ForEach
ACADEMY

Qualité Logicielle
JAVA

foreach-academy.fr

Qualité Logicielle JAVA

Programme

1. Introduction à la qualité logicielle
2. Métriques dans un environnement Java
3. Outils de qualimétrie dans Java
4. Tests
5. Développement en TDD
6. Développement en BDD

Introduction à la qualité logicielle

Introduction à la qualité logicielle

- Notion de dette technique / Problèmes de la non-qualité.
- Qualité, normes et certification.
- Importance de la qualité du code dans le développement logiciel.
- TP : Analyse de quelques exemples de code non qualitatis

Notion de dette technique

- La **dette technique** est un concept en génie logiciel qui **reflète le coût implicite de l'ajout de fonctionnalités supplémentaires à un logiciel**, dû au choix de solutions faciles (mais incorrectes) au lieu d'utiliser une meilleure approche qui prendrait plus de temps.
- L'analogie de la dette est appropriée parce qu'une **dette a un coût** : l'**intérêt**.
- De même, choisir une solution rapide et facile peut avoir un coût supplémentaire plus tard, lorsque vous devrez refactoriser votre code, le déboguer ou le documenter. **Plus vous attendez pour "rembourser" cette dette, plus les "intérêts" s'accumulent sous forme de coûts de maintenance supplémentaires.**

Problèmes de la non-qualité

La dette technique peut se manifester de différentes façons :

1. **Code mal structuré** : Le code peut être difficile à lire, mal organisé, ou ne pas respecter les conventions de codage standard. Cela peut rendre le code difficile à comprendre et à maintenir.
2. **Duplication de code** : Au lieu de réutiliser le code existant ou de créer des fonctions réutilisables, le même code peut être répété à plusieurs endroits. Cela peut rendre le code plus difficile à maintenir et plus sujet aux bugs.
3. **Manque de tests** : Si le code n'est pas bien testé, il peut contenir des bugs non détectés. Les tests sont également importants pour vérifier que les modifications du code n'introduisent pas de nouveaux bugs.
4. **Manque de documentation** : Sans une bonne documentation, il peut être difficile pour d'autres développeurs de comprendre comment le code fonctionne et comment l'utiliser ou le modifier.

Qualité du logiciel

- La **qualité du logiciel** est une évaluation de la façon dont le logiciel se conforme aux exigences fonctionnelles et non fonctionnelles, aux normes de développement et aux caractéristiques désirables telles que la maintenabilité, la performance et la facilité d'utilisation.

1. **Fonctionnalité** : Le logiciel doit répondre à toutes les exigences fonctionnelles spécifiées. Il doit accomplir les tâches pour lesquelles il a été conçu de manière précise et fiable.
2. **Performance** : Le logiciel doit être efficace dans son utilisation des ressources système et doit maintenir une vitesse d'exécution acceptable même lorsque la charge est élevée.
3. **Fiabilité** : Le logiciel doit être capable de fonctionner de manière constante et correcte dans diverses conditions, et doit être capable de se récupérer gracieusement des erreurs.
4. **Utilisabilité** : Le logiciel doit être facile à utiliser et intuitif, minimisant la courbe d'apprentissage pour les nouveaux utilisateurs.
5. **Maintenabilité** : Le logiciel doit être conçu de telle manière qu'il soit facile à comprendre, à modifier et à étendre. Un code bien structuré, lisible et documenté est essentiel pour la maintenabilité.
6. **Évolutivité** : Le logiciel doit être capable de s'adapter et de croître en fonction des exigences changeantes et de l'augmentation de la charge.

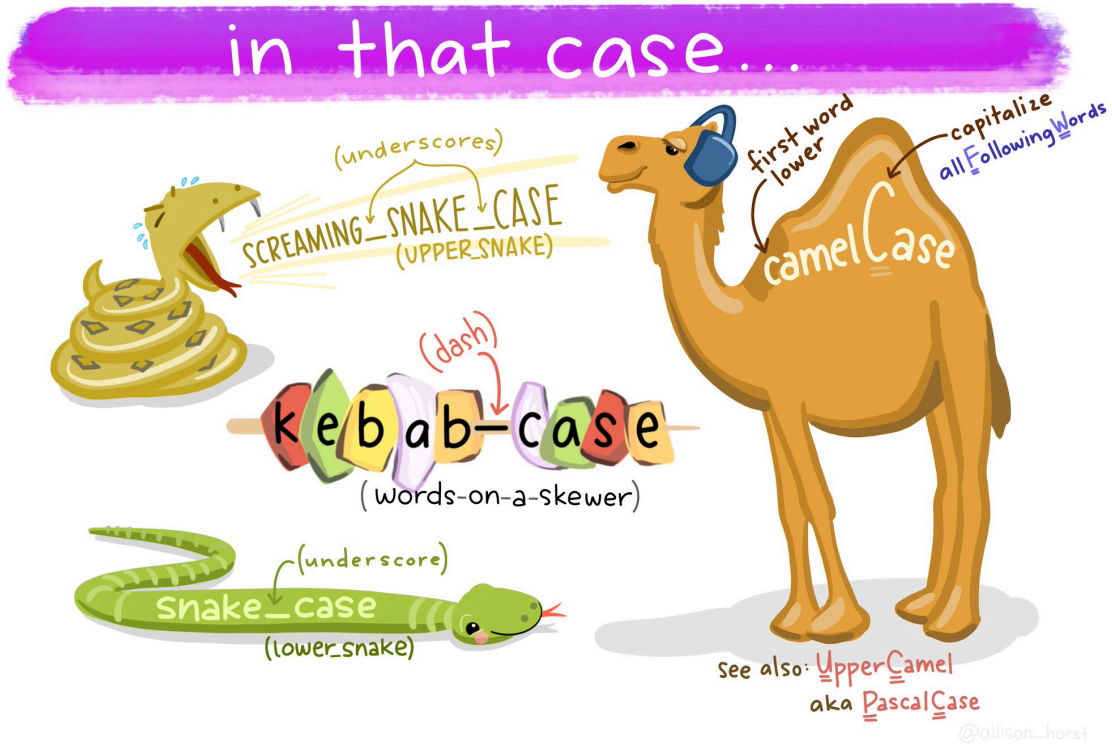
Normes de codage

- Les **normes de codage**, également connues sous le nom de conventions de codage, sont un ensemble de lignes directrices, de règles et de spécifications que les développeurs de logiciels suivent pour écrire le code source d'un programme.
- Ces règles régissent la structure, l'apparence, les conventions de nommage et d'autres aspects du code.

1. **Lisibilité et compréhension du code** : Les normes de codage aident à garder le code propre, organisé et facile à lire, ce qui est crucial pour la compréhension et la maintenabilité du code.
2. **Cohérence** : Elles assurent une cohérence dans le code, ce qui signifie que le code semble avoir été écrit par une seule personne, même s'il a été écrit par une équipe.
3. **Productivité** : En rendant le code plus facile à comprendre, les normes de codage peuvent augmenter la productivité des développeurs.
4. **Qualité** : Les normes de codage aident à prévenir les erreurs, en particulier en forçant les bonnes pratiques.

