CMSC433, Fall 2002
Design Patterns

Adam Porter

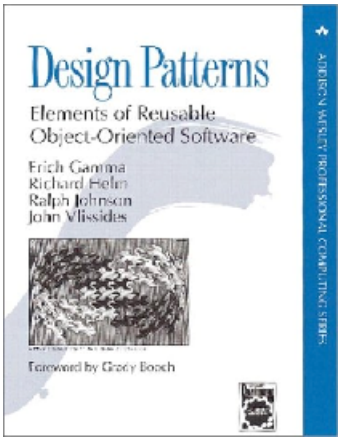
Sep 24, 2002

What is a pattern?

- Patterns = problem/solution pairs in context
- Patterns facilitate reuse of successful software architectures and design
- Not code reuse
  - Instead, solution/strategy reuse
  - Sometimes, interface reuse

Gang of Four

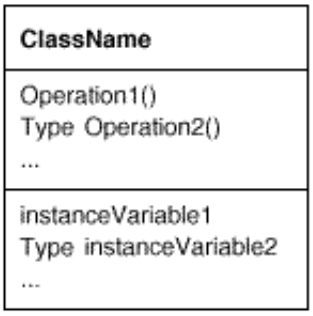
- The book that started it all
- Community refers to authors as the "Gang of Four"
- Figures and some text in these slides come from book
- On reserve in CS library (3rd floor AVW)



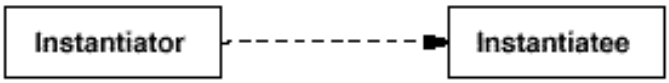
Object Modeling Technique (OMT)

- Used to describe patterns in GO4 book
- Graphical representation of OO relationships
  - **Class diagrams** show the static relationship between classes
  - **Object diagrams** represent the state of a program as series of related objects
  - **Interaction diagrams** illustrate execution of the program as an interaction among related objects

