



make it clever



**Software Craftsmanship**  
**Devoxx 2023**



- **Développeur passionné**
  - Plusieurs projets Open-Source
  - Qualité Logicielle
    - Qualité du code
    - Intégration continue
  - [cmarchand@oxiane.com](mailto:cmarchand@oxiane.com)
- **Formateur**
  - Java
  - Informatique documentaire
  - Software Craftsmanship
- **Intervenant dans conférences internationales**
  - XML Prague
  - Markup UK



<https://twitter.com/JosePaumard>



<https://github.com/JosePaumard>



<https://www.youtube.com/hashtag/jepcafe>

<https://www.youtube.com/c/JosePaumard01>

<https://www.youtube.com/c/coursenlignejava>



<https://fr.slideshare.net/jpaumard>



<https://www.pluralsight.com/authors/jose-paumard>



<https://www.jchateau.org/>

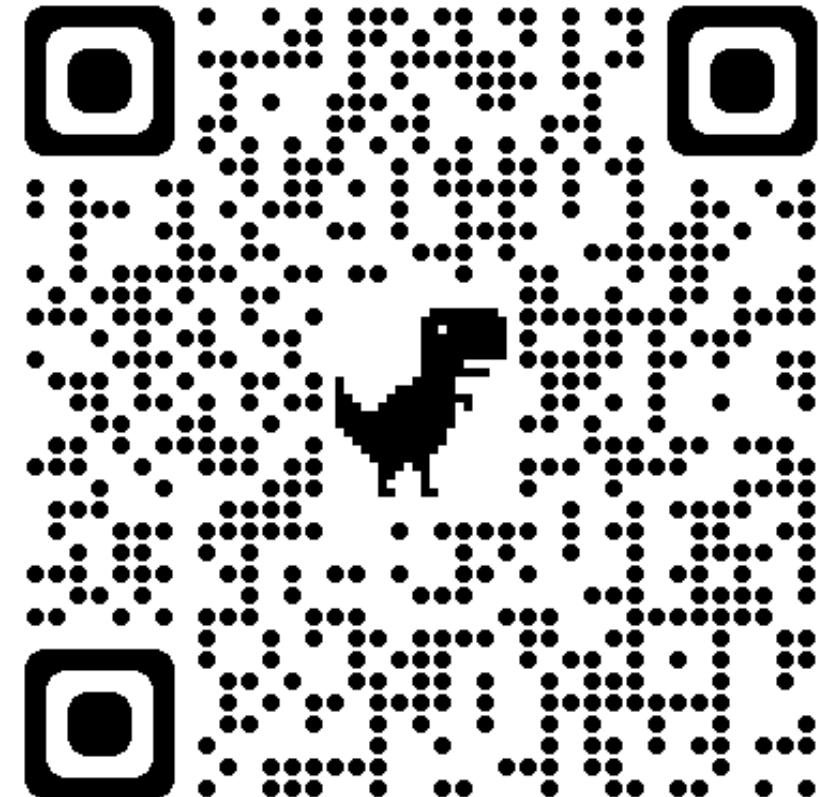
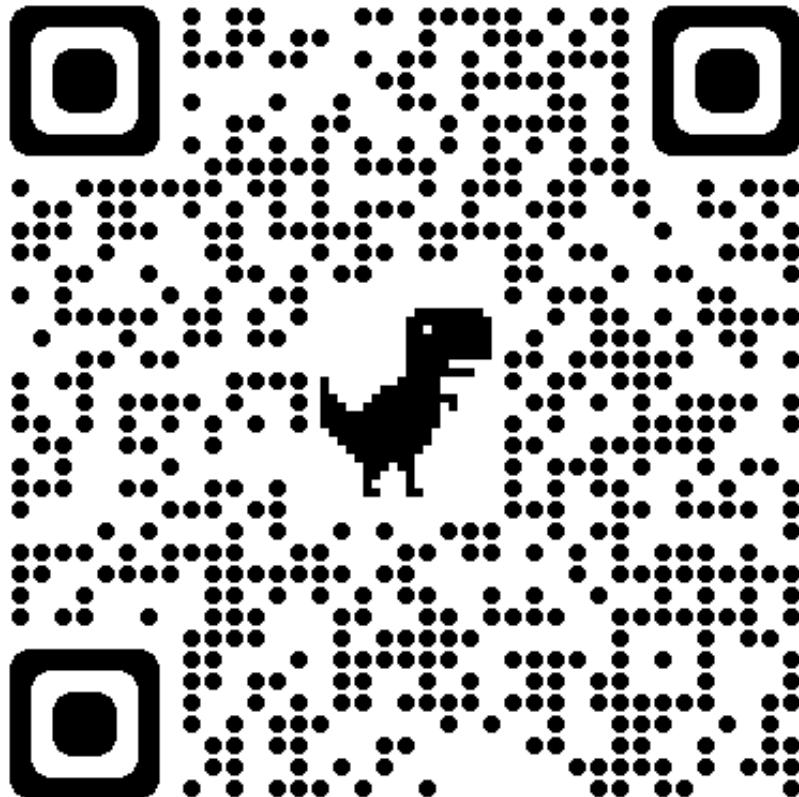


# Préparation des Katas

- **Repository pour les katas**

- <https://gitlab.com/oxiane-institut/craft-devoxx-2023>

- <https://github.com/cmarchand/craft-devoxx-2023>



# Software Craftsmanship Pourquoi ?

- **OOPSLA'92**
  - Retour d'expérience sur une application de gestion de portefeuille



**Addendum  
to the  
Proceedings**

*Experience Report—  
The WyCash Portfolio Management System*

**Report by:**  
Ward Cunningham  
Cunningham & Cunningham, Inc.

U.S. pension funds, corporations, and banks invest billions of dollars in the “cash” markets. Cash securities are generally considered those with a knowledge of all aspects of the roughly four megabytes of source code. This includes some libraries provided by the vendor and others written

- **Introduit la notion de dette technique**
  - La dette technique accélère le développement tant qu'elle est remboursée par une réécriture

- La dette technique doit être remboursée

The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt.

