Dev Quality

Rappels sur les principes SOLID

CC BY NC SA

Plan

1. Single Responsability Principle
2. Open Close principle
3. Liskov's Substitution Principle
4. Interface Segregation Principle
5. Dependency Inversion Principle

Plan

S olid : *Single Responsability Principle*

SRP

Un module ne sert qu'un propos, une fonction qu'une action

Une chose, une seule, pas deux !

- le nom d'une classe/fonction est un indicateur : il doit exprimer un seul concept ou une seule action
- le plus de lignes, le moins de chance de respecter SRP
- méthodes : limiter les arguments nécessaires ⇒ encapsulez les dans un concept au besoin
- un même niveau d'abstraction dans les instructions

SRP est aussi une des clés pour appliquer DRY ⇒ pas de redondance : extraire les lignes similaires dans une méthode

Plan

sOlid : Open/close principle

That smells...

ShapeUtils.java

```
public class ShapeUtils {

  public double computePerimetersSum(List<Shape> shapeList){
    double perimeterSum = 0;
    for (Shape shape : shapeList) {
      if(shape instanceof Square s) {
        perimeterSum += s.getSideLength();
      } else if (shape instanceof Rectangle r) {
        perimeterSum += r.getHeight()*2 + r.getWidth()*2;
      }
    }
    return perimeterSum;
  }
}
```

Ajout d'une nouvelle forme ⇒ modification de cette classe...
2 solutions : par héritage ou en utilisant une interface

sOlid : Open/close principle

par héritage : Shape.java et Square.java

```
public abstract class Shape {

  public abstract double getPerimeter();
}

public class Square extends Shape {

  private double sideLength;

  public Square(double sideLength) {
    this.setSideLength(sideLength);
  }

  public double getSideLength() {
    return sideLength;
  }

  public void setSideLength(double sideLength) {
    this.sideLength = sideLength;
  }

  @Override
  public double getPerimeter() {
    return getSideLength()*4;
  }
```

sOlid : Open/close principle

par interface : Shape.java et Square.java

```
public interface Shape {

  public abstract double getPerimeter();
}

public class Square implements Shape {

  private double sideLength;

  public Square(double sideLength) {
    this.setSideLength(sideLength);
  }

  public double getSideLength() {
    return sideLength;
  }

  public void setSideLength(double sideLength) {
    this.sideLength = sideLength;
  }

  @Override
  public double getPerimeter() {
    return getSideLength()*4;
  }
```

sOlid : Open/close principle

ShapesUtil.java

```
import java.util.List;

public class ShapeUtils {

  public double computePerimetersSum(List<Shape> shapeList) {
    double perimeterSum = 0;
    for (Shape shape : shapeList) {
      perimeterSum += shape.getPerimeter();
    }
    return perimeterSum;
  }
}
```

⇒ plus besoin d'y toucher si on ajoute une nouvelle forme

Plan

soLid : Liskov's Substitution Principle (LSP)

▸ LSP sur Wikipédia Si S est un sous-type de T, alors tout objet de type T peut être remplacé par un objet de type S sans altérer les propriétés désirables du programme concerné

LSP impose des restrictions sur les signatures des méthodes héritées

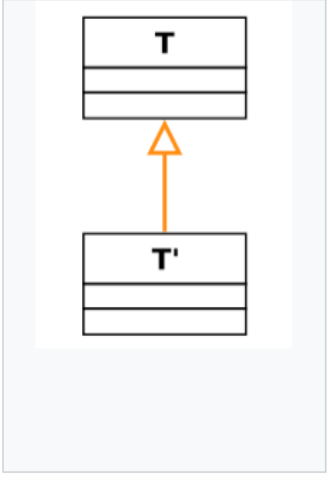
- Contravariance des arguments dans le sous-type
- Covariance du type de retour dans le sous-type.
- pas de nouvelle exception, sauf si elles sont d'un sous-type de l'originale

soLid : Liskov's Substitution Principle (LSP)

