Conduite de projet :
Quelques conseils, de développeur à développeur.

Aldric Degorre
(IRIF, U-Paris) – adegorre@irif.fr
(d'après transparents originaux de Yann Régis-Gianas)

17 novembre 2023

Plan

En cours

- Partiel
- Documenter
- Programmer avec style

Plan

Question(s) de style

- Aucun programme n'est écrit directement dans sa version définitive.
- Il doit donc pouvoir être facilement modifié par la suite.
- Pour cela, ce qui est déjà écrit doit être **lisible et compréhensible**.
  - lisible par le programmeur d'origine
  - lisible par l'équipe qui travaille sur le projet
  - lisible par toute personne susceptible de travailler sur le code source (pour le logiciel libre : la Terre entière !)

Les commentaires[1] et la Javadoc peuvent aider, mais rien ne remplace un code source bien écrit.

1. Si un code source contient plus de commentaires que de code, c'est en réalité assez « louche ».

Question de goût ?

- « être lisible » → évidemment très subjectif
- un programme est lisible s'il est écrit tel qu'« on » a l'habitude de les lire
- → habitudes communes prises par la plupart des programmeurs Java (d'autres prises par seulement par telle ou telle organisation ou communauté)

Langage de programmation → comme une langue vivante !

Il ne suffit pas de connaître par cœur le livre de grammaire pour être compris des locuteurs natifs (il faut aussi prendre l'accent et utiliser les tournures idiomatiques).

Une hiérarchie de normes

Habitudes dictées par :

1. le compilateur (la syntaxe du Langage de programmation [2])
2. l'éventuel guide [3] de style publié par l'éditeur du langage
   (pour Java, guide publié par Sun → conventions à vocation universelle pour tout programmeur Java)
3. les directives de son entreprise/organisation
4. les directives propres au projet

… et ainsi de suite (il peut y avoir des conventions internes à un package, à une classe, etc.)

» et enfin… le bon sens ! [4]

2. L'équivalent du livre de grammaire dans l'analogie avec la langue vivante.
3. À rapprocher des avis émis par l'Académie Française ?
4. Mais le bon sens ne peut être acquis que par l'expérience.

Une hiérarchie de normes

Nous prendrons en exemple dans ce cours les conventions associées habituellement au langage Java.

Les grands principes valent cependant pour les autres langages.

Plan

Nommer les entités

(classes, méthodes, variables, ...)

Règles de capitalisation pour les noms (auxquelles on ne déroge pratiquement jamais en Java) :

- ... de classes, interfaces, énumérations et annotations [5] ⟶ `UpperCamelCase`
- ... de variables (locales et attributs), méthodes ⟶ `lowerCamelCase`
- ... de constantes (`static final` ou valeur d'`enum`) ⟶ `SCREAMING_SNAKE_CASE`
- ... de packages ⟶ tout en minuscules sans séparateur de mots [6]. Exemple : `com.masociete.biblitothequetruc` [7].

⟶ rend possible de reconnaître à la première lecture quel genre d'entité un nom désigne.

5. c.-à-d. tous les types référence
6. « _ » autorisé si on traduit des caractères invalides, mais pas spécialement encouragé
7. pour une bibliothèque éditée par une société dont le nom de domaine internet serait `masociete.com`

Nommer les entités

Codage, et langue

- **Se restreindre aux caractères suivants :**
  - `a-z`, `A-Z` : les lettres minuscules et capitales (non accentuées),
  - `0-9` : les chiffres,
  - `_` : le caractère soulignement (seulement pour `snake_case`).

  Explication :
  - `$` (dollar) est autorisé mais réservé au code automatiquement généré ;
  - les autres caractères ASCII sont réservés (pour la syntaxe du langage) ;
  - la plupart des caractères unicode non-ASCII sont autorisés (p. ex. caractères accentués), mais aucun standard de codage imposé pour les fichiers `.java`.[8]
- **Interdits :** commencer par `0-9` ; prendre un nom identique à un mot-clé réservé.
- **Recommandé :** Utiliser l'Anglais américain (pour les noms utilisés dans le programme **et** les commentaires **et** la Javadoc).

