Partie 2 TP 3 de POO

# Concept SOLID

## S comme Single Responsibility Principe :

Ce concept étroitement lié à la cohésion, il permet d’expliquer une classe en une seule phrase :

La classe Y fait une seule chose. Cette phrase ne doit pas contenir de « et » ou de « ou » car cela voulant dire qu’elle a plus d’une responsabilité.

Dans notre exemple, nous allons prendre le cas d’un système de paramêtre pour changer un email :

|  |
| --- |
| public class UserSettingService  {    public void changeEmail(User user)    {      if(checkAccess(user))      {         //Grant option to change      }    }    public boolean checkAccess(User user)    {      //Verify if the user is valid.    }  } |

Le problème ici, est que la classe UserSettingService a deux responsabilités : Changer le mot de passe et savoir si nous avons accès au compte. Si nous voulons corriger cela donnera :

|  |
| --- |
| public class UserSettingService  {    public void changeEmail(User user)    {      if(SecurityService.checkAccess(user))      {         //Grant option to change      }    }  }  public class SecurityService  {    public static boolean checkAccess(User user)    {      //check the access.    }  } |

## O comme Open/Closed Principle :

Ce principe par d’une méthode fermée à la modification et ouverte à l’extension. Créer une sous-classe ou ajouter des membres, permettent de modifier l’état ou le comportement sans modifier ce qui existe déjà.

Dans notre exemple, nous avons un système de vérification d’âge pour un film :

|  |
| --- |
| public class UserAgeValidatorFilm {    public boolean isOldEnoughToWatchMovie(int age) {      return age >= 18;    }  } |

Maintenant, nous allons ajouter plusieurs critères pour les film -12 et -6 :

|  |
| --- |
| public interface UserAgeValidatorFilm {    boolean isOldEnoughToWatchMovie(int age);  }    public class AdulteAgeValidator implements UserAgeValidatorFilm {    public boolean isOldEnoughToWatchMovie(int age) {      return age >= 18;    }  }    public class KidAgeValidator implements UserAgeValidatorFilm {    public boolean isOldEnoughToWatchMovie(int age) {      return age >= 6;    }  }  public class AdolescentAgeValidator implements UserAgeValidatorFilm {    public boolean isOldEnoughToWatchMovie(int age) {      return age >= 12;    }  } |

## L comme Liskov Substitution Principle :

Si une classe hérite d’une autre classe, elle doit se conformer sinon le principe n’est pas respecté. Dans cet exemple, nous avons à faire a des véhicules

|  |
| --- |
| class TrasportationDevice  {  String name;  String getName() { ... }  void setName(String n) { ... }  double speed;  double getSpeed() { ... }  void setSpeed(double d) { ... }    Engine engine;  Engine getEngine() { ... }  void setEngine(Engine e) { ... }  void startEngine() { ... }  }  class Car extends TransportationDevice  {  @Override  void startEngine() { ... }  }  class Bicycle extends TransportationDevice  {  @Override  void startEngine() /\*problem!\*/  } |

Ici, nous avons un problème car la classe bicyclette hérite de Transportation device, or, un vélo ne possède pas de moteur donc il n’a pas le même comportement que la classe dont elle hérite.

|  |
| --- |
| class TrasportationDevice  {  String name;  String getName() { ... }  void setName(String n) { ... }  double speed;  double getSpeed() { ... }  void setSpeed(double d) { ... }  }  class DevicesWithoutEngines extends TransportationDevice  {  void startMoving() { ... }  }  class DevicesWithEngines extends TransportationDevice  {  Engine engine;  Engine getEngine() { ... }  void setEngine(Engine e) { ... }  void startEngine() { ... }  }  class Car extends DevicesWithEngines  {  @Override  void startEngine() { ... }  }  class Bicycle extends DevicesWithoutEngines  {  @Override  void startMoving() { ... }  } |

Ici, nous avons créé deux classes supplémentaires, une pour les véhicules avec moteur et une autre pour les véhicule sans moteur.

## I comme Interface Segregation Principle :

Ce principe consiste d’éviter qu’une classe responsable d’une chose sache qu’une autre classe peut faire autre chose.

|  |
| --- |
| package com.gkatzioura.solid.segragation;    public interface Athlete {       void compete();       void swim();       void highJump();       void longJump();   }  package com.gkatzioura.solid.segragation;    public class JohnDoe implements Athlete {       @Override      public void compete() {          System.out.println("John Doe started competing");      }       @Override      public void swim() {          System.out.println("John Doe started swimming");      }       @Override      public void highJump() {      }       @Override      public void longJump() {      }  } |

Le problème est que JonhDoe est un nageur et n’est pas capable de faire du saut en hauteur ou en longueur. Donc cela représente, une classe qui a accès à des méthodes qu’elle ne devrait pas avoir.

|  |
| --- |
| package com.gkatzioura.solid.segragation;   public interface Athlete {       void compete();       void swim();       void highJump();       void longJump();   }  package com.gkatzioura.solid.segragation;  public interface SwimmingAthlete extends Athlete {      void swim();  }  package com.gkatzioura.solid.segragation;  public interface JumpingAthlete extends Athlete {      void highJump();      void longJump();  }  package com.gkatzioura.solid.segragation;  public class JohnDoe implements SwimmingAthlete {      @Override      public void compete() {          System.out.println("John Doe started competing");      }      @Override      public void swim() {          System.out.println("John Doe started swimming");      }  } |

Ici, nous avons corrigé ne problème car JohnDoe ne possède comme méthode que ce qu’il peut utiliser.

D comme Dependency Inversion Principle :

Ce principe, peut être résumé en deux conditions :

* Les modules de haut niveau ne doivent pas dépendre de ceux de bas niveaux (les deux doivent être dépendants d’abstract class)
* L’abstraction ne doit pas dépendre de détails mais les détails doivent dépendre d’abstractions

|  |
| --- |
| package com.gkatzioura.solid.di;  public class BackEndDeveloper {  public void writeJava() {  }  }  package com.gkatzioura.solid.di;  public class FrontEndDeveloper {  public void writeJavascript() {  }  }  package com.gkatzioura.solid.di;  public class Project {  private BackEndDeveloper backEndDeveloper = new BackEndDeveloper();  private FrontEndDeveloper frontEndDeveloper = new FrontEndDeveloper();  public void implement() {  backEndDeveloper.writeJava();  frontEndDeveloper.writeJavascript();  }  } |

Le project (qui est un module de haut niveau, dépend de back et front developer et des détails constituent les classes donc cela doit être corrigé :

|  |
| --- |
| package com.gkatzioura.solid.di;  public interface Developer {  void develop();  }  package com.gkatzioura.solid.di;  public class BackEndDeveloper implements Developer {  @Override  public void develop() {  writeJava();  }  private void writeJava() {  }  }  package com.gkatzioura.solid.di;  public class FrontEndDeveloper implements Developer {  @Override  public void develop() {  writeJavascript();  }  public void writeJavascript() {  }  }  package com.gkatzioura.solid.di;  import java.util.List;  public class Project {  private List<Developer> developers;  public Project(List<Developer> developers) {  this.developers = developers;  }  public void implement() {  developers.forEach(d->d.develop());  }  } |