SOFTWARE CRAFT

SOFTWARE CRAFT

TDD, Clean Code et autres pratiques essentielles

Cyrille Martraire
CTO et co-fondateur d'Arolla

Arnaud Thiéfaine
Développeur, coach craft et formateur chez Arolla

Dorra Bartaguiz
Développeuse, coach technique et formatrice chez Arolla

Fabien Hiegel
Développeur chez Shodo Nantes

Houssam Fakih
CTO de La Combe du Lion Vert

DUNOD

Illustration de couverture : © Dhujmontra – Shutterstock

© Dunod, 2025
11 rue Paul Bert, 92240 Malakoff
www.dunod.com
ISBN 978-2-10-088669-2

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2° et 3° a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

TABLE DES MATIÈRES

Note sur la seconde edition IX

Avant-propos XI

PARTIE 1
Les pratiques incontournables du craft

1 Le Développement Dirigé par les Tests (TDD) 2

1. Une technique pour programmer efficacement des fonctionnalités complexes 2
2. Les principes et les étapes du TDD 3
3. Rigueur nécessaire à chaque étape 5
4. Anatomie d'un test 6
5. Autres conseils sur les tests 8
6. TDD en pratique sur un kata très simple 8
7. Conclusion 20

2 Techniques et principes de propreté de code (Clean Code) 22

1. Programmer comme un exercice de communication 22
2. L'importance d'être exemplaire 23
3. Principes essentiels pour un code lisible et évolutif 24
4. Des commentaires à consommer avec modération 34
5. Ne pas se répéter (DRY) 37
6. La propreté de code aussi pour les tests 38
7. Conclusion 39

3 Les spécifications agiles avec le développement dirigé par le comportement (BDD) 40

1. Les 3 amigos (tres amigos) 41
2. Des rôles complémentaires 41
3. 3+ amigos 42
4. Les ateliers de spécifications 42
5. Exemples concrets 43
6. Des exemples exprimés dans le langage du métier 43
7. Après les ateliers de spécifications 46

8. Automatisation des scénarios 47
9. Un effort, beaucoup de bénéfices 51
10. Techniques complémentaires 52
11. Démarrer BDD 54

4 Collaborer efficacement avec le pair & mob programming 56
1. Le binômage (pair programming) 56
2. Comment binômer ? 57
3. Styles de binômage 58
4. Motivations pour choisir de binômer 64
5. La vie à deux 66
6. Le mob programming (ou ensemble programming) 72

5 L'importance des techniques de refactoring 75
1. Transformer le code pour l'améliorer 75
2. Le code : une matière en perpétuelle transformation 76
3. Le refactoring comme antidote de la dette technique 76
4. Les freins au refactoring 77
5. Un principe fondateur : exprimer l'intention 78
6. Principes méthodologiques de refactoring 78
7. Patterns de refactoring : la boîte à outils 79
8. Refactoring vers des design patterns 84
9. Combinaison de patterns 85
10. Situations nécessitant du refactoring 86
11. Situations impropres au refactoring 86
12. Refactorer aussi le code de test 87
13. Conclusion 88

6 Travailler avec du code legacy 89
1. L'enfer du legacy 90
2. Apprivoiser le code legacy 91
3. Tester le code legacy 94
4. Rendre testable le code legacy 104
5. Refactorer le code legacy 110
6. Pour aller plus loin : l'approche scratch refactoring 113
7. Pour aller plus loin : l'approche d'amélioration marginale 113
8. Conclusion 114

7 Étude détaillée du kata Fraction 115
1. Pourquoi les fractions ? 115
2. Présentation du kata Fraction 116
3. Premier exercice : implémentation sans contraintes 116

4. Décomposition 118
5. Exploration 119
6. Découpage 120
7. Second exercice : implémentation avec l'approche TDD 122
8. Les quatre règles de design simple appliquées au kata 135

PARTIE 2
Techniques avancées et élargissements

8 Principes et outils pour tester efficacement 139
1. Les caractéristiques des tests : les tests unitaires F.I.R.S.T. 139
2. Doublures de tests (test double) 142
3. Types de tests, au-delà des tests unitaires 154
4. Conclusion 162

9 Outils et techniques avancées de TDD 163
1. Baby steps et ordonnancement des tests 163
2. Styles de TDD 165
3. L'idée du TDD poussée au maximum : *TDD as if you meant it* 174

