PROGRAMMEZ.COM

# PROGRAMMEZ!

Mensuel n°190 - Novembre 2015

le magazine du développeur

# App Store : qui veut gagner des millions ?

## Faut-il passer à Drupal 8 ?

## Office 365

*Les développeurs à la fête !*


## Mouvement Craftsmanship

*Pour un code de qualité et bien documenté !*

Mission : MAKER

**ESP8266** : puissant, petit, WiFi inclus pour 2€

**Test de la carte MicroView**

Printed in EU - Imprimé en UE - BELGIQUE 6,45 € SUISSE 12 FS - LUXEMBOURG 6,45 € - DOM Surf 6,90 € Canada 8,95 $ CAN - TOM 940 XPF - MAROC 50 DH

M 04319 - 190 - F: 5,95 € - RD

# Le mouvement Craftsmanship : un moyen de garantir un code de qualité et bien documenté.

*Il est fréquent de constater que, quelle que soit la méthode choisie, la durée de vie des projets informatiques dépasse rarement cinq ans : code non maintenable, dette technique incontrôlable, connaissance perdue au gré des départs des "développeurs clés". Cet amer constat a poussé les premiers concernés, à savoir les équipes de développement, à agir. C'est ainsi qu'est né le mouvement Software Craftsmanship.*

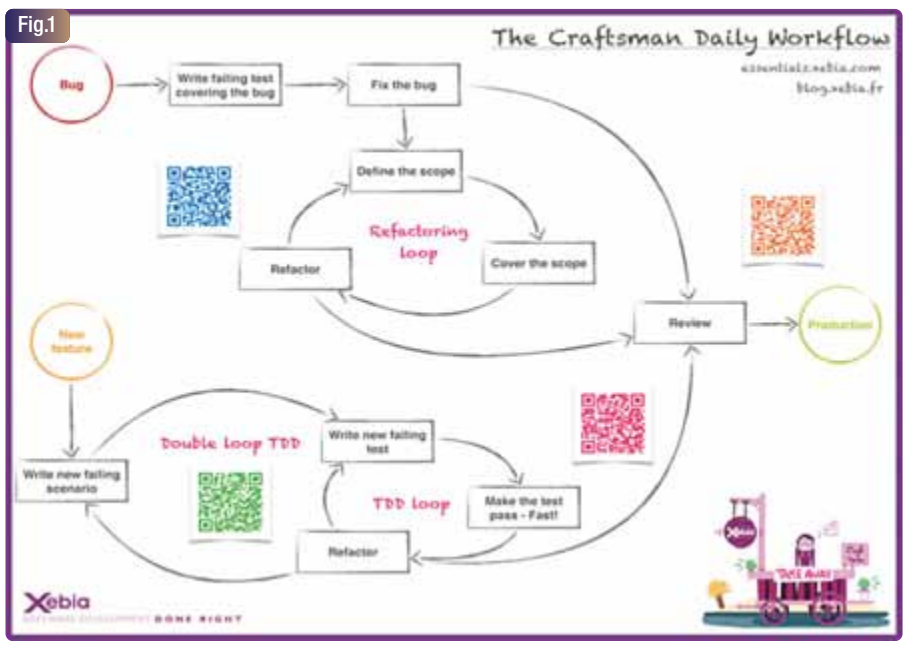
David Caramelo,

Marie-Laure Thuret

et Fatiha Bouad

*Consultants* **Xebia**

Fig.1




Nous vous proposons de découvrir certaines pratiques issues du Craftsmanship qui vous permettront de documenter efficacement une application et vous garantiront une base de code saine et de qualité. Enfin, vous verrez comment transformer une équipe de développeurs en équipe de craftsmen accomplis.

## Documenter ou ne pas documenter ?

Traditionnellement, la réponse à cette question consiste à rédiger de la documentation. Elle se retrouve alors disséminée dans deux endroits :

- Dans les spécifications métiers afin qu'un développeur sache ce qu'il doit développer.
- Dans le code, à travers des commentaires supposés afin d'aider la compréhension.

