Savoir être développeur ?

Plan

Plan

La programmation est un art

Vous êtes un artiste : *Celui, celle qui cultive un art, qui pratique un des beaux-arts*

> **Artisan qui excelle dans son travail, qui a acquis une technique, une maîtrise d'exécution, un savoir-faire qui lui permet de reproduire habilement un modèle ou même d'en inventer de nouveaux**

Nous allons donc travailler ***l'art du code propre***

L'art du code propre

Coder proprement

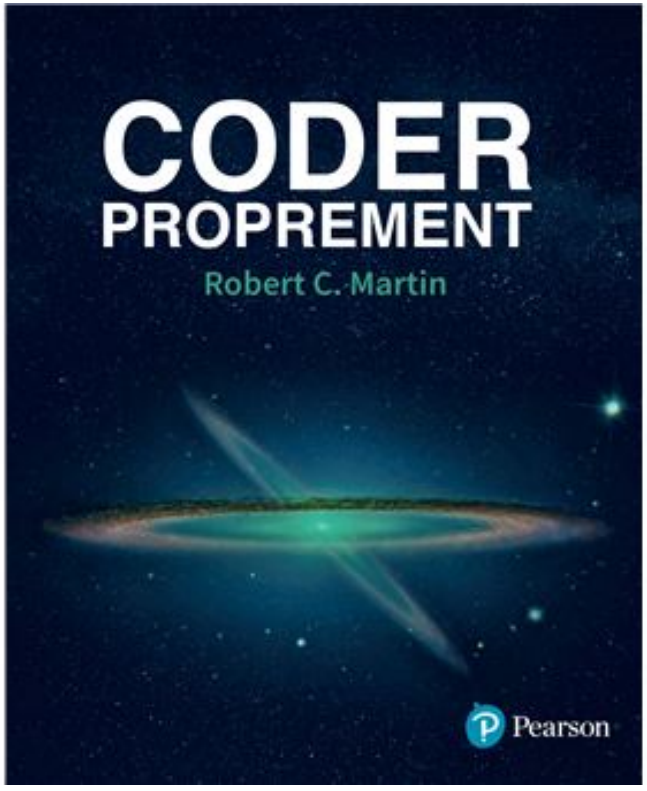
- application de règles et de techniques
- maîtrise des outils
- $\Rightarrow$ discipline au quotidien

mais c'est avant tout...

Une philosophie

Plan

Quelques citations extraites de *Coder proprement* de Robert C. Martin



Bjarne Stroustrup, inventeur du C++

« J'aime que mon code soit élégant et efficace »

- *La logique doit être simple* ⇒ lisibilité
- *Les dépendances doivent être minimes* ⇒ maintenance
- *La gestion des erreurs doit être totale* ⇒ robuste
- *Un code propre fait une chose et la fait bien* ⇒ principe SRP

Grady Booch, auteur du livre *Object Oriented Analysis and Design with Applications*

« Un code propre est un code simple et direct »

- *se lit comme une prose parfaitement écrite* ⇒ lisibilité
- *ne cache jamais les intentions du concepteur* ⇒ principe IRI
- *constitué d'abstractions nettes* ⇒ principe IRI

⇒ Il y a ici aussi l'idée qu'un code propre n'est pas commenté !

Dave Thomas, CEO de Object Technology International et à l'origine d'Eclipse

« Un code propre peut être lu et amélioré par un développeur autre que l'auteur d'origine »

- *dispose de tests unitaires* ⇒ TDD
- *utilise des noms significatifs* ⇒ lisibilité
- *dépendances minimales et explicitement définies*
- *API claire et minimale* ⇒ principe IRI
- *doit être littéraire* ⇒ lisibilité (par des humains !)

