Bonnes pratiques de programmation

Arnaud Labourel arnaud.labourel@univ-amu.fr

22 septembre 2021

Section 1

Une méthodologie pour bien nommer

Pourquoi bien nommer est important

Albert Camus (1944)

“Mal nommer un objet, c’est ajouter au malheur de ce monde”

Que veut dire le texte suivant ?

La L3 info : MIAGE ne dépend pas de la même UFR que la L2 info : MI, elle dépend de la FEG et non de la FS. Pour faire vos IA et IP, vous devez donc contacter la scol de Forbin et non celle de SCH.

Pièges à éviter pour le nommage de variables/attributs

Vous ne devez pas avoir des variables avec des :

- noms en une lettre (même pour les indices) : `i`, `j`, ...
- noms numérotés : `a1`, `a2`, `a3`, ...
- abréviations ayant plusieurs interprétations : `rec`, `res`, ...
- noms ne donnant pas le sens précis : `temporary`, `result`, ...
- des noms trompeurs : par exemple un `accountList` doit être une `List` (et pas un `array` ou un autre type)
- types de l'objet au singulier pour une collection d'objet : une liste de personnes doit s'appeler `persons` et non `person`.
- noms imprononçables : `genymdhms`, ...

Règles de nommage (1/2)

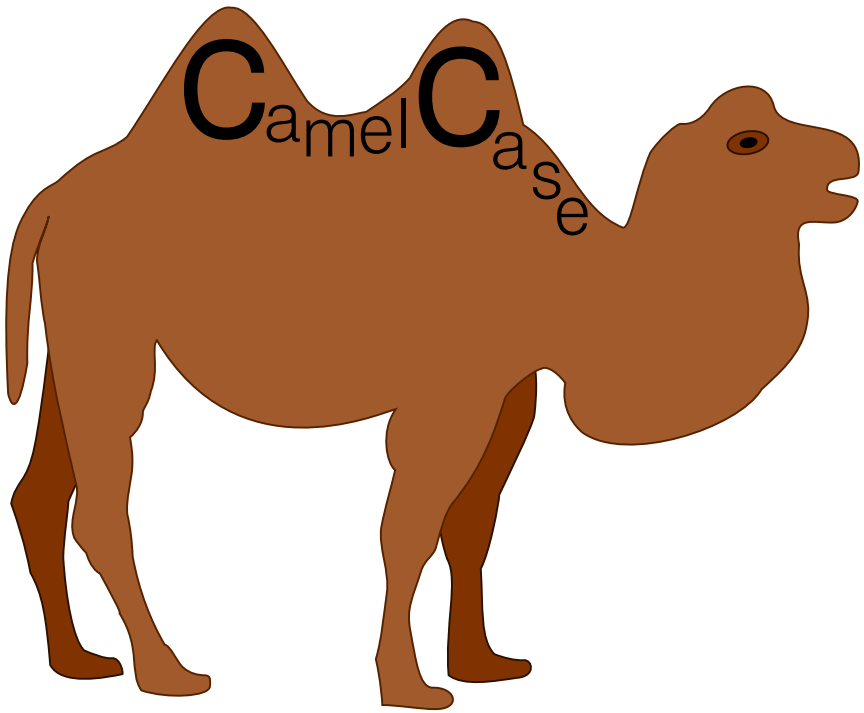
En anglais

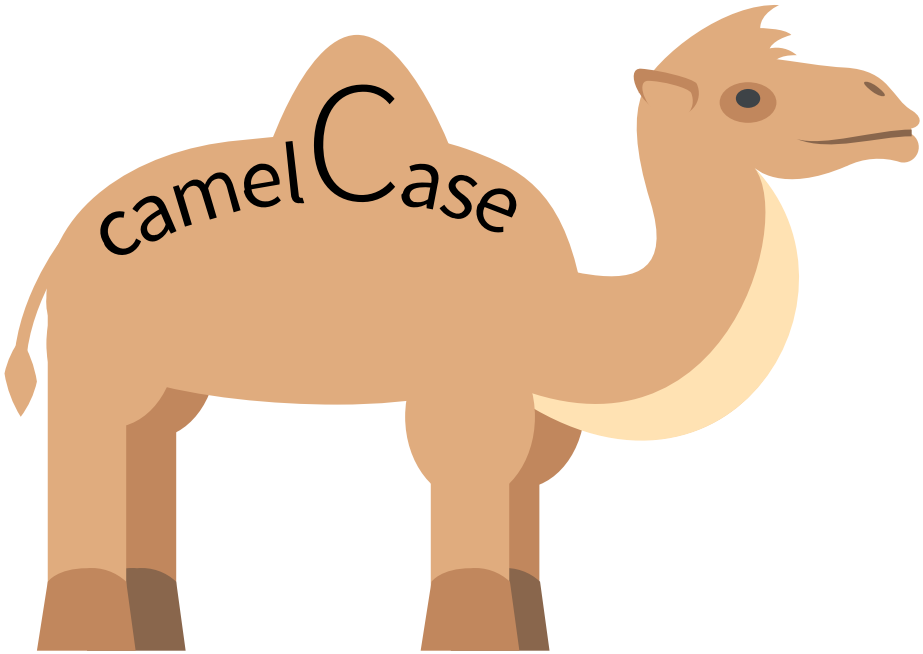
- Le **Java** et la quasi-totalité des langages de programmation utilise l'anglais (dans les mot-clés et les librairies standards).
  $\Rightarrow$ On doit programmer en anglais pour avoir la cohérence du code
- Utiliser l'anglais permet aussi d'augmenter le nombre de personnes pouvant lire le code et d'avoir de nombreux exemples existants pour s'inspirer.

En "camel case"

Un nom composé de plusieurs mots n'utilise ni espace ni ponctuation, et sépare les mots en mettant en capitale la première lettre de chaque mot.