10 Techniques de conception 180
1. Principes essentiels de conception 180
2. Le jeu de contraintes « Object Calisthenics » 185
3. Principes de programmation fonctionnelle 192
4. Code smells courants 193
5. Design patterns classiques comme destinations de refactorings 203

11 Transformations de code à caractère architectural 210
6. Découpler un lien navigable par une méthode de recherche 211
7. Refactorer vers une architecture hexagonale 214
8. Renommer une colonne de base de données, un champ d'une requête ou une réponse d'un service 218
9. Évoluer progressivement vers un style réactif orienté chorégraphie 218
10. Rendre un service stateless 226
11. Rendre un service idempotent naturellement 231

PARTIE 3
Craft et attitudes

12 **Introduire le craft dans votre contexte** 234
1. Démarrer le craft à titre individuel ou entre collègues 234
2. Introduire le craft dans une organisation 236
3. Les causes d'échec du craft 239

13 **Au-delà des pratiques, un état d'esprit** 241
1. Développer son savoir-faire 241
2. Cultiver son savoir-être 247
3. Professionnalisme 254

14 **Craft 2.0** 260
1. Attitudes et aptitudes 260
2. Le craft en contexte (big) data 262
3. Le craft en contexte DevOps 263
4. Le craft en contexte machine learning 263
5. Le craft en contexte cloud & serverless 264
6. Évolution et suite du craft 264

15 **IA et Craft** 266
1. Le raz de marée de l'IA générative 266
2. Applications de l'IA générative pour le développement logiciel 269
3. Techniques de prompting pour le développement logiciel 271
4. IA et automatisation, au cœur de nombreux paradoxes 277
5. IA et enjeux de société 279

16 **Craft, Conception et Domain-Driven Design (DDD)** 283
1. L'impact de DDD sur le craft 283
2. L'importance du langage 284
3. Modéliser en code 285
4. Des Bounded Contexts, avec chacun son modèle métier propre 286

Annexe 1 : Katalogue, les katas incontournables 289

Annexe 2 : La bibliothèque idéale des crafters 291

Index 297

NOTE SUR LA SECONDE EDITION

La première édition de ce livre a reçu de nombreux retours positifs, ce qui nous a encouragés à conserver l'essentiel du texte et à limiter les modifications au strict nécessaire, c'est-à-dire aux corrections indispensables.

Cependant, le monde a déjà changé depuis la première édition. L'intelligence artificielle (IA) est soudainement devenue un sujet incontournable, suscitant à la fois des inquiétudes et des fantasmes dans nos métiers du logiciel. Nous avons donc choisi d'ajouter un chapitre spécifiquement consacré au lien entre l'IA et le craft, le chapitre 15.

En réponse à la question « Pourquoi ce livre sur le craft ne parle-t-il pas de Domain-Driven Design ? », nous avons jugé utile d'y répondre brièvement dans un petit chapitre additionnel, le chapitre 16.

Mentionnons quelques évolutions dans le monde du software craft depuis la première édition. En ce qui concerne le TDD et la conception, le livre « Tidy First? » de Kent Beck est paru. Cet ouvrage est le fruit de plusieurs années de réflexion sur le métier de développeur, par l'auteur de « Extreme Programming ». À travers une quinzaine de patterns, sortes de mini-refactorings, Kent montre comment réaliser de petits nettoyages de code qui peuvent considérablement simplifier le travail quotidien.

Concernant le Mob-Programming, le terme de 'Teaming' ou 'Ensemble Programming' a peu à peu gagné en usage comme alternative au terme de 'mob' et à ses connotations négatives. De plus, des retours d'expérience ont relevé des limites sur le plan de l'inclusivité du teaming ou mob-programming, qui n'est pas adapté à certaines situations de handicap, par exemple pour les personnes autistes ou sourdes et malentendantes.

Nous tenons à remercier tous nos lecteurs et lectrices qui nous ont fait confiance et qui ont régulièrement recommandé ce livre à leur entourage. Merci de nous avoir signalé les quelques coquilles cachées dans le livre, avec un immense merci à Matthieu Bonnot et à sa vigilance, qui a lui seul les a presque toutes trouvées.

Un merci tout particulier à celles et ceux qui ont partagé des appréciations en ligne et des retours qui nous ont fait chaud au cœur !