- Le remboursement n'est possible que si le code est propre !
  - On peut alors refactorer le code, et le simplifier

new architecture as time became available. The ultimate removal of the immature architecture would leave us with a program that had been simplified in the course of adding features—a truly enviable situation.

- **Reprend la métaphore de la dette technique en 2003**
  - L'applique au code mal écrit
- **Dette technique**
  - Le code que l'on livre et dont on sait que l'on va revenir dessus



- **Reprend la métaphore de la dette technique en 2003**
  - L'applique au code mal écrit
- **Pourquoi ?**
  - Parce que le code est illisible
  - Parce qu'il est non testé



- **Après 10 ans d'agilité :**

- Le cycle SCRUM se grippe au bout de 6 mois
- La vitesse diminue
- Plus de livraison de code fonctionnel

- **Pourquoi ?**

- Accumulation de dette technique

- **L'agilité sans technique de développement...**

- Extreme programming, Pair programming, Simple Design
- Clean code, Refactoring, Test Driven Development

- **... ne peut pas fonctionner !**

- On passe son temps à "payer les intérêts" de la dette

- **Il faut donc racheter la dette !**



- **Assimilé à tort à un ensemble de techniques et de méthodes**
  - TDD, Clean Code
  - Extreme Programming, Agilité
- **Il s'agit en fait de la définition d'une attitude à tenir**
  - En tant que développeur
- **On veut bannir la création de dette "insouciante"**
  - Attitude vis-à-vis de la qualité du code
- **Mise en application des règles du "clean code"**
  - Sur les nouveaux développements
  - Sur le code existant (legacy)
- **Ecriture systématique de tests = adoption du TDD**
- **Application des règles du refactoring**

# Priorités du développement

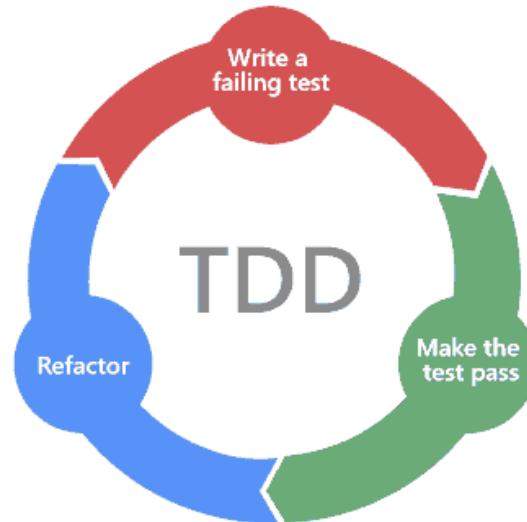
## Lisibilité

- Optimiser la lisibilité
- Constat : on passe 10x plus de temps à lire qu'à écrire du code
- Il faut donc optimiser la lisibilité du code
  - Passer 4x plus de temps à écrire du code 2x plus lisible permet de gagner 20% de temps de travail
    - $1h + 10h = 11h$
    - $4h + 5h = 9h$

- **Test First !**
  - Pourquoi cette pratique est-elle préconisée ?
- **Parce qu'elle permet d'exprimer ce que le code doit faire**
  - En regardant les tests, on comprend ce que fait le code
  - Donc on sait comment le modifier
  - Quand un test échoue, on sait ce qui ne marche plus dans l'application

# Priorités du développement

## Test Driven Development



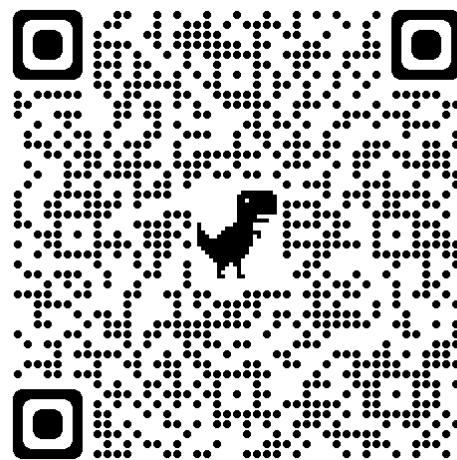
- Pas de code de production sans avoir auparavant écrit un test unitaire qui échoue
- Pas plus d'un test unitaire qui soit suffisant pour échouer, et une erreur de compilation est un échec
- Pas plus de code de production que nécessaire pour faire passer le test unitaire qui échouait
- Quand tous les tests unitaires passent, refactorer le code de production pour respecter les règles du clean code

- **Fizz Buzz Kata !**

- On compte de 1 à 100
- Quand le nombre est divisible par 3, on dit **Fizz**
- Quand le nombre est divisible par 5, on dit **Buzz**
- Quand le nombre est divisible par 3 et par 5, on dit **FizzBuzz**
- Dans les autres cas, on dit le nombre

- **Ecrire la fonction qui détermine ce qui doit être dit**

- Appliquer strictement le TDD



Kata

# Gherkin / Cucumber

- **Les tests unitaires ne garantissent pas le respect du cas métier**
  - Un test unitaire mal écrit est toxique
  - Il garantit que ce qui n'est pas voulu par le métier fonctionne correctement
- **Le Behavior Driven Development fait écrire les tests par le métier**
  - Dans un langage proche du langage naturel
  - Avec les termes du métier
  - Dans la langue des utilisateurs
- **Gherkin est le langage de description des règles métier**

Feature: The invoice service can compute the VAT

For each product, a VAT is computed, as a percentage of the price of the product.  
The VAT of each product is then added to the price of that product.