Michael Feathers, auteur de *Working Effectively with Legacy Code*

« Un code propre semble toujours avoir été écrit par quelqu'un de soigné »

- *Rien ne permet de l'améliorer de manière évidente*
- *Tout a déjà été réfléchi par l'auteur du code*

« un code laissé par quelqu'un qui se souciait énormément de son métier »

Ron Jeffries, auteur de *Extreme Programming Installed*

« Un code propre peut être lu et amélioré par un développeur autre que l'auteur d'origine »

- *passe tous les tests* ⇒ TDD
- *n'est pas redondant* ⇒ lisibilité
- *exprime toutes les idées de conception présentes dans le système* ⇒ principe IRI
- *minimise le nombre d'entités, comme les classes, les méthodes, les fonctions et assimilées*
- *doit être littéraire* ⇒ lisibilité (par des humains !)

Ward Cunningham, inventeur des wikis, co-inventeur de l'eXtreme Programming

> *« Vous savez que vous travaillez avec du code propre lorsque chaque méthode que vous lisez correspond presque parfaitement à ce que vous attendiez. Vous pouvez le qualifier de beau code lorsqu'il fait penser que le langage était adapté au problème »*

Robert C. Martin ⇒ *Coder proprement*

> ***« Nous sommes des auteurs »***

« Le rapport entre le temps passé à lire et le temps passé à écrire est bien supérieur à 10:1 »

- *Les auteurs ont des lecteurs* ⇒ lisibilité
- *la lecture du code doit être facile* ⇒ lisibilité
- *en rendant un code facile à lire, on le rend plus facile à écrire* ⇒ lisibilité
- *le code doit rester propre avec le temps*

En résumé

La qualité principale et fondamentale d'un bon développeur est de...

rendre son code lisible

notamment car, bien plus que coder

vous allez passer votre vie à lire du code

⇒ **KISS, DRY, SOC et IRI** : autant de moyens d'aller vers une lisibilité maximale : **il faut prendre le temps** de les appliquer

Plan

FTF : Respectez les conventions du langage

principe FTF : ***First Thing First***

Java

▸ Java Naming Conventions ▸ plus d'information

- une classe commence par une majuscule : `Employee`
- variable, attribut et méthode par une minuscule : `int time`
- Camel case si plus qu'un mot : `ProcessBuilder`
- constantes : `static final int MIN_WIDTH = 4;`

Python

▸ Python Naming Conventions ▸ Variable naming

- moins consistent au niveau de la communauté mais ⇒
- variable : `shopping_cart_total`
- Camel Case pour les classes

IRI ⇒ Savoir nommer les entités

IRI : ***Intention Revealing Interfaces***

On ne peut pas assez insister sur cet aspect !

Choisir des noms expressifs / éviter la désinformation

- pas de noms abstraits pour les attributs : `list` ⇒ de quoi ?
- pas d'information superflue ou de codification, e.g. :
  - `String nameString` pour un attribut
  - `IComparable` pour une interface
  - exception : `SocketImpl` OK, car l'interface `Socket` existe
  - exception : `NullPointerException`, par convention
- Noms **prononçables** et compatibles avec une recherche

Nommer classes et méthodes

À respecter toujours :

Classe

- **pas de verbe : noms ou groupes nominaux**
- éviter les mots `Data, Info, Interface...`

