

Design Patterns



Nelson Padua-Perez
Chau-Wen Tseng

Department of Computer Science
University of Maryland, College Park

Design Patterns

- Descriptions of **reusable** solutions to common software design problems
- Captures the experience of experts
 - Rationale for design
 - Tradeoffs
 - Codifies design in reusable form
- Example
 - Iterator pattern

Goals

- Solve common programming challenges
- Improve reliability of solution
- Aid rapid software development
- Useful for real-world applications

Observations

- Design patterns are like recipes – generic solutions to expected situations
- Design patterns are language independent
- Recognizing when and where to use design patterns requires familiarity & experience
- Design pattern libraries serve as a glossary of idioms for understanding common, but complex solutions

Observations (cont.)

- **Many design patterns may need to fit together**
 - **Design Patterns (by Gamma et al., a.k.a. Gang of Four, or GOF) list 23 design patterns**
 - **Around 250 common OO design patterns**
- **Design patterns are used throughout the Java Class Libraries**

Documentation Format

1. **Motivation or context for pattern**
2. **Prerequisites for using a pattern**
3. **Description of program structure**
4. **List of participants (classes & objects)**
5. **Collaborations (interactions) between participants**
6. **Consequences of using pattern (good & bad)**
7. **Implementation techniques & issues**
8. **Example codes**
9. **Known uses**
10. **Related patterns**

Types of Design Patterns

- **Creational**
 - Deal with the best way to create objects
- **Structural**
 - Ways to bring together groups of objects
- **Behavioral**
 - Ways for objects to communicate & interact

Creational Patterns

1. **Abstract Factory**- Creates an instance of several families of classes
2. **Builder** - Separates object construction from its representation
3. **Factory Method** - Creates an instance of several derived classes
4. **Prototype** - A fully initialized instance to be copied or cloned
5. **Singleton** - A class of which only a single instance can exist

Structural Patterns

6. **Adapter** - Match interfaces of different classes
7. **Bridge** - Separates an object's interface from its implementation
8. **Composite** - A tree structure of simple and composite objects
9. **Decorator** - Add responsibilities to objects dynamically
10. **Facade** - Single class that represents an entire subsystem
11. **Flyweight** - Fine-grained instance used for efficient sharing
12. **Proxy** - Object representing another object

Behavioral Patterns

13. **Chain of Responsibility** - A way of passing a request between a chain of objects
14. **Command** - Encapsulate a command request as an object
15. **Interpreter** - A way to include language elements in a program
16. **Iterator** - Sequentially access the elements of a collection
17. **Mediator** - Defines simplified communication between classes
18. **Memento** - Capture and restore an object's internal state

Behavioral Patterns (cont.)

19. **Observer** - A way of notifying change to a number of classes
20. **State** - Alter an object's behavior when its state changes
21. **Strategy** - Encapsulates an algorithm inside a class
22. **Template Method** - Defer the exact steps of an algorithm to a subclass
23. **Visitor** - Defines a new operation to a class without changing class

Iterator Pattern

- **Definition**
 - Move through list of objects without knowing its internal representation
- **Where to use & benefits**
 - Use a standard interface to represent data objects
 - Uses standard iterator built in each standard collection, like List, Set, or Map
 - Need to distinguish variations in the traversal of an aggregate

Iterator Pattern

■ Example

- Iterator for collection
- Original
 - Examine elements of collection directly
- Using pattern
 - Collection provides Iterator class for examining elements in collection

Iterator Example

```
public interface Iterator {
    Bool hasNext();
    Object next();
}

Iterator it = myCollection.iterator();

while ( it.hasNext() ) {
    MyObj x = (MyObj) it.next();    // finds all objects
    ...                            // in collection
}
```

Singleton Pattern

■ Definition

- One instance of a class or value accessible globally

■ Where to use & benefits

- Ensure unique instance by defining class final
- Access to the instance only via methods provided

Singleton Example

```
public class Employee {
    public static final int ID = 1234; // ID is a singleton
}
public final class MySingleton {
    // declare the unique instance of the class
    private static MySingleton uniq = new MySingleton();
    // private constructor only accessed from this class
    private MySingleton() { ... }
    // return reference to unique instance of class
    public static MySingleton getInstance() {
        return uniq;
    }
}
```

