Design Patterns
Part 9

Mohamed Youssfi

Laboratoire Signaux Systèmes Distribués et Intelligence Artificielle (SSDIA)

ENSET, Université Hassan II Casablanca, Maroc

Email : med@youssfi.net

Supports de cours : http://fr.slideshare.net/mohamedyoussfi9

Chaîne vidéo : http://youtube.com/mohamedYoussfi

Recherche : http://www.researchgate.net/profile/Youssfi_Mohamed/publications

Patterns de création d'objets :

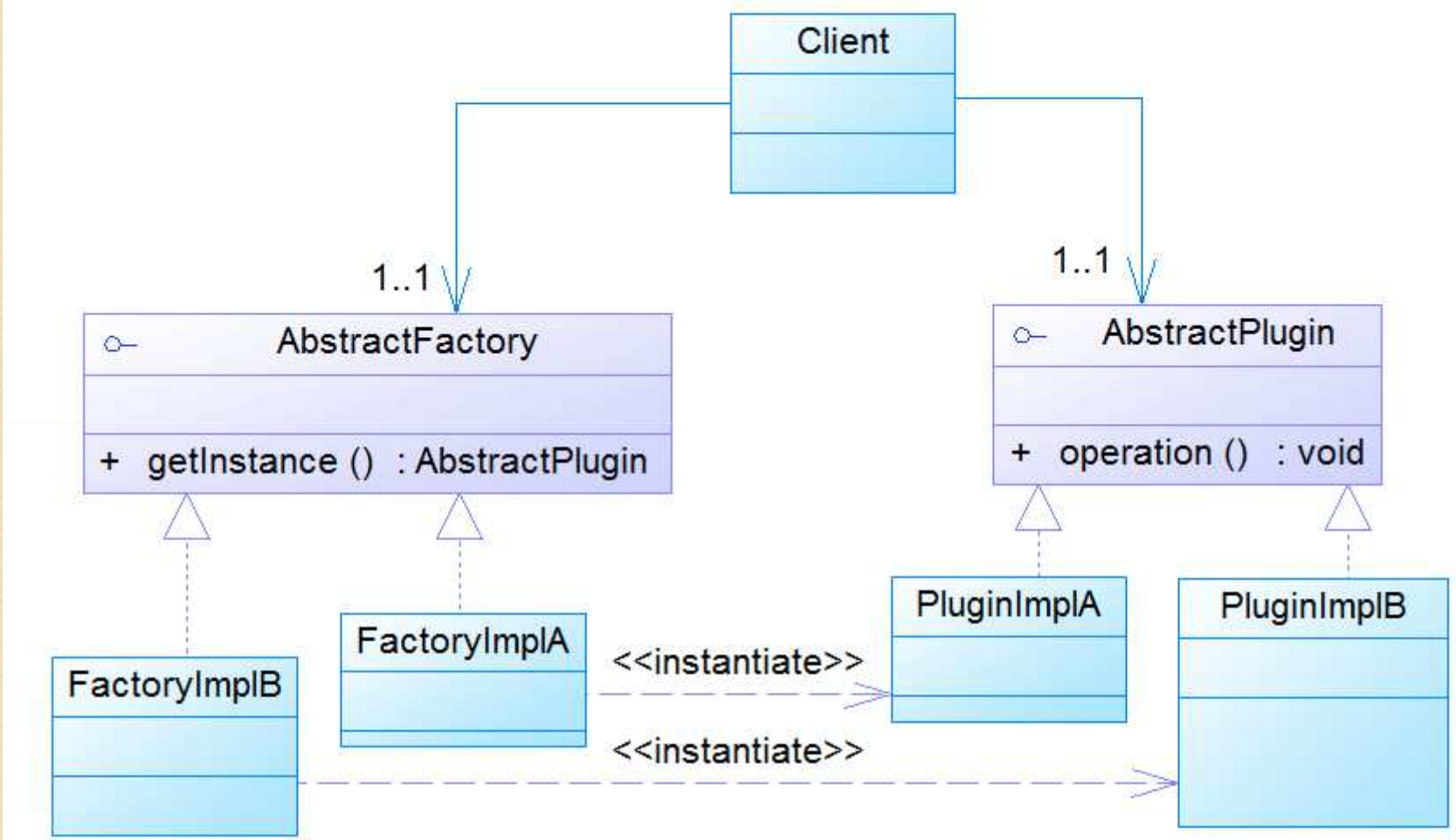
- AbstractFactory
- Singleton
- Prototype

