Je code : Les bonnes pratiques de développement logiciel

Lila Ammour, Olivier Cappé, Thierry Chaventre, Karin Dassas, Marc Dexet, Patrick Moreau, C. Mouton, Jean-Christophe Souplet

▶ **To cite this version:**

Lila Ammour, Olivier Cappé, Thierry Chaventre, Karin Dassas, Marc Dexet, et al.. Je code : Les bonnes pratiques de développement logiciel. 2019. hal-02083801

HAL Id: hal-02083801
https://hal.science/hal-02083801v1

Submitted on 9 Dec 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - ShareAlike 4.0 International License

JE CODE : LES BONNES PRATIQUES DE DEVELOPPEMENT

Le CNRS souhaite mieux valoriser les logiciels issus des travaux de recherche. Dans cette perspective, un grand plan d'actions a été initié afin de mettre l'accent sur la formation, la sensibilisation et l'utilisation d'outils de développement telle qu'une forge, et dans le même temps, favoriser une meilleure utilisation du logiciel libre comme outil de valorisation.

Dans ce cadre, accompagné par le Réseau DevLOG, nous avons établi 3 plaquettes de bonnes pratiques :

- Quels sont mes droits ? Quelles sont mes obligations ?
- Les bonnes pratiques de développement
- Les bonnes pratiques en matière de diffusion

Il faut bien évidemment garder à l'esprit que les concepteurs et développeurs au sens large (chercheurs, ingénieurs, techniciens) sont le public visé par ces plaquettes.

INTRODUCTION

Il existe plusieurs raisons d'adopter des bonnes pratiques de développement :

- En premier, cela permet de réduire la dette technique du logiciel. Les coûts ultérieurs, tant pour la maintenance corrective (corriger les bugs informatiques) que pour la maintenance évolutive (mise à jour des composants de tiers, ajout de nouvelles fonctionnalités au logiciel) peuvent être plus ou moins importants selon la mise en œuvre de bonnes pratiques de développement. Ces coûts additionnels représentent la dette technique d'un logiciel. Cela permet de maitriser les temps de développement et de réduire la fréquence des bugs.
- Avoir de bonnes pratiques de développement permet d'accroitre la sécurité des logiciels développés.
- Cela permet également d'obtenir un logiciel ayant une bonne qualité juridique au moment où une opération de valorisation est envisagée.
- L'application de bonnes pratiques rend propice le travail en collaboration en établissant un "code de conduite" commun à tous les acteurs.

OBLIGATION DE SECURITE DANS LES DEVELOPPEMENTS INFORMATIQUES

Tout développement informatique **fait peser des risques** sur les données qui sont traitées ou produites et sur les infrastructures qui hébergent ces développements. Lorsqu'un code présente des failles et que celles-ci sont exploitées, la prise de contrôle des serveurs d'hébergement peut constituer **une porte d'entrée aisée dans le SI de l'unité** et **provoquer des dégâts très importants** (fuite de données, divulgation de contenus sensibles, perturbation de l'activité de l'unité, etc…).

CC-BY-SA-4.0

Le CNRS, conscient de ces risques et de la nécessité de les réduire, impose au travers de sa politique de sécurité des SI la prise en compte de la sécurité dans les développements dès la phase de conception. Le document de synthèse [1] résume les points clés qui doivent être pris en compte et appliqués par le développeur pour permettre la réduction des risques décrits supra.

METHODE DE DEVELOPPEMENT

Il est nécessaire en premier lieu d'adopter une méthode de développement.

Le développement en cycle en V est intéressant à connaitre. Le cycle en V est pertinent dans les domaines métiers où il n'y a quasiment pas ou peu d'inconnues en amont ou très vite résolues, ou quand les phases de réalisation sont très couteuses, comme dans l'industrie ou dans le BTP. Le cycle en V en informatique est souvent lourd à mettre en œuvre et pas forcément adapté au contexte de travaux de laboratoire.

D'autres approches ont été créées et misent en œuvre pour pallier ses défauts: méthodes itératives, agiles, etc...

ENVIRONNEMENT DE TRAVAIL

La gestion du logiciel sous une Forge est vivement encouragée. Les outils potentiellement offerts par une Forge sont principalement :

- un système de gestion des versions (par exemple, via Git ou Mercurial) ;
- un gestionnaire de listes de discussion (et/ou de forums) ;
- un outil de suivi des bugs ;
- un gestionnaire de documentation (souvent sur le principe du wiki) ;
- un gestionnaire des tâches.

