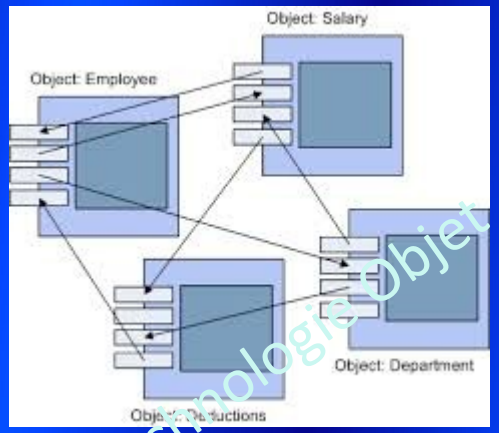
Chapitre 2

Design Patterns



Module : Technologie objet
Master 1 IRC

Mr A. Dekhinet
Université de BATNA
Département d'Informatique

Design patterns : Introduction

- ✓ Design patterns / Patrons de conception / Motifs de conception.
- ✓ Une **solution** générale réutilisable à un **problème** récurrent de conception orientée objet dans un **contexte** donnée.
- ✓ Tirent leur origine des travaux de l'architecte Christopher Alexander dans les années ’70
- ✓ Formalisés pour la première fois en 1995 dans le livre du ”Gang of Four ” (GoF : Bande des Quatres)
  GoF : Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides
  ” Design Patterns - Elements of Reusable Object-Oriented Software ”
- ✓ Christopher Alexander says, ” Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice. ”
- ✓ GoF says, ”Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it. ”
- ✓ Gof says, ” Design pattern are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. ”
- ✓ En général un design pattern possède quatre éléments essentiels : Nom, Problème, Solution (UML), Conséquence (Mise en œuvre : Complexité, Portabilité, …)
- ✓ Principaux patterns :
  - Model View Controller introduit dans Smalltalk-76 en 1970
  - GoF : 23 patterns (Patterns de référence dans le développement OO)
  - GRASP : 9 principes fondamentaux de conception OO

