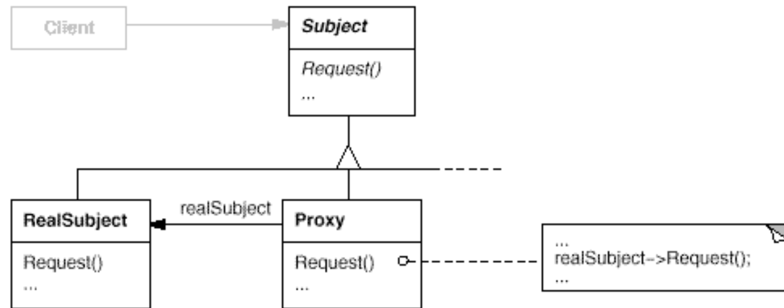
Proxy Pattern Motivation

- 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, location, etc. of protected object

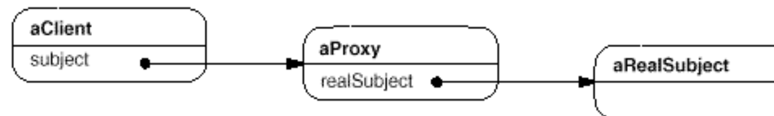
Uses of the Proxy Pattern

- *Virtual proxy*: impose a lazy creation semantics, to postpone expensive object creations until strictly necessary.
- *Monitor proxy*: impose security constraints on the original object, say by intercepting some method calls to proxied object.
- *Remote proxy*: hide the fact that an object resides on a remote location.

Proxy Pattern Class Diagram



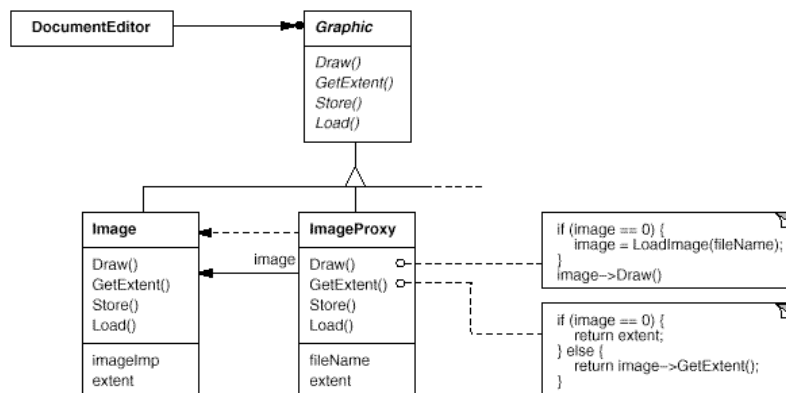
Object Diagram



Proxy vs. Adapter

- Proxies implement *the same* interface as the objects they adapt
  - But may restrict some operations
  - E.g., refuse to perform a sensitive operation
- Adapters implement *a different* interface than the objects they adapt

Example Usage Class Diagram



More OMT Notation

- Arrow ending in filled circle
  - More than one

Decorator Pattern

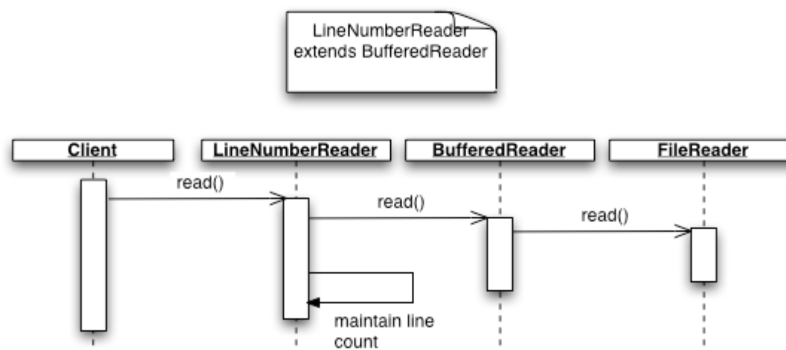
- Motivation
  - Want to add responsibilities/capabilities to individual objects, not to entire class
  - Inheritance requires a compile-time choice of parent class
- Solution
  - Enclose the component in another object that adds the responsibility/capability
    - The enclosing object is called a **decorator**.

Example: Java I/O

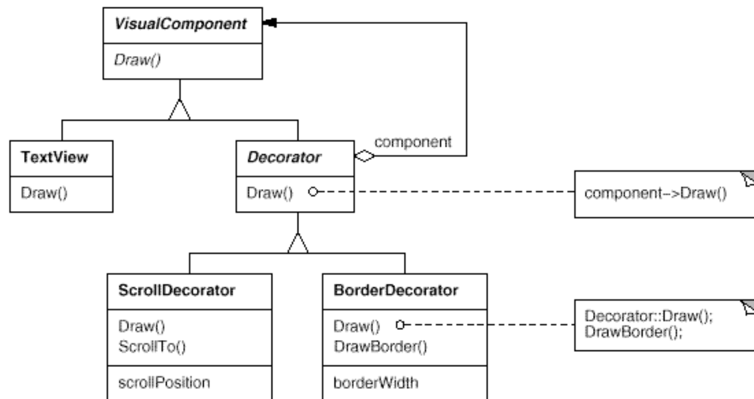
```
class LineNumberReader extends BufferedReader {
    private int lineNumber;
    public LineNumberReader(Reader in) { super(in); }
    public int getLineNumber() { return lineNumber; }

    public int read() {
      int c = super.read();
      if (c == '\n') {
        lineNumber++;
        return '\n';
      }
      return c;
    }
}
```


