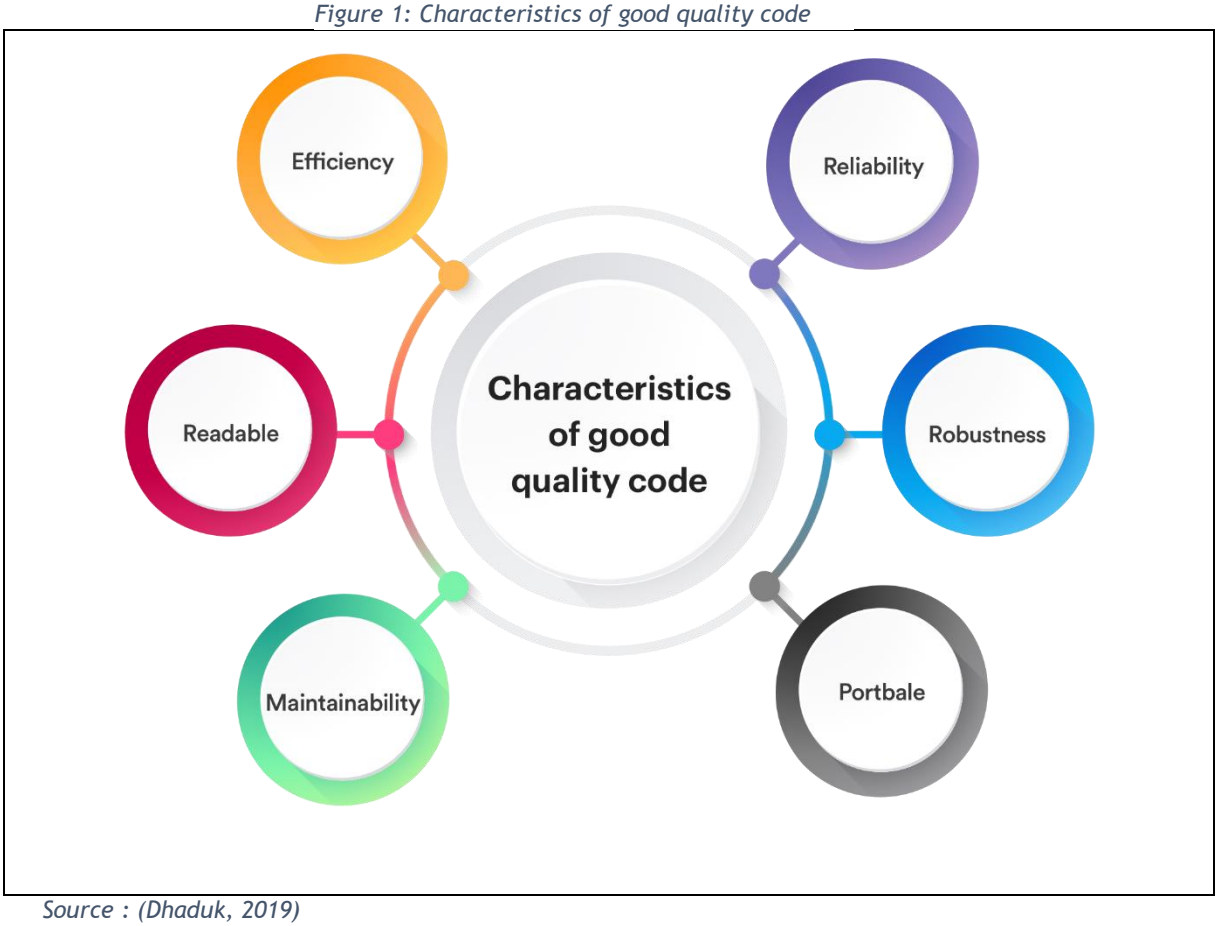
Hes·so VALAIS WALLIS
Haute Ecole de Gestion
Hochschule für Wirtschaft

Filière Informatique de gestion
Studiengang Wirtschaftsinformatik

Team Academy - Article réflexif

Intégrer les tests et l'assurance qualité dans le développement d'application

Figure 1: Characteristics of good quality code



Source : (Dhaduk, 2019)

Etudiant : Dasek Joiakim

Professeur : Russo David

Table des matières

Introduction

L'assurance qualité et les tests sont des piliers fondamentaux dans le développement de logiciels, en l'occurrence pour ce cas dans la création d'applications web où la complexité et les exigences des utilisateurs ne cessent de croître. Dans un monde en constante évolution technologique, où les applications web deviennent de plus en plus sophistiquées, l'importance de garantir leur fonctionnement, leur performance et leur sécurité ne peut être sous-estimée. Les tests (unitaires, d'intégration, de charge, etc.) et l'assurance qualité sont essentiels pour identifier et corriger les défauts, améliorer la convivialité et la performance, et finalement fournir une expérience utilisateur de qualité.

