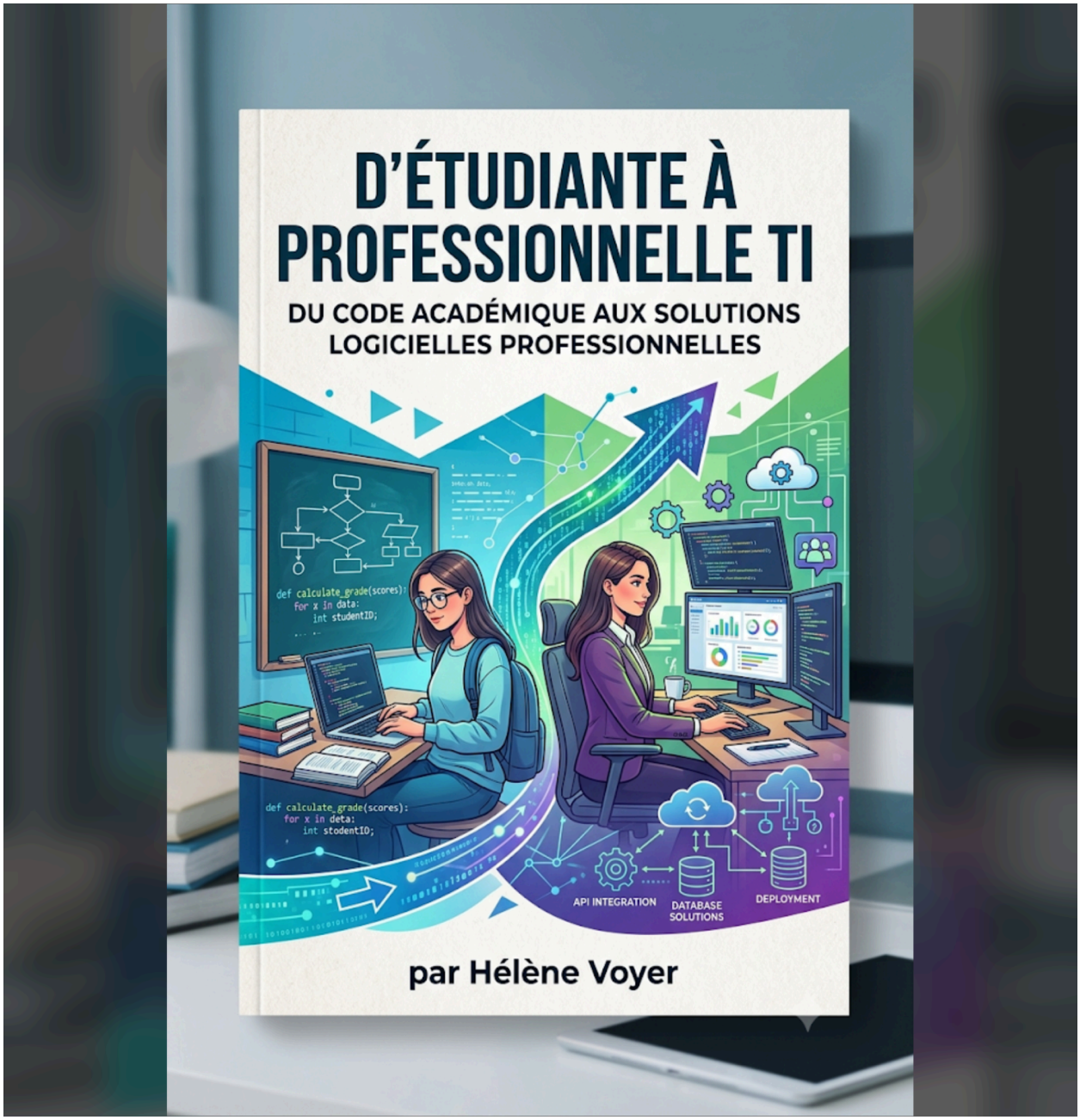
D'ÉTUDIANTE À PROFESSIONNELLE TI
DU CODE ACADÉMIQUE AUX SOLUTIONS LOGICIELLES PROFESSIONNELLES
def calculate_grade(scores):
for x in data:
int studentID;
def calculate_grade(scores):
for x in deta:
int stodentIO;
API INTEGRATION
DATABASE SOLUTIONS
DEPLOYMENT
par Hélène Voyer