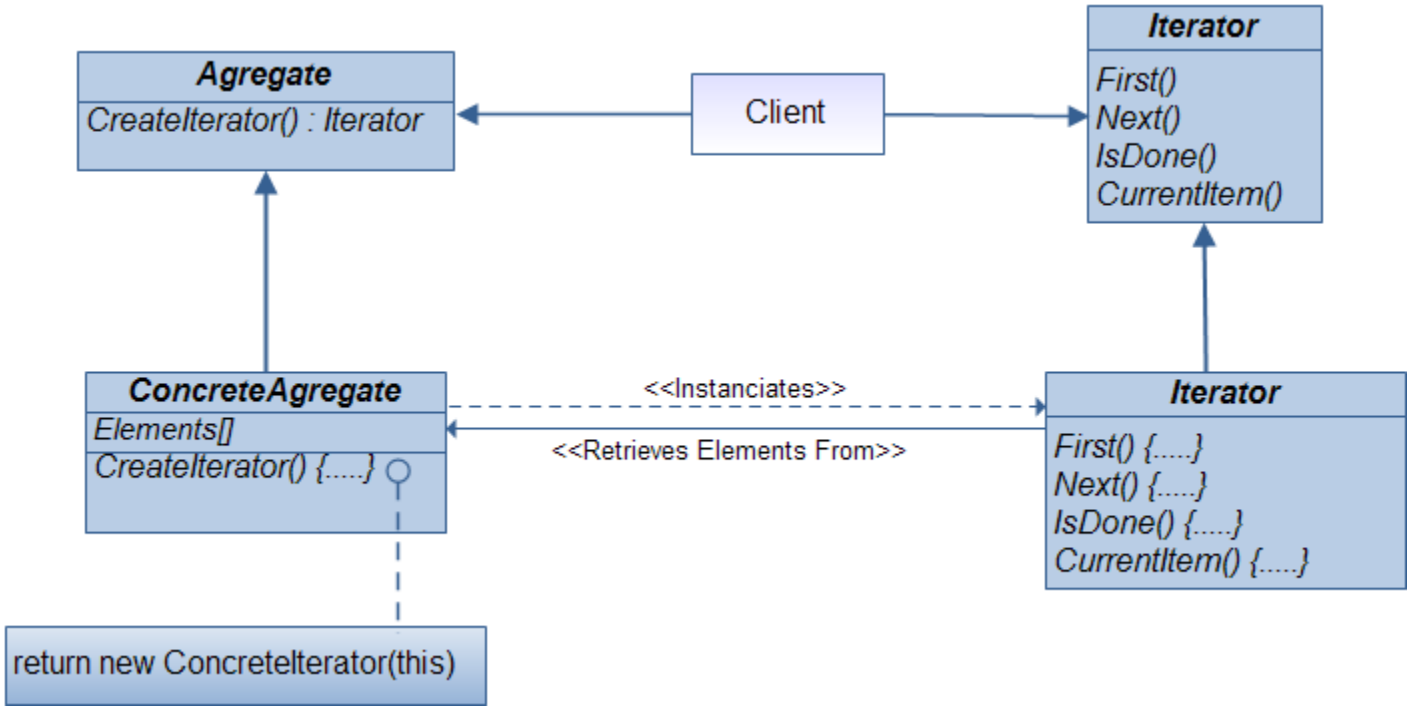
Gof Design patterns

- ✓ Les Design patterns sont classés en trois catégories :
  - Patterns de création (Creational patterns)
  - Patterns structuraux (Structural patterns)
  - Patterns comportementaux (Behavioral patterns)
- ✓ Patterns de création : Se préoccupent des problèmes d'instanciation ou de création des objets
- ✓ Patterns structuraux : Se préoccupent des problèmes des structures des classes (Séparant l'interface de l'implémentation)
- ✓ Patterns comportementaux : Se préoccupent des problèmes de collaboration des objets

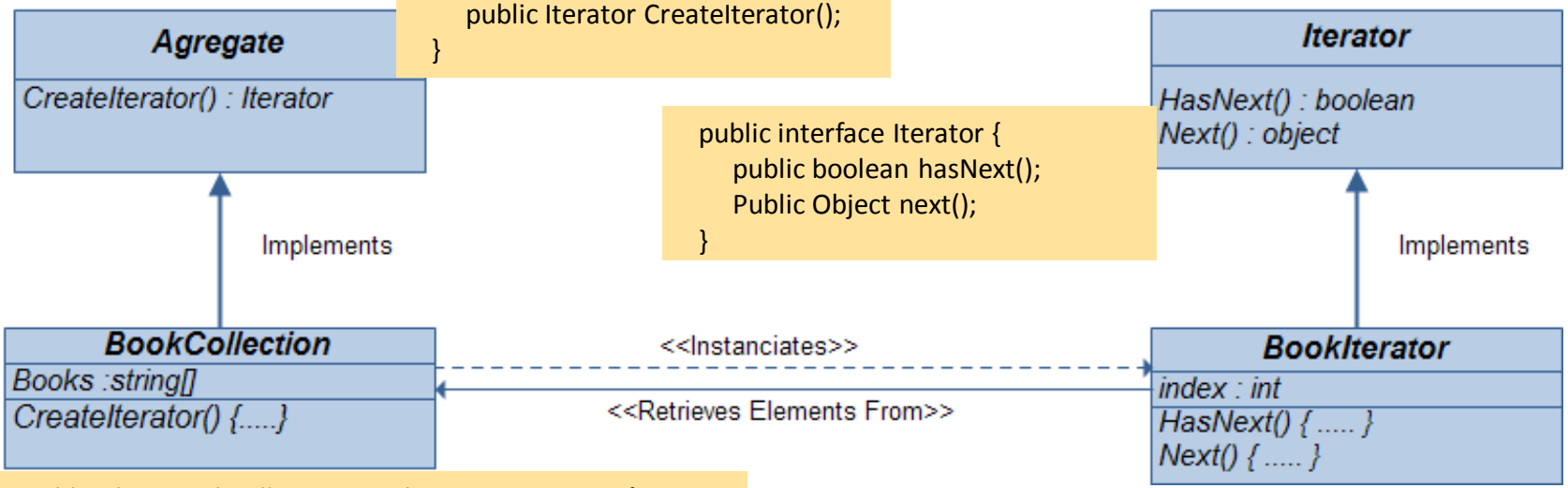
Patterns de Création	Patterns Structuraux	Patterns Comportementaux
Abstract Factory	Adapter	Chaine of responsability
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Facade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template method
		Visitor