L'outil de gestion de code permet l'archivage des versions, la traçabilité des contributions. L'organisation des fichiers permet de facilement distinguer le logiciel développé par le laboratoire des composants de tiers. La Forge permet en outre la sauvegarde du code et évite les pertes de données. Elle améliore le travail collaboratif.

Il existe des outils d'intégration continue pour automatiser un certain nombre de tâches et gagner en efficacité. L'automatisation des tests permet d'améliorer la qualité de code. Le même jeu de tests est systématiquement appliqué dès que le code évolue, permettant d'identifier rapidement si ces évolutions ont fait apparaitre des régressions.

[1] https://intranet.cnrs.fr/protection_donnees/Documents/Sensib_d%C3%A9veloppeurs.pdf

CC-BY-SA-4.0

LES DIFFERENTES ETAPES DE DEVELOPPEMENT

Spécifications fonctionnelles (Le QUOI)

La spécification fonctionnelle correspond à l'expression du besoin. Elle doit être précise. Elle doit couvrir l'intégralité des cas d'utilisation du logiciel, en expliquant ce qu'il doit faire et non pas comment il va le faire. Chaque spécification doit pouvoir être testée. Un modèle mathématique ou le fonctionnement d'un instrument sont des exemples de spécifications fonctionnelles.

Les spécifications du logiciel permettent :

- d'identifier les besoins, puis de les traduire en termes de fonctionnalités, d'interfaces avec l'extérieur et entre elles ;
- de préciser les enchaînements d'actions ;
- de préciser les contraintes (performances, priorités, encombrement mémoire) ;
- d'analyser, en fonction des besoins à couvrir, les logiciels, éventuellement de tiers, qui pourraient être réutilisés et d'évaluer les impacts de leur réutilisation sur le développement.

Les spécifications fonctionnelles doivent s'affranchir de la façon dont sera implémenté le logiciel.

Il est important d'établir des spécifications fonctionnelles pour ne pas se lancer aveuglément dans le codage.

Le choix de la licence de distribution du logiciel à développer peut faire partie des spécifications fonctionnelles, contraignant le cas échéant, le choix des composants de tiers (cf. plaquette Je code : Quels sont mes droits ? Quelles sont mes obligations ?).

Conception (Le COMMENT)

Cette étape permet de décrire comment le code est censé fonctionner, avant de rentrer dans la phase de codage.

C'est la réponse technique aux spécifications fonctionnelles.

L'activité de conception consiste à :

- définir le découpage structurel de l'application en constituants et composants logiciels (architecture) puis à détailler chacun d'eux ;
- définir le flux de données et les interfaces entre composants ;
- effectuer les estimations de ressources ;
- dégrossir les interfaces homme / machine.

L'architecture du logiciel doit être conçue en amont de l'écriture du code. Sa conception permet de structurer le logiciel, de séparer ses différentes fonctions et d'obtenir un codage modulaire. Sans codage modulaire, le logiciel serait sous la forme d'un seul fichier de plusieurs milliers de lignes de code. Il serait difficile de faire évoluer un tel logiciel. L'architecture permettra l'identification des composants de tiers, souvent sous licence libre.

La conception de l'architecture peut se faire manuellement ou alors via l'utilisation de méthodologie de modélisation standardisée, comme UML par exemple. Chaque composant défini aura sa propre spécification technique.

CC-BY-SA-4.0

Dans les laboratoires où le développement se fait par des ingénieurs, basé sur des besoins exprimés par des chercheurs non-informaticiens, l'architecture est un outil puissant de dialogue. Ce sera également un outil de dialogue entre les développeurs et les services de valorisation.

Exemple d'outils d'aide à la conception

- Présentation (ex : Libreoffice Impress, ...)
- UML (Modelio, tableau blanc, ...)
- Design patterns
- Prototypage d'IHM (ex : Balsamiq, ...)

Codage (La REALISATION)

C'est la phase de réalisation à proprement parler, pendant laquelle sont développés des composants qui sont ensuite assemblés pour créer le logiciel répondant aux spécifications fonctionnelles.