Exemples: `getElementsByTagName`, `ListArray`, ...

Règles de nommage (2/2)





CamelCase vs camelCase

- Les noms qui définissent un type (classes, interfaces, enum, record, ...) commence par un majuscule.
- Les autres noms (méthodes, variables, attributs, ...) commencent par une minuscule.

Nommage des méthodes : cas 1

Méthodes procédurales

Méthodes modifiant l'état de l'objet
$\Rightarrow$ groupe verbal à l'infinitif.

Exemples

- `boolean add(E element)`
- `E set(int index, E element)`
- `boolean removeAll(Collection<?> c)`

Nommage des méthodes : cas 2

Expressions non booléennes

Méthodes renvoyant une partie de l'état de l'objet ⇒ groupe nominal ou getter.

Exemples

- `int size()`
- `List<E> subList(int fromIndex, int toIndex)`
- `int hashcode()`
- `ListIterator<E> listIterator()`
- `E get(int index)`
- `Color getBackground()`
- `float getOpacity()`

Nommage des méthodes : cas 3

Expressions booléennes

Méthodes testant un prédicat sur l'objet $\Rightarrow$ groupe verbal au présent.

Exemples

- `boolean isEmpty()`
- `boolean contains(Object o)`
- `boolean equals(Object o)`

Nommage des méthodes

Méthodes de conversion $\Rightarrow$ utilisation du to

Exemples :

- `String toString()`
- `Object[] toArray()`

Les règles ne sont pas absolues mais juste des conventions qui peuvent avoir des exceptions.

En général le plus simple est de s'inspirer de l'existant : par exemple la **JDK** et de bien réfléchir lorsqu'on souhaite déroger aux règles.

Comment rendre le nommage des méthodes facile ?

En écrivant des méthodes courtes

De préférence une dizaine de ligne maximum.

Comment écrire des méthodes courtes

En extrayant le plus possible les partie du code d'une méthode à d'autres méthodes.

