



# TP SOLID

**KBOUBI OUMAIMA**

**2021-2022**

## SOLID

Théorisé en 2002 par Robert C. Martin dans son ouvrage *Agile Software Development, Principles, Patterns and Practices*, l'acronyme SOLID est un moyen mnémotechnique pour retenir **5 grands principes** applicables au développement d'applications logicielles pour les rendre plus faciles à comprendre, à maintenir et à faire évoluer.

Ces 5 principes sont :

- **S** comme **Single Responsibility** Principle
- **O** comme **Open/Closed** Principle
- **L** comme **Liskov Substitution** Principle
- **I** comme **Interface Segregation** Principle
- **D** comme **Dependency Inversion** Principle.



# SRP

# SINGLE RESPONSIBILITY PRINCIPLE

Le principe SRP pour "Single Responsibility Principle" énonce qu'**une classe ne devrait avoir qu'une et une seule raison de changer.** L'idée ici est de faire en sorte qu'une classe ne soit responsable que d'une seule fonction de votre application, et que cette responsabilité soit complètement encapsulée dans la classe. L'objectif du principe SRP est de réduire la complexité de votre projet.

## Car.java

```
SRP > src > com > directi > training > srp > exercise > Car.java
1  package com.directi.training.srp.exercise;
2
3  public class Car
4  {
5      private final String _id;
6      private final String _model;
7      private final String _brand;
8
9      public Car(String id, String model, String brand)
10     {
11         _id = id;
12         _model = model;
13         _brand = brand;
14     }
15
16     public String getId()
17     {
18         return _id;
19     }
20
21     public String getModel()
22     {
23         return _model;
24     }
25
26     public String getBrand()
27     {
28         return _brand;
29     }
30 }
```

## CarManager.java

```
SRP > src > com > directi > training > srp > exercise > CarManager.java > {} com.directi.training.srp.exercise
```

```
1 package com.directi.training.srp.exercise;
2
3 import java.util.Arrays;
4 import java.util.List;
5
6 public class CarManager
7 {
8     private List<Car> _carsDb = Arrays
9         .asList(new Car("1", "Golf III", "Volkswagen"), new Car("2", "Multipla", "Fiat"),
10             new Car("3", "Megane", "Renault"));
11
12     public Car getFromDb(final String carId)
13     {
14         for (Car car : _carsDb) {
15             if (car.getId().equals(carId)) {
16                 return car;
17             }
18         }
19         return null;
20     }
21
22     public String getCarsNames()
23     {
24         StringBuilder sb = new StringBuilder();
25         for (Car car : _carsDb) {
26             sb.append(car.getBrand());
27             sb.append(" ");
28             sb.append(car.getModel());
29             sb.append(", ");
30         }
31         return sb.substring(0, sb.length() - 2);
32     }
33
34     public Car getBestCar()
35     {
36         Car bestCar = null;
37         for (Car car : _carsDb) {
38             if (bestCar == null || car.getModel().compareTo(bestCar.getModel()) > 0) {
39                 bestCar = car;
40             }
41         }
42         return bestCar;
43     }
44 }
```

## Car.java

```
SRP > src > com > directi > training > srp > exercise_refactored > Car.java > {  
1  package com.directi.training.srp.exercise_refactored;  
2  
3  public class Car  
4  {  
5      private final String _id;  
6      private final String _model;  
7      private final String _brand;  
8  
9      public Car(String id, String model, String brand)  
10     {  
11         _id = id;  
12         _model = model;  
13         _brand = brand;  
14     }  
15  
16     public String getId()  
17     {  
18         return _id;  
19     }  
20  
21     public String getModel()  
22     {  
23         return _model;  
24     }  
25  
26     public String getBrand()  
27     {  
28         return _brand;  
29     }  
30 }
```

## CarDao.java

```
SRP > src > com > directi > training > srp > exercise_refactored > CarDao.java > {} com.directi.training.srp.exercise_refactored

1  package com.directi.training.srp.exercise_refactored;
2
3  import java.util.ArrayList;
4  import java.util.Arrays;
5  import java.util.List;
6
7  public class CarDao
8  {
9      private List<Car> _carsDb = Arrays
10         .asList(new Car("1", "Golf III", "Volkswagen"), new Car("2", "Multipla", "Fiat"),
11             new Car("3", "Megane", "Renault"));
12
13     public Car findById(String carId)
14     {
15         for (Car car : _carsDb) {
16             if (car.getId().equals(carId)) {
17                 return car;
18             }
19         }
20         return null;
21     }
22
23     public List<Car> findAll()
24     {
25         return new ArrayList<>(_carsDb);
26     }
27 }
28
29
```

## CarFormatter.java

```
SRP > src > com > directi > training > srp > exercise_refactored > CarFormatter.java > ...
1  package com.directi.training.srp.exercise_refactored;
2
3  import java.util.List;
4
5  public class CarFormatter
6  {
7      public String getCarsNames(List<Car> cars)
8      {
9          StringBuilder sb = new StringBuilder();
10         for (Car car : cars) {
11             sb.append(car.getBrand());
12             sb.append(" ");
13             sb.append(car.getModel());
14             sb.append(", ");
15         }
16         return sb.substring(0, sb.length() - 2);
17     }
18 }
19
```