8. Or il en existe plusieurs. En ce qui vous concerne : il est possible que votre PC personnel et celle de la salle de TP n'aient pas le même réglage par défaut → incompatibilité du code source.

Nommer les entités

Nature grammaticale (1)

Nature grammaticale des identifiants :

- types (noms des classes et interfaces) : nom au singulier
  ex : `String`, `Number`, `List`, ...
- classes-outil (= conteneurs, non instantiables, de membres statiques) : nom au pluriel
  ex : `Arrays`, `Objects`, `Collections`, ... [9]
- variables : nom, singulier sauf pour collections (souvent nom pluriel) ; et booléens (souvent adjectif ou verbe au participe présent ou passé). ex :

```
int count = 0; // noun (singular)
boolean finished = false; // past participle
while (!finished) {
    finished = ...;
    ...
    count++;
}
```

9. attention, il y a des contre-exemples au sein même du JDK : `System`, `Math`... oh !

Nommer les entités

Nature grammaticale (2)

Les noms de méthodes contiennent généralement **un verbe**, qui est :

- `get` si c'est un accesseur en lecture (« getteur ») ; ex : `String getName();`
- `is` si c'est un accesseur en lecture d'une propriété booléenne ;
  ex : `boolean isInitialized();`
- `set` si c'est un accesseur en écriture (« setteur ») ;
  ex : `void getName(String name);`
- tout autre verbe, à l'indicatif, si la méthode retourne un booléen (méthode prédicat) ;
- à l'impératif [10], si la méthode effectue une action avec **effet de bord** [11]
  `Arrays.sort(myArray);`
- au participe passé si la méthode retourne une version transformée de l'objet, sans modifier l'objet (ex : `list.sorted()`).

10. ou infinitif sans le « to », ce qui revient au même en Anglais
11. c.-à-d. mutation de l'état ou effet physique tel qu'un affichage ; cela s'oppose à fonction pure qui effectue juste un calcul et en retourne le résultat

Nommer les entités

Concision versus information

- Pour tout identificateur, il faut trouver le bon compromis entre information (plus long) et facilité à l'écrire (plus court).
- Plus l'usage est fréquent et local, plus le nom est court.
  exemple typique, variables de boucle :
  ```
  for (int idx = 0; idx < anArray.length; idx++) { ... }
  ```
- Plus l'usage est loin de la déclaration, plus le nom doit être informatif.
  (concerne notamment : classes, membres publics… mais aussi les paramètres des méthodes !)
  ex. : paramètres de constructeur
  ```
  public Rectangle(
    double centerX,
    double centerY,
    double width,
    double length
  ) { ... }
  ```

Toute relecteur s'attend à cette stratégie → ne pas l'appliquer peut l'induire en erreur.

Plan

Nombre de caractères par ligne

- On limite généralement le nombre de caractères par ligne de code.
  Raisons :
  - tous les programmeurs n'utilisent pas le retour à la ligne automatique [12] ;
  - la coupure automatique ne se fait pas forcément au meilleur endroit ;
  - les longues lignes illisibles pour le cerveau humain
    (même si entièrement affichées) ;
  - certains programmeurs aiment pouvoir afficher 2 fenêtres côte à côte ;
  - les « diffs » sont plus lisibles (p. ex. commande `git diff`).
- Limite traditionnelle : 70 caractères/ligne
  (les vieux terminaux avaient 80 colonnes [13]).
  De nos jours (écrans larges, haute résolution), 100-120 est plus raisonnable [14].

12. Et historiquement, les éditeurs de texte n'avaient pas le retour à la ligne automatique.
13. Pourquoi 80 ? C'est le nombre du de colonnes dans le standard de cartes perforées d'IBM inventé en… 1928 ! Et pourquoi ce choix en 1928 ? Parce que les machines à écrire avaient souvent 80 colonnes… bref c'est un héritage très ancien !
14. Selon moi, mais attention, c'est un sujet de débat houleux !

Nombre de caractères par ligne

- Arguments contre des lignes trop petites :
  - découpage trop élémentaire rendant illisible l'intention globale du programme ;
  - incitation à utiliser des identifiants plus courts pour pouvoir écrire ce qu'on veut en une ligne ($\rightarrow$ identifiants peu informatifs, mauvaise pratique).