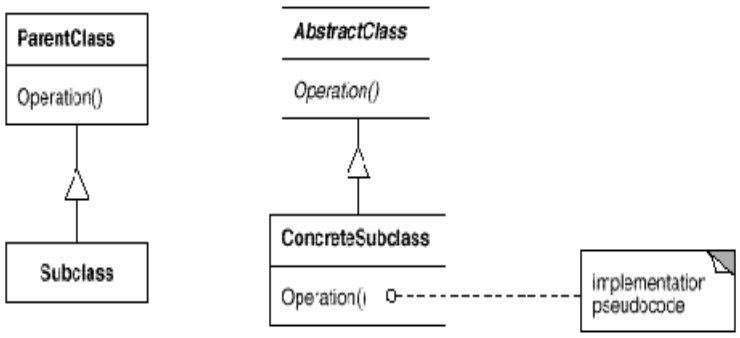
Classes



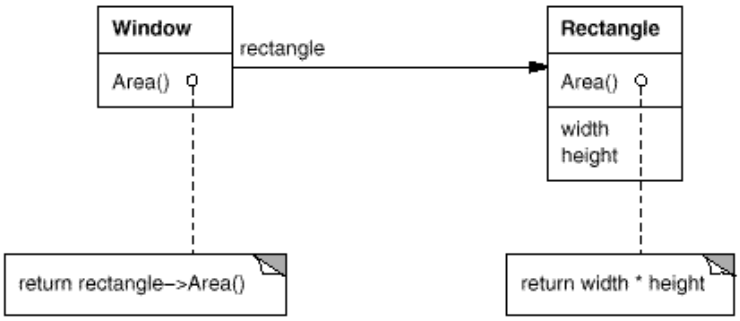
Object instantiation



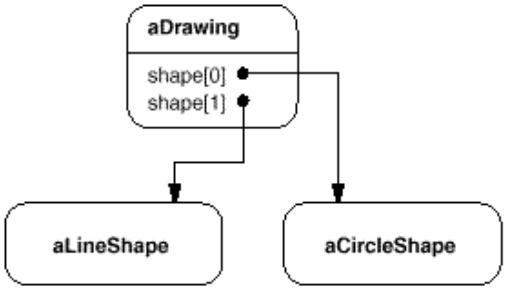
Subclassing and Abstract Classes



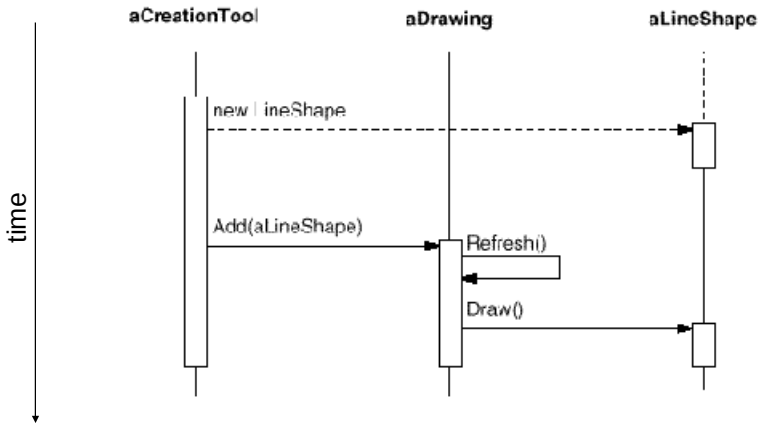
Pseudo-code and Containment



Object diagrams



Interaction diagrams



Components of a Pattern

- Pattern name
  - identify this pattern; distinguish from other patterns
  - define terminology
- Pattern alias – "*also known as*"
- Real-world example
- Context
- Problem

Components of a Pattern (cont'd)

- Solution
  - typically natural language notation
- Structure
  - class (and possibly object) diagram in solution
- Interaction diagram (optional)
- Consequences
  - advantages and disadvantages of pattern
  - ways to address residual design decisions

Components of a Pattern (cont'd)

- Implementation
  - critical portion of plausible code for pattern
- Known uses
  - often systems that inspired pattern
- References - See also
  - related patterns that may be applied in similar cases

Design patterns taxonomy

- Creational patterns
  - concern the process of object creation
- Structural patterns
  - deal with the composition of classes or objects
- Behavioral patterns
  - characterize the ways in which classes or objects interact and distribute responsibility.

Creation patterns

- Singleton
  - Ensure a class only has one instance, and provide a global point of access to it.
- Typesafe Enum
  - Generalizes Singleton: ensures a class has a fixed number of unique instances.
- Abstract Factory
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Structural patterns

- Adapter
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces
- Proxy
  - Provide a surrogate or placeholder for another object to control access to it
- Decorator
  - Attach additional responsibilities to an object dynamically

Behavioral patterns

- Template
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses
- State
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class
- Observer
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

Principles Underlying Patterns

- Rely on abstract classes to hide differences between subclasses from clients
  - object class vs. object type
    - class defines how an object is implemented
    - type defines an object's interface (protocol)
- Program to an interface, not an implementation

Principles (cont'd)

- Black-box vs. white-box reuse
  - black-box relies on object references, usually through instance variables
  - white-box reuse by inheritance
  - black-box reuse preferred for information hiding, run-time flexibility, elimination of implementation dependencies
  - disadvantages: Run-time efficiency (high number of instances, and communication by message passing)
- Favor composition over class inheritance

Principles (cont'd)

- Delegation
  - powerful technique when coupled with black-box reuse
  - Allow delegation to different instances at run-time, as long as instances respond to similar messages
  - disadvantages:
    - sometimes code harder to read and understand
    - efficiency (because of black-box reuse)

