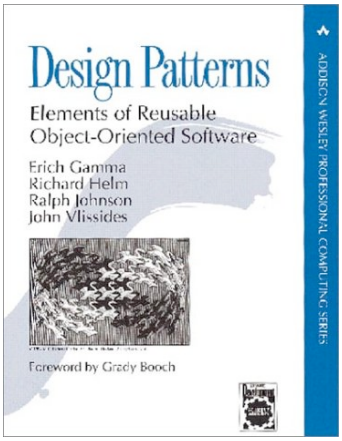
Design Patterns

- Iterators are an example of a *design pattern:*
  - Design pattern = problem + solution in context
  - Iterators: solution for providing generic traversals
- Design patterns capture software architectures and designs
  - Not direct code reuse!
  - Instead, solution/strategy reuse
  - Sometimes, interface reuse

Gang of Four

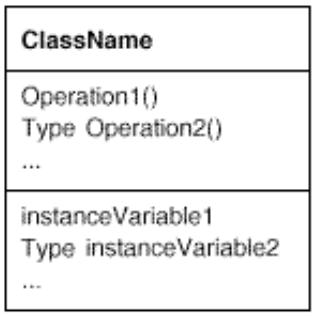
- The book that started it all
- Community refers to authors as the "Gang of Four"
- Figures and some text in these slides come from book



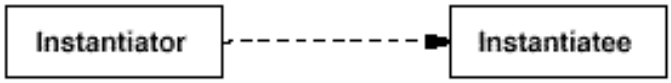
Object Modeling Technique (OMT)