méthodes

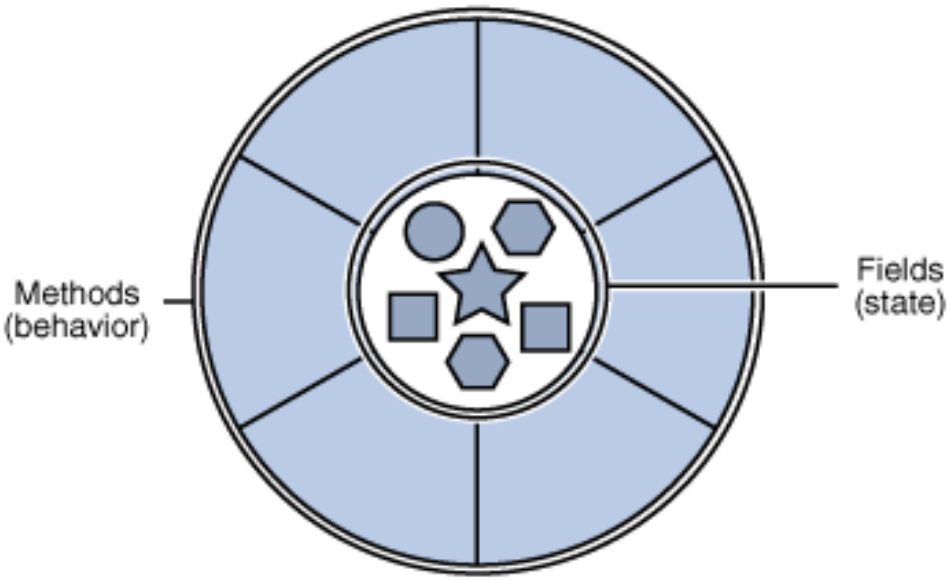
- **verbes ou groupes verbaux**
- standard JavaBean : utiliser systématiquement `get/set/is` pour les méthodes qui accèdent/modifient les attributs
- utiliser le *Factory Method Pattern* lorsqu'une instance peut être construite suivant différents constructeurs ⇒ rendez privé les constructeurs correspondants.

OK, mais une seconde...

Pourquoi on fait (comme) ça ?

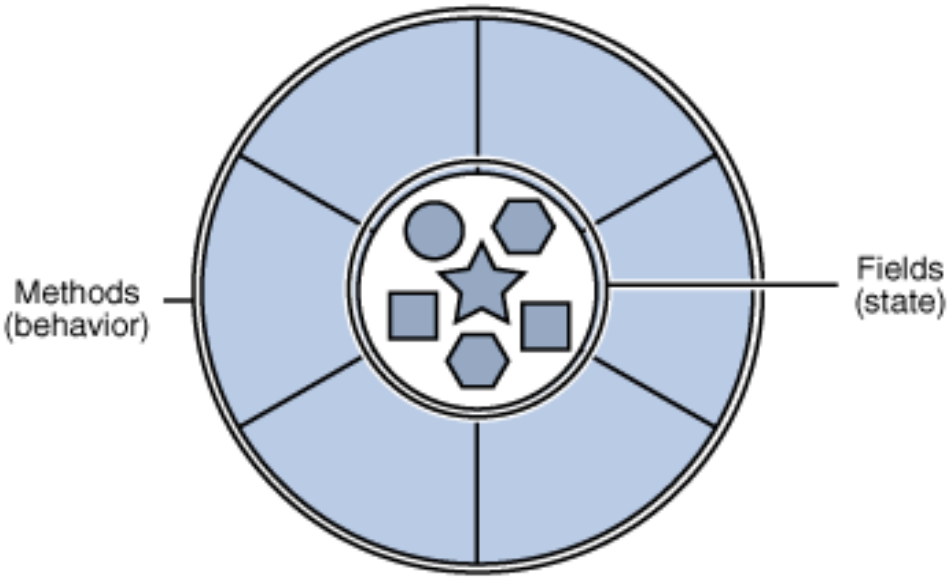
⇒ Un petit *flashback*...

Qu'est-ce qu'un objet?



- Une entité logicielle cohérente définie par son **état** et son **comportement**.
- Utilisé pour modéliser informatiquement des "objets" de la vie courante (voiture, personne, etc.) ou des concepts (date, couleur,etc.).

Qu'est-ce qu'un objet?



- Son **état** est défini à l'aide d'**attributs** (Fields) : des variables associées à l'objet.
- Son **comportement** est défini par des **méthodes** (Methods) : une fonction rattachée à l'objet qui permet de déclencher un des comportements associés à l'objet.

KISS ⇒ Ne faites pas le malin

KISS : ***Keep It Simple Stupid***

> ***You're smart, but others don't care at all!***

Évitez tout contexte culturel
Restez dans le domaine métier

- Évitez les jeux de mots ou les références culturelles...
- Utilisez des noms qui expriment votre solution, e.g. avec le nom de l'algorithme ou du pattern utilisé :
  - MVC ⇒ `PersonModel, PersonView, PersonController`

Et aussi...

