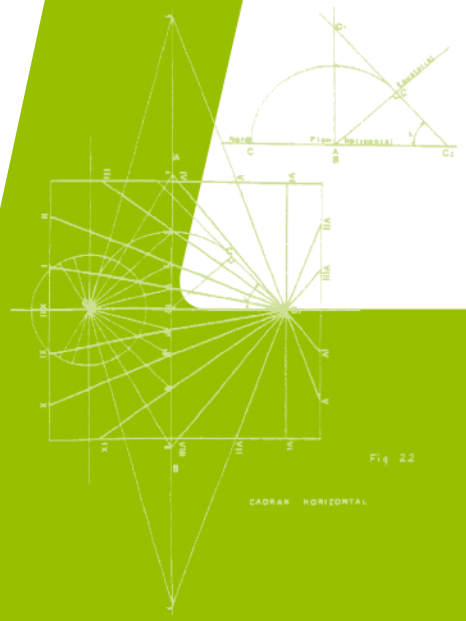
ICT301
2025-2026

Université de Yaoundé I

ICT301 : Architecture logicielle et conception

Les principes SOLID



Décembre 2025

Valéry MONTHE

valery.monthe@facsciences-uy1.cm

Bureau R114, Bloc pédagogique 1

Plan du cours

1. Un exemple pour commencer
2. Pourquoi bien concevoir
3. Concepts de base
4. Les cinq principes SOLID
    1. **S**ingle Responsibility Principle
    2. **O**pen-Closed Principle
    3. **L**iskov Substitution Principle
    4. **I**nterface Segregation Principle
    5. **D**ependency Inversion Principle

Un exemple

Une système de boutique en ligne gère plusieurs mode de paiement : payement mobile (opérateur de téléphonie), carte de crédit, crypto-monnaie, PayPal. En parcourant le code de l'application, on a trouvé cet extrait.

```java
package solid.exemples;
public class PaymentProcessor {
    public void processPayment(String paymentType, double amount) {
        if (paymentType.equals(anObject:"mobil_pay")) {
            System.out.println("Processing mobile payment: " + amount +" CFA");
            // Logique spécifique aux payements mobiles
        } else if (paymentType.equals(anObject:"credit_card")) {
            System.out.println("Processing credit card payment: " + amount +" CFA");
            // Logique spécifique aux cartes bancaires
        } else if (paymentType.equals(anObject:"paypal")) {
            System.out.println("Processing PayPal payment: " + amount +" CFA");
            // Logique spécifique à PayPal
        } else if (paymentType.equals(anObject:"crypto")) {
            System.out.println("Processing crypto payment: " + amount +" CFA");
            // Logique spécifique aux crypto-monnaies
        }
    }
```

```java
    // Utilisation
     public class Main {
         public static void main(String[] args) {
             PaymentProcessor processor = new PaymentProcessor();
             processor.processPayment(paymentType:"mobil_pay", amount:1500);
             processor.processPayment(paymentType:"credit_card", amount:6500);
             processor.processPayment(paymentType:"paypal", amount:4500);
         }
     }
```

Exemple : Problèmes

- PaymentProcessor gère toutes les moyens de paiement
- Il faut modifier PaymentProcessor pour ajouter un nouveau mode de payement
- Le code dépend directement des détails du mode de paiement

Risques de dégénérescence de l'application

Lorsqu'une application est en PRODUCTION, les phénomènes suivantes sont observés pendant les activités de développement :

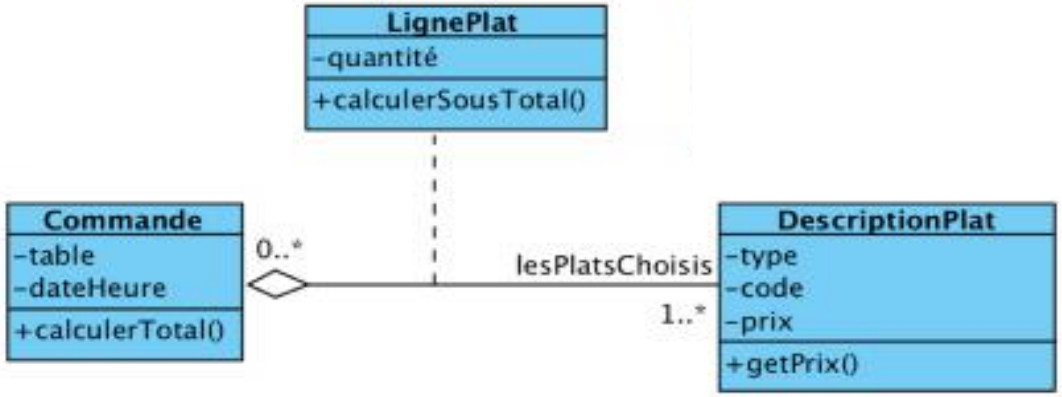
- La **rigidité** : chaque évolution risque d'impacter d'autres parties de l'application. Le coût de développement augmente et avec l'approche de la date de livraison, la qualité de code est négligée.
- La **fragilité** : modifier une partie du code entraîne des erreurs dans d'autres parties du logiciel qui devient peu robuste.
- **L'immobilité** / **non réutilisabilité** : il est difficile de retirer une partie du code pour la réutiliser ailleurs.