Exemples de normes de codage en Java

1. **Conventions de nommage** : Par exemple, les noms de classe en Java devraient commencer par une majuscule, et si le nom est composé de plusieurs mots, chaque mot supplémentaire doit également commencer par une majuscule => PascalCase (ex : ClassName). Les variables et les méthodes doivent commencer par une minuscule et suivre la règle camelCase (ex : variableName).
2. **Directives de formatage** : Par exemple, l'indentation doit être de 4 espaces, et les accolades doivent suivre le style K&R ("Egyptian brackets").
   [Liste des indentations]



Exemples de normes de codage en Java

3. **Commentaires** : Toutes les classes et méthodes devraient avoir des commentaires Javadoc expliquant leur but, leurs paramètres et leur valeur de retour. Les commentaires de ligne ou de bloc peuvent être utilisés pour expliquer des parties de code particulièrement complexes.

```java
//Exemple de Javadoc pour une classe
/**
 * La classe {@code Person} représente une personne avec un nom, un âge et une adresse.
 * Elle fournit des méthodes pour accéder et modifier ces attributs.
 *
 * <p>Exemple d'utilisation :</p>
 * <pre>{@code
 * Person person = new Person("John Doe", 30, "123 Main St");
 * System.out.println(person.getName());
 * }</pre>
 *
 * @see Address
 */
public class Person {
    // ...
}

//Exemple de Javadoc pour un constructeur
/**
 * Crée une nouvelle instance de {@code Person} avec le nom, l'âge et l'adresse spécifiés.
 *
 * @param name le nom de la personne
 * @param age l'âge de la personne
 * @param address l'adresse de la personne
 */
public Person(String name, int age, String address) {
    this.name = name;
    this.age = age;
    this.address = address;
}
```

```java
// Exemple de Javadoc pour une méthode

/**
 * Retourne le nom de la personne.
 *
 * @return le nom de la personne
 */
public String getName() {
    return name;
}

/**
 * Modifie le nom de la personne.
 *
 * @param name le nouveau nom de la personne
 */
public void setName(String name) {
    this.name = name;
}

/**
 * Calcule l'âge de la personne en fonction de l'année de naissance.
 *
 * @param birthYear l'année de naissance de la personne
 * @return l'âge calculé de la personne
 */
public int calculateAge(int birthYear) {
    return Year.now().getValue() - birthYear;
}
```

Certification

- La **certification** de qualité logicielle est un processus par lequel une organisation indépendante évalue les processus de développement de logiciels d'une entreprise pour s'assurer qu'ils respectent certaines normes de qualité. L'objectif est de valider que le logiciel est développé de manière contrôlée et efficace, réduisant ainsi les risques d'erreurs, de défaillances et de problèmes de sécurité.

Différentes certifications de qualité logicielle existent :

1. **ISO 9000** : Il s'agit d'une série de normes internationales qui établissent des critères pour les systèmes de gestion de la qualité. L'ISO 9000 est applicable à tous types d'organisations, pas seulement aux entreprises de logiciels. Il s'agit essentiellement de s'assurer que les organisations ont des processus standardisés et contrôlés pour fournir un produit ou un service de qualité.
2. **CMMI (Capability Maturity Model Integration)** : C'est un modèle de maturité qui fournit des directives pour améliorer les processus d'une organisation, avec un accent particulier sur le développement de produits, y compris les logiciels. Les organisations peuvent être évaluées sur une échelle de 1 à 5 pour déterminer leur niveau de "maturité" en matière de processus.

Certification Clean Code

Concernant la **certification Clean Code**, celle-ci est centrée sur le développement de logiciels. Elle vise à valider que les développeurs comprennent et appliquent les principes de développement "propre" - c'est-à-dire un code qui est facile à lire, à comprendre et à maintenir.

La certification Clean Code couvre une variété de sujets, tels que :

- Les principes SOLID
- Les conventions de codage et de formatage
- Les bonnes pratiques de commentaires
- L'importance des tests unitaires et de l'intégration continue
- Comment éviter les anti-modèles et le code spaghetti

Les principes SOLID

- Les **principes SOLID** sont un ensemble de principes de conception logicielle qui visent à créer des systèmes évolutifs, flexibles et faciles à maintenir. Ces principes sont souvent mentionnés dans le contexte du Clean Code,

1. Principe de responsabilité unique (**Single Responsibility Principle - SRP**) : Ce principe stipule qu'une classe ou un module ne devrait avoir qu'une seule raison de changer. En respectant le SRP, on s'assure que chaque classe a une responsabilité claire et bien définie, ce qui facilite la compréhension, la maintenance et les modifications du code.
2. Principe d'ouverture/fermeture (**Open/Closed Principle - OCP**) : Ce principe énonce que les entités logicielles (classes, modules, etc.) devraient être ouvertes à l'extension, mais fermées à la modification. Cela signifie que vous devriez pouvoir ajouter de nouvelles fonctionnalités en ajoutant de nouveaux modules ou en dérivant de classes existantes, sans avoir à modifier le code existant. Cela permet d'éviter les effets de bord et de maintenir la stabilité du système.

Les principes SOLID

3. Principe de substitution de Liskov (**Liskov Substitution Principle - LSP**) : Ce principe met l'accent sur l'héritage et stipule que les objets d'une classe dérivée doivent pouvoir être substitués par des objets de leur classe de base sans altérer la cohérence du système. En respectant le LSP, on garantit une utilisation cohérente des sous-classes et une meilleure modularité du code.
4. Principe de ségrégation des interfaces (**Interface Segregation Principle - ISP**) : Ce principe recommande de découpler les interfaces en des ensembles de méthodes cohérentes plutôt que de créer des interfaces monolithiques. Cela permet aux clients d'utiliser uniquement les méthodes dont ils ont besoin, évitant ainsi les dépendances inutiles. En appliquant le ISP, on facilite la maintenance et la réutilisation du code.
5. Principe d'inversion des dépendances (**Dependency Inversion Principle - DIP**) : Ce principe indique que les modules de haut niveau ne devraient pas dépendre directement des modules de bas niveau, mais plutôt d'abstractions. Cela favorise la modularité, la flexibilité et la testabilité du code. En respectant le DIP, on peut réduire les dépendances directes et rendre le code plus facilement extensible.

Amélioration continue

- L'**amélioration continue** de la qualité est une approche systématique pour améliorer les processus de travail et les produits de manière continue. Elle souligne que la qualité ne doit pas être considérée comme un objectif unique, mais comme un processus en constante évolution visant à augmenter la satisfaction des clients et à améliorer les performances opérationnelles.
  => Mise en place dans les Méthodes Agiles, DevOps, ...
- La **méthodologie Kaizen**, originaire du Japon, est l'un des cadres les plus populaires pour l'amélioration continue. Kaizen, qui signifie "amélioration continue" en japonais, se concentre sur l'identification et l'élimination des gaspillages - c'est-à-dire des processus ou des activités qui n'ajoutent aucune valeur.
  => Similaire au Lean du DevOps (créé par Toyota pour l'industrie)

Revue de code et Tests

- En matière de développement logiciel, la **revue de code** et les tests sont deux pratiques essentielles pour maintenir et améliorer la qualité du code :

1. **Revue de code** : Cette pratique implique que d'autres développeurs examinent le code pour vérifier la logique, la clarté, la qualité et la conformité avec les conventions de codage. Les revues de code peuvent aider à identifier les bugs, les erreurs de logique ou de conception et d'autres problèmes avant qu'ils ne deviennent trop coûteux ou difficiles à résoudre.
2. **Tests** : Les tests de logiciels sont un autre pilier de l'amélioration de la qualité. Les tests unitaires, les tests d'intégration, les tests de charge, les tests de sécurité et d'autres formes de tests peuvent aider à identifier les problèmes avant qu'ils ne soient découverts par les utilisateurs. L'utilisation de techniques telles que l'intégration continue et le déploiement continu peut également aider à garantir que les tests sont effectués régulièrement et automatiquement.