Scenario: Compute VAT

Given The amount of the invoice is 1000

And the VAT rate is 15%

When The total amount is computed

Then The result is 1150

- **Les tests unitaires ne garantissent pas le respect du cas métier**
  - Un test unitaire mal écrit est toxique
  - Il garantit que ce qui n'est pas voulu par le métier fonctionne correctement
- **Le Behavior Driven Development fait écrire les tests par le métier**
  - Dans un langage proche du langage naturel
  - Avec les termes du métier
  - Dans la langue des utilisateurs
- **Gherkin est le langage de description des règles métier**

```
# language: fr
```

```
Fonctionnalité: Le service de commande peut calculer la TVA
```

```
Pour chaque produit, une TVA est calculée, comme un pourcentage du prix du produit. La TVA de chaque produit est ensuite ajoutée au prix du produit.
```

```
Scénario: Calculer la TVA
```

```
Etant donné que Le montant de la commande est 1000
```

```
Et que le taux de TVA est 15%
```

```
Lorsque Le montant total est calculé
```

```
Alors Le résultat est 1150
```

- **Il est très simple de généraliser les scénarios par des paramètres**
  - Avec des templates de scenario

Feature: The invoice service can compute the VAT

For each product, a VAT is computed, as a percentage of the price of the product.  
The VAT of each product is then added to the price of that product.

Scenario Template: Compute VAT

Given The amount of the invoice is <invoice\_amount>

And the VAT rate is <vat\_rate>

When The total amount is computed

Then The result is <result>

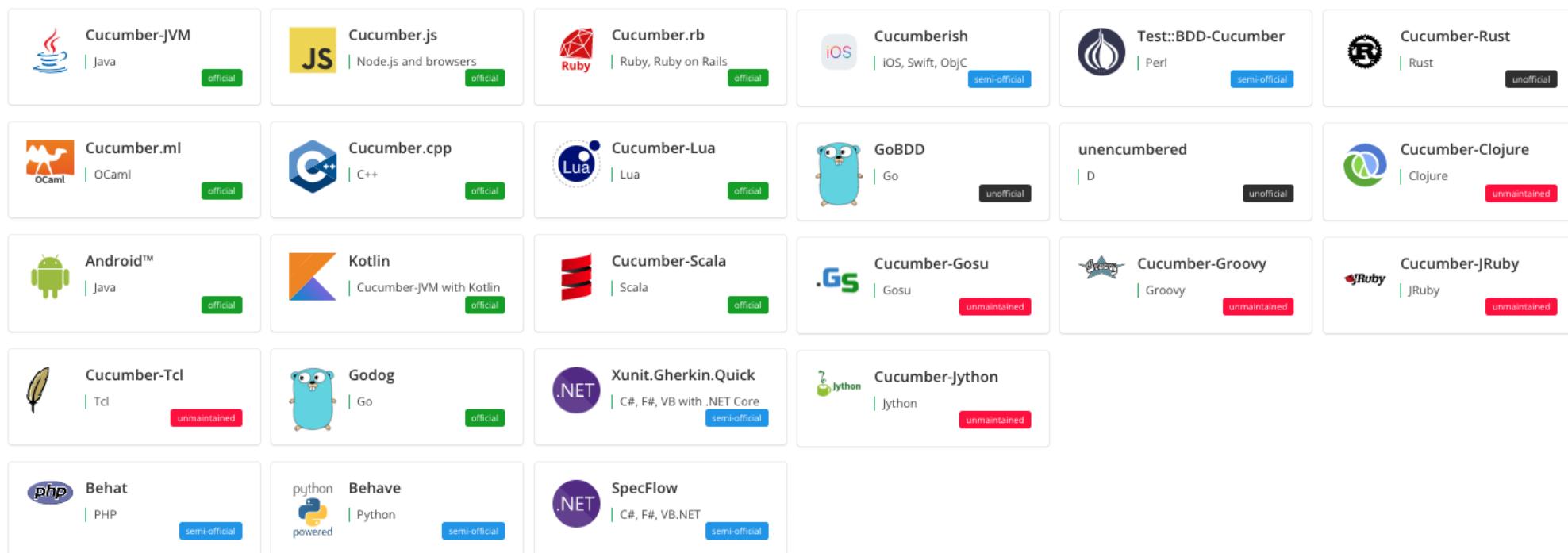
Examples:

invoice_amount	vat_rate	result
1000	15%	1150
1000	18%	1180
1000	20%	1200
1500	15%	1225
1500	18%	1270
1500	20%	1300

- Et donc d'écrire de très nombreux exemples !
- Voir de décrire tous les cas métiers

# Simplifier l'écriture des tests Cucumber

- Cucumber permet de rendre exécutables ces descriptions
  - Et de transformer le besoin métier en test
- Il existe de nombreuses implémentations de Cucumber
  - Une par langage de programmation
  - ... ou presque ...



- **Il suffit d'écrire une glue ou une Step Definition**
  - Faire le lien entre la feature et le code à tester
  - Basée sur des regex

```
Given The amount of the invoice is <invoice_amount>
And the VAT rate is <vat_rate>%
When The total amount is computed
Then The result is <result>
```

```
public class VATServiceStepDefs {
    @Given("The amount of the invoice is {float}")
    public void invoice_amount_is(float invoiceAmount) { }
    @Given("the VAT rate is {float}%")
    public void vat_rate_is(float rate) { }

    @When("The total amount is computed")
    public void total_amount_calculated() {
        actual = VATService.calculate(invoiceAmount, rate);
    }

    @Then("The result is {float}")
    public void theResultIs(float expected) {
        Assertions.assertThat(actual).isEqualTo(expected);
    }
}
```