## CarRater.java

```
SRP > src > com > directi > training > srp > exercise_refactored > CarRater.java > {} com.directi.training.srp.exercise_refactored
1  package com.directi.training.srp.exercise_refactored;
2
3  import java.util.List;
4
5  public class CarRater
6  {
7      public Car getBestCar(List<Car> cars)
8      {
9          Car bestCar = null;
10         for (Car car : cars) {
11             if (bestCar == null || car.getModel().compareTo(bestCar.getModel()) > 0) {
12                 bestCar = car;
13             }
14         }
15         return bestCar;
16     }
17 }
```



## CarManager.java

```
SRP > src > com > directi > training > srp > exercise_refactored > CarManager.java > {} com.directi.training.srp.exerc
1 package com.directi.training.srp.exercise_refactored;
2
3 public class CarManager
4 {
5     private final CarDao _carDao;
6     private final CarFormatter _carFormatter;
7     private final CarRater _carRater;
8
9     public CarManager(CarDao carDao, CarFormatter carFormatter, CarRater carRater)
10    {
11        _carDao = carDao;
12        _carFormatter = carFormatter;
13        _carRater = carRater;
14    }
15
16    public Car getCarById(final String carId)
17    {
18        return _carDao.findById(carId);
19    }
20
21    public String getCarsNames()
22    {
23        return _carFormatter.getCarsNames(_carDao.findAll());
24    }
25
26    public Car getBestCar()
27    {
28        return _carRater.getBestCar(_carDao.findAll());
29    }
30 }
31
```

**Avant la modification** du code la classe CarManager s'occupe de la récupération des noms des voitures et de la recherche de la meilleure voiture ainsi que la récupération d'une des voitures de la base de données par son id.  
=> la classe CarManager a plusieurs responsabilités!

**Après la modification**, on a dévisé cette classe en plusieurs classes chacune à responsabilité limitée comme suit:

- La classe CarDao s'occupe de la récupération des données de la base de données.
- La classe CarFormatter s'occupe de la transformation de l'objet Car en une chaîne contenant les caractéristiques de la voiture concernée.
- La classe CarRater s'occupe de la récupération de la meilleure voiture.

**Après cette modification chaque classe a une seule responsabilité, le code est plus facile à lire, à maintenir et à tester.**

# OCP OPEN/CLOSED PRINCIPLE

Le second principe SOLID est l'"Open/Closed Principle" (OCP) : les classes d'un projet devraient être **ouvertes à l'extension, mais fermées à la modification**.

L'idée de fond derrière ce principe est la suivante : ajouter de nouvelles fonctionnalités ne devrait pas casser les fonctionnalités déjà existantes.

Changer le comportement d'une classe existante pour l'adapter à un nouveau besoin il vaut mieux étendre cette classe et en adapter son comportement : elle est donc ouverte à l'extension et fermée à la modification.

L'intérêt de faire cela, c'est d'éviter de casser ou d'introduire des bugs dans une application qui fonctionne correctement.

## ResourceAllocator.java

```
OCP > src > com > directi > training > ocp > exercise > ResourceAllocator.java > ResourceAllocator > allocat
1 package com.directi.training.ocp.exercise; ResourceAllocator.java is not on the
2 public class ResourceAllocator
3 {
4     private static final int INVALID_RESOURCE_ID = -1;
5     public int allocate(ResourceType resourceType)
6     {
7         int resourceId;
8         switch (resourceType) {
9             case TIME_SLOT:
10                resourceId = findFreeTimeSlot();
11                markTimeSlotBusy(resourceId);
12                break;
13             case SPACE_SLOT:
14                resourceId = findFreeSpaceSlot();
15                markSpaceSlotBusy(resourceId);
16                break;
17             default:
18                System.out.println("ERROR: Attempted to allocate invalid resource");
19                resourceId = INVALID_RESOURCE_ID;
20                break;
21        }
22        return resourceId;
23    }
24    public void free(ResourceType resourceType, int resourceId)
25    {
26        switch (resourceType) {
27            case TIME_SLOT:
28                markTimeSlotFree(resourceId);
29                break;
30            case SPACE_SLOT:
31                markSpaceSlotFree(resourceId);
32                break;
33            default:
34                System.out.println("ERROR: attempted to free invalid resource");
35                break;
36        }
37    }
38    private void markSpaceSlotFree(int resourceId){}
39    private void markTimeSlotFree(int resourceId){}
40    private void markSpaceSlotBusy(int resourceId){}
41    private int findFreeSpaceSlot()
42    {return 0;}
43    private void markTimeSlotBusy(int resourceId){}
44    private int findFreeTimeSlot()
45    {return 0;}
46 }
47
```

## ResourceType.java

```
OCP > src > com > directi > training > ocp > exercise > ☕ ResourceType.java
1  package com.directi.training.ocp.exercise;
2
3  /**
4   * Created by IntelliJ IDEA.
5   * User: goyalamit
6   * Date: Jul 11, 2011
7   * Time: 1:17:04 PM
8   * To change this template use File | Settings | File Templates.
9   */
10 public enum ResourceType
11 {
12     TIME_SLOT,
13     SPACE_SLOT
14 }
15 |
```