Importance de la qualité du code dans le développement logiciel

- La qualité du code est cruciale dans le développement logiciel pour de nombreuses raisons.
- Un code de haute qualité est plus facile à lire, à comprendre, à maintenir et à modifier. Cela facilite la détection des bugs, la mise en œuvre de nouvelles fonctionnalités, l'adaptation à l'évolution des exigences et l'extension du logiciel à mesure qu'il grandit et évolue. De plus, un code de qualité supérieure est souvent plus performant, plus sûr et plus fiable.
- **Facilité de maintenance** : Un code de haute qualité est plus facile à maintenir et à modifier. Cela est particulièrement important à mesure que le logiciel évolue et que de nouvelles fonctionnalités sont ajoutées ou que des modifications sont apportées.
    - Exemple en Java Spring : Si vous avez une application Spring Boot bien structurée, où le code est divisé en services modulaires, chaque service pouvant être développé, déployé et mis à jour indépendamment des autres, cela facilite la maintenance. Par ailleurs, l'utilisation d'une architecture propre et d'un codage cohérent améliore la lisibilité et facilite la maintenance.

Importance de la qualité du code dans le développement logiciel

- **Détection des bugs** : Un code bien organisé et cohérent rend plus facile la détection des bugs. Cela est d'autant plus vrai lorsque le code est accompagné de tests unitaires et d'intégration bien conçus qui peuvent être exécutés automatiquement.
  - Exemple en Java Spring : Dans Spring Boot, l'utilisation de frameworks de test comme JUnit avec Mockito facilite la rédaction de tests unitaires et d'intégration. Ces tests permettent de détecter rapidement les bugs avant qu'ils ne deviennent un problème en production.
- **Performances** : Un code de haute qualité est souvent plus performant. En évitant les goulots d'étranglement, en optimisant les algorithmes et en utilisant les ressources de manière efficace, un code de qualité supérieure peut souvent améliorer les performances d'une application.
  - Exemple en Java Spring : Si vous utilisez Spring Data JPA pour gérer l'accès aux données dans votre application Spring Boot, une utilisation efficace des requêtes personnalisées, des requêtes nommées, des spécifications ou des critères peut aider à améliorer les performances en minimisant le nombre de requêtes SQL et en récupérant uniquement les données nécessaires.

Importance de la qualité du code dans le développement logiciel

- **Sécurité** : Un code de haute qualité est souvent plus sûr. En évitant les vulnérabilités courantes, en utilisant des pratiques de codage sécurisées et en prenant en compte la sécurité dès le début du développement, vous pouvez rendre votre code plus résistant aux attaques.
  - Exemple en Java Spring : Dans Spring Boot, l'utilisation de Spring Security peut aider à sécuriser votre application en fournissant des fonctionnalités pour l'authentification, l'autorisation, la protection contre les attaques CSRF, etc. En suivant les meilleures pratiques de codage sécurisé et en tenant compte de la sécurité dès le début, vous pouvez rendre votre code plus sûr.
- **Fiabilité** : Enfin, un code de haute qualité est souvent plus fiable. En évitant les erreurs de codage courantes, en gérant correctement les exceptions et en utilisant des pratiques de codage robustes, vous pouvez rendre votre code plus résistant aux erreurs et aux défaillances.
  - Exemple en Java Spring : En gérant correctement les exceptions dans votre application Spring Boot, en utilisant par exemple le contrôleur de conseils (@ControllerAdvice) pour gérer les exceptions de manière centralisée, vous pouvez rendre votre application plus robuste et plus fiable. De plus, l'utilisation de stratégies de reprise après erreur, telles que les réessaies et les circuits ouverts avec Spring Retry et Hystrix, peut aider à améliorer la fiabilité de votre application.

Métriques dans un environnement Java

Définition et importance des métriques de qualimétrie

- **Définition** : Les métriques de qualimétrie sont des indicateurs quantitatifs utilisés pour mesurer la qualité d'un produit ou d'un service, dans notre cas, la qualité du code.
- **Importance** : Les métriques de qualimétrie sont essentielles pour identifier les zones du code qui nécessitent des améliorations, pour garantir la maintenabilité, la fiabilité, la testabilité, et l'efficacité du code.

Présentation de certaines métriques de qualimétrie clés

- **Complexité cyclomatique** : Comme mentionné précédemment, cette métrique mesure **le nombre de chemins d'exécution à travers le code**. Une valeur élevée peut indiquer que le code est trop complexe et difficile à maintenir ou à tester.
- **LOC (Lines of Code)** : Mesure **le nombre de lignes dans votre code**. C'est une métrique simple, mais elle peut donner une idée de la taille du projet.
- **Couplage** : Il existe plusieurs métriques de couplage, mais en général, elles mesurent **à quel point les classes dépendent les unes des autres**. Un couplage élevé peut rendre le code difficile à modifier et à tester.
- **Cohésion** : Mesure **à quel point les responsabilités d'une classe sont liées**. Une classe doit avoir une seule responsabilité, et donc une cohésion élevée.
- **Profondeur de l'héritage** : Mesure **le nombre de niveaux de la hiérarchie d'héritage d'une classe**. Une profondeur élevée peut rendre le code plus difficile à comprendre et à maintenir.

Exercice Complexité cyclomatique

- Supposons que vous développiez une application de gestion des tâches où vous avez une classe TaskService qui contient une méthode calculatePriority() pour calculer la priorité d'une tâche en fonction de certains critères. Cependant, cette méthode a une complexité cyclomatique élevée, ce qui peut rendre le code difficile à comprendre et à maintenir.

1. Analysez la méthode calculatePriority() pour identifier les points où la complexité cyclomatique est élevée.
2. Réduisez la complexité cyclomatique en refactorisant le code pour améliorer sa lisibilité et sa maintenabilité.

Exercice Complexité cyclomatique

```java
@Service
public class TaskService {
    // Méthode d'origine avec une complexité cyclomatique élevée
    public int calculatePriority(Task task) {
        int priority = 0;
        if (task.isUrgent()) {
            if (task.getDeadline().isBefore(LocalDate.now())) {
                priority = 10;
            } else {
                priority = 5;
            }
        } else {
            if (task.getCategory().equals("High")) {
                priority = 8;
            } else if (task.getCategory().equals("Medium")) {
                priority = 5;
            } else {
                priority = 2;
            }
        }
        return priority;
    }
}
```

```java
@Entity
public class Task {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private LocalDate deadline;
    private boolean isUrgent;
    private String category;

    // Getters and Setters
}

@RestController
@RequestMapping("/tasks")
public class TaskController {
    @Autowired
    private TaskService taskService;

    @PostMapping("/priority")
    public int calculateTaskPriority(@RequestBody Task task) {
        // Appel de la méthode refactorisée
        return taskService.calculatePriorityRefactored(task);
    }
}
```

Exercice LOC

- On souhaite réduire la LOC de la méthode createOrder du code suivant, proposez une nouvelle méthode :

```
@RestController
public class OrderController {
    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private CustomerRepository customerRepository;

    @PostMapping("/orders")
    public ResponseEntity<Order> createOrder(@RequestBody Order order, @RequestParam Long customerId) {
        Optional<Customer> customerOptional = customerRepository.findById(customerId);
        if (customerOptional.isPresent()) {
            Customer customer = customerOptional.get();
            order.setCustomer(customer);
            order.setDate(new Date());
            order.setStatus("CREATED");
            Order savedOrder = orderRepository.save(order);
            return ResponseEntity.ok(savedOrder);
        } else {
            return ResponseEntity.notFound().build();
        }
    }
}
```