Une instance par test

La classe peut être stateful

Le code à tester

Les assertions à faire

- Utiliser des tournures de phrases qui permettent le mapping
  - Sans ambiguïté entre les différentes propositions

```
Given The amount is 100
And the VAT rate is 15%
When We calculate VAT
Then The amount is 115
```

Collision entre les expressions  
Given et Then

Les annotations @Given et @Then ne sont  
pas discriminantes

```
Given The amount is 100
And the VAT rate is 15%
When We calculate VAT
Then The taxes-included amount is 115
```

- **La majorité des IDE supportent Gherkin / Cucumber**
  - Complétion, coloration syntaxique, exécution, etc...
- **Cucumber s'interface avec les outils de test habituels**
  - **JUnit** pour Java
  - **npm test** pour Node JS
  - ...
- **Il est donc simple d'intégrer Cucumber dans la suite de tests**
  - Et d'automatiser l'exécution des tests Cucumber
  - Y compris dans la CI

- **Kata ! Roman Numerals**

- Ecrire une fonction qui convertit un nombre d'écriture arabe en écriture romaine

- **Règles**

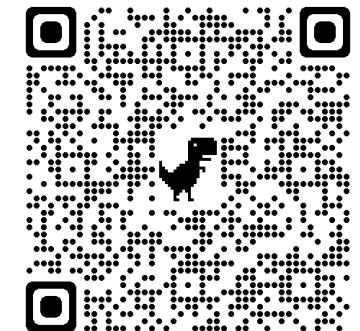
- **I** -> 1
- **V** -> 5
- **X** -> 10
- **L** -> 50

- **C** -> 100
- **D** -> 500
- **M** -> 1000

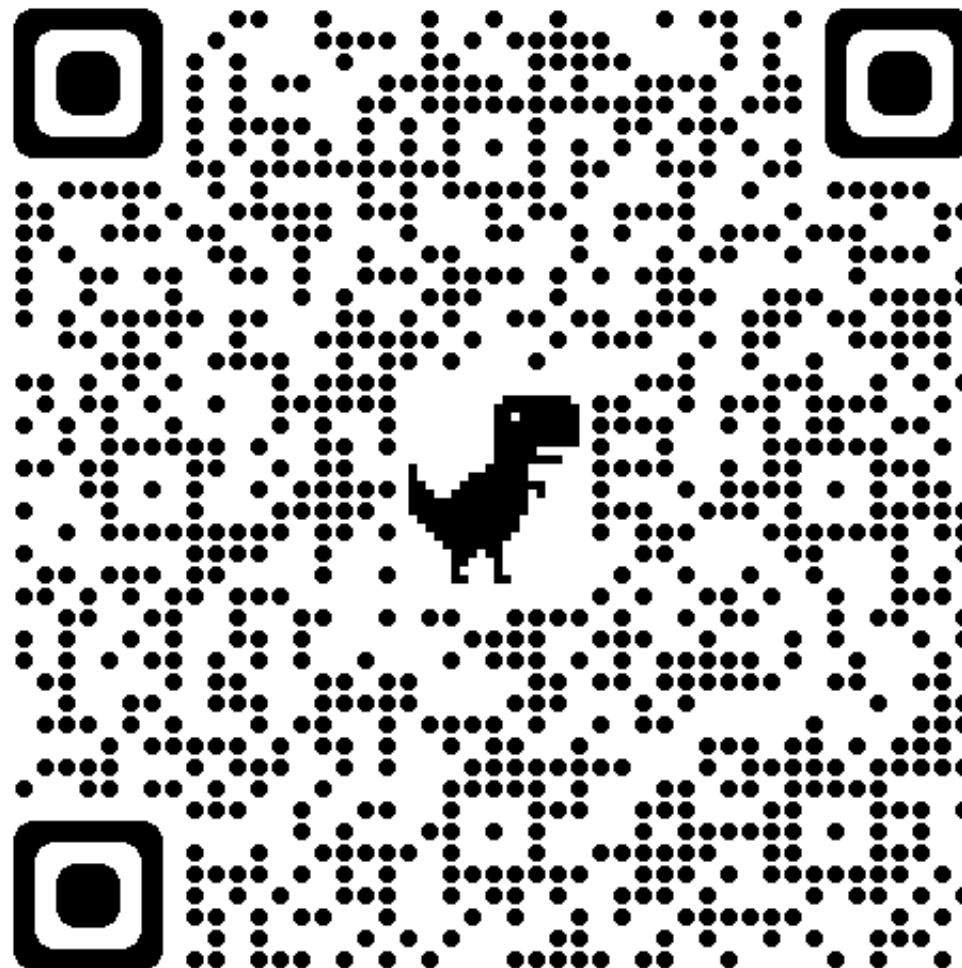
- Aucun nombre supérieur ou égal à 4000
- **I**, **X**, **C** et **M** peuvent être répétés 3 fois à la suite, pas plus
- **V**, **L**, **D** ne peuvent jamais être répétés
- **I** peut être soustrait de **V** et de **X** => **IV**, **IX**
- **X** peut être soustrait de **D** et **M** => **XD**, **XM**

- **Instructions**

- Utiliser Gherkin / Cucumber
- Ecrire tous les cas de test [1;4000[, utiliser un tableur...



Kata



Kata

# Software Craftsmanship

- **Maîtriser en permanence le code développé**
  - Le code livré est systématiquement testé
  - La connaissance du code est partagée
  - Le code est conforme aux pratiques du clean code

- **Nouveaux développements suivant les bons principes**
  - Suivi des Design Patterns
  - Principes SOLID
  - Bannir la sur-ingénierie et la sur-spécification

- **Intervention sur le code legacy : Boy Scout Rule**
  - On laisse le code plus propre que... (rachat de la dette technique)
  - On teste systématiquement le code sur lequel on intervient
  - On refactorise dans les principes du Clean Code

- La bonne pratique n°1 :

**S'entraîner en permanence aux bonnes pratiques**



- Kata !