Source de problème : la gestion des dépendances

- Modifications de code inévitables avec l ’évolution des besoins
- La conception vise à amortir l’impact des dépendances et à aboutir aux qualités de :
  - **Robustesse**: les changement n’introduisent pas de régression;
  - **Extensibilité** : l’ajout de fonctionnalités doit être facile;
  - **Réutilisabilité** : possibilité de réutiliser certaines parties du système pour en construire d’autres.

Concepts de base

- Les **responsabilité** d'une classe :
  - Ce qu'elle **SAIT**
  - Ce qu'elle est capable de **FAIRE**
- Exemple : pour la classe **LignePlat** suivant
  - Elle **sait** : à quel objet **Commande** elle appartient, et quel objet **DescriptionPlat** elle comprend
  - Elle **sait** combien de plats elle comporte
  - Elle **peut calculer** son sous-total(prix*quantité de plats)



Concepts de base

- Le **contrat** = services rendus par une classe
  - Exprimé par les opérations de classe ou d'interface
  - **Stable**
  - Masque les détails de réalisation
- **L'implémentation**
  - Représente les classes concrètes
  - **Peut évoluer**
- Toujours chercher à **bien les dissocier**

Concepts de base : Cohésion / couplage

- La **Cohésion** = esprit de famille
  - Degré avec lequel les tâches d'un module sont fonctionnellement reliées entre elles
    - ✓Quel est sont objectif?
    - ✓Fait-il une ou plusieurs choses?
    - ✓Quelle est sa fonction au sein du système?
- Le **Couplage** = Dépendance
  - Force de l'interaction entre les modules d'un système
    - ✓Comment les modules travaillent ensemble?
    - ✓Qu'ont-ils besoin de savoir l'un de l'autre?
    - ✓Quand font-ils appel aux fonctionnalités de chacun?
    - ✓Exemple : une classe qui crée une instance d'une autre classe = couplage fort.

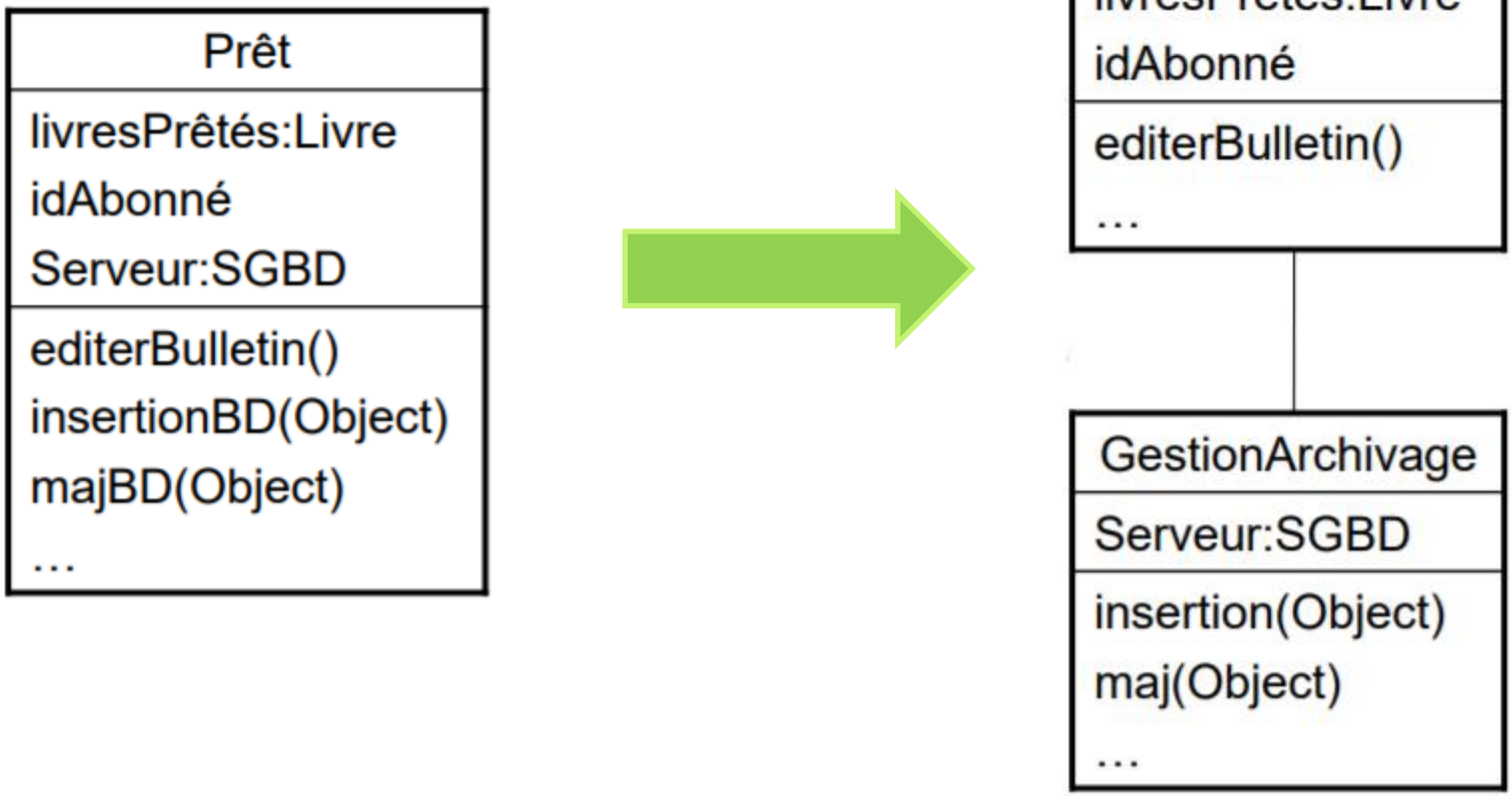
      -> Car ne peut pas être tester indépendamment de l'autre classe.