Mohamed Youssfi
med@youssfi.net

Fabrique abstraite (Abstract Factory)

- **Catégorie :**
  - Création
- **OBJECTIFS :**
  - Fournir une interface pour créer des objets d'une même famille sans préciser leurs classes concrètes.
- **RESULTAT :**
  - Le Design Pattern permet d'isoler l'appartenance à une famille de classes.

Diagramme de classes



Abstract Factory

- **RESPONSABILITES :**
  - **AbstractFactory :** définit l'interface des méthodes de création.
  - **FactoryImplA et FactoryImplB :** implémentent l'interface et instancient la classe concrète appropriée.
  - **AbstractPlugin :** définit l'interface d'un type d'objet instancié.
  - **PluginA et PluginB :** sont des sous-classes concrètes. Elles sont instanciées respectivement par par les FactoryImplA et FactoryImplB
  - **La partie cliente** fait appel à une Fabrique pour obtenir une nouvelle instance d'AbstractPlugin**.** L'instanciation est transparente pour la partie cliente. Elle manipule une AbstractPlugin**.**

Implémentation

AbstractPlugin.java

```
package dp;
public interface AbstractPlugin {
  public void traitement();
}
```

AbstractFactory.java

```
package dp;
public interface AbstractFactory {
  public AbstractPlugin getInstence();
}
```

Implémentation

PluginImplA.java

```
package dp;
public class PluginImplA implements AbstractPlugin {
@Override
public void traitement() {
 System.out.println("Traitement par le plugin A .....");
}
}
```

PluginImplB.java

```
package dp;
public class PluginImplB implements AbstractPlugin {
@Override
public void traitement() {
 System.out.println("Traitement par le plugin B .....");
}
}
```

Implémentation

FactoryImplA.java

```
package dp;
public class FactoryImplA implements AbstractFactory {
@Override
public AbstractPlugin getInstence() {
 return new PluginImplA();
}
}
```

FactoryImplB.java

```
package dp;
public class FactoryImplB implements AbstractFactory {
@Override
public AbstractPlugin getInstence() {
 return new PluginImplB();
}
}
```

Implémentation

Application.java

```
package dp;
public class Application {
public static void main(String[] args) throws Exception{
 // Instanciation statique de la fabrique
 AbstractFactory factory=new FactoryImplA();
 AbstractPlugin plugin=factory.getInstence();
 plugin.traitement();
 // Instanciation dynamique de la fabrique
 factory =(AbstractFactory) Class.forName("dp.FactoryImplB").newInstance();
 plugin=factory.getInstence();
 plugin.traitement();
}
}
```

Traitement par le plugin A .....
Traitement par le plugin B .....

PATTERN SINGLETON

med@youssfi.net

Pattern Singleton

- Le pattern Singleton permet de garantir la création d'une instance unique d'une classe durant toute la durée d'exécution d'une application
- Le pattern Singleton fait partie des patterns Fabrique.
- Il très exploité dans les application qui s'exécutent dans un environnement multi-thread.