### Les problèmes de la documentation classique

Supposons qu'un développeur reçoive une spécification métier incomplète, puis un complément d'informations oral, ou bien qu'un bloc de code contenant un commentaire soit copié-collé puis son comportement modifié. Que se passe-t-il si la documentation n'est pas mise à jour au même moment ? Elle devient obsolète et la plupart du temps, cela se produit dès lors qu'on la considère comme achevée. On peut se dire qu'il suffit de faire attention afin de la garder synchronisée avec l'état actuel de l'application, mais il serait impossible de le garantir. Une documentation dans laquelle on ne peut pas avoir confiance et qui peut potentiellement devenir trompeuse, n'est pas une solution envisageable. Cependant, tout n'est pas perdu. Une application dispose en effet d'un élément qui reflète à n'importe quel moment la réalité : son code source. Après tout, pourquoi ne pas s'en servir directement à des fins documentaires ? Lire du code est une tâche difficile. Même s'il est évidemment possible pour un développeur de comprendre les fonctionnalités métiers d'une application en lisant son code, celui-ci contient de nombreuses abstractions qui peuvent rendre ce processus long et compliqué. On va plutôt chercher à s'appuyer sur des tests automatisés, qui sont plus directs et descriptifs que le code de production. Ils constituent un point d'entrée non négligeable pour comprendre un système. On appelle cela créer une documentation vivante, reflet du code actuel.

### Comment créer une documentation vivante ?

Tous les types de tests sont concernés et seule la façon de les rédiger changera en fonction de la manière dont on décrit du comportement métier ou du comportement technique. Il faut cependant faire l'effort de rédiger ses tests avec attention et clarté. En effet, pour qu'ils aient une valeur documentaire, ils doivent être accessibles et faciles à comprendre. Lorsqu'il s'agit d'exprimer le comportement métier de l'application, ils doivent être écrits sous forme de spécifications et utiliser le langage du domaine.

Écrire une documentation revient à s'intéresser à deux aspects d'une application en particulier :

- Capturer son comportement,
- Capturer ses concepts métiers.

Cela tombe bien car il existe deux pratiques importantes du mouvement Craftsmanship qui vont vous permettre de couvrir ces deux points : le Behavior Driven Development et le Domain Driven Design.

#### Behavior Driven Development

Le Behavior Driven Development (BDD) est l'art de rédiger de façon collaborative les spécifications d'une application. On parle aussi de spécification par l'exemple, car les scénarios sont illustrés à l'aide de cas précis. Concrètement, il s'agit de faire en sorte que les trois acteurs types d'un projet (représentant du métier, développeurs et testeurs) travaillent ensemble pour faire émerger le comportement de l'application. Cette pratique vise à favoriser la compréhension des fonctionnalités par tous.

Les scénarios sont décrits en utilisant un langa-

ge spécifique appelé Gerkin. Il a pour particularité d'être lisible par n'importe qui et suit la syntaxe suivante :

- Given : décrit l'état initial du système ;
- When : décrit l'action effectuée ;
- Then : décrit le résultat attendu.

Le Gerkin impose un formalisme obligeant tout le monde à partager un langage commun.

Une fois ces scénarios déterminés, il existe de nombreux outils permettant aux développeurs de les "brancher" sur leur application afin de vérifier que celle-ci réponde correctement aux comportements attendus. Parmi les plus connus, citons par exemple Cucumber et JBehave.

**Domain Driven Design**

Le Domain Driven Design (DDD) est une approche initialement décrite par Eric Evans qui part du constat suivant : afin de réaliser un bon logiciel, on doit être en capacité d'en comprendre son domaine. Tout comme le BDD, il faut que les développeurs et les experts métiers se parlent et collaborent. L'ensemble du code doit être conçu, organisé et écrit, en vue de refléter le domaine pour lequel il existe (découpage et nom des classes, nom des packages, des méthodes, etc). Qui n'a jamais travaillé sur une application où un même concept se retrouve non seulement nommé de manières complètement différentes à l'intérieur du code lui-même, mais aussi dans les différents documents créés et uploadés sur le file sharing de l'entreprise ?

