Introduction to Design Patterns

- What is Design Pattern?
- The Container Pattern.
- The Iterator Pattern.
- The Visitor Pattern.
- The SearchableContainer Pattern.
- The Association Pattern.
- The class Hierarchy.
- Review Questions.

What is Design Pattern?

- We've seen that inheritance allows related classes to share code, thus, allowing for code reusability, flexibility, etc.
- How about unrelated classes - can we make them reusable?
- Experience in the object-oriented design community has shown that interaction among even unrelated objects often takes the form of a recurring pattern or set of patterns.
- A number of these patterns have been identified and are being referred to as object-oriented design patterns.
- Learning to use these patterns makes code to become even more reusable, more flexible, etc.
- We shall use some of these patterns throughout the course and the basic ones are being introduced in this lecture.

The Container Pattern

- A container is an object that holds within it other objects.
- Many of the data structures we study in this course can be viewed as containers. i.e. they have the container pattern.
- The Container interface can be defined as below:

```
public interface Container {
        int getCount ();
        boolean isEmpty ();
        boolean isFull ();
        void purge ();
        void accept (Visitor visitor);
        Iterator iterator ();
}
```

- The first four methods are obvious. We explain the other two after introducing Iterator and Visitor patterns.

The AbstractContainer Class

- The following is the AbstractContainer class, that implements the Container interface and which will be used as base from which concrete container classes are derived.

```
public abstract class AbstractContainer
      implements Container {
   protected int count;

   public int getCount () {return count;}
   public boolean isEmpty () {return getCount () == 0;}

   public boolean isFull () {
       return false;
   }
   public abstract void purge();
   public void accept(Visitor v){. . . }
   public abstract Iterator iterator();
   // ...
}
```

The Iterator Pattern

- The Iterator pattern provides a means to access one-by-one, all the objects in a container.
- The following shows the Iterator interface:

```
public interface Iterator {
        boolean hasNext ();
        Object next () throws NoSuchElementException;
}
```

- An Iterator interacts closely with a container. Recall that a container has a method iterator, which returns an Iterator.
- The following shows how the Iterator interface is used:

```
Container c = new SomeContainer();
Iterator e = c.iterator();
while (e.hasNext())
   System.out.println(e.next ());
```

The Visitor Pattern

- Many operations on data structures are observed to have the pattern of visiting each object - hence, the Visitor pattern.

- For this pattern we define the Visitor interface as follows:

```
public interface Visitor  {
       void visit (Object object);
       boolean isDone ();
}
```

- A visitor interacts closely with a container. The interaction goes as follows:
  - The container is passed a reference to a visitor by calling the container's **`accept`** method.
  - The container then calls the **`visit`** method of the visitor one-by-one for each object it contains.

The Visitor Pattern (Contd.)

- The design framework for the accept method is as follows:

```
   public void accept(Visitor visitor)  {
     for each object, obj, in this container
       visitor.visit(obj);
}
```

- The code for a sample visitor is shown below:

```
public class PrintingVisitor implements Visitor  {
   public void visit(Object object)  {
     System.out.println(object);
}
```

- Note: The loop that iterates the container's objects is not in the visit method; It is in the container's accept method.
- To print all objects in an instance, c of SomeContainer, the accept method is called as follows:

```
Container c = new SomeContainer();
//....
c.accept (new PrintingVisitor());
```

The isDone method

```
public void accept (Visitor visitor){
   for each Object o in this container
      if (visitor.isDone ())
          return;
      visitor.visit (o);
}
```

- The following shows the usefulness of the isDone method:

```
public class MatchingVisitor implements Visitor  {
      private Object target; private boolean found;
      public MatchingVisitor(Object target){
            this.target = target;
      }
      public void visit (Object object)  {
        if(object.equals (target))
            found = true;
     }
     public boolean isDone (){return found;}
}
```

The AbstractVisitor Class

- Many operations on a container involves visiting all the elements. i.e. they do not need to call the isDone method.
- Thus, forcing the implementation of the isDone method for such operations may not be desirable.
- To avoid this, we define the following AbstractVistor class.

```
public abstract class AbstractVisitor implements Visitor {

	public abstract void visit (Object object);

	public boolean isDone () {
		return false;
	}
}
```

The toString Method

- The following defines the **`toString`** method for the AbstractContainer class using a visitor.
- Defining it here is aimed at simplifying the implementation of classes extending this class.

```
public abstract class AbstractContainer implements Container {
   public String toString() {
        final StringBuffer buffer = new StringBuffer();
        AbstractVisitor visitor = new AbstractVisitor() {
                private boolean comma;
                public void visit(Object obj) {
                   if(comma) buffer.append(“, ”);
                   buffer.append(obj);
                   comma = true;
                }
        };

        accept(visitor);
        return "" + buffer;
   }
       // ...
}
```

The accept Method

- We now define the accept method for the AbstractContainer class using an iterator.

```
public abstract class AbstractContainer
      implements Container {
   public void accept(Visitor visitor) {
       Iterator iterator = iterator();

       while ( iterator.hasNext() && !visitor.isDone())
          visitor.visit(iterator.next());
   }
   // ...
}
```

- While the **accept** method takes only one visitor, a container can have more than one Iterator at the same time.

The SearchableContainer Pattern

- Some of the data structures we shall study have the additional property of being searchable.
- The SearchableContainer interface extends the Container interface by adding four more methods as shown below:

```
public interface SearchableContainer extends Container {
        boolean isMember (Comparable object);
        void insert (Comparable object);
        void withdraw (Comparable obj);
        Comparable find (Comparable object);
}
```

- The **`find`** method is used to locate an object in the container and returns its reference. It returns null if not found.

The Association Pattern

- An association is an ordered pair of objects.
- The first element is called the key, while the second is the value associated with the key.
- The following defines the Association class which we shall use whenever we need to associate one object to another.

```
public class Association implements Comparable {
    protected Comparable key;
    protected Object value;
    public Association(Comparable comparable, Object obj){
        key = comparable;
        value = obj;
    }

    public Association(Comparable comparable){
        this(comparable, null);
    }
    // ...
```

The Association Pattern (contd.)

```
    public Comparable getKey(){return key;}
    public void setKey(Comparable key){this.key = key;}
    public Object getValue(){return value;}
    public void setValue(Object value){this.value = value;}

    public int compareTo(Object obj){
        Association association = (Association)obj;
        return key.compareTo(association.getKey());
    }

    public boolean equals(Object obj){
        return compareTo(obj) == 0;
    }
    public String toString() {
        String s = "{ " + key;
        if(value != null)
            s = s + " ,   " + value;
        return s + " }";
    }
}
```

Review Question

1. Let c be an instance of some concrete class derived from the AbstractContainer class. Explain how the following statement prints the content of the container: `System.out.println(c);`
2. Suppose we have a container which contains only instances of the Integer class. Design a Visitor that computes the sum of all the integers in the container.
3. Using visitors, devise implementations for the isMember and find methods of the AbstractSearchableContainer class. Using visitors, devise implementations for the isMember and find methods of the AbstractSearchableContainer class.
4. Consider the following pair of Associations:

   ```
   Comparable a=new Association (new Integer(3), new Integer(4));
   Comparable b=new Association (new Integer(3));
   ```

   Give the sequence of methods called in order to evaluate a comparison such as "**`a.equals(b)`**". Is the result of the comparison true or false?