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 state, so can be shared
- Easy to add new states
  - 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(); }
}

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;}
}

---

**Decorator Pattern**

- **Motivation**
  - Want to add responsibilities to individual objects, not to an entire class
  - Inheritance requires a compile-time choice
- **Solution**
  - Enclose the component in another object that adds the responsibility
    - The enclosing object is called a decorator.
  - Decorator conforms to the interface of the component it decorates so that its presence is transparent to the component's clients.
  - Decorator forwards requests to the component and may perform additional actions before or after forwarding.
  - Can nest decorators recursively, allowing unlimited added responsibilities.
  - Can add/remove responsibilities dynamically
Example: Java I/O

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

- Solution:
  - define abstract class Glyph for all displayed objects
  - glyph responsibilities:
    - know how to draw itself
    - knows what space it occupies
    - knows its children and parent
  - use recursive composition for defining and handling complex objects
  - define composition of Glyph as instances of Glyph

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 …)

Glyph Class Diagram
Composite Pattern (cont’d)

• 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

• Handle 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
• Goal: maintain orthogonality between formatting and
  representation (glyphs)
• Solution
  – define root class that supports many algorithms
  – each algorithm implemented in a subclass
Compositor and Composition

• Relevant design decisions:
  – compositor: class containing formatting algorithm
  – pass objects to be formatted as parameters to compositor methods
  – parameters are instances of a Glyph subclass called Composition
  – uniform interface between formattable objects and compositor algorithms

Compositor and Composition

• Relevant design decisions (cont’d):
  – each Composition instance has a reference to a compositor object
  – when a composition needs to format itself, it sends a message to its compositor instance
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

- **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
      - as usual with OO programming
Strategy Pattern (cont’d)

• Consequences (continued)
  – 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