## Resource.java

```
OCP > src > com > directi > training > ocp > exercise_refactored > Resource.java
1  package com.directi.training.ocp.exercise_refactored;
2
3  public interface Resource
4  {
5      int findFree();
6
7      void markBusy(int resourceId);
8
9      void markFree(int resourceId);
10 }
11
```


## ResourceAllocator.java

```
OCP > src > com > directi > training > ocp > exercise_refactored > ResourceAllocator.java
1  package com.directi.training.ocp.exercise_refactored;
2
3  public class ResourceAllocator
4  {
5      public int allocate(Resource resource)
6      {
7          int resourceId = resource.findFree();
8          resource.markBusy(resourceId);
9          return resourceId;
10     }
11
12     public void free(Resource resource, int resourceId)
13     {
14         resource.markFree(resourceId);
15     }
16 }
17
```

## SpaceResource.java

```
OCP > src > com > directi > training > ocp > exercise_refactored >  SpaceResource.java
1  package com.directi.training.ocp.exercise_refactored;
2
3  public class SpaceResource implements Resource
4  {
5      @Override
6      public int findFree()
7      {return 0;}
8      @Override
9      public void markBusy(int resourceId){}
10     @Override
11     public void markFree(int resourceId){}
12 }
13
```

## TimeResource.java

```
OCP > src > com > directi > training > ocp > exercise_refactored >  TimeResource.java
1  package com.directi.training.ocp.exercise_refactored;
2
3  public class TimeResource implements Resource
4  {
5      @Override
6      public int findFree()
7      {return 0;}
8      @Override
9      public void markBusy(int resourceId){}
10     @Override
11     public void markFree(int resourceId){}
12 }
13
```

**Avant la modification** du code la classe ResourceType contient une énumération des types de ressource ("TIME\_SLOT" ou bien "SPACE\_SLOT"), La classe RessourceAllocator s'occupe des traitements de toutes les ressources.

=> Lors de l'ajout d'un nouveau type, il faut modifier la classe de RessourceAllocator se qui peut engendrer des problèmes dans l'application!

**Après la modification**, on a dévisé cette classe comme suit:

- L'interface Resource contient les méthodes génériques qui vont être utilisées par les différents types de ressources.
- La classe SpaceResource et TimeResource implémente l'interface générique Resource, donc par défaut elle doit redéfinir les méthodes de l'interface selon la spécificité de son type "Space" ou "Time".
- La classe ResourceAllocator utilise la méthode de type spécifique passé en paramètre comme Resource de façon générique (surclassement)

**=> De cette façon, pour ajouter un nouveau type de ressource il suffit d'implémenter l'interface Resource et redefinir les méthodes nécessaires selon la spécifité du nouveau type et puis passer l'instance de la nouvelle classe en paramètre au RessourceAllocator pour exécuter ses méthodes spécifiques.**



# LSP

# LISKOV SUBSTITUTION PRINCIPLE

Le troisième principe SOLID met en valeur non pas une, mais deux brillantes développeuses. En effet, le principe de substitution a été formulé pour la première fois par Barbara Liskov et Jeannette Wing dans un article intitulé Family Values : A Behavioral Notion of Subtyping [EN] en 1994.

Il complète le second principe : il doit être possible de substituer une classe "parente" par l'une de ses classes enfants. Pour cela, nous devons garantir que les classes enfants auront le même comportement que la classe qu'elles étendent.


Une classe devrait implémenter une interface, notamment si elle a pour objectif d'être étendue.

Si nous lançons des exceptions, il vaut mieux avoir une classe d'exception par type d'erreur, puis étendre celle-ci pour chaque cas d'erreur. Nous nous assurons dans ce cas que le code qui dépend de notre implémentation sera en capacité de gérer proprement les cas d'erreur.

Pour résumer:

- Le principe de substitution de Liskov reprend le principe Open/Closed et l'applique au cas particulier de l'héritage de classes : si une classe enfant est une implémentation valide, alors une classe parent doit également l'être (et vice versa).
- D'un point de vue fonctionnel, cela revient à formaliser un contrat sur nos objets au sujet de leur implémentation, c'est à dire le comportement de leurs fonctions.



## Duck.java

```
LSP > src > com > directi > training > lsp > exercise >  Duck.java >  
1  package com.directi.training.lsp.exercise;  
2  
3  public class Duck  
4  {  
5      public void quack()  
6      {  
7          System.out.println("Quack...");  
8      }  
9  
10     public void swim()  
11     {  
12         System.out.println("Swim...");  
13     }  
14 }  
15
```

## Pool.java

```
LSP > src > com > directi > training > lsp > exercise > Pool.java > {} com.d
1 package com.directi.training.lsp.exercise; Pool.java
2
3 public class Pool
4 {
5     public void run()
6     {
7         Duck donaldDuck = new Duck();
8         Duck electricDuck = new ElectronicDuck();
9         quack(donaldDuck, electricDuck);
10        swim(donaldDuck, electricDuck);
11    }
12
13    private void quack(Duck... ducks)
14    {
15        for (Duck duck : ducks) {
16            duck.quack();
17        }
18    }
19
20    private void swim(Duck... ducks)
21    {
22        for (Duck duck : ducks) {
23            duck.swim();
24        }
25    }
26
27    Run | Debug
28    public static void main(String[] args)
29    {
30        Pool pool = new Pool();
31        pool.run();
32    }
33
```

