Guide de bonnes pratiques en génie logiciel
version complète de travail

mailto:equipe-gl@telecom-bretagne.eu

v0.8

IMT Atlantique

Table des matières

Introduction

Ce document a pour but de recenser les bonnes pratiques de conception et de développement utiles pour mener au mieux un projet logiciel. Les bonnes pratique permettent de produire du code de qualité, plus lisible, mieux organisé, plus facilement testable, plus facilement maintenable. Mais, les bonnes pratiques permettent aussi au développeur de se sentir bien : il est moins effrayant de changer un bout de code quand on sait que les tests nous diront si nous avons une erreur ou si nous sommes sur la bonne voie. Il est plus agréable d'ajouter une fonctionnalité ou de corriger une erreur (un « *bug* ») si ça n'entraîne pas la modification de la moitié du projet. Écrire du *beau* code, utiliser de bonnes pratiques de développement n'est pas uniquement l'affaire du concepteur-développeur puriste. Les bonnes pratiques aident à l'amélioration de la qualité du logiciel et à sa maintenabilité. L'enjeu n'est donc pas de satisfaire une obsession du développeur « *geek* » mais bien de limiter et maîtriser les coûts de maintenance du logiciel (entre autres). En effet, ne pas viser immédiatement la bonne conception (faire du « *quick and dirty* »), c'est engranger de la *dette technique* dont il faut avoir conscience (le choix de la dette technique doit être volontaire) [1].

Dans la préface de [8], Jim Coplien fait référence à l'approche Japonaise de la qualité. Dans les années 50, les Japonais ont mis en place une organisation de leur industrie dont l'objectif est la *maintenance* plutôt que la production. Or, il se trouve que le coût principal du logiciel est dans la maintenance : (1) une erreur est vite arrivée mais la corriger est complexe, (2) plus l'erreur est corrigée tardivement dans le cycle de développement plus elle coûte cher, mais surtout parce que le logiciel possède une longue durée de vie et qu'il doit (3) s'adapter aux technologies qui progressent, et (4) évoluer avec les besoins qui changent [2].

Appliquer cette approche au logiciel est donc pertinent. Pour assurer la maintenabilité, cette organisation repose sur les cinq principes suivants qui sont à la base du *Lean* [5], une approche de management très à la mode. Les concepts du 5S sont :

- *Seiri*, ou avoir de l'organisation (dans le sens de rangement). Savoir où sont les choses, comment elles sont identifiées, bien choisir leurs noms est crucial.
- *Seiton*, ou penser systématiquement. Les décisions sont régulières et les choix systématisés. Par exemple, le code d'un traitement doit être là où l'on s'attend qu'il soit, s'il n'y est pas, il faut réorganiser pour l'y mettre.
- *Seiso*, ou nettoyer (penser à lustrer). Garder son espace de travail propre. Par exemple les commentaires décrivent ce qui est, pas l'histoire du code, ni les souhaits d'évolution.
- *Seiketsu*, ou standardiser. Utilisez des règles (de codage) et des pratiques partagées.
- *Shutsuke*, ou discipline (y compris pour soi-même). Avoir la rigueur d'appliquer les standards, pratiques et principes ici énoncés.

Les bonnes pratiques de programmation découlent plus ou moins directement de la concrétisation de ces principes dans le monde du logiciel. La revue de son propre code est une discipline simple et en accord avec les principes ci-dessus.

Avertissement

Les principes qui suivent ne sont pas exhaustifs ; ils ne doivent pas être appliqués aveuglément ; ils ne fonctionnent pas toujours et sont souvent dépendants du contexte [6] ; ils s'opposent parfois. L'art de l'ingénieur consiste à les connaître, les maîtriser, les adapter (au contexte), les sélectionner ou les rejeter quand il le faut, en étant capable de le justifier.

1. `http://frank.taillandier.me/2014/11/06/intro-dette-technique`
2. Ce contexte complexe rend le développement de logiciels difficile et rapproche parfois le génie logiciel des systèmes chaotiques. C'est pourquoi, un effort d'organisation pour y mettre de l'ordre est indispensable.

Note

Dans la suite, nous utilisons autant que possible les mots issus du glossaire [3]. Ils apparaitront en *italique bleu*.

1 Bonnes pratiques d'organisation (O)

O1 Faire simple

L'organisation du projet doit être aussi simple que possible, adaptée à la taille et à la nature du projet. Elle doit être compréhensible et maîtrisée par toutes les *parties prenantes* du projet.

Appliquer autant que possible le principe KISS (= « *Keep It Simple, Stupid* »[3]), qui rejoint celui du rasoir d'Ockham. Ne pas ajouter de choses inutiles.

O2 Documenter

Il est important de laisser des *traces*. C'est fondamental pour les productions finales et intermédiaires, mais également pour le *processus* de construction avec la justification et l'explication des choix importants. Les *artéfacts* (*programmes*, *modèles*, etc.) sont donc accompagnés de *documentation* et de *commentaires* (voir annexe A) qui justifient l'existence des productions et expliquent tout ce qui est nécessaire.

O3 Suivre des standards

Il est important de respecter des règles :
- discipline de programmation (conventions de nommage, conventions de programmation, « *patterns* » (*patrons*) de programmation, etc.) ;
- discipline de *documentation* ;
- uniformité des *artéfacts* (règles de nommage, documents types) ;
- langage commun (« *English* » pour coder) ;
- standards de la profession (ISO, IEEE, ITU, AFNOR, etc.) ;
- d'utilisation des outils (ex. *intégration continue*, *tests*).

Les standards portent leurs propres justifications ; ils sont le résultat de consensus de professionnels.

O4 Échapper aux standards

Il est important de savoir s'adapter pour :
- donner du sens spécifique ;
- ne pas faire tout *en automate* ;
- prendre en compte le contexte ;
- faire simple (O1).

en étant capable de justifier pourquoi un standard ou une règle commune n'est pas appliquée.

O5 Vérifier, valider

Chaque étape de l'organisation, chaque production doit être vérifiée. A-t-on bien fait ce que l'on avait dit que l'on ferait ? Suit-on le *processus* de développement choisi ?

3. `https://fr.wikipedia.org/wiki/Principe_KISS`

Le produit en cours de fabrication est-il conforme aux exigences exprimées ? Fait-il ce que l'on attend de lui ? Une approche classique consiste à *exécuter souvent* grâce à des *tests* qui sont capitalisés au cours développement.

Plus généralement, tous les *artéfacts* (programmes, *documentations*, *tests*, etc.) produits lors du développement doivent être vérifiés (*vérification*) et validés (*validation*).

O6 Faire relire

Les *artéfacts* produits doivent être relus par des personnes différentes des producteurs (principe de la *revue* de *code*, des audits, etc.). Le *code* doit être relu. Pour faciliter la relecture, le code doit donc être correctement présenté et documenté (O2).

O7 Impliquer l'utilisateur

Les utilisateurs ne savent pas souvent expliquer ce qu'ils veulent *a priori*. Ils le découvrent souvent en *regardant* le *système* fonctionner. Un prototype du logiciel est souvent utile pour mieux concrétiser les idées. Toutefois, c'est un cas possible de *dette technique* dont il faut mesurer les impacts dans le projet.

L'*analyse* des *besoins* et l'élicitation des *exigences* sont en informatique des activités délicates et presque toujours *itératives*.

Les outils utiles sont : cas d'utilisation, histoire d'utilisateurs, maquette, scénarios, diagrammes de séquence...

O8 Itérer (décomposition temporelle)

Les projets (de logiciel) sont généralement itératifs avec, pour chaque itération, 5 phases :

1. Identification des *besoins* (« *requirements* »)
2. Conception (« *design* ») (*Concevoir*)
3. *Réalisation* (« *build* »)
4. *Tests*
5. *Revue* de sa propre production (diagrammes, documentations, codes, etc.) afin de simplifier, corriger, améliorer, rendre plus générique.

Chaque *itération* enrichit les fonctions et *qualités* du produit et l'utilisateur voit le progrès de chaque itération. Il faut produire quelque chose qui marche dès que possible ; *test*er et *intégrer de façon continue*.

Avoir des jalons à court terme et un pilotage par les risques en commençant par les plus élevés.

O9 Décomposer (décomposition spatiale)

Les projets (de logiciel) sont généralement *incrémentaux*. Les *systèmes* sont constitués de sous-systèmes (des *composants*), eux même décomposables.