Exercice Couplage

- Prenons l'exemple d'un contrôleur BookController qui utilise un service BookService. Le service lui-même utilise un dépôt BookRepository.

```
@RestController
public class BookController {
    @Autowired
    private BookService bookService;

    @GetMapping("/books")
    public List<Book> getAllBooks() {
        return bookService.findAllBooks();
    }
}
@Service
public class BookService {
    @Autowired
    private BookRepository bookRepository;

    public List<Book> findAllBooks() {
        return bookRepository.findAll();
    }
}
public interface BookRepository extends JpaRepository<Book, Long> {}
```

Exercice Cohésion

- Supposons que vous ayez une classe CustomerService qui est responsable de la gestion des clients dans votre application, mais aussi du calcul de certains indicateurs financiers

```
@Service
public class CustomerService {
    @Autowired
    private CustomerRepository customerRepository;

    public Customer createCustomer(Customer customer) {
        // Logique de création de client
        return customerRepository.save(customer);
    }

    public double calculateAverageRevenue(List<Customer> customers) {
        // Logique pour calculer le revenu moyen à partir d'une liste de clients
    }

    public double calculateProfitMargin(List<Customer> customers) {
        // Logique pour calculer la marge bénéficiaire à partir d'une liste de clients
    }
}
```

Exercice Profondeur d'héritage

Dans un chenil, il y a des animaux. Actuellement, le code utilise une hiérarchie d'héritage profonde pour représenter les différents types d'animaux. Voici la hiérarchie existante, à 4 niveaux.

```
Animal (nom, age)
   |
   -> Mammifere
        |
        -> Chien
            |
            -> Doberman
            |
            -> Labrador
        -> Chat
```

Simplifiez la hiérarchie pour réduire la profondeur maximale à 2 niveaux tout en gardant la possibilité d'avoir des spécificités différentes pour chaque type d'animal.

Outils de qualimétrie dans Java

Introduction frameworks et outils de qualimètrie.

La **qualimétrie logicielle** est le processus d'analyse et d'amélioration de la qualité d'un logiciel.

Elle inclut la **mesure** de divers attributs du logiciel, tels que sa **fiabilité**, sa **maintenabilité**, sa **complexité**, etc.

Dans l'écosystème Java, plusieurs outils et frameworks sont disponibles pour aider à ce processus.

- **SonarQube** : C'est un outil de qualimétrie de code populaire qui prend en charge de nombreux langages, y compris Java et C#. SonarQube peut analyser le code pour détecter les bugs, code smells, les vulnérabilités de sécurité, et fournir une couverture de code. Il peut également mesurer la dette technique, qui est une estimation du temps nécessaire pour corriger tous les problèmes de code identifiés.
- **Checkstyle** : Checkstyle est un outil de développement qui aide les programmeurs à écrire du code Java qui adhère à un ensemble de règles de codage. Il automatisera le processus de vérification du code Java pour le rendre plus conforme à certaines conventions de codage.

Introduction frameworks et outils de qualimètrie.

- **PMD (Programming Mistake Detector)** : C'est un autre outil d'analyse de code statique qui peut détecter les bugs potentiels, les mauvaises pratiques de codage, les expressions compliquées ou inutiles, les duplications de code, etc.
- **FindBugs/SpotBugs** : C'est un outil d'analyse statique de bytecode Java qui détecte les bugs potentiels dans le code. SpotBugs est le successeur de FindBugs, offrant des fonctionnalités similaires.
- **JaCoCo (Java Code Coverage)** : JaCoCo est un outil de couverture de code qui identifie les parties de votre code qui ne sont pas testées. Il est généralement utilisé en conjonction avec des outils de test unitaire comme JUnit.
- **JUnit** : Bien que ce soit un framework de test unitaire, JUnit joue également un rôle important dans la qualimétrie en permettant aux développeurs d'écrire et d'exécuter des tests pour vérifier que le code se comporte comme prévu

SonarQube

SonarQube est un outil d'analyse statique de code largement utilisé pour améliorer la qualité du code en identifiant les problèmes potentiels.

1. **Analyse de Code** : SonarQube peut analyser le code source pour une grande variété de langages de programmation (Java, JavaScript, C#, Python, etc.) pour détecter les problèmes de qualité.
2. **Détection des Bugs et Vulnérabilités** : Il est capable de détecter une variété de problèmes de qualité, y compris les bugs potentiels et les vulnérabilités de sécurité.
3. **Détection des "Code Smells" (Odeurs de Code)** : SonarQube peut identifier les "code smells", qui sont des caractéristiques du code qui indiquent une mauvaise conception. Les "code smells" peuvent rendre le code plus difficile à comprendre et à maintenir.
4. **Mesure de la Couverture de Tests** : En se connectant à des outils de couverture de code comme JaCoCo pour Java ou dotCover pour C#, SonarQube peut mesurer le pourcentage de votre code qui est couvert par des tests.
5. **Calcul de la Dette Technique** : SonarQube estime le temps qu'il faudrait pour corriger tous les problèmes de qualité qu'il a détectés, une mesure appelée "dette technique". Il affiche également une note de maintenabilité sur une échelle de A à E pour aider à comprendre rapidement la qualité du code.

SonarQube

6. **Duplication de Code** : SonarQube peut détecter les parties du code qui sont dupliquées, ce qui peut indiquer une mauvaise conception ou un risque accru d'erreurs.
7. **Analyse de l'Historique du Code** : SonarQube peut analyser l'historique du code pour identifier les tendances dans la qualité du code au fil du temps.
8. **Rapports et Tableaux de Bord** : SonarQube fournit des rapports détaillés et des tableaux de bord qui peuvent être utilisés pour surveiller la qualité du code à différents niveaux, de l'ensemble de l'entreprise jusqu'au module individuel.
9. **Intégration Continue/Déploiement Continu (CI/CD)** : SonarQube s'intègre facilement dans les pipelines CI/CD, ce qui permet d'effectuer une analyse de la qualité du code à chaque commit ou avant chaque déploiement.
10. **Règles et Profils de Qualité** : SonarQube fournit un ensemble de règles par défaut pour chaque langage qu'il supporte, et les utilisateurs peuvent également définir leurs propres règles ou modifier les règles existantes. Ces règles peuvent être regroupées en "profils de qualité" qui peuvent être appliqués à un ou plusieurs projets.

Checkstyle

Checkstyle est un outil de développement qui aide à garantir que votre code Java respecte certaines conventions de codage. Il est très configurable et peut être ajusté pour correspondre à vos propres conventions de codage.

- **Vérification du Style de Code** : Checkstyle peut vérifier que votre code adhère à une variété de conventions de style, comme les règles de formatage, le nommage des variables, l'utilisation des accolades, etc.
- **Vérification de la Conception** : Checkstyle peut détecter les mauvaises pratiques de conception, comme les classes avec trop de responsabilités, les dépendances cycliques, etc.
- **Vérification des commentaires** : Checkstyle peut vérifier la présence et le format des commentaires de documentation Javadoc.
- **Vérification des Importations** : Checkstyle peut vérifier que votre code n'utilise que les importations nécessaires et qu'il ne contient pas d'importations inutilisées.

Checkstyle.

1. Installation du plugin (maven, gradle,...)
2. Executer le check
3. Consulter le rapport généré en format html, target/site/checkstyle.html

PMD.

