:
  - Document structure will be determined based on rules for justification, margins, line breaking, etc.
  - Many good algorithms exist.
    - different tradeoffs between quality and speed
- Design decision: implement different algorithms, decide at run-time which algorithm to use
  - define root class that supports many algorithms
  - each algorithm implemented in a subclass

Strategy Pattern

- Name
  - Strategy (aka Policy)
- Applicability
  - many related classes differ only in their behavior
  - many different variants of an algorithm
  - need to encapsulate algorithmic information

Strategy Pattern: Structure
Strategy Pattern: Consequences

- Clear separation of algorithm definition and use
  - glyphs and formatting algorithms are independent
  - alternative (many subclasses) is unappealing
    - proliferation of classes
    - algorithms cannot be changed dynamically
- Elimination of conditional statements
  - Like State, Template, …
  - Typical in OO programming

Strategy Pattern Consequences (cont’d)

- Clients must be aware of different strategies
  - when initializing objects
- Proliferation of instances at run-time
  - each Glyph has a strategy object with formatting information
  - if strategy is stateless, share strategy objects

Lexi: Using Strategy

- Compositor and Composition classes
  - Compositor: class encapsulating formatting algorithm
    - pass Composition objects to be formatted as parameters to Compositor methods
  - Composition: things being formatted
    - Glyph subclass
    - Each Composition object refers to its Compositor object
    - When a Composition needs to format itself, it sends a message to its Compositor instance

Class Diagram
Adding Scroll Bars and Borders: Decorator

- How to define classes for scrollbars and borders?
- Define as subclasses of Glyph
  - Scrollbars and borders are displayable objects
  - Will use notion of transparent enclosure
    - Clients don’t need to know whether they are dealing with a component or with an enclosure
- Inheritance increases number of classes
  - Use composition instead (“has a”)

Transparent Enclosure

- Two features:
  - Single-child composition
    - Calls its child, then adds its own behavior
  - Compatible interfaces
    - Can use the enclosing object in place of the one it encloses
- Implemented by the Decorator pattern
  - Saw this earlier

Monoglyph class: a Decorator

```java
Class Monoglyph {...
  void Draw (Window w) {
    component.Draw(w);
    }
  }
Class Border extends Monoglyph {...
  void Draw (Window w) {
    super.Draw(w);
    DrawBorder(w);
    }
  }
```

Changing look-and-feel: Abstract Factory

- Goal: easily change Lexi’s look-and-feel
  - When new libraries are available (future variability)
  - At run-time by switching between them (present variability)
- Thoughtless implementation technique:
  - use distinct class for each widget and standard
  - let clients handle different instances for each standard
    - Button pb = new MotifButton(); // bad
Abstracting Creation

- Concrete Creation problems:
  - Class of object is fixed at compile-time
    - can’t change standard at run-time
  - Changing the class means making changes all over the code
- Instead:
  - Use a class to create abstract classes:
    - Button pb = guiFactory.createButton(); better

Solution: Use Abstract Factory

- Define abstract class GUIFactory with creation methods for widgets
  - Concrete subclasses of GUIFactory actually define creation methods for each look-and-feel standard
    - MotifFactory, MacFactory, etc.
  - Specialize each widget into subclasses for each look-and-feel standard
- Thus, can easily change the kind of factory without changes all over the place
Abstract Factory pattern

• Name
  – Abstract Factory or Kit

• Applicability
  – different families of components (products)
  – must be used in mutually exclusive and consistent way
  – hide existence of multiple families from clients

Abstract Factory: Consequences

• Isolate instance creation and handling from clients
• Can easily change look-and-feel standard
  – Reassign a global variable;
  – Recompute and redisplay the interface
• Enforce consistency among products in each family
• Adding to family of products is difficult
  – Have to update factory abstract class and all concrete classes

Structure of Abstract Factory

Multiple Window Systems

• Want portability to different window systems
  – similar to multiple look-and-feel problem, but different vendors will build widgets differently

• Solution:
  – define abstract class Window, with basic window functionality (e.g., draw, iconify, move, resize, etc.)
  – define concrete subclasses for specific types of windows (e.g., dialog, application, icon, etc.)
  – define WindowImp hierarchy to handle window implementation by a vendor
Implementation

[Diagram showing various window types and their relationships with each other, including ApplicationWindow, DialogWindow, kWinWindow, MacWindow, WinWindow, and others, each with specific methods like `DrawArea` and `DrawEffect`.]