Les stratégies de décompositions réduisent la complexité de chaque partie (augmentant la *cohérence*), mais introduisent une complexité liée à la coordination (augmentant le *couplage*) des parties.

Il est donc intéressant d'appliquer les principes de forte cohésion et de faible couplage des modules.

Il faut penser à *test*er les parties indépendamment les unes des autres et leur *intégration* dans différents assemblages.

Les outils utiles sont : *architecture*, diagramme de classe, diagramme de collaboration, composition.

O10 Hiérarchiser

Hiérarchisez bien les priorités. Les améliorations de détails sont ajoutées tardivement.

Une classification classique est : MoSCoW, « *Must have, Should have, Could have, Wish have* ». On se concentre sur le « *Must have* ».

Voir aussi **optimiser en dernier** (C10).

O11 Capitaliser

À la fin d'un projet, les acteurs clés prennent le temps d'analyser les problèmes rencontrés et leurs solutions.

Tracer et *documenter* les bonnes et mauvaises décisions/solutions.

Apprendre et partager pour les futurs projets.

O12 Principes de gestion de projet

Définir des objectifs clairs et précis. On doit être capable de dire quand l'objectif est atteint, quand le travail est fini. Définir les objectifs, c'est préparer les *tests* de *validation* (O5).

Hiérarchiser les objectifs. Choisir ce qui est important (O10).

Identifier les responsabilités (des personnes). C'est s'assurer que chaque objectif est suivi par quelqu'un.

Avoir un seul point de responsabilité. C'est s'assurer qu'une seule personne suit et décide.

Déléguer. Pour répartir la charge. Mais déléguer n'est pas s'affranchir de ses responsabilités.

Communiquer. Évidemment

Planifier, planifier. Évidemment

Pouvoir se tromper (Le droit à l'erreur). Tout le monde se trompe ; il faut, par contre, apprendre (O11) et ne pas renouveler les mêmes erreurs.

Il est plus important d'être clair que parfait. On ne peut justifier l'absence de communication ou l'absence de progrès par l'argument : *ce n'est pas fini* ou *ce n'est pas parfait*. Par exemple, on peut « *commiter* » des travaux sur un *code* source, même si des erreurs résiduelles restent. Il est important de partager des progrès, même partiels. [4]

2 Bonnes pratiques de conception (C)

C1 Faire simple

Appliquer autant que possible le principe KISS (= « *Keep It Simple, Stupid* ») qui rejoint celui du rasoir d'Ockham. Ne pas ajouter de choses inutiles.

4. *Perfection & Feedback Loops or : why worse is better*, Marcus Denker, ESUG16, `http://www.esug.org/wiki/pier/Conferences/2016/Friday/0930-1000-feedback-loops?_s=UEaX2D6SoOYFwN9s&_k=bFTw9O_BBUo1xs2v&_n&29` (vidéo et support en anglais)

C2 Documenter

Les productions (*code*, *modèles*, *tests*, etc.) sont accompagnées de *documents* et de *commentaires* (voir annexe A) qui justifient l'existence des productions et expliquent tout ce qui est nécessaire.

C3 La qualité prime

La *qualité* est une propriété émergente. Elle doit être pensée dès le début et intégrée à tous les choix de *conception*. Négliger la qualité puis espérer la retrouver a posteriori est coûteux et difficile.

C4 Chercher la modularité

Identifier des frontières (*modules*, *composants*, *fonctions*, paquetages, etc.) en les justifiant selon le principe **faire simple** (C1).

Une justification classique est la recherche de *responsabilité* : quelle est la responsabilité de cette entité ?

Bien définir les *interfaces* – comment utiliser l'entité identifiée – pour permettre aux autres entités de l'utiliser sans avoir à connaître l'intérieur (sa mise en œuvre précise). On parle d'*encapsulation*. Permet de cacher les détails internes et ainsi de faciliter des évolutions ultérieures de la *réalisation*.

C5 Cacher

Pour chaque *composant*, il faut se poser la question de ce qui est visible de l'extérieur et de ce qui est caché à l'intérieur. Il faut trouver un équilibre entre exposer (*boite blanche*) suffisamment d'information pour pouvoir utiliser le composant et en cacher le plus possible (*boite noire*) pour simplifier l'usage et cacher les détails d'implantation.

Par exemple, une façon classique de décrire une fonction avec précision sans détails d'implantation est de contractualiser (C6.) En programmation objet, on cache les attributs d'une classe en leur donnant une *visibilité* `private`.

C6 Contractualiser

Les *fonctions* sont décrites par un *contrat*.

— *Précondition* : une assertion en entrée de méthode qui protège la fonction et décrit ce que l'appelant devrait respecter.

— *Postcondition* : une assertion en sortie de méthode qui décrit le changement ou le calcul opéré par la méthode.

Contrat	Obligation	Gain
Client	n'appeler que si la *précondition* est valide	être sûr de la *postcondition*
Fonction	satisfaire la *postcondition*	être protégé par *précondition*

La *postcondition* est un *oracle* de *test*.

Quand il s'agit d'une classe ou d'une structure de données, il faut penser à décrire son *invariant* : un énoncé établissant ses propriétés non triviales. Par exemple : les données de la listes sont triée de façon strictement croissante suivant l'information de poids.

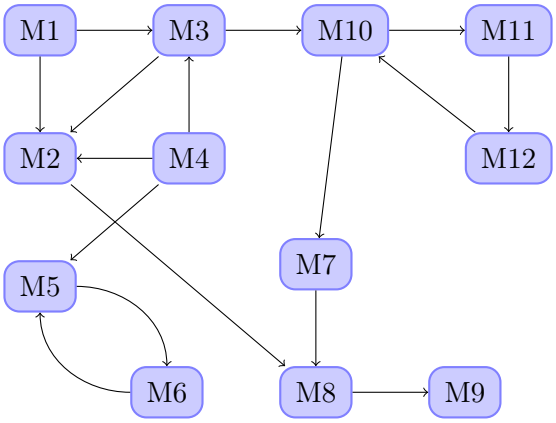

FIGURE 1 – Un exemple de graphe de dépendances

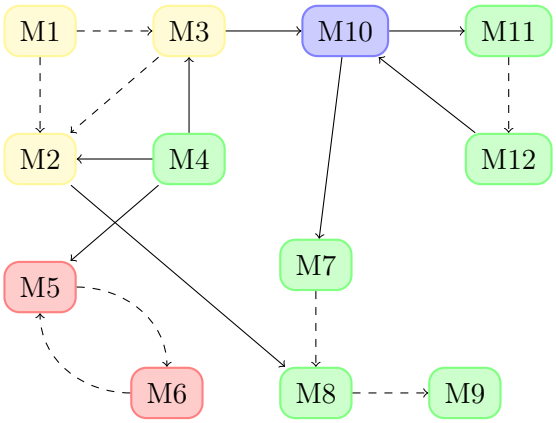

FIGURE 2 – Un mauvais exemple de modularisation : 4 modules (couleurs), 8 dépendances intermodules (traits pleins), 8 intra-modules (pointillés).

C7 Éviter le *code spaghetti*

Identifier les interactions entre entités en cherchant à les réduire en jouant sur la modularité. Chercher à maîtriser et réduire les *couplages*.

C8 Couplage et cohésion

Ces notions sont générales et se rencontrent dans beaucoup de situations en informatique. Par exemple cela peut être des classes, des méthodes ou des paquets qui utilisent ou importent des éléments les uns des autres. Une vue abstraite et simple est de voir cela comme un ensemble d'entités ayant des relations entre elles et représentées par un graphe (figure 1).

Étant donné un regroupement des entités, le *couplage* est l'ensemble des liens intergroupes (ou modules) et la *cohésion* l'ensemble des liens intra-modules.

La figure 2 montre un groupement mais on voit que dans le module B (vert), M4 n'a pas de lien avec M11, M12, M7, M8 et M9. Par ailleurs, le module B a 7 liens avec les autres modules. Cette situation n'est pas satisfaisante et il est possible d'avoir une cohésion plus forte et un couplage plus faible comme dans la figure 3.