Table des matières

Introduction

Vous avez choisi ce ebook car vous débutez votre premier stage ou votre premier emploi dans le développement logiciel ou un autre domaine de la technologie. Vous constaterez rapidement que travailler dans ce domaine est très différent des études universitaires. Ce ebook vous expliquera les différences entre rendre un devoir universitaire et accomplir une tâche qui vous sera confiée dans le cadre de votre nouvel emploi.

Dans le premier chapitre, vous découvrirez que la compréhension de la description des tâches qui vous sont assignées est la première étape. Une fois que vous avez une bonne compréhension des tâches, il vous faudra passer à l’étape de la planification et de la conception de votre travail. Il pourrait être nécessaire de demander l’avis d’un de vos collègues de travail plus expérimenté pour confirmer que vous n'aviez rien omis dans votre planification ou conception. Une fois que votre conception est correcte, ce sera le temps de passer à l’étape d’implémenter vos changements et de les tester. Votre tâche ne finit pas là, vos changements seront intégrés au code existant en utilisant le processus des *pull requests (PRs)*. L’étape finale est de voir votre code être déployé dans un environnement de production. Évidemment vous aurez à corriger des bogues au cours de ce processus, aussi bien dans votre code que dans le code existant. Ce chapitre vous donnera les ressources nécessaires pour que votre code soit rendu à l’étape cruciale de créer une *pull request (PR)*.

Le deuxième chapitre traitera des différentes stratégies utilisées dans l’industrie pour intégrer votre nouveau code au code existant pour le logiciel ou système que vous allez aider au développement. Votre équipe approuvera et fusionnera chaque demande de fusion que vous soumettrez.base de code.

Le versionnage consiste à identifier les nouvelles fonctionnalités et les correctifs qui seront inclus dans la prochaine version. Il s'agit surtout de décider si une révision, une version mineure ou une version majeure sera déployée à une date ultérieure. Votre organisation peut décider de publier une version par trimestre, ce qui signifie que vous travaillerez sur jusqu'à quatre versions par an. Plus de détails seront inclus à ce sujet dans le deuxième chapitre.

Le troisième chapitre abordera toutes les étapes nécessaires pour soumettre votre première demande de fusion. La soumettre est une bonne chose, mais il est préférable qu'elle soit

approuvée du premier coup. Ce chapitre vous expliquera comment faire pour que votre demande de fusion soit approuvée dès la première tentative, sans que vous ayez à vous en soucier outre mesure.

Enfin, le quatrième chapitre vous présentera les différents rythmes de travail, que votre organisation utilise les méthodologies Waterfall ou Agile. En Waterfall, les parties prenantes définissent les fonctionnalités du futur système et les rôles des utilisateurs finaux. Les analystes rédigent ensuite les spécifications, et votre équipe répartit les tâches en fonction de ces documents. Les analystes fonctionnels et techniques répondront à vos questions en cas d'ambiguïté dans la documentation.

La méthodologie Agile commence par la définition de certains éléments lors du premier sprint, mais pas de tous. Chaque sprint suivant comprend une session de planification basée sur le backlog. Ce backlog sert de référentiel évolutif où les nouvelles exigences sont ajoutées au fur et à mesure de leur apparition. Lors de la planification, l'équipe sélectionne les éléments prioritaires à inclure dans la prochaine version du sprint. Dans cette méthodologie, le processus de décision est différent et sera expliqué plus en détail dans ce chapitre.

Un changement au code n’arrête pas sa progression une fois intégré. Il doit voyager dans plusieurs environnements avant d’être utilisé en production. Dans le cinquième chapitre, nous irons au-delà du *commit* et de la fusion pour explorer ce qui se passe après la mise en production de votre code. Vous découvrirez les réalités du support, notamment la gestion des contraintes, l'interprétation des tableaux de bord de supervision et la contribution aux analyses post-mortem constructives.

