



Université  
de Lille  
**1** SCIENCES  
ET TECHNOLOGIES



**UFR IEEA**  
Formations en  
Informatique de  
Lille 1

# Conception Orientée Objet

Romain Rouvoy

*Licence mention Informatique*  
**Université de Lille**



# Menu du jour

1. 1 minute de philosophie
2. Ambitions
3. Principes
4. Un peu d'Art
5. Rappels
6. Méthodes
7. Promesses
8. Questions

# Pourquoi ??

LA minute de philosophie

# Un peu d'histoire...

In 1996, a European Ariane 5 rocket was set to deliver a payload of satellites into Earth orbit, but **problems with the software** caused the launch rocket to veer off its path a mere 37 seconds after launch. As it started disintegrating, it self-destructed (a security measure). The **problem was the result of code reuse** from the launch system's predecessor, Ariane 4, which had very different flight conditions from Ariane 5. **More than \$370 million were lost** due to this error.

# Beaucoup de classes de bugs

- **Arithmétiques** (division par 0, *over/underflow*, etc.)
- **Logiques** (boucles infinies, décalage)
- **Syntaxe** ( == ou =)
- **Resource** (*null pointer*, variable non initialisée)
- **Multi-tâches** (interblocage, accès concurrents)
- **Interfaçage** (API, protocole, versions)
- **Performance** (complexité, accès aléatoires)
- **Équipe** (communication, commentaires)

**“With Great Power  
Comes Great  
Responsibility ,”**

Spider-Man

Saturday - Nov 10, 2012(2:00 am)

# Architecte logiciel

L'informaticien architecte de logiciel se charge de **dresser les plans informatiques d'une application**. Il spécifie les besoins du client et dresse les articulations de différents programmes. Exercer le métier demande de **bonnes compétences en informatique**, notamment sur les langages internet, **mais aussi un bon relationnel**, puisqu'il est amené à encadrer des projets et à communiquer avec ses clients. La pratique de l'anglais est de plus en plus demandée.

# Architecte logiciel

- Salaire initial
  - autour de 2 500 € par mois
- Avec de l'expérience
  - entre 4 000 et 7 000 € par mois

# Architecte logiciel

- Maîtriser les fondamentaux de conception
- Réaliser une veille technologique continue
- **Pratiquer, pratiquer, pratiquer et... pratiquer !**

# Quelques principes clés

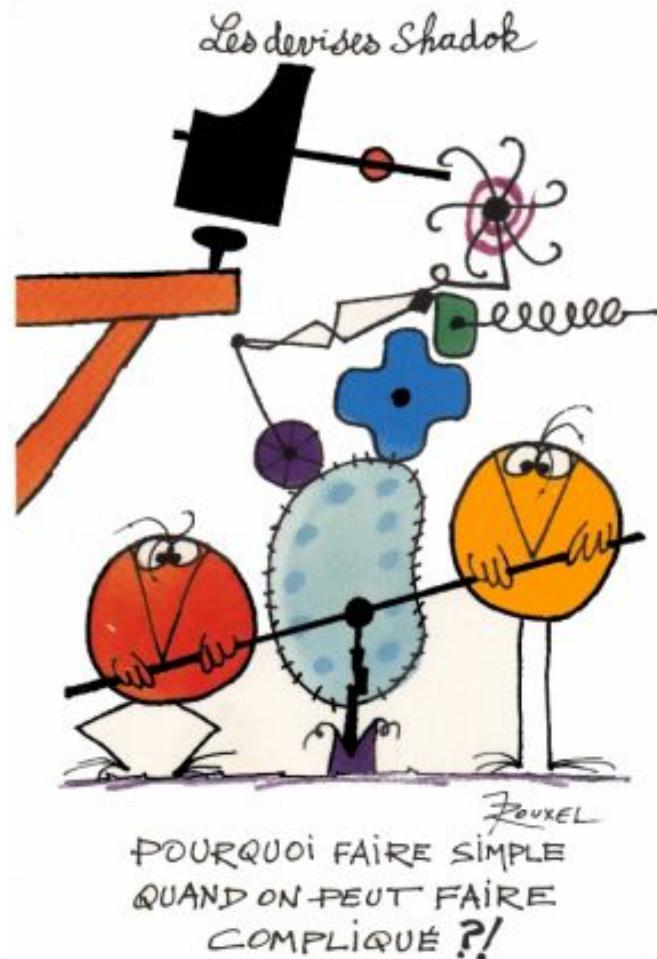
...qui sauvent la vie (d'un projet)

