

R3.02 : Développement efficace

Plan & Introduction Notion de contrat

J-F. Kamp

Septembre 2023

PLAN du cours R3.02

- **Semaine 35/2023 : Cours1 & Cours2**

- Outils, évaluations
- Les contrats
- Listes chaînées

- **Semaine 36 : TP1**

- **TP1** : Liste chaînée

- **Semaine 37 : Cours3 – TP2**

- **Cours3** : Table Hachage
- **TP2** : Table Hachage

- **Semaine 38 : Cours4 – TP3**

- **Cours4** : Types génériques
- **TP3** : Généricité – API Java

PLAN du cours R3.02

- **Semaine 39 : Cours5 – TP4**
 - **Cours5** : Structure en arbre 1
 - **TP4** : Structure en arbre
- **Semaine 40 : Cours6 – TP4**
 - **Cours6** : Structure en arbre 2
 - **TP4** : Structure en arbre
- **Semaine 41 :**
 - **Pas de Cours**
 - **TP4** : Structure en arbre (fin)

Evaluations

1 contrôle continu : évaluation d'un TP parmi 4 (poids 1/3)

Contrôle terminal : QCM en semaine 42 (poids 2/3)

PLAN

1. Outils de développement

2. Les contrats

Outils de développement

Outils professionnels : IDE



Java 11 (en tout cas
pas au-delà)



- De préférence : Oxygen, Photon, Neon, Mars, Luna
- Mars Windows 64bits disponible sur Moodle

Outils professionnels : test

JUnit

JUnit4 obligatoire

Sous Eclipse

New JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

☐ New JUnit 3 test ☒ New JUnit 4 test

Source folder:

Package:

Name:

Superclass:

Which method stubs would you like to create?

☒ setUpBeforeClass() ☒ tearDownAfterClass()
☒ setUp() ☒ tearDown()
☐ constructor

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

Class under test:

JUnit JUnit4 obligatoire

- Test unitaire de classe
- Chaque méthode publique *maMeth* de la classe à tester
=> 1 méthode de test de signature

public void testMaMeth() {...}

- Utiliser les assertions
*assertTrue(String message,
boolean condition)*
*assertFalse(String message,
boolean condition)*

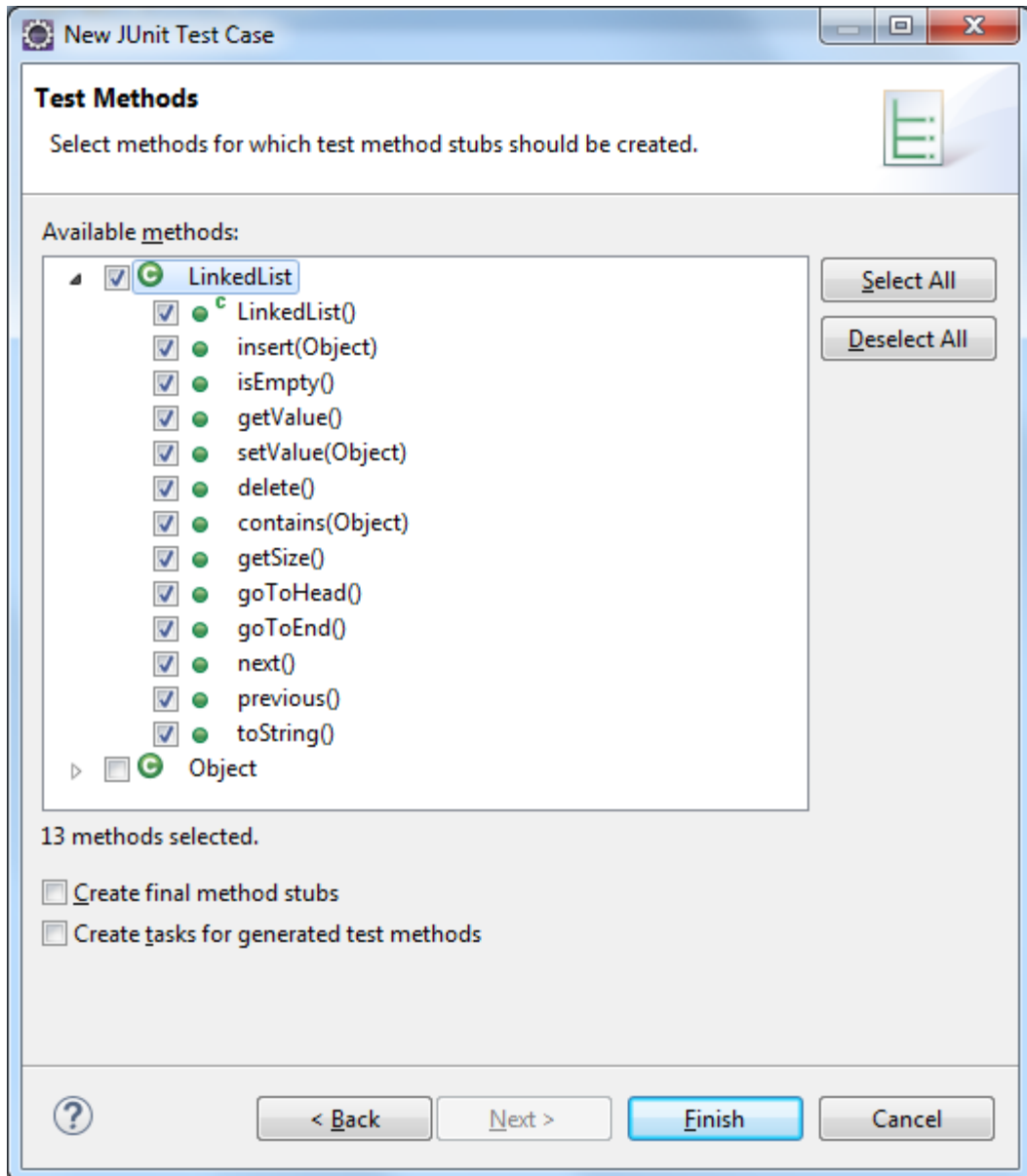
...

Outils professionnels : test

JUnit

JUnit4 obligatoire

Sous Eclipse





Sous Eclipse Oxygen, Photon,
Neon, Mars, Luna

Outil d'analyse de la qualité du
code

A installer par *drag and drop* à
partir de

[https://marketplace.eclipse.org/co
ntent/sonarlint](https://marketplace.eclipse.org/content/sonarlint)



- Aide à écrire du code de qualité
- Souligne en bleu le « mauvais codage » (voire erroné) :
 - selon 3 catégories : Bugs, Code Smells, Vulnerabilities
 - selon une gradation : Minor, Major



Exemples

// Exemple1

```
boolean trouve, continuer;
```

Multiple variables should not be declared on the same line (squid:S1659)

Code Smell - Minor

// Exemple2

```
while ( tmp != sentinel ) {
```

```
    ret = ret + tmp.theValue.toString() + " - ";
```

```
    tmp = tmp.next;
```

```
}
```

Strings should not be concatenated using '+' in a loop, use append instead (squid:S1643)

Code Smell - Minor

Evaluation des TPs

1 seul corrigé sur les 4 à rendre

Eclipse + JUnit4 obligatoires

Sonarlint obligatoire avec un commentaire dans le code lorsque pris en compte

Les sources *.java* uniquement (pas l'environnement Eclipse)

JavaDoc en anglais

Un seul TP non rendu => 0/20
-2 pts par TP rendu en retard

Les contrats

Rappel sur l'encapsulation

Technique de l'encapsulation : protéger les données (les attributs) en autorisant la modification uniquement par l'intermédiaire de méthodes.

```
class Cercle {  
  
    // Attribut de la classe, le rayon  
    private double rayon;  
  
    ...  
    public void setRayon( double leRayon ) {  
  
        if ( leRayon <= 0 ) {  
            System.out.println("Erreur !");  
        }  
        else {  
            rayon = leRayon;  
        }  
    }  
    ...  
}
```


Notion de contrat

Idée de base : une classe construite et bien testée sera à la disposition du monde extérieur (`public class MaClasse {...}`) pour utilisation.

Elle doit garantir 3 principes par rapport au « client » qui va utiliser la classe (par création d'objets du type de la classe) :

1. La classe doit empêcher une mauvaise utilisation de ses services par le client.
2. Tous les services offerts par la classe doivent être valides.
3. A aucun moment on ne tolérera un état caduque (valeurs impossibles d'attributs) d'un objet issu de la classe.

Le contrat du logiciel

Les seuls services accessibles depuis une classe cliente sont ceux ayant une visibilité publique (mot-clé **public**) => essentiellement :

- les constructeurs
- les modificateurs
- les accesseurs

Pour garantir le contrat, on vérifie des conditions avant et après l'appel d'une méthode publique :

- la pré-condition est la condition à vérifier AVANT l'utilisation de la méthode,
- la post-condition est la condition à vérifier APRES l'utilisation de la méthode.

L'invariant est la troisième condition à vérifier A TOUT MOMENT de la vie d'un objet.

Pré-Condition

La pré-condition se traduit par une expression booléenne que l'on doit évaluer avant même l'exécution de la méthode.

Exemple :

```
// pré-condition : vérifier que leRayon est > 0  
  
public void setRayon( double leRayon ) {  
  
    // code de la méthode  
}
```

Si la pré-condition est vraie, la méthode (le service) s'exécute sinon elle ne s'exécute pas et la classe cliente en est avertie.

Post-Condition

La post-condition est une expression booléenne que l'on évalue juste après l'exécution de la méthode. Son rôle est de vérifier que le travail s'est correctement accompli.

Exemple :

```
// post-condition : vérifier que l'attribut « leTemps »  
// s'est bien incrémenter de la valeur  
// « uneDuree.getLeTemps() »  
  
public void ajoute( Duree uneDuree ) {  
  
    // code de la méthode  
}
```

Si la post-condition est vraie, le service est correctement rendu au client sinon le développement de la classe fournisseur est à revoir !

Invariant

L'invariant se traduit par une expression booléenne qui vérifie que l'objet reste dans un état cohérent. Il faut vérifier qu'à tout moment les attributs de l'objet sont valides.

« A tout moment » signifie :

- en fin de création de l'objet,
- dès que l'objet « se transforme » c-à-d dès qu'un de ses modificateurs publique est appelé (ou une méthode privée qui modifie son état).

Exemple :

```
// invariant : vérifier que leTemps est toujours >= 0  
  
class Duree {  
  
    private long leTemps;  
  
    ...  
  
}
```

Dès que l'invariant devient faux, l'objet est dans un état impossible, la classe est à revoir !

Quel est la syntaxe en Java qui permet d'exprimer les contrats ?

Aucune officiellement contrairement au langage Eiffel par exemple.

Il faut donc les exprimer avec des mécanismes propres à Java :

- Les exceptions vont exprimer les pré-conditions.
- Les assertions vont exprimer les post-conditions et les invariants.

Java et la pré-condition

On lancera une exception
(`RuntimeException(...)` par exemple) pour
signaler la violation de la pré-condition.

```
// pré-condition : vérifier que leRayon est > 0

public void setRayon( double leRayon ) {

    if ( leRayon <= 0 ) {
        throw new RuntimeException ( "Le
rayon doit être strictement positif ! " );
    }
    ...
}
```

Note : une méthode privée n'a pas de pré-condition car elle ne peut pas être utilisée à l'extérieur de la classe.

Java et la post-condition

L'assertion en Java (depuis 1.4) est une instruction qui permet au développeur d'effectuer des tests sur l'état du programme.

Syntaxe : **assert Expression1 : Expression2;**

- Expression1 est une expression booléenne
- Expression2 est une chaîne de caractères qui est un message d'erreur

Exemple d'assertion simple :

```
// je pense que « a » égale « b » mais je veux le  
// vérifier en cours de programme
```

```
...
```

```
assert ( a == b ) : "!! Erreur a est différent de b";
```

Si l'expression est vraie rien ne se passe.
Sinon, Java lance une *Error* **AssertionError** avec le message d'erreur.

Java et la post-condition

Exemple de post-condition :

```
// post-condition : vérifier que l'attribut « leTemps »  
// s'est bien incrémenter de la valeur  
// « uneD.getLeTemps() »
```

```
public void ajoute( Duree uneD ) {  
    long tmp;
```

```
    tmp = leTemps;  
    leTemps = leTemps + uneD.getLeTemps();
```

```
    assert ( leTemps == ( tmp +  
uneD.getLeTemps() ) ) : "Incrémentation du temps  
non valide !";
```

```
}
```

Java et l'invariant

L'invariant en Java se traduira par une méthode privée `private boolean invariant() {...}` qui **teste les attributs un par un** et qui sera appelée après chaque exécution de méthode (souvent un modificateur).

```
// invariant : vérifier que leTemps est toujours >= 0
```

```
class Duree {  
    private long leTemps;  
    ...  
}
```

Java et l'invariant

// Méthode privée = évaluation de l'invariant

```
private boolean invariant() {  
    boolean ret = true;  
  
    if ( this.leTemps < 0 ) {  
        ret = false;  
        System.out.println("Temps négatif !");  
    }  
  
    return ret;  
}
```

// Modificateur

```
public void ajoute( Duree uneDuree ) {  
    long tmp = this.leTemps;  
  
    leTemps = leTemps + uneDuree.getLeTemps();  
  
    // post-condition  
    assert ( leTemps == ( tmp +  
uneDuree.getLeTemps() ) ) : "Incrémentation du temps  
non valide !";  
  
    // invariant  
    assert ( this.invariant() ) : "Invariant violé !";  
}
```

Démarche de test unitaire avec les contrats

Au début du développement d'une nouvelle classe, il faut écrire un maximum de contrats car :

- je ne suis pas certain que chacun des services proposés est valide,
- je ne suis pas certain qu'aucun des services proposés ne met l'objet dans un état impossible.

En fin de développement (test unitaire réussi et complet de la classe) :

- tous les services sont testés => les post-conditions et les invariants ne déclenchent plus (**ne pas oublier -ea sous Eclipse !**),
- les mauvaises utilisations des services sont rejetés par les pré-conditions.

En pratique et sous Eclipse

Pour chaque méthode testée avec JUnit :

- Annoncer clairement (message au terminal) ce que vous testez
- Communiquer clairement le résultat du test (échec ou réussi)

Pour chaque méthode, tester les cas suivants :

- Cas normaux
- Cas limites
- Cas d'erreurs : dans ce cas la pré-condition est violée, la capture de l'exception doit se faire et la communication doit être « Test réussi »

Pour l'exécution de la classe de test, **NE PAS OUBLIER l'argument -ea sous Eclipse** sinon les assertions ne se déclenchent pas !