Indentation

- **Indenter** = mettre du blanc en tête de ligne pour souligner la structure du programme. Ce blanc est constitué d'un certain nombre d'**indentations**.
- En Java, typiquement, 1 indentation = 4 espaces (ou 1 tabulation).
- Le nombre d'indentations est égal à la profondeur syntaxique du début de la ligne $\simeq$ nombre de paires de symboles [15] ouvertes mais pas encore fermées. [16]
- Tout éditeur raisonnablement évolué sait indenter automatiquement (règles paramétrables dans l'éditeur). Pensez à demander régulièrement l'indentation automatique, afin de vérifier qu'il n'y a pas d'erreur de structure !

Exemple :

```
voici un exemple (
    qui n'est pas du Java;
    mais suit ses "conventions
        d'indentation"
)
```

15. Parenthèses, crochets, accolades, guillemets, chevrons, ...
16. Pas seulement : les règles de priorité des opérations créent aussi de la profondeur syntaxique.

Où couper les lignes

- On essaye de privilégier les retours à la ligne en des points du programme « hauts » dans l'arbre syntaxique ($\rightarrow$ minimise la taille de l'indentation).
  P. ex., dans « $(x + 2) * (3 - 9/2)$ », on préfèrera couper à côté de « $*$ » $\rightarrow$

```
(x + 2)
* (3 - 9 / 2)
```

- Parfois difficile à concilier avec la limite de caractères par ligne $\rightarrow$ compromis nécessaires.
- $\rightarrow$ pour le lieu de coupure et le style d'indentation, essayez juste d'être raisonnable et consistent. Dans le cadre d'un projet en équipe, se référer aux directives du projet. [17]

17. Cf. querelles de clocher sur le retour à la ligne avant ou après l'accolade ouvrante...

Taille des classes

Quelle est la bonne taille pour une classe ?

- Déjà, plusieurs critères de taille : nombre de lignes, nombre de méthodes, ...
- Le découpage en classes est avant tout guidé par l'abstraction objet retenue pour modéliser le problème qu'on veut résoudre.
- En pratique, une classe trop longue est désagréable à utiliser. Ce désagrément traduit souvent une décomposition insuffisante de l'abstraction. [18]
- Conseil : se fixer une limite de taille et décider, au cas par cas, si et comment il faut « réparer » les classes qui dépassent la limite (cela incite à améliorer l'aspect objet du programme).
- En général, pour un projet en équipe, suivre les directives du projet.

18. Le « S » de « SOLID » : single reponsibility principle/principe de responsabilité unique.

Taille des méthodes

- Pour une méthode, la taille est le nombre de lignes.
- Principe de responsabilité unique [19] : une méthode est censée effectuer une tâche précise et compréhensible.
  → Un excès de lignes
  - nuit à la compréhension ;
  - peut traduire le fait que la méthode effectue en réalité plusieurs tâches probablement séparables.
- Quelle est la bonne longueur ?
  - Un critère : on ne peut pas bien comprendre une méthode si on ne peut pas la parcourir en un simple coup d'œil
    → faire en sorte qu'elle tienne en un écran ($\sim$ 30-40 lignes max.)
  - En général, suivre les directives du projet.

Quand une méthode ne fait qu'une seule chose et qu'elle est bien nommée, on n'a plus besoin de commenter chacun de ses appels (en effet, ce qu'on fait à cette ligne devient évident juste en lisant le code).

19. Oui, là aussi !

Nombre de paramètres des méthodes

Autre critère : le nombre de paramètres. Trop de paramètres (>4) implique :

- Une signature longue et illisible.
- Une utilisation difficile (« ah ce paramètre là, il était en 8ème ou en 9ème position, déjà ? »)

Il est souvent possible de réduire le nombre de paramètres

- en utilisant la surcharge (méthodes de même nom, mais signatures différentes),
- ou bien en séparant la méthode en plusieurs méthodes plus petites (en décomposant la tâche effectuée),
- ou bien en passant des objets composites en paramètre
  ex : un `Point p` au lieu de `int x, int y`.
  Voir aussi : patron « monteur » (le constructeur prend pour seul paramètre une instance du `Builder`).