La partie technique et analytique vous ayant été présentée et expliquée, le sixième chapitre traitera des aspects professionnels et humains de l'ingénierie. Il aborde les compétences « invisibles » nécessaires à la progression, telles que trouver des mentors, gérer ses relations avec son supérieur, réagir au *feedback* de vos pairs ou de vos supérieurs, contribuer activement au travail d'équipe et réussir la transition entre les résultats scolaires et la reconnaissance professionnelle.

Finalement, le dernier chapitre vous introduira aux différences entre les aspects techniques et les objectifs commerciaux. Vous apprendrez à communiquer efficacement avec les parties

prenantes, à transformer les obstacles techniques en impact commercial et à maîtriser l'art de la communication axée sur la valeur.

Je vous recommande de lire chaque chapitre dans l'ordre, surtout si vous êtes en première année de développement logiciel.

Chapitre 1 : Le cycle de vie d’un billet d’entreprise

Dans une organisation informatique, le travail est documenté et attribué via un système de gestion des tickets d'entreprise. La plupart des organisations utilisent Confluence JIRA, mais certaines sont déjà passées à Microsoft DevOps. Il existe des différences importantes entre les travaux scolaires et le travail en entreprise. Contrairement aux devoirs scolaires, vous travaillerez sur du code existant et y ajouterez votre propre code. Il est donc essentiel de bien comprendre la description du ticket et les actions nécessaires pour le résoudre. Vous participerez à une réunion de planification de sprint au cours de laquelle vous fournirez des estimations de temps pour vos tickets. Ne vous inquiétez pas.—Ces informations n'ont pas besoin d'être exactes. Lorsque vous travaillez sur vos premiers tickets, il est important de savoir reconnaître les moments où vous êtes bloqué et avez besoin d'aide. Vous entendrez de plus en plus parler de dette technique. Il s'agit du coût caché et croissant des solutions de facilité, souvent sous-optimales, utilisées lors du développement logiciel pour respecter des délais serrés.
Même les développeurs les plus expérimentés sollicitent l'avis de leurs collègues pour résoudre un problème lorsqu'ils ne sont pas certains de la meilleure solution. Une différence majeure entre les devoirs et le travail en entreprise réside dans le coût et les conséquences d'une erreur.

1.1 Le choc académique : passer des devoirs aux récits utilisateurs

À l'université, vos devoirs consistaient en des morceaux de code indépendants que vous deviez créer en fonction des consignes. Aucun ne s'intégrait au code existant. Désormais, en tant que nouvel employé en développement logiciel, votre code doit interagir avec le code existant sans perturber son fonctionnement. Certaines entreprises proposent des classes de tests unitaires JUnit qui vous permettent de vérifier que votre nouveau code n'a pas affecté le reste du logiciel ou système.

Une fois le devoir rendu à l'université, il n'était plus jamais consulté ni exécuté. Dans une entreprise informatique, le code source est examiné quotidiennement pendant des années, jusqu'à la fin de vie du logiciel ou système.Le logiciel[1] ou système basé sur ce code source fera

[1]La fin de vie d'un logiciel ou système survient lorsqu'il est jugé trop coûteux ou lorsque la clientèle cible a cessé de l'utiliser pour se tourner vers un autre logiciel ou système de la même société ou non.

l'objet de plusieurs versions dans différents environnements de test avant son déploiement final en production. Une fois en production, les utilisateurs finaux[2] l'utiliserons quotidiennement jusqu'à ce que l'organisation décide de ne plus le soutenir ou qu'il arrive en fin de vie.

 # Votre code restera dans la base de code pendant des années. Inutile de le complexifier inutilement, car vous devrez probablement y revenir plus tard pour le modifier. D'autres développeurs pourraient avoir à le consulter longtemps après votre passage à un autre projet, au sein ou non de la même organisation.