Iterator (Cursor)

- ✓ Fournir un moyen d'accéder et de parcourir les éléments d'un objet agrégat (collection, conteneur) indépendamment de sa représentation interne et de son implémentation.
- ✓ L'idée fondamentale de ce pattern est de décharger l'objet agrégat de la responsabilité de l'accès et du parcours, et les déléguer a un autre objet : Itérateur
- ✓ Pointeur ou indice généralisé qui permet d'abstraire le parcours d'un agrégat



Iterator (Cursor)




```
public interface Agregate {
    public Iterator CreateIterator();
}
```

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

```
public class BookCollection implements Agregate {
private String  Books  [] = {"GoF" ,"OS" , "IRC"};
    @Override
    private Iterator CreateIterator() {
    return new BookIterator();
    }
}
```

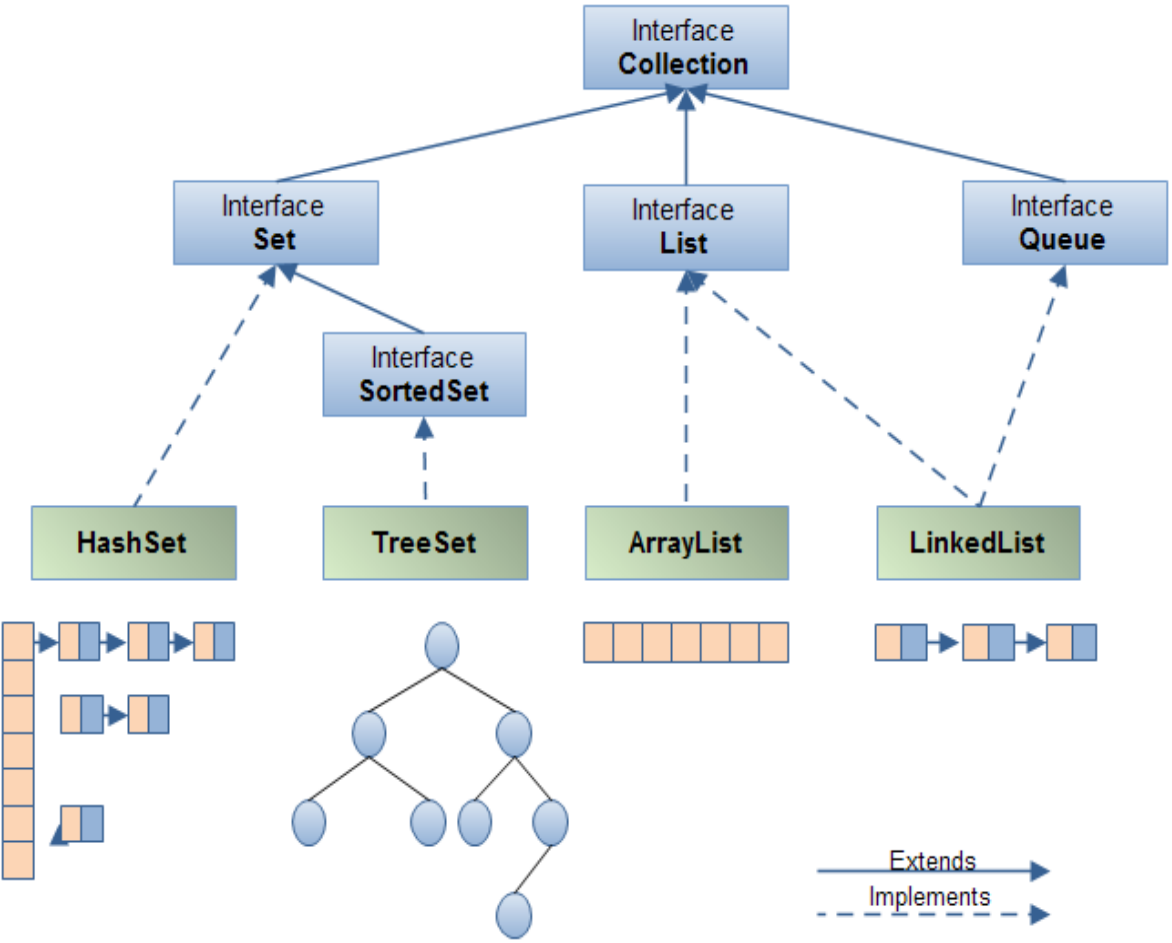
```
private class BookIterator implements Iterator  {
    int index;
    @Override
    public boolean hasNext() {
    if(index < Books.length){
                return true; }
                return false;
    }
    @Override
    public Object next() {
    if(this.hasNext()){
                return Books[index++];  }
                return null; }
    }
}
```