Et ainsi de suite

- Pour chaque composant contenant des sous-composants, la question « combien de sous-composants ? » se pose.
- « Combien de packages dans un projet (ou module) ? »
  « Combien de classes dans un package ? »
- Dans tous les cas essayez d'être raisonnable et homogène/consistent (avec vous-même… et avec l'organisation dans laquelle vous travaillez).

Plan

Commentaires

Plusieurs sortes de commentaires (1)

- En ligne :

```
int length; // length of this or that
```

  Pratique pour un commentaire très court tenant sur une seule ligne (ou ce qu'il en reste...)

- en bloc :

```
/*
 * Un commentaire un peu plus long.
 * Les "*" intermédiaires ne sont pas obligatoires, mais Eclipse
 * les ajoute automatiquement pour le style. Laissez-les !
 */
```

  À utiliser quand vous avez besoin d'écrire des explications un peu longues, mais que vous ne souhaitez pas voir apparaître dans la documentation à proprement parler (la Javadoc).

Commentaires

Plusieurs sortes de commentaires (2)

- en bloc Javadoc :

```
/**
* Returns an expression equivalent to current expression, in which
* every occurrence of unknown symbol was substituted by the
* expression specified by parameter by.
*
* @param symbol symbol that should be substituted in this expression
* @param by     expression by which the symbol should be substituted
* @return       the transformed expression
*/
Expression subst(UnknownExpr var, Expression by);
```

Plan

Patrons de conception (1)

ou design patterns

- Analogie langage naturel : patron de conception = figure de style
- Ce sont des stratégies standardisées et éprouvées pour arriver à une fin.
  ex : créer des objets, décrire un comportement ou structurer un programme
- Les utiliser permet d'éviter les erreurs les plus courantes (pour peu qu'on utilise le bon patron !) et de rendre ses intentions plus claires pour les autres programmeurs qui connaissent les patrons employés.
- Connaître les noms des patrons permet d'en discuter avec d'autres programmeurs. [20]

20. De la même façon qu'apprendre les figures de style en cours de Français, permet de discuter avec d'autres personnes de la structure d'un texte...

Patrons de conception (2)

ou design patterns

- Quelques exemples : adaptateur, décorateur, observateur/observable, monteur, fabrique, visiteur, stratégie, ...
  Vous en rencontrerez un certain nombre dans votre cursus.
- Patrons les plus connus décrits dans le livre du « Gang of Four » (GoF) [21]
- Les patrons ne sont pas les mêmes d'un langage de programmation à l'autre :
  - les patrons implémentables dépendent de ce que la syntaxe permet
  - les patrons utiles dépendent aussi de ce que la syntaxe permet :
    quand un nouveau langage est créé, sa syntaxe permet de traiter simplement des situations qui autrefois nécessitaient l'usage d'un patron (moins simple).
    Plusieurs concepts aujourd'hui fondamentaux (comme les « classes », comme les énumérations, ...) ont pu apparaître comme cela.

21. E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns : Elements of Reusable Object-oriented Software, 1995, Addison-Wesley Longman Publishing Co., Inc.

Plan

La conception d'un logiciel

Toute construction humaine significativement complexe à besoin d'être pensée avant d'être réalisée.

Voici les deux questions importantes de la phase de conception :

Quoi faire ? et Comment le faire ?

- L'analyse fonctionnelle répond à "Quoi faire".
- La conception architecturale répond à "Comment le faire".

L'analyse fonctionnelle

Pour l'analyse fonctionnelle, on doit répondre aux questions suivantes :

1. À quel(s) problème(s) nouveau(x) doit répondre le logiciel ?
   Pour répondre à cette question, on cherche à comprendre le besoin qui existe et qui nécessite la création d'un nouveau logiciel. Il s'agit d'innover en trouvant un problème que les (futurs) utilisateurs ont sans le savoir ou en supposant que l'on ne peut pas le résoudre à l'aide d'un logiciel. Ce sont des problèmes pour lesquels il faut inventer une solution.
2. À quel(s) problème(s) important(s) doit répondre le logiciel ?
   Ce sont des problèmes qu'il faudra résoudre pour répondre aux problèmes précédents. Peut-être que certains de ces problèmes sont déjà résolus.
3. Comment se décomposent ces problèmes ?
   Tant qu'un problème n'est pas décomposé en sous-problèmes pour lesquels on connaît (ou entrevoit suffisamment précisément) une solution, il faut continuer à le découper en problèmes plus simples !

