23 Patterns in 80 Minutes: a Whirlwind Java-centric Tour of the Gang-of-Four Design Patterns

Josh Bloch  Charlie Garrod
Administrivia

- Homework 6 checkpoint due Friday 5 pm
- Final exam Tuesday, May 3, 5:30-8:30 pm, PH 100
- Final review session Sunday, May, 7-9 pm, DH 1112
Key concept from Tuesday...
MapReduce with key/value pairs (Google style)

- **Master**
  - Assign tasks to workers
  - Ping workers to test for failures

- **Map workers**
  - Map for each key/value pair
  - Emit intermediate key/value pairs

The shuffle:
Key concept from Tuesday...
MapReduce with key/value pairs (Google style)

- E.g., for each word on the Web, count the number of times that word occurs
  - For Map: key1 is a document name, value is the contents of that document
  - For Reduce: key2 is a word, values is a list of the number of counts of that word

\[
f_1(\text{String } \text{key1}, \text{String } \text{value}): \\
\text{for each word } w \text{ in value:} \\
\quad \text{EmitIntermediate}(w, 1);
\]

\[
f_2(\text{String } \text{key2}, \text{Iterator } \text{values}): \\
\quad \text{int } \text{result} = 0; \\
\quad \text{for each } v \text{ in values:} \\
\quad \quad \text{result} += v; \\
\quad \text{Emit}((\text{key2}, \text{result}));
\]

MapReduce: (docName, docText) \rightarrow (word, wordCount)
Outline

I. Creational Patterns
II. Structural Patterns
III. Behavioral Patterns
Pattern Name

• Intent – the aim of this pattern
• Use case – a motivating example
• Key types – the interfaces that define pattern
• JDK – example(s) of this pattern in the JDK
Illustration

• Code sample, diagram, or drawing
  – Time constraints make it impossible to include illustrations from some patterns
• Some patterns lack an illustration 😞
I. Creational Patterns

1. Abstract factory
2. Builder
3. Factory method
4. Prototype
5. Singleton
Abstract Factory

• Intent – Allow creation of families of related objects independent of implementation
• Use case – look-and-feel in a GUI toolkit
• Key type – *Factory* with methods to create each family member
• JDK – Not common
Builder

• Intent – Separate construction of complex object from representation so same creation process can create different representations

• Use case – converting rich text to various formats

• Key types – (Abstract) Builder
  – GoF has extra layer of indirection (“Director”)

• JDK – StringBuilder, StringBuffer*
  – But both produce String
  – And most builders in the JDK are concrete
My take on Builder

• Emulates named parameters in languages that don’t support them
• Reduces exponential $O(2^n)$ creational methods to $O(n)$ by allowing them to be combined freely, at the cost of an intermediate (Builder) object
Builder Illustration


public class NutritionFacts {
    public static class Builder {
        public Builder(String name, int servingSize, int servingsPerContainer) { ... }
        public Builder totalFat(int val) { totalFat = val; }
        public Builder saturatedFat(int val) { satFat = val; }
        public Builder transFat(int val) { transFat = val; }
        public Builder cholesterol(int val) { cholesterol = val; }
        ... // 15 more setters
        
        public NutritionFacts build() {
            return new NutritionFacts(this);
        }
    }
    private NutritionFacts(Builder builder) { ... }
}
Factory Method

• Intent – abstract creational method that lets subclasses decide which class to instantiate
• Use case – creating documents in a framework
• Key types – Creator, which contains abstract method to create an instance
• JDK – not common. Iterable.iterator()
• Related Static Factory pattern is very common
  – Technically not a GoF pattern, but close enough
Factory Method Illustration

public interface Iterable<E> {
    public abstract Iterator<E> iterator();
}

class ArrayList<E> implements List<E> {
    public Iterator<E> iterator() { ... } ... } ...

class HashSet<E> implements Set<E> {
    public Iterator<E> iterator() { ... } ... }

Prototype

• Intent – Create an object by cloning another and tweaking as necessary
• Use case – writing a music score editor in a graphical editor framework
• Key types – *Prototype* (AKA Cloneable)
• JDK – *clone*, but don’t use it (except on arrays)
  – Java and Prototype pattern are a poor fit
Singleton

