6.170 Recitation #6: Design Patterns

Topics
- What is a design pattern?
- Strategy
- Composite
- Java Puzzler

What is a design pattern?

A design pattern is not part of the Java language. It is not even part of the Java library, although there are some parts of the Java library, such as **`java.util.Observer`**, that try to make design patterns easier to implement in Java.

A design pattern is just a name that software engineers assigned to a particular solution that can be reused for many problems. It is just a specification that describes a particular design. For example, now that we have taught you the complex interactions of the Observer pattern, if you see that one word mentioned in a comment, you will instantly grasp the intention of the developer.

You can implement a design pattern without knowing it. Consider this analogy in Java:

```
// Graph ADT
public class Graph<N, E> {
    /** @return an iterator over the nodes in the graph */
    public Iterator<N> iterator() {
        return Collections.unmodifiableCollection(nodes).iterator();
    }
    // Remainder omitted
}
public class GraphClient {
    Graph<Integer, String> g = new Graph<Integer, String>();
    // For each Integer i in g...
    for (Integer i : g)  // COMPILE ERROR
        System.out.println(i);
}
```

QUESTION: What is the error and how can we fix it?

The compile error states, "Can only iterate over an array or instance of **`java.lang.Iterable`**."

Even though Graph implicitly implements the **`Iterable`** interface, Java will not recognize the pattern until you say **`public class Graph<N, E> implements Iterable<N>`**.

Similarly, your class may be designed implicitly as an observer. However, another developer may not instantly understand that your class is an observer unless you call it what it is in your specification, but it is still an observer whether you say so or not.

Below is the Java 1.4 source code for HashIterator, a private inner class of HashMap. (From http://www.docjar.com/html/api/java/util/HashMap.java.html)

```
private final class HashIterator implements Iterator
{
  /**
   * The type of this Iterator: {@link #KEYS}, {@link #VALUES},
   * or {@link #ENTRIES}.
   */
  private final int type;
  /**
   * The number of modifications to the backing HashMap that we know about.
   */
  private int knownMod = modCount;
  /** The number of elements remaining to be returned by next(). */
  private int count = size;
  /** Current index in the physical hash table. */
  private int idx = buckets.length;
  /** The last Entry returned by a next() call. */
  private HashEntry last;
  /**
   * The next entry that should be returned by next(). It is set to something
   * if we're iterating through a bucket that contains multiple linked
   * entries. It is null if next() needs to find a new bucket.
   */
  private HashEntry next;

  /**
   * Construct a new HashIterator with the supplied type.
   * @param type {@link #KEYS}, {@link #VALUES}, or {@link #ENTRIES}
   */
  HashIterator(int type)
  {
    this.type = type;
  }

  /**
   * Returns true if the Iterator has more elements.
   * @return true if there are more elements
   * @throws ConcurrentModificationException if the HashMap was modified
   */
  public boolean hasNext()
  {
    if (knownMod != modCount)
```

```
        throw new ConcurrentModificationException();
      return count > 0;
    }

    /**
     * Returns the next element in the Iterator's sequential view.
     * @return the next element
     * @throws ConcurrentModificationException if the HashMap was modified
     * @throws NoSuchElementException if there is none
     */
    public Object next()
    {
      if (knownMod != modCount)
        throw new ConcurrentModificationException();
      if (count == 0)
        throw new NoSuchElementException();
      count--;
      HashEntry e = next;

      while (e == null)
        e = buckets[--idx];

      next = e.next;
      last = e;
      if (type == VALUES)
        return e.value;
      if (type == KEYS)
        return e.key;
      return e;
    }

    /**
     * Removes from the backing HashMap the last element which was fetched
     * with the next() method.
     * @throws ConcurrentModificationException if the HashMap was modified
     * @throws IllegalStateException if called when there is no last element
     */
    public void remove()
    {
      if (knownMod != modCount)
        throw new ConcurrentModificationException();
      if (last == null)
        throw new IllegalStateException();

      HashMap.this.remove(last.key);
      last = null;
      knownMod++;
    }
  }
```

EXERCISE: Draw the MDD for the Iterator pattern in Java for HashMap.