## ElectronicDuck.java

```
LSP > src > com > directi > training > lsp > exercise >  ElectronicDuck.java > {} com.directi
1  package com.directi.training.lsp.exercise;      ElectronicDuck.java
2  
3  public class ElectronicDuck extends Duck
4  {
5      private boolean _on = false;
6
7      @Override
8      public void quack()
9      {
10         if (_on) {
11             System.out.println("Electronic duck quack...");
12         } else {
13             throw new RuntimeException("Can't quack when off");
14         }
15     }
16
17     @Override
18     public void swim()
19     {
20         if (_on) {
21             System.out.println("Electronic duck swim");
22         } else {
23             throw new RuntimeException("Can't swim when off");
24         }
25     }
26
27     public void turnOn()
28     {
29         _on = true;
30     }
31
32     public void turnOff()
33     {
34         _on = false;
35     }
36 }
```

## Duck.java

```
LSP > src > com > directi > training > lsp > exercise_refactored > Duck.java > ..
1  package com.directi.training.lsp.exercise_refactored;
2
3  public class Duck implements IDuck
4  {
5      @Override
6      public void quack()
7      {
8          System.out.println("Quack.....");
9      }
10
11     @Override
12     public void swim()
13     {
14         System.out.println("Swim....");
15     }
16
17 }
18
```

## IDuck.java

```
LSP > src > com > directi > training > lsp > exercise_refactored > IDuck.java > ..
1  package com.directi.training.lsp.exercise_refactored;
2
3  public interface IDuck
4  {
5      void quack() throws IDuckException;
6
7      void swim() throws IDuckException;
8
9      class IDuckException extends Exception
10     {
11         public IDuckException(String message)
12         {
13             super(message);
14         }
15     }
16 }
```

## ElectronicDuck.java

```
LSP > src > com > directi > training > lsp > exercise_refactored > ElectronicDuck.java > {} com
1  package com.directi.training.lsp.exercise_refactored;
2
3  public class ElectronicDuck implements IDuck
4  {
5      private boolean _on = false;
6
7      @Override
8      public void quack() throws DuckIsOffException
9      {
10         if (_on) {
11             System.out.println("Electronic duck quack...");
12         } else {
13             throw new DuckIsOffException("Can't quack when off");
14         }
15     }
16
17     @Override
18     public void swim() throws DuckIsOffException
19     {
20         if (_on) {
21             System.out.println("Electronic duck swim...");
22         } else {
23             throw new DuckIsOffException("Cant swim when off");
24         }
25     }
26
27     public void turnOn()
28     {
29         this._on = true;
30     }
31
32     public void turnOff()
33     {
34         this._on = false;
35     }
36
37
38     public static class DuckIsOffException extends IDuckException
39     {
40         public DuckIsOffException(String message)
41         {
42             super(message);
43         }
44     }
45
46 }
```

## Pool.java

```
LSP > src > com > directi > training > lsp > exercise_refactored > Pool.java > {} com.dir
1  package com.directi.training.lsp.exercise_refactored;    Pool.
2
3  public class Pool
4  {
5      public void run()
6      {
7          Duck donaldDuck = new Duck();
8          ElectronicDuck electronicDuck = new ElectronicDuck();
9          quack(donaldDuck, electronicDuck);
10         swim(donaldDuck, electronicDuck);
11     }
12
13     private void quack(IDuck... ducks)
14     {
15         for (IDuck duck : ducks) {
16             try {
17                 duck.quack();
18             } catch (IDuck.IDuckException e) {
19                 e.printStackTrace();
20             }
21         }
22     }
23
24     private void swim(IDuck... ducks)
25     {
26         for (IDuck duck : ducks) {
27             try {
28                 duck.swim();
29             } catch (IDuck.IDuckException e) {
30                 e.printStackTrace();
31             }
32         }
33     }
34
35     Run | Debug
36     public static void main(String[] args)
37     {
38         Pool pool = new Pool();
39         pool.run();
40     }
```

**Avant la modification** du code la classe Duck précise deux méthodes qui sont **quack** et **swim**. La classe ElectronicDuck hérite son comportement de la classe Duck donc implécitement les méthodes quack et swim mais en ajoutant une exception selon le fonctionnement particulier de cette classe.

=>On ne peut pas remplacer la classe parente Duck par la classe fille ElectronicDuck car cette dernière lance une exception et sa classe mère ne lance aucune exception lors de son fonctionnement!

**Après la modification**, on a procédé comme suit:

- L'interface **IDuck** contient le type générique ayant les méthodes quack et swim ainsi que la classe d'exception spécifique à ce type **IDuckException** qui hérite de la classe générique des exceptions **Exception**.