Qui n'a jamais travaillé sur une application où le domaine métier est relégué derrière les spécificités techniques, et où les premiers niveaux des répertoires sont "controllers", "services" ou bien encore "model" ? Le DDD fait émerger la nécessité de définir et figer un langage et un glossaire commun afin de mieux appréhender la complexité du domaine métier de l'application. On parle d'Ubiquitous Language. Il faut tout de même noter que le DDD est une pratique vivante. Les concepts et mots retenus initialement décrivant le domaine d'une application vont évoluer avec le temps et les nouveaux besoins métiers.

Ainsi, le BDD et le DDD nous permettent de créer une documentation vivante : notre application décrit son comportement et son domaine métier. Il faut maintenant être capable d'ajouter facilement de nouvelles fonctionnalités tout en garantissant la qualité de l'application.

## Coder proprement : une boîte à outils

Le mouvement Craftsmanship prône l'écriture d'un code propre à travers l'adoption de bonnes pratiques dont la plupart sont extraites de l'eXtrême Programming (XP). Bien entendu, la mise en place de ces pratiques prend du temps, les développeurs doivent changer leurs habitudes de développement et adhérer à ces nouveaux changements.

**Test First**

Cette notion correspond à l'écriture d'un test unitaire avant l'implémentation. Il décrit le comportement du code que l'on veut rédiger, garantissant que le code produit par la suite sera de qualité, et cohérent avec les attentes. Cela permet de ne coder que ce qui est nécessaire et de rester simple comme le préconise le principe KISS (Keep It Simple Stupid). Écrire les tests avant l'implémentation permet aussi de faire émerger un design, de se focaliser sur les fonctionnalités et de repérer rapidement les régressions.

**Test Driven Development(TDD)**

Cette pratique découle de la précédente (Test First). La particularité du TDD est qu'elle est basée sur un cycle « Red/Green/Refactor » :

- Red : le développeur écrit un test automatisé qui teste le comportement attendu de la nouvelle fonctionnalité. Dans un premier temps, ce test doit échouer car le code permettant de le satisfaire n'est pas encore rédigé.
- Green : le développeur écrit le minimum de code nécessaire permettant au test précédent d'être exécuté avec succès.
- Refactor : le développeur améliore le code précédemment écrit tout en maintenant les tests en succès.

**Double Loop TDD**

C'est une double boucle TDD qui consiste à ajouter d'abord un test d'acceptance (par exemple un scénario BDD) qui va entraîner une boucle TDD jusqu'à l'implémentation complète de la fonctionnalité ajoutée par le test d'acceptance Fig.1.

**Refactoring**

Le refactoring consiste à revoir de manière continue, la conception et la lisibilité du code en vue de son amélioration. Lorsque l'on achève le développement d'une fonctionnalité, revenir sur le code pour éliminer des "Code Smell" et améliorer son design semble être une bonne approche. Dans cette situation, la présence de tests se révèle un atout indispensable. Intervenir sur du code existant devient alors possible, avec l'assurance de ne pas introduire de régressions.

**Pair-Programming**

L'utilisation d'une seule machine pour deux développeurs, favorisant le partage de la connaissance métier et technique, garantit la montée en compétence globale de l'équipe.

Cette pratique est souvent critiquée car de prime abord, on a l'impression que la fonctionnalité coûte plus cher que si elle avait été implémentée par une seule personne. Mettre en évidence les difficultés que présente la fonctionnalité à être développée et résoudre les problèmes via deux angles différents représentent un réel bénéfice. Les échanges et connaissances de chacun des développeurs permettent d'aboutir à des implémentations plus efficaces, à du code plus lisible, de meilleur qualité et donc d'éviter de générer de la dette technique.

**Code review**

La relecture du code peut se faire par un équipier ou l'ensemble de l'équipe. C'est un autre moyen qui, même s'il est moins dynamique que le pair-programming, permet de partager la connaissance avec toute l'équipe, et de faire émerger des critiques constructives afin de retravailler le code en conséquence.