- 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

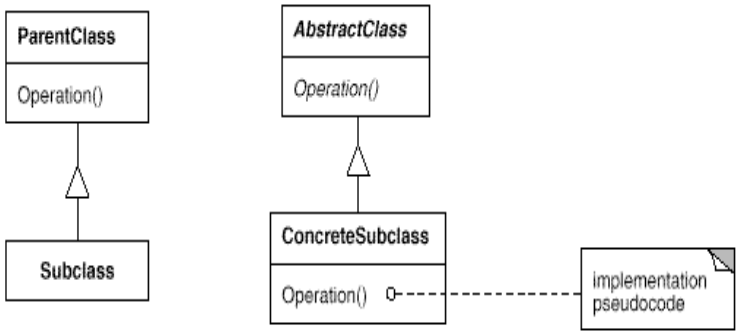
Classes



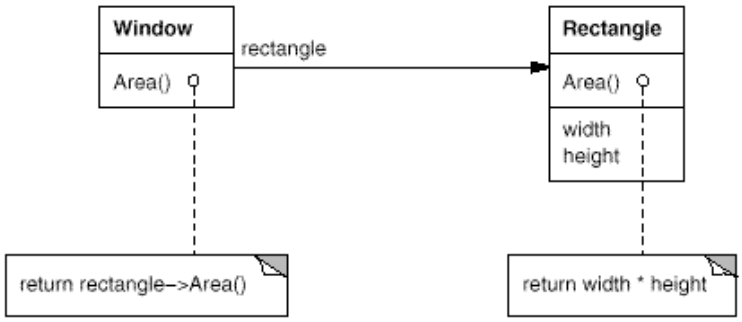
Object instantiation



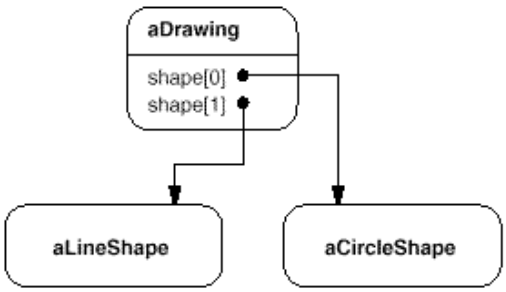
Subclassing and Abstract Classes



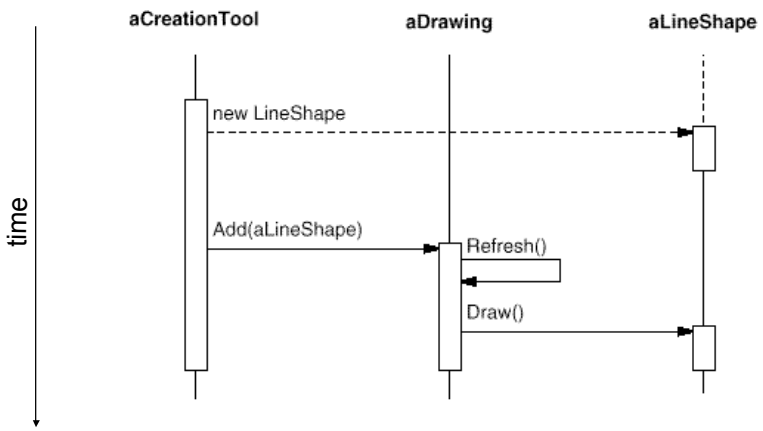
Pseudo-code and Containment



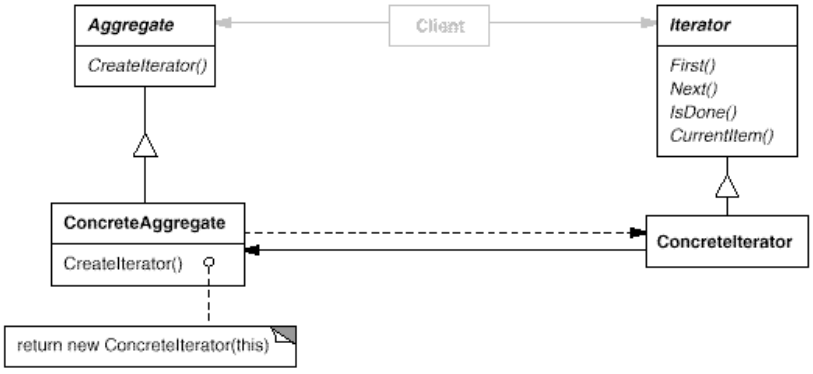
Object diagrams



Interaction diagrams



Structure of Iterator (Cursor) Pattern



Components of a Pattern

- Name(s)
- Problem
  - Context
  - Real-world example
- Solution
  - Design/structure
  - Implementation
- Consequences
- Variations, known uses

Iterator Pattern, Again

- **Name**: Iterator (*aka* Cursor)
- **Problem**:
  - How to process the elements of an aggregate in an implementation-independent manner?
- **Solution**:
  - Define an Iterator interface
    - next(), hasNext(), etc. methods
  - Aggregate returns an instance of an implementation of Iterator interface to control the iteration

Iterator Pattern

- **Consequences**:
  - Support different and simultaneous traversals
    - Multiple implementations of Iterator interface
    - One traversal per Iterator instance
  - Requires coherent policy on aggregate updates
    - Invalidate Iterator by throwing an exception, or
    - Iterator only considers elements present at the time of its creation
- **Variations**:
  - Internal vs. external iteration
    - Java Iterator is external

Internal Iterators

```
public interface InternalIterator<Element> {
  void iterate(Processor<Element> p);
}
public interface Processor<Element> {
  public void process(Element e);
}
```

- The internal iterator applies the processor instance to each element of the aggregate
  - Thus, entire traversal happens “at once”
  - Less control for client, but easier to formulate traversal

Design Patterns: Goals

- To support **reuse** of successful designs
- To facilitate **software evolution**
  - Add new features easily, without breaking existing ones
- In short, we want to **design for change**

Underlying Principles

- Reduce implementation dependencies between elements of a software system
- Sub-goals:
  - Program to an interface, not an implementation
  - Favor composition over inheritance
  - Use delegation

Program to Interface, Not Implementation

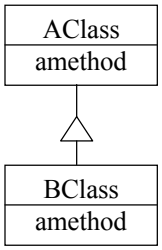
- Rely on abstract classes and interfaces to hide differences between subclasses from clients
  - Interface defines an object's use (protocol)
  - Implementation defines particular policy
- *Example*: **Iterator** interface, compared to its implementation for a **LinkedList**

Rationale

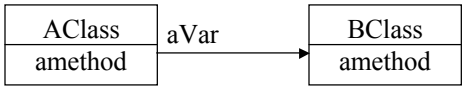
- Decouples clients from the implementations of the applications they use
- When clients manipulate an interface, they remain unaware of the specific object types being used.
- Therefore: clients are less dependent on an implementation, so it can be easily changed later.