Cet article réflexif vise à explorer l'importance des tests et de l'assurance qualité dans le développement d'applications web, en examinant les défis, les stratégies et les meilleures pratiques pour intégrer efficacement une culture de tests au sein d'une équipe agile. En se basant sur l'expérience vécue pendant le développement de « Koloka », l'article se propose d'analyser l'état actuel, de réfléchir aux erreurs et aux leçons apprises, et de formuler un plan d'action concret pour rectifier le tir et assurer un avenir où la qualité et la fiabilité sont au cœur du développement.

À travers une démarche structurée inspirée de la méthode expérientielle de Kolb, cet article réflexif cherchera à transformer les expériences vécues en un cadre méthodologique robuste, guidant l'équipe vers une meilleure intégration des tests et d'assurance qualité. En adoptant une approche introspective et analytique, nous plongerons au cœur des enjeux, des défis et des opportunités que présente l'intégration des tests et de l'assurance qualité dans le développement agile d'applications web, avec un focus particulier sur le cas pratique de « Koloka ».

Dans ce voyage d'introspection et d'amélioration, nous explorerons les différentes dimensions des tests – unitaires, de couverture de code et autres. Nous examinerons comment ces pratiques, souvent perçues comme des contraintes, peuvent en réalité devenir des leviers de qualité, de performance et de satisfaction client. Nous discuterons également des défis spécifiques rencontrés par l'équipe de « Koloka » et des stratégies pour surmonter les obstacles, renforcer la collaboration et cultiver une culture de qualité omniprésente.

Expérience Concrète

Le développement de « Koloka », une application ambitieuse conçue avec Strapi et Next.js, a marqué un jalon important dans mon parcours académique. En tant qu'étudiant en deuxième année, j'ai plongé dans ce projet avec enthousiasme et dévouement, conscient des défis mais excité par les possibilités.

Cependant, à l'approche de la phase finale de développement, une prise de conscience importante a émergé : j'avais omis d'intégrer un processus systématique de tests et d'assurance qualité. Cette réalisation a été le point de départ d'une réflexion profonde sur les implications de cette omission pour la fiabilité, la sécurité et la performance de l’application.

Initialement, j'étais tellement absorbé par l'excitation de la création et par le désir de développer des fonctionnalités innovantes que la question des tests semblait secondaire. J'étais pris dans le rythme effréné de l'agilité, célébrant chaque sprint réussi et chaque fonctionnalité ajoutée. Cependant, au fur et à mesure que le projet avançait, les signes d'alerte ont commencé à se manifester. Des bugs mineurs sont apparus lors de la démonstration de nouvelles fonctionnalités et des questions de sécurité ont été soulevées durant le développement de l’application.

Le tournant a été initié par mon processus de veille technologique, qui a souligné l'importance de tests systématiques et de mesures d'assurance qualité. Cela fut un moment de vérité, me faisant prendre conscience des risques considérables liés à la mise en production d'une application non testée. Au fur et à mesure des sprints j'ai remarqué que non seulement nous n'avions pas intégré de tests unitaires mais aussi l'apprentissage du framework en profondeur montrait des fonctionnalités et stratégies pour pallier ces erreurs.

A plusieurs reprises lorsque j'ai vu des erreurs, je corrigeais immédiatement ceux-ci mais la pratique en est tout autre lorsque je me suis penché sur les stratégies pour éviter les problèmes dans le code. J'ai pu remarquer dans la partie de l'API de Strapi certains éléments dont je n'étais pas de toute confiance, par exemple le fait que certains « endpoints » permettaient l'accès à d'autres ressources d'où la connaissance de certains éléments particulier dans le framework comme les « policies » pour définir des règles en plus de la logique métier dans les « controllers ».

Les questions soulevées étaient multiples : Comment pouvais-je garantir la fiabilité de « Koloka » sans tests unitaires ? Étais-je prêt à gérer les problèmes d'intégration sans tests d'intégration appropriés ? Comment la performance pourrait-elle être évaluée sans tests de charge ? Et surtout, quelle serait la perception des utilisateurs face à une application potentiellement truffée d'erreurs ?