**Les outils**

1) L'intégration Continue

Cette pratique permet de vérifier rapidement que les changements apportés par les membres d'une équipe restent cohérents et n'entrent pas en collision. À chaque fois qu'un développeur pousse un changement sur le dépôt de code, tous les tests sont exécutés pour vérifier qu'aucune régression n'a pas été introduite. Pour être efficace, cette pratique doit être combinée avec celle du "Stop & Fix". C'est à dire que tous les membres de l'équipe doivent corriger en priorité les erreurs remontées par l'intégration continue avant de reprendre leurs tâches en cours.

2) Indicateur de qualité

Mesurer la qualité du code est difficile. Le plus simple est de choisir des indicateurs permettant de dresser un « carnet de santé ». L'outil le plus connu dans ce domaine est Sonar. Beaucoup de ces indicateurs permettent de mesurer et d'identifier la dette technique. Nous pouvons citer :

- Les portions de code non utilisées ;
- Les erreurs mal gérées ;
- La couverture des tests qui est un ratio entre la quantité de code et les fonctionnalités testées.

Bien que toutes ces pratiques aient des bienfaits, il ne faut pas chercher à les appliquer systématiquement. Si elles ne sont pas nécessaires ou que de meilleures alternatives existent, il faut savoir rester pragmatique. C'est d'ailleurs l'une des qualités indispensables d'un craftsman. Il faut constamment se demander s'il n'existe pas des pratiques plus efficaces ou plus adaptées à sa situation. La meilleure façon de comparer ces pratiques est de comparer leurs valeurs ajoutées.

# Les individus et leurs interactions au centre de vos préoccupations

*Pour réussir un projet, l'équipe doit disposer des bonnes compétences, mais cela n'est pas suffisant. En effet, fonctionner de manière totalement isolée et sans réelle interaction avec les membres de l'équipe représente pour le projet des risques de ne pas aboutir. Le produit fonctionnera sûrement, mais il ne disposera d'aucune cohérence et sera difficilement maintenable.*

Pour que votre projet soit couronné de succès, la recette à suivre est la suivante :

- Une complexité adaptée, pas "d'over-engineering " : Keep It Simple Stupid (KISS) ;
- Une documentation à jour via une documentation vivante ;
- Un code propre et lisible grâce au Pair Programming et à la Code Review.

Cependant, en ce qui concerne la mise en oeuvre, il manque un liant : "Les individus et leurs interactions" qui forment l'un des principes fondamentaux de l'agilité. En temps que Craftsman, cela doit être au centre de vos préoccupations. Les Code Review et le Pair-Programming ont pour but de favoriser les échanges durant les phases de développement mais cela reste limité. En complément, il faut être capable d'insuffler un état d'esprit dans l'équipe ; pour cela, soyez moteur afin de créer d'autres points de rencontre.

## Les sessions d'échanges ou chapters

Les sessions d'échanges ne doivent pas être prises à la légère, il faut vous réserver une plage horaire dédiée afin d'échanger sur vos dernières trouvailles et sur les outils qui vous font gagner du temps. Les informations échangées durant ces sessions n'ont pas de prix.

## Sessions de veille technologique

Prenez d'assaut les salles de réunion le midi afin de regarder les dernières conférences Web ou d'organiser un Brown Bag Lunch (BBL). Ces sessions au format conférence ou table ronde, voient généralement la présence d'un speaker désigné, interne ou externe à l'entreprise, qui présentera pendant 1 à 2 heure(s), un sujet qu'il maîtrise.

## Les coding dojos

Puisque c'est en forgeant que l'on devient forgeron, quoi de mieux que de coder pour s'améliorer ? Lors d'un Coding Dojo, regroupez-vous pour vous entraîner sur un problème en un temps limité. À la fin du temps imparti, le code est effacé, puis on recommence à coder avec un autre partenaire de travail. Le but est ainsi de se perfectionner et de découvrir de nouvelles techniques de conception ou de programmation. Ce simple moment de partage de connaissances peut s'avérer très bénéfique et enrichissant.