PMD est un outil d'analyse de code source pour les langages de programmation tels que Java, JavaScript, XML, et plus encore. Il examine le code pour détecter les mauvaises pratiques potentielles comme les variables non utilisées, les blocs catch vides, les classes inutiles, et plus encore.

1. **Détection de bugs potentiels** : PMD peut identifier les erreurs de programmation courantes comme les variables non initialisées, les exceptions vides, etc.
2. **Amélioration de la lisibilité du code** : Il détecte les "code smells", qui sont des indices de problèmes potentiels dans le code qui peuvent rendre le code plus difficile à lire et à maintenir.
3. **Performance** : PMD détecte les problèmes de performance courants, comme l'utilisation inutile d'objets String ou les boucles inutiles.
4. **Sécurité** : Il détecte les problèmes de sécurité comme l'utilisation de hard-coding, les exceptions silencieuses, etc.

PMD.

1. Installation du plugin (maven, gradle,...)
2. Executer le check
3. Consulter le rapport généré en format html, target/site/pmd.html

FindBugs.

FindBugs est un outil d'analyse statique de code pour Java qui détecte les erreurs de programmation potentielles. Il utilise l'analyse de bytecode, ce qui signifie qu'il fonctionne sur les fichiers compilés (.class) et non sur le code source directement. FindBugs est capable de trouver une variété de problèmes de qualité de code, y compris :

1. Les erreurs de nullité (par exemple, les références null potentiellement déréférencées).
2. Les violations de la précision des points flottants.
3. Les problèmes de performance, tels que les objets String mal utilisés.
4. Les problèmes de sécurité, comme les variables de classe exposées publiquement.
5. Les mauvaises pratiques de codage, telles que les instructions switch sans default.

JaCoCo.

JaCoCo est un outil populaire de couverture de code pour Java. Il vous permet de voir quelles parties de votre code sont couvertes par vos tests unitaires et quelles parties ne le sont pas. Cela peut être très utile pour identifier les zones de votre code qui pourraient nécessiter des tests supplémentaires.

- Les principales fonctionnalités de JaCoCo comprennent :

1. **Couverture d'instructions** : JaCoCo peut indiquer le nombre d'instructions Java bytecode qui ont été exécutées par vos tests.
2. **Couverture de branches** : JaCoCo peut montrer la couverture des branches de vos instructions if et switch.
3. **Couverture de lignes** : JaCoCo peut indiquer quelles lignes de votre code ont été exécutées par vos tests.

Intégration des outils de qualimétrie dans une PIC.

PIC = plateforme d'intégration continue

L'intégration de ces outils d'analyse de code statique et de couverture de code dans un système d'intégration continue (Continuous Integration, CI) peut grandement améliorer la qualité du code, en aidant à identifier et à résoudre les problèmes tôt dans le cycle de développement.

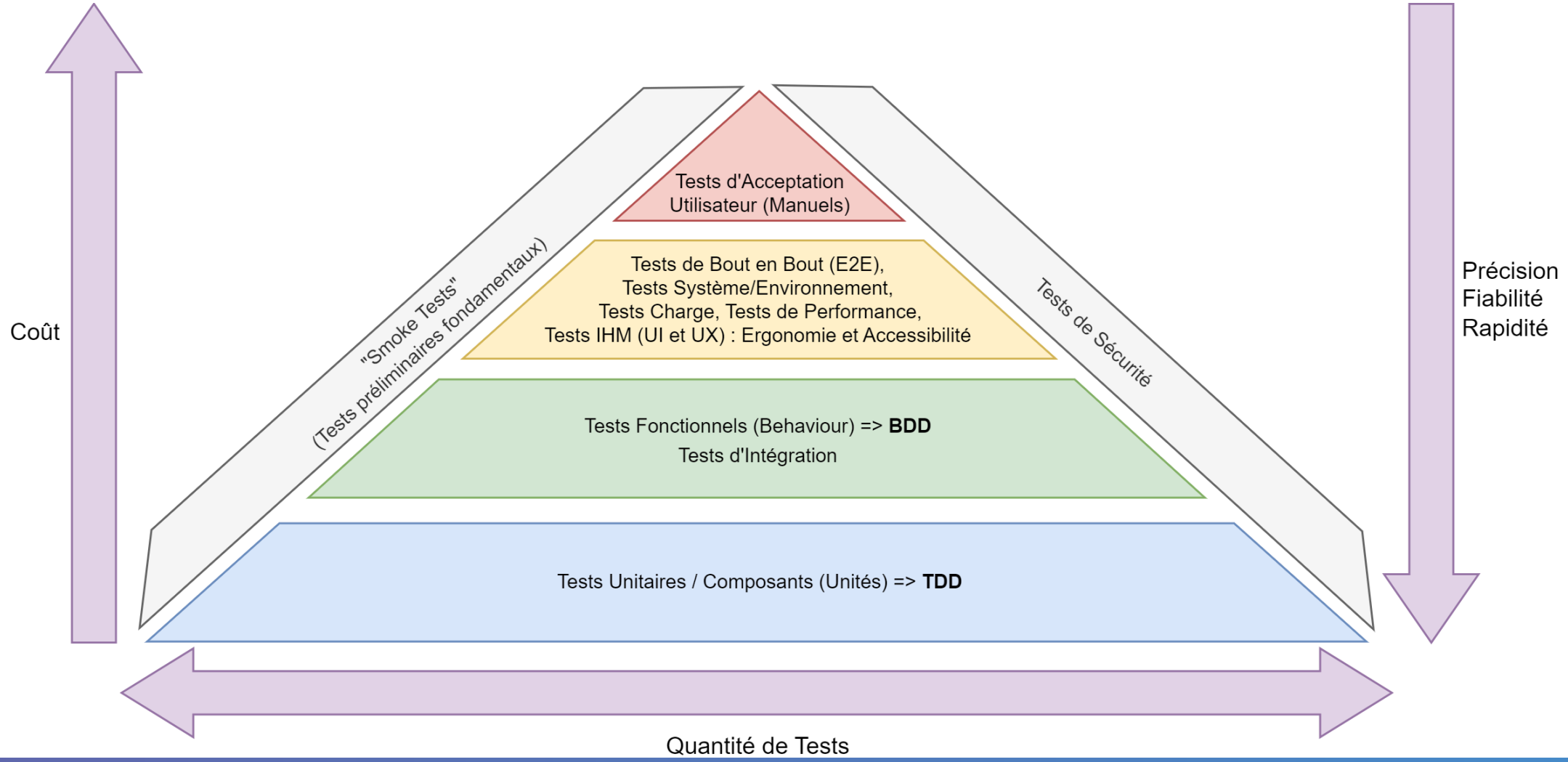
1. SonarQube :
    - SonarQube peut être intégré à de nombreux serveurs CI, comme Jenkins, GitLab CI/CD et GitHub Actions. Vous pouvez ajouter une étape dans votre pipeline CI pour exécuter l'analyse SonarQube. Dans le cas de Jenkins, vous pouvez utiliser le plugin Jenkins SonarQube pour automatiser cette tâche.
2. Checkstyle, PMD, FindBugs,JaCoCo :
    - Ces outils peuvent être exécutés comme une partie de votre build Maven. Vous pouvez ajouter une étape dans votre pipeline CI pour exécuter mvn clean verify (ou une autre commande Maven qui exécute les plugins correspondants). Les rapports générés par ces outils peuvent ensuite être recueillis et affichés par votre serveur CI.

Atelier

Mise en place et utilisation de SonarQube en local avec Docker et un projet au choix.