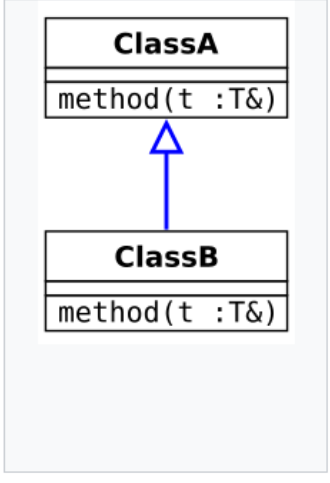
Variance en POO

Wikipédia

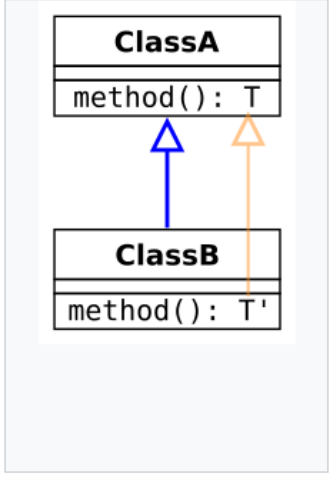
Variance and method overriding: overview



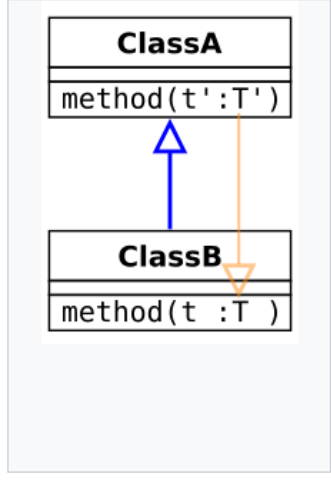
Subtyping of the parameter/return type of the method.



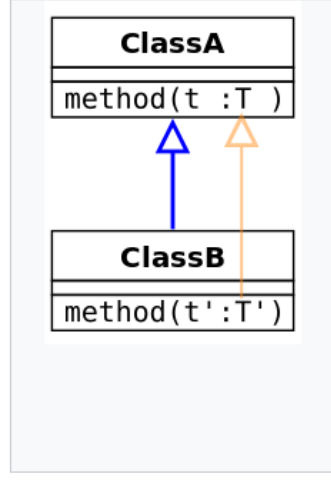
Invariance. The signature of the overriding method is unchanged.



Covariant return type. The subtyping relation is in the same direction as the relation between ClassA and ClassB.



Contravariant parameter type. The subtyping relation is in the opposite direction to the relation between ClassA and ClassB.



Covariant parameter type. Not type safe.

soLid : Liskov's Substitution Principle (LSP)

Mais... *In an interview in 2016, Liskov herself explains that what she presented in her keynote address was an "informal rule"*

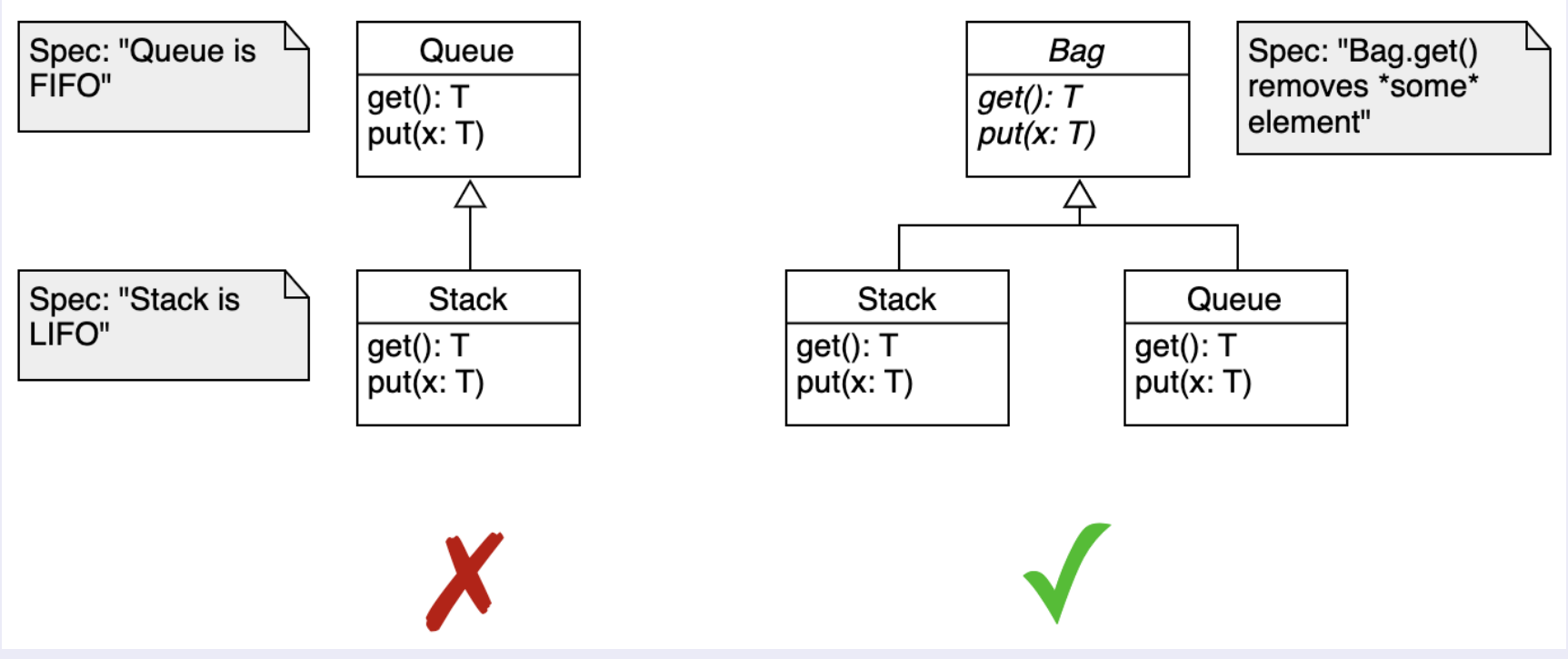
Ce principe est critiqué car parfois trop contraignant : ▸ LSP

