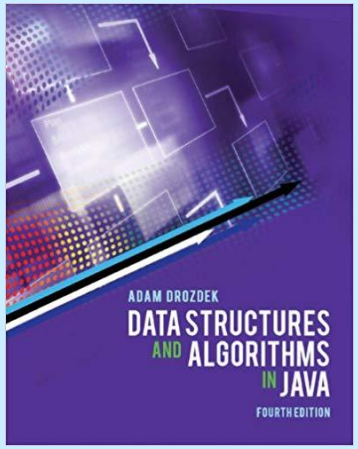
CSE 202 : DATA STRUCTURES

Lecture # 02:



Introduction to Design Patterns

Dr. Ahmed Abba Haruna
Assistant Professor,
University of Hafr Al-Batin (UoHB)
College of Computer Science and Engineering (CCSE)
Hafr Al-Batin 31991, Saudi Arabia.
aaharuna@uhb.edu.sa

Introduction to Design Patterns

- What is a Design Pattern?
  1. *Creational Design Patterns*
  2. *Structural Design Patterns*
  3. *Behavioral Design Patterns*
  - Example: Using **Iterator** Pattern
- Examples of some important Design Patterns
  - The Container Pattern.
  - The Visitor Pattern.
  - The SearchableContainer Pattern.
  - The Iterator Pattern.
  - The Association Pattern.
- Example: Implementation of Iterator
- Review Questions.

What is a Design Pattern?

- **Design patterns**, as name suggest, are ***solutions for most commonly (and frequently) occurred problems while designing a software***.
- A design pattern is a standard ***interface, e.g., Comparable***.
- None of these patterns force you anything in regard to implementation; ***they are just guidelines*** to solve a particular problem – in a particular way – in particular contexts. Code implementation is your responsibility.
- There are 3 types of design patterns:
  1. ***Creational Design Patterns***
  2. ***Structural Design Patterns***
  3. ***Behavioral Design Patterns***

1. Creational Design Patterns

- Creational patterns are often ***used in place of direct instantiation with constructors***. They can provide flexibility about which objects are created, how those objects are created, and how they are initialized.

Name	Purpose
Builder	Builder design pattern is an alternative way to construct complex objects and should be used only when we want to build different types of immutable **objects** using same object building process.
Prototype	Prototype design pattern is used in scenarios where application needs to create a **large number of instances** of a class, which have almost same state or differ very little.
Factory	Factory design pattern is most suitable when **complex steps of object creation** are involved. To ensure that these steps are centralized and not exposed to composing classes.
Abstract factory	Abstract factory pattern is used whenever we need **another level of abstraction** over a group of factories created using factory pattern.
Singleton	Singleton enables an application to have **one and only one instance** of a class per JVM.

2. Structural Design Patterns

- Structural design patterns show us ***how to glue different pieces of a system together*** in a flexible and extensible fashion. They assure that when one of the parts changes, the entire application structure does not.

Name	Purpose
Adapter	An adapter **converts the interface of a class** into another interface clients expect. It lets classes work together that couldn't otherwise due to incompatible interfaces.
Bridge	Bridge design pattern is used to **decouple a class into two parts – *abstraction* and it's *implementation*** – so that both can evolve in future without affecting each other. It increases the loose coupling between class abstraction and it's implementation.
Composite	Composite design pattern helps to **compose the objects into tree structures** to represent whole-part hierarchies. It lets clients treat individual objects and compositions of objects uniformly.
Decorator	Decorator design pattern is used to **add additional features or behaviors** to a particular instance of a class, while not modifying the other instances of same class.
Facade	Facade design pattern provide a **unified interface to a set of interfaces** in a sub-system. It defines a higher-level interface that makes the subsystem easier to use.
Flyweight	Flyweight design pattern enables use **sharing of objects** to support large numbers of fine-grained objects efficiently. A flyweight is a shared object that can be used in multiple contexts simultaneously. It acts as an independent object in each context.
Proxy	In proxy design pattern, a proxy object **provide a surrogate or placeholder** for another object to control access to it. Proxy is heavily used to implement lazy loading related use-cases where we do not want to create full object until needed.

3. Behavioral Design Patterns

- Behavioral patterns ***abstract an action we want to take on the object or class that takes the action***. By changing the object/class, we can change the algorithm used, the objects affected, or the behavior, while still retaining the same basic interface for client classes.