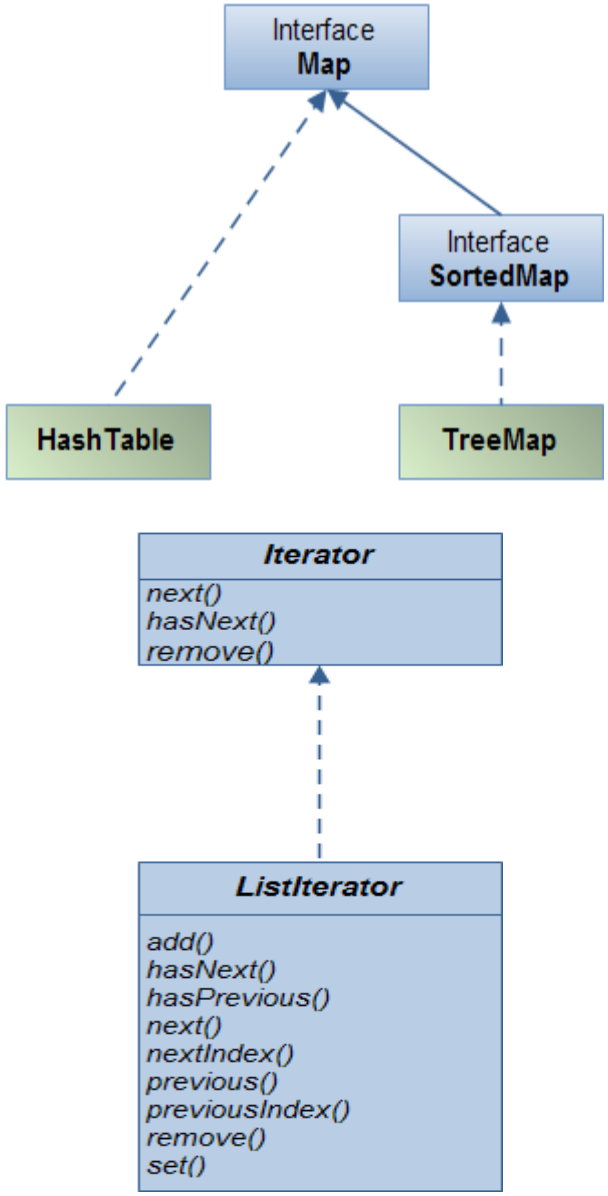
```
public class IteratorPatternDemo {
   public static void main(String[] args) {
   BookCollection bookCollection = new BookCollection();
      for (Iterator iter = bookCollection getIterator(); iter.hasNext();){
           String book = (String)iter.next(); System.out.println(book);
      }
   }
}
```