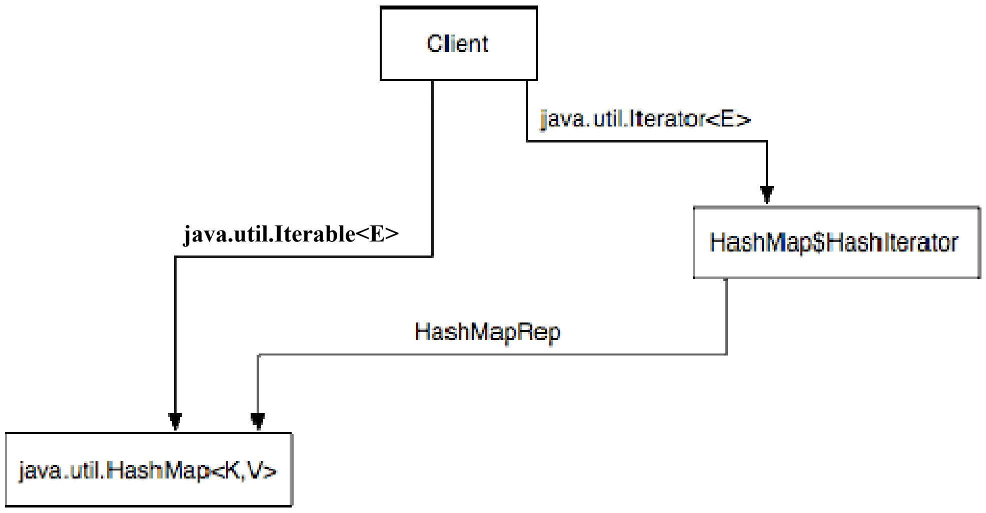

Figure 1: MDD for HashMap Iterator Pattern

These notes draw heavily from *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, & Vlissides (the "Gang of Four" book).

Design Pattern 1: Strategy

In Problem Set 4, some of you implemented Breadth First Search (BFS) as part of your graph:

```
public class Graph {
    /** @return the shortest path from start to end
      * or throws PathNotFoundException if no path exists.
      */
    public Node[] search(Node start, Node end) {
        // Implementation of BFS
    }
    // Remainder omitted
}
```

But what did you do in Problem Set 5 when you needed to use Dijkstra's Algorithm instead? Some of you chose to use subclasses to override the search method, while others added another method such as `public Node[] searchWithDijkstra(Node start, Node end)`.

The strategy design pattern allows you to create a family of interchangeable algorithms without relying subclassing and all of the perils that go with it. Here, `Graph` defines a search method that takes an arbitrary `SearchAlgorithm`:

```
public class Graph {
    /** This is the CONTEXT INTERFACE.
```

```
      * @return the shortest path from start to end
      * or throws PathNotFoundException if no path exists.
      */
    public Node[] search(Node start, Node end, SearchAlgorithm sa) {
        return sa.search(this, start, end);
    }
    // Remainder omitted
}

// THIS IS THE STRATEGY
public interface SearchAlgorithm {
    /** THIS IS THE ALGORITHM INTERFACE
      * @return the shortest path from start to end in graph
      * or throws PathNotFoundException if no path exists.
      */
    public Node[] search(Graph g, Node start, Node end);
}
```

Now it is possible to create an arbitrary number of concrete search algorithms such as BFS, Dijkstra's algorithm, Depth First Search, and so forth. The client can specify which search algorithm to use without changing the context interface or the implementation of graph.

Benefits:

- Provides a choice of implementations with different time and space tradeoffs.
- Avoids unnecessary subclassing.
- Defines neat hierarchy of algorithms.

Drawbacks:

- Clients must know about different algorithms, exposing implementation details.
- New algorithms may require more information than provided by the original context interface.