Some Design Patterns

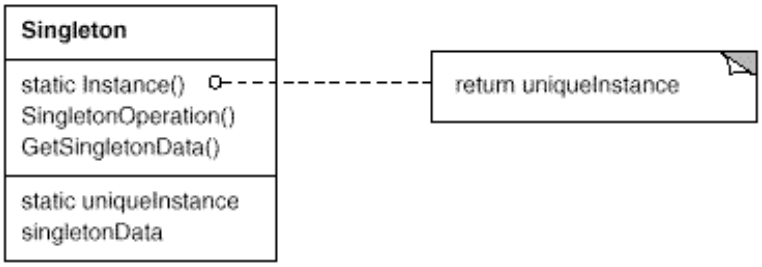
Singleton objects

- 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

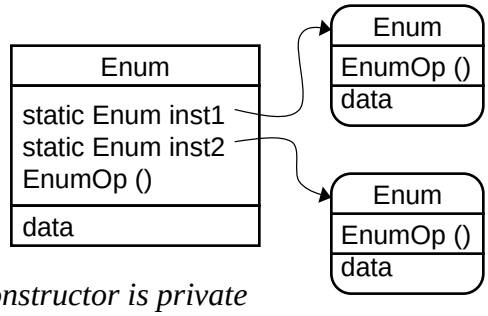
```
public class Singleton {
  private Singleton() {…}
  final private static Singleton instance
          = new Singleton();
  public Singleton getInstance() { return instance; }
}
```