L'analyse fonctionnelle du projet `Pac-Man`

1. À quel(s) problème(s) nouveau(x) doit répondre le logiciel ?
   P_1 : Rendre attrayant, selon des critères de 2022, un jeu de 1980.
2. À quel(s) problème(s) important(s) doit répondre le logiciel ?
   P_2 : Rénover le graphisme.
   P_3 : Renouveler et diversifier les mécaniques de jeu.
   P_4 : Permettre de s'opposer à des adversaires intéressants.
3. Comment se décomposent ces problèmes ?
   P_2 : doit être précisé $\rightarrow$ P_2 est plus un travail de structuration du code qu'un travail d'artiste !
   $P_{2,1}$: objectivement améliorer le graphisme grâce des effets fournis par JavaFX. [22]
   $P_{2,2}$: permettre la personnalisation (couleurs, textures, polices, ...) ! Il faut une gestion efficace des configurations, et pourquoi pas un moteur de thèmes échangeables ?
   P_3 : cas de figure similaire à P_1 (doit être précisé, nécessitera travail architectural)
   P_4 : plusieurs solutions (non-exclusives)
   $P_{3,1}$: développer une IA qui sera un adversaire convaincant.
   $P_{3,2}$: permettre le multi-joueur en réseau (sur un seul PC on atteint vite les limites du pratique !)

22. qui ne pouvaient pas être exécutés par le matériel de l'époque

Validation de l'explicitation des problèmes

A-t-on oublié quelque chose ?

Poursuite de l'analyse fonctionnelle

L'étape suivante consiste à mieux délimiter la **frontière** du système :

De quoi le logiciel est-il responsable ?
De quoi n'est-il pas responsable ?
Quels sont les acteurs [23] du système ?

Produits possibles de cette phase :

- Un manuel utilisateur.
- Des "user stories" : des cas d'utilisation du logiciel par des utilisateurs.
- Un cahier des charges, il contient des **exigences** :
  - Exigence **fonctionnelle** : ce que calcule le système.
  - Exigence **non fonctionnelle** : comment le système fonctionne.
  - Contraintes : propriétés externes imposées.

23. Les entités qui intéragissent avec lui.

À vous de jouer !

10 minutes en binôme pour proposer une analyse fonctionnelle pour :

Une application pour résoudre le problème de **la charge mentale**.

La conception architecturale

On peut s'appuyer sur la décomposition fonctionnelle pour concevoir une **architecture du logiciel**, qui répond à la question "Comment faire ?"

Architecture logiciel

L'architecture d'un logiciel est l'ensemble des éléments qui le composent, leur responsabilité et leur interrelation.

C'est une sorte de "vue d'avion" du logiciel pour comprendre quels sont ses composants principaux et pourquoi ces composants sont effectivement une solution au problème posé par le logiciel.

Comment architecturer un logiciel ?

- En respectant le principe de **responsabilité unique** :
  Je dois pouvoir décrire en une phrase la responsabilité d'un composant. Sa contribution dans la résolution du problème doit être originale et claire.
- En maximisant les **propriétés de qualité** du logiciel :
  la modularité, la maintenabilité, la simplicité, l'extensibilité, la réutilisabilité, l'intégrité, la robustesse, la portabilité, la compatibilité, ...
- En s'inspirant des **grands styles architecturaux**.

Les grands styles architecturaux

- Architecture hiérarchique
- Architecture par couches
- Architecture centrée sur les données
- Architecture par flots de données
- Architecture par objets
- Architecture par micro-services

Les grands styles architecturaux

- Architecture hiérarchique
  le logiciel a un module “racine” qui utilise des sous-modules, qui utilisent à leur tour des sous-sous-modules, etc, ...
  Exemple : Les utilitaires UNIX.
- Architecture par couches
- Architecture centrée sur les données
- Architecture par flots de données
- Architecture par objets
- Architecture par micro-services