Pattern Singleton

- **Catégorie :**
  - Création
- **OBJECTIFS** :
  - Restreindre le nombre d' **instances** d'une classe à une et une seule.
  - Fournir une **méthode** pour accéder à cette instance unique.
- **RAISONS DE L'UTILISER** :
  - La classe ne doit avoir qu'une seule instance.
  - Cela peut être le cas d'une ressource système par exemple.
  - La classe empêche d'autres classes de l'instancier. Elle possède la seule instance d'elle-même et fournit la seule méthode permettant d'accéder à cette instance.
- **RESULTAT** :
  - Le Design Pattern permet d'isoler l'unicité d'une instance.
- **RESPONSABILITES** :
  - **Singleton** doit restreindre le nombre de ses propres instances à une et une seule. Son constructeur est privé : cela empêche les autres classes de l'instancier. La classe fournit la méthode statique **getInstance()** qui permet d'obtenir l'instance unique.

Diagramme de classes

Exemple d'implémentation

```
public class Singleton {
  private static final Singleton instance;
  private int compteur;
  static {  instance=new Singleton();  }
  private Singleton() { System.out.println("Instanciation"); }
  public static Singleton getInstance(){  return instance;  }
  public  void traiter(String tache){
    System.out.println("---------- Tâche "+tache+"----------");
     for(int i=1;i<=5;i++){
        ++compteur; System.out.print("."+compteur);
        try { Thread.sleep(1000); } catch (InterruptedException e) {
        e.printStackTrace();}
      }
     System.out.println(); System.out.println("Compteur="+compteur);
  }
}
```

Application

```
public class Application {
 public static void main(String[] args) {
   for(int i=1;i<=5;i++){
      Singleton singleton=Singleton.getInstance();
      singleton.traiter("T"+i);
   } }}
```

```
Instanciation
----------Tâche T1----------
.1.2.3.4.5
Compteur=5
----------Tâche T2----------
.6.7.8.9.10
Compteur=10
----------Tâche T3----------
.11.12.13.14.15
Compteur=15
----------Tâche T4----------
.16.17.18.19.20
Compteur=20
----------Tâche T5----------
.21.22.23.24.25
Compteur=25
```

Application multi threads et le singleton

- Dans une application multi thread, le fait que plusieurs threads utilisent un singleton, peut engendrer des problèmes de synchronisation.
- Pour éviter le problème, les méthodes du singleton peuvent être déclarée synchronized.
- Ceci entraine la création d'une file d'attente au niveau de l'accès à ces méthodes.
- Ce qui handicape les performances de l'application.

Exemple d'application multi threads

```
public class ThreadedTask extends Thread {
  private String taskName;
  public ThreadedTask(String taskName) {
       this.taskName = taskName;
  }
   @Override
   public void run() {
     Singleton singleton=Singleton.getInstance();
     singleton.traiter(taskName);
   }
}
```

Exemple d'application multi threads

```
public class ThreadedTask extends Thread {
  private String taskName;
  public ThreadedTask(String taskName) {
       this.taskName = taskName;
  }
   @Override
   public void run() {
     Singleton singleton=Singleton.getInstance();
     singleton.traiter(taskName);
   }
}
```

Exemple d'application multi threads

```
public class AppMultiThread {
  public static void main(String[] args) {
    for(int i=1;i<=5;i++){
     ThreadedTask t=new ThreadedTask("T"+i);
     t.start();
  }
}
}
```

Instanciation
---------- Tâche T1----------
.1---------- Tâche T3----------
---------- Tâche T5----------
.2.3---------- Tâche T4----------
.4---------- Tâche T2----------
.5.6.8.9.7.10.11.12.14.13.15.16.18.18.18.19.20.21.23.23.24
Compteur=24
Compteur=24

