Design Patterns in Java

A design patterns are well-proved solution for solving the specific problem/task.

For example

Problem Given: Suppose you want to create a class for which only a single instance (or object) should be created and that single object can be used by all other classes.

Solution: Singleton design pattern is the best solution of above specific problem. So, every design pattern has some specification or set of rules for solving the problems. What are those specifications, you will see later in the types of design patterns.

But remember one-thing, design patterns are programming language independent strategies for solving the common object-oriented design problems. That means, a design pattern represents an idea, not a particular implementation.

By using the design patterns you can make your code more flexible, reusable and maintainable. It is the most important part because java internally follows design patterns.

To become a professional software developer, you must know at least some popular solutions (i.e. design patterns) to the coding problems.

Advantage of design pattern:

They are reusable in multiple projects.

They provide the solutions that help to define the system architecture.

They capture the software engineering experiences.

They provide transparency to the design of an application.

They are well-proved and testified solutions since they have been built upon the knowledge and experience of expert software developers.

Design patterns don't guarantee an absolute solution to a problem. They provide clarity to the system architecture and the possibility of building a better system.

When should we use the design patterns?

We must use the design patterns during the analysis and requirement phase of SDLC(Software Development Life Cycle).

Design patterns ease the analysis and requirement phase of SDLC by providing information based on prior hands-on experiences.

What is Gang of Four (GOF)?

In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book titled **Design Patterns - Elements of Reusable Object-Oriented Software** which initiated the concept of Design Pattern in Software development.

These authors are collectively known as **Gang of Four (GOF)**. According to these authors design patterns are primarily based on the following principles of object orientated design.

- Program to an interface not an implementation
- Favor object composition over inheritance

Usage of Design Pattern

Design Patterns have two main usages in software development.

Common platform for developers

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

Best Practices

provide best solutions to certain problems faced during software development. Learning these patterns helps un experienced developers to learn software design in an easy and faster way.

Types of Design Patterns

As per the design pattern reference book **Design Patterns - Elements of Reusable Object-Oriented Software** , there are 23 design patterns which can be classified in three categories: Creational, Structural and Behavioral patterns. We'll also discuss another category of design pattern: J2EE design patterns.

S.N.	Pattern & Description
1	**Creational Patterns** These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.

2	**Structural Patterns** These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
3	**Behavioral Patterns** These design patterns are specifically concerned with communication between objects.
4	**J2EE Patterns** These design patterns are specifically concerned with the presentation tier. These patterns are identified by Sun Java Center.

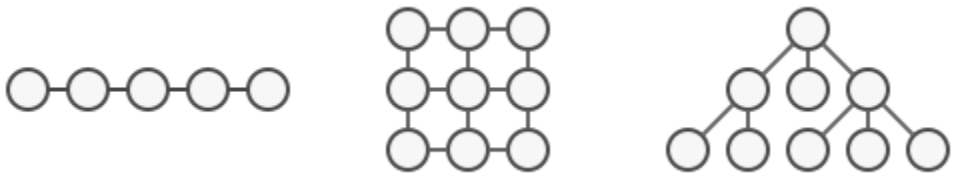
Iterator Pattern

Iterator pattern is very commonly used design pattern in Java and .Net programming environment. This pattern is **used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation**.

Iterator pattern falls under behavioral pattern category.

Problem

Collections are one of the most used data types in programming. Nonetheless, a collection is just a container for a group of objects.



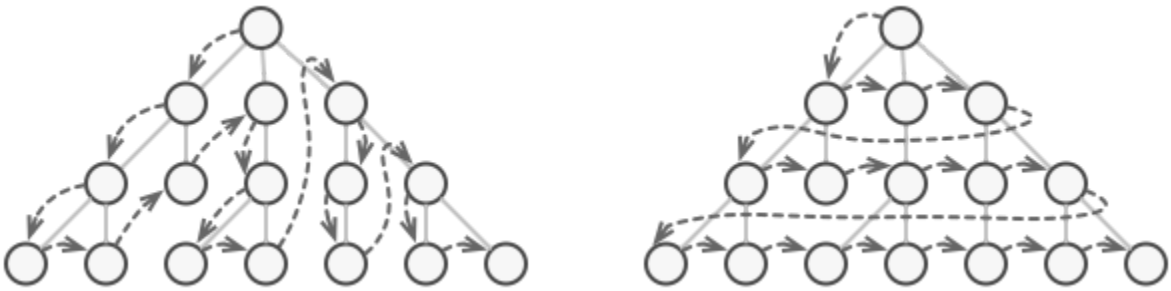
Various types of collections.