Concepts de base : Cohésion / couplage

- Une **faible cohésion** altère :
  - ✓La compréhension
  - ✓La réutilisabilité
  - ✓La maintenabilité
  - ✓Le code est fragile, car subit toute sorte de changement très fréquemment.
- Un **Fort couplage** :
  - ✓Maintenance difficile et Lisibilité faible
  - ✓Un changement dans une classe impacte plusieurs autres
  - ✓Les classes prises isolément sont difficiles à comprendre
  - ✓Réutilisation difficile : il faut les classes dont elle dépend.

Parfois volontaire : code rendu impénétrable pour protéger ses sources de rétro-ingénierie

Concepts de base : Cohésion / couplage



Concepts de base : Cohésion / couplage

Concepts de base : Cohésion / couplage

Forte cohésion : Quelques règles

- Regrouper les éléments en **forte relation**
- Regrouper les classes qui rendent des **services de même nature aux utilisateurs**
- Isoler les **classes stables** de celles qui risquent d'évoluer au cours du projet
- Isoler les classes **métiers** des classes **applicatives**

Faible couplage: Quelques règles

- Préférer un couplage avec des interfaces, pas des classes concrètes
- Ne pas ajouter plus de dépendance que nécessaire

Sur l'exemple: Cohésion / couplage

```java
package solid.exemples;

// Interface commune pour tous les moyens de paiement
interface PaymentMethod {
    void processPayment(double amount);
}

// Implémentations concrètes
class CreditCardPayment implements PaymentMethod {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing credit card payment: " + amount + "CFA");
    }
}
```

Sur l'exemple: Cohésion / couplage

```java
public class PaymentProcessor2 {
    private PaymentMethod paymentMethod;
    public PaymentProcessor2(PaymentMethod paymentMethod) {
        this.paymentMethod = paymentMethod;
    }

    public void processPayment(double amount) {
        paymentMethod.processPayment(amount);
    }
}
```

```java
// Utilisation
class Main {
    Run | Debug
    public static void main(String[] args) {
        PaymentMethod creditCard = new CreditCardPayment();
        PaymentProcessor2 processor1 = new PaymentProcessor2(creditCard);
        processor1.processPayment(amount:10000);

        PaymentMethod paypal = new PayPalPayment();
        PaymentProcessor2 processor2 = new PaymentProcessor2(paypal);
        processor2.processPayment(amount:5000);
    }
}
```

Les principes SOLID

5 principes

- **S**ingle Responsibility Principle
- **O**pen-Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

SOLIDE = Single, Open, Liskov, Interface, Dependency

SRP : Single Responsability Principle

- **SRP : Une classe = une responsabilité**
- Un module(fonction,classe, paquet, etc.) ne devrait avoir qu'une seule raison de changer
- **SRP : Une classe = une responsabilité**
- **Une seule responsabilité = une seule raison d'être modifiée**
- On a souvent tendance à donner trop de responsabilité à un objet :
  - ✓ Analyser les méthodes de la classe
  - ✓ Les regrouper pour constituer des ensembles homogènes : accès BD, API, etc
  - ✓ Affecter si possible les responsabilités correspondants aux informations que la classe possède.

SRP : Exemple

- Soit l'extrait de code : classe **Book**

```java
package ict301.solid.srp;
public class Book {
    private String title;
    private String author;
    private String content;

    public Book(String title, String author, String content) {
        this.title = title;
        this.author = author;
        this.content = content; }

    // Responsabilité 1 : Gérer les données du livre
    public String getTitle() { return title; }
    public String getAuthor() { return author; }
    public String getContent() { return content; }

    // Responsabilité 2 : Afficher le livre (présentation)
    public void printToScreen() {
        System.out.println("Titre: " + title);
        System.out.println("Auteur: " + author);
        System.out.println("Contenu: " + content); }

    // Responsabilité 3 : Sauvegarder le livre (persistance)
    public void saveToDatabase() {
        System.out.println("Sauvegarde du livre '" + title + "' en base de données..."); }

    // Responsabilité 4 : (logique métier)
    public void emprunter(String lecteur) {
        System.out.println("Emprunt du livre '" + title + "' par " + lecteur); } }
```

SRP : Exemple

- Utilisation de la classe Book

```
1.  package ict301.solid.srp;
2.
    // Utilisation de la classe Book
3.  public class Main {
4.      public static void main(String[] args) {
5.          Book book = new Book("Les principes SOLID", "Etudiants de M1-GL", "Révision des
    principes SOLID");
6.          book.printToScreen();
7.          book.saveToDatabase();
8.          book.emprunter("Marcial");
9.      }
10. }
```

SRP : Exemple

La classe Book viole le SRP car elle a plusieurs responsabilités :

- Définir le modèle de données : les données sur le ivre (***entité***)
- Afficher le livre : ***présentation***
- Gérer l'emprunt : ***logique métier***
- Sauvegarder le livre : ***persistance***

Modifier chacun de ces éléments force à modifier la même classe Book.