Tests

Pyramide des tests



Exemple d'outils de tests en Java

Outil	Type de tests
JUnit	Tests Fonctionnels Tests d'Intégration Tests Unitaires
Selenium PostMan	Tests End-to-End
Selenium Jest	Tests IHM (UI et UX) : Ergonomie et Accessibilité
JMeter	Tests Charge Tests de Performance

Clean Test

Le génie logiciel est une question de savoir-faire où nous devons écrire toutes les parties du logiciel avec le même soin, qu'il s'agisse de code de production ou de test.

- Rédiger des tests fait partie de notre savoir-faire.
  Nous ne pouvons avoir un code propre que si nous avons des tests propres.
- Un test propre se lit comme une histoire.
- Un test propre doit contenir toutes les informations nécessaires pour comprendre ce qui est testé

Clean Test

Le génie logiciel est une question de savoir-faire où nous devons écrire toutes les parties du logiciel avec le même soin, qu'il s'agisse de code de production ou de test.

- Rédiger des tests fait partie de notre savoir-faire.
  Nous ne pouvons avoir un code propre que si nous avons des tests propres.
- Un test propre se lit comme une histoire.
- Un test propre doit contenir toutes les informations nécessaires pour comprendre ce qui est testé

Écriture d'un test clean

- Le nom d'un test doit révéler le cas de test exact, y compris le système testé.
- Il doit spécifier l'exigence du cas de test aussi précisément que possible.
- L'objectif principal d'un bon nom de test est que si un test échoue, nous devrions être en mesure de récupérer la fonctionnalité cassée à partir du
  nom du test.
- Les conventions de dénomination populaires :
  - ShouldWhen : Should_ExpectedBehavior_When_StateUnderTest
    (ex : ShouldHaveUserLoggedIn_whenUserLogsIn)
  - MethodName_StateUnderTest_ExpectedBehavior
    (ex : isAdult_AgeLessThan18_False)
  - testFeatureBeingTested
    (ex : testIsNotAnAdultIfAgeLessThan18)
  - GivenWhenThen (BDD): Given_Preconditions_When_StateUnderTest_Then_ExpectedBehavior
    (ex : GivenUserIsNotLoggedIn_whenUserLogsIn_thenUserIsLoggedInSuccessfully)

Écriture d'un test clean - AAA

Le modèle **Arrange-Act-Assert** est une manière descriptive et révélatrice d'intention de structurer des cas de test. Il prescrit un ordre des opérations:

- La section **Arrange** doit contenir la logique de configuration des tests. Ici, les objets sont initialisés et préparés pour l'exécution des tests.
- La section **Act** invoque le système que nous sommes sur le point de tester. Il peut s'agir par exemple d'appeler une fonction, d'appeler une API REST ou d'interagir avec certains composants.
- La section **Assert** vérifie que l'action du test se comporte comme prévu. Par exemple, nous vérifions ici la valeur de retour d'une méthode, l'état final du test , les méthodes que le test a appelées, ou les éventuelles exceptions attendues et les résultats d'erreur

Développement en TDD

Rappel des frameworks java pour les Test

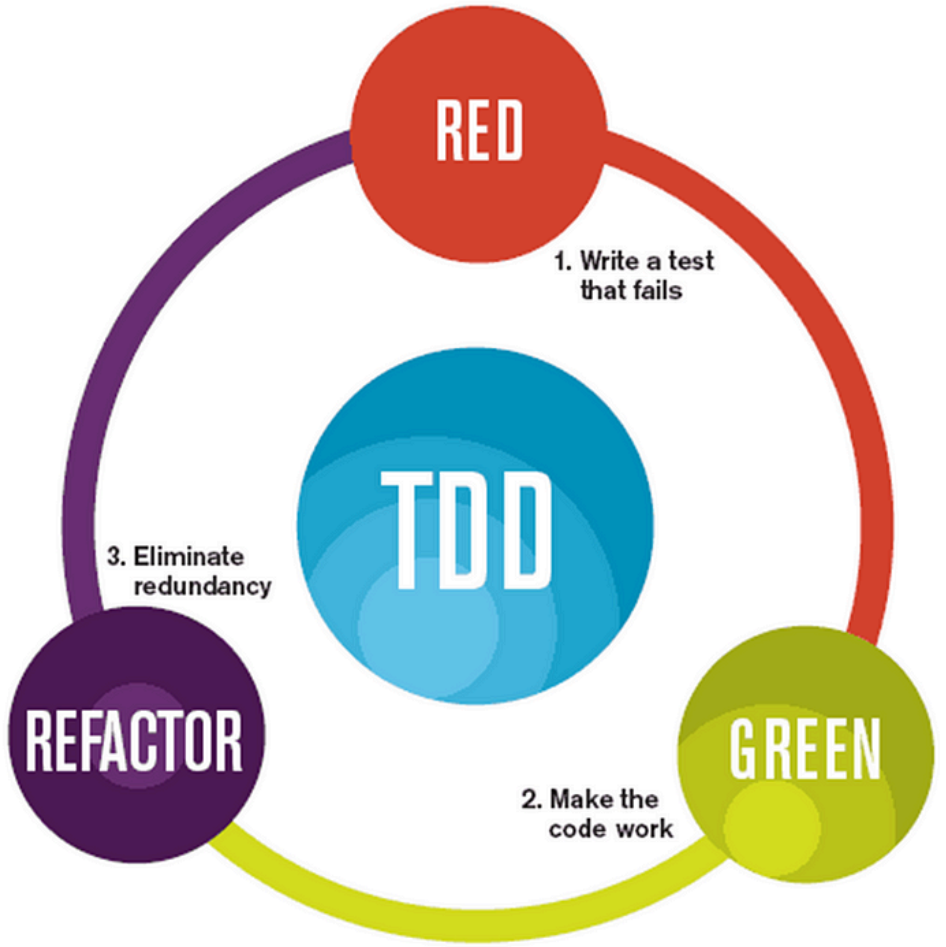
En Java, plusieurs frameworks sont disponibles pour vous aider à créer et à gérer des tests unitaires, d'intégration et fonctionnels.

- **JUnit** : C'est le framework de test le plus populaire pour Java. Il est utile pour les tests unitaires et peut être intégré à des outils d'automatisation de construction comme Maven et Gradle.
- **Mockito** : Il s'agit d'un framework de simulation (mocking) populaire en Java. Mockito est souvent utilisé en combinaison avec JUnit. Il vous permet de créer et de gérer des objets fictifs (mocks) pour simuler le comportement de classes et d'interfaces complexes.