• Intent – ensuring a class has only one instance
• Use case – GoF say print queue, file system, company in an accounting system
  – Compelling uses are rare but they do exist
• Key types – Singleton
• JDK – java.lang.Runtime
Singleton Illustration

public enum Elvis {
    ELVIS;

    public sing(Song song) { ... }
    public playGuitar(Riff riff) { ... }
    public eat(Food food) { ... }
    public take(Drug drug) { ... }
}
My take on singleton

• It’s an *instance-controlled class*; others include
  – Static utility class (non-instantiable)
  – Enum – one instance per value, all values known at compile time
  – Interned class – one canonical instance per value, new values created at runtime

• There is a duality between singleton and static utility class
II. Structural Patterns

1. Adapter
2. Bridge
3. Composite
4. Decorator
5. Façade
6. Flyweight
7. Proxy
Adapter

• Intent – convert interface of a class into one that another class requires, allowing interoperability
• Use case – numerous, e.g., arrays vs. collections
• Key types – Target, Adaptee, Adapter
• JDK – Arrays.asList(T[])
Adapter Illustration

Have this and this? Use this!
Bridge

• Intent – Decouple an abstraction from its implementation so they can vary independently
• Use case – portable windowing toolkit
• Key types – Abstraction, Implementor
• JDK – JDBC, Java Cryptography Extension (JCE)
  – Both are Service Provider Interface (SPI) frameworks
  – SPI is Bridge Implementor!
Bridge Illustration

Diagram showing the structure of a Java Application with layers for JNDI API, Naming Manager, and JNDI SPI. Various protocols such as LDAP, DNS, NIS, NDS, RMI, and CORBA are connected to these layers, indicating implementation possibilities.
Composite

• Intent – Compose objects into tree structures. Let clients treat primitives & compositions uniformly.
• Use case – GUI toolkit (widgets and containers)
• Key type – Component that represents both primitives and their containers
• JDK – javax.swing.JComponent
Composite Illustration

public interface Expression {
    double eval();  // Returns value
    String toString(); // Returns infix expression string
}

public class UnaryOperationExpression implements Expression {
    public UnaryOperationExpression(UnaryOperator operator, Expression operand);
}

public class BinaryOperationExpression implements Expression {
    public BinaryOperationExpression(BinaryOperator operator, Expression operand1, Expression operand2);
}

public class NumberExpression implements Expression {
    public NumberExpression(double number);
}
Decorator

• Intent – attach features to an object dynamically
• Use case – attaching borders in a GUI toolkit
• Key types – *Component*, implement by decorator and decorated
• JDK – Collections (e.g., Synchronized wrappers), java.io streams, Swing components
Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.

For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with
Façade

- Intent – Provide a simple unified interface to a set of interfaces in a subsystem
  - GoF allow for variants where the complex underpinnings are exposed and hidden
- Use case – any complex system; GoF use compiler
- Key types – Façade (the simple unified interface)
- JDK – java.util.concurrent.Executors
Facade Illustration

Subsystem classes
Flyweight

• Intent – use sharing to support large numbers of fine-grained objects efficiently
• Use case – characters in a document
• Key types – the Flyweight (instance-controlled!)
  – State can be made *extrinsic* to keep Flyweight sharable
• JDK – Pervasive! All enums, many others. *j.u.c.TimeUnit* has # units as extrinsinc state.
Flyweight Illustration
Proxy

• Intent – surrogate for another object
• Use case – delay loading of images till needed
• Key types – *Subject*, Proxy, RealSubject
• Gof mention several flavors
  – virtual proxy – stand-in that instantiates lazily
  – remote proxy – local representative for remote obj
  – protection proxy – denies some ops to some users
  – smart reference – does locking or ref. counting, e.g.
• JDK – RMI, collections wrappers
Proxy Illustrations

**Virtual Proxy**

- `aTextDocument` image
- `anImageProxy` fileName
- `anImage` data

In memory

On disk

**Smart Reference**

- `SynchronizedList`
- `ArrayList`

**Remote Proxy**

- Client
- Proxy
- Server
III. Behavioral Patterns

1. Chain of Responsibility
2. Command
3. Interpreter
4. Iterator
5. Mediator
6. Memento
7. Observer
8. State
9. Strategy
10. Template method
11. Visitor
Chain of Responsibility

• Intent – avoid coupling sender to receiver by passing request along until someone handles it

• Use case – context-sensitive help facility

• Key types – RequestHandler

• JDK – Classloader, Properties

• Exception handling could be considered a form of Chain of Responsibility pattern
Command

• Intent – encapsulate request as object, letting you parameterize clients with different actions, queue or log requests, etc.