>>> fort couplage et donc faible maintenabilité

SRP : Exemple (refactoring)

- **Solution avec SRP**
  - On sépare les 4 responsabilités en 4 classes distinctes
  - 1. Classe **BookSRP**

```
1.  package ict301.solid.srp;
2.  // Gère les données du livre
3.  public class BookSRP {
4.      private String title;
5.      private String author;
6.      private String content;
7.
        public BookSRP(String title, String author, String content) {
8.          this.title = title;
9.          this.author = author;
10.         this.content = content;
11.     }
12.
        // Méthodes pour accéder aux données (getters)
13.     public String getTitle() { return title; }
14.     public String getAuthor() { return author; }
15.     public String getContent() { return content; }
16. }
```

SRP : Exemple (refactoring)

- Solution avec SRP
  - 2. Classe **BookPrinter**

```
1.  // Classe BookPrinter : présentation
2.  class BookPrinter {
3.      // Méthode pour afficher le livre à l'écran
4.      public void printToScreen(BookSRP book) {
5.          System.out.println("===Print to Screen=== ");
6.          System.out.println("Titre: " + book.getTitle());
7.          System.out.println("Auteur: " + book.getAuthor());
8.          System.out.println("Contenu: " + book.getContent());
9.      }
10.
        // On peut ajouter d'autres méthodes d'affichage sans toucher à BookSRP
11.     public void printToHTML(BookSRP book) {
12.         System.out.println("\n===Print to HTML=== ");
13.         System.out.println("<h1>" + book.getTitle() + "</h1>");
14.         System.out.println("<h2>Par " + book.getAuthor() + "</h2>");
15.         System.out.println("<p>" + book.getContent() + "</p>");
16.     }
17. }
18.
```

SRP : Exemple (refactoring)

- Solution avec SRP
  - 3 et 4. Classes **BookSaver** et **BookBusinessLogic**

```
1.  // Classe BookSaver : persistance
2.  class BookSaver {
3.      public void saveToDatabase(BookSRP book) {
4.          System.out.println("\nSauvegarde de '" + book.getTitle() + "' en base de données..."); }
5.
        //On peut ajouter d'autres façons de sauvegarder
6.      public void saveToFile(BookSRP book, String filename) {
7.          System.out.println("\nSauvegarde de '" + book.getTitle() + "' dans " + filename); }
8.  }
9.
    // Classe BookBusinessLogic : Logique métier
10. class BookBusinessLogic {
11.     public void emprunter(BookSRP book, String lecteur) {
12.         System.out.println("\nEmprunt du livre '" + book.getTitle() + "' par " + lecteur); }
13.
        //On peut ajouter d'autres logiques
14.     public void autreService(BookSRP book) {
15.         System.out.println("\nAutre logique métier sur le livre '" + book.getTitle());}
16. }
```

OCP : Open Close Principle

- **OCP = Etendre sans modifier**
- La rigidité et la fragilité du code viennent de l'impact d'un changement d'une partie de l'application sur d'autres.
- Les entités logicielles (classes, packages, etc) doivent êtres :
  - ✓ **Ouvertes à l'extension** : on peut ajouter des fonctionnalités non prévues à la création
  - ✓ **Fermées à la modification** : les changements introduits ne modifient pas le code existant
- L'extensibilité se traduit par l'ajout de code uniquement
- Une fois le code produit, testé et livré en production, le seul moyen de modifier est d'étendre le code
- L'**abstraction** et le **polymorphisme** sont les moyens pour y parvenir

OCP : Exemple

- Soit le bout de code suivant :
- Quel problème pose t-il ?

```
1.  package ict301.solid.ocp;
2.
    public class AreaCalculator {
3.      public double calculateArea(Object shape) {
4.          if (shape instanceof Rectangle) {
5.              Rectangle rectangle = (Rectangle) shape;
6.              return rectangle.getWidth() * rectangle.getHeight();
7.          } else if (shape instanceof Circle) {
8.              Circle circle = (Circle) shape;
9.              return Math.PI * circle.getRadius() * circle.getRadius();
10.         }
11.         throw new IllegalArgumentException("Unknown shape");
12.     }
13. }
14.
```

OCP : Exemple (refactoring)

```java
package ict301.solid.ocp;

interface Shape {
    double calculateArea();
}

class Rectangle implements Shape {
    private double width;
    private double height;
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
    @Override
    public double calculateArea() {
        return width * height;
    }
}

class Circle implements Shape {
    private double radius;
    public Circle(double radius) {
        this.radius = radius;
    }
    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}
```