- Les deux classes **ElectronicDuck** et **Duck** implémentent l'interface **IDuck**, la classe **ElectronicDuck** spécifie une classe d'exception lorsque le duck n'est pas en marche

**DuckIsOffException** qui hérite de la classe d'exception **IDuckException** précisée dans l'interface **IDuck**.

- Les méthodes de la classe Pool reçoit comme paramètre l'interface IDuck, alors on peut passer n'importe quel type de classe implémentant cette interface et utilisé sa classe d'exception spécifique

=> **De cette façon, On peut remplacer la classe parente par sa classe enfant.**



# ISP

# INTERFACE SEGREGATION PRINCIPLE

Le quatrième principe SOLID commence donc par la lettre I pour "Interface Segregation Principle". L'idée ici est d'éviter d'avoir des interfaces aux multiples responsabilités et de les redécouper en multiples interfaces qui ont elles **une seule responsabilité qui peut se traduire par l'implémentation de plusieurs méthodes.** ce principe revisite le premier principe (Single Responsibility Principle) en l'appliquant cette fois à nos interfaces.

## Door.java

```
ISP > src > com > directi > training > isp > exercise > ☕ Door.java
1  package com.directi.training.isp.exercise;
2
3  public interface Door
4  {
5      void lock();
6
7      void unlock();
8
9      void open();
10
11     void close();
12
13     void timeOutCallback();
14
15     void proximityCallback();
16 }
17
```

## SensingDoor.java

```
ISP > src > com > directi > training > isp > exercise > SensingDoor.java > SensingDoor > SensingDoor.java is not o
1 package com.directi.training.isp.exercise;
2
3 import sun.reflect.generics.reflectiveObjects.NotImplementedException;
4
5 public class SensingDoor implements Door
6 {
7     private boolean _locked;
8     private boolean _opened;
9
10    public SensingDoor(Sensor sensor)
11    {
12        sensor.register(this);
13    }
14
15    @Override
16    public void lock()
17    {
18        _locked = true;
19    }
20
21    @Override
22    public void unlock()
23    {
24        _locked = false;
25    }
26
27    @Override
28    public void open()
29    {
30        if (!_locked) {
31            _opened = true;
32        }
33    }
34
35    @Override
36    public void close()
37    {
38        _opened = false;
39    }
40
41    @Override
42    public void timeOutCallback()
43    {
44        throw new NotImplementedException();
45    }
46
47    @Override
48    public void proximityCallback()
49    {
50        _opened = true;
51    }
52 }
53
```


## Sensor.java

```
ISP > src > com > directi > training > isp > exercise > ☒ Sensor.java :
1  package com.directi.training.isp.exercise;
2
3  import java.util.Random;
4
5  public class Sensor
6  {
7      public void register(Door door)
8      {
9          while (true) {
10             if (isPersonClose()) {
11                 door.proximityCallback();
12                 break;
13             }
14         }
15     }
16
17     private boolean isPersonClose()
18     {
19         return new Random().nextBoolean();
20     }
21 }
22
```

## TimedDoor.java

```
ISP > src > com > directi > training > isp > exercise > TimedDoor.java > {} com.directi.training.isp.ex
1 package com.directi.training.isp.exercise; TimedDoor.java is not on t
2
3 import sun.reflect.generics.reflectiveObjects.NotImplementedException;
4
5 public class TimedDoor implements Door
6 {
7     private static final int TIME_OUT = 100;
8     private boolean _locked;
9     private boolean _opened;
10
11     public TimedDoor(Timer timer)
12     {
13         timer.register(TIME_OUT, this);
14     }
15
16     @Override
17     public void lock()
18     {
19         _locked = true;
20     }
21
22     @Override
23     public void unlock()
24     {
25         _locked = false;
26     }
27
28     @Override
29     public void open()
30     {
31         if (!_locked) {
32             _opened = true;
33         }
34     }
35
36     @Override
37     public void close()
38     {
39         _opened = false;
40     }
41
42     @Override
43     public void timeOutCallback()
44     {
45         _locked = true;
46     }
47
48     @Override
49     public void proximityCallback()
50     {
51         throw new NotImplementedException();
52     }
53 }
54
```

## Timer.java

```
ISP > src > com > directi > training > isp > exercise >  Timer.java > {} com.directi.trainin
1  package com.directi.training.isp.exercise;    Timer.java is no
2  
3  import java.util.TimerTask;
4
5  public class Timer
6  {
7      public void register(long timeOut, final Door door)
8      {
9          java.util.Timer timerUtility = new java.util.Timer();
10         timerUtility.schedule(new TimerTask()
11         {
12             @Override
13             public void run()
14             {
15                 door.timeOutCallback();
16             }
17         }, timeOut);
18     }
19 }
20
```

## Door.java

```
ISP > src > com > directi > training > isp > exercise_refactored > Door.java > ..
1  package com.directi.training.isp.exercise_refactored;
2
3  public interface Door
4  {
5      void lock();
6
7      void unlock();
8
9      void open();
10
11     void close();
12 }
13
```

## TimerClient.java


```
ISP > src > com > directi > training > isp > exercise_refactored > TimerClient.java > ..
1  package com.directi.training.isp.exercise_refactored;
2
3  public interface TimerClient
4  {
5      void timeOutCallback();
6  }
7
```