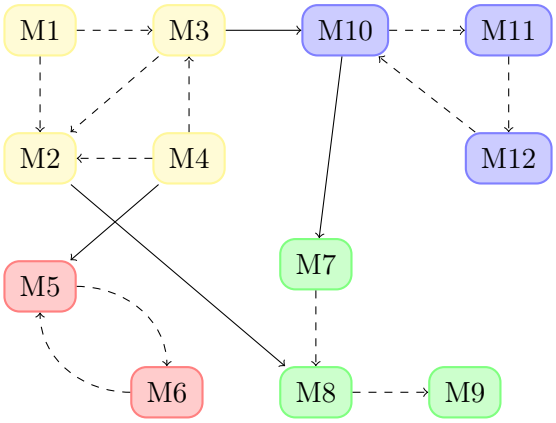

FIGURE 3 – Une meilleur modularisation : 4 modules (couleurs), 4 dépendances intermodules (traits pleins), 12 intra-modules (pointillés).

C9 Réutiliser

La *réutilisation* se décline en deux points de vue :

1. coder en réutilisant ;
2. coder pour la réutilisation.

Le premier consiste à ne pas *réinventer la roue*, profiter du travail déjà accompli et de l'expérience des communautés pour résoudre un problème. On peut réutiliser des unités très différentes : *fonction*, classe, *module*, *composant*, *bibliothèque*, etc.

Le second consiste à produire du code dans le but de favoriser sa réutilisation. On s'appuie en général dans ce cas sur les bonnes pratiques décrites dans ce document, mais aussi sur des *patrons* de conception. En effet, certains problèmes de *conception* logicielle sont maintenant considérés comme classiques et ont une ou plusieurs solutions. Les *patrons* de conception (ou « *design patterns* ») sont des solutions classiques à des problèmes classiques. Il faut savoir les utiliser à bon escient et avec parcimonie. Cette seconde compétence est plus délicate et demande une maîtrise plus grande des bons principes de développement logiciel.

C10 Optimiser en dernier

Les *optimisations* précoces peuvent réduire la modularité d'un produit ainsi que sa lisibilité (et donc sa *maintenance* future). Il faut se concentrer sur l'essentiel – les fonctionnalités – puis, une fois cela atteint, si nécessaire, optimiser.

Dans certains contextes, la performance est au cœur du *besoin*. Dans ce cas, l'optimisation arrive plus tôt dans le développement.

3 Bonnes pratiques de programmation (P)

3.1 Bonnes pratiques « génériques »

Une liste, en partie issue de [17], avec pour chaque point, une motivation, des conseils de mise en œuvre et parfois des objections...

Catégorie - Style et convention

P1 Écrire du code commenté

Le *commentaire*, c'est ce que l'on commence à lire. Si le commentaire est court et pertinent, il facilite la *maintenance*.

Le commentaire peut avoir plusieurs destinations :

1. Quelqu'un qui utilise ce code. On se focalise sur l'usage, sur les caractéristiques publiques, sur l'API, etc.
2. Quelqu'un qui maintient ce code. On ajoute des informations concernant des choix de conception, on commente aussi les caractéristiques privées, etc.

P2 Respecter style et conventions

Le *code* respecte des règles de présentation. Par exemple :
— Les indentations sont homogènes.
— La forme des noms respecte le style du langage.
— L'ordre des déclarations est toujours le même.

P3 Écrire du code propre

Le *code* propre (et avec des *commentaire*s) est plus facile à comprendre et donc à *maintenir*.

Il convient de ne pas écrire de *code* qui dépend de l'ordre d'évaluation des paramètres. Il convient d'éviter autant que possible les mécanismes très complexes ou mal maîtrisés dans les langages de programmation.

Catégorie - Modularité

AB:La modularité est un facteur de maintenabilité et d'évolutivité...Mais son importance mérite catégorie. Elle apparait avant, car c'est une préoccupation plus urgente, plus élémentaire.

P4 Séparer les responsabilités dans les modules

Un *module* doit avoir peu de *responsabilité*s car cela induit une meilleure *cohésion*. Moins un *module* a de *responsabilité*s, plus il restera simple et *petit*. De plus, une *responsabilité* doit être assignée à un unique *module*. Chaque *module* doit cacher ses détails de *réalisation* derrière une interface (*encapsulation*).

P5 Avoir des interfaces d'unités simples

Il faut limiter le nombre de paramètres par unité (souvent 4 maximum). Cela peut imposer de définir des valeurs composites pour regrouper des données comme par exemple un nouveau type d'objet. Ainsi, l'interface est plus facile à comprendre, *réutilise*r et *teste*r.

P6 Coupler les composants faiblement

Des changements dans une *architecture* faiblement couplée provoquent moins d'effets *domino* que si le *couplage* est fort. Pour cela, il faut minimiser le *code* exposé (offert en interface). La *maintenance* est alors facilitée car des *composant*s indépendants peuvent être traités de manière isolée.

P7 Écrire de petites unités de code

Limiter la taille des unités de *code* (par exemple à 15 lignes) que l'on produit grâce à la décomposition. La *maintenance* est plus facile car de petites unités de *code* sont plus faciles à comprendre, *test*er et *réutilise*r. Éventuellement réfléchir et généraliser le bout de *code* pour pouvoir l'appliquer dans plusieurs contextes, par exemple en en faisant une fonction avec des paramètres.

P8 Écrire des unités de code simples

Il faut limiter le nombre de branches par unité de *code* (par exemple à 4) [5]. Cela impose de découper et *décomposer* et rend le *code* plus simple à lire, comprendre et modifier.

P9 Écrire un bout de code une seule fois

Il ne faut jamais copier du *code*. Il faut factoriser le *code* au lieu de le recopier. Pour cela, il faut programmer de manière réutilisable, générique et en appelant des fonctions/méthodes existantes. En effet, quand un *code* est copié d'éventuelles *erreur*s doivent être corrigées à plusieurs endroits, ce qui est inefficace et propice aux *erreur*s résiduelles.

P10 Limiter au maximum les variables globales

Une variable globale introduit un *couplage* implicite entre tous les *module*s qui l'utilisent. Donc pour suivre le principe précédent, il vaut mieux ne pas utiliser de variables globales.

P11 Séparer la spécification de sa (ou ses) réalisation(s)

Une *spécification* peut avoir plusieurs réalisations. On permet plus facilement de réutiliser la *spécification* en la séparant de la *réalisation*. Par exemple, on va utiliser des fichiers `.h` en C ou des interfaces en Java pour les *spécification*s et des fichiers `.c` ou des classes en Java pour les *réalisation*s.

La *spécification* et la *réalisation* n'évoluent pas à la même vitesse également, il est donc intéressant qu'elles soit séparées. Ainsi, une *réalisation* peut évoluer sans que sa *spécification* ne change. Cela permet de préparer les évolutions.

Catégorie - Évolutivité, Maintenabilité

P12 Programmer en anglais

Le *code* réutilisé est écrit en anglais, les mots clés sont la plupart du temps en anglais. On évite ainsi les mélanges de langues qui rende la *revue* plus complexe.

P13 Bien choisir les noms

Il faut bien choisir les noms des variables, des méthodes, des classes, des *module*s, etc. [6] Cela améliore la lisibilité et donne du sens aux éléments du *programme*. Ainsi, il est possible de lire en diagonale : les noms explicites permettent de ne pas avoir à se plonger dans la *réalisation* des concepts mal nommés.

5. Limiter la *complexité cyclomatique* à 4.
6. « Mal nommer les choses, c'est ajouter au malheur du monde » – Albert Camus

P14 Ne pas utiliser de « *magic number* »

Il ne faut pas utiliser de littéraux mais plutôt préférer des constantes bien nommées. Cela permet de faciliter la *revue* du *code* (on a le nom plutôt que la valeur). De cette façon, changer une valeur se fait en un seul endroit. Cela permet également de gérer la localisation [7] efficacement.

P15 Utiliser des messages explicites

Les messages d'*erreur*, d'alertes, d'exceptions doivent être explicites et contextualisés.

P16 Avoir une base de code de taille raisonnable

Il convient de ne pas trop faire grossir sa base de *code* qui risque de devenir ingérable. Il faut mettre de l'énergie dans la réduction de la taille du *code*. Cela oblige à factoriser, abstraire/généraliser et à *ré-ingéniérer*. Il est, en effet, plus facile de maintenir un *code* de taille raisonnable.

Catégorie - Sûreté, Fiabilité

P17 Petits pas

Exécuter et tester le code dès que quelques lignes ont été créées ou modifiées. Ne vous lancez pas dans l'écriture de centaines de lignes de code sans tester.