Iterator en JAVA

- ✓ L'interface **Iterable** en JAVA est le support principal du pattern Iterator
- ✓ L'API JAVA Fournit , par le biais du paquetage **java.util**, fournit des agrégats itérables (collection) :
  java.util.Iterator , **java.util.Collections**, …
- ✓ Exemples : Pile, File, Liste, Ensemble, Tableaux associatifs
- ✓ Les collections sont définies à partir de deux interfaces : **Collection<E>**, **Map<K,V>**
- ✓ Collection <E> : Collection simple dont l'accès se fait par un indice. Elle est paramétrée par le type des éléments.
- ✓ Map<K,V> : Collection associative (maps, hashtables) . L' accès à un élément **V** se fait à travers un indexe ou une clé **K**.
- ✓ Les classes collection, qui implémente l'interface Collection<E>, sont :
  AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayDeque, ArrayList, AttributeList, BeanContextServicesSupport, BeanContextSupport, ConcurrentLinkedQueue, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingDeque, LinkedBlockingQueue, LinkedHashSet, LinkedList, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, Stack, SynchronousQueue, TreeSet, Vector
- ✓ Les classes Map , qui implémente l'interface Map<K,V>, sont :
  AbstractMap, Attributes, AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, SimpleBindings, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap
- ✓ L'interface **Iterator<E>** correspond à la classe Iterator du GoF.
- ✓ L'interface **ListIterator<E>** étend Iterator<E> en enricheant l'ensemble des opérations
- ✓ Toutes les collections doivent avoir une méthode qui crée et renvoi un Iterator

Iterator en JAVA





Iterator en JAVA

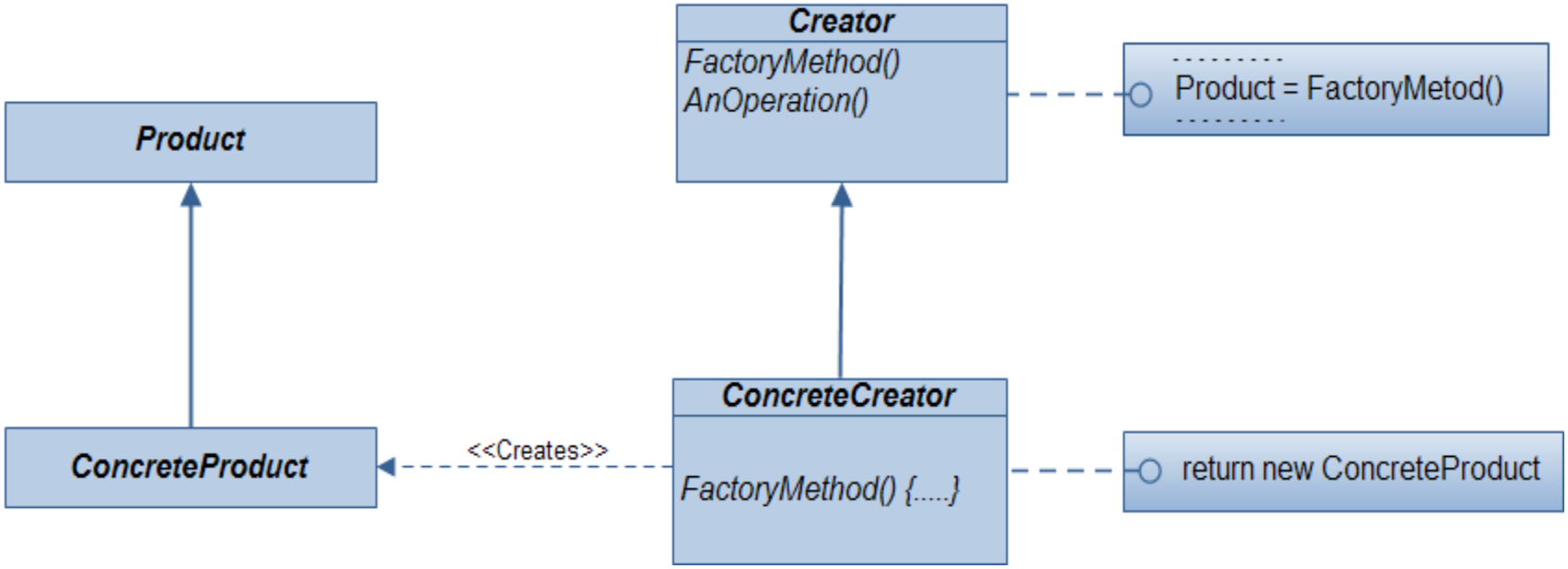
✓ Exemple de programme illustrant l'utilisation des classes collection et Iterator

```
import java.util.Iterator;

import java.util.List;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Set;
import java.util.HashSet;
import java.util.TreeSet;

public class IterTest {
   public static void main(String[] args) {
      List <Integer> lis1 = new ArrayList<Integer>();
      List <Integer> lis2 = new LinkedList<Integer>();
      Set <Integer> set1 = new HashSet<Integer>();
      Set <Integer> set2 = new TreeSet<Integer>();

      for (int i=0; i<6; i++) lis1.add(i);

      for (int i=0; i<6; i++) lis2.add(i);

      for (int i=0; i<6; i++) set1.add(i);

      for (int i=0; i<6; i++) set2.add(i);
```