AVANT-PROPOS

Le Software Craft est une approche de développement logiciel qui connaît une certaine popularité depuis 2009 dans le monde entier. Cette approche, désormais revendiquée par de nombreux professionnels, s'inscrit dans le prolongement des approches agiles et les complète avec les aspects d'ingénierie.

L'essor du Software Craft dans le monde

Chaque année des milliers de développeurs et développeuses se retrouvent lors du Global Day of Code Retreat dans 250 villes du monde pour une journée de pratique intensive autour d'exercices de code en simultané, durant 24 heures au total[1].

En outre, depuis 2011, des centaines de professionnels du développement logiciel se retrouvent durant trois jours dans un lieu tranquille pour échanger leurs expériences en pratiquant ensemble. Les plus connues de ces conférences sans programme pré-établi, qu'on appelle ainsi des « non-conférences », sont les événements SoCraTES (International Conference of Software Craft & TESting) qui existent désormais dans de nombreux pays, dont bien entendu la France[2] à l'initiative de Houssam Fakih (un des co-auteurs de ce livre). À Paris, la communauté Software Crafters[3] créée par Cyrille Martraire (également co-auteur de ce livre) fin 2011 comptait en 2020 plus de 4 200 membres, dont environ 60 d'entre eux se retrouvent une soirée chaque mois pour échanger et pratiquer.

Les entreprises suivent et cherchent à adopter ces pratiques, dans une optique d'amélioration de leur processus de développement. Les compétences de Software Craft deviennent désirables voire requises sur les CV des candidats. Les entreprises veulent sponsoriser les communautés Software Crafters. Les entreprises de services se positionnent en réponse à la demande croissante de leurs clients. L'offre commerciale est aujourd'hui bien développée avec des formations, des formules d'accompagnement en coaching ou avec des « coachs craft embarqués », avec des personnes rompues au Software Craft qui intègrent une équipe pour lui transmettre les pratiques et techniques de l'intérieur.

Même si ce livre peut sembler tardif sur ce sujet, il reste pourtant absolument d'actualité, ce qui est étonnant dans une industrie où une technologie est souvent caduque après dix ans. Mais le Software Craft n'est pas une technologie de plus qu'on télécharge, qu'on installe et qu'on apprend avec un tutoriel avant de l'abandonner au profit

1. Pour en savoir plus sur cet événement annuel, rendez-vous sur le site https://www.coderetreat.org
2. Toutes les informations sur l'événement SoCraTES sont disponibles ici : https://socrates-fr.github.io/
3. Le site officiel de la communauté Software Crafters Paris se situe à l'adresse suivante : https://www.meetup.com/fr-FR/paris-software-craftsmanship

d'une autre ; c'est un ensemble *d'attitudes* qu'on apprend à adopter pour être plus efficace sur la façon d'utiliser les technologies, quelles qu'elles soient. Il s'agit d'acquérir de nouveaux réflexes, de faire le deuil de certaines illusions, de prêter attention à des enjeux qu'on ignorait ou négligeait auparavant. Il s'agit surtout de redéfinir ce qu'on appelle un « bon » logiciel. Et ce qui est remarquable avec les attitudes est qu'elles sont bien plus pérennes que les technologies sur lesquelles on les applique.

Changer sa vision du monde et sa façon de se comporter demande du temps, des efforts et un peu d'inconfort temporaire ; il faut considérer cela comme un investissement avec des bénéfices persistants sur une longue période. Apprendre le Test-Driven Développement (TDD) en Java, C#, Typescript ou dans tout autre langage consiste avant tout à adopter un état d'esprit porteur d'attitudes fondamentales et essentielles qui seront tout aussi utiles dans n'importe quel autre langage existant ou encore à inventer. Une fois les attitudes acquises, vous les appliquerez avec profit dans n'importe quel univers ; par exemple, vous voudrez probablement utiliser une approche similaire au TDD même sur des configurations d'infrastructures, des bases de données ou du cloud, et même dans des environnements orientés data ou machine learning.

Votre parcours vers le Software Craft consiste donc à pratiquer des techniques sélectionnées, idéalement en compagnie d'autres praticiens, en vue d'adopter de nouvelles attitudes. Pour cela, préparez-vous à être surpris !

Les enjeux du développement logiciel remis en question

Votre voyage dans le Software Craft commence par changer ce qui vous paraît le plus important, en prenant conscience du fait que les enjeux suivants sont parmi les plus importants aujourd'hui :

Réussir à faire fonctionner le logiciel n'est que la première étape

Working code is a low bar.