Compteur=24
Compteur=24

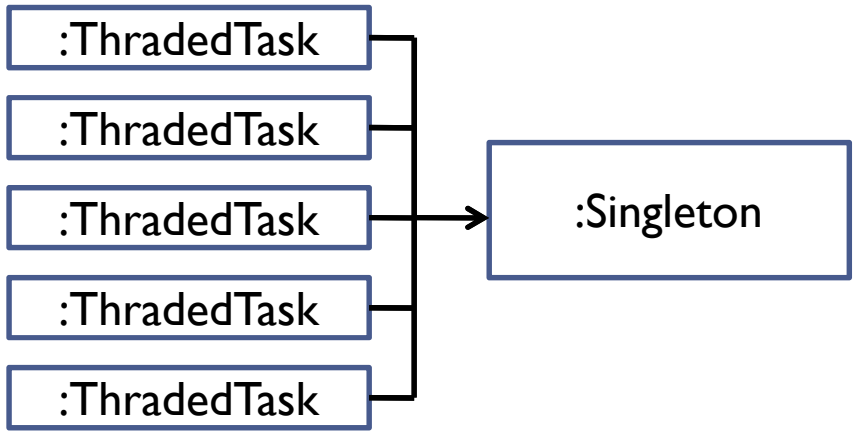
Compteur=24

?

Solution de synchronisation

- `public synchronized void traiter(String tache){`

```
Instanciation
---------- Tâche T1----------
.1.2.3.4.5
Compteur=5
---------- Tâche T4----------
.6.7.8.9.10
Compteur=10
---------- Tâche T5----------
.11.12.13.14.15
Compteur=15
---------- Tâche T3----------
.16.17.18.19.20
Compteur=20
---------- Tâche T2----------
.21.22.23.24.25
Compteur=25
```