- **L'entraînement passe par les Katas**
  - Il en existe de nombreux, sur différents sujets
- **Qu'est-ce qu'un Kata ?**
  - Un exercice, plus ou moins simple, que l'on fait et refait
  - En groupe (Coding Dojo), en duo ou en solo, de 60 à 90 minutes
- **Idée : pratiquer des exercices en groupe**
  - Pour s'exercer à l'application de méthodes, de patterns, etc...
- **Moyen : dialoguer, communiquer, échanger, découvrir**

- La plus mauvaise idée que l'on puisse avoir :

## Écrire du code « réutilisable »

- On ne réutilise jamais le code réutilisable...
- Inutile d'y passer du temps
- Inutile (et toxique) de designer un code pour qu'il soit réutilisable
  - Surtout après coup

- La meilleure idée que l'on puisse avoir :

## Écrire du code « jetable »

- Découplé du code qui l'appelle
- Occupe un périmètre bien délimité
- Dépend peu de l'extérieur
- **Le code ne convient plus ? On le remplace par un autre**

# Principes SOLID

- A quoi servent les principes SOLID ?

**À simplifier les évolutions du code**

- A produire à coût constant
- **Un code qui ne change jamais a-t-il besoin de les respecter ?**
- **Exemple : la classe `java.lang.String`**

- **SOLID !**
- **S = Single Responsability Principle**
  - Bob Martin
  - On ne doit avoir qu'une raison de modifier une classe / méthode
  - « Mutualiser » du code est une mauvaise idée



```
public class Employee {  
  
    public double calculatePay() {  
        ...  
    }  
  
    public PDFReport getReport() {  
        ...  
    }  
  
    public boolean persist() {  
        ...  
    }  
}
```

# Single Responsibility Principle

```
public void promoEmailCampaign() {  
    List<User> users = getUsers();  
    sendPromoEmails(users);  
    updateUsers(users);  
}
```

```
public void greetingsEmailCampaign() {  
    List<User> users = getUsers();  
    sendGreetingsEmails(users);  
    updateUsers(users);  
}
```

```
public void emailCampaign(boolean promo) {  
    List<User> users = getUsers();  
    if (promo)  
        sendPromoEmails(users);  
    else  
        sendGreetingsEmails(users);  
    updateUsers(users);  
}
```

- **Quand se rend-on compte que l'on viole le principe S ?**
- **Quand le nommage d'une méthode est complexe**
- **Quand une méthode métier prend des paramètres techniques**
  - De configuration, de contexte, etc...
- **Quand l'ajout d'une fonctionnalité entraîne la modification de plusieurs méthodes**
  - Et de leurs signatures

- **SOLID !**
- **S = Single Responsibility Principle**
- **O = Open Closed (Bertrand Meyer)**
  - Ouvert à l'extension
  - Fermé à la modification
  - On peut modifier le comportement d'une application sans la modifier, par extension du code
  - Attention : on préfère la composition à l'héritage



# Open Closed Principle

```
if (shape instanceof Circle) {  
    processCircle(shape);  
} else if (shape instanceof Square) {  
    processSquare(shape);  
} else if (shape instanceof Rectangle) {  
    processRectangle(shape);  
} else {  
    processOther(shape);  
}
```