L'anglais est la langue de l'informatique :

Écrivez TOUT en ANGLAIS !

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

sOlid : Open/close principle

That smells... [alert]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

[example]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

[example]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

[example]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

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

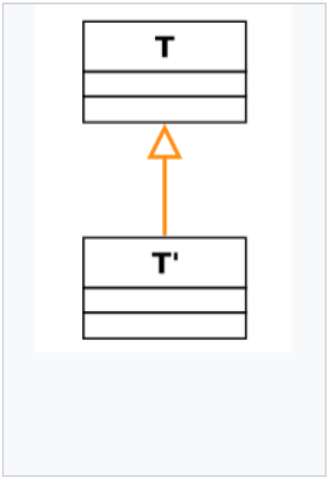
- Contravariance des arguments dans le sous-type
- Covariance du type de retour dans le sous-type.
- pas de nouvelle exception, sauf si elles sont d'un sous-type de l'originale

soLid : Liskov's Substitution Principle (LSP)

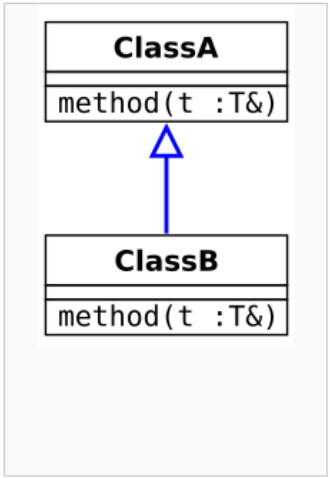
Variance en POO

▸ Wikipédia

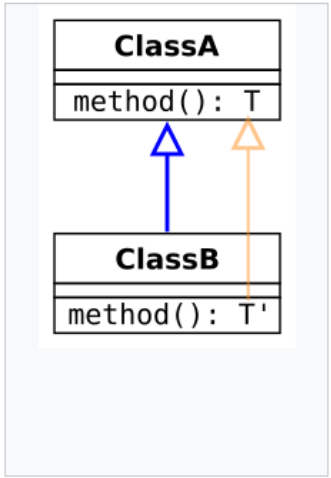
Variance and method overriding: overview



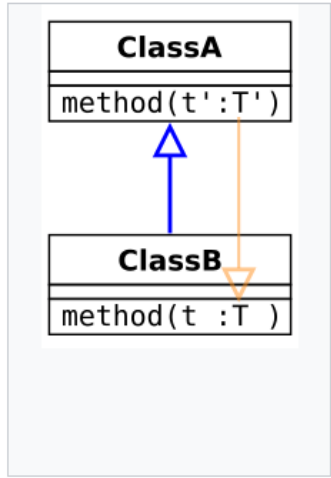
Subtyping of the parameter/return type of the method.



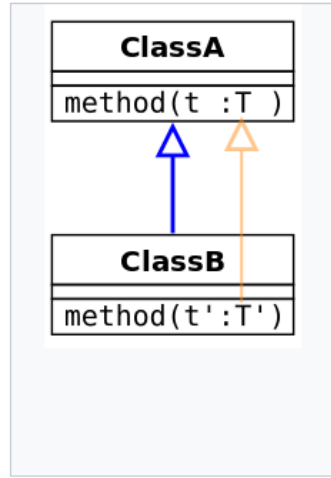
Invariance. The signature of the overriding method is unchanged.



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.