P18 Automatiser au maximum les tests

Si les *test*s sont automatiser, il sera plus facile de tester et donc ce sera fait plus souvent. Pour cela, il faut définir des cas de *test*s à partir de scénarios, d'exemples, de situations limites, de cas d'*erreur*s antérieures, etc.

Il existe des « *framework* » dédiés au *test* (par exemple JUnit) et plus précisément à leur automatisation qu'il convient d'utiliser.

Si la base de *test*s est bien construite et que le *test* est systématique, on évitera les *régression*s et on diminuera les *risque*s.

P19 Ne modifier que du code qui ne passe pas des tests

« *If it ain't broke, don't fix it.* » Si aucun *test* n'échoue, on ne modifie pas le *code*.

C'est une version forte de la programmation dirigée par les *test*s.

Si le *code* doit être modifié, on s'assure que ce que l'on modifie est couvert par des *test*s. Si besoin, il faut commencer par créer des tests. Sinon, comment s'assurer que les modifications sont correctes et la *non-régression* ?

P20 Privilégier la programmation défensive

Il est important de ne pas supposer qu'une donnée sera vérifiée dans un autre *composant* mais plutôt la vérifier au sein du *composant* (typiquement, *test*er le `null`). Voir **contractualisation** C6.

7. La localisation consiste à adapter un logiciel à une langue locale

P21 Éviter les conversions

Éviter les conversions implicites et assurez vous que vos conversions explicites sont saines (ie toujours possible quelque soit la valeur dynamique à convertir).

P22 Attention à null

Éviter l'usage de *null*. Il y a deux situations générales où l'on veut utiliser ce fameux `null` : pour servir de bouchon à une structure récursive ou pour désigner l'échec en retour d'une fonction. Le deuxième cas est souvent traité par des exceptions ce qui est une bonne solution mais pas la seule. En fait, les professionnels déconseillent d'utiliser cette valeur explicitement, elle est implicite pour les objets non initialisés. Une question similaire apparaît avec les fonctions qui doivent rendre un résultat mais parfois il n'y a pas de résultat correct à produire. On peut utiliser le "null object pattern" `https://en.wikipedia.org/wiki/Null_Object_pattern`, ou parfois des mécanismes spécifiques à un langage. Par contre, du fait de la réutilisation de bibliothèques tierces, vous n'êtes pas à l'abri d'une valeur nulle. Des tests `X != null` sont recommandés ; malheureusement ils peuvent alourdir votre code. La même chose est également vraie pour les traitements d'exceptions.

P23 Choisissez votre paradigme

fonctionnel ou impératif, soyez uniforme et ne mélanger pas les deux sans de bonnes raisons. FD:Discutable... pour FD. JCR:JCR en dirai bien plus mais ...

3.2 Bonnes pratiques « objet »

Nous présentons ici des bonnes pratiques qui s'appuient sur le paradigme objet. On parle donc de classes, de méthodes et d'héritage. Certaines sont des exemples d'applications des bonnes pratiques de programmation générales (3.1).

Catégorie - Commentaires

Po1 Écrire du code commenté

Spécialisation de P1. Voir l'annexe A pour des suggestions.

Catégorie - Modularité

Po2 Cohérence par la responsabilité

Réserver à chaque chose (package, classe, variable, fonction, méthode, ...) une responsabilité limitée et bien définie.

Po3 Encapsulation

Application de P5 et P11, il s'agit ici de cacher l'implémentation. Ne pas exposer les attributs d'une classe (encapsulation) et définir une interface.

Po4 Réifier

Ne pas hésiter à définir de nouveaux types car cela accroît la modularité, enrichit les concepts et améliore le contrôle de types.

Po5 Petit préférable

Spécialisation de P7, préférer de petites classes génériques, faciles à écrire, à tester, à valider et à utiliser, plutôt qu'une grosse classe qui ne peut servir que dans un contexte particulier.

Catégorie - Évolutivité, Maintenabilité

Po6 Une classe calcule

Une classe qui ne contient que des méthodes d'accès (`get()`, `set()`) n'est pas une bonne classe. Une classe devrait faire des calculs, assumer une responsabilité. Sinon préférer une structure de données.

Po7 Héritage parcimonieux

Utiliser l'héritage avec précaution :
— Quand la relation est forte et statique entre les classes,
— Quand vous devez redéfinir une méthode,
— Évitez l'héritage ad-hoc, i.e. juste pour un besoin de partage de code.
— Évitez une profondeur abusive d'héritage qui pénalise la compréhension.
— Sinon, pensez à la composition (délégation).

Po8 Déclarer abstrait, instancier concret

Déclarer (*type apparent*) un attribut, une variable, un paramètre le plus abstrait possible ; instancier (*type réel*) le plus concret possible.

Po9 Principe de Demeter

Ne communiquer qu'avec ses accointances (voisins) pour éviter de créer des couplages cachés. On évite donc les instructions contenant des appels en chaîne tels que `a.b().c().d().e()` par exemple.

Po10 Principe Ouvert-Fermé

Programmer pour être étendu par héritage ou référence, pas en modifiant le code. Une classe embarque tout le code qui lui est potentiellement utile. C'est le principe Ouvert-Fermé (OCP, *Open-Closed Principle*) de Bertrand Meyer [9] [8] : « ouvert à l'extension, fermé à la modification »

Catégorie - Sûreté, Fiabilité

Po11 Petits pas

Une déclinaison de la règle P17. Commencer par définir vos attributs, getter, setter *nécessaires* et constructeurs de votre classe. Puis définir et tester vos constructions d'instances et la sortie standard. Ensuite, définir progressivement vos méthodes et testez-les au fur et à mesure.

8. `https://en.wikipedia.org/wiki/Open/closed_principle`

Po12 Pas de fuite d'exceptions

Ne pas propager les exceptions entre différents modules (*cf.* ne pas jeter vos pierres dans le jardin d'à côté ...). Au minimum, une application doit pouvoir attraper toutes les exceptions dans :

- Le `main`, sinon, votre programme plantera lamentablement...
- Les *callbacks* depuis une bibliothèque que vous ne maitrisez pas (le mécanisme de gestion des exceptions peut en effet être conçu différemment)
- Les différents *threads* de votre programme
- Les interfaces d'une classe (préférer sortir des codes d'erreur simple pour le reste du monde)

Po13 Redéfinitions invariantes

Faites des redéfinitions de méthode *invariantes* ie sans changer le nombre ou le type des arguments. Tous les langages les acceptent. Certains langages ont des règles plus souples, mais plus compliquées (covariance, contra-variance, etc.).

3.3 Java

En plus des conseils généraux concernant la conception et programmation orientées objet décrits dans la section 3.2, quelques bonnes pratiques et habitudes spécifiques à Java peuvent être suivies :

Catégorie - Commentaires

Pj1 Utiliser Javadoc

Pour les commentaires, Java propose une syntaxe et un outil spécifique (Javadoc) pour écrire et produire une documentation hypertexte. La Javadoc est essentiellement à destination des utilisateurs d'une classe. Pour la maintenance, les commentaires sont intégrés dans le code.

Pj2 Commentez les paquetages

Les paquetages (voir Pj10) sont commentés avec le fichier `package.info` de leur répertoire.

Pj3 Conventions de nommage

Les conventions de nommage en Java sont les suivantes :

- le style *CamelCase*[9] est utilisé : un nom est composé par concaténation directe de plusieurs mots dont la première lettre est en capitale et les autres en minuscules. La toute première lettre du nom est en minuscule sauf dans le cas d'un type (types primitifs exclus). Exemple : `ceciEstUnNomDeVariableEnCamlCase`.
- on utilise rarement des caractères autres qu'alpha-numériques (`_` par exemple) dans les noms de types, de variables ou de méthodes.
- on différencie le nom d'une interface de celui d'une classe par un motif donné. Par exemple avec le préfixe `I` (`i` capitale).

9. `https://fr.wikipedia.org/wiki/CamelCase`

Pj4 Style des accolades

Les accolades permettent de délimiter les blocs, néanmoins Java n'oblige pas à écrire ces accolades lors qu'il n'y a qu'une seule instruction dans une conditionnelle ou une boucle. Pour éviter des *bugs* faisant suite à de la maintenance, il est préférable d'utiliser systématiquement les accolades, même autour d'une seule instruction sur une seule ligne (*fully-bracketed style*). Par exemple : `if (exp) { instr }`

Catégorie - Modularité

Pj5 Séparer la spécification de sa (ou ses) réalisation(s)

En Java, le point P11 de la section 3.1 sur les bonnes pratiques « génériques » (séparation de la spécification de l'implémentation) se traduit par l'utilisation *d'interfaces* (spécification) implémentées par des *classes*.