Autrement dit, un logiciel qui fonctionne, ce n'est que le strict minimum. Le vrai enjeu du développement logiciel est de rester évolutif dans la durée, c'est-à-dire permettre d'ajouter facilement des fonctionnalités, et de rester maintenable, c'est-à-dire permettre de changer facilement des fonctionnalités existantes.

Un logiciel n'est jamais fini et change en permanence

Un logiciel, ça change tout le temps. Si le logiciel lui-même ne s'use pas, son utilité dépend en revanche de son adéquation au besoin, et ce besoin change régulièrement. Un logiciel n'est donc jamais fini, sauf lorsqu'il est mis à la poubelle en fin de vie.

En anglais le mot correspondant à « logiciel » est *software* et désigne ce qui est facile à changer, par opposition au *hardware*, le matériel. Pourtant, sans soin particulier, le code devient rapidement difficile à changer, jusqu'à parfois devenir totalement impossible à faire évoluer. C'est fâcheux et il faut alors envisager de le réécrire, ce qui est à la fois très coûteux et très risqué.

Développer c'est lire et comprendre le code au moins autant que l'écrire

Conséquence du point précédent, changer du code nécessite de le lire au préalable afin de le comprendre. Dans nos métiers, le code est bien plus souvent lu et relu qu'écrit (ou réécrit), et il convient donc d'en tirer les conséquences en termes de facilité de lecture.

Définir le besoin est au moins aussi difficile que d'écrire le code correspondant

Les développeurs ont l'habitude d'attendre une expression de besoin parfaitement définie dans chaque détail. Plus précisément, ils ou elles ont l'habitude de se plaindre de ne pas avoir une telle qualité de définition du besoin, reproches adressés aux analystes ou Product Owners.

Le Software Craft accepte l'idée que définir le besoin est éminemment difficile, avec comme conséquences, d'une part, que c'est un processus de découverte progressive, souvent même en cours d'écriture du code, et d'autre part que c'est une démarche collective incluant ceux et celles qui écrivent le code, au côté des futurs utilisateurs et des experts du domaine.

Pas de hacker ni de virtuose

Dans les communautés Software Craft, les termes *hacker* et *recherche de virtuosité* ne sont pas en vogue. Il ne s'agit pas d'impressionner par des astuces originales que seul leur auteur comprend. Il ne s'agit pas de taper le plus vite possible au clavier. Tout au contraire, c'est la réflexion sur l'opportunité même d'écrire ou non du code, puis de choisir parmi les alternatives possibles qui importent et qui méritent de prendre du temps.

Il ne s'agit pas de connaître par cœur des listes entières de commandes, alors que comprendre le métier et l'avoir en tête serait nettement plus utile. Il ne s'agit pas de faire du code qui prévoit tous les cas futurs, quand on a accepté que le futur sera toujours plein d'imprévus et qu'on préfère la simplicité et la capacité de changer facilement.

Il ne s'agit pas de trouver, dans son coin, une solution complète et qui fonctionne du premier coup au prix d'un énorme effort cérébral, alors que le travail se fait en équipe et que les problèmes vraiment intéressants sont trop complexes pour « tenir dans une seule tête ».

Retour historique sur une cascade de réactions dans l'industrie logicielle

Pour bien comprendre Software Craft, il faut rappeler le contexte historique dans lequel il est apparu, à commencer par le *Manifeste agile* publié en 2001. Durant les années précédant ce manifeste, nombre de praticiens avaient proposé des approches ou méthodes de développement itératives, à cycle court, en accordant de l'importance à la communication entre les personnes impliquées et aux boucles de rétroaction.

Si la méthode Scrum est devenue depuis la plus populaire, l'approche Extreme Programming (XP) était la plus complète, combinant les pratiques d'organisation du travail et d'ingénierie de code dans un tout cohérent. Extreme Programming, portant en substance les germes du craft, a connu au départ un succès d'estime et continue d'avoir une grande influence.

À partir de 2001, l'agilité, principalement avec Scrum, a connu un formidable essor jusqu'à devenir l'approche de référence de l'industrie du développement logiciel. La dynamique agile était une réaction au cycle en V de moins en moins adapté face à la complexité croissante des logiciels.

À peine dix ans plus tard, l'agilité avait remplacé progressivement le cycle en V dans la gestion du cycle de développement, mais trop souvent les pratiques d'ingénierie de code avaient été négligées. Les équipes ont gagné en efficacité de production du code, mais, à défaut de techniques d'ingénierie, elles empilent du code sur du code, et cette accumulation entraîne de la souffrance car elle devient de plus en plus difficile à entretenir. Des équipes ressentent alors ce que Sandro Mancuso appelle la « gueule de bois agile » (*the Agile hangover*).

Certaines figures connues du développement logiciel s'en émeuvent et réagissent. Robert C. Martin reprend le terme « Software Craftsmanship » issu du livre *Software craftsmanship: the new imperative* (2001, Addison-Wesley) et en fait un manifeste destiné à compléter le *Manifeste agile* (http://manifesto.softwarecraftsmanship.org) :

« *We have come to value:*

Not only working software, but also well-crafted software

Not only responding to change, but also steadily adding value

Not only individuals and interactions, but also a community of professionals

Not only customer collaboration, but also productive partnerships

That is, in pursuit of the items on the left we have found the items on the right to be indispensable. »

En initiant la dynamique Software Craftsmanship, l'idée est de rappeler l'importance des pratiques de code, souvent négligées depuis des années. Ces pratiques et techniques sont celles qui permettent qu'un logiciel évolue dans la durée sans se dégrader trop vite. Pour une large part, il s'agit de redonner une seconde vie à des éléments issus de l'Extreme Programming pour compléter Scrum. De façon caricaturale, on pourrait être tenté d'affirmer que le Software Craft apporte ce qui manque à Scrum pour reconstituer un semblant de Extreme Programming. Toute inexacte et grossière que soit cette affirmation, si vous connaissez ou apprenez Extreme Programming, vous avez, de fait, déjà un engagement significatif dans le Software Craft.

On a vu que l'agilité était une réaction au cycle en V et à la rigidité des processus de développement logiciel à la fin des années 1990 ; le Software Craft est à son tour

une réaction à la dérive de l'agilité qui a oublié ou négligé ses racines techniques. Ironiquement c'est donc une réaction à une réaction. À ce titre, le Software Craft est aussi affaire de marketing, avec un terme qui devient populaire pour passer des idées et donner envie de changer son comportement. Ce n'est pas honteux, c'est même utile.

Une communauté bienveillante et soucieuse de la transmission des savoir-faire

Notre expérience après dix ans dans des communautés Software Craft dans toute l'Europe et au-delà est de croiser des personnes ouvertes d'esprit et avec beaucoup de bienveillance. Ça fait très bisounours mais les exceptions sont rares. Lorsqu'il a été constaté que le nom même de « Software Craftsmanship » était genré au masculin, la décision de renommer en a été immédiate dans la plupart des communautés. La communauté de Paris s'est ainsi renommée en Software Crafters Paris.

En insistant sur l'excellence technique, le risque est de dériver vers l'élitisme, mais le Software Craft inclut un antidote qui est l'importance de la transmission des savoir-faire. Il s'agit non seulement de progresser soi-même vers l'excellence technique, mais d'aider d'autres, de tout niveau, à progresser aussi. Et bien entendu les deux objectifs se renforcent plus qu'ils ne s'opposent.

Une boîte à outils de techniques

Le Software Craft comprend bien entendu une sélection de techniques formant une boîte à outils à bien connaître et à perfectionner sans cesse. Les techniques majeures sont :

- ✓ TDD (Test-Driven Development) ou « développement dirigé par les tests », qui n'est pas une technique de test mais bien une technique de développement et de conception, avec tous les refactorings.
- ✓ BDD (Behavior-Driven Development) ou « développement dirigé par le comportement », issu de TDD mais qui a développé un état de l'art propre, en particulier autour de l'expression du besoin.
- ✓ Techniques dites de propreté de code (Clean Code), qui rassemblent les pratiques-clés pour écrire du code propre à l'échelle de la ligne de code.
- ✓ Domain-Driven Design (conception dirigée par le métier), pour guider la conception avancée et l'architecture jusqu'à l'échelle du système d'information.
- ✓ Techniques spécifiques de « Legacy Remediation » (remaniement de code hérité), pour reprendre le contrôle, mettre sous tests et réparer du code ancien et mal entretenu.
- ✓ Techniques de conception orientée objet (OO) et style de programmation fonctionnelle (FP), principes de conception SOLID, etc.