Observation Réfléchie

J'ai pris conscience de manière critique que j'avais négligé d'intégrer un cadre de test et d'assurance qualité dans notre processus de développement agile. Malgré les progrès réalisés dans la conception et le développement de fonctionnalités, cette omission a mis en avant une vulnérabilité majeure dans notre approche. Avec la date de mise en production qui approche rapidement, la pression pour rectifier cette situation et minimiser les risques potentiels pour l'application est devenue une priorité absolue.

L'absence de tests structurés signifiait que les bugs et les failles de sécurité pouvaient passer inaperçus jusqu'à ce qu'ils soient découverts par les utilisateurs finaux, ce qui pourrait nuire à la réputation de l'application et, par extension, à notre crédibilité en tant que développeurs. De plus, sans tests, la capacité de l'équipe à effectuer des mises à jour et des améliorations de manière agile et réactive était sérieusement compromise.

Face à cette prise de conscience, j'ai dû réévaluer mes priorités et reconnaître l'importance cruciale des tests et de l'assurance qualité. Cette réévaluation a été accompagnée d'une série de lecture à ce sujet, au cours desquelles j'ai pris conscience de mes lacunes et de la nécessité de rectifier le tir.

Cette période de réflexion a également été l'occasion de reconnaître mes forces. Mon engagement envers le projet était indéniable, et ma capacité à travailler en équipe et à apprendre rapidement m'a donné la confiance nécessaire pour relever ce nouveau défi.

Avec cette nouvelle prise de conscience, j'étais prêt à entamer la prochaine phase de mon projet : une réflexion approfondie et l'élaboration d'une stratégie pour intégrer efficacement les tests et l'assurance qualité dans le développement de « Koloka ».

En réfléchissant à notre parcours, j'ai identifié plusieurs raisons qui ont conduit à cette omission cruciale.

Premièrement, une compréhension incomplète de l'importance des tests et de l'assurance qualité. Beaucoup d'entre nous considéraient ces pratiques comme un luxe ou un effort supplémentaire plutôt que comme une nécessité fondamentale. De plus étant apprenant, ce type d'erreurs me fait relativiser, voyant que cela est une erreur plutôt courante. Bien que j'avais conscience de l'existence des tests ainsi que l'assurance de qualité, je ne l'avais pas exposé aux autres camarades au moment où il le fallait. J'avais donc bien compris que cette omission était une faute à laquelle il fallait sous peu m'y plonger. J'avais néanmoins proposé lors de la revue des user stories, une démarche de test de simulation d'une fonctionnalité faite par une personne externe pour effectuer un scénario classique afin d'uniquement vérifier le bon déroulement des user stories et qu'elles soient approuvé par l'équipe.

Deuxièmement, un manque de compétences et de connaissances spécifiques sur la manière d'implémenter efficacement les tests dans un environnement agile. Cette omission n'était pas due à un manque de connaissance ou d'accessibilité aux outils de test, mais plutôt à un manque de priorisation et à une méconnaissance de l'intégration efficace de ces pratiques dans notre processus agile.

Conceptualisation Abstraite

Fort de cette prise de conscience, j'ai entrepris de rechercher des théories et des pratiques établies pour guider mon intégration de l'assurance qualité et des tests. Des méthodologies telles que le TDD (Test-Driven Development), le BDD (Behavior-Driven Development) et les principes de l'intégration continue sont apparues comme des cadres pertinents pour ma situation.

En examinant diverses sources, j'ai découvert que l'intégration de tests rigoureux n'est pas simplement une mesure de prévention des bugs, mais un investissement dans la qualité et la durabilité de l'application. Les tests unitaires, par exemple, permettent de vérifier le fonctionnement correct de composants individuels, tandis que les tests d'intégration assurent que ces composants fonctionnent bien ensemble. Les tests de charge sont cruciaux pour s'assurer que l'application peut gérer un grand nombre d'utilisateurs simultanés, et la couverture de code offre une mesure de la quantité de code testé.

