Components of a Pattern

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

- **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

**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**

```java
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
  - Existing code
- 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 has 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

```
Window
area()

rectangle

Rectangle
area()
width
height

Return width * height

return rectangle.area()
```
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
Implementing the Singleton method

• In Java, just define a final static field
  ```java
  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

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

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 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

• Solution: adapter class to implement client’s expected interface, forwarding methods