Il est conseillé de séparer le code en plusieurs fichiers (ceci basé sur la découpe architecturale)

Règles de codage

Il est souvent nécessaire de définir des conventions de codage, de nommage des variables au sein d'une équipe de développement. Il est également conseillé de mettre des commentaires qui permettront une meilleure compréhension du code. Ces commentaires doivent évoluer avec le logiciel. Pour limiter le nombre de commentaires, il est conseillé d'écrire un code très lisible et compréhensible par un lecteur humain.

Agrégation de code

Il est de plus en plus rare qu'un logiciel soit écrit « from scratch ». Ainsi, les bonnes pratiques de développement doivent intégrer le fait que des composants de tiers sont utilisés pour le développement du logiciel. Ces composants de tiers peuvent prendre différentes formes : code source de collègues "récupéré" et qualifié puis intégré dans le logiciel, modifié ou non, ou encore utilisation de bibliothèques existantes éventuellement sous licence libre, ou d'un « framework ».

Il faut bien choisir la licence des composants de tiers au regard de la licence de diffusion choisie du logiciel. Dans tous les cas, il est conseillé de :

- Assurer la traçabilité des composants de tiers en vérifiant que ces composants comportent des entêtes. Le standard SPDX [2] peut être utilisé pour tracer les composants sous licence libre. Il est conseillé de tenir à jour la nomenclature des composants de tiers, encore appelée « BOM » pour Bill of Material,
- Tracer les composants qui n'ont pas d'entête,
- Isoler ces composants de tiers en faisant des architectures modulaires : ceci facilitera l'identification du code extérieur et les analyses juridiques du logiciel pour permettre ainsi d'optimiser les schémas de transfert et se donner éventuellement les moyens de retirer ce composant, si cela était nécessaire.

[2] SPDX (Software Package Data Exchange) est un format de fichier utilisé pour documenter des informations sur les licences de logiciels.

CC-BY-SA-4.0

Il est également conseillé de mettre à jour régulièrement l'architecture du logiciel ; ceci aidera la traçabilité du développement.

Langage de programmation

Il devient alors nécessaire de choisir un langage. Le bon langage, c'est celui que l'on maîtrise. Il faut être conscient de ses particularismes, de ses forces et faiblesses. Les paramètres à prendre en compte pour le choix d'un langage de programmation sont :

- La facilité d'usage,
- La production du code sans avoir à descendre trop bas (allocation, pointeurs) sauf nécessité. Il doit être clair et compréhensible,
- Le langage qui doit permettre d'obtenir des logiciels performants,
- L'écosystème qui doit être riche : notamment autour de l'outillage (compilateurs, éditeurs) et également autour du support (communauté, forum),
- Le langage doit être maintenu (suivi, réactivité, vulnérabilités patchées, …) pour assurer une bonne sécurité,
- Il ne doit pas être exotique en ce sens que le recrutement de développeurs maîtrisant le langage est facile,
- Un langage non-propriétaire est recommandé.

Entête

Un bon entête permet un bon suivi : historique, juridique des composants du logiciel. L'entête doit mentionner le copyright, l'année et éventuellement la licence de diffusion. Il est possible d''utiliser des outils pour vérifier que toutes les informations nécessaires sont présentes dans les entêtes. Il ne faut jamais modifier les entêtes des composants de tiers que l'on intègre.

Exemple d'outils pour la phase de codage :

- Éditeur de code (complétion du code, surlignage, linter[3])
- Environnement de développement intégré (IDE) : Eclipse, Netbeans, Pycharm, Codeblocks, Atom,...
- Compilateur et/ou interpréteur
- Debugger
- Packaging
- Outils de création d'IHM (Interface Homme Machine)
- Outil d'analyse statique: SonarQube, SonarLint

Tests unitaires

Ces tests interviennent à un niveau « atomique ». Chaque composant a été modélisé puis codé durant les étapes précédentes. Les tests unitaires assurent que ces composants respectent de manière

[3] Un linter est un outil d'analyse statique de code source durant l'édition.

CC-BY-SA-4.0

individuelle leur spécification. L'ensemble de ces tests unitaires doivent être capitalisables au sein d'un plan de test qui évolue avec le logiciel.

Le test unitaire doit être exécutable automatiquement.