# KISS

*Keep It Simple, Stupid!*

# *Keep It Simple, Stupid*

- Principe qui n'est pas propre à l'informatique



# Keep It Simple, Stupid

«Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais **quand il n'y a plus rien à retrancher.**»

Antoine de Saint-Exupéry

- Un programme simple est plus facile à maintenir et à comprendre
- Éviter la **sur-inflation fonctionnelle**
- Quel est le code plus simple à lire pour réaliser une fonctionnalité requise par le client ?

# DRY

*Don't Repeat Yourself!*

# Don't Repeat Yourself

« Dans un système, toute connaissance doit avoir **une représentation unique**, non-ambiguë, faisant autorité. »

*The Pragmatic Programmer*

- Le copier-coller doit être interdit !
- Tirer parti de l'héritage et de la composition
- Tirer parti des patrons de conception

# YAGNI

*You Ain't Gonna Need It!*

# You Ain't Gonna Need It

« Mettez toujours en œuvre les choses **quand vous en avez effectivement besoin**, pas lorsque vous prévoyez simplement que vous en aurez besoin. »

Ron Jeffries

- Les fonctionnalités supplémentaires doivent être debugguées, documentées, et maintenues
- Tant que la fonctionnalité n'est pas réellement nécessaire, il est difficile de définir complètement ce qu'elle doit faire, et comment la tester

# You Ain't Gonna Need It

- Méthode **MoSCoW**
  - **M** : *must have this (vital)*
  - **S** : *should have this if at all possible (essentiel)*
  - **C** : *could have this if it does not affect anything else (confort)*
  - **W** : *won't have this time but would like in the future (luxe, c'est votre zone d'optimisation budgétaire)*

«Tester c'est douter»



# Tester, c'est douter

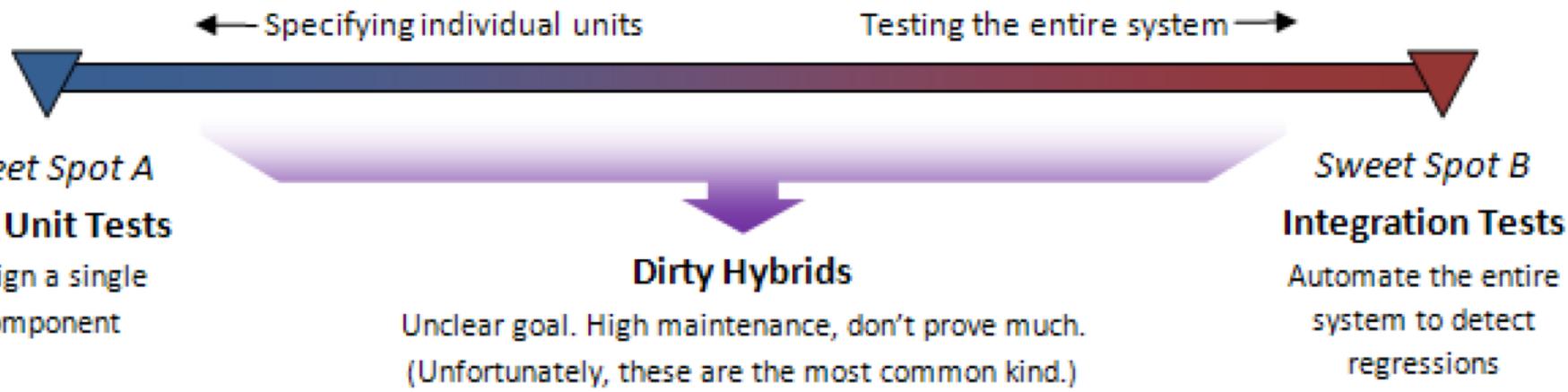
« *Le test de programmes peut montrer la présence de bugs, mais ne permet pas de garantir leur absence* »