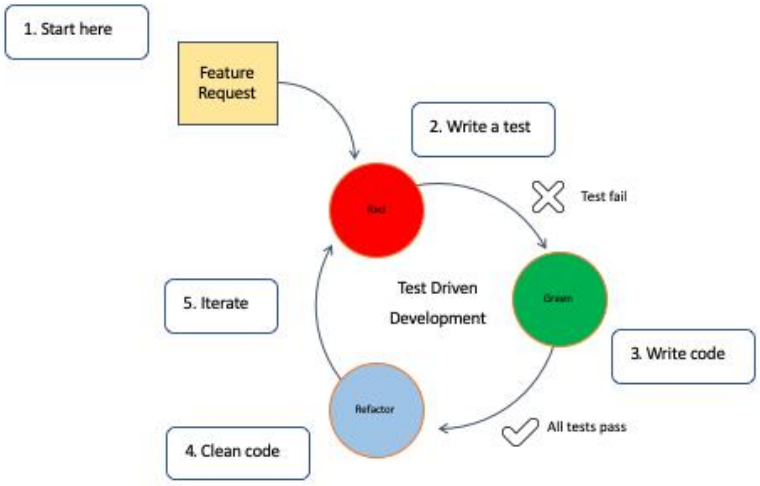
Ces pratiques, une fois intégrées, peuvent transformer le développement en un processus plus prévisible et fiable, minimisant les risques et augmentant la confiance de l'équipe dans le code qu'elle produit.

En me plongeant dans la théorie existante et en réfléchissant à mon expérience, j'ai cherché à conceptualiser une approche plus robuste et intégrée pour le projet :

1. **Importance des tests dans le cycle de développement agile :**

Je comprends qu'il est primordial de reconnaître que les tests ne sont pas un ajout après coup, mais une partie intégrale de mon développement. Les méthodologies comme TDD (Test-Driven Development) soulignent pour moi l'importance de rédiger des tests avant même le code fonctionnel. Cette approche garantit non seulement que mon code répond aux exigences dès le départ, mais favorise également un design modulaire et moins sujet aux erreurs.

Le Test Driven Development, une méthode agile, joue un rôle crucial dans l'amélioration de la qualité et de la structure du projet « Koloka ». Ce processus inversé de développement, où les tests sont écrits avant le code, guide le développement avec une approche orientée résultats. Cela se traduit par une meilleure qualité de code, une architecture plus propre et des coûts de maintenance réduits (dette technique).



[1]

Définir les tests d'abord (phase rouge) :

Nous allons devoir commencer par rédiger des tests pour les fonctionnalités souhaitées, en me concentrant sur les besoins de l'utilisateur. Ces tests doivent échouer initialement, car le code correspondant n'existe pas encore.

Écrire le code minimum nécessaire (phase verte) :

Nous devons développer ensuite le code juste suffisant pour que les tests réussissent. Cette approche garantit que nous ne créons que le code nécessaire pour répondre aux exigences spécifiques du test.

[1] https://developer.ibm.com/articles/5-steps-of-test-driven-development/

Refactoring (amélioration du code) :

Après avoir réussi les tests, il faut améliorer et nettoyez le code pour le rendre plus compréhensible, efficace et maintenable. Cela inclut l'élimination des duplications de code et l'optimisation de la structure.

Répéter le cycle TDD :

Il faut poursuivre ce cycle, écriture de test, développement du code, refactoring, pour chaque nouvelle fonctionnalité ou amélioration. Cela assure un développement incrémental et continu, où chaque étape améliore et élargit la fonctionnalité de l'application.

Évaluation continue et adaptation :

Finalement il faudra évaluer régulièrement les tests et le code pour vous assurer qu'ils répondent aux exigences et standards de qualité.

2. Stratégies CI/CD :

L'implémentation de stratégie d'intégration continue et la livraison continue (CI/CD) permettent des cycles de feedback rapides, essentiels pour un développement agile réactif. Cela englobe les tests unitaires et les tests d'intégrations.

Il s'agit d'un sujet dont je développerai au cours du semestre prochain lors d'une lecture individuelle avec un playground où je vais implémenter dans un projet prototype, l'automatisation de ces différents tests. Je ne vais donc pas aborder ce vaste sujet qu'est le CI/CD ici. Je vais simplement introduire brièvement le sujet :

L'approche CI/CD favorise des mises à jour fréquentes et fiables de l'application grâce à l'automatisation des phases de développement. Cela comprend l'intégration continue (CI),

la distribution continue (CD), et le déploiement continu, permettant une mise en production rapide et sûre.