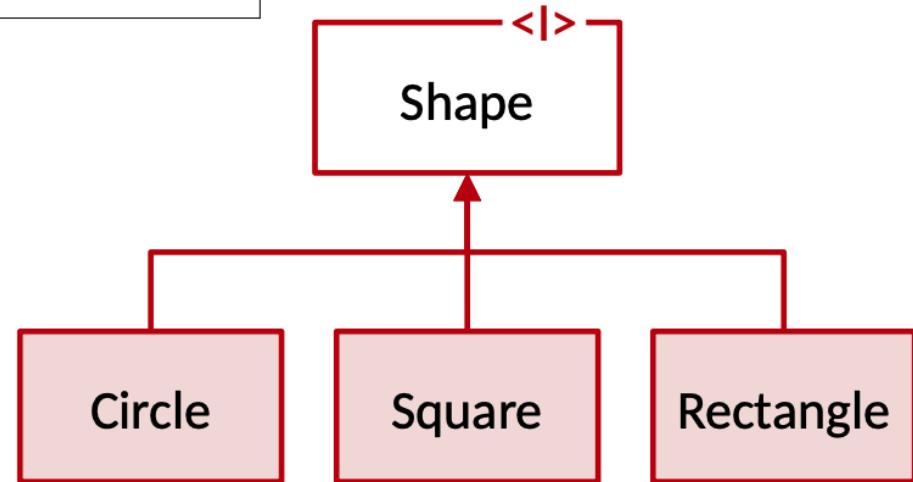
Circle

Square

Rectangle

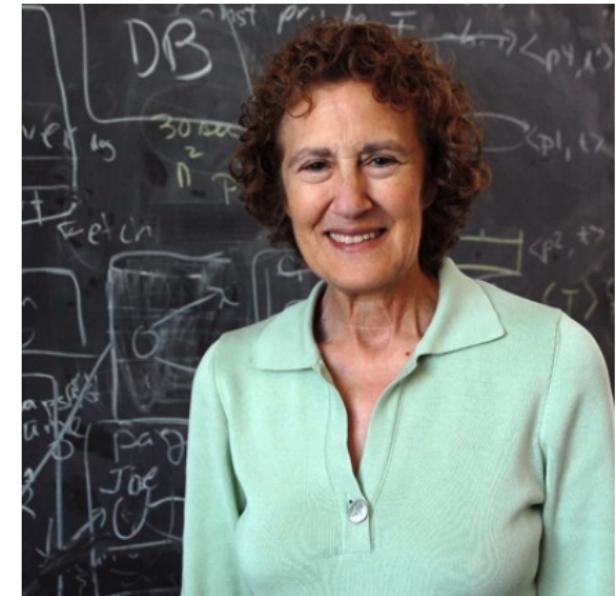
# Open Closed Principle

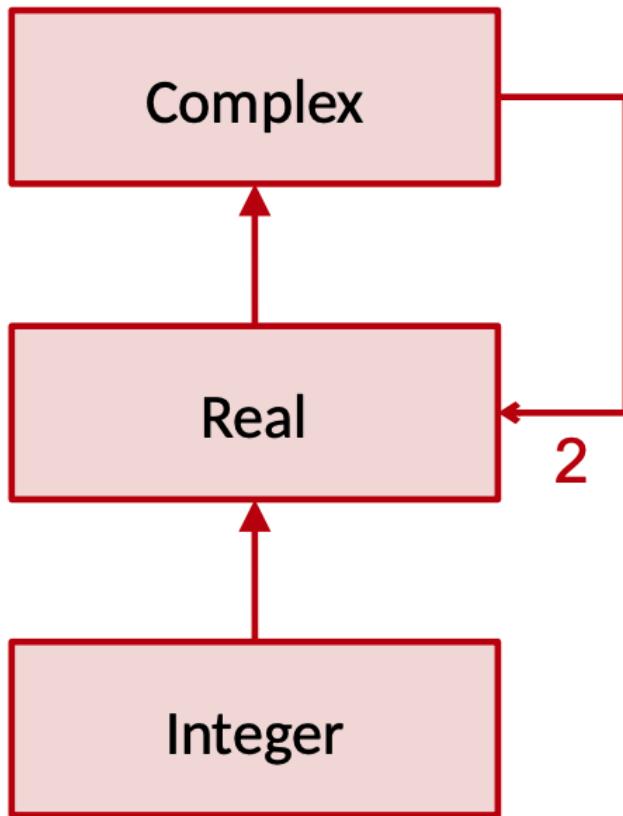
```
ShapeProcessor processor = Shape.processor(...);  
processor.process(shape);
```



- **Quand se rend-on compte que l'on viole l'OCP ?**
- Quand l'ajout d'un cas d'utilisation impose le changement du code client

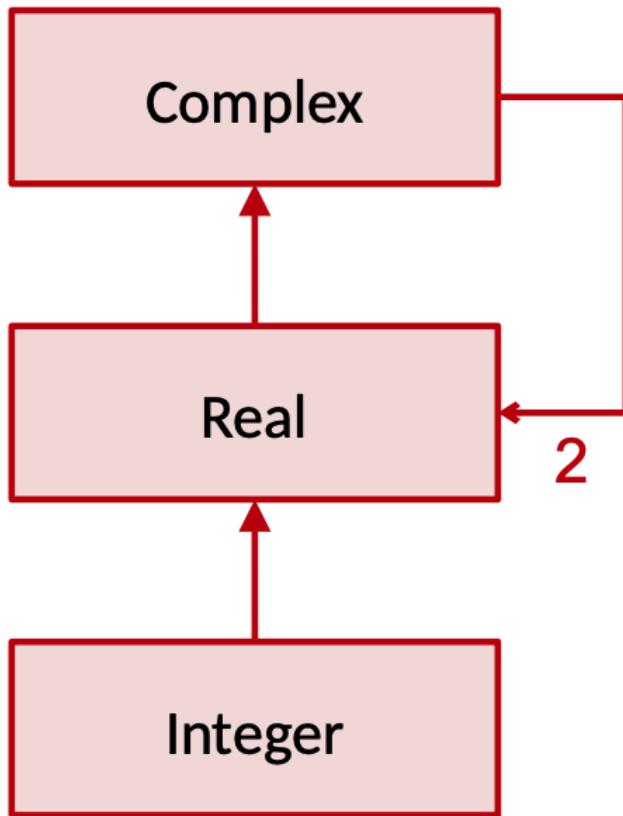
- **SOLID !**
- **S = Single Responsability Principle**
- **O = Open Closed Principle**
- **L = Liskov Substitution**
  - Un programme doit pouvoir utiliser une extension d'une classe sans constater de différence avec la classe de base





```
public class Complex {  
  
    private Real re, im;  
  
    public Complex(double re, double im) {  
        this.re = new Real(re);  
        this.im = new Real(im);  
    }  
}  
  
public class Real extends Complex {  
  
    public Real(double re) {  
        super(re, 0d);  
    }  
}
```

```
Complex c = new Complex(1d, 0d);
```



```

public class Complex {

    private Real re, im;

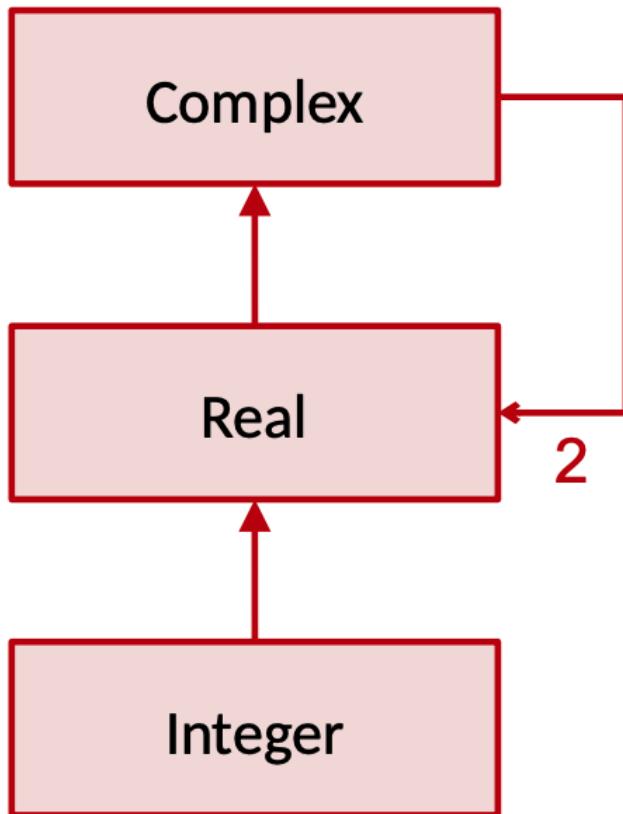
    public Complex(double re, double im) {
        this.re = new Real(re);
        this.im = new Real(im);
    }
}

public class Real extends Complex {

    public Real(double re) {
        super(re, 0d);
    }
}
  
