Decorator Pattern: Features

- Decorator conforms to interface of decorated component
  - Its presence is transparent to the component's clients.
- Decorator forwards requests to encapsulated component
  - May perform additional actions before or after
- Can nest decorators recursively
  - Allows 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
  - Otherwise use Strategy

Decorator vs. Adapter

- A decorator adds to the responsibilities of an object
  - Usually has the same interface plus more features
- An adapter changes the interface
  - But usually not the responsibilities

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
Structure

Example: JUnit

• JUnit uses template methods pattern for run()

    package junit.framework;
    public class TestCase {
        ...
        public void run() {
            setUp(); runTest(); tearDown()
        }
    }

• In class example, we subclass TestCase and override setUp() and tearDown()

Observer Pattern

• Problem
  – Dependent must be consistent with master’s state
• Solution structure: 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

• 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 opinions
    • Client
      – Error-prone if an observer forgets to send notification message

• 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 nontrivial
  – 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.
**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 solution: use dynamic dispatch

**State Pattern Approach**

- Encode different states as objects with 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();
}

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; }
}

---

**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

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

Lexi User Interface

Design Issues

- Representation and manipulation of document
- Formatting a document
- Adding scroll bars and borders to Lexi windows
- Support multiple look-and-feel standards
  • Motif and Presentation Manager (!)
- 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
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 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 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**

1. **Context**
2. **Strategy**
3. **Consequences**

**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