Les paradigmes du TDD

Le **Test Driven Development (TDD)**, ou Développement Dirigé par les Tests, est une méthode de développement de logiciel qui encourage l'écriture de tests avant l'écriture du code de production. En Java, spécifiquement avec le framework Spring, le TDD suit généralement ce processus :

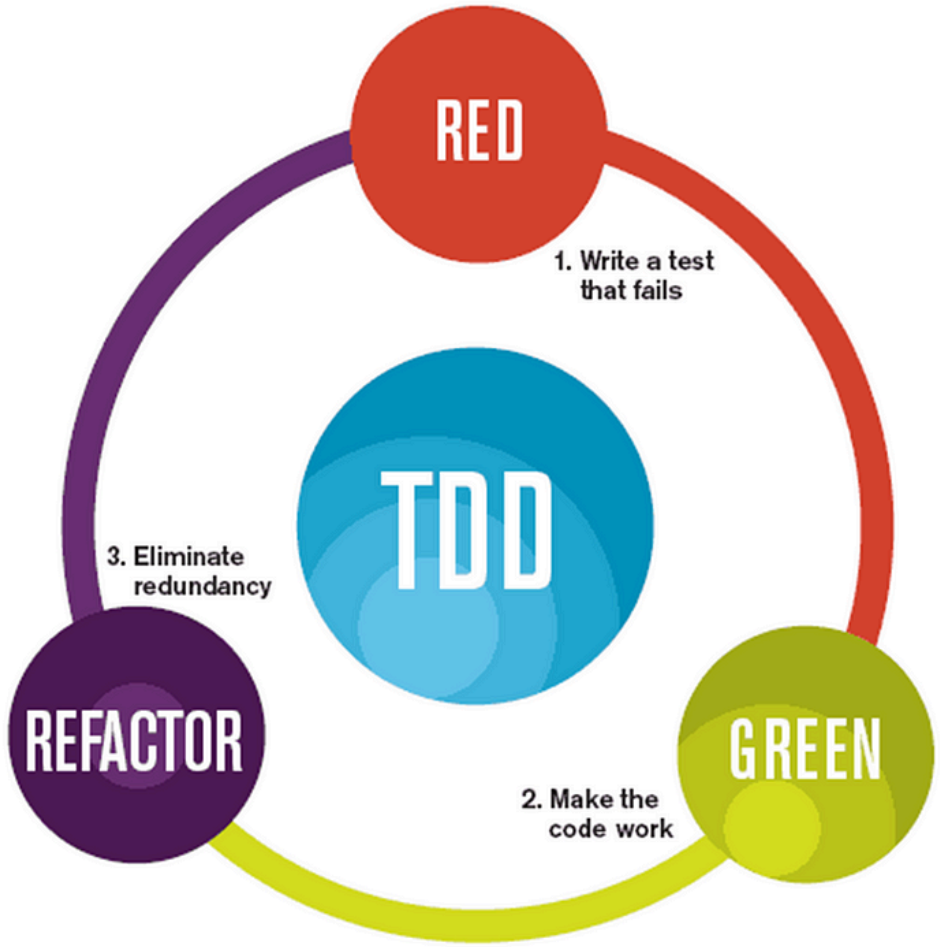
- **Écrire un test** : Vous commencez par écrire un test pour la nouvelle fonctionnalité que vous voulez implémenter. À ce stade, le test échouera, car vous n'avez pas encore écrit le code de production.
- **Exécuter tous les tests et voir si le nouveau test échoue** : Ceci est fait pour s'assurer que le nouveau test ne passe pas par accident. C'est une étape importante pour valider que le test est bien écrit et teste la bonne chose.



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

Les paradigmes du TDD

- **Écrire le code de production** : Vous écrivez maintenant le code qui fera passer le test. À ce stade, vous ne vous concentrez que sur le fait de faire passer le test et non sur la propreté ou l'efficacité du code.
- **Exécuter les tests** : Vous exécutez maintenant tous les tests pour vous assurer que le nouveau code de production passe le nouveau test et que tous les autres tests passent toujours.
- **Refactoriser le code** : Si les tests passent, vous pouvez alors refactoriser le code de production pour l'améliorer tout en gardant les mêmes fonctionnalités. L'objectif est d'améliorer la structure du code sans changer son comportement. Après la refactorisation, vous exécutez à nouveau les tests pour vous assurer qu'ils passent toujours.



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

Les bonnes pratiques du TDD

- **Comprendre les exigences** : Avant d'écrire vos tests, vous devez avoir une compréhension claire de ce que le code doit réaliser. Cela comprend la compréhension des exigences fonctionnelles et non fonctionnelles.
- **Garder les tests simples** : Chaque test doit être indépendant et tester une seule fonctionnalité ou un seul comportement. Cela signifie que vous devriez vous efforcer de garder vos tests aussi simples et spécifiques que possible.
- **Ne pas anticiper les fonctionnalités futures** : Lorsque vous écrivez vos tests, concentrez-vous sur la fonctionnalité actuellement requise, pas sur ce que vous pensez qu'elle pourrait être requise à l'avenir. C'est ce qu'on appelle le principe YAGNI (You Aren't Gonna Need It).
- **Écrire le code juste nécessaire pour passer le test**: Après avoir écrit un test, écrivez le code de production minimum nécessaire pour passer le test. Ne vous inquiétez pas de la perfection à ce stade, il suffit que le code passe le test.

Les bonnes pratiques du TDD

- **Refactorisation régulière** : Une fois que votre code passe les tests, prenez le temps de le refactoriser. Cela signifie le rendre plus lisible, le simplifier, éliminer les duplications, etc. N'oubliez pas de réexécuter vos tests après chaque refactorisation pour vous assurer que rien n'a été brisé.
- **Faire des tests automatisés** : Les tests doivent être automatisés afin qu'ils puissent être exécutés à chaque modification du code. Cela vous aidera à attraper les bugs tôt et facilitera l'intégration continue.
- **Mise en place des doubles de test (test doubles)** : Utilisez des bouchons (stubs), des mocks et des objets factices (dummy objects) pour isoler le code que vous testez. Cela vous permet de concentrer vos tests sur le code que vous écrivez, plutôt que sur ses dépendances.
- **Exécuter les tests régulièrement** : Les tests doivent être exécutés régulièrement, idéalement à chaque changement du code, pour s'assurer que tout fonctionne toujours comme prévu.

Les bonnes pratiques du TDD

- **Maintenir une bonne couverture de test** : Toutes les parties du code doivent être testées. Vous devriez viser une haute couverture de code par les tests, bien que 100% ne soit pas toujours réaliste ou nécessaire.
- **Tester aux différents niveaux** : Il ne suffit pas de faire des tests unitaires, pensez également aux tests d'intégration, aux tests système, aux tests d'acceptation, etc

Principe FIRST

F - **Fast (Rapide)** : Les tests doivent être rapides, car cela encourage les développeurs à les exécuter fréquemment, ce qui facilite la détection précoce des problèmes.
I - **Independent (Indépendant)** : Les tests doivent être indépendants les uns des autres, ce qui signifie qu'ils ne doivent pas avoir de dépendances mutuelles. Cela permet d'isoler les problèmes plus facilement et d'améliorer la maintenance.
R - **Repeatable (Reproductible)** : Les tests doivent être reproductibles dans n'importe quel environnement. Ils ne doivent pas dépendre de conditions externes instables ou d'états changeants, afin de garantir des résultats cohérents.
S - **Self-validating (Auto-vérifiable)** : Les tests doivent s'auto-vérifier et retourner un résultat clair (passé/échoué) sans nécessiter de vérification manuelle. Cela facilite l'intégration des tests dans les processus d'intégration continue et d'automatisation.
T - **Timely (Opportun)** : Les tests doivent être écrits en même temps que le code, voire avant. Les retards dans l'écriture des tests peuvent entraîner des problèmes de qualité et de maintenabilité.

