UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Java clean code

Sommaire

Jour 1 : Philosophie du "Clean Code"

- "Clean Code" : Philosophie du code
- Evoquer les écrits de Robert C. Martin
- Bénéfice de "Clean Code" pour un développeur
- Bénéfices de "Clean Code" pour le logiciel
- Les bonnes raisons de coder « proprement »
- Pourquoi coder proprement ?
- Apports de "Clean Code"
- Comment identifier un code sale
- Quelle est votre méthode ?
- Les 7 péchés capitaux : Rigidité, Fragilité, etc.
- Quels péchés rencontrez-vous le plus ?
- Rappel des critères de qualité d'une application (ISO)
- Qui est responsable de la dette technique ?
- Les principes "Clean Code"
- "Clean Code" ça se pratique, qui a déjà fait un kata ?
- Procédé pour devenir "Clean Coder"
- Connaître les règles "Clean Code"
- Savoir tester
- Connaître les code smells
- Savoir refactorer

Sommaire

Jour 2 : Code design

- L'architecture logicielle
- Pourquoi a-t-on besoin d'une bonne architecture ?
- 4 notions importantes pour une application propre
  - Couplage faible,
  - Cohésion,
  - Changements locaux,
  - Nommage
- Connaître les principes de programmation objet SOLID
- Single Responsibility Principle,
- Open-closed principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle
- Comprendre la notion de couplage
- Héritage,
- Composition et
- Agrégation
- Les patrons de conceptions
- Origine
- Liste des patrons de conception (catégories de patrons + description)
- TP
- Les Values Objects
- Principes
- Immutable
- Effectuer des opérations en parallèle
- TP

Sommaire

Jour 3 : Test

- Principes du clean tests
- 1 Test = 1 Scénario
- Simple et indépendant
- Sans assertion inutile
- Bouchonné
- FIRST
- Connaître les différents types de tests
- Pyramide de tests
- Tests unitaires
- Tests d'intégration
- Tests d'acceptance
- Quelles sont les stratégies de tests chez Pôle emploi ?
- Comprendre la notion de couverture de code
- Couverture de code
- Couverture par instruction
- Couverture par chemin (d'algorithme par exemple)
- Les principaux outils (EclEmma, Cobertura, JaCoCo, etc.)
- Comment commencer ?
- Comment testez-vous une application existante ?
- Méthodologie de tests
- Comment identifier du code testable ?
- TP

Sommaire

Jour 4 : Refactoring et code smells

- Refactorer une application legacy
- Méthodologie (Tester, Refactorer, Apporter de la valeur)
- TP ?
- Savoir ce qu'on appelle un code smell
- C'est quoi un code smell ?
- Liste des code smells
- Connaître les catégories de code smells et les princpaux code smells associés
- Bloaters
- Long Method, Large Class,
- Etc.
- Object oriented abuser
- Switch Statements, Temporary Field,
- Etc.
- Change preventer
- Divergent change, Shotgun Surgery,
- Etc.
- Etc.
- Savoir refactorer les principaux code smells
- La loi de Demeter
- https://refactoring.guru/fr/refactoring/smells
- Design Pattern
- TP

Sommaire

Jour 5 : Outillage (+ ancrage), TP final

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Jour 1

"Clean Code" : Philosophie du code

- Le *Clean Code* est une approche de programmation qui met l'accent sur l'écriture d'un code clair, simple, et facile à maintenir. Popularisée par Robert C. Martin, également connu sous le nom d'Uncle Bob, cette philosophie repose sur l'idée que le code doit être compréhensible non seulement par son auteur, mais aussi par d'autres développeurs qui pourraient avoir à travailler dessus plus tard. En pratique, cela signifie créer des structures de code qui minimisent la complexité, évitent les duplications, et sont composées de petites unités faciles à tester. Un *Clean Code* est non seulement fonctionnel, mais aussi élégant et intuitif, permettant de résoudre des problèmes de manière efficace tout en garantissant la pérennité du projet.

Essentiellement, la philosophie du *Clean Code* incite les développeurs à considérer leur code non seulement comme un moyen d'atteindre un objectif technique, mais comme un **outil de communication** avec d'autres membres de l'équipe ou avec soi-même à l'avenir.

- La philosophie du Clean Code est basée sur une idée centrale : le code est lu bien plus souvent qu'il n'est écrit. Par conséquent, il doit être écrit de manière à être facilement compréhensible par tout autre développeur (ou par soi-même dans quelques mois). Un code "propre" ou Clean Code est un code qui adhère à des principes de simplicité, de lisibilité, et de maintenabilité, permettant ainsi à une équipe de développement de travailler de