Most collections store their elements in simple lists. However, some of them are based on stacks, trees, graphs and other complex data structures.

But no matter how a collection is structured, it must provide some way of accessing its elements so that other code can use these elements. There should be a way to go through each element of the collection without accessing the same elements over and over.

This may sound like an easy job if you have a collection based on a list. You just loop over all of the elements. But how do you sequentially traverse elements of a complex data structure, such as a tree? For example, one day you might be just fine with depth-first traversal of a tree. Yet the

next day you might require breadth-first traversal. And the next week, you might need something else, like random access to the tree elements.

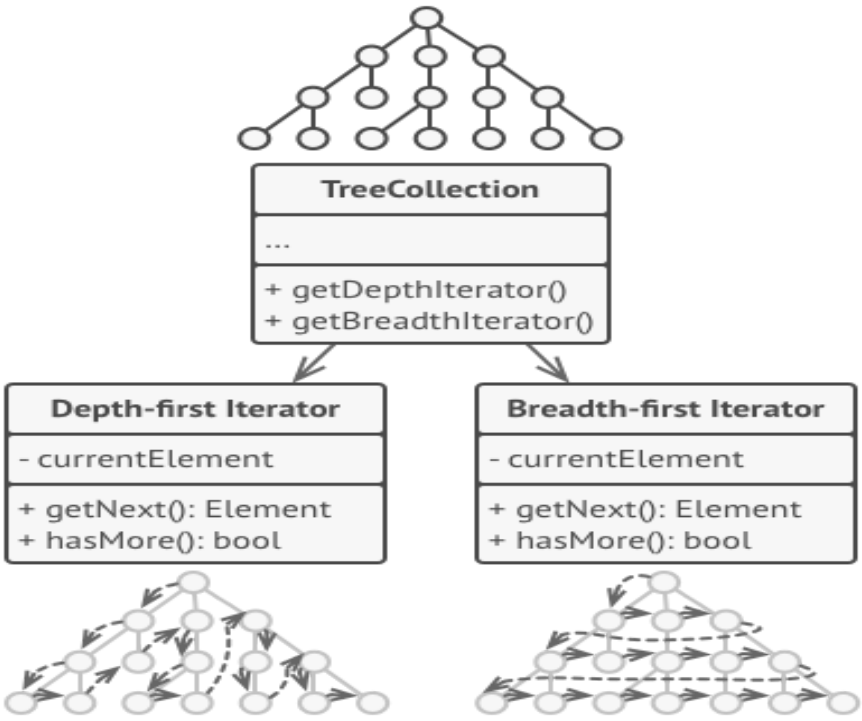


The same collection can be traversed in several different ways.

Adding more and more traversal algorithms to the collection gradually blurs its primary responsibility, which is efficient data storage. Additionally, some algorithms might be tailored for a specific application, so including them into a generic collection class would be weird.

On the other hand, the client code that's supposed to work with various collections may not even care how they store their elements. However, since collections all provide different ways of accessing their elements, you have no option other than to couple your code to the specific collection classes.

Solution



The main idea of the Iterator pattern is to extract the traversal behavior of a collection into a separate object called an *iterator*.

Iterators implement various traversal algorithms. Several iterator objects can traverse the same collection at the same time.

In addition to implementing the algorithm itself, an iterator object encapsulates all of the traversal details, such as the current position and how many elements are left till the end. Because of this, several iterators can go through the same collection at the same time, independently of each other.