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**

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

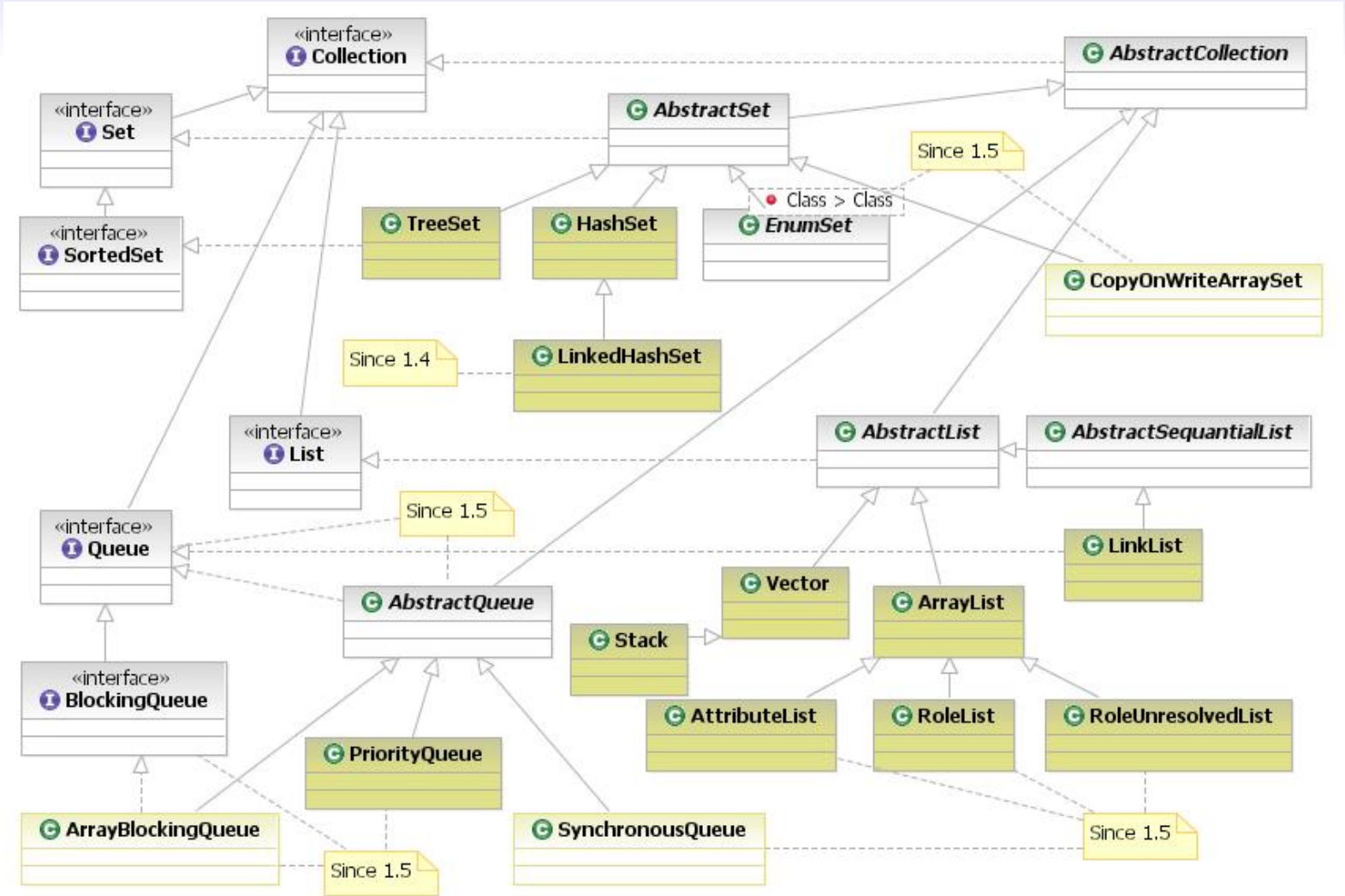
- Les classes qui implémentent les interfaces
- E.g. `ArrayList<E>`, `HashMap<K,V>`, `ArrayDeque<E>`...

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...

[alert]Cette signature n'est pas bonne

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

[example]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

Plan

Les commentaires dans le code

Comment commenter son code ?

Ce que dit la théorie :

N'en mettez JAMAIS !