```

```

Exception in thread "main" java.lang.StackOverflowError
at org.paumard.numbers.Complex.<init>(Complex.java:14)
at org.paumard.numbers.Real.<init>(Real.java:6)
at org.paumard.numbers.Complex.<init>(Complex.java:9)
at org.paumard.numbers.Complex.<init>(Complex.java:14)
at org.paumard.numbers.Real.<init>(Real.java:6)
  
```



- **est ou is a ne dénote pas l'héritage**
- **La proposition qui représente l'héritage est se comporte comme**
- **Cela vient du principe de Liskov**

- **Quand se rend-on compte que l'on viole le LSP ?**

- Lorsque les méthodes de l'extension violent le contrat de la super classe
- Lorsque ces méthodes ont des implémentations vides, ou qui jettent des exceptions
- Lorsque des méthodes des super classes doivent tester le type de l'objet dans lequel elles sont
  - C'est aussi une violation de l'OCP

- **Quand se rend-on compte que l'on viole le LSP ?**

```
public interface Iterator<E> {  
  
    boolean hasNext();  
  
    E next();  
  
    void remove();  
  
}
```

- **Que se passe-t-il pour les collections immutables ?**

- **SOLID !**
- **S = Single Responsability Principle**
- **O = Open Closed Principle**
- **L = Liskov Substitution**
- **I = Interface Segregation**
  - Petit nombre de méthodes dans les interfaces

```
public class InvoiceProcess {  
  
    private VATService vatService;  
  
    public InvoiceProcess(VATService vatService) {  
        this.vatService = vatService;  
    }  
  
    public double computeVAT(double price, String countryCode) {  
        return price*(1 + vatService.getVATRate(countryCode)/100);  
    }  
}
```

```
public interface VATService {  
  
    double getVATRate(String countryCode);  
    double getVATRate(CountryCode countryCode);  
  
    CountryCode getCountryCode(String countryCode);  
}
```

- **Une classe ne dépend d'autres modules qu'au travers d'interfaces**
  - Ces interfaces ne doivent pas comporter de méthodes non utilisées par cette classe
- **Pourquoi ?**
  - Pour ne pas avoir à recompiler cette classe sans raison !

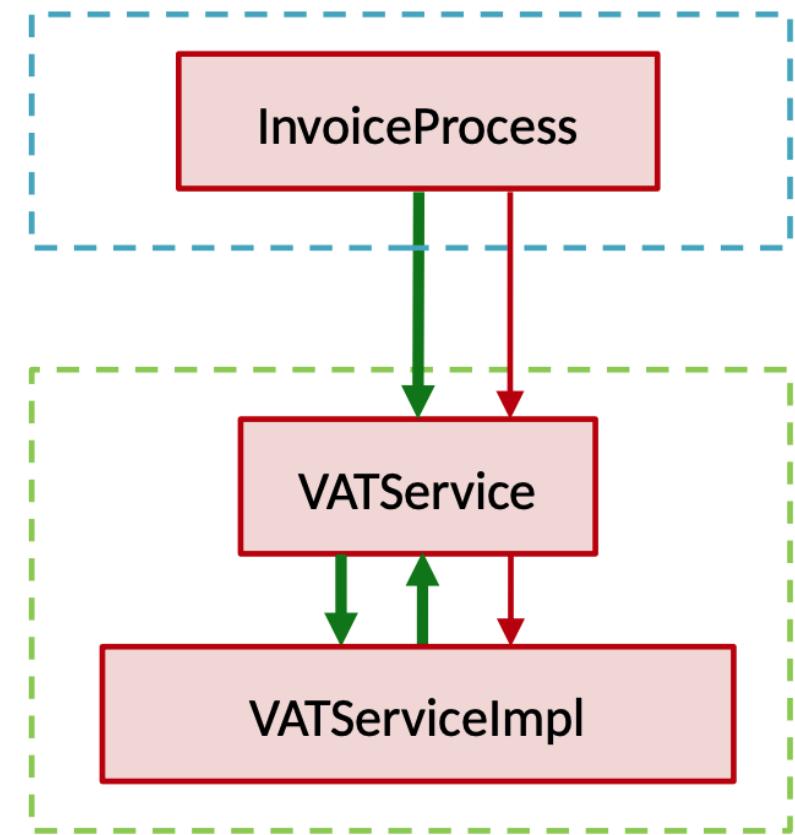
- **Quand se rend-on compte que l'on viole le principe I ?**
- Quand on doit reconstruire une classe alors qu'elle n'a pas été modifiée

- **SOLID !**
- **S = Single Responsability Principle**
- **O = Open Closed Principle**
- **L = Liskov Substitution**
- **I = Interface Segregation**
- **D = Dependency Inversion**
  - Impose une façon d'organiser le code
  - Les dépendances doivent référencer des abstractions
  - Les règles de haut niveau ne doivent pas dépendre de détails d'implémentation