Edsger W. Dijkstra , *The Humble Programmer* (1972)

- **Coût du test:** Autant que le programme !
- KISS !
- YAGNI !

# Différentes sortes de tests

- Tests fonctionnels (*black box test*)
- Tests structurels (*white box test*)
- Tests de charge / performance
- Tests d'intégration
- Tests d'acceptation (recette)
- ...



# Anatomie d'un test

- Nom + Description
- Code, aka **le cas de test**
- Données, aka **le jeu de test**
- Propriétés, aka **les oracles**
- **Tester, ce n'est pas déboguer**
- **Tester, ce n'est pas essayer**
- **Un test doit être reproductible**

# Anatomie d'un test

- Chaque test est **orthogonal** aux autres
- Chaque couvre **un et un seul comportement**
- Éviter les assertions inutiles
- Une **seule assertion logique** par test
- Tester une **seule unité logique** à la fois
- **Bouchonner** tout le reste
- Éviter les préconditions
- Ne pas tester les paramètres de configuration
- Bien **nommer vos tests** (sujet/scenario/résultat)

# Anatomie d'un test

```
public interface Division {  
    int divide(int a, int b);  
}
```

```
public class BadDivisionTest {  
    @Test  
    void divide() {...}  
}
```

```
public class GoodDivisionTest {  
    @Test  
    void divideZero_Should_ReturnZero() {...}  
  
    @Test  
    void divideByZero_Should_RaiseArithmeticException() {...}  
}
```

# Test fonctionnel

- Test dit «en boite noire»
  - N'assume **pas avoir une connaissance du code**
  - Peut être **écrit avant le code**
  - Se base sur une **spécification externe**
  - Partitionne les **données par classes d'équivalence**
  - Peut éventuellement **tester aux bornes**

# Test fonctionnel

Comment tester l'opérateur  
int divide(int, int) ?

# Test structurel

- Tests dit «en boite blanche»
  - Assume avoir une **connaissance du code**
  - Maximiser la **couverture du code** (graphe de flot)
  - Se base sur une **spécification interne** du code
  - Ne garantit pas l'absence de bug

# Test structurel

```
@Override
public int divide(int a, int b) {
    if (b == 0)
        throw new ArithmeticException();
    if (b == 1)
        return b;
    boolean resEstNegatif = false;
    int res = 0;
    if (a < 0) {
        resEstNegatif = true;
        a = -a;
    }
    if (b < 0) {
        resEstNegatif = !resEstNegatif;
        b = -b;
    }
    while (a > 0) {
        a = subtract(a, b);
        ++res;
    }
    return resEstNegatif?-res:res;
}
```