Catégorie - Évolutivité, Maintenabilité

Pj6 Éviter les tests d'appartenance à une classe

Éviter l'utilisation du mot-clef `instanceof` (hors d'une fonction `equals`). Il est souvent possible de faire un tel test en définissant de simples méthodes et en les redéfinissant judicieusement dans les classes. Dans d'autres cas définir une méthode auxiliaire peut éviter de recourir à l'utilisation de `instanceof`.

Pj7 Déclarer abstrait, instancier concret

En Java, le point Po8 de la section 3.2 sur les bonnes pratiques « objet » se traduit par l'usage des noms d'interface dans les déclarations de variables, dans les types des paramètres de méthodes, etc. plutôt que leurs implémentations. Par exemple, si possible, déclarer un type `java.util.List` même si on utilise le type `java.util.ArrayList`.

Pj8 Importation

Dans le préambule d'une classe Java, privilégier les imports de classes spécifiques (`import java.util.ArrayList` par exemple) plutôt que les imports de packages complets (`import java.util.*`)

Pj9 Rangez les artéfacts

Ne pas mélanger le produit de la compilation (fichiers `.class`) avec le code source (fichiers `.java`). Les sources sont souvent placées dans un dossier `src` tandis que les fichiers `.class` sont dans `build` ou `bin` (les IDE génèrent normalement par défaut les fichiers `.class` dans un répertoire séparé).

Pj10 Packagez

Structurer le code en utilisant les packages, voire même séparer les interfaces des implémentations dans des packages distincts.

Pour des raisons de sécurité, éviter d'utiliser le package racine (sans nom.)

Catégorie - Sûreté, Fiabilité

Pj11 Petits pas

Spécialisation de P17, définir les méthodes standards comme `toString`, `equals` (et `hashcode`) ; les tester au fur et à mesure.

Pj12 Les exceptions gèrent des cas exceptionnels/anormaux

Les exceptions doivent permettre de gérer les cas. . .exceptionnels

Pj13 Attention à null

Pour éviter l'usage de null : Utilisez Optional (`http://www.oracle.com/technetwork/articles/java/java8-optional-2175753.html`).

3.4 Python

[Question : Qu'y a-t-il comme bonnes pratiques en Python ? Est-ce que l'on détaille un peu la PEP8 (qui donne un style officiel pour Python), ou une référence suffit ?]

En plus des conseils généraux de programmation, il existe quelques bonnes pratiques et habitudes spécifiques à Python. Certaines bonnes pratiques sont d'ailleurs proposées dans des « propositions d'amélioration de Python » (*PEP*, *Python Enhancement Proposal*, notamment dans les PEP 8[10] et 20[11]. Le contenu de cette dernière peut être affiché directement dans l'interpréteur interactif en tapant `import this`.

- Utiliser doctest[12] : cet outil est inclus dans Python, il permet de documenter et de tester un logiciel.
- [Ici, on peut éventuellement reprendre les grandes lignes de la PEP8 : présence/absence du caractère espace dans les expressions et les déclarations ; nombre d'instructions par ligne (une pour une) ; ordonnancement des imports ; constantes ; etc.]
- Concernant les directives d'importation (mot-clef `import`)
  - il convient d'en écrire plusieurs (donc sur plusieurs lignes) plutôt qu'une seule (sur une ligne, en séparant les modules importés par une virgule)
  - il convient de les organiser comme suit :
    1. imports de la bibliothèque standard
    2. imports tiers liés
    3. imports spécifiques de bibliothèques ou d'applications locales
- [TODO]

3.5 Arduino

[TODO: Complétez-moi !]

3.6 Android

[TODO: Complétez-moi !]

10. PEP 8 – Style Guide for Python Code : `https://www.python.org/dev/peps/pep-0008/`
11. PEP 20 – The Zen of Python : `https://www.python.org/dev/peps/pep-0020/`
12. `https://docs.python.org/3.5/library/doctest.html`

3.7 C

[TODO: Complétez-moi !]

3.8 C++

[TODO: Complétez-moi !]
[Question : Est-ce qu'il y a des choses que l'on peut factoriser avec C ? (section précédente vide pour le moment)]

Pj1 Abuser des fonctionnalités offertes par le compilateur

Utiliser au maximum les capacités du compilateur en termes de vérification de l'adéquation entre votre code et vos intentions de programmation = fournir au compilateur un maximum d'informations indiquant vos intentions :

- `const` est votre ami ! A utiliser dès que possible (pour les "variables", les paramètres d'appel des fonctions/méthodes et les méthodes de classe) : spécification si les données manipulées sont modifiables ou non, si les méthodes modifient le contenu de l'objet manipulé, vérification de la cohérence des appels par le compilateur & optimisation des performances de manipulation des données,
- Eviter l'utilisation des macro, si elles sont utilisées pour définir des fonctions génériques que le compilateur ne pourra vérifier. Penser alors éventuellement à l'utilisation des templates.
- Eviter l'utilisation des macro, si elles sont utilisées pour définir des constantes dont vous ne pourrez spécifier le type et donc que le compilateur ne pourra vérifier l'utilisation. Penser alors éventuellement à l'utilisation de variables statiques constantes.
- Ne pas utiliser NULL pour des pointeurs ne pointant sur rien mais plutôt nullptr qui possède un type spécifique que le compilateur pourra exploiter en termes de vérification de type (vs. entier nul) et de surcharge de fonctions, par exemple.
- Favoriser l'utilisation de la surcharge d'opérateur pour éviter les conversions implicites. Pensez aussi à utiliser le mot-clef `explicit` pour la déclaration de certains constructeurs notament à un seul paramètre.
- Utiliser les mots-clés introduits par C++11 pour indiquer au compilateur explicitement vos intentions en termes de constructions d'objet (création, copie, appropriation) : la présence/absence des différents constructeurs peut être explicitée pour produire explicitement des mécanismes de création d'objets (classes non copiables par exemple, ...) : `= delete`.
- Utiliser les mots-clés introduits par C++11 pour indiquer au compilateur explicitement vos intentions en termes de redéfinition des méthodes virtuelles : le mot-clef `override` stipule que la méthode de la classe fille doit spécialiser une méthode existante de la classe mère (ce qui permet au compilateur de vérifier la cohérence des signatures des méthodes existantes), le mot-clef `final` indique explicitement que vous ne souhaitez pas que cette méthode virtuelle soit redéfinie dans les classes-fille, ...Ces mots-clefs restent facultatifs mais permettent d'une part au compilateur de vérifier l'adéquation entre vos intentions et votre code mais aussi d'indiquer à un potentiel utilisateur vos intentions dans le codage de vos classes.
- Ne pas utiliser `reinterpret_cast` de manière abusive, qui suppose que le programmeur sait mieux que le compilateur ce qu'il a à faire (cela peut être vrai dans certains cas, comme la lecture de données extérieures selon un format particulier)...

— Eviter le `static_cast` sur les pointeurs, lorsque le type de l'objet sous-jacent est perdu (et que l'on n'utilise pas le polymorphisme), toujours préférer un `dynamic_cast` qui permet de savoir si le typage est correct (c'est-à-dire si l'objet pointé est du type demandé).
— Eviter d'utiliser le *casting* du C, utiliser les opérateurs de typage dédiés `static_cast` ou `dynamic_cast`. Le programmeur spécifie explicitement alors la nature du transtypage ce qui aide et le compilateur et celui qui lit le code en explicitant les intentions du programmeur.

Pj2 La STL est toujours votre amie

Penser à utiliser au maximum les fonctions / algorithmes / conteneurs de la STL (Standard Template Library) : éviter de réinventer la roue sauf cas très spécifiques. La STL ne se limite pas aux conteneurs mais propose de nombreuses fonctionnalités très intéressantes.