Conseils

- Réfléchir avant de coder au rôle de la méthode
- Se demander ce qui peut être confié à d'autres méthodes

Ne pas mentir

```
class User {
    private boolean authenticated;
    private String password;

    public boolean checkPassword(String password) {
        if (password.equals(this.password)) {
            authenticated = true;
            return true;
        }
        return false;
    }
}
```

La méthode authentifie l'utilisateur alors qu'elle ne devrait que vérifier la validité du mot de passe d'après son nom.

Section 2

Des principes pour bien programmer

Do One Thing

Une fonction ne doit faire qu'**une seule chose**.

Pour cela, elle ne doit réaliser que des étapes de même niveau d'abstraction.

On décompose la fonction :

Pour faire la cuisine je dois (premier niveau d'abstraction) :

- choisir une recette;
- réunir les ingrédients;
- suivre la recette.

Pour choisir une recette, je dois (deuxième niveau d'abstraction):

- réfléchir à ce que j'ai envie de manger;
- chercher sur marmiton.

Mauvaise approche

```
void cook(){
    // On choisit la recette
    Food wantToEat = thinkAboutFood();
    Recipe recipe = lookOnMarmiton(wantToEat);
    // On réunit les ingrédients
    openFridge();
    for(Ingredient ingredient :
        recipe.getFreshIngredients()){
        takeInFrige(ingredient);
    }
    closeFridge();
    openCupboard();
    ...
    // On suit la recette
    ...
}
```

Bonne approche

```
void cook(){
    Recipe recipe = chooseRecipe();
    gatherIngredients(recipe);
    followRecipe(recipe);
}

Recipe chooseRecipe(){
    Food wantToEat = thinkAboutFood();
    Recipe recipe = lookOnMarmiton(wantToEat);
    return recipe;
}

...
```

Programme bien conçu

Un programme est “bien conçu” s’il permet de :

- Absorber les changements avec un minimum d’effort
- Implémenter les nouvelles fonctionnalités sans toucher aux anciennes
- Modifier les fonctionnalités existantes en modifiant localement le code

Objectifs

- Limiter les modules impactés
  - Simplifier les tests unitaires
  - Rester conforme à la partie des spécifications qui n’ont pas changé
  - Facilité l’intégration
- Gagner du temps

Section 3

Tests

Différents types de de tests

Règle

Un code non testé n'a aucune valeur.

Corollaire

Tout code doit être testé

Différents type de tests

- **Test unitaires** : Tester les différentes parties (méthodes, classes) d'un programme indépendamment les unes des autres.
- **Test d'intégration** : Tester des portions du programme combinant plusieurs classes.
- **Test de non régression** : Vérifier que le nouveau code ajouté ne corrompt pas les codes précédents : les tests précédemment réussis doivent encore l'être.

Tests unitaires

- Tester une unité de code : classe, méthodes, ...
- Vérifier un comportement :
  - cas normaux
  - cas limites
  - cas anormaux

Tests unitaires en java : `JUnit` avec `AssertJ`

- Un framework de test unitaire pour Java
- S'appuie sur des **assertions**

Utilisation de JUnit (1/2)

Règles

- 1 classe de test = un ensemble de méthodes de test
- 1 classe de test par classe à tester
- 1 méthode de test = 1 cas de test
- 1 cas de test = (description, données d'entrée, résultat attendu)

Structure d'une méthode de test de base

- méthode d'instance publique
- annotée avec `@Test` (à mettre avant la déclaration de la méthode)
- ne prend aucun paramètre
- ne renvoie rien
- lève une `AssertionError` en cas de test échoué

Utilisation de JUnit (2/2)

Conventions de nommage

- nom d'une classe de test : *NameTestedClass*Test
- nom d'une méthode de test : test*NameMethodTested*

Structure d'un projet avec tests

les tests sont séparés du code de production : répertoire `main` pour le code de production et répertoire `test` pour le code de tests.