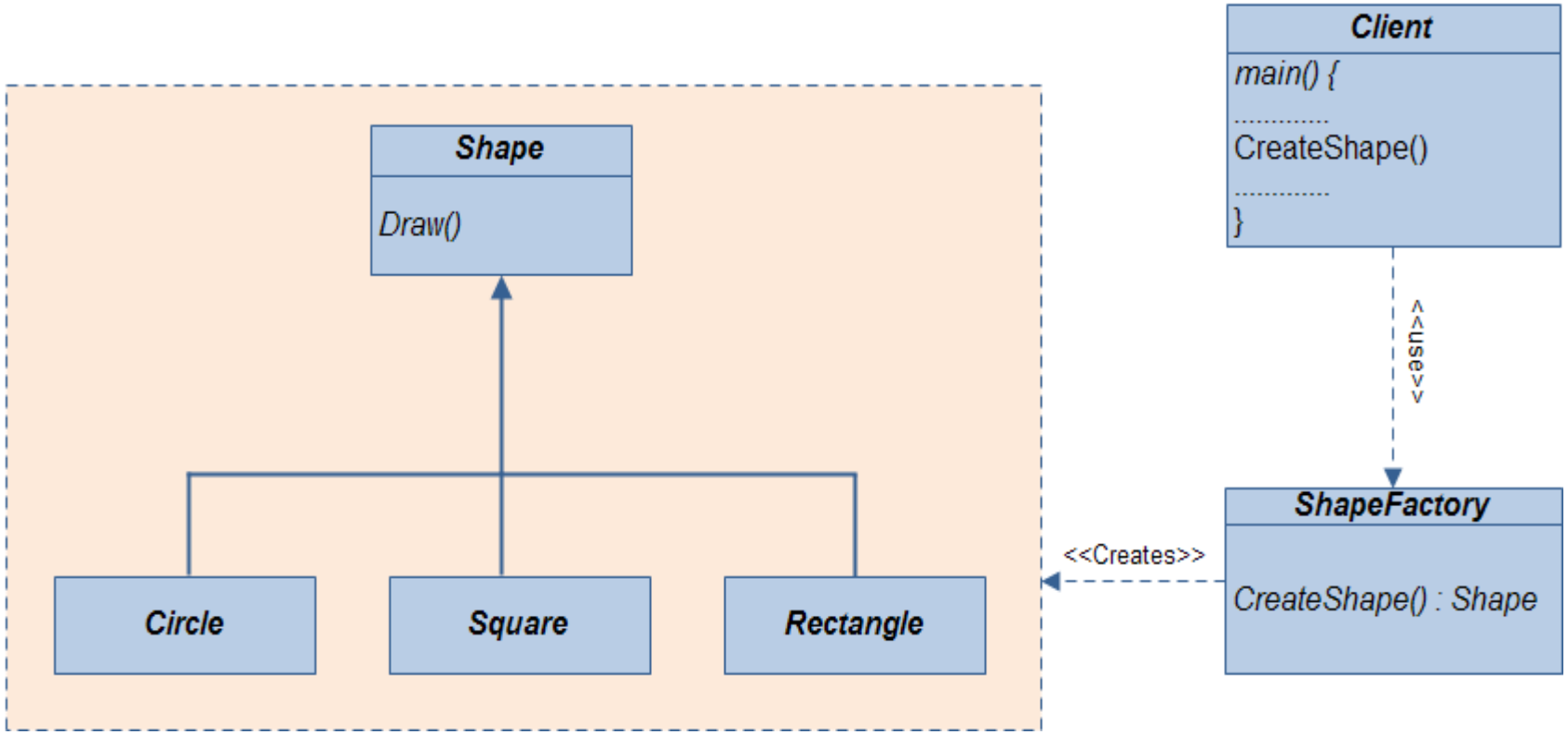
```
      Iterator <Integer> itl1 = lis1.iterator();
      Iterator <Integer> itl2 = lis2.iterator();
      Iterator <Integer> its1 = set1.iterator();
      Iterator <Integer> its2 = set2.iterator();

      System.out.println("ArrayList");
      while(itl1.hasNext()){
         System.out.print(itl1.next() );}

      System.out.println("\nLinkedList");
      while(itl2.hasNext()){
         System.out.print(itl2.next() );}

      System.out.println("\nHashSet");
      while(its1.hasNext()){
         System.out.print(its1.next() );}

      System.out.println("\nTreeSet");
      while(its2.hasNext()){
         System.out.print(its2.next() );}

      }
}
```