— Dans la `STL` utiliser de préférence `vector` en cas d'accès indexé aux éléments sinon utiliser le conteneur le plus approprié en fonction de la façon d'accéder aux éléments stockés et aux performances (complexité) souhaitées. La STL garantit pour chaque type de conteneur une complexité de temps d'accès (en lecture et en écriture) selon les opérations effectuées.
— Préférer les conteneurs `std::vector` et `std::string` de la STL, pour des tableaux dynamiques, et `array` pour des tableaux dont la taille est connue à la compilation, plutôt que les tableaux standards du langage. L'avantage de ces conteneurs est la présence d'itérateurs qui permettent un mode de parcours unifié du contenu de ces conteneurs. Ces classes de la STL sont également compatibles avec des échanges de données hors C++.
— Utiliser de préférence des *fonctor* (ou objet fonction) plutôt que des fonctions dans les arguments des algorithmes et des comparateurs. Ils sont en effet plus génériques et adaptables car en tant qu'objet, ils possèdent un ensemble de données persistant entre leurs appels.
— Privilégier l'utilisation des lambdas en lieu des fonctors en cas de besoin d'un objet fonction à utilisation unique (utilisation locale) ou dont le code est très court. L'avantage de ces lambdas est qu'elles sont écrites où elles sont déclarées d'où une compréhension plus immédiate de leur rôle.
— Ne pas utiliser de polymorphisme dans un tableau (ou dans un conteneur de la STL) qui stocke directement des objets = phénomène de slicing. En effet, tous les éléments d'un tableau possèdent le même type et sont donc tronqués aux données de ce type. Si vous devez utiliser un tableau ou un conteneur de la STL comme `std::vector`, rassemblant des objets issues de classes A et B (B héritant de A), stocker non pas les objets mais des références, des pointeurs voire des smart pointeurs sur le type A. Le polymorphisme peut alors fonctionner sur ces références ; en revanche, la durée de vie des objets n'est plus gérée par le conteneur et reste à la charge du programmeur. Certains smart pointeurs permettent de gérer de manière automatique la durée de vie de ces objets en fonction de la politique choisie par le programmeur (propriété unique par `unique_ptr`, propriété partagée par `shared_ptr`).
— Les problématiques de gestion mémoire et de propriété exclusive ou partagé des objets alloués dynamiquement, doivent être gérées par des smart pointeurs adaptés (`unique_ptr`, `shared_ptr`).

Pj3 Misc.

Autres conseils :

— Rassembler les méthodes d'allocation (`new`) et de libération mémoire (`delete`) ensemble (dans une classe, un espace de nom ou un module). Penser à une approche orientée

`factory, ...` de la famille des design patterns.
— Les destructeurs *doivent* se terminer normalement. Jamais d'exception.
— Ne pas placer de `namespace using ...` avant un `#include`.
— Préférer l'utilisation des exceptions pour lever des erreurs d'exécution. Mais utiliser des *status* d'erreur (`return`, `errno`) pour des conditions d'utilisation qui ne sont pas des erreurs...
— Lancer des exceptions par valeur (et non par pointeur) et les attraper comme référence (éventuellement constante). [Question : propagation en préservant le polymorphisme ? `throw;` plutôt que `throw e;` ?]

4 Mauvaises odeurs

Dans le contexte du développement logiciel, les *code smells*, ou *mauvaises odeurs*, sont des indicateurs de potentiels problèmes dans la conception et le développement d'un logiciel. Cependant, elles n'indiquent pas systématiquement un problème : si une *mauvaise odeur* se détecte généralement rapidement, évaluer le fait qu'elle indique effectivement un problème demande d'inspecter plus profondément le code.

Plusieurs auteurs ont travaillé sur le sujet et sur la classification des *bad smells*. Dans [11] notamment, les auteurs proposent une taxonomie qui reprend les travaux précédents [18, 2, 1]. La structure de cette section suit cette taxonomie.

Quelques références en lignes, un peu plus accessibles que des articles de recherche :
— `https://sourcemaking.com/refactoring` : reprend cette taxonomie, chaque odeur est expliquée succinctement ;
— `https://martinfowler.com/bliki/CodeSmell.html` et `https://www.refactoring.com/` : sites de Martin Fowler autour du *refactoring* et de manière plus générale de l'ingénierie du logiciel.

Il est à noter que cette taxonomie se place essentiellement dans le cadre du développement objet.

JCB:on pourrait aussi imaginer une classification par portée de mauvaise odeur (logiciel complet/classe/méthode) ; est-ce qu'il vaut mieux une section dédiée aux mauvaises odeurs ou alors référencer les mauvaises odeurs un peu partout ? JCR:perso cela me semble bien de les regrouper, mettre des ref dans les bonnes pratiques peut avoir du sens

4.1 *The Bloaters*

Code qui a tellement grossi qu'il ne peut plus être maintenu correctement (lisible, structuré, évolutif), ex : logiciels avec un gros historique. En général, les *bloaters* apparaissent petit à petit (au fur et à mesure de l'évolution et de la croissance du logiciel, on parle de dégénérescence logicielle), ils sont rarement initialement présents.

JCB:Écho à 3.1

— *Long Method* : de longues méthodes, ex : `doAll()`. On peut se fixer des règles type « une méthode ne dépasse pas un "écran" » par exemple ⇒ utiliser un logiciel métrologie et le configurer en conséquence
— *Large Class* : des classes gigantesques, Blob, *god class* ⇒ utiliser un logiciel métrologie et le configurer en conséquence
— *Primitive Obsession* : cas où il n'y a pas de petites classes pour de petites entités ⇒ la fonctionnalité est ajoutée dans une autre classe, ce qui augmente la taille des classes et méthodes. Cause plus des bloaters que ça n'est un bloater JCR:me semble louche cela

— *Long Parameter List* : des méthodes avec beaucoup de paramètres, des classes avec beaucoup d'attributs. Souvent, ces paramètres ont été ajoutés au fur et à mesure, ⇒ d'où l'importance de faire ses propres revues de code
— *DataClumps* : lors que l'on voit des paramètres qui se retrouvent toujours ensemble (ex : 3 entiers pour le code couleurs RGB), cela augmente mécaniquement la taille du code où ils apparaissent. ⇒ peut-être un problème d'encapsulation de ces données et services et leur restructuration doit être envisagée

4.2 *The Object-Orientation Abusers*

Ces mauvaises odeurs sont plutôt liées à une mauvaise utilisation / non utilisation des concepts objets dans le cadre d'un logiciel développé en objet. JCB:Écho à 3.2

— *Switch Statements* : utilisation d'un switch en Objet alors que normalement on devrait utiliser des hiérarchies de classes et jouer sur les types des instances manipulées
— *Temporary Field* : cas d'une variable d'instance alors que cette variable pourrait n'être qu'une variable dans une méthode JCR:comment les caractériser plus précisément ?
— *Refused Bequest* : redéfinition d'une méthode d'une classe de base de telle sorte que le contrat de la classe de base n'est pas respecté par les sous-classes ; il faire assurer le principe de substitution de Liskov (une propriété pour les objets de type A, doit être vraie vraie pour tout objet de type B, avec B< :A). ⇒ problème dans la conception de la hiérarchie de classes et des redéfinitions
— *Alternative Classes with Different Interfaces* : cas de classes liées qui n'ont pas une interface commune ⇒ soit on les sépare soit on restructure pour faire apparaître ce qui doit être commun

4.3 *The Change Preventers*

Tout ce qui gène les futures évolutions de code.

— *Divergent Change* : une seule classe qui doit être modifiée de partout, ⇒ il s'agit d'un couplage trop fort
— *Shotgun Surgery* : cas où il faut effectuer des modifications sur plusieurs classes lorsque l'on effectue une seule modification dans le logiciel ⇒ il s'agit d'un couplage trop fort
— *Parallel Inheritance Hierarchies* : hiérarchies de classes parallèles, si on modifie l'une, il faut modifier l'autre (pourrait être dans OO-abusers ou dans Dispensables) ⇒ il s'agit d'un couplage trop fort

4.4 *The Dispensables*

Cette odeur représente ce qui n'est pas nécessaire et qui devrait être supprimé du code source. JCB:Écho à la section C1.