PATTERN PROTOTYPE

med@youssfi.net

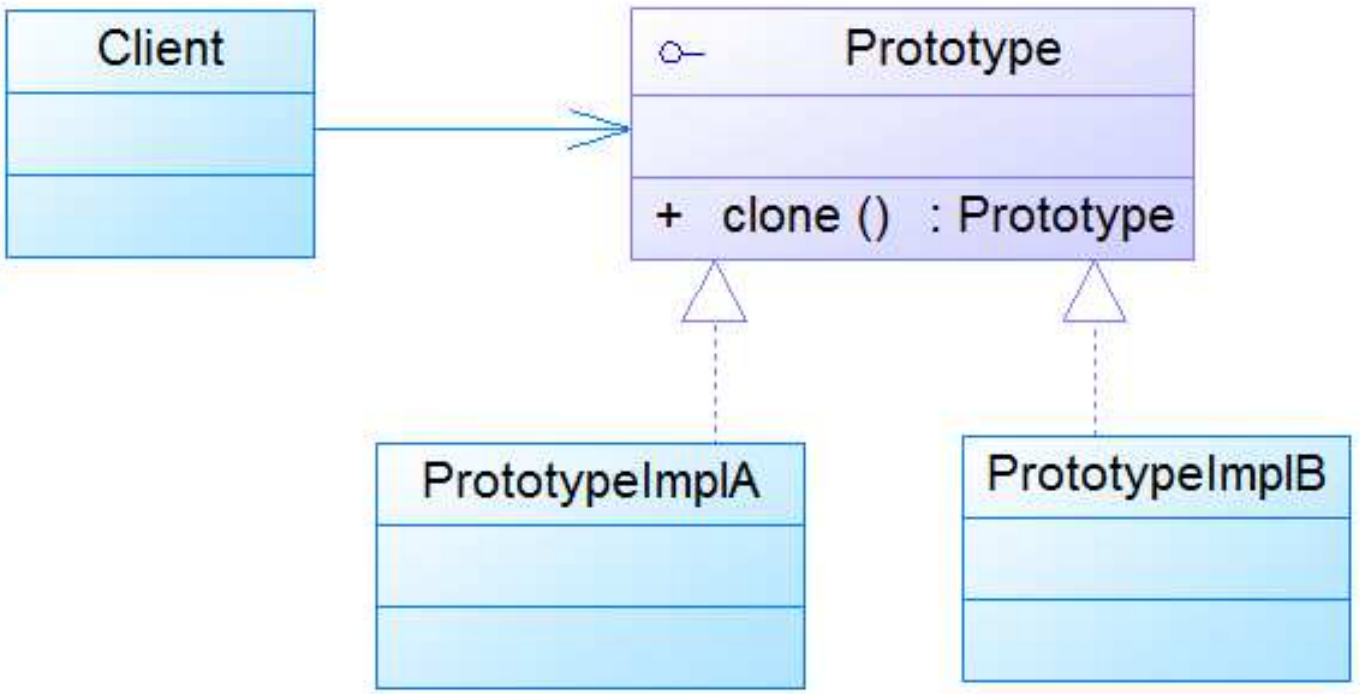
Pattern Prototype

- Catégorie :
  - Création
- Objectifs :
  - L'objectif du pattern PROTOTYPE est de fournir de nouveaux objets par la copie d'un exemple plutôt que de produire de nouvelles instances non initialisées d'une classe.
- Raisons d'utilisation
  - Le système doit créer de nouvelles instances, mais il ignore de quelle classe. Il dispose cependant d'instances de la classe désirée.
  - La duplication peut être également intéressante pour les performances (la duplication est plus rapide que l'instanciation).
- Résultat :
  - Le Design Pattern permet d'isoler l'appartenance à une classe.