Préférez le principe de ***behavioral subtyping*** ▸ plus d'information

Grossièrement, l'idée est surtout de **respecter le contrat que vous passez en héritant d'une classe ou en implémentant une interface**
⇒ le comportement doit être compatible et cohérent.

behavioral subtyping illustré @Wikipedia

L'héritage et les redéfinitions doivent être cohérents avec les spécifications !



Plan

sol I d : Interface Segregation Principle (ISP)

ISP sur Wikipédia

Aucun *client* ne devrait dépendre de méthodes qu'il n'utilise pas

ISP

- Une interface ne comporte que des méthodes utiles pour celui qui l'implémente
- ⇒ sinon, il faut découper en interfaces plus petites
- ⇒ permet un couplage faible

Petite parenthèse : certaines interfaces n'ont pas de méthode, pourquoi ?
en POO, implémenter un type est déjà quelque chose en soi ⇒ **expliciter que l'objet est de ce type**

Plan

soli D : Dependency Inversion Principle (DIP)

Objectif : découpler au maximum les modules logiciels

- Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. **Les deux doivent dépendre d'abstractions** (e.g. interfaces).
- Les abstractions ne doivent pas dépendre des détails. Les détails (les implémentations concrètes) doivent dépendre des abstractions.

OK, c'est bon ?

Exemple avec le *Collections Framework* de Java

Notion de *Collection*

Définition

- Une *collection* est un regroupement structuré de données
- En Java : c'est un **objet qui structure d'autres objets**
- Exemple : `ArrayList<E>`, `HashMap<K,V>`, etc.

L'API des collections Java en 2 packages

- `java.util` : les collections essentielles et le cadriciel associé
- `java.util.concurrent` : programmation concurrente

Exemples de modélisation

- Une main dans un jeu de cartes ⇒ ensemble de `n` cartes
- Un répertoire téléphonique ⇒ tableau associatif `nom` → `numéro`

Les *collections framework* orientés objet

Des interfaces

- Définissent les opérations possibles sur un type de collection
- E.g. `List<E>`, `Map<K,V>`, `Queue<E>`...

