Conduite de projet

Aldric Degorre
(IRIF, U-Paris) – adegorre@irif.fr

27 octobre 2023

Plan

Plan

Documenter ?

On vous demande toujours de documenter vos projets. Mais de quoi parle-t-on ?

- Qu'est-ce qu'une documentation ?
- Que faut-il documenter (de quoi parle-t-elle ?) ?
- Qui en est le destinataire ?

Documenter ?

On vous demande toujours de documenter vos projets. Mais de quoi parle-t-on ?

- Qu'est-ce qu'une documentation ?
  $\rightarrow$ Un ensemble de documents en rapport avec le logiciel développé. (Nous verrons lesquels un peu plus loin)
- Que faut-il documenter (de quoi parle-t-elle ?) ?
- Qui en est le destinataire ?

Documenter ?

On vous demande toujours de documenter vos projets. Mais de quoi parle-t-on ?

- Qu'est-ce qu'une documentation ?
  → Un ensemble de documents en rapport avec le logiciel développé. (Nous verrons lesquels un peu plus loin)
- Que faut-il documenter (de quoi parle-t-elle ?) ?
  → Voir plus loin.
- Qui en est le destinataire ?

Documenter ?

On vous demande toujours de documenter vos projets. Mais de quoi parle-t-on ?

- Qu'est-ce qu'une documentation ?
  → Un ensemble de documents en rapport avec le logiciel développé. (Nous verrons lesquels un peu plus loin)
- Que faut-il documenter (de quoi parle-t-elle ?) ?
  → Voir plus loin.
- Qui en est le destinataire ?
  → Voir plus loin.
  (En tout cas, ce n'est pas nécessairement la personne qui vous a donné le projet...)

Mais qu'obtient-on ?

- Souvent on vous demande au début de rédiger des documents dans les phases initiales d'un projet (cahier des charges, spécification fonctionnelle, spécification détaillée ...)
  Si vous l'avez fait, sont-ils à jour ?
  (Est-ce que les joindre à votre rendu a une valeur ajoutée ?)
- Inversement, si vous rédigez quelques documents à la hâte la veille de la deadline.
  La qualité sera-t-elle là ?

L'envers du décors

- Quand c'est vous l'utilisateur et/ou l'installateur, quelle documentation vous est-elle utile ? (Et puis, lisez-vous vraiment sérieusement la documentation ?)
- Question adressée à ceux qui sont arrivés en cours de semestre : qu'est-ce qui vous a le plus aidé à vous intégrer, la documentation ou bien le code ?

De l'utilité et de la destination de la documentation

Certaines réponses peuvent faire penser que la documentation est inutile.

→ Une documentation utile est une documentation rédigée en pensant à qui va la lire !

Que comporte la documentation d'un projet ?

Documenter

De quoi on parle

Que comporte la documentation d'un projet ?

- la Big picture (vision d'ensemble) : descriptif très court du logiciel, destiné à fixer son périmètre en une phrase. (Question : c'est destiné à qui ?)
- documentation utilisateur
- documentation de suivi de projet
- documentation technique

Documentation utilisateur

Celle-ci peut prendre des formes diverses. Pas forcément un manuel !

Regardez les logiciels autour de vous, est-ce que la plupart du temps vous avez vraiment besoin d'ouvrir un document séparé pour avoir de l'aide ?

(Question : quelles formes de documentation utilisateur rencontrez-vous souvent ?)

Documentation de suivi de projet

(Ah, c'est destiné à qui, au fait ?)

Elle est généralement produite pendant le déroulement du projet.

Dans notre cas : le contenu de votre site GitLab fournit cette documentation si vous avez effectivement utilisé correctement la plateforme tout au long du projet.

Documentation technique

Même question, qui est-ce que ça intéresse ?

À quoi peut-elle leur servir ?

Documentation technique

Même question, qui est-ce que ça intéresse ?

À quoi peut-elle leur servir ?

→ c'est pour les développeurs et uniquement eux, qu'ils viennent du projet ou d'ailleurs.

→ cette documentation doit fournir le plus rapidement possible des réponses claires, précises et fiables afin de les aider à ajouter plus vite de nouvelles fonctionnalités.

On peut généralement distinguer :

- la documentation d'architecture
- la documentation exhaustive des API

La première doit généralement écrite « à la main » et s'illustre de divers diagrammes et schémas.

La seconde est idéalement générée depuis le code source annoté.

Plan

Documentation des API

API : application programming interface C'est l'« interface »[1] de votre code, la partie avec laquelle on intéragit pour utiliser votre code dans un autre projet.

Quelles qualités pour une bonne documentation des API ?

1. entendre « partie émergée de l'iceberg »

Documentation des API

API : application programming interface C'est l'« interface »[1] de votre code, la partie avec laquelle on intéragit pour utiliser votre code dans un autre projet.

Quelles qualités pour une bonne documentation des API ?

- elle doit être synchrone (tout le temps à jour)
  → elle doit être écrite en même temps que le code
- lisible depuis le code et depuis l'extérieur (multidiffusion)
  → elle doit être écrite dans le code (commentaires)... sous une forme qui peut être traitée automatiquement pour générer un document séparé

1. entendre « partie émergée de l'iceberg »

Documenter le code

Pourquoi

Mais un code beau et compréhensible ne vaut-il pas mieux que des commentaires ou une documentation ?