Factory method

- ✓ Factory Method : Méthode Fabrique (Usine)
- ✓ Définir une interface pour la création d'objet, mais laisser les sous-classes décider de l'objet a instancier
- ✓ La Factory Method délègue l'instanciation aux sous classes
- ✓ Séparer l'instanciation en l'encapsulant dans des classes dédiées



- ✓ Exemple de Fabrique simple : Fabrique de forme géométriques
- ✓ L'interface **Creator** fait partie du client (main)



Factory method

```
public interface Shape {
  void draw(); }
```

```
public class Circle implements Shape {
 @Override
  public void draw() {
    System.out.println(“ Draw Circle");
  }
}
```

```
public class Square implements Shape {
 @Override
  public void draw() {
    System.out.println(“ Draw Square");
  }
}
```

```
public class Rectangle implements Shape {
 @Override
  public void draw() {
    System.out.println(“ Draw Rectangle");
  }
}
```

```
public class Client {

  public static void main(String[] args) {
    ShapeFactory shapeFactory = new ShapeFactory();

    Shape shape1 = shapeFactory.CreateShape("Circle");
    shape1.draw();
    Shape shape2 = shapeFactory.CreateShape("Rectangle");
    shape2.draw();
    Shape shape3 = shapeFactory.CreateShape("Square");
    shape3.draw();
  }
}
```

```
public class ShapeFactory {
 public Shape CreateShape(String shapeType){
    if(shapeType == null) { return null; }

    if(shapeType.equals(“Circle")){
      return new Circle();
    } else if(shapeType.equals("Rectangle")){
      return new Rectangle();
    } else if(shapeType.equals("Square")){
      return new Square();
    }
    return null;
  }
}
```