Name	Purpose
Chain of responsibility	Chain of responsibility design pattern gives more than one object an opportunity to handle a request by **linking receiving objects together** in form of a chain.
Command	Command design pattern is useful to abstract the **business logic into discrete actions** which we call commands. These command objects help in loose coupling between two classes where one class (invoker) shall call a method on other class (receiver) to perform a business operation.
Interpreter	Interpreter pattern specifies how to **evaluate sentences in a language**, programatically. It helps in building a grammar for a simple language, so that sentences in the language can be interpreted.
Iterator	Iterator pattern provides a way to **access the elements** of an aggregate object sequentially without exposing its underlying representation.
	Cont…

Cont…

Name	Purpose
Mediator	Mediator pattern **defines an object that encapsulates how a set of objects interact**. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets us vary their interaction independently.
Memento	Memento pattern is used to **restore state of an object** to a previous state. It is also known as ***snapshot pattern***.
Observer	Observer pattern defines a **one-to-many dependency between objects** so that when one object changes state, all its dependents are notified and updated automatically. It is also referred to as the ***publish-subscribe pattern***.
State	In state pattern allows an **object to alter its behavior when its internal state changes**. The object will appear to change its class. There shall be a separate concrete class per possible state of an object.
Strategy	Strategy pattern is used where we **choose a specific implementation** of algorithm or task in run time – out of multiple other implementations for same task.
Template method	Template method pattern defines the **sequential steps to execute a multi-step algorithm** and optionally can provide a default implementation as well (based on requirements).
Visitor	Visitor pattern is used when we want a **hierarchy of objects to modify their behavior** but without modifying their source code.

Example : Using Iterator Pattern

- ***Iterator Pattern is used "to access the elements of an aggregate object sequentially without exposing its underlying implementation".***

```
import java.util.ArrayList;
import java.util.Iterator;

public class IteratorPatternDemo {
  public static void main(String args[]){
    ArrayList names = new ArrayList();
    names.add("Dr. Muhammad Akhlaq");
    names.add("Dr. Lawan Mohammed");
    names.add("Dr. Omer Marey");
    names.add("Dr. Zeshan Mehfooz");
    names.add("Dr. Musab Isah");

    names.remove("Dr. Lawan Mohammed");

    Iterator it = names.iterator();

    while(it.hasNext()) {
      String obj = (String)it.next();
      System.out.println(obj);
    }
  }
}
```

Output
Dr. Muhammad Akhlaq
Dr. Omer Marey
Dr. Zeshan Mehfooz
Dr. Musab Isah

Examples of some important Design Patterns

The Container Pattern

- A container is an object that holds other objects within it.
- 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 is defined as follows:

```
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 Visitor and Iterator patterns.

The AbstractContainer Class

- The following is the AbstractContainer class, that implements the Container interface and which will be used as a 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 abstract void accept(Visitor v);
   public abstract Iterator iterator();
   // ...
}
```

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 **`accept`** method then calls the **`visit`** method of the visitor, one-by-one,
    - for each object in the container, OR
    - for each object in the container as long as a certain condition is satisfied/not satisfied.

The Visitor Pattern (Contd.)

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

```
public class SomeContainer implements Container  {
   public void accept(Visitor visitor)  {
     for each object, o, in this container
       visitor.visit(o);
   }
}
```

- The code for a sample visitor is shown below:

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

- 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 Object found;
      public MatchingVisitor (Object target)  {
            this.target = target;}
      public void visit (Object object)  {
        if (!isDone () && object.equals (target))
            found = object;
     }
     public boolean isDone (){return found != null;}
}
```

The AbstractVisitor Class

- Many operations on a container involve 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;//converts result to a string
   }
       // ...
}
```

The Iterator Pattern

- Like Visitor, an 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 Iterator interface is used:

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

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

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());
   }
   // ...
}
```

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);
}
```

isMember(): tests whether a given object instance is in the container

insert(): puts objects into the container

withdraw(): removes objects from the container

find(): locates objects 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 + " }";
    }
}
```

Example: Implementation of ***Iterator***

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

interface Container {
        public Iterator getIterator();
}

class CollectionofNames implements Container {
        public String name[] = {    "Dr. Muhammad Akhlaq",
                                    "Dr. Omer Marey",
                                    "Dr. Zeshan Mehfooz",
                                    "Dr. Musab Isah"};

        @Override
        public Iterator getIterator() {
                return new CollectionofNamesIterate() ;
        }
```

Cont…

```
        private class CollectionofNamesIterate implements Iterator{
                int i;

                @Override
                public boolean hasNext() {
                        if (i<name.length){
                                return true;
                        }
                        return false;
                }

                @Override
                public Object next() {
                        if(this.hasNext()){
                                return name[i++];
                        }
                        return null;
                }
        }
}
```

Cont…

```
public class IteratorPatternDemo {

      public static void main(String[] args) {

            CollectionofNames   cmpnyRepository = new   CollectionofNames();

            for(Iterator iter = cmpnyRepository.getIterator(); iter.hasNext();){
                  String name = (String)iter.next();
                  System.out.println("Name : " + name);
             }
      }
}
```

Review Questions

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.
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?

Sources:

- https://howtodoinjava.com/gang-of-four-java-design-patterns/
- https://www.javatpoint.com/iterator-pattern