1.2 Décomposition du billet : les critères d’acceptation constituent votre plan directeur

À l'université, les critères d'acceptation étaient simples : obtenir la note de passage ou non. Dans votre nouveau travail, les critères sont différents. Le client ou le commanditaire peut avoir des testeurs qui vérifieront que le logiciel ou système fonctionne comme prévu. Mais il y a aussi les utilisateurs finaux, dont les critères d'acceptation seront différents et probablement davantage axés sur leur expérience quotidienne avec le logiciel ou système.

Chaque ticket sur lequel vous travaillerez possède ses propres critères d'acceptation. Vous les trouverez dans les spécifications ou les documents d'exigences (souvent utilisés dans les environnements de développement en cascade) ou directement dans la description du ticket.
Ne réfléchissez pas trop à votre solution. Oui, vous avez hâte de montrer vos compétences, mais ce n'est pas le sujet ici. Il s'agit de coder une solution dont la conception est suffisamment bonne pour qu'elle s'intègre bien au reste du code. Si vous surdimensionnez votre solution, vous risquez d'introduire involontairement de la dette technique ou des bogues. N'oubliez pas que les solutions à architecture complexe sont optimales lorsqu'elles sont mises en œuvre pour un problème métier qui le requiert.
Soyez fier de votre code. Voyez-le comme un moyen de bâtir votre réputation dans le secteur du développement logiciel. Assurez-vous que le prochain développeur qui travaillera sur votre code puisse le comprendre.

 # Il ne s'agit pas d'étaler votre intelligence ! Il s'agit de trouver une solution efficace immédiatement, sans engendrer de dette technique.

[2]Ce sont les utilisateurs finaux qui utilisent le logiciel. Contrairement au client ou au commanditaire, qui décident du contenu du logiciel et qui financent son développement.

1.3 Estimation des points d'histoire sans conjecture

Avec l'expérience, vous apprendrez que les développeurs seniors et les chefs d'équipe utilisent des critères pour estimer les points d'effort. Ces critères sont les suivants :

- Complexité;
- Risque et incertitude ;
- Efforts et frais généraux ;

Lors de l'évaluation de la complexité, vérifiez si le nouveau code sera isolé du code existant ; plus il sera isolé, moins la solution sera complexe. Si la solution dépend de plusieurs modules ou d'autres systèmes, sa complexité augmente.

Ensuite, vérifiez les risques potentiels et les incertitudes. Avez-vous déjà travaillé sur un projet similaire ? Si oui, le risque est moindre. Si la modification nécessite l’utilisation d’une interface de programmation d’applications ou APi[3] que vous n'avez jamais utilisée auparavant, dans ce cas, vous aurez besoin de plus de temps simplement pour apprendre à utiliser la nouvelle API[4] Les développeurs seniors ajouteraient également une marge à leur estimation pour les éléments que nous ne connaissons pas avant de concevoir ou coder activement la solution.

Enfin, il faut tenir compte des tâches annexes au développement, comme les tests unitaires, la rédaction de la documentation et le temps nécessaire à l'examen, à l'approbation et à la fusion de votre *pull request(PR)*. Le traitement de l'histoire utilisateur peut nécessiter plus de temps en cas de modifications de la base de données.

 # Dans certaines organisations qui suivent la méthodologie en cascade, vous devrez peut-être fournir deux séries d'estimations : une première qui correspond au niveau d'effort et une seconde qui sera plus précise après analyse.

[3]API est l'abréviation de Application Programming Interface (interface de programmation d'applications) et sert à partager des informations entre systèmes.
[4]Identique à la note de bas de page précédente.

1.4 La méthode professionnelle pour signaler un bloqueur

Ne vous inquiétez pas, il arrive à tout le monde d'être bloqué. Même les meilleurs développeurs de votre équipe rencontrent parfois des difficultés, mais ils ne s'y attardent pas. Ils discutent avec leurs collègues pour trouver ensemble la meilleure solution et se débloquer.