Intégrer Continuellement (CI) : Fusionnez fréquemment les modifications de code pour identifier et résoudre les conflits rapidement. Ici nous pouvons effectuer l'automatisation des différents tests avant le déploiement dans un environnement de production ou de « staging ».

Distribuer Continuellement (CD) : Automatisez la publication du code validé dans un référentiel pour maintenir une base de code prête à être déployée.

Déployer Continuellement : Automatisez le lancement d'applications dans l'environnement de production pour des mises à jour rapides et sécurisées.

3. Pratiquer les revues de code et les analyses statiques :

Les revues de code doivent être considérées comme essentielles pour maintenir une haute qualité dans le développement logiciel. Elles me permettent de partager mes connaissances et de détecter les erreurs qui pourraient autrement passer inaperçues. En intégrant des analyses statiques, où le code est examiné sans être exécuté, je peux détecter des vulnérabilités, des erreurs de syntaxe et d'autres problèmes avant même que le code ne soit intégré. Ces pratiques encouragent non seulement une meilleure qualité de mon code mais renforcent également la cohésion et la collaboration au sein de mon équipe.

Nous utilisons déjà des librairies intégrées au projet « Koloka » comme :

- ESLint est un linter JavaScript qui peut être configuré pour fonctionner avec Next.js. Il vous permet de détecter et de corriger des erreurs de code, d'appliquer des règles de style et de garantir la cohérence du code.
- Prettier est un formateur de code qui peut être utilisé en conjonction avec ESLint pour maintenir un style de code uniforme dans votre projet Next.js.

Pour la revue de code, j'ai déjà mis en place et effectuer plusieurs « reviews » pour les « pull requests » que j'aborde dans mon premier article réflexif, pour rappel, il va falloir que je sensibilise l'équipe afin de faire comprendre l'importance et comment utiliser ces fonctionnalités au sein du projet !

4. **Adopter le pair programming pour les cas de test :**

Pratiquer le pair programming, où deux développeurs travaillent ensemble sur le même code, s'avère une excellente stratégie pour améliorer la qualité des tests. Lorsque deux développeurs collaborent, nous sommes plus susceptibles de penser à des cas de test variés et complexes, ce qui améliore la couverture des tests. De plus, cette méthode permet un partage de connaissances constant et augmente l'engagement de l'équipe, assurant que les tests sont pris au sérieux et effectués minutieusement.

Les points clés à retenir sont :

Collaboration active : Le pair programming implique une collaboration active entre deux programmeurs qui travaillent ensemble pour résoudre un problème ou développer une fonctionnalité.

Rotation des rôles : Les développeurs alternent souvent les rôles de « conducteur » (celui qui écrit le code) et de « navigateur » (celui qui réfléchit à la conception, aux tests et aux problèmes potentiels).

Meilleure qualité du code : Le pair programming favorise la discussion et l'examen approfondi du code, ce qui conduit généralement à un code de meilleure qualité et à une réduction des erreurs.

Apprentissage mutuel : Les membres de l'équipe peuvent apprendre les uns des autres, partager des connaissances et des compétences, ce qui renforce la cohésion de l'équipe.

Réduction des problèmes de code : Les erreurs sont détectées plus tôt et corrigées plus rapidement, ce qui évite les problèmes de code coûteux à résoudre à l'avenir.

Productivité accrue : Bien que le pair programming puisse sembler plus lent au premier abord, il permet souvent d'accélérer le développement global en évitant les retours en arrière et en réduisant les temps d'attente pour la résolution des problèmes.

Amélioration de la communication : Le pair programming encourage la communication continue et ouverte entre les membres de l'équipe, ce qui réduit les malentendus et les divergences.

L’atout fort sera de conjuguer « TDD » ainsi que pair programming, cela veut dire que lors d’une session que j’aurais programmé à l’avance avec un camarade du projet, nous allons pourvoir effectuer du pair programming en intégrant le processus des trois phases du TDD !

5. Psychologie de la qualité :

Cultiver une culture de qualité au sein d'une équipe est crucial et va au-delà de l'intégration des pratiques de test. Il est essentiel de reconnaître le rôle de chaque membre dans la qualité du produit final, en valorisant la proactivité face aux tests et en percevant

la découverte de bugs comme une contribution positive. Encourager une mentalité où la qualité du code et des produits est aussi importante que la rapidité de livraison peut transformer les tests d'une « tâche supplémentaire » en une partie essentielle de la fiabilité et de la performance.