Pourquoi

- un commentaire indique un code mal écrit
- ça rend le code moins lisible
- ça vieillit mal avec le temps et peut devenir trompeur
- commenter un code sale est tentant ⇒ il faut le reprendre !

Mais...

Ne surtout pas confondre

commentaire
et
documentation

Documenter son code est FONDAMENTAL !
Que serait Java sans la JavaDoc ?
La théorie dit de ne pas documenter l'évidence, e.g. les accesseurs.
OK mais une JavaDoc trouée, c'est pas génial ⇒ commenter systématiquement vos packages, classes et méthodes visibles depuis l'extérieur !

Autres exemples de documentation

Documentation obligatoire

- La licence associée à votre code : en début de fichier
- Les README dans les dépôts
- Les notices utilisateurs
- Les messages de commit

Il faut y mettre le même soin que pour le code

Car c'est du code ! ⇒ Maintenabilité

Plan

Mise en forme du code source

Les IDE en font une partie mais, sachant qu'un fichier source se lit de haut en bas, et de gauche à droite :

Mise en forme verticale

- Distance verticale : les concepts proches ne sont pas éloignés, e.g. fonctions dépendantes
- Variables locales : déclarées au plus près de l'utilisation
- Variables d'instance : en début de classe
- Fonctions dépendantes : l'appelante avant l'appelée

Mise en forme horizontale (densité)

- Largeur d'une ligne : pas plus 120 caractères
- Opérateurs d'affectation entourés d'espaces `int a = 10`
- Paramètres : espace après la virgule `(double a, double b)`
- Illustrer la précédence des opérateurs `b*b - 4*a*c`

Organisation du code, e.g. en package Java

Imaginez les 4000+ classes de Java sans les packages...

Organiser son code en fonction de la logique métier est crucial

Un package est définit par

- un regroupement de classes conceptuellement cohérent par rapport à leurs fonctionnalités (ex : `java.io` `java.util` `java.time`)
- un espace de nommage (`java.util.List` $\neq$ `java.awt.List`)
- une visibilité particulière : permet de ne pas polluer l'API

Remarque : les modules introduits en Java 9 permettent un niveau supplémentaire d'encapsulation

Plan

Tests unitaires et TDD

Extrait de *coder proprement* : *les trois lois du TDD*

1. Vous ne devez pas écrire un code de production tant que vous n'avez pas écrit un test unitaire d'échec
2. Vous devez uniquement écrire le test unitaire suffisant pour échouer
3. Vous devez uniquement écrire le code de production suffisant pour réussir le test d'échec courant

Ce qui veut dire, pour les tests unitaires

1. Écrire un test d'échec à partir de la spécification de la méthode
2. Le code du test est minimal et **comporte un seul `assert`**
3. le code de production ⇒ réussir le test, mais pas plus ⇒ cela pour assurer une couverture des tests maximale

Maintenir les tests est fondamental

Le code de test est aussi important que le code de production, voire plus

Pourquoi ?

- Les tests permettent de faire évoluer une application sans crainte
- $\Rightarrow$ sans tests, chaque modification peut être source de bugs qui seront détectés trop tard (après plusieurs modifications)
- tests $\Rightarrow$ code flexible, maintenable et réutilisable

Principales qualités d'un code de test ?

LISIBILITÉ

Il faut apporter le même soin qu'au code de production

- ⇒ un nommage bien pensé et cohérent de toutes les méthodes de test
- ⇒ une mise en forme soignée
- ⇒ l'utilisation d'un framework dédié ⇒ Junit, TestNG, ...