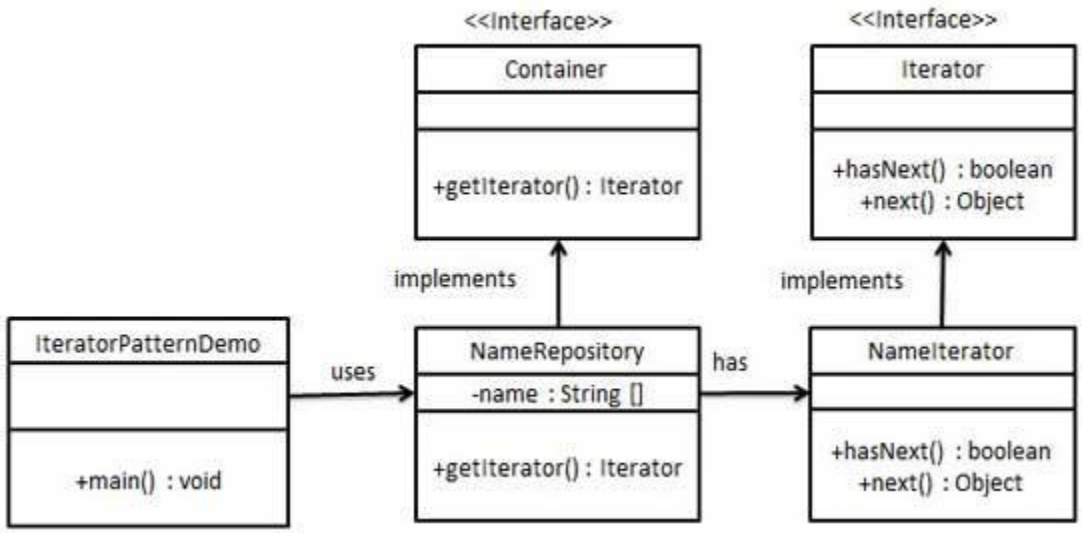
Usually, iterators provide one primary method for fetching elements of the collection. The client can keep running this method until it doesn't return anything, which means that the iterator has traversed all of the elements.

All iterators must implement the same interface. This makes the client code compatible with any collection type or any traversal algorithm as long as there's a proper iterator. If you need a special way to traverse a collection, you just create a new iterator class, without having to change the collection or the client.

Implementation

We're going to create a *Iterator* interface which narrates navigation method and a *Container* interface which retruns the iterator . Concrete classes implementing the *Container* interface will be responsible to implement *Iterator* interface and use it

IteratorPatternDemo, our demo class will use *NamesRepository*, a concrete class implementation to print a *Names* stored as a collection in *NamesRepository*.



References-

https://refactoring.guru/design-patterns/iterator

https://www.tutorialspoint.com/design_pattern/iterator_pattern.htm

Code Example

```
interface Iterator {
   public boolean hasNext();
   public Object next();
}
interface Container {
   public Iterator getIterator();
}
class NameRepository implements Container {
   public String names[] = {"Student1" , "Student2" ,"Student3" , "Student4"};

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

   private class NameIterator implements Iterator {

      int index;

      @Override
```

```
    public boolean hasNext() {

      if(index < names.length){
        return true;
      }
      return false;
    }

    @Override
    public Object next() {

      if(this.hasNext()){
        return names[index++];
      }
      return null;
    }
  }
}
public class IteratorPatternDemo {

  public static void main(String[] args) {
    NameRepository namesRepository = new NameRepository();

    for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
      String name = (String)iter.next();
```

```
        System.out.println("Name : " + name);
    }
  }
}
```

Output

```
$java -Xmx128M -Xms16M IteratorPatternDemo
Name : Student1
Name : Student2
Name : Student3
Name : Student4
```

MVC Pattern

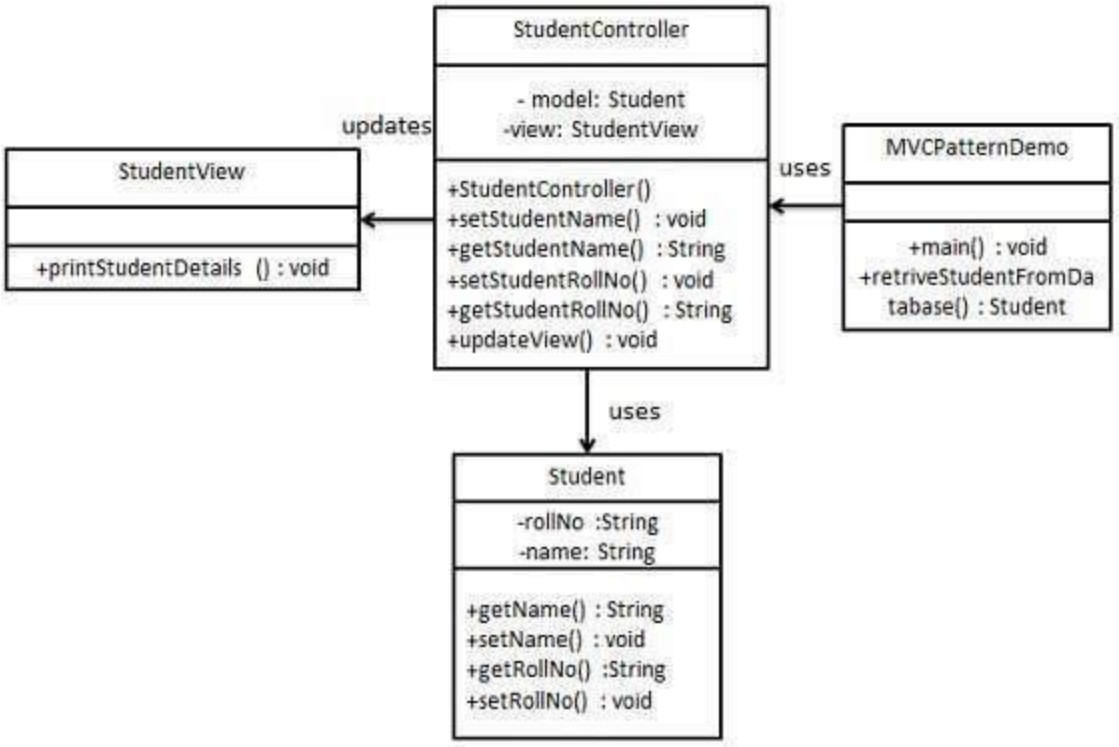
stands for Model-View-Controller Pattern. This pattern is used to separate application's concerns.