## SensingDoor.java

```
ISP > src > com > directi > training > isp > exercise_refactored > SensingDoor.java
1  package com.directi.training.isp.exercise_refactored;
2
3  public class SensingDoor implements Door, SensorClient
4  {
5      private boolean _locked;
6      private boolean _opened;
7
8      public SensingDoor(Sensor sensor)
9      {
10         sensor.register(this);
11     }
12
13     @Override
14     public void lock()
15     {
16         _locked = true;
17     }
18
19     @Override
20     public void unlock()
21     {
22         _locked = false;
23     }
24
25     @Override
26     public void open()
27     {
28         if (!_locked) {
29             _opened = true;
30         }
31     }
32
33     @Override
34     public void close()
35     {
36         _opened = false;
37     }
38
39     @Override
40     public void proximityCallback()
41     {
42         _opened = true;
43     }
44 }
```

## Sensor.java

```
ISP > src > com > directi > training > isp > exercise_refactored >  Sensor.java > ..
1  package com.directi.training.isp.exercise_refactored;
2
3  import java.util.Random;
4
5  public class Sensor
6  {
7      public void register(SensorClient sensorClient)
8      {
9          while (true) {
10             if (isPersonClose()) {
11                 sensorClient.proximityCallback();
12                 break;
13             }
14         }
15     }
16
17     private boolean isPersonClose()
18     {
19         return new Random().nextBoolean();
20     }
21 }
22
```

## SensorClient.java

```
ISP > src > com > directi > training > isp > exercise_refactored >  SensorClient
1  package com.directi.training.isp.exercise_refactored;
2
3  public interface SensorClient
4  {
5      void proximityCallback();
6  }
```

## TimedDoor.java

```
ISP > src > com > directi > training > isp > exercise_refactored > TimedDoor.java
1 package com.directi.training.isp.exercise_refactored;
2
3 public class TimedDoor implements Door, TimerClient
4 {
5     private static final int TIME_OUT = 100;
6     private boolean _locked;
7     private boolean _opened;
8
9     public TimedDoor(Timer timer)
10    {
11        timer.register(TIME_OUT, this);
12    }
13
14    @Override
15    public void lock()
16    {
17        _locked = true;
18    }
19
20    @Override
21    public void unlock()
22    {
23        _locked = false;
24    }
25
26    @Override
27    public void open()
28    {
29        if (!_locked) {
30            _opened = true;
31        }
32    }
33
34    @Override
35    public void close()
36    {
37        _opened = false;
38    }
39
40    @Override
41    public void timeOutCallback()
42    {
43        _locked = true;
44    }
45 }
46
```

## TimedDoor.java

```
ISP > src > com > directi > training > isp > exercise_refactored > TimedDoor.java
1 package com.directi.training.isp.exercise_refactored;
2
3 public class TimedDoor implements Door, TimerClient
4 {
5     private static final int TIME_OUT = 100;
6     private boolean _locked;
7     private boolean _opened;
8
9     public TimedDoor(Timer timer)
10    {
11        timer.register(TIME_OUT, this);
12    }
13
14    @Override
15    public void lock()
16    {
17        _locked = true;
18    }
19
20    @Override
21    public void unlock()
22    {
23        _locked = false;
24    }
25
26    @Override
27    public void open()
28    {
29        if (!_locked) {
30            _opened = true;
31        }
32    }
33
34    @Override
35    public void close()
36    {
37        _opened = false;
38    }
39
40    @Override
41    public void timeOutCallback()
42    {
43        _locked = true;
44    }
45 }
46
```

## Timer.java

```
ISP > src > com > directi > training > isp > exercise_refactored > Timer.java > {} com.directi.trainin
1 package com.directi.training.isp.exercise_refactored; Timer.java is
2
3 import java.util.TimerTask;
4
5 public class Timer
6 {
7     public void register(long timeOut, final TimerClient timerClient)
8     {
9         java.util.Timer timerUtility = new java.util.Timer();
10        timerUtility.schedule(new TimerTask()
11        {
12            @Override
13            public void run()
14            {
15                timerClient.timeOutCallback();
16            }
17        }, timeOut);
18    }
19 }
20
```

**Avant la modification** du code l'interface Door contient plusieurs méthodes spécifiques à la porte avec d'autre méthode comme

"timeOutCallback()" et "proximityCallback()"

Donc la classe SensingDoor en implementant l'interface Door est forcée de définir la méthode "timeOutCallback()". Contrairement à la classe qui TimesDoor qui doit définir la méthode "proximityCallback()" !

=>L'interface Door n'a pas une responsabilité unique donc elle ne peut être implementée par les deux types de portes "SensingDoor" et "TimedDoor".

**Après la modification**, on enlève ces deux méthodes "timeOutCallback()" et "proximityCallback()" de l'interface Door et on ajoute deux interfaces TimerClient qui contient la première méthode et l'interface SensorClient qui contient la deuxième méthode. Donc chaque type de porte implemente l'interface qui convient à son type ainsi que l'interface de base Door



# DIP

# DEPENDENCY INVERSION PRINCIPLE

Le cinquième et dernier chapitre du principe SOLID est le principe d'inversion de dépendances : "Dependency Inversion Principle".

Les classes de haut niveau ne devraient pas dépendre directement des classes de bas niveau, mais d'abstractions.