Différents éléments à prendre en compte et à partager avec l'équipe :

Définir des Équipes Axées sur la Qualité : Nous devons impliquer des rôles dédiés à la qualité comme les analystes QA, les chefs de projet et les managers QA et ainsi encourager chaque membre de l'organisation à se concentrer sur la qualité.

Intégrer la qualité dans le développement : Il faut s'assurer que les développeurs, en collaboration avec les leaders techniques (développeurs expérimentés) et les ingénieurs de qualité, appliquent et respectent les normes de qualité à chaque étape du développement.

Prioriser la qualité et l'éducation : Il faut éduquer l'équipe sur l'importance de la qualité, des revues de code, des tests automatisés, du CI/CD et d'autres éléments du développement de qualité. Cela serait un point à prendre en compte pour l'expérimentation active !

Embrasser l'automatisation : Utilisez l'automatisation pour améliorer la qualité et la productivité, et veillez à ce que le code ne soit publié qu'après avoir réussi les tests automatisés. Comme vu précédemment avec le « TDD », l'intégration du CI/CD.

Adapter Agile et DevOps : Intégrez les méthodologies Agile et DevOps pour renforcer la culture de qualité et accroître l'efficacité du développement et de la livraison.

Expérimentation Active

Après une réflexion approfondie sur les lacunes et les opportunités dans notre processus de développement pour « Koloka », il est temps de passer à l'expérimentation active. Cette phase est cruciale pour transformer les idées théoriques de la conceptualisation abstraite en actions tangibles et mesurables. Je vais élaborer un plan d'action chronologique pour intégrer les tests et l'assurance qualité de manière efficace et durable dans notre cycle de développement agile.

Voici mon plan d'action :

1. **Réunion initiale avec l'équipe de développeurs :**

- Objectif : Partager les découvertes et souligner l'importance de la qualité du code.
- Actions :
    - Organiser une réunion pour discuter des risques et des conséquences de négliger les tests et l'assurance qualité.
    - Présenter des exemples concrets où l'absence de tests a entraîné des problèmes majeurs.
    - Souligner comment une meilleure qualité du code peut bénéficier à tous, à court et à long terme.

2. **Formation et sensibilisation :**

- Objectif : Éduquer l'équipe sur les meilleures pratiques de test et d'assurance qualité.
- Actions :
    - Programmer des sessions de formation sur le Test Driven Development (TDD), les revues de code, et l'analyse statique.
    - Encourager le partage de connaissances et l'apprentissage mutuel à travers des ateliers pratiques.

3. **Implémentation du Test Driven Development (TDD) :**

- Objectif : Intégrer TDD dans notre cycle de développement quotidien.
- Actions :
    - Commencer chaque nouveau sprint par la définition des tests.
    - Écrire le code nécessaire pour passer ces tests avant d'ajouter toute nouvelle fonctionnalité.
    - Effectuer des séances de refactoring régulières pour maintenir un code propre et efficace.
    - Montrer l'exemple concret de TDD avec « Next.js », notre framework pour le projet sur ce lien : https://learntdd.in/next/#write-the-code-you-wish-you-had.

4. **Intégration Continue et Livraison Continue (CI/CD) :**

- Objectif : Mettre en place une chaîne CI/CD pour des mises à jour fréquentes et sécurisées.
- Actions :
    - Configurer un serveur d'intégration continue pour automatiser les tests et les déploiements.
    - S'assurer que tout code nouvellement intégré passe par des tests automatisés avant de fusionner.
    - Préparer une lecture individuelle avec un playground où tout le monde participe de A à Z à son intégration sur un projet prototype.

5. **Adoption du pair programming pour les cas de test complexes :**

- Objectif : Améliorer la couverture des tests et la qualité du code grâce à la collaboration.
- Actions :
    - Identifier les fonctionnalités complexes ou critiques et les assigner à des paires de développeurs.

- Organiser des sessions régulières de pair programming, en alternant les rôles pour une meilleure expérience d'apprentissage.

6. **Établir une routine de revues de code et d'analyses statiques :**

- Objectif : Assurer une cohérence et une qualité de code élevées.
- Actions :
  - Intégrer des outils comme ESLint et Prettier pour des vérifications automatiques du code.
  - Planifier des revues de code régulières où chaque membre peut contribuer et apprendre des autres.