Stratégie de nommage des méthodes de test

Il existe de nombreuses stratégies, exemples :

1. `testMethodName` : préfixer les méthodes avec le mot test
   ⇒ A fait son temps. . . le mot test est inutile / pas de stratégie claire
2. `method_Should_ExpectedBehavior_When_StateUnderTest`
   `repeat_Should_ThrowIllegalArgumentException_When_NegativeValueProvided`
3. `methodName_stateUnderTest_ExpectedBehavior` ou
   `methodName_ExpectedBehavior_stateUnderTest`
   ⇒ `repeat_NegativeValue_IllegalArgumentException~~Thrown~~`
4. Style langage Gherkin : *Given, When, and Then* adapté au *behavior-driven development* (BDD)
   `given_EmptyEmployeesList_when_ComputeEmployeesSalariesSum_then_return0`

Le choix dépend du contexte : 2 et 3 sont mieux adaptés aux tests unitaires, tandis que 4 est plus adapté aux tests d'intégration, ainsi qu'aux démarches où des non développeurs participent à leur définition. Le principal est de rester cohérent !

Plan

TOUT EST CODE !

Si vous y réfléchissez, certains des principes vus ici peuvent s'appliquer partout en informatique :

Notamment le DevSecOps consiste à transformer les tâches en code pour automatiser... tout

- Les fichiers de build (Maven / Gradle), c'est du code
- Un pipeline de CI/CD, c'est du code
- On parle de IaC *Infrastructure as code* : fichiers de configuration
- ⇒ **Tous ces fichiers sont du code et méritent le même soin**

Et au-delà

- vos fichiers de configuration de session
- les noms utilisés dans vos arborescences : dossiers / fichiers / mails
- les raccourcis de votre navigateur
- vos requêtes dans les moteurs de recherche
- ...

Deux méta-principes importants

TSAE

This Should Already Exist

- Vous ne coderez quasiment jamais d'algorithmes : ils existent déjà !
- Votre travail consiste à utiliser et assembler des briques et outils logiciels élégamment
- Avant de vous lancer dans le code...
  - Identifiez clairement le problème que vous voulez résoudre
  - Cherchez si une classe/framework/librairie ne fait pas le job (99% des cas)
  - Vos n'avez pas trouvé ?
  - Cherchez encore !
- Seule la logique métier doit être source de code nouveau

Deux méta-principes importants

RTFM

Read The Fucking Manual

- Il faut vous l'appliquer à vous ! (pas aux autres)
- Google / Stackoverflow / IA . . . sont vos ennemis en phase d'apprentissage
- Ne googler pas, **lisez les documentations**
  ⇒ Apprentissage, compréhension, maîtrise, solutions
- Vous devez **passer plus de temps à lire de la doc qu'à coder**
  ⇒ Vous utilisez une classe Java, lisez TOUT le résumé de sa JavaDoc
- Les gains de temps sur le long terme sont énormes
  ⇒ **Vous serez un meilleur développeur**

Quid de l'IA ?

L'utilisez-vous pour développer ?

Franchement, c'est juste incroyable !
Et il va devenir impossible de s'en passer

Mais il faut l'utiliser avec discernement

- Si vous l'utilisez en mode flemme ⇒ alors vous êtes terminé !
  - comme pour les *quick fix* des IDE, si vous n'êtes pas en mesure de comprendre la solution, ça peut virer à la catastrophe
  - **vous ne prendrez pas l'habitude de lire de la documentation**

Vos qualités de développeurs stagneront !!

Conclusion

Le métier de développeur

- **la programmation est un art**, une philosophie
  ⇒ apprentissage / connaissances techniques / rigueur et soin
- vous allez **passer votre vie à lire du code**
  ⇒ **un code propre est un code lisible** (utilisez l'anglais)
  ⇒ nommage, KISS, DRY, SOC, IRI...
- Il faudra refactorer, refactorer et refactorer !
- TDD ⇒ Seuls les tests permettent de refactorer en toute quiétude

Méta-règles

- **TSAE** ⇒ Quel est le problème ? Existe-t-il une solution (99%) ?
- **FTF** ⇒ Maîtriser les principes sous-jacents à la solution technique
- **RTFM** ⇒ Lisez de la doc / Lisez de la doc / Lisez de la doc
- ***AI to the rescue?*** ⇒ Ne vous laissez pas transformer en légume