— *Lazy class* : classe qui ne fait pas grand chose / pas assez ⇒ la supprimer ou augmenter sa responsabilité
— *Data class* : classe "conteneurs" (avec uniquement des champs et des accesseurs), des classes qui contiennent uniquement des données, sans comportement propre ⇒ devraient être des types individuels ou alors il manque des méthodes dans la classe
— *Duplicate Code* : code dupliqué, utiliser un outil peut s'avérer très pratique ici ⇒ on partage le code en créant une (ou des) nouvelle méthode ou classe
— *Dead Code* : code mort = non appelé, pas forcément facile à voir d'où l'utilisation d'un outil ⇒ code inutile à supprimer ou alors à reconsidérer

— *Speculative Generality* : « just in case code », cas des classes/méthodes/attributs non utilisés mais prévus pour des fonctionnalités futures éventuelles ⇒ on oublie

4.5 *The Couplers*

JCB:Écho à la section C8 (conception) et aux points 10 et 11 de la section 3.1 (pratiques génériques)

— *Feature Envy* : cas de méthodes qui effectuent beaucoup d'appels extérieurs (des get par exemple), problème de couplage ⇒ il faut réduire le couplage en localisant où partageant ces appels
— *Inappropriate Intimacy* : cas de deux classes trop fortement couplées ⇒ il faut réduire le couplage
— *Message Chains* : chaîne de message pour accéder à une donnée, c'est typiquement ce qui se passe lors que l'on enchaîne les get() et viole la loi de Demeter (ex : `myA.getB().getC().getD().getThe` JCR:problème et solution ?
— *Middle Man* : trop (JCB:c'est quand « trop » ?) de délégations simples plutôt que de vraiment contribuer/ajouter un service au logiciel JCR:pas clair

5 Tester, tester, et tester encore

Ça me semble une idée que de placer aussi des bonnes pratiques de test... [plutôt d'accord, mais je vais plutôt privilégier le reste (ce qui existe déjà dans ce document) d'abord avant de me plonger dans cette section (JC)]

Sources :

— `https://sites.google.com/site/markussprunck/blog-1/unit-testing-best-practices`
— `http://www.developer.com/mgmt/top-five-best-practices-for-writing-unit-test-scripts.html`
— `https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters`

6 Gestion partagée avec GIT

AB:Une première version à partir de [16], complété par [13].

G1 Commit cohérent

Un *commit* englobe un ensemble modifications reliées. Par exemple, corriger deux erreurs différentes doit conduire à deux *commits* différents.

De petits *commits* facilitent la compréhension par les autres membres de l'équipe et le retour en arrière (« *rollback* ») si un problème apparait.

Git permet une gestion fine des *commits* grâce au *staging*.

G2 Commits fréquents

Faire souvent des *commit* permet d'assurer de petits *commits* cohérents (voir G1). Cela permet aussi de partager plus souvent avec les autres, et donc de simplifier l'intégration et de réduire le nombre de conflits.

G3 Commit complet

On ne fait un *commit* que lorsqu'une action est terminée (pas du travail à moitié fait). Pour ce faire, il faut séparer des actions en de nombreuses petites modifications cohérentes pour faire de fréquents (G1) *commits* cohérents (G2).

G4 Tester avant un commit

Résister à la tentation de faire un *commit* sur un code non testé.

G5 Écrire de bons messages de commit

Commencer le message par un résumé d'une ligne (50 caractères). Sauter une ligne puis décrire le message détaillé. Le détail contient la réponse aux questions suivantes : Quelle est la raison du changement et quelle est la différence avec la version précédente.

AB:Il y a des travaux intéressants sur des langages contrôles/dédiés pour cela.

G6 Faites des branches

La gestion des branches est l'un des points forts de Git. Il faut l'utiliser intensivement. Elles permettent de séparer chaque ligne de développement : pour une nouvelle propriété, pour une correction d'erreur, pour tester une idée, ...

G7 S'accorder sur une stratégie

Git permet d'appliquer de nombreuses stratégies d'utilisation : des branches de long terme, des branches thématiques, de la fusion (« *merge* ») ou du « *rebase* », ...

Le choix se fait en fonction de la nature du projet, des processus de développement et de déploiement utilisés et de vous et de vos co-développeurs. Pour assurer une cohérence globale, il est important que tout le monde adopte une stratégie identique.

G8 Ne pas ré-écrire une histoire publiée

Une fois que vous avez poussé[13] (« *push* ») vos modifications dans le dépot commun de référence, ou rendu vos *commits* et *tags* publics, vous devez les considérer comme gravés dans le marbre.

Si vous vous apercevez que vous avez fait une bêtise, faites de nouveaux *commits*, revenez en arrière (« *revert* »), mais évitez de cacher cela en ré-écrivant l'histoire.

G9 Maintenance périodique

Nettoyer le dépot aussi souvent que possible.

Vérifier le « *stash* » pour voir les travaux en cours oubliés (`git stash list`)

7 Outils

Les outils sont significatifs de la maturité d'une ingénierie, ils augmentent la productivité, la qualité, la traçabilité et la normalisation des productions. Il est fondamental qu'un ingénieur sache utiliser les outils de base (genre compilateur, éditeur, ...) mais aussi apprenne à installer

13. ou en théorie si quelqu'un tire de votre dépot, mais les gens qui tirent d'un dépot de travail méritent souvent ce qui leur arrive.

et à évaluer de nouveaux outils. Un ingénieur informaticien doit être conscient que l'informatique évolue et que les outils aussi et donc de la veille technologique lui sera nécessaire. Il doit aussi comprendre qu'utiliser un outil demande un minimum de maîtrise et que celle-ci vient en pratiquant (un mooc peut aider mais ne peut remplacer).

Les outils supports pour le génie logiciel et la qualité sont nombreux, ils sont de disponibilité et de maturité variable. Il ne s'agit donc pas d'être exhaustif mais de donner une classification sommaire et des exemples intéressants à connaître. La classification JCR:mettre une ref sur le wiki est valable pour Java bien que certains (IDE notamment) sont utilisables pour d'autres langages, notamment C++. Cette liste présente principalement des outils libres, disponible sous licence GNU ou autres, mais aussi quelques outils commerciaux avec une licence gratuite de préférence sans contrainte de temps. Pour beaucoup, ils font partie de la vie courante du développeur ou de l'ingénieur logiciel, par exemple Eclipse, Checkstyle, PMD, Junit, etc. Mais en pratique il y a de très nombreux et bons outils commerciaux. Toutefois dans certains domaines les outils présentés ici sont de bons prototypes montrant des avancées qui n'ont pas encore été intégré dans des produits commerciaux.

A Commentez !

On invoque toujours la bonne pratique : documentez (O2, C2) ou commentez (3.1). Mais qu'est-ce qu'un bon commentaire ? Que peut-on bien décrire ? Voici ici quelques idées de ce qu'un relecteur peut s'attendre à trouver dans un bon commentaire.

Généralités

Les commentaires sont là pour aider les personnes qui vont devoir lire le programme à le comprendre [15] :

- Ils doivent être *informatifs*. Ils ne doivent pas simplement paraphraser le programme mais fournir de l'information supplémentaire. Par exemple, le commentaire « *An abstract class implementing a visitor for formula nodes* » pour la classe *AbstractFormulaNodeVisitor* est inutile et n'apporte aucune information. Cela peut même provoquer des incohérences lors de *refactorings*, ces derniers ne s'appliquant pas sur les commentaires.
- Les commentaires doivent être *de plus haut niveau* que le programme : expliciter l'intention du programmeur, la place du programme dans l'architecture générale de l'application, les raisons des choix de programmation...
- *Plus n'est pas toujours mieux*. Des commentaires trop verbeux deviennent encombrants et finalement nuisent à la compréhension du programme. Ils doivent faciliter la compréhension du programme courant et ne font donc pas apparaître l'histoire des versions successives, ni ce qui n'est pas encore prévu.
- Commentaires et programmes doivent être cohérents afin de ne pas créer de confusion. Il faut notamment être vigilant à l'occasion de modifications et ne pas oublier de modifier les commentaires en conséquence.

En pratique, on peut utiliser des annotations (ou des commentaires spécifiques) pour signaler des particularités comme (avec la syntaxe Java) :

- `\\TODO` : pour signaler un complément de code à faire. Par exemple une méthode qui doit redéfinir une méthode abstraite.
- `\\FIXME` : pour signaler une erreur à corriger. Il est mieux de le faire que de placer un commentaire. Mais il vaut mieux mettre le commentaire que de ne rien faire.
- `\\XXX` : pour signaler une erreur, mais qui n'en déclenche pas, ou quelque chose qui demande une attention eXXXtraordinaire.

— certains utilisent CHECKME, DOCME, TESTME, PENDING

Des IDE comme `eclipse` peuvent exploiter cela et gérer des listes de ces étiquettes (*tags*) pour simplifier le suivi.