- **Model** - Model represents an object or JAVA POJO (**plain old Java object**) carrying data. It can also have logic to update controller if its data changes.
- **View** - View represents the visualization of the data that model contains.
- **Controller** - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

Implementation

We are going to create a *Student* object acting as a model.*StudentView* will be a view class which can print student details on console and *StudentController* is the controller class responsible to store data in *Student* object and update view *StudentView* accordingly.

MVCPatternDemo, our demo class, will use *StudentController* to demonstrate use of MVC pattern.



Step 1-Create Model.

Student.java

```
public class Student {
   private String rollNo;
   private String name;

   public String getRollNo() {
      return rollNo;
   }

   public void setRollNo(String rollNo) {
      this.rollNo = rollNo;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }
}
```

Step 2

Create View.

StudentView.java

```
public class StudentView {
  public void printStudentDetails(String studentName, String studentRollNo){
    System.out.println("Student: ");
    System.out.println("Name: " + studentName);
    System.out.println("Roll No: " + studentRollNo);
  }
}
```

Step 3

Create Controller.

StudentController.java

```
public class StudentController {
  private Student model;
  private StudentView view;

  public StudentController(Student model, StudentView view){
    this.model = model;
    this.view = view;}

  public void setStudentName(String name){
    model.setName(name);                    }

  public String getStudentName(){
    return model.getName();                              }

  public void setStudentRollNo(String rollNo){
    model.setRollNo(rollNo);
  }

  public String getStudentRollNo(){
    return model.getRollNo();
  }

  public void updateView(){
    view.printStudentDetails(model.getName(), model.getRollNo());
  }
}
```

Step 4

Use the *StudentController* methods to demonstrate MVC design pattern usage.

MVCPatternDemo.java

```
public class MVCPatternDemo {
   public static void main(String[] args) {

      //fetch student record based on his roll no from the database
      Student model  = retriveStudentFromDatabase();

      //Create a view : to write student details on console
      StudentView view = new StudentView();

      StudentController controller = new StudentController(model, view);

      controller.updateView();

      //update model data
      controller.setStudentName("John");

      controller.updateView();
   }

   private static Student retriveStudentFromDatabase(){
      Student student = new Student();
      student.setName("Robert");
      student.setRollNo("10");
      return student;
   }
}
```

Step 5

Verify the output.