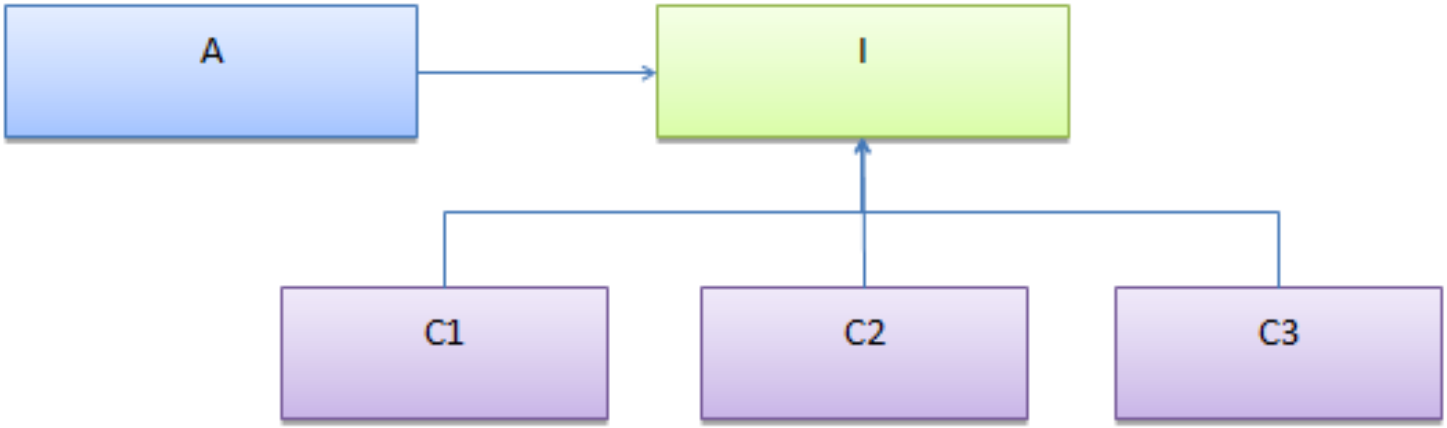
```java

public class AreaCalculator2 {
    public double calculateArea(Shape shape) {
        return shape.calculateArea();
    }
}
```

```java
package ict301.solid.ocp;
public class Main {
    public static void main(String[] args) {
        Shape shape = new Rectangle(4, 3);
        System.out.println("Area =  '" + shape.calculateArea() + "'");
    }
}
```

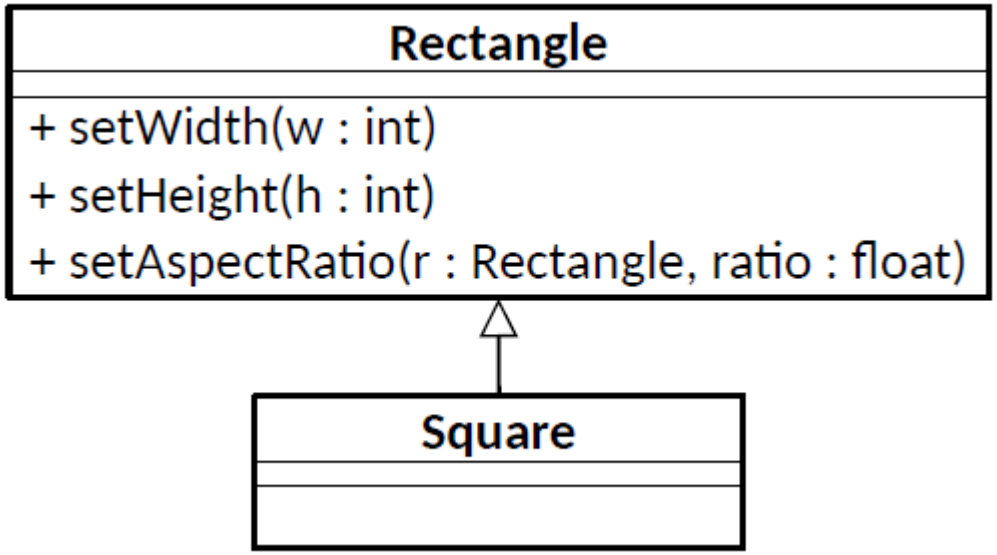
LSP: Liskov Substitution Principle

- **LSP : Les sous-classes doivent être substituables.**
- On doit pouvoir placer la sous-classe partout où figure la classe parent
- Un sous-type S doit être substituable à son type de base T dans toute l'application où T est utilisé sans causer de comportement non désiré dans le programme.
- *Si B et C sont des implémentations de A, alors B et C doivent pouvoir être inter-changées sans affecter l'exécution du programme.*