Oui, mais...

2. Rappelez-vous du conseil d'un des cours précédents : il faut être parano et toujours décrire clairement le cadre dans lequel une méthode peut être utilisée. Et bien c'est dans la documentation des API qu'il faut l'écrire.

Documenter le code

Pourquoi

Mais un code beau et compréhensible ne vaut-il pas mieux que des commentaires ou une documentation ?

Oui, mais...

- écrire un code clair est plus facile à dire qu'à faire
- un bon commentaire peut être bien plus synthétique...
  ... et les IDE savent afficher la documentation dans les info-bulles quand on survole un nom de méthode ou de classe (corolaire : c'est bien quand ça tient dans la bulle !)
- le code ne peut pas exprimer tous les contrats [2]

2. Rappelez-vous du conseil d'un des cours précédents : il faut être parano et toujours décrire clairement le cadre dans lequel une méthode peut être utilisée. Et bien c'est dans la documentation des API qu'il faut l'écrire.

Javadoc

En Java, le système de documentation des API s'appelle Javadoc.

Il consiste en :

- une syntaxe adéquate de commentaire pour générer la documentation
- une commande « `javadoc` » qui génère la documentation sous forme de pages HTML depuis les commentaires au format Javadoc.

Javadoc – Syntaxe

Un exemple de commentaire 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 symbol, Expression by);
```

Javadoc – Syntaxe

Plus précisément

Tout commentaire de Javadoc est de cette forme :

```
/**
  * Un texte descriptif.
  * Celui-ci peut contenir <b>des tags HTML</b>.
  * On y fait une présentation générale,
  * on y exprime les contrats,
  * mais on peut aussi ajouter tout élément technique
  * nécessaire à la compréhension du bon fonctionnement.
  *
  * @tag1 un premier tag javadoc dont voici le descriptif.
  * @tag2 bla bla bla
  * @tag4 à noter qu'on peut aussi insérer des {@tagenlignes}
  * @tag3 etc.
  */
```

Javadoc – Syntaxe

Plus précisément

Tags les plus courants :

- `@author` : auteur de l'élément
- `@param` (suivi du nom du paramètre) : description d'un paramètre de méthode
- `@return` : description de la valeur de retour pour une méthode
- `@throws` (suivi du nom de l'exception) : description d'une exception susceptible d'être déclenchée par la méthode
- `@see` : mettre en relation avec la doc d'un autre élément
- `@deprecated` : signale que l'élément est obsolète (on doit expliquer par quoi le remplacer)

Javadoc

intégration dans les IDE

Exemple dans l'IDE Eclipse :

- Pour générer la documentation HTML : menu « Project », « Generate Javadoc… ».
- Quand on est sur une définition de classe/interface/enum/méthode/… , la combinaison de touches `Alt+Shift+J` créée un « squelette » de Javadoc à compléter.

Par ailleurs, le survol à la souris de tout identificateur affiche au bout d'environ une seconde une infobulle avec le contenu de la Javadoc associée à cet identificateur.

(Cela est vrai dans Eclipse, mais aussi dans IntellIJ IDEA ou Visual Studio Code et probablement d'autres…)

Javadoc

Quelles entités faut-il documenter ?

- Il est **indispensable** de documenter tout ce qui est public.
- Il est **fortement recommandé** de documenter tout ce qui n'est pas privé (car utilisable par d'autres programmeurs, qui n'ont pas accès au code source).
- Il est **utile** de documenter ce qui est privé, pour soi-même et les autres membres de l'équipe.

On peut documenter les types (classes, interfaces, enums, ...), les méthodes, les attributs... mais aussi les packages (créer un fichier `package-info.java` dans le répertoire du package, et y insérer le commentaire Javadoc.)

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

3. 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 [4])
2. l'éventuel guide [5] 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 ! [6]

4. L'équivalent du livre de grammaire dans l'analogie avec la langue vivante.
5. À rapprocher des avis émis par l'Académie Française ?
6. 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 [7] ⟶ `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 [8]. Exemple : `com.masociete.biblitothequetruc` [9].

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

7. c.-à-d. tous les types référence
8. « _ » autorisé si on traduit des caractères invalides, mais pas spécialement encouragé
9. 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`.[10]
- **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).

10. 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`, ... [11]
- 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++;
}
```

11. 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 [12], si la méthode effectue une action avec **effet de bord** [13]
  `Arrays.sort(myArray);`
- au participe passé si la méthode retourne une version transformée de l'objet, sans modifier l'objet (ex : `list.sorted()`).

12. ou infinitif sans le « to », ce qui revient au même en Anglais
13. 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[14] ;
  - 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[15]).
  De nos jours (écrans larges, haute résolution), 100-120 est plus raisonnable[16].

14. Et historiquement, les éditeurs de texte n'avaient pas le retour à la ligne automatique.
15. 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 !
16. 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[17] ouvertes mais pas encore fermées.[18]
- 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"
)
```

17. Parenthèses, crochets, accolades, guillemets, chevrons, ...
18. 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. [19]

19. 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. [20]
- 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.

20. 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 [21] : 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).

21. 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. [22]

22. 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) [23]
- 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.

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