Interaction Diagram



Decorator Pattern: Another Example



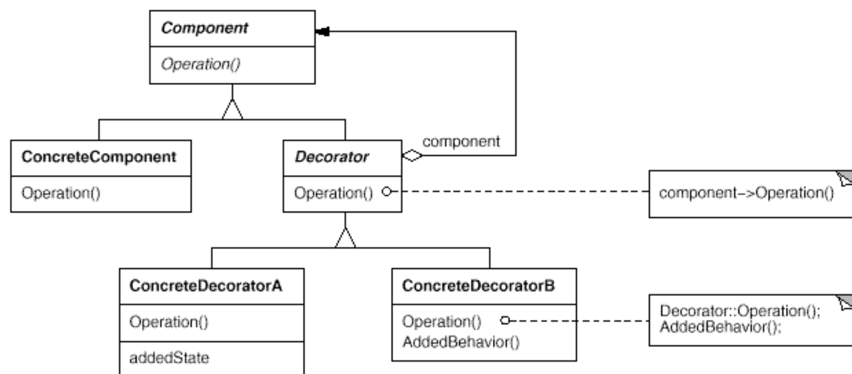
More OMT Notation

- Arrow beginning with diamond
  - "Part-of" or aggregation
  - Only accessed by object pointing to it

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

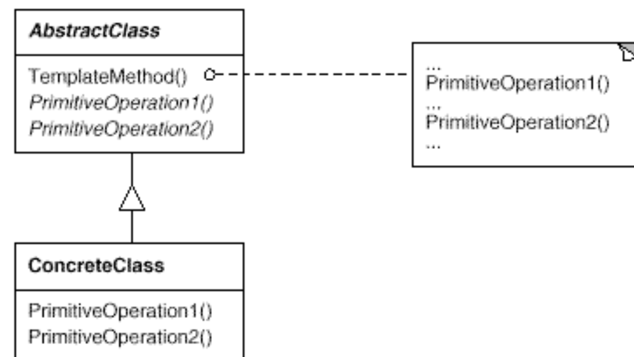
Decorator vs. Adapter

- 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
  - Put 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()**

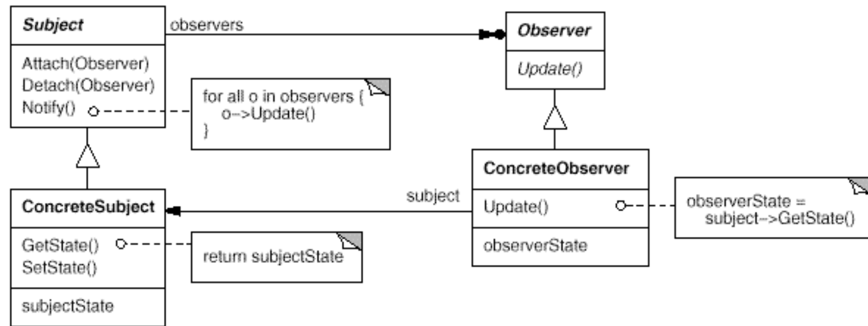
```
package junit.framework;
public class TestCase {
  ...
  public void run() {
    setUp(); runTest(); tearDown()
  }
}
```

- Users can subclass TestCase and override setUp(), runTest() and tearDown()

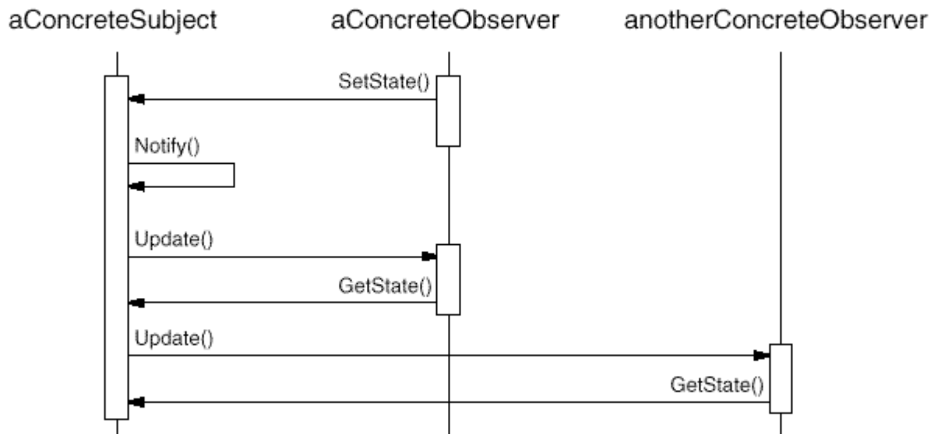
Observer Pattern

- Problem
  - Dependent must be consistent with master's state
- Solution structure: Four kinds of objects
  - Abstract subject (master)
    - Maintains list of dependents; notifies them when master changes
  - Abstract observer (dependents)
    - Defines protocol for updating dependents
  - Concrete subject
    - Manages data for dependents; notifies them when master changes
  - Concrete observers
    - Gets 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 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

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.
- Make 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.