# Test de charge / performance

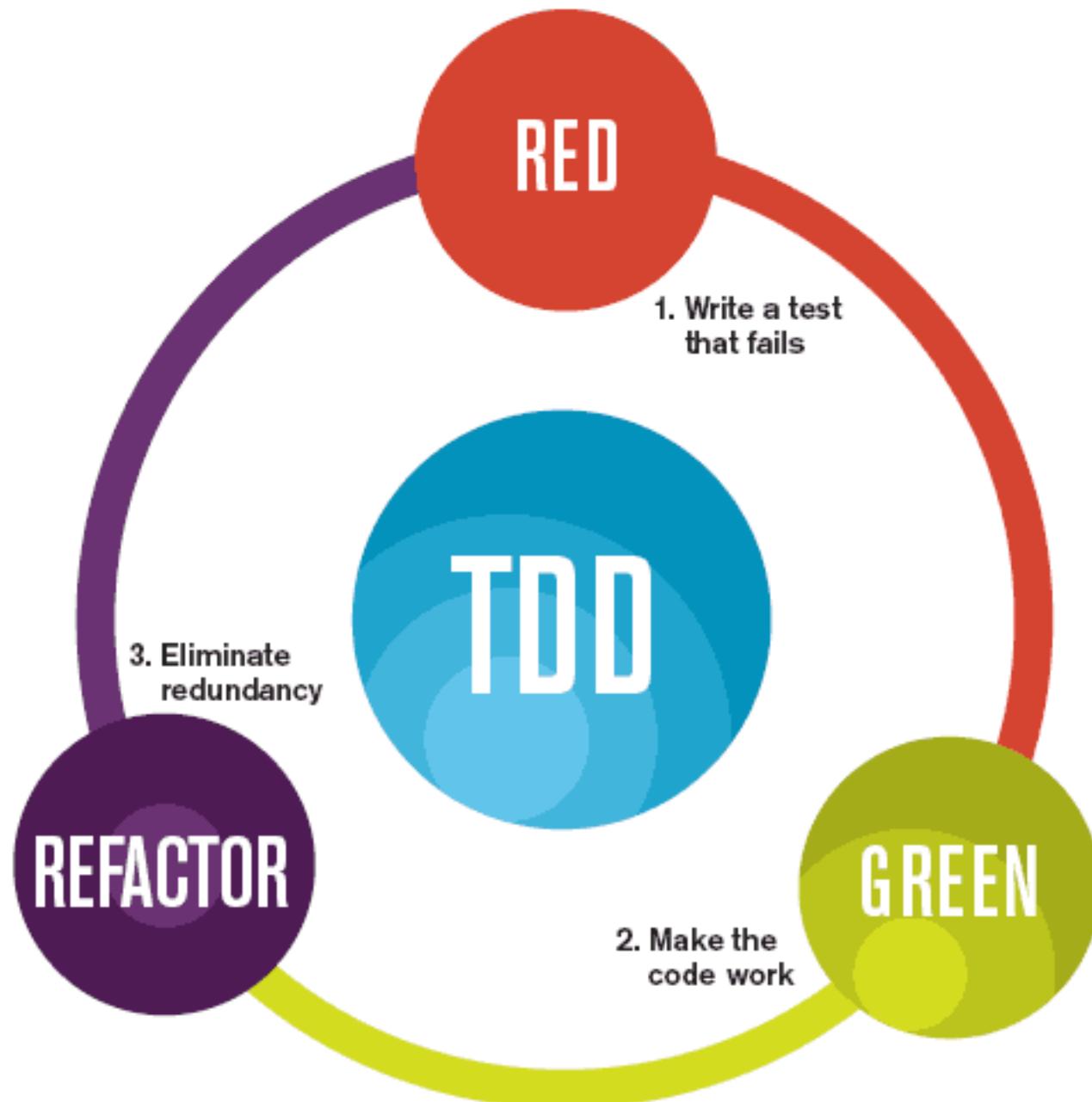
- Test non-fonctionnel
  - Assume une **validité fonctionnelle**
  - Proche de la notion de *(micro-)benchmark*
  - Évalue la **complexité algorithmique**

# Test d'intégration

- Test en condition
  - Jouer des **scénarios utilisateurs**
  - Utiliser un assemblage des composants
  - Vérifier que la **composition est correcte**

# TDD: *Test-Driven Development*

- Je teste, je code, j'ajuste... [GOTO 0]
1. J'écris un test qui échoue (*Test Fail First*)
  2. Je corrige le programme pour que le test passe
  3. Je réusine le code pour améliorer la qualité
  4. Je commit mes changements



The mantra of Test-Driven Development (TDD) is “red, green, refactor.”

# TDD Code Kata

<https://youtu.be/bGa90QVWw0I>

