Guide de bonnes pratiques en génie logiciel

Modélisation et programmation objet

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 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 ?

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*).

O5 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).

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

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.

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 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.

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

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 *test*er.

P6 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.

P7 Écrire des unités de code simples

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

P8 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.

P9 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.

P10 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.

4. Limiter la *complexité cyclomatique* à 4.

Catégorie - Évolutivité, Maintenabilité

P11 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.

P12 Bien choisir les noms

Il faut bien choisir les noms des variables, des méthodes, des classes, des *module*s, etc.[5] 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.

P13 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[6] efficacement.

Catégorie - Sûreté, Fiabilité

P14 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.

P15 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.

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).

5. « Mal nommer les choses, c'est ajouter au malheur du monde » – Albert Camus
6. La localisation consiste à adapter un logiciel à une langue locale

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 P10, 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 P6, 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.

Catégorie - Sûreté, Fiabilité

Po7 Petits pas

Une déclinaison de la règle P14. 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.

Po8 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)

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 Commentez les paquetages

Les paquetages (voir Pj5) sont commentés avec le fichier `package.info` de leur répertoire.

Pj2 Conventions de nommage

Les conventions de nommage en Java sont les suivantes :
— le style *CamelCase*[7] 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).

Pj3 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

Pj4 Séparer la spécification de sa (ou ses) réalisation(s)

En Java, le point P10 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

Pj5 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.)

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

Catégorie - Sûreté, Fiabilité

Pj6 Petits pas

Spécialisation de P14, définir les méthodes standards comme `toString`, `equals` (et `hashcode`) ; les tester au fur et à mesure.

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 ,
  — 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 ;
— si cette méthode est faite pour être redéfinie, et éventuellement des variantes connues ou possibles ;
— les exceptions éventuelles ;
— des exemples d'utilisation.

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.

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.