Exemple d'outils

- JUnit pour Java,
- Unitest pour Python,
- CUnit,
- CppUnit,
- Pytest (cf. https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks)

Tests d'intégration

On s'assure que le logiciel intégrant les composants, au préalable testés unitairement, fonctionne. Ce sont là les premiers tests grandeur nature du logiciel dans sa globalité. Le logiciel fonctionne mais peut ne pas répondre aux besoins. Il faut ensuite passer par la phase de validation.

Exemple d'outils

- Jenkins, Travis, GitLab-CI,...

Tests de validation

Le produit est à ce moment testé en regard de la spécification fonctionnelle. On parle aussi de test de recette. Toutes les utilisations qui y ont été définies doivent pouvoir se vérifier dans les faits. Les tests de validation sont effectués par le prescripteur "métier". C'est par exemple le chercheur si le développement a été effectué par une équipe de développement.

Remarque

Il est recommandé d'effectuer l'ensemble des tests avec des données fictives, notamment et surtout lorsque le logiciel comporte des données personnelles ou des traitements de données à caractère personnel.

Si exceptionnellement cela ne peut pas être le cas, dans l'objectif de protéger la vie privée des personnes concernées, le délégué à la protection des données de l'unité doit être sollicité pour conseil et aide à la réalisation des tests.

DOCUMENTATION

Une documentation sera rédigée pour l'utilisateur final. Une seconde documentation sera établie pour les développeurs. Elle comprendra les commentaires, la description de l'architecture. Il est possible de générer automatiquement une documentation de référence. Ceci a plusieurs intérêts :

- Garantir que la documentation reflète le code au fil du temps,
- Faciliter la rédaction de la documentation technique,
- Faciliter la mise à jour de la documentation lorsque le code évolue.

CC-BY-SA-4.0

De même, le plan de test (tests unitaires, tests d'intégration, tests de validation) devra être rédigé.

Ces documents se maintiennent au fur et à mesure de l'évolution du logiciel.

Exemple d'outils

Les outils pour générer une documentation automatique associée au code sont par exemple :

- Doxygen
- Javadoc
- Sphinx, etc…

DIFFUSION

(Cf. plaquette « Je code : les bonnes pratiques en matière de diffusion »)

CONCLUSION

Différentes méthodologies de développement existent. Il est important de choisir celle la plus adaptée à votre contexte de travail.

Le choix de composants tiers doit se faire en anticipant les besoins futurs de diffusion (cf. plaquette « Je code : Les bonnes pratiques en matière de diffusion ») et dans le respect des clauses de propriété intellectuelle (cf. plaquette « Je code : Quels sont mes droits ? Quelles sont mes obligations ? »). De même, ces aspects de diffusion et de propriété intellectuelle peuvent avoir un impact sur l'architecture logicielle à mettre en place.

Pour aller plus loin :

https://gns.cnes.fr/fr/qualite-logiciel-ap-7
http://en.tldp.org/HOWTO/Software-Release-Practice-HOWTO/
https://www.projet-plume.org/files/ENVOL2010-BonnesPratiques_Dev_VLVB.pdf
https://en.wikipedia.org/wiki/Systems_development_life_cycle
https://disic.github.io/politique-de-contribution-open-source/pratique/
https://www.cac.cornell.edu/education/training/StampedeJan2015/BestPracticesForSoftwareDevelopment.pdf
https://www.cnil.fr/fr/kit-developpeur

Pour toute question :

gt_sw_ingenierie@services.cnrs.fr

Auteurs :

Lila AMMOUR, Direction des relations avec les entreprises du CNRS
Olivier CAPPE, Directeur adjoint scientifique Projets scientifiques transverses, INS2I, CNRS
Thierry CHAVENTRE, Réseau DevLOG, Ingénieur en Ingénierie Logicielle, LPC Caen, CNRS/IN2P3
Karin DASSAS, Réseau DevLOG, Informaticienne projet, Institut d'Astrophysique Spatiale, CNRS/INSU
Marc DEXET, Réseau DevLOG, Ingénieur en Ingénierie Logicielle, Institut d'Astrophysique Spatiale, CNRS/INSU
Patrick MOREAU, Direction des relations avec les entreprises du CNRS
Claire MOUTON, Réseau DevLOG, Ingénieur de Recherche Développement, Déploiement d'Applications et Calcul Scientifique CREATIS, Université de Lyon
Jean-Christophe SOUPLET, Réseau DevLOG, Directeur-adjoint en charge du secteur Informatique à Open Edition, USR2004

CC-BY-SA-4.0