# Dependency Inversion

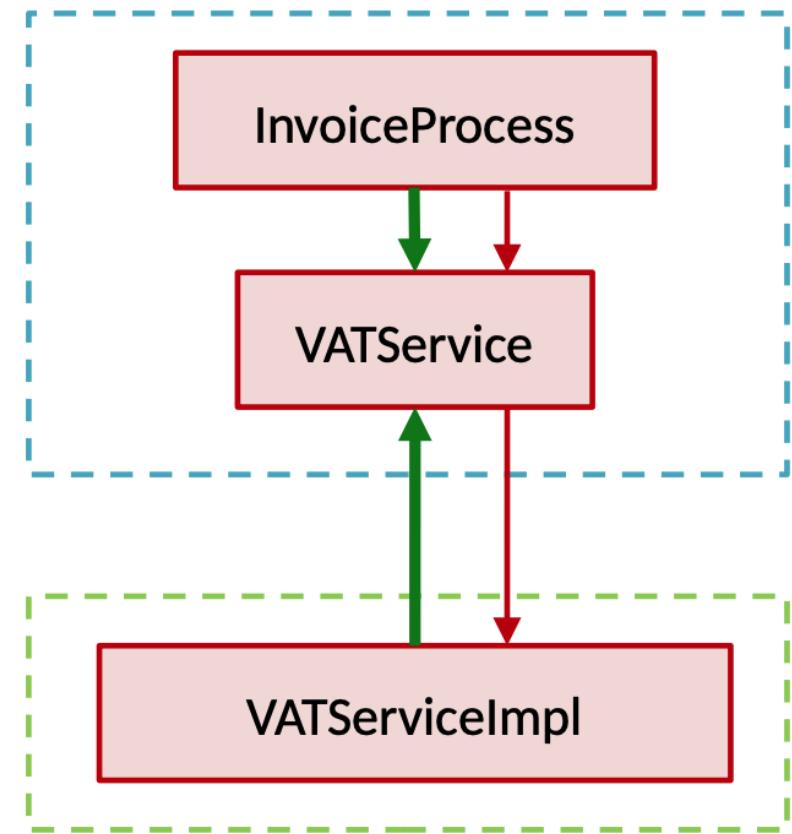
- Exemple

```
public class InvoiceProcess {  
  
    private VATService vatService =  
        VATService.getInstance();  
}
```



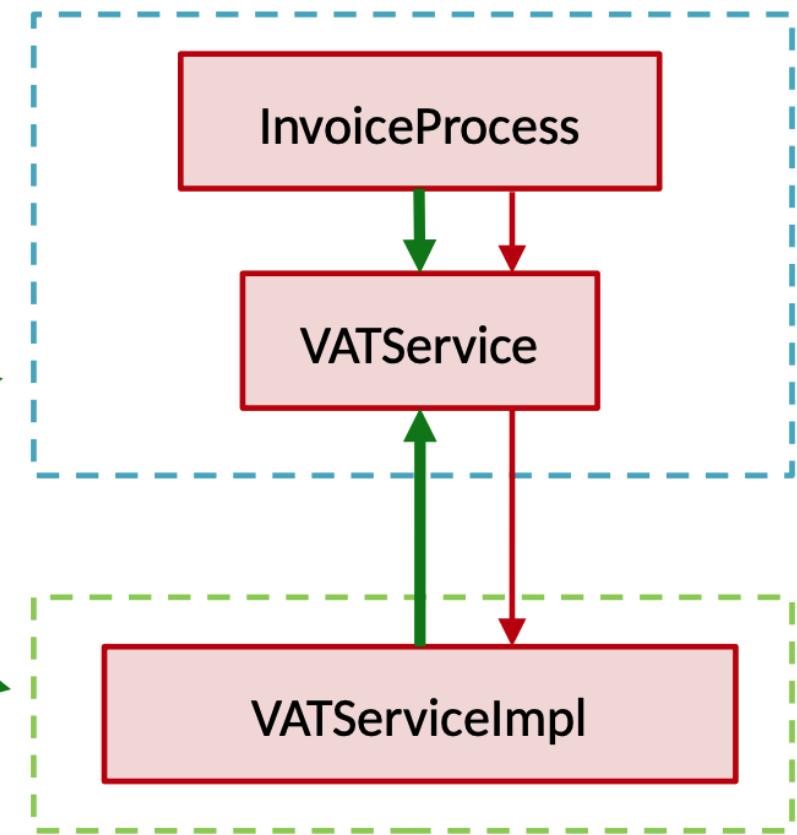
- Exemple

```
public class InvoiceProcess {  
  
    private VATService vatService;  
  
    InvoiceProcess(VATService vatService) {  
        this.vatService = vatService;  
    }  
}
```



# Dependency Inversion

```
public static void main(String... args) {  
  
    VATService vatService =  
        new VARServiceImpl()  
  
    InvoiceProcess invoiceProcess =  
        new InvoidProcess(vatService);  
}
```



- **Quand se rend-on compte que l'on viole le principe D ?**
- Quand il est impossible de déployer les modules de notre application indépendamment les uns des autres
- Quand une abstraction dépend en retour de son implémentation

- **Movie Rental Kata**
  - Un classique de Martin Fowler
- **Objectifs du refactoring**
  - On veut gérer un nouveau type de film, avec des règles de facturation différentes
  - On veut pouvoir choisir entre plusieurs formats d'impression



Kata

# Conclusion

- **Software Craftsmanship**
  - Se donner les moyens de délivrer des features à coût constant
- **Des techniques simples et plaisantes**
  - TDD
  - Refactoring
- **Nécessite de l'entraînement**
  - Pratique des katas
  - Connaissance des Design Patterns
- **Si vous êtes intéressés par aller plus loin**
  - <https://www.oxiane.com/formation/software-craftsmanship/>

**Merci de votre attention**