QUESTION: What support does Java provide for sorting in the Collection framework?

ANSWER: Java provides a `sort(List)` method in `java.util.Collections` that implements a modified mergesort (which is stable and guarantees O(n lg n) performance). Note that this is not a Strategy and additional sort algorithms would require new method names.

EXERCISE: Work in small groups to come up with a Strategy for sorting a `java.util.List`.

HINTS/SOLUTION:
Students should develop a Context Interface, Strategy, Algorithm Interface, and some sample implementation specifications. For example:

```
// This method could be reasonably implemented in Collections or List
// Generics have been omitted but should be present in a real
solution.
public interface List extends Collection {
    /** This is the CONTEXT INTERFACE.
      * @modifies this
      * @effects Sorts this according to the natural ordering of its
elements.
      */
    public void sort(Sort Algorithm sa) {
        return sa.sort(this);
    }
    // Remainder omitted
}

// THIS IS THE STRATEGY
public interface SortAlgorithm {
    /** THIS IS THE ALGORITHM INTERFACE
      * @modifies this
      * @effects Sorts this according to the natural ordering of its
elements.
      */
    public void sort(List list);
}

// SAMPLE SPECIFICATIONS
public class MergeSort implements SortAlgorithm {
    /** @effects Performs an O(n lg n) mergesort on the list
(stable).
    public void sort(List list) {
        //...
    }

public class QuickSort implements SortAlgorithm...
public class BubbleSort implements SortAlgorithm...
public class InsertionSort implements SortAlgorithm...
public class HeapSort implements SortAlgorithm...
```

QUESTION: Can we include Radix Sort in our Strategy? It can only sort a list of Integers.

ANSWER: One could argue no, because it does not refine the Context Interface by placing a requires clause on the contents of the list. On the other hand, a user must choose a SortAlgorithm by reading the concrete class specs, and it would not be unreasonable for sort to throw `IllegalArgumentException` if the list does not fit the algorithm's requires clause.

Design Pattern 2: Composite

Human beings see the world in composites naturally. When you look at another person, you (usually) see just that, another person. But a person is not a fundamental type.

Persons contain organs. But wait! Organs are not a fundamental type either. Organs contain molecules, which contain atoms, which contain protons, which contain quarks. And if quarks are not the fundamental type, we do not know what they might contain.

Nevertheless, we want to be able to treat containers as fundamental types (called a ***Leaf***) , abstracting away their children until they become relevant to us. Graphical user interfaces can be treated this way. We have a screen full of windows. But the window contains a button and a dialog box, and the dialog contains a button, and the button is a Leaf.

Here is an example of GUI hierarchy in AWT:

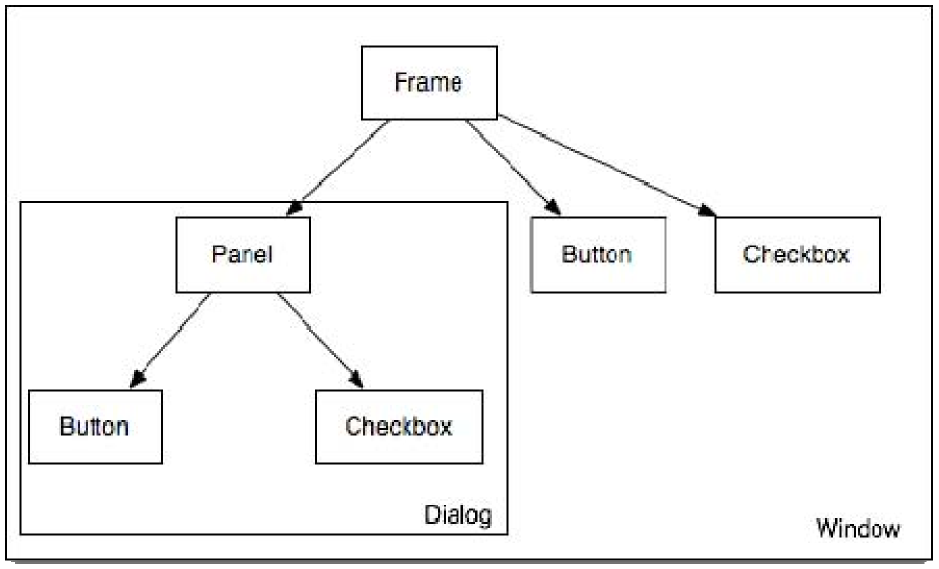

Figure 2: GUI hierarchy in AWT

`Button` and `Checkbox` are subclasses of `java.awt.Component`. Button and Checkbox are leaves. `Frame` and `Panel` are subclasses of `java.awt.Container`. They represent composites that contain leaves.

But wait a minute! The `Frame` in this example contains a `Panel`. How can it treat `Panel` like Leaf? `java.awt.Container` is also a subclass of `java.awt.Component`, so composites can be accessed through the same interface as the leaves, which cannot behave like containers.

Benefits:

- Defines hierarchies consisting of fundamental types (leaves) and composites.
- Simplifies clients, which can treat all objects in hierarchy uniformly.
- Makes it easier to expand the hierarchy with new classes.

Issues to consider:

- Should Component define child management? (i.e. should leaves have methods for children?)
- Should children reference their parents?
- Does the root node of the hierarchy need special properties (as in `javax.swing.JFrame`)?
- How are children ordered?

Some discussion points for above:
Defining child management at the root allows all components to be treated uniformly, but you have to handle clients performing meaningless add and remove operations on leaves. This is how Swing does it; `javax.swing.JComponent` extends `java.awt.Container`.

Defining child management for the composite interface only gives better safety but you lose some transparency. This was the choice for AWT, as described above.

Having children include references to their parents is sometimes useful for traversing up the tree, but the root will have to be handled specially, and it makes the implementation of add and remove methods slightly more complex. AWT and Swing provide the `getParent()` method.

In Swing, components are "lightweight" objects to improve performance. However, a lightweight component is inadequate to provide the features needed for a `JFrame` that serves as the main window for a Java program. Thus, the root of the tree in Swing needs a different interface. This is not the case for AWT, in which all components including `Frame` are "heavyweight."

Do you order children front-to-back, back-to-front, or not at all (isomorphic trees)? Swing uses front-to-back "z-order" to determine which children get painted on top in a GUI.

Java Puzzler

"It's Absurd, It's a Pain, It's a Superclass!"
Puzzle 80 by Joshua Bloch

Many of you made use of inner classes in Problem Sets 4 & 5 to construct `Node` or `Edge` classes. This puzzle describes one of the many pitfalls of inner classes.

```
public class Outer {
    class Inner1 extends Outer {}
    class Inner2 extends Inner1 {}
}
```

Why does this code generates the following cryptic compile error?

```
Outer.java:3: cannot reference this before
                supertype constructor has been called
    class Inner2 extends Inner1 {}
    ^
```

The answer lies in the fact that classes have implicit default constructors with an implicit call to the superclass default constructor. Since Inner2 also inherits Inner1 indirectly from Outer, the innermost enclosing instance is referenced with `this`.

```
public Inner2() {
    this.super();
}
```

But `this` (the enclosing instance of `Inner1`) has not been constructed yet! The brute-force way to fix this problem is to provide a more reasonable enclosing instance explicitly:

```
public Inner2() {
    Outer.this.super(); // Works, but really cryptic!
}
```

There is a better solution: **Whenever you write a member class, ask yourself, Does this class really need an enclosing instance? If the answer is no, make it static.** Inner classes are sometimes useful, but they can easily introduce complications that make a program difficult to understand. They have complex interactions with generics, reflection, and inheritance (as seen here). If you declare `Inner1` to be `static`, the problem goes away. If you also declare `Inner2` to be `static`, you can actually understand what the program does.

Finally, it is rarely appropriate to extend an inner class.