LSP: Exemple 1

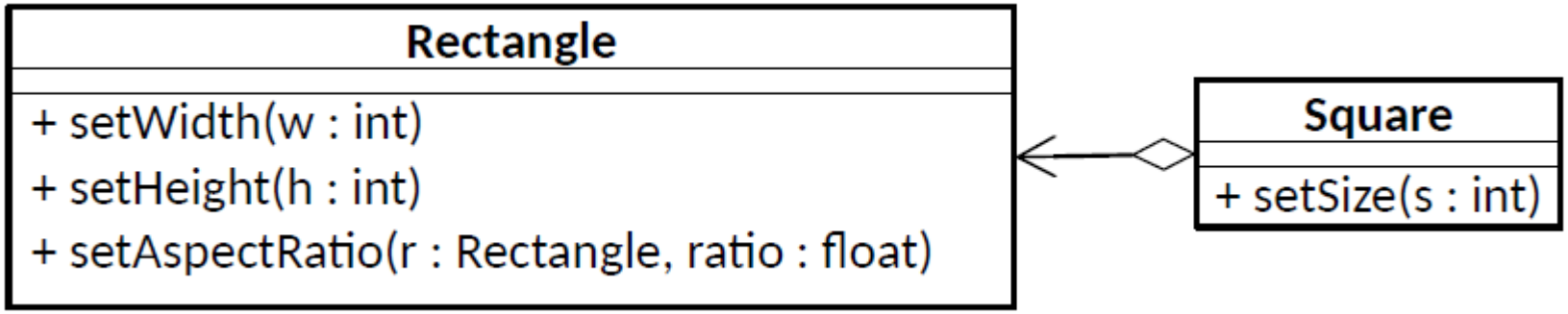
- Soit la modélisation suivante : un carré est un rectangle particulier



- Le carré ne respecte pas tout le contrat de Rectangle,
- **Par exemple** : après l'appel de ***setWith()*** on s'attend à ce que la largeur ait la nouvelle valeur et la hauteur conserve son ancienne valeur. Or ce n'est pas le cas pour le carré

LSP: Exemple 1 (refactoring)

- Le carré n'hérite plus de rectangle
- Le carré utilise le rectangle par composition



- Cette fois le carré n'est pas substituable au rectangle

LSP: Exemple2

- Soit le bout de code suivant :
- Quel problème pose t-il ?

```
1.  package C.solid.lsp;
2.   class Rectangle {
3.      protected int width;
4.      protected int height;
5.      public void setWidth(int width) {
6.          this.width = width;
7.      }
8.      public void setHeight(int height) {
9.          this.height = height;     }
10.     public int getArea() {
11.         return width * height; }
12. }
```

```
1.  class Square extends Rectangle {
2.      @Override
3.      public void setWidth(int width) {
4.          super.setWidth(width);
5.          super.setHeight(width);
6.      }
7.
        @Override
8.      public void setHeight(int height) {
9.          super.setWidth(height);
10.         super.setHeight(height); }
11. }
```

```
1.  public class Mainlsp {
2.      public static void main(String[] args) {
3.          Rectangle rectangle = new Rectangle();
4.          rectangle.setWidth(5);
5.          rectangle.setHeight(4); // Attend une aire de 20
6.          System.out.println("Aire du Rectagle ="+rectangle.getArea());
7.
            Rectangle rectangle1 = new Square();
8.          rectangle1.setWidth(5);
9.          rectangle1.setHeight(4); // Attend une aire de 20, mais obtient 16 (4x4)
10.         System.out.println("Aire du Carré ="+rectangle1.getArea()); // Résultat inattendu }
11. }
```

LSP: Exemple2 (refactoring)

- Application de LSP

```java
package ict301.solid.lsp;

interface Shape {
    int getArea();
}
```

```java
class Square implements Shape {
    private int side;
    public Square(int side) {
        this.side = side;
    }

    @Override
    public int getArea() {
        return side * side;
    }
}
```

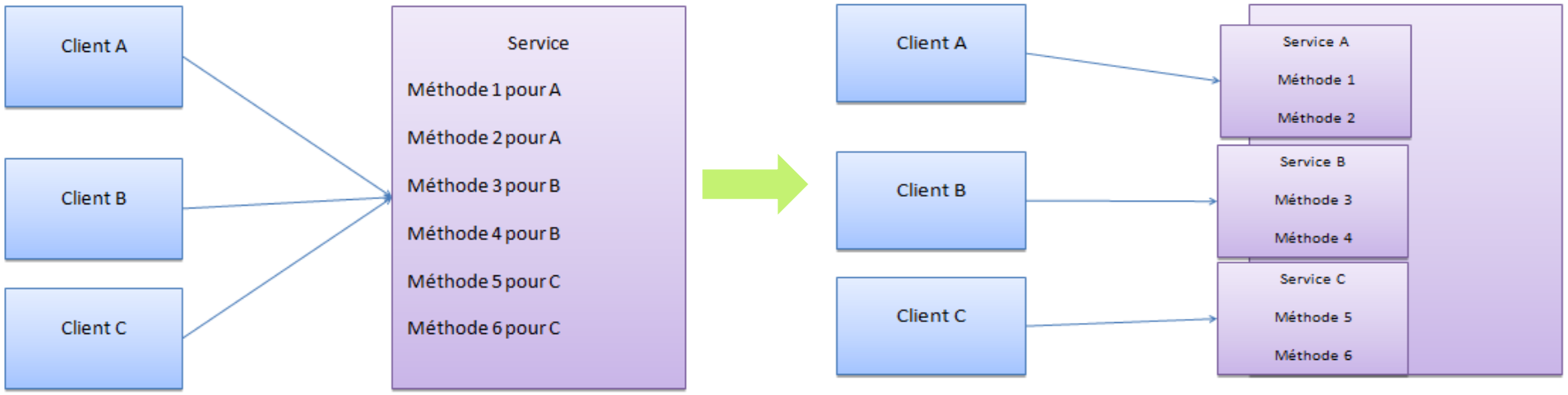
```java
class Rectangle implements Shape {
    private int width;
    private int height;
    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public int getArea() {
        return width * height;
    }
}
```

```java
public class Mainlsp2 {
    public static void main(String[] args) {
        Shape square = new Square(3);
        Shape rectangle = new Rectangle(3,4);
        System.out.println("Square Area : "+square.getArea());
        System.out.println("Rectangle Area: "+rectangle.getArea());
    }
}
```