7. **Cultiver une Psychologie de la Qualité au sein de l'équipe :**

- Objectif : Encourager une culture où la qualité est la responsabilité de tous.
- Actions :
  - Créer des incitations et reconnaître les contributions à la qualité du code.
  - Encourager une attitude où trouver et corriger des bugs est vu comme une contribution positive.

8. **Évaluation et Adaptation Continues :**

- Objectif : S'assurer que les stratégies mises en place sont efficaces et les ajuster si nécessaire.
- Actions :
  - Planifier des réunions régulières pour évaluer l'efficacité des nouvelles pratiques.
  - Être ouvert aux retours d'informations et prêt à adapter les méthodes en fonction des résultats obtenus et des besoins de l'équipe.

Conclusion

L'intégration de tests et d'assurance qualité dans le développement d'applications web, comme illustré par notre expérience avec « Koloka », est impérative pour garantir des applications performantes, sécurisées et fiables. Cet article a exploré en profondeur les risques associés à la négligence de ces pratiques et a proposé un plan d'action structuré pour les intégrer efficacement dans les cycles de développement.

L'expérience avec « Koloka » a clairement démontré que les tests et l'assurance qualité ne sont pas des options, mais des nécessités. L'absence de ces pratiques peut entraîner des bugs, des failles de sécurité et, finalement, une dégradation de la confiance des utilisateurs. En revanche, leur intégration systématique permet de détecter et de corriger les problèmes dès les premières phases du développement, d'optimiser la performance et d'améliorer l'expérience utilisateur.

Les méthodologies telles que le Test-Driven Development (TDD), le pair programming, et les stratégies d'intégration continue et de livraison continue (CI/CD) se sont révélées être des moyens efficaces pour intégrer les tests et l'assurance qualité dans le processus de développement. Le TDD, en particulier, encourage une conception réfléchie et un code plus robuste en écrivant les tests avant le code lui-même. Le pair programming améliore la qualité du code grâce à la collaboration et à la révision en temps réel, tandis que les stratégies CI/CD permettent des déploiements fréquents et sûrs avec une rétroaction rapide.

Il est également essentiel de cultiver une culture de qualité au sein des équipes de développement. Cela implique de reconnaître le rôle de chaque membre dans la qualité du produit final et de valoriser la découverte et la correction des bugs comme contributions positives. Des outils comme ESLint et Prettier, ainsi que des pratiques régulières de revues

de code et d'analyses statiques, peuvent aider à maintenir une haute qualité de code et à renforcer la collaboration au sein de l'équipe.

Cependant, il est important de noter que l'intégration des tests et de l'assurance qualité est un processus continu. Il nécessite une évaluation régulière et une adaptation pour rester efficace face aux évolutions technologiques et aux changements dans les exigences des projets. Le plan d'action proposé dans cet article doit être considéré comme évolutif, avec une flexibilité intégrée pour s'adapter aux nouvelles connaissances et aux retours des utilisateurs.

En fin de compte, s'engager envers l'assurance qualité et les tests n'est pas seulement une stratégie pour éviter les problèmes, mais un investissement dans la qualité et la durabilité des applications web. Par l'adoption d'une approche méthodique et l'intégration de pratiques éprouvées, les équipes de développement peuvent non seulement améliorer la qualité de leurs applications mais aussi renforcer la confiance des utilisateurs et réduire les coûts et les efforts à long terme.

Bibliographie

Dhaduk, H. (2019, 12 26). *code-quality*. Récupéré sur https://www.simform.com/: https://www.simform.com/blog/code-quality/

Make-quality-a-priority-in-your-software-engineering-culture. (2022, 09 13). Récupéré sur https://www.techtarget.com/: https://www.techtarget.com/searchsoftwarequality/tip/Make-quality-a-priority-in-your-software-engineering-culture

pair-programming. (s.d.). Récupéré sur https://www.codementor.io/: https://www.codementor.io/pair-programming

quest-ce-que-le-test-driven-development. (2020, 09 29). Récupéré sur https://www.ionos.fr/: https://www.ionos.fr/digitalguide/sites-internet/developpement-web/quest-ce-que-le-test-driven-development/

what-is-ci-cd. (2023, 07 28). Récupéré sur https://www.redhat.com/: https://www.redhat.com/fr/topics/devops/what-is-ci-cd

write-the-code-you-wish-you-had. (s.d.). Récupéré sur https://learntdd.in/: https://learntdd.in/next/#write-the-code-you-wish-you-had