Adapter Pattern

- **Definition**
 - Convert existing interfaces to new interface
- **Where to use & benefits**
 - Help match an interface
 - Make unrelated classes work together
 - Increase transparency of classes

Adapter Pattern

- **Example**
 - Adapter from integer Set to integer Priority Queue
 - Original
 - Integer set does not support Priority Queue
 - Using pattern
 - Adapter provides interface for using Set as Priority Queue
 - Add needed functionality in Adapter methods

Adapter Example

```
public interface PriorityQueue { // Priority Queue
    void add(Object o);
    int size();
    Object removeSmallest();
}
```

Adapter Example

```
public class PriorityQueueAdapter implements PriorityQueue {
    Set s;
    PriorityQueueAdapter(Set s) { this.s = s; }
    public void add(Object o) { s.add(o); }
    int size() { return s.size(); }
    public Integer removeSmallest() {
        Integer smallest = Integer.MAX_VALUE;
        Iterator it = s.iterator();
        while ( it.hasNext() ) {
            Integer i = it.next();
            if (i.compareTo(smallest) < 0)
                smallest = i;
        }
        s.remove(smallest);
        return smallest;
    }
}
```

Factory Pattern

■ Definition

- Provides an abstraction for deciding which class should be instantiated based on parameters given

■ Where to use & benefits

- A class cannot anticipate which subclasses must be created
- Separate a family of objects using shared interface
- Hide concrete classes from the client

Factory Pattern

■ Example

- Car Factory produces different Car objects
- Original
 - Different classes implement Car interface
 - Directly instantiate car objects
 - Need to modify client to change cars
- Using pattern
 - Use carFactory class to produce car objects
 - Can change cars by changing carFactory

Factory Example

```
class 350Z implements Car;           // fast car
class Ram implements Car;            // truck
class Accord implements Car;         // family car
Car fast = new 350Z();                // returns fast car

public class carFactory {
    public static Car create(String type) {
        if (type.equals("fast"))      return new 350Z();
        if (type.equals("truck"))     return new Ram();
        else if (type.equals("family")) return new Accord();
    }
}

Car fast = carFactory.create("fast"); // returns fast car
```

Decorator Pattern

- **Definition**
 - Attach additional responsibilities or functions to an object dynamically or statically
- **Where to use & benefits**
 - Provide flexible alternative to subclassing
 - Add new function to an object without affecting other objects
 - Make responsibilities easily added and removed dynamically & transparently to the object

Decorator Pattern

■ Example

- Pizza Decorator adds toppings to Pizza
- Original
 - Pizza subclasses
 - Combinatorial explosion in # of subclasses
- Using pattern
 - Pizza decorator classes add toppings to Pizza objects dynamically
 - Can create different combinations of toppings without modifying Pizza class

Decorator Example

```
public interface Pizza {
    int cost();
}
public class SmallPizza extends Pizza {
    int cost()    { return 8; }
}
public class LargePizza extends Pizza {
    int cost()    { return 12; }
}
public class PizzaDecorator implements Pizza {
    Pizza p;
    PizzaDecorator (Pizza p) { this.p = p; }
    int cost()    { return p.cost(); }
}
```

Decorator Example

```
public class withOlive extends PizzaDecorator {
    int cost()    { return p.cost() + 2; }
}
public class withHam extends PizzaDecorator {
    int cost()    { return p.cost() + 3; }
}

Pizza HamOlivePizza = new withHam (
    new withOlive ( new LargePizza() ) );
... = HamOlivePizza.cost();           // returns 12+2+3

Pizza DoubleHamPizza = new withHam (
    new withHam ( new SmallPizza() ) );
... = DoubleHamPizza.cost();         // returns 8+3+3
```

Decorator Pattern

- Examples from Java I/O
 - Interface
 - InputStream
 - Concrete subclasses
 - FileInputStream, ByteArrayInputStream
 - Decorators
 - BufferedInputStream, DataInputStream
 - Code
 - `InputStream s = new DataInputStream(new BufferedInputStream (new FileInputStream()));`