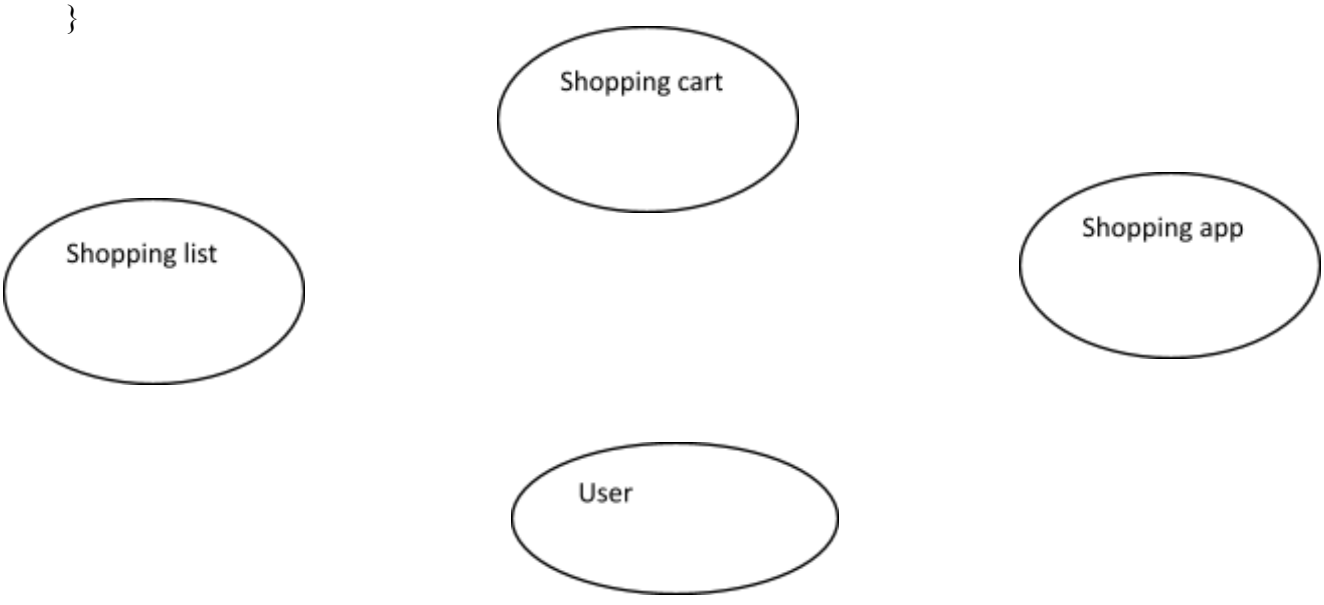
```
Student:
Name: Robert
Roll No: 10
Student:
Name: John
Roll No: 10
```

```
interface Iterator {
```

```
   public boolean hasNext();
   public Object next();
}
interface Container {
   public Iterator getIterator();
}
class NameRepository implements Container {
   public String names[] = {"Student1" , "Student2" ,"Student3" , "Student4"};

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

   private class NameIterator implements Iterator {

      int index;

      @Override
      public boolean hasNext() {

         if(index < names.length){
            return true;
         }
         return false;
```

```
  }

  @Override
  public Object next() {

    if(this.hasNext()){
      return names[index++];
    }
    return null;
  }
}private class RevIterator implements Iterator {

  int index=3;

  @Override
  public boolean hasNext() {

    if(index >= 0){
      return true;
    }
    return false;
  }

  @Override
  public Object next() {
```

```
        if(this.hasNext()){
           return names[index--];
        }
        return null;
      }
   }
}
public class IteratorPatternDemo {

   public static void main(String[] args) {
      NameRepository ob = new NameRepository();

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



```
class Student //model
{
   private String rollNo;
   private String name;

   public String getRollNo() {
      return rollNo;
   }

   public void setRollNo(String rollNo) {
      this.rollNo = 1000+rollNo;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }
}

class StudentView //view
```

```
{
   public void printStudentDetails(String studentName, String studentRollNo){
      System.out.println("Student info printed in GUI: ");
      System.out.println("Name: " + studentName);
      System.out.println("Roll No: " + studentRollNo);
   }

   public void printStudentDetails(String studentName){
      System.out.println("Student info printed in GUI: ");
      System.out.println("Name: " + studentName);

   }
}

class StudentController //controller
{
   private Student model;
   private StudentView view;

   public StudentController(Student model, StudentView view){
      this.model = model;
      this.view = view;}

   public void setStudentName(String name){
      model.setName(name);                  }
```

```
   public String getStudentName(){
      return model.getName();               }

   public void setStudentRollNo(String rollNo){
      model.setRollNo(rollNo);
   }

   public String getStudentRollNo(){
      return model.getRollNo();
   }

   public void updateView(){
      view.printStudentDetails(model.getName());
      //view.printStudentDetails(model.getName(), model.getRollNo());
   }
}

public class MVCPatternDemo //user
{
   public static void main(String[] args) {

      //fetch student record based on his roll no from the database
      Student model  = retriveStudentFromDatabase();
```

```
      //Create a view : to write student details on console
      StudentView view = new StudentView();

      StudentController controller = new StudentController(model, view);

      controller.updateView();// access view

      //update model data
      controller.setStudentName("John");// update model

      controller.updateView();


   }

   private static Student retriveStudentFromDatabase(){
      Student student = new Student();
      student.setName("ABC");
      student.setRollNo("10");
      return student;
   }
}
```