$\Rightarrow$ nécessaire de séparer les test du code de production car :

- on ne donne pas l'accès au code de test au client par exemple
- les tests ont un rôle spécifique différent du code de production

Assertions AssertJ (1/2)

- `assertThat(condition).isTrue()` : vérifie que `condition` est vraie.
- `assertThat(condition).isFalse()` : vérifie que `condition` est faux.
- `assertThat(actual).isEqualTo(expected)` : vérifie que `expected` est égal à `actual`

  égal : `equals` pour les objets et == pour les types primitifs.
- `assertThat(actual).isCloseTo(expected, within(delta))` : vérifie que $|expected - actual| \leq delta$
- `assertThat(object).isNull()` : vérifie que la référence est `null`

Assertions AssertJ (2/2)

- `assertThat(object).isNotNull()` : vérifie que la référence **n'**est **pas** `null`
- `assertThat(actual).isSameAs(expected)` : vérifie que les deux objets sont les mêmes (même référence).
- `assertThat(list).containsExactly(e1, e2, e3)` : vérifie que la liste `list` contient uniquement les éléments `e1`, `e2` et `e3` dans cet ordre.
- `assertThat(list1).containsExactlyElementsOf(list2)` : vérifie que les deux listes `list1` et `list2` contiennent les mêmes éléments dans le même ordre.
- `fail(message)` : échoue toujours en affichant message

Message

Il est possible de provoquer l'affichage d'un `message` lors d'un test faux en appelant `as(message)` sur le retour d'un `assertThat`.

Exemple de classe à tester : RationalNumber

```
public class RationalNumber {
  public final int numerator;
  public final int denominator;

  public RationalNumber(int numerator, int denominator) {
    int gcd = gcd(numerator, denominator);
    this.numerator = numerator / gcd;
    this.denominator = denominator / gcd;
  }

  public RationalNumber add(RationalNumber val) {
    int numerator   = (this.numerator * val.denominator)
    int denominator = this.denominator * val.denominator;
    return new RationalNumber(numerator, denominator);
  }
```

Exemple de classe de test

```java
import org.junit.jupiter.api.Test;
import static org.assertj.core.api.Assertions.*;
public class RationalNumberTest {
  @Test
  void testAdd(){
    RationalNumber one = new RationalNumber(1, 1);
    RationalNumber onePlusOne = one.add(one);
    assertThat(onePlusOne.numerator)
            .as("Numerator of one plus one is two.")
            .isEqualTo(2);
    assertThat(onePlusOne.denominator)
            .as("Denominator of one plus one is one.")
            .isEqualTo(1);
  }
}
```

Exemple de classe à tester (1/2) : Emails

```java
public class Emails {
  private String text;
  public Emails(String text) { this.text = text; }
  public List<String> userNames() {
    int pos = 0;
    List<String> users = new ArrayList<String>();
    for(;;) {
      int atIndex = text.indexOf('@', pos);
      if (atIndex == -1) break;
      String userName = userName(atIndex);
      if (userName.length() > 0) users.add(userName);
      pos = atIndex + 1;
    }
    return users;
  }
```

Exemple de classe à tester (2/2) : Emails

```
  private String userName(int atIndex) {
    int back = atIndex - 1;
    while (back >= 0 &&
           (Character.isLetterOrDigit(text.charAt(back))
           || text.charAt(back) == '.')) {
      back--;
    }
    return text.substring(back + 1, atIndex);
  }
}
```

Exemple de classe de test (1/3)

```java
import org.junit.jupiter.api.Test;
import static org.assertj.core.api.Assertions.*;

public class EmailsTest{
  @Test
  public void testUsersBasic() {
    Emails emails =
      new Emails("foo bart@cs.edu xyz marge@ms.com baz");
    assertThat(emails.getUserNames())
      .containsExactly("bart", "marge");
  }
```

Exemple de classe de test (2/3)

```java
@Test
public void testUsersChars() {
  Emails emails =
    new Emails("fo f.ast@cs.edu bar&a.2.c@ms.com ");
  assertThat(emails.getUserNames())
    .containsExactly("f.ast", "a.2.c");
}
@Test
public void testUsersChars() {
  Emails emails =
    new Emails("fo f.ast@cs.edu bar&a.2.c@ms.com ");
  assertThat(emails.getUserNames())
    .containsExactly("f.ast", "a.2.c");
}
```