- 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



Note: constructor is private

Typesafe Enum: Example

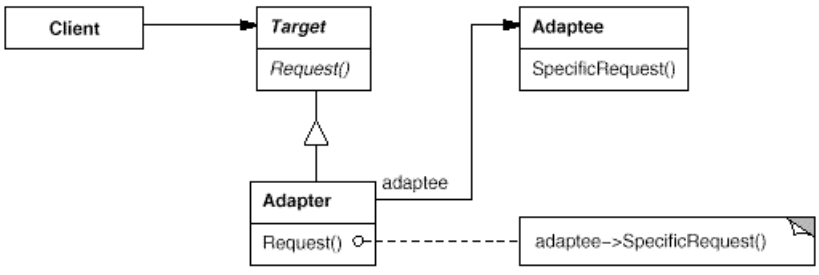
```
public class Suit {
  private final String name;

  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



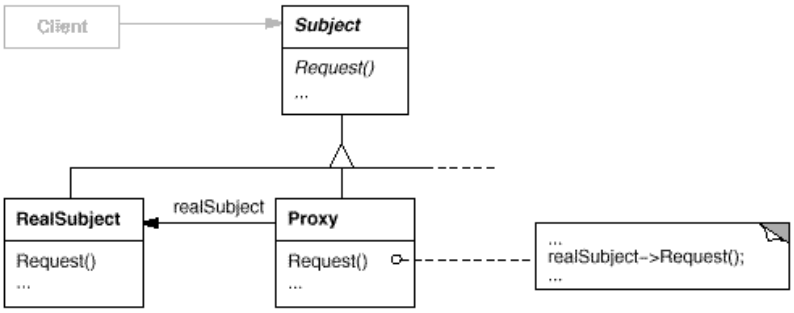
Proxy Pattern Motivation

- Goal:
  - Prevent an object from being accessed directly by its clients
- Solution:
  - Use an additional object, called a proxy
  - Clients access to protected object only through proxy
  - Proxy keeps track of status and/or location of protected object

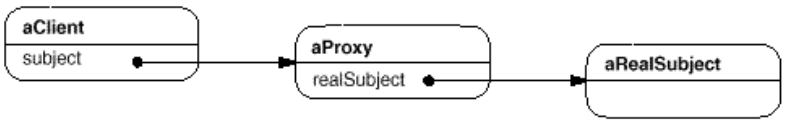
Uses of the Proxy Pattern

- *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; e.g. the **RemoteLogClient** is essentially a remote proxy for a **LocalLog**.

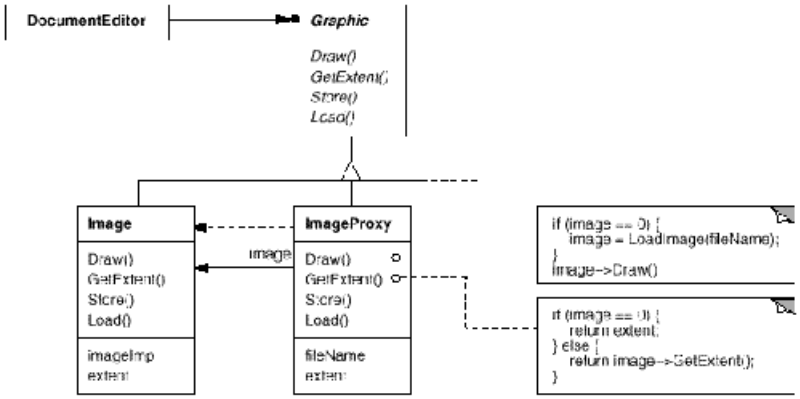
Proxy Pattern Class Diagram



Object Diagram



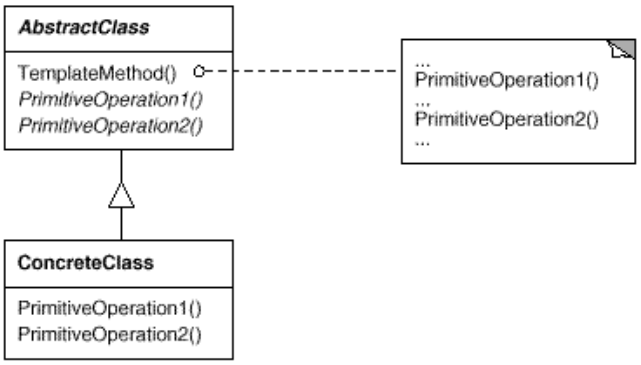
Example Usage Class Diagram



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

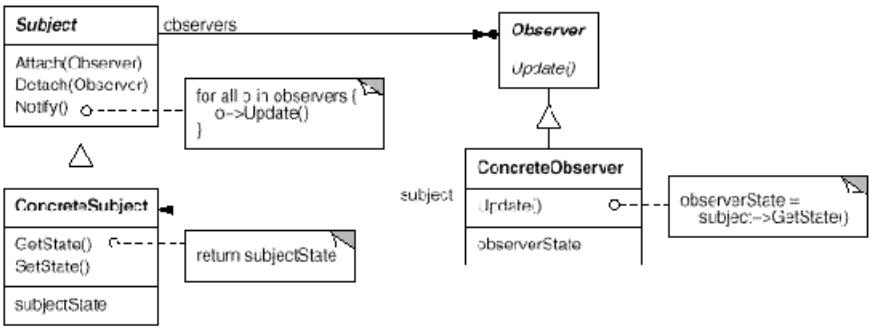
```
Junit.framework.TestCase.run() {
  setUp(); runTest(); tearDown()
}
```

- In class example, subclass (LogRecordTest) overrides runTest() and setUp()

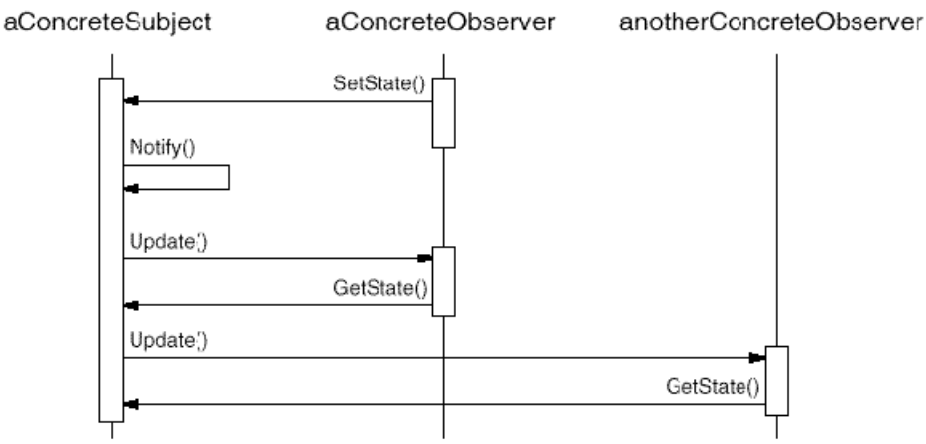
Observer pattern

- Problem
  - dependent's state must be consistent with master's state
- Solution structure
  - define four kinds of objects:
    - abstract subject
      - maintain list of dependents; 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

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.
  - 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