Les grands styles architecturaux

- Architecture hiérarchique
- Architecture par couches
  Le système est une empilement de couches. La couche n est uniquement utilisée par la couche $n+1$ et utilise uniquement la couche $n-1$.
  Exemple : Modèle OSI.
- Architecture centrée sur les données
- Architecture par flots de données
- Architecture par objets
- Architecture par micro-services

Les grands styles architecturaux

- Architecture hiérarchique
- Architecture par couches
- Architecture centrée sur les données
  Un module central s'occupe de la gestion des données. Les modules satellites interagissent à travers le module de gestion des données.
  Exemple : Architecture modèle-vue-contrôleur.
- Architecture par flots de données
- Architecture par objets
- Architecture par micro-services

Les grands styles architecturaux

- Architecture hiérarchique
- Architecture par couches
- Architecture centrée sur les données
- Architecture par flots de données
  Le logiciel prend la forme d'un circuit : chaque module attend en entrée les sorties d'un ou plusieurs autres modules.
  Exemple : Compilateur, apprentissage automatique, traitement BigData.
- Architecture par objets
- Architecture par micro-services

Les grands styles architecturaux

- Architecture hiérarchique
- Architecture par couches
- Architecture centrée sur les données
- Architecture par flots de données
- Architecture par objets
  Le logiciel est formé d'objets qui collaborent en s'envoyant des messages.
  Exemple : Logiciel bureautique, progiciel.
- Architecture par micro-services

Les grands styles architecturaux

- Architecture hiérarchique
- Architecture par couches
- Architecture centrée sur les données
- Architecture par flots de données
- Architecture par objets
- Architecture par micro-services
  Le logiciel est l'union de petits services autonomes.
  Exemple : Netflix.

Conception architecturale de `Pac-Man`

Quel est le style architectural suivi par `Pac-Man` ?

À vous de jouer !

10 minutes en binôme :

Proposez une architecture logicielle pour résoudre la charge mentale.

Plan

Ce chapitre

Comment devenir un meilleur développeur ?

... quelques conseils...

Ce chapitre

Comment devenir un meilleur développeur ?

... quelques conseils...

1. Comment bien travailler dans de bonnes conditions ?
2. Comment bien collaborer ?
3. Comment bien programmer ?
4. Comment progresser ?

La plupart de ces conseils sont inspirés ou tirés de :

"The pragmatic programmer, your journey to mastery"
– Hunt / Thomas

(réédition de 2020 de "The pragmatic programmer, from journeyman to master", 1999)

Plan

L'importance des outils

Utilisez les bons outils !
Connaissez-les !

L'importance des outils

Utilisez les bons outils !
Connaissez-les !
Connaissez les outils en ligne de commande pour automatiser.
En mode graphique, connaissez les raccourcis clavier.
Dans IntellIJ IDEA, pensez au plugin Key Promoter X.

Une chose à la fois

Concentrez-vous !
Comprenez ce que vous avez à faire !

Une chose à la fois

Concentrez-vous !
Désactivez les notifications…
Utilisez le mode « zen » de votre IDE !
Comprenez ce que vous avez à faire !

Une course de fond

Dormez !
Repoussez les deadlines !

Une course de fond

Dormez !
Repoussez les deadlines !
Donnez-vous le temps de bien faire.
Soulagez votre charge mentale en espaçant les jalons indépendants.

De la bonne paresse

Automatisez les tâches répétitives de développement !
Utilisez et écrivez des générateurs de code !

De la bonne paresse

Automatisez les tâches répétitives de développement !

Écrivez des scripts shell.

Utilisez gradle.

Pensez a l'IC.

Utilisez et écrivez des générateurs de code !

Utilisez les modèles/templates de votre IDE !

Écrivez les vôtres.

Restez à jour

Terminez ce que vous commencez !

Restez à jour

Terminez ce que vous commencez!
On compte sur vous...
et c'est bon pour l'estime de soi!

Plan

Soyez honnêtes

The greatest of all weaknesses is the fear of appearing weak.
J. B. Bossuet, Politics from Holy Writ, 1709

Prenez vos responsabilités !
Admettez vos erreurs !
Explicitez vos faiblesses !

puis

Proposez des solutions, pas des excuses.

L'histoire de la soupe aux cailloux

Soyez un catalyseur, pas un "boulet"!
Gardez un oeil sur la vue d'ensemble du projet!