TP

- On souhaite développer une classe Frame, qui représente une Frame dans le jeu du bowling, en utilisant les TDD.
- Les tests pour réaliser la classe Frame du jeu de bowling doivent couvrir les scénarios suivants:
  - S'il s'agit d'une série standard (round 1 par exemple)
  - Le premier lancer d'une série doit augmenter le score de la série
  - Le second lancer d'une série doit augmenter le score de cette série
  - En cas de strike, il ne doit pas être possible de lancer de nouveau au cours de cette même série.
  - En cas de lancers standards, il ne doit pas être possible de lancer plus de 2 fois

TP

- S’il s’agit d’une série finale (dernier round)
  - En cas de strike, il doit être possible de lancer une nouvelle fois au cours d’une série
  - En cas de strike puis de lancer, le score est censé augmenter en accord avec le résultat du lancer.
  - En cas de strike puis d’un lancer, il doit être possible de lancer une nouvelle fois
  - En cas de strike puis de lancer, le score est censé augmenter en accord avec le résultat
  - En cas de spare, il doit être possible de lancer une nouvelle fois au cours d’une série
  - En cas de spare puis de lancer, le score est censé augmenter en accord avec le résultat du lancer.
  - En cas de lancers standards, il ne doit pas être possible de lancer plus de 4 fois

Développement en BDD

Les paradigmes du BDD

Le **Behavior Driven Development (BDD)** est une pratique de développement de logiciel qui met l'accent sur la **collaboration entre les différentes parties prenantes d'un projet** (comme les développeurs, les testeurs, les responsables produit, etc.) et l'**explication du comportement du système en termes compréhensibles par tous**. Il est souvent utilisé dans le développement Agile

- **Communication et collaboration** : Le BDD insiste sur la nécessité d'une collaboration étroite entre toutes les parties prenantes du projet. Il met l'accent sur le "partage des connaissances", en veillant à ce que tout le monde comprenne le comportement désiré du système.
- **Développement basé sur le comportement** : Comme son nom l'indique, le BDD met l'accent sur **le comportement du système** plutôt que sur les détails techniques de son implémentation. Il s'agit de décrire **ce que le système doit faire de manière compréhensible par tous**, et non de se concentrer sur la manière dont ces fonctionnalités seront mises en œuvre.
- **Spécification exécutable** : Le BDD utilise un **langage naturel** pour **décrire le comportement du système**. Ces descriptions sont ensuite utilisées comme spécifications exécutables, qui peuvent être exécutées comme tests. Cela signifie que les spécifications servent à la fois de documentation et de vérification du système.

Les paradigmes du BDD

- **Tests orientés comportement** : Le BDD utilise un format spécifique pour les tests, connu sous le nom de "**Given-When-Then**" (Étant donné - Quand - Alors). Cela décrit le contexte (Given), l'action qui est effectuée (When), et le résultat attendu (Then). Cela aide à structurer les tests de manière à ce qu'ils reflètent le comportement désiré du système.
- **Développement itératif** : Comme d'autres pratiques Agile, le BDD suit **une approche itérative du développement**. Il s'agit de construire progressivement le système, en ajoutant un comportement à la fois, et en vérifiant constamment que le système se comporte comme prévu.

Les bonnes pratiques du BDD

- **Définissez clairement les comportements** : Vos spécifications **devraient décrire clairement le comportement attendu du système**. Elles ne devraient pas se concentrer sur les détails techniques de l'implémentation, mais plutôt sur **ce que l'utilisateur peut faire et ce qu'il peut s'attendre à voir**.
- **Utilisez le format Given-When-Then** : Ce format est un excellent **moyen de structurer vos scénarios de manière claire et compréhensible**. Il vous aide à vous concentrer sur le contexte (Given), l'action (When) et le résultat (Then). Il est compatible avec le français (**Étant donné-Quand-Alors**)
- **Gardez vos scénarios courts et concentrés** : Chaque scénario doit **tester une seule fonctionnalité ou comportement**. S'il y a trop de choses dans un seul scénario, il devient difficile à comprendre et à maintenir.
- **Automatisez vos scénarios** : Les scénarios BDD doivent **être automatisés pour pouvoir les exécuter régulièrement**. Cela vous permet de vérifier rapidement que votre système se comporte toujours comme prévu, même après des modifications.
- **Revoyez et affinez vos scénarios régulièrement** : Comme tout autre aspect de votre système, vos scénarios BDD devraient être revus et affinés régulièrement. Cela vous aide à maintenir leur pertinence et leur utilité.

Cucumber

Cucumber : Cucumber est l'un des frameworks les plus populaires pour le BDD. Il vous permet d'**écrire des scénarios de tests en langage naturel** qui peuvent être **exécutés comme des tests automatisés**. Cucumber dispose d'une **intégration étroite avec Spring**, ce qui vous permet d'utiliser des fonctionnalités comme l'injection de dépendances dans vos tests.

Gherkin

Gherkin est un langage de spécification utilisé dans le BDD pour écrire des scénarios de tests de manière lisible par les humains.
Il utilise une syntaxe simple et naturelle pour décrire les comportements attendus sous forme de "Features" (fonctionnalités) et de "Scenarios" (scénarios).

Mot Anglais	Mot Français	Description
Feature	**Fonctionnalité**	Description de la fonctionnalité à tester.
Scenario	**Scénario**	Exemple concret illustrant une fonctionnalité.
Given	**Étant donné que**	Contexte initial du scénario.
When	**Quand**	Action ou événement déclencheur.
Then	**Alors**	Résultat attendu après l'action.
And/But	**Et/Mais**	Étapes supplémentaires dans le scénario.

Gherkin

```gherkin
Feature: Jeu du Pendu

  Scenario: Proposition correcte d'une lettre
    Given le mot à deviner est "chat"
    And l'état actuel du mot est "_ _ _ _"
    When le joueur propose la lettre "a"
    Then l'état actuel du mot doit être "_ _ a _"
    And le nombre de tentatives restantes est inchangé

  Scenario: Proposition incorrecte d'une lettre
    Given le mot à deviner est "chat"
    And l'état actuel du mot est "_ _ _ _"
    When le joueur propose la lettre "z"
    Then l'état actuel du mot doit rester "_ _ _ _"
    And le nombre de tentatives restantes doit être diminué de 1

  Scenario: Le joueur gagne en devinant toutes les lettres
    Given le mot à deviner est "chat"
    And l'état actuel du mot est "c h a _"
    When le joueur propose la lettre "t"
    Then l'état actuel du mot doit être "c h a t"
    And le joueur doit voir un message de victoire

  Scenario: Le joueur perd après avoir épuisé toutes les tentatives
    Given le mot à deviner est "chat"
    And le nombre de tentatives restantes est 1
    When le joueur propose la lettre "z"
    Then le joueur doit voir un message de défaite
    And le mot complet "chat" doit être révélé
```

```gherkin
Fonctionnalité: Jeu du Pendu

  Scénario: Proposition correcte d'une lettre
    Étant donné que le mot à deviner est "chat"
    Et que l'état actuel du mot est "_ _ _ _"
    Quand le joueur propose la lettre "a"
    Alors l'état actuel du mot doit être "_ _ a _"
    Et le nombre de tentatives restantes est inchangé

  Scénario: Proposition incorrecte d'une lettre
    Étant donné que le mot à deviner est "chat"
    Et que l'état actuel du mot est "_ _ _ _"
    Quand le joueur propose la lettre "z"
    Alors l'état actuel du mot doit rester "_ _ _ _"
    Et le nombre de tentatives restantes doit être diminué de 1

  Scénario: Le joueur gagne en devinant toutes les lettres
    Étant donné que le mot à deviner est "chat"
    Et que l'état actuel du mot est "c h a _"
    Quand le joueur propose la lettre "t"
    Alors l'état actuel du mot doit être "c h a t"
    Et le joueur doit voir un message de victoire

  Scénario: Le joueur perd après avoir épuisé toutes les tentatives
    Étant donné que le mot à deviner est "chat"
    Et que le nombre de tentatives restantes est 1
    Quand le joueur propose la lettre "z"
    Alors le joueur doit voir un message de défaite
    Et le mot complet "chat" doit être révélé
```