

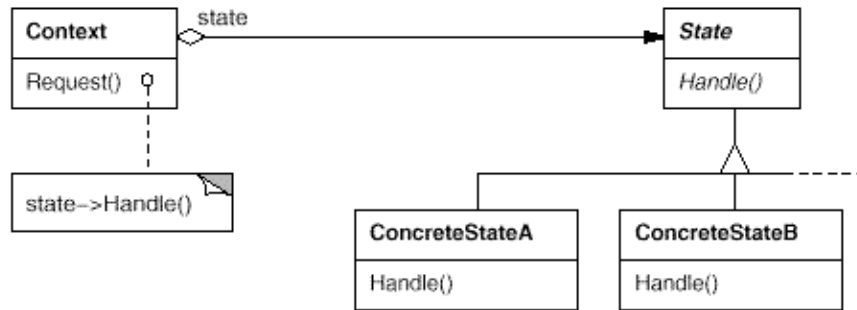
State pattern

- 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

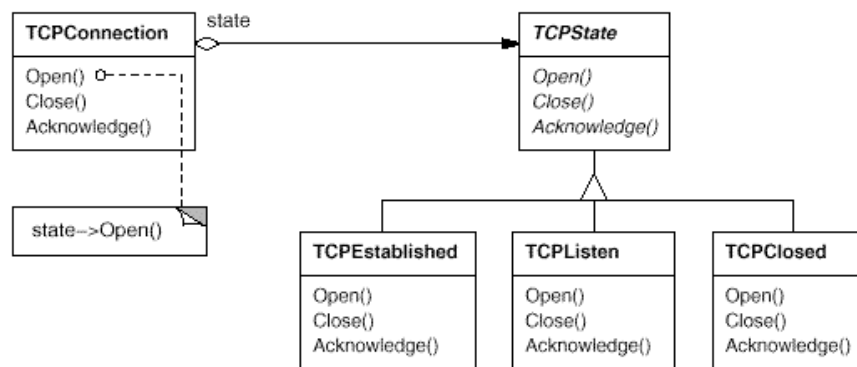
Structure of State pattern



CMSC 433, Fall 2002

53

Instance of State Pattern



CMSC 433, Fall 2002

54

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(); }
}

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;}
}
```

CMSC 433, Fall 2002

57

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**.

CMSC 433, Fall 2002

58

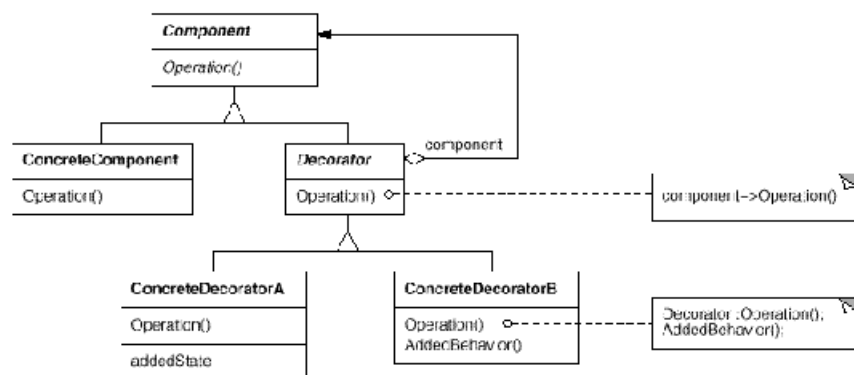
Decorator Pattern: Features

- 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.

CMSC 433, Fall 2002

59

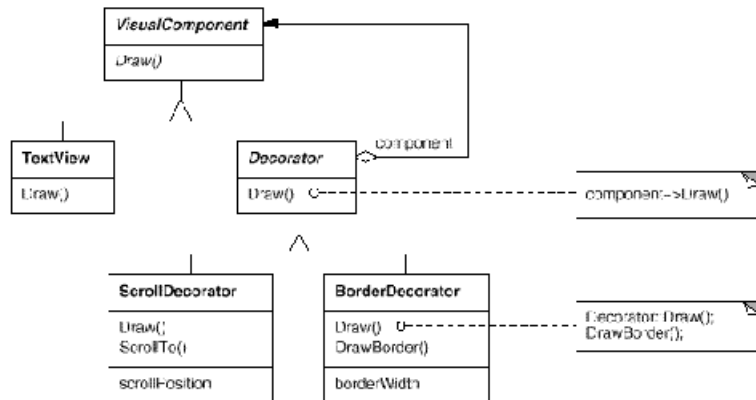
Structure



CMSC 433, Fall 2002

60

Decorator Pattern: Example



CMSC 433, Fall 2002

61

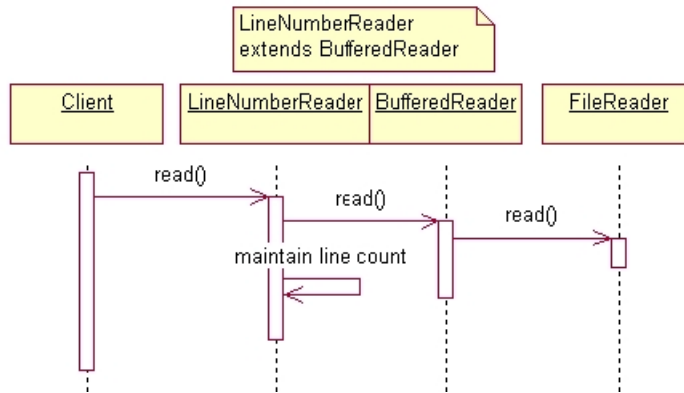
Decorator Pattern Analysis

- 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

CMSC 433, Fall 2002

62

Interaction diagram



CMSC 433, Fall 2002

63

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);
}
```

CMSC 433, Fall 2002

64

This document was created with Win2PDF available at <http://www.daneprairie.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.