Votre message

Communiquez !
Réfléchissez à ce que vous voulez dire !
Réfléchissez à comment vous voulez dire !
Mettez-vous à la place de votre interlocuteur !

Plan

Du perfectionnisme (version positive)

Ne laissez pas une fenêtre cassée dans votre code !

Du perfectionnisme (version négative)

Le mieux est l'ennemi du bien !
Apprenez à fixer un niveau de qualité suffisant pour votre projet !

Du perfectionnisme (version négative)

Le mieux est l'ennemi du bien !

YAGNI ?

Apprenez à fixer un niveau de qualité suffisant pour votre projet !

« loi des 80% »

De l'endurance

Retravaillez votre code !

Du débogage

Reproduisez les bugs !
Ne supposez rien, prouvez-le !
Ecrivez du code qui s'observe !
Ecrivez du code qui se teste facilement !

Du débogage

Reproduisez les bugs !
Cf. cours sur le test...
Ne supposez rien, prouvez-le !
Ecrivez du code qui s'observe !
Mais pour ce qui ne s'observe pas, pensez au débogueur.
Ecrivez du code qui se teste facilement !
Par exemple, petites méthodes et petits objets.

C'est tombé en marche !

Programmez en pleine conscience !

Programmer, c'est structurer de la connaissance

Ne dupliquez pas la connaissance !
Ne dispersez pas la connaissance !
Représentez la connaissance dans des fichiers textes !

Mais au fait… de quelle connaissance s'agit-il ?

Programmer, c'est structurer de la connaissance

Ne dupliquez pas la connaissance !
DRY : do not repeat yourself !
Ne dispersez pas la connaissance !
Représentez la connaissance dans des fichiers textes !

Mais au fait… de quelle connaissance s'agit-il ?
Votre connaissance par rapport au projet…
Mais aussi les données traitées par le programme !

Embrasser le changement

Apprêtez-vous à tout jeter !

Atteindre le but, très vite

Construisez d'abord le squelette complet du produit !
Programmez incrémentalement !
Faites, si vous savez comment faire !

Atteindre le but, très vite

Construisez d'abord le squelette complet du produit !
« tracer bullet code »
Programmez incrémentalement !
une des justifications des méthodes agiles
Faites, si vous savez comment faire !

Prendre le temps de comprendre.

Prototypez mais pas forcément avec du code !
Expérimentez les nouvelles technologies !

Prendre le temps d'estimer.

Estimez les ordres de grandeurs en espace et en temps !

On programme pour des humains

Communiquez avec votre code!
Introduisez le bon langage pour exprimer votre solution!
Utilisez le vocabulaire du domaine du problème!
Faites passer des interrogatoires à vos utilisateurs!

On programme pour des humains

Communiquez avec votre code !
Cf. le cours « programmer avec style ».
Introduisez le bon langage pour exprimer votre solution !
Utilisez le vocabulaire du domaine du problème !
Faites passer des interrogatoires à vos utilisateurs !

On programme avec des humains

Soyez paranoïaque !
Etablissez des contrats d'usage de vos composants logiciels !

On programme avec des humains

Soyez paranoïaque !

Pensez à **`final`**, **`private`**, ... aux exceptions.

Etablissez des contrats d'usage de vos composants logiciels !

La modularisation

Suivez la loi de Demeter !
Configurez, n'intégrez pas !
Mettez des abstractions dans le code, des détails dans les données !
Structurer l'enchaînement des calculs !

La modularisation

Suivez la loi de Demeter !
Configurez, n'intégrez pas !
Préférez des composants largement indépendants les uns des autres et laissez la classe principale les instancier en les associant les uns aux autres (en fonction des options de configuration.
Mettez des abstractions dans le code, des détails dans les données !
Structurer l'enchaînement des calculs !

La loi de Demeter ?

... ou Principe de connaissance minimale : « Ne parlez qu'à vos amis immédiats. »

Éviter les appels du genre `x.getComponentA().getCompontentB().doSomething()`.