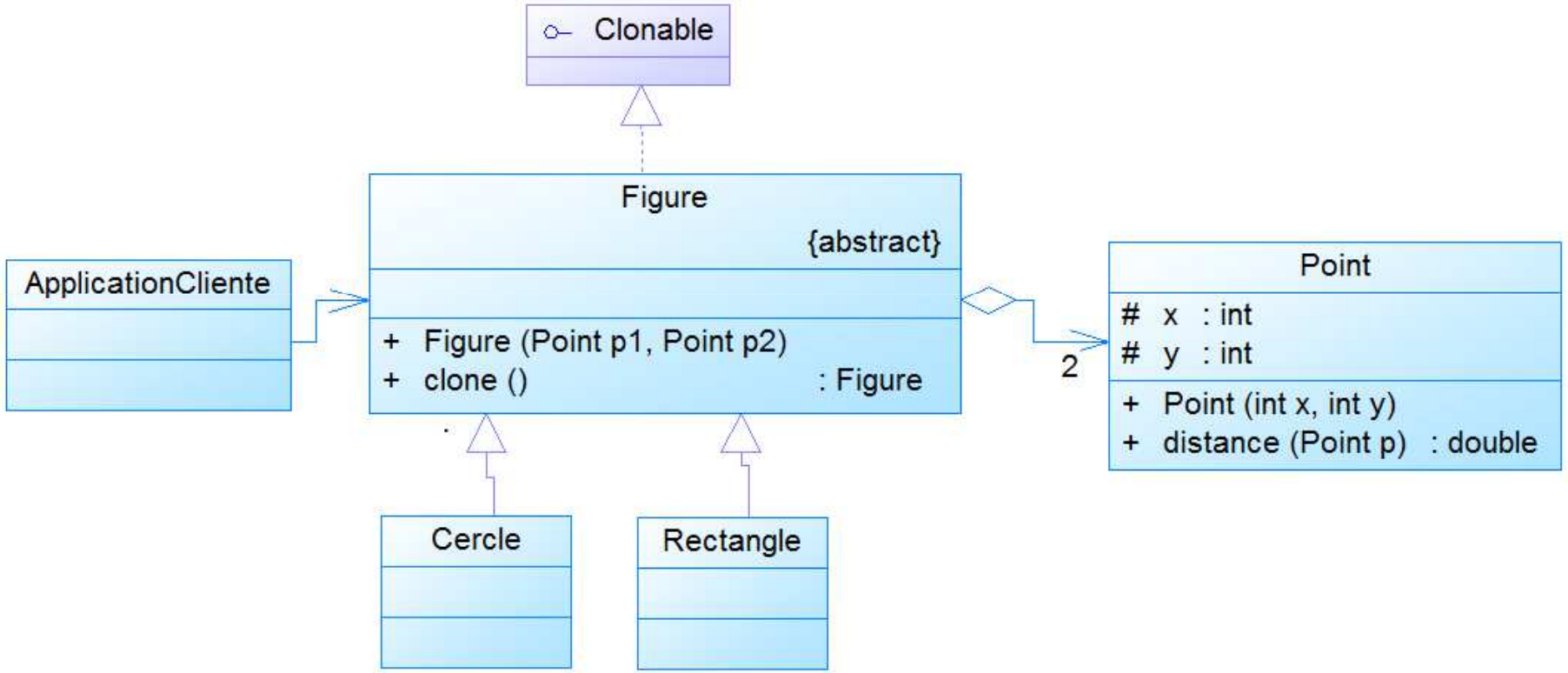
Prototype

- Responsabilités :
  - **Prototype** : définit l'interface de duplication de soi-même.
  - **PrototypeImplA** et **PrototypeImplA** : sont des sous-classes concrètes de Prototype. Elles implémentent l'interface de duplication.
  - **La partie cliente** appelle la méthode clone() de la classe Prototype. Cette méthode retourne un double de l'instance.

Diagramme de classes



Exemple d'implémentation



Pattern Prototype

Point.java

```
package dp;
public class Point {
 protected int x;  protected int y;
 public Point(int x, int y) { this.x = x; this.y = y;  }
 public double distance(Point p){
  int a=p.x-this.x;
  int b=p.y-this.y;
  return Math.sqrt(a*a+b*b);
 }
  @Override
 public String toString() {
  return "Point [x=" + x + ", y=" + y + "]";
 }
}
```

Pattern Prototype

Figure.java

```
package dp;
public abstract class Figure implements Cloneable {
    protected Point p1;
    protected Point p2;
public Figure(Point p1, Point p2) {
  this.p1 = p1; this.p2 = p2;
}
@Override
protected Figure clone() throws CloneNotSupportedException {
  return (Figure)super.clone();
}
public abstract double getSurface();
@Override
public String toString() {
  return "p1=" + p1 + ", p2=" + p2;
}
}
```

Pattern Prototype

Cercle.java

```
package dp;
public class Cercle extends Figure {
 public Cercle(Point p1, Point p2) {
  super(p1, p2);
 }
@Override
 public double getSurface() {
  double r=p1.distance(p2);
  return Math.PI*r*r;
}
@Override
public String toString() {
  return "Cercle [" + super.toString() + "]";
}
}
```

Pattern Prototype

Rectangle.java

```
package dp;
public class Rectangle extends Figure {
 public Rectangle(Point p1, Point p2) {
  super(p1, p2);
 }
 @Override
 public double getSurface() {
   int l=p1.x-p2.x;
   int h=p1.y-p2.y;
   return l*h;
}
@Override
public String toString() {
  return "Rectangle [" + super.toString() + "]";
}
}
```

Pattern Prototype

Application.java

```
package dp;
public class Application {
 public static void main(String[] args) throws CloneNotSupportedException {
   Figure f1=new Cercle(new Point(10, 10), new Point(20, 20));
   Figure f2=new Rectangle(new Point(30, 30), new Point(40, 40));
   System.out.println("Surface de "+f1+" est :"+f1.getSurface());
   System.out.println("Surface de "+f2+" est :"+f2.getSurface());
   System.out.println("----------------------------------------");
   Figure f3=f1.clone();
   System.out.println("Copie de f1 :");
   System.out.println("Surface de "+f3+" est :"+f3.getSurface());
   System.out.println("----------------------------------------");
   Figure f4=f2.clone();
   System.out.println("Copie de f2 :");
   System.out.println("Surface de "+f4+" est :"+f4.getSurface());
 }}
```

```
Surface de Cercle [p1=Point [x=10, y=10], p2=Point [x=20, y=20]] est :628.3185307179587
Surface de Rectangle [p1=Point [x=30, y=30], p2=Point [x=40, y=40]] est :100.0
----------------------------------------
Copie de f1 :
Surface de Cercle [p1=Point [x=10, y=10], p2=Point [x=20, y=20]] est :628.3185307179587
----------------------------------------
Copie de f2 :
Surface de Rectangle [p1=Point [x=30, y=30], p2=Point [x=40, y=40]] est :100.0
```

med@youssfi.net