ISP: Interface Segregation Principle

- **ISP : Préférer les petites interfaces ciblées.**
- La dépendance d'une classe à une autre doit être restreinte à l'interface la plus petite possible
  - ✓ Le client d'une classe ne doit pas être forcé de dépendre de méthodes qu'il n'utilise pas.
  - ✓ Le client ne doit voir que les services dont il a besoin
  - ✓ Toute classe client qui utilise une BigInterface a (sauf pour son concepteur) un comportement flou
  - ✓ Toute classe réalisant une interface doit implémenter chacune de ses fonctions



ISP: Exemple

- Soit le bout de code suivant :
- Quel problème pose t-il ?

```java
package ict301.solid.isp;
public interface Worker {
    void work();
    void eat();
}
```

```java
class HumanWorker implements Worker {
    @Override
    public void work() {
        System.out.println("Les humains travaillent");
    }

    @Override
    public void eat() {
        System.out.println(" Les humains mangent");
    }
}
```

```java
class RobotWorker implements Worker {
    @Override
    public void work() {
        System.out.println("Les Robots travaillent sans fatigue");
    }

    @Override
    public void eat() {
        System.out.println("l'on ne doit pas faire manger un robot");
        throw new UnsupportedOperationException("Les Robots ne mangent pas");
    }
}
```

ISP: Exemple (refactoring)

- **Solution avec ISP**

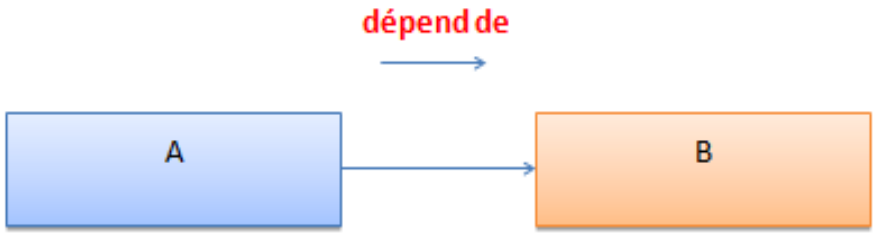
```
1.  interface Workable {
2.      void work();
3.  }
4.
    interface Eatable extends Workable {
5.      void eat();
6.  }
```

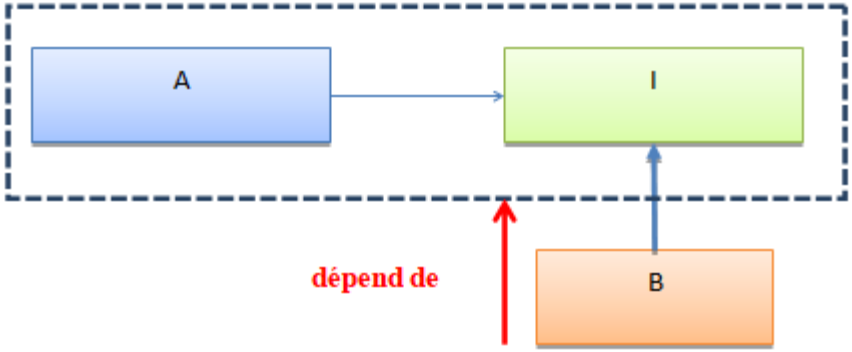
```
1.  package ict301.solid.isp;
2.
    public class MainISP {
3.      public static void main(String[] args) {
4.          HumanWorker human = new HumanWorker();
5.          human.work();
6.          human.eat();
7.
            RobotWorker robot = new RobotWorker();
8.          robot.work();
9.      }
10. }
```

```
1.  class HumanWorker implements Eatable  {
2.      @Override
3.      public void work() {
4.          System.out.println("Les humains travaillent"); }
5.
        @Override
6.      public void eat() {
7.          System.out.println("Les humains mangent"); }
8.  }
9.
    class RobotWorker implements Workable {
10.     @Override
11.     public void work() {
12.         System.out.println("Les Robots travaillent sans fatigue");}
13. }
```