Package

Un *package* (ou paquetage) est, en général, un répertoire du système de fichier et ne dispose donc pas d'une description propre. Le meilleur moyen d'y associer un commentaire est d'ajouter dans son contenu un fichier texte. Il est d'usage d'appeler ce fichier README (ou README.txt).

Le fichier README peut contenir différentes informations selon la nature du paquetage. Au minimum, il contient :

— une description textuelle du paquetage qui évoque le but de ce paquetage ;

— une liste de son contenu (fichiers, autres répertoires/*packages*) et une description courte.

Pour Java en principe on doit définir un fichier `package-info.java` qui sert à cela.

Classe

Pour une classe (ou une bibliothèque dans le cas de programmation non objet), on peut décrire :

— son rôle et ses responsabilités : pourquoi cette classe est utile et a été réifiée ;

— les services offerts : les méthodes publiques importantes ;

— l'information gérée : les attributs importants ;

— tout ce qui mérite d'être dit :

  — des invariants de classe,

  — les services pensés pour être redéfinis (C9),

  — des hypothèses de conception (être clair),

  — des choix de conception spécifiques (rôle dans des patrons de conception) ;

— des exemples d'utilisation :

  — des constructeurs,

  — des services.

En suivant les principes faire simple, faire petit (peu de responsabilités, peu de services publics) la taille du commentaire devrait rester raisonnable.

Code Commenting Styles

Code : int x = (really complicated expression) ;

The Stater of the Obvious // Computes x.	**The More Precise Stater of the Obvious** // Computes the value of x.	**The Jedi** // This is not the line you are looking for.

The Wizard
// This is magic. Don't edit
// unless you are Gandalf.
	The Edisson
	// I didn't fail, I just found 10,000 wrong
	// explanations for how this line works.
		The Kubrick
		//This line was given to us by a
		//monolith from outer space.

The Naive Planner
// To be replaced with new
// implementation in 2018.
	The Naive Coder
	// This logic needs to be modified,
	// so we don't need to document it so well.
		The Naive Interpreter of Good Practices
		// Consider this line to be a black box.
		// Encapsulation is a good practice, right ?

The Forward Thinker
// Computes the value of x
// for the next step.
	The Advisor
	// I wouldn't try to understand, much
	// less modify, this line if I were you.
		The Terminator
		// Hasta la refactoring, baby...

The Carelessly Vague
// Does stuff with the input.
	The Eloquently Vague
	// Performs the required computations
	// and produces the proper result.
		The Fermat
		// I'd document this but it wouldn't fit
		// in 80 columns.

The Carpe Diem
// Seems to work for now,
// let's not worry until it
// breaks.
	The Zen
	// To find happiness we must be open
	// to new beliefs. I choose to believe this
	// line works.
		The Machine
		// Humans cause bugs. So this line was
		// designed to maximize machine
		// performance and minimize human
		// understanding.

The Lazy // see docs	**The Honest Lazy** // see docs (TODO : write docs)	**The Liar** // To be documented later

The Museum Curator
// Preserved intact from old
// system.
	The Chooser
	// We had to choose between meeting the
	// deadline or writing organized code.
	// (edit : we failed both)
		The Zork Adventurer
		// Don't look at this line too long.
		// You are likely to be eaten by a grue

The Purely Empirical
// This line produces correct
// results for all tests, so it is,
// by definition, correct.
	The Time Traveler
	// Future me : please forgive me for
	// this code. After you forgive me, can you
	// also please fix the code ?
		The Knight Guardian
		// You had a choice : looking at this code,
		// or living in blissful ignorance.
		// You chose poorly.

TABLE 1 – Différents styles de commentaires.

Fonction, méthode

Pour une méthode (ou une fonction), on peut décrire :
— son but : pourquoi cette fonction est utile ;
— ses arguments : rôle et domaine de définition ;
— son contrat (C6)
  — précondition : une assertion en entrée de méthode qui protège la fonction et décrit ce que l'appelant (client) devrait respecter,
  — postcondition : une assertion en sortie de méthode qui décrit le changement ou le calcul opéré par la méthode ;
— si cette méthode est faite pour être redéfinie, et éventuellement des variantes connues ou possibles (C9) ;
— les exceptions éventuelles ;
— des exemples d'utilisation.

Le contrat (voir C6) peut rester informel ou être inclus dans le programme. C'est un bon outil de test et de mise au point qui permet de localiser efficacement les erreurs.

Attribut, variable

Pour un attribut (ou une variable), on peut décrire :
— son rôle : pourquoi il est utile ;
— son domaine de définition (les valeurs qu'il accepte, ses bornes), si le type ne suffit pas à le décrire ;
— s'il peut être `null` ou pas ;
— les méthodes qui le modifient ;
— les contraintes qui le lient à d'autres.

B Code d'éthique

Software Engineering Code of Ethics and Professional Practice [14] (Version 5.2)

as recommended by the IEEE-CS/ACM Joint Task Force on Software Engineering Ethics and Professional Practices Short Version [14, p26-27].

Preamble

The short version of the code summarizes aspirations at a high level of abstraction ; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious ; without the details, the aspirations can become high sounding but empty ; together, the aspirations and the details form a cohesive code. Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles :

1. *Public* — Software engineers shall act consistently with the public interest.

14. © 1999 by the Institute of Electrical and Electronics Engineers, Inc., and the Association for Computing Machinery, Inc.

2. *Client and Employer* — Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. *Product* — Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. *Judgment* — Software engineers shall maintain integrity and independence in their professional judgment.
5. *Management* — Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. *Profession* — Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. *Colleagues* — Software engineers shall be fair to and supportive of their colleagues.
8. *Self* — Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Références

[1] Kent Beck, John Brant, Martin Fowler, William Opdyke, and Don Roberts. *Refactoring : improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[2] Kent Beck, Martin Fowler, and Grandma Beck. Bad smells in code. *Refactoring : Improving the design of existing code*, pages 75–88, 1999.

[3] equipe-gl@telecom bretagne.eu. Un glossaire du génie logiciel. Document formation IMT-Atlantique, V1.0 CC-BY-SA, 2016.

[4] Janna Hastings, Kenneth Haug, and Christoph Steinbeck. Ten recommendations for software engineering in research. `https://gigascience.biomedcentral.com/articles/10.1186/2047-217X-3-31`, 2014. visité le 23/3/2016.

[5] Curt Hibbs, Steve Jewett, and Mike Sullivan. *The Art of Lean Software Development : A Practical and Incremental Approach.* O'Reilly Media, Inc., 1st edition, 2009.

[6] Capers Jones. *Software Engineering Best Practices.* Lessons from Successful Projects in the Top Companies. McGraw Hill Professional, November 2009.

[7] David Loo. Software engineering best practices, practical things we can all do. IMS Health, `http://cusec.net/archives/2002/loo.pdf`, CUSEC, 2002. visité le 23/3/2016.

[8] Robert C. Martin. *Clean Code : A Handbook of Agile Software Craftsmanship.* Prentice Hall, 2008.

[9] Bertrand Meyer. *Object-oriented software construction*, volume 2. Prentice hall New York, 1988.

[10] Bertrand Meyer. *Touch of Class : Learning to Program Well with Objects and Contracts.* Springer, 2009.

[11] M. V. Mäntylä. A taxonomy for "bad code smells". `http://mikamantyla.eu/BadCodeSmellsTaxonomy.html`.

[12] M. V. Mäntylä and C. Lassenius. Subjective evaluation of software evolvability using code smells : An empirical study. *Journal of Empirical Software Engineering*, 11(3) :395 – 431, 2006. `http://mikamantyla.eu/ESE_2006.pdf`.

[13] Seth Robertson. Commit often, perfect later, publish once : Git best practices. Web, `https://sethrobertson.github.io/GitBestPractices/`, 2012. visité le 11/4/2018.

[14] Stephen R Schach. *Object-Oriented and Classical Software Engineering.* Mc Graw Hill, Vanderbilt University, 8th edition, 2011.

[15] Allan M. Stavely. *Writing in Software Development.* The New Mexico Tech Press, 2011.

[16] Tower. Version control best practices. Web, `https://www.git-tower.com/learn/git/ebook/en/command-line/appendix/best-practices`, 2018. visité le 11/4/2018.

[17] Joost Visser. *Building Maintainable Software.* O'Reilly, 2015.

[18] W C Wake. Refactoring workbook. *Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA*, 1 :8, 2003.