• Use case – menu tree

• Key types – Command (an execute method)

• JDK – Runnable, executor framework

• Is it Command pattern if you run it more than once? If it takes an argument? Returns a val?
Interpreter

- Intent – Given a language, define class hierarchy for parse tree, recursive method to interpret it
- Use case – regular expression matching
- Key types – Expression, NonterminalExpression, TerminalExpression
- JDK – no uses I’m aware of
  - Our expression evaluator (HW2) is a classic example
- Necessarily uses Composite pattern!
public interface Expression {
    double eval();     // Returns value
    String toString(); // Returns infix expression string
}

public class UnaryOperationExpression implements Expression {
    public UnaryOperationExpression(UnaryOperator operator, Expression operand);
}

public class BinaryOperationExpression implements Expression {
    public BinaryOperationExpression(BinaryOperator operator,
                                       Expression operand1, Expression operand2);
}

public class NumberExpression implements Expression {
    public NumberExpression(double number);
}
Iterator

- Intent – provide a way to access elements of a collection without exposing representation
- Use case – collections
- Key types – `Iterable, Iterator`
  - But GoF recognize internal iteration too
- JDK – Collections, for-each statement, etc.
Mediator

• Intent – Define an object that encapsulate how a set of objects interact to reduce coupling.
  – $O(n)$ couplings instead of $O(n!) = O(2^n)$

• Use case – dialog box where change in one component affects behavior of others

• Key types – Mediator, components

• JDK – Unclear
Mediator Illustration
Memento

- Intent – Without violating encapsulation, allow client to capture an object’s state, and restore
- Use case – undo stack for operations that aren’t easily undone, e.g., line-art editor
- Key type – Memento (opaque state object)
- JDK – none that I’m aware of (not serialization)
Observer

• Intent – Let objects observe the behavior of other objects so they can stay in sync
• Use case – multiple views of a data object in a GUI
• Key types – *Subject* ("observable"), *Observer*
  – GoF are agnostic on many details!
• JDK – Swing, left and right
State

• Intent – use an object internally to represent the state of another object; delegate method invocations to the state object
• Use case – TCP Connection (which is stateful)
• Key type – State
• JDK – none that I’m aware of but
  – Works great in Java
  – Use enums as states
  – Use AtomicReference&lt;State&gt; to store it
Strategy

• Intent — represent a behavior that parameterizes an algorithm for behavior or performance
• Use case — line-breaking for text compositing
• Key types — Strategy
• JDK – Comparator
Template method

• Intent – define skeleton of an algorithm or data structure, deferring some decisions to subclasses
• Use case – application framework that lets plugins implement all operations on documents
• Key types – AbstractClass, ConcreteClass
• JDK – Skeletal collection impls (e.g., AbstractList)
• Note – template method is dual to strategy, you can mechanically convert one to the other
Template Method Illustration

// List adapter for primitive int arrays
public static List<Integer> intArrayList(final int[] a) {
    return new AbstractList<Integer>() {
        public Integer get(int i) {
            return a[i];
        }

        public Integer set(int i, Integer val) {
            Integer oldVal = a[i];
            a[i] = val;
            return oldVal;
        }

        public int size() {
            return a.length;
        }
    };
}
Visitor

- Intent – Represent an operation to be performed on elements of an object structure (e.g., a parse tree). Visitor lets you define a new operation without modifying the type hierarchy.
- Use case – type-checking, pretty-printing, etc.
- Key types – Visitor, ConcreteVisitor, all the types that get visited
- JDK – None that I’m aware of
More on Visitor

• Visitor is NOT merely traversing a graph structure and applying a method
  – That’s Iterator

• The essence of visitor is *double-dispatch*
  – First dynamically dispatch on the Visitor
  – Then on the object being visited
Summary

• Now you know all the Gang of Four patterns
• Definitions can be vague
• Coverage is incomplete
• But they’re extremely valuable
  – They gave us a vocabulary
  – And a way of thinking about software
• Look for patterns as you read and write software
  – GoF, non-GoF, and undiscovered