DIP : Dependency Inversion Principle

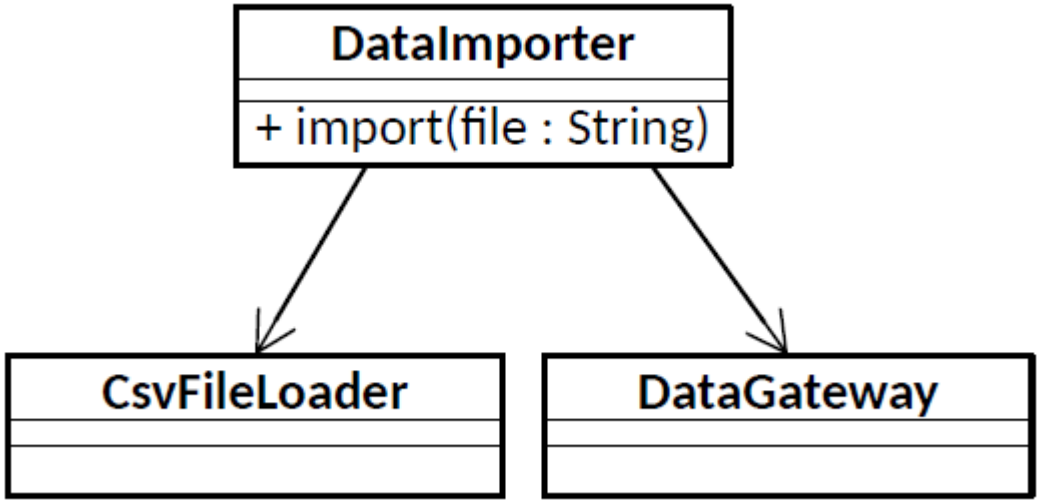
- Les modules de haut niveau (aspect métier) ne doivent pas dépendre de modules de bas niveau (aspect implémentation)
- Les modules d'une application devraient dépendre d'abstractions
  - ✓ Les abstractions ne doivent pas dépendre de détails
  - ✓ Les détails doivent dépendre des abstractions
- Les dépendances d'une classe ne devraient pas être concrètes
  - ✓ Elle ne doit pas connaître l'implémentation de ses dépendances





DIP : Exemple 1

- Dans l'exemple ci-dessous, la classe DataImporter est dépendante du chargeur de fichier et la passerelle de sctockage dans la BD.



- **Problème** : on ne peut pas réutiliser la classe d'importation sans réutiliser le chargeur de fichier et la passerelle de stockage des données.

DIP : Exemple 2

- Soit le bout de code suivant, quel problème pose t-il ?

```
1.  package ict301.solid.dip.old;
2.  class MySQLDatabase {
3.      public void save(String data) {
4.          System.out.println("Saving to MySQL: " + data);
5.      }
6.  }
7.
    public class OrderProcessor {
8.      private MySQLDatabase database;
9.      public OrderProcessor() {
10.         this.database = new MySQLDatabase();
11.     }
12.     public void processOrder(String order) {
13.         database.save(order);
14.     }
15. }
```

```
1.  package ict301.solid.dip.old;
2.  public class Main {
3.      public static void main(String[] args) {
4.          OrderProcessor order = new OrderProcessor();
5.          order.processOrder("'Données à sauvegarder'");
6.      }
7.  }
```

DIP : Exemple 2(Refactoring)

- **Solution avec DIP**

```
1.  package ict301.solid.dip;
2.  interface Database {
3.      public void save(String data);
4.  }
5.
    class MySQLDatabase implements Database {
6.      @Override
7.      public void save(String data) {
8.          System.out.println("Saving to MySQL: " + data);
9.      }
10. }
```

```
1.  class MongoDBDatabase implements Database{
2.      @Override
3.      public void save(String data) {
4.          System.out.println("Saving to MongoDB: " + data);
5.      }
6.  }
7.
    public class OrderProcessor {
8.      private Database database;
9.      public OrderProcessor(Database database) {
10.         this.database = database;
11.     }
12.     public void processOrder(String order) {
13.         database.save(order);
14.     }
15. }
```

DIP : Exemple 2(Refactoring)

- **Solution avec DIP**

```
1.  package ict301.solid.dip;
2.  public class Main {
3.      public static void main(String[] args) {
4.          Database database;
5.
6.          database = new MySQLDatabase();
7.          OrderProcessor order = new OrderProcessor(database);
8.          order.processOrder("'Données à sauvegarder'");
9.
10.         database = new MongoDBDatabase();
11.         OrderProcessor order1 = new OrderProcessor(database);
12.         order1.processOrder("'Données à sauvegarder'");
13.     }
14. }
```

Bibliographie et ressources

Ouvrages recommandés

- Software Architecture in Practice, 3e édition, Len Bass, Paul Clements et Rick Kazman, Addison-Wesley, 2012.
- Architecture logicielle : Concevoir des applications simples, sûres et adaptable, 2e edition, Jacques Printz, Dunod.

Notes de cours

- Architecture logicielle et conception avancée, Ecole polytechnique de Montréal, Yann-Gaël Guéhéneuc.
- Architecture logicielle, Université Joseph Fourier, Lydie du Bousquet
- Architectures logicielles : Livre Blace, Vincent Composieux
- [GLO-3001] : Architecture logicielle, cours de Luc Lamontagne