Une méthode M d'un objet O ne devrait invoquer que des méthodes :

- de O
- des attributs de O
- des paramètres de M
- des objets instanciés par M

Si ça ne suffit pas, c'est que

- soit les abstractions des objets ci-dessus sont insuffisantes (des méthodes manquent)
- soit M essaye d'en faire trop

Attention, ce n'est qu'un conseil.
Appliquer la loi de Démeter à la lettre peut aussi poser des problèmes (lesquels ?).

Antipatrons et mauvaises odeurs

Antipatron (antipattern) : mauvaise pratique de développement provoquant des problèmes (bugs, performance, lenteurs de développement, ...)

Mauvaise odeur (code smell) : mauvaise pratique de programmation favorisant l'apparition d'erreurs (différence : il n'y a pas forcément de problème actuellement... mais l'apparition d'erreur est favorisée tant que le code smell n'est pas retiré).

Dans quelle catégorie mettre une violation de la loi de Démeter ?
Connaissez-vous d'autres exemples ?
Savez-vous les détecter ?

Quelques mauvaises pratiques de codage

- God Object : objet qui en sait trop ou qui en fait trop
- Duplication de code
- Programmation spaghetti : programme dont le flux de contrôle est peu clair (beaucoup d'appels imbriqués, retours par exceptions rattrapées à des endroits inattendus, etc.)
- Hardcoding
- Feature Creep
- Noms mystérieux
- Feature envy (classe qui utilise systématiquement les fonctionnalités d'une autre)
- ...

Lesquelles sont des antipatrons, lesquelles sont de symples mauvaises odeurs ?

Mais alors, les bonnes pratiques ?

En POO, il faut être SOLID :

- S : Single responsibility
- O : Open/closed
- L : Liskov substitution
- I : Interface segregation
- D : Dependency inversion

Il existe d'autres acronymes avec des idées du même genre. Ce cours ne prétend pas à l'exhaustivité !

Mais alors, les bonnes pratiques ?

En POO, il faut être SOLID :

- S : Single responsibility
  Une seule responsabilité par classe ou méthode.
- O : Open/closed
  Quand une classe fonctionne, s'interdire de la modifier. Ajouter des nouvelles fonctionnalités par extension.
- L : Liskov substitution
  Quand un objet d'un type donné est attendu dans une partie correcte du programme, fournir une instance de n'importe quelle de **vos** sous-classes garantit un comportement cohérent.
- I : Interface segregation
  Chaque interface est simple et correspond à un but précis (permettant d'exposer juste les fonctionnalités nécessaires là où c'est nécessaire).
- D : Dependency inversion
  Dépendre des abstractions (écrites par nous), pas des implémentations (écrites par des tiers).

Il existe d'autres acronymes avec des idées du même genre. Ce cours ne prétend pas à l'exhaustivité !

La complexité des logiciels

Supprimez du code !
Gardez-le stupide et simple !
Luttez contre la complexité avec de la généralité !
Soyez pertinents !

La complexité des logiciels

Supprimez du code !

Ici aussi, DRY !

Mais aussi, ne pas hésiter à jeter ce qui n'est plus utile.

Gardez-le stupide et simple !

KISS

Luttez contre la complexité avec de la généralité !

Encore DRY…

Soyez pertinents !

Plan

Que sais-je ?

Investissez dans vos connaissances !
Fixez-vous des compétences à acquérir à courts et à moyens termes !
N'utilisez jamais du code que vous ne comprenez pas !

Connaissez-vous le culte du cargo ?

Extraire l'information

Posez des questions !
Lisez du code !

Plan

Résumé

Au risque d'être un peu normatif, un bon développeur :

- **s'adapte rapidement** à de nouvelles technologies, de nouvelles idées, de nouvelles équipes ;
- **veut comprendre** et pose donc continuellement des questions pour savoir "comment cela fonctionne" ;
- **a un esprit critique**, ne se contente pas d'un "c'est comme ça !" et cherche des justifications ;
- **est réaliste**, en essayant d'affronter les difficultés sans les éluder ;
- **a un large spectre de compétences** qui lui permettent de construire une compréhension profonde et une approche raisonnée des problèmes de développement logiciel.

mais surtout le plus important :

- Un bon développeur accorde une grande importance à ce qu'il fait.
- Un bon développeur réfléchit à ce qu'il fait.

Pour résumer

REFLECHISSEZ !