Des implémentations

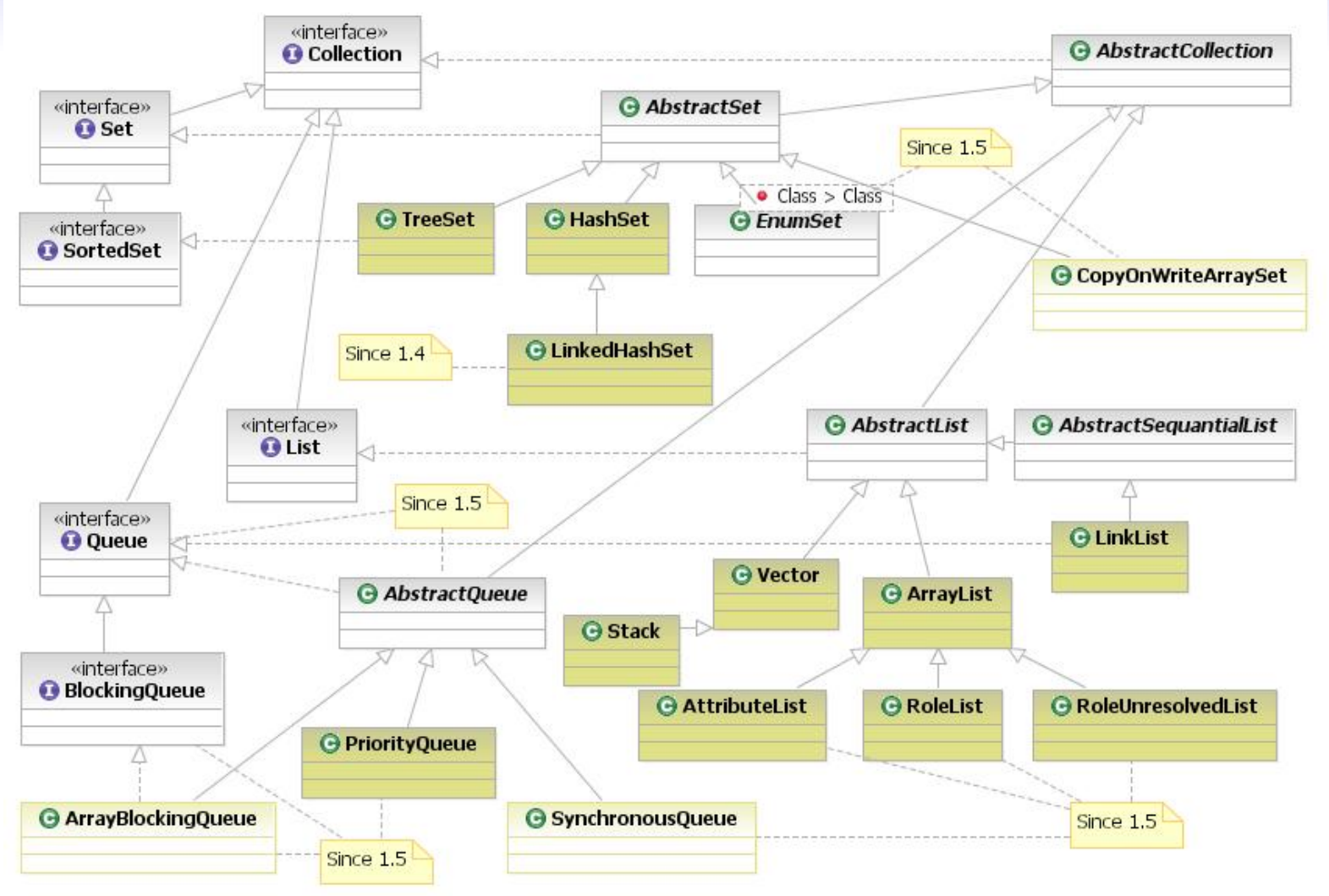
- Les classes qui implémentent les interfaces
- E.g. `ArrayList<E>`, `HashMap<K,V>`, `ArrayDeque<E>`...

Des algorithmes

- Méthodes polymorphes manipulant les collections
- E.g. `Collections.sort(List<T> list)`

Exemples : ▸ STL en C++ ▸ collections SmallTalk et hors POO ▸ Python ...

Diagramme de classes partiel



- Image tirée du Wiki book sur les collections Java
- Description complète du cadriciel par Oracle

D'ailleurs, par exemple...

Cette signature n'est pas bonne

```
public double computePerimetersSum(ArrayList<Shape> shapeList){
. . .
}
```

Il faut privilégier l'abstraction la plus élevée possible

```
public double computePerimetersSum(Collection<Shape> shapeList)

// ou, si besoin de l'index dans la collection

public double computePerimetersSum(List<Shape> shapeList)
```

⇒ `shapeList` pourra être du type `ArrayList`, `CopyOnWriteArrayList`, `LinkedList` ou `TreeSet`, etc., suivant les cas, sans avoir à toucher à cette méthode !

▸ Annexe : mon cours sur le *Collections Framework* de Java