En générale, **les classes de haut niveau** ne devraient pas dépendre directement **des classes de bas niveau**, mais d'abstractions. donc ce sont les classes de bas niveau qui doivent dépendre des contrats de haut niveau


- **Les classes de bas niveau** qui implémentent des fonctionnalités de base : écrire dans un fichier, se connecter à une base de données, retourner une réponse HTTP...
- **Les classes de haut niveau** qui concernent le métier de l'application ("business logic") : la gestion des coûts de livraison, par exemple.

## EncodingModule.java


```
DIP > src > com > directi > training > dip > exercise > EncodingModule.java > {} com.directi.training.dip.exercise
1 package com.directi.training.dip.exercise; EncodingModule.java is not on the classpath of project
2 import java.io.BufferedReader;
3 import java.io.BufferedWriter;
4 import java.io.FileReader;
5 import java.io.FileWriter;
6 import java.io.IOException;
7 import java.io.InputStream;
8 import java.io.InputStreamReader;
9 import java.net.URL;
10 import java.util.Base64;
11 public class EncodingModule
12 {
13     public void encodeWithFiles() throws IOException
14     {
15         BufferedReader reader = null;
16         BufferedWriter writer = null;
17         try {
18             reader = new BufferedReader(
19                 new FileReader("DIP/src/com/directi/training/dip/exercise/beforeEncryption.txt"));
20             writer = new BufferedWriter(
21                 new FileWriter("DIP/src/com/directi/training/dip/exercise/afterEncryption.txt"));
22             String aLine;
23             while ((aLine = reader.readLine()) != null) {
24                 String encodedLine = Base64.getEncoder().encodeToString(aLine.getBytes());
25                 writer.append(encodedLine);
26             }
27         } finally {
28             if (writer != null) {
29                 writer.close();
30             }
31             if (reader != null) {
32                 reader.close();
33             }
34         }
35     }
36     public void encodeBasedOnNetworkAndDatabase() throws IOException
37     {
38         URL url;
39         url = new URL("http", "myfirstappwith.appspot.com", "/index.html");
40         InputStream in;
41         in = url.openStream();
42         InputStreamReader reader = new InputStreamReader(in);
43         StringBuilder inputString1 = new StringBuilder();
44         int c;
45         c = reader.read();
46         while (c != -1) {
47             inputString1.append((char) c);
48             c = reader.read();
49         }
50         String inputString = inputString1.toString();
51         String encodedString = Base64.getEncoder().encodeToString(inputString.getBytes());
52         MyDatabase database = new MyDatabase();
53         database.write(encodedString);
54     }
55 }
56
```



## EncodingModuleClient.java

```
DIP > src > com > directi > training > dip > exercise >  EncodingModuleClient.java > ...  
1  package com.directi.training.dip.exercise;      EncodingModuleCli  
2  
3  import java.io.IOException;  
4  
5  public class EncodingModuleClient  
6  {  
7      Run | Debug  
8      public static void main(String[] args) throws IOException  
9      {  
10         EncodingModule encodingModule = new EncodingModule();  
11         encodingModule.encodeWithFiles();  
12         encodingModule.encodeBasedOnNetworkAndDatabase();  
13     }  
14
```

## MyDatabase.java

```
DIP > src > com > directi > training > dip > exercise >  MyDatabase.java > ...  
1  package com.directi.training.dip.exercise;      MyDatabase.java is  
2  
3  import java.util.HashMap;  
4  import java.util.Map;  
5  
6  public class MyDatabase  
7  {  
8      private static Map<Integer, String> _data = new HashMap<>();  
9      private static int _count = 0;  
10  
11     public int write(String inputString)  
12     {  
13         _data.put(++_count, inputString);  
14         return _count;  
15     }  
16 }  
17
```

## EncodingModule.java

```
DIP > src > com > directi > training > dip > exercise_refactored > EncodingModule.java > ...
1  package com.directi.training.dip.exercise_refactored;
2
3  import java.io.IOException;
4  import java.util.Base64;
5
6  public class EncodingModule
7  {
8      public void encode(IReader reader, IWriter writer) throws IOException
9      {
10         String aLine = reader.read();
11         String encodedLine = Base64.getEncoder().encodeToString(aLine.getBytes());
12         writer.write(encodedLine);
13     }
14 }
15
```

## EncodingModuleClient.java

```
DIP > src > com > directi > training > dip > exercise_refactored > EncodingModuleClient.java > {} com.directi.training.dip.exerc
1  package com.directi.training.dip.exercise_refactored;
2
3  import java.io.IOException;
4
5  public class EncodingModuleClient
6  {
7      Run | Debug
8      public static void main(String[] args) throws IOException
9      {
10         EncodingModule encodingModule = new EncodingModule();
11
12         IReader reader = new MyFileReader(
13             "DIP/src/com/directi/training/dip/exercise_refactored/beforeEncryption.txt");
14         IWriter writer = new MyFileWriter(
15             "DIP/src/com/directi/training/dip/exercise_refactored/afterEncryption.txt");
16         encodingModule.encode(reader, writer);
17
18         reader = new MyNetworkReader("http", "myfirstappwith.appspot.com", "/index.html");
19         writer = new MyDatabaseWriter();
20         encodingModule.encode(reader, writer);
21     }
22 }
```

## IReader.java

```
DIP > src > com > directi > training > dip > exercise_refactored > IReader.java
1 package com.directi.training.dip.exercise_refactored;
2
3 import java.io.IOException;
4
5 public interface IReader
6 {
7     String read() throws IOException;
8 }
9
```