Favor Composition over Class Inheritance

- White box reuse:
  - Inheritance



- Black box reuse:
  - Composition

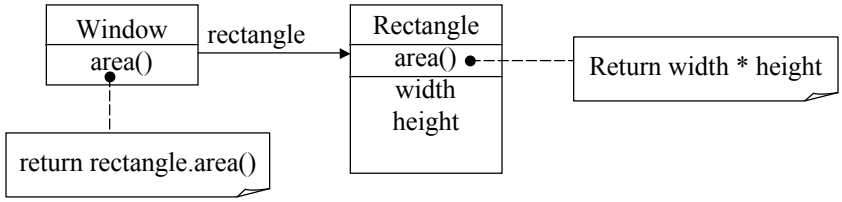


Rationale

- White-box reuse results in implementation dependencies on the parent class
  - Reusing a subclass may require rewriting the parent
  - But inheritance easy to specify
- Black-box reuse often preferred
  - Eliminates implementation dependencies, hides information, object relationships non-static for better run-time flexibility
  - But adds run-time overhead (additional instance allocation, communication by dynamic dispatch)

Delegation

- Forward messages (delegate) to different instances at run-time; a form of composition
  - May pass invoking object's **this** pointer to simulate inheritance



Rationale

- Object relationships dynamic
  - Composes or re-composes behavior at run-time
- But:
  - Sometimes code harder to read and understand
  - Efficiency (because of black-box reuse)

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

Catalogue of Patterns: 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

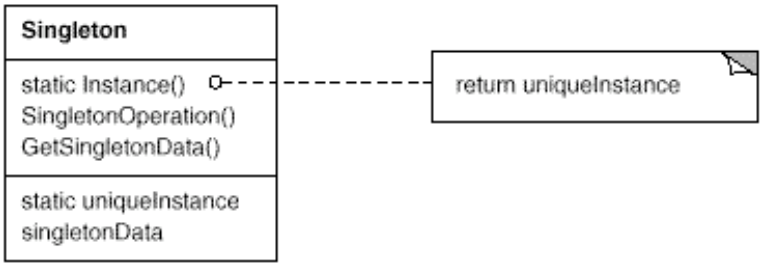
Singleton Objects

- Problem:
  - Some classes have conceptually one instance
    - Many printers, but only one print spooler
    - One file system
    - One window manager
  - Creating many objects that represent the same conceptual instance adds complexity and overhead
- Solution: 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 static Singleton getInstance()
       { return instance; }
}
```

- Java semantics guarantee object is created immediately before first use

Marshalling

- *Marshalling* is the process of transforming internal data into a form that can be
  - Written to disk
  - Sent over the network
  - Etc.
- *Unmarshalling* is the inverse process

Marhsalling in Java

- Java provides support for marshalling objects
  - Classes implement the *Serializable* interface
  - The JVM imlpements standard marhsalling and unmarshalling automatically
    - E.g., enables you to create persistent objects, stored on disk
    - This can be useful for building a light-weight database
    - Also useful for distributed object systems
- Often, generic implementation works fine
  - But let's consider singletons...

Marhsalling and Singletons

- What happens when we unmarshall a singleton?

- Problem: JVM doesn't know about singletons
  - It will create two instances of Singleton.instance!
  - Oops!

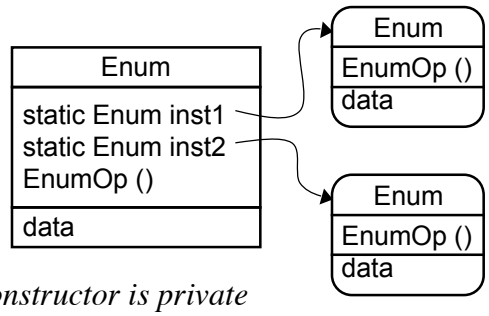
Marhsalling and Singletons (cont'd)

- Solution: Implement
  - Object readResolve() throws ObjectStreamException;
    - This method will be called after standard unmarshilling
    - Returned result is substituted for standard unmarshalled result
- E.g., add to Singleton class the following method
  - Object readResolve() { return instance; }
- Notes: Serialization is wacky!
  - For example, objects can only be nested 1001 deep????

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");
}
```


Enumerators in Java 1.5

- New version of Java has type safe enums
  - Built-in: Don't need to use the design pattern
- `public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }`
  - Type checked at compile time
  - Implemented as objects (translated as prev slide?)
  - Two extra class methods:
    - public static <this enum class>[] values() -- the enumeration elts
    - public static <this enum class> valueOf(String name) -- get an elt

Adapter (*aka* Wrapper) Pattern

- Problem:
  - 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 (cont'd)

- Here's what we have:

  - Client is already written, and it uses the Target interface
  - Adaptee has a method that works, but has the wrong name/interface
- How do we enable the Client to use the Adaptee?

Adapter Pattern (cont'd)

- Solution: adapter class to implement client's expected interface, forwarding methods