## Échanger avec la communauté

### Les User Groups, Meetups et Tech Events

Ces dix dernières années ont vu l'apparition de nombreuses guildes modernes : les "User Groups". Ces groupes de travail réunissent périodiquement des utilisateurs passionnés autour d'une même thématique technologique.
Ouverts à tous, ils permettent des sessions d'échange riches et vivantes. Que vous soyez simple spectateur ou bien acteur de ces communautés, les craftsmen pourront continuer à apprendre au contact de pairs.
Au delà de ces réunions mensuelles, la communauté s'est aussi organisée sur une échelle plus large en créant des conférences annuelles partout en France (Devoxx Paris, Mix-It, Breizh-Camp, XebiCon) et en Europe (Devoxx Anvers, GeeCon, Jax). Elles réunissent des milliers de passionnés et, sur un ou plusieurs jours, abordent des thèmes variés. Elles permettent de cumuler à grande échelle tous les bienfaits de la veille technologique :

- Aborder et défricher de nouveaux sujets,
- Confronter sa vision avec une vision extérieure,
- Se bâtir un réseau de "sachants",
- Apprendre des retours terrain sur la mise en place des nouvelles technologies.

### Les MOOC

Autrefois plébiscités par des étudiants de tous âges, l'Université de Tous les Savoirs (des cours universitaires gratuits et accessibles à tous) se retrouve aujourd'hui massivement en ligne à travers les Massive Open Online Courses. Pour une fois, les cordonniers ne sont pas les plus mal chaussés car une grande partie de ces cours en ligne traite d'informatique. Si vous avez la volonté d'apprendre, il y a forcément une ressource en ligne qui vous permettra d'assouvir votre curiosité.

### Les hackathons

Vous avez sûrement entendu parler des hackathons. Sur un jour ou un week-end entier, un hackathon est un challenge d'innovation vous proposant de développer un logiciel en équipe, dans un délai imparti. À la fin du concours, chaque équipe présente son travail à un jury, constitué le plus souvent de dirigeants. Le produit le plus convaincant, le plus innovant ou le plus fun aura une récompense.
Ces évènements offrent l'opportunité de progresser collectivement via divers biais :

- Renouveler les équipes : de par leur format, les hackathons proposent souvent de faire collaborer des personnes qui d'ordinaire se trouvent dans des pôles différents. C'est une bonne occasion de favoriser la mixité.
- Proposer un cadre plus libre : généralement, la seule règle imposée par le hackathon est le thème. Le choix des frameworks de développement et l'idée précise est laissée aux équipes. Il s'agit souvent d'une belle occasion pour expérimenter de nouvelles technologies, voire mettre en avant des idées innovantes d'un point de vue fonctionnel.
- Faire émerger des produits et des cas d'usages innovants : c'est souvent le but recherché par les hackathons.

Pour résumer la réussite d'un projet passe par :

- La documentation qui, à l'image d'un mode d'emploi, ne doit pas différer du produit : il faut que la documentation vive avec le produit.
- La mise en oeuvre des techniques telles que le Double Loop TDD, le pair-programming ou encore les Coding-Dojo permet de produire un code de qualité et ceci de manière interactive.
- Favoriser la collaboration et la communication entre les différents acteurs afin d'assurer la cohérence de votre projet.

Le développement logiciel n'est qu'un maillon de la chaîne qui conduit au succès d'un produit. Il existe également des pratiques pour bien définir son besoin et gérer l'exploitation du produit de façon efficace ; jetez un oeil aux pratiques XP ou Agiles telles que Scrum, Kanban.
Fini le temps où il fallait essayer de tout prévoir à l'avance et de cadrer l'ensemble du projet, qui, de plus, enlève la souplesse et toute possibilité d'innovation. Cette devinette est insoluble, il y a trop de facteurs inconnus ou variables. Aujourd'hui, le produit, l'équipe, le code, la documentation doivent vivre et évoluer ensemble. Et vous, vous commencez quand ? ⊞