## IWriter.java

```
DIP > src > com > directi > training > dip > exercise_refactored > IWriter.java >
1 package com.directi.training.dip.exercise_refactored;
2
3 import java.io.IOException;
4
5 public interface IWriter
6 {
7     void write(String encodedLine) throws IOException;
8 }
9
```

## MyDatabaseWriter.java

```
DIP > src > com > directi > training > dip > exercise_refactored > MyDatabaseW
1 package com.directi.training.dip.exercise_refactored;
2
3 public class MyDatabaseWriter implements IWriter
4 {
5     @Override
6     public void write(String input)
7     {
8         MyDatabase database = new MyDatabase();
9         database.write(input);
10    }
11 }
```

## MyDatabase.java

```
DIP > src > com > directi > training > dip > exercise_refactored > MyDatabase.java > {} com
1  package com.directi.training.dip.exercise_refactored;
2
3  import java.util.HashMap;
4  import java.util.Map;
5
6  public class MyDatabase
7  {
8      private static Map<Integer, String> _data = new HashMap<>();
9      private static int _count = 0;
10
11     public int write(String inputString)
12     {
13         _data.put(++_count, inputString);
14         return _count;
15     }
16 }
17
```

## MyFileReader.java

```
DIP > src > com > directi > training > dip > exercise_refactored > MyFileReader.java > {} com.directi.training.dip.exercise_refactored.MyFileReader.java is not found
1  package com.directi.training.dip.exercise_refactored;
2
3  import java.io.BufferedReader;
4  import java.io.FileReader;
5  import java.io.IOException;
6
7  public class MyFileReader implements IReader
8  {
9      private String _fileName;
10
11     public MyFileReader(String fileName)
12     {
13         _fileName = fileName;
14     }
15
16     @Override
17     public String read() throws IOException
18     {
19         StringBuilder lines = new StringBuilder();
20         BufferedReader reader = new BufferedReader(new FileReader(_fileName));
21         String aLine;
22         while ((aLine = reader.readLine()) != null) {
23             lines.append(aLine);
24         }
25         reader.close();
26         return lines.toString();
27     }
28 }
```

## MyFileWriter.java

```
DIP > src > com > directi > training > dip > exercise_refactored > MyFileWriter.java > {} com.directi.training.dip
1 package com.directi.training.dip.exercise_refactored;
2
3 import java.io.BufferedWriter;
4 import java.io.FileWriter;
5 import java.io.IOException;
6
7 public class MyFileWriter implements IWriter
8 {
9     private String _fileName;
10
11     public MyFileWriter(String fileName)
12     {
13         _fileName = fileName;
14     }
15
16     @Override
17     public void write(String encodedLine) throws IOException
18     {
19         BufferedWriter writer = new BufferedWriter(new FileWriter(_fileName));
20         writer.write(encodedLine);
21         writer.close();
22     }
23 }
24
```

## MyNetworkReader.java

```
DIP > src > com > directi > training > dip > exercise_refactored > MyNetworkReader.java > {} com.d
1 package com.directi.training.dip.exercise_refactored; MyNetworkReader
2
3 import java.io.IOException;
4 import java.io.InputStream;
5 import java.io.InputStreamReader;
6 import java.net.URL;
7
8 public class MyNetworkReader implements IReader
9 {
10     private String _protocol;
11     private String _host;
12     private String _file;
13
14     public MyNetworkReader(String protocol, String host, String file)
15     {
16         _protocol = protocol;
17         _host = host;
18         _file = file;
19     }
20
21     @Override
22     public String read() throws IOException
23     {
24         URL url = new URL(_protocol, _host, _file);
25         InputStream in = url.openStream();
26         InputStreamReader reader = new InputStreamReader(in);
27         StringBuilder inputString = new StringBuilder();
28         int c;
29         c = reader.read();
30         while (c != -1) {
31             inputString.append((char) c);
32             c = reader.read();
33         }
34         return inputString.toString();
35     }
36 }
37
```

**Avant la modification** du code la classe EncodingModule contient deux méthodes d'encodage "encodeWithFiles()" et "encodeBasedOnNetworkAndDatabase()". La classe EncodingModuleClient dépend entièrement de la classe EncodingModule => Il existe un couplage très fort entre les deux classes!

**Après la modification**, on a procédé comme suit:

- On ajoute deux interfaces "IRead" et "IWriter"
- On ajoute deux classes pour chaque type d'encodage qui permettent de lire et d'écrire :
  - MyFileWriter et MyDatabaseWriter pour écrire des données sur un fichier ou bien dans la base de données
  - MyFileReader et MyNetworkReader pour lire les données soit d'un fichier ou bien du réseau



- La classe EncodingModule contient une méthode générique "encode()" qui a comme paramètres des interfaces IWriter et IReader, donc on peut utiliser l'une des deux méthodes d'encodage selon le besoin avec un couplage faible entre les classes
- Dans la classe EncodingModuleClient on va préciser le type d'encodage dont on a besoin en passant les données nécessaires pour les méthodes génériques et laisser le soin d'instancier les objets nécessaires au programme.



**MERCI**

**2021-2022**