Évitez le piège de la lutte silencieuse ; si vous rencontrez un problème technique, n’attendez pas la fin de la journée pour demander de l’aide. Une heure de concentration intense suffit amplement avant de solliciter un soutien, afin d’éviter de vous épuiser inutilement. Il est bien plus professionnel de demander cinq minutes de conseils à un responsable en début de journée que de perdre toute une journée à cause de l’isolement.

Une fois que vous avez déterminé avoir besoin de l'aide d'un développeur senior, veillez à lister toutes les méthodes que vous avez essayées et à expliquer pourquoi chacune a échoué. Soyez également prêt à décrire le résultat attendu du code.

Si vous êtes bloqué en fin de journée, parlez-en lors de la prochaine réunion quotidienne. C’est le moment idéal pour un brainstorming rapide afin de trouver une solution.

À l'université, vous pouviez demander de l'aide à un assistant d'enseignement. Désormais, vous disposez de développeurs expérimentés pour vous accompagner.

 # Ne restez pas bloqué. Si vous avez passé 30 à 60 minutes à essayer différentes solutions sans succès, il est temps de demander de l'aide. Solliciter des conseils est la preuve d'un professionnalisme qui valorise le temps de l'équipe.

1.5 Objectifs et échéances

À l'université, l'objectif est simple : soumettre son code avant la date limite. Pour réussir, il faut satisfaire aux critères d'évaluation afin d'obtenir la note de passage. Une fois soumis, le code est rarement consulté ou exécuté à nouveau. Il s'agit d'une opération ponctuelle, avec un début et une fin bien définis.

Dans le monde professionnel, une date limite n'est pas qu'une simple échéance : c'est un objectif coordonné pour une mise en production. Ne pas respecter une date limite, c'est bien plus qu'un simple retard dans la réalisation d'une tâche. Cela perturbe les plannings d'assurance qualité, bloque le pipeline d'intégration et de déploiement continus (CI/CD) et décale le rythme de publication de toute l'entreprise. Respecter une date limite en bâclant le travail engendre une dette technique, obligeant vos collègues à réparer les dégâts par la suite. La fiabilité est tout aussi importante que la rapidité.

 # Les objectifs académiques visent à démontrer des compétences à court terme en vue de l'obtention d'une note. Les objectifs d'entreprise, quant à eux, visent à contribuer à un écosystème pérenne et générateur de revenus grâce à des actifs stables et hautement maintenables.

1.6 Collaboration et échelle

À l'université, les projets d'équipe reposent souvent sur une coordination informelle. On se répartit les tâches, on travaille chacun de son côté, puis on « fusionne » au dernier moment en combinant des blocs de code. En cas de conflit, un message rapide à son partenaire suffit à le résoudre. Cet isolement est possible car les projets ont une portée et une durée de vie limitées.

En entreprise, la collaboration prend une toute autre dimension. Il ne s'agit plus seulement d'écrire du code ; vous contribuez à un système complexe et évolutif, utilisé simultanément par des dizaines de développeurs. Vos modifications peuvent impacter des fonctionnalités que vous n'avez pas touchées, perturber les compilations ou engendrer des conflits logiques qui ne sont pas immédiatement visibles.

La collaboration professionnelle exige de passer d'une approche centrée sur le « mon code » à une approche centrée sur « notre logiciel ou système ». Avant d'apporter des modifications affectant la logique partagée, communiquez avec votre équipe. Considérez chaque modification comme une perturbation potentielle jusqu'à ce que vous en ayez vérifié l'impact. La mise à l'échelle ne se limite pas à l'architecture technique ; elle repose sur la discipline sociale et procédurale nécessaire pour prévenir les « collisions logiques ». Ces « collisions logiques » compromettent la stabilité du code source partagé.

 # Bien communiquer avec ses collègues est une compétence qui vous sera très utile dans votre carrière. Être introverti n'est pas un problème, mais il est essentiel de pouvoir discuter de code avec d'autres développeurs.

Définition de la liste de contrôle « Terminé » (aide-mémoire)

Catégorie de la liste de contrôle	Élément de la liste de contrôle	Description
Qualité fonctionnelle et du code	Répond aux critères d'acceptation	Toutes les exigences fonctionnelles et les conditions des récits utilisateurs définies dans le ticket sont entièrement mises en œuvre et vérifiables.
	Respecte les conventions de nommage et de style Java.	Le code respecte les règles de style de l'équipe (par exemple, camelCase pour les variables/méthodes, PascalCase pour les classes) et passe les règles de formatage standard.
	Auto-documentation et propreté	Les noms des variables et des méthodes expriment clairement l'intention ; la logique complexe est décomposée en petites fonctions lisibles, commentées uniquement lorsque cela est nécessaire.
	Aucun code mort	Les variables inutilisées, le code commenté, les importations inutilisées et la logique inaccessible ont été supprimés.
	Gestion appropriée des exceptions	Les exceptions sont interceptées ou propagées de manière appropriée, les messages d'erreur sont significatifs et les traces de pile brutes ne sont pas exposées aux utilisateurs finaux.

Tests et couverture	Tests unitaires rédigés et réussis	Les logiques nouvelles ou modifiées sont couvertes par des tests unitaires isolés qui réussissent systématiquement lors de l'exécution locale.
	Cas particuliers couverts	Les tests valident explicitement les conditions limites, les entrées nulles/vides, les scénarios négatifs et les entrées utilisateur inattendues.
	Objectif de couverture du code atteint	Le code source atteint ou dépasse le seuil de pourcentage de couverture de test requis par l'équipe pour les lignes nouvelles/modifiées.
	Tests d'intégration :	Les suites de tests d'intégration de bout en bout ou de composants permettent de vérifier que l'interaction avec les dépendances externes, les API et les bases de données fonctionne comme prévu.
Architecture et sécurité	Principes SOLID	Les principes de conception logicielle (responsabilité unique, ouvert/fermé, substitution de Liskov, ségrégation des interfaces, inversion des dépendances) sont respectés afin de garantir un code modulaire et maintenable.
	Enregistrement	Les événements, avertissements et erreurs de l'application sont enregistrés aux niveaux appropriés (DEBUG, INFO, ERROR) sans enregistrer les

		données sensibles de l'utilisateur (PII).
	Aucune valeur codée directement dans le core ni aucun secret	Les valeurs de configuration, les clés API, les identifiants et les URL sont stockés dans des variables d'environnement ou des fichiers de configuration plutôt que d'être codés directement dans le code.
	Base de données et performances	Les requêtes sont optimisées (par exemple, utilisation d'index, absence de problèmes de requêtes N+1), les ressources (flux de fichiers, connexions à la base de données) sont correctement fermées et les fuites de mémoire sont évitées.
Construction et CI/CD	Le projet local est un succès	Le projet se compile sans erreurs ni avertissements critiques à l'aide de l'outil de construction local (par exemple, Maven, Gradle).
	Analyse statique réussie	Les outils d'analyse de code (par exemple, SonarQube, SpotBugs) ne signalent aucun bug, vulnérabilité ou anomalie de code de haute gravité.
	Pipeline CI Vert	Les tâches automatisées de compilation, de test et de vérification s'exécutent avec succès sur la demande d'extraction dans l'environnement d'intégration continue.
Évaluation par les pairs et déploiement	Relations publiques auto-évaluées	L'auteur a procédé à une auto-relecture approfondie des différences dans l'interface

		utilisateur du système de contrôle de version afin de repérer les erreurs mineures avant de solliciter des commentaires.
	Description complète des relations publiques	La demande de fusion comprend un résumé des modifications, les numéros de ticket/problème associés, ainsi que les étapes ou les captures d'écran montrant comment vérifier le travail effectué.
	Approbation des pairs obtenue	Les examinateurs requis ont inspecté le code, soulevé des questions ou des préoccupations et accordé l'approbation officielle.
	Fusion et nettoyage de la branche	Le code est fusionné dans la branche cible (par exemple, main ou develop), et la branche de fonctionnalité/sujet est supprimée en toute sécurité.