Exemple de classe de test (3/3)

```
  @Test
  public void testUsersHard() {
    Emails emails = new Emails("x y@cs 3@ @z@");
    assertThat(emails.getUserNames())
      .isNotEmpty();
      .containsExactly("y", "3", "z");
    emails = new Emails("no emails here!");
    assertThat(emails.getUserNames()).isEmpty();
    emails = new Emails("@@@");
    assertThat(emails.getUserNames()).isEmpty();
    emails = new Emails("");
    assertThat(emails.getUserNames()).isEmpty();
  }
}
```

Test unitaires (à retenir)

- Il est essentiel de tester son code.
- Écrire au moins une méthode de test pour chaque méthode du code de production.
- Il est important des tester tous les types de cas :
  - cas normaux (utilisation naturelle de la méthode sur une donnée naturelle)
  - cas limites (utilisation de la méthode sur une donnée “étrange”)
  - cas anormaux (vérification que les erreurs d’utilisation, c’est-à-dire que les cas d’erreurs sont bien pris en compte et gérés)

La suite : le TDD (Test Driven Development)

- Écrire un test qui échoue avant de pouvoir écrire du code
- Écrire une assertion à la fois qui fait échouer un test
- Écrire le minimum de code pour que l’assertion soit satisfaite

Gestion de version

Principe

- Le code d'un projet est stocké dans un serveur.
- Les développeurs soumettent des modifications avec des commentaires à chaque fois.
- Le serveur conserve l'historique des mises à jour

Pourquoi la gestion de version ?

- Pour travailler de manière harmonieuse en équipe sans se marcher dessus
- Pour revenir en arrière en cas de problèmes (chaque modification est génère une version du code sur laquelle il est possible de revenir)
- Possibilité de faire valider le code (via des tests) par le serveur et de rendre le déploiement automatique

Git

- Logiciel de gestion de version le plus populaire
- Serveur gratuit : github
- Version libre de logiciel serveur : gitlab
- Gestion de version décentralisée : la gestion de version se fait aussi en local

Utilisation de git

- Via l'IDE : VCS (Version Control Systems) dans le menu d'IntelliJ
- En ligne de commande : commande git

Git (source: Randall Munroe (xkcd))

Fonctionnement basique de git

- À la première utilisation, on crée une copie locale du dépôt git (`clone` ou `init`)
- À chaque commit :
  - on récupère la version courante du dépôt sur le serveur (`pull`)
  - On ajoute les fichiers à modifier (`add`)
  - On finalise le commit en donnant un message résumant les modifications (`commit`)
- Après un ou plusieurs commits, on met à jour la version distante avec nos modifications (`push`)

Exemples de commandes git

`git clone adresse_projet`

⇒ Clone un projet en local depuis un serveur

`git add nom_de_fichier`

⇒ Ajoute un fichier à la prochaine mise à jour.

`git commit -m"commentaire"`

⇒ Fait une mise à jour en local

`git pull`

⇒ Récupère les mises à jour du serveur en local

`git push`

⇒ Pousse les mises à jour locales sur le serveur

Gestion de version

- Importance d'avoir une gestion de version de son code
- Commandes de base : `clone`, `add`, `commit`, `push`, `pull`, ...

Git avancé (dans vos prochains cours)

- Notion de branche : on "fork" le dépôt pour créer une branche séparée sur laquelle on va travailler et on demande ensuite de l'intégrer dans la branche principale une fois les modifications effectuées.
- CI/CD (Intégration Continue/Déploiement (ou livraison) continue) : automatisation des tests et du déploiement (par exemple sur les serveurs qui utilise le code) ou de la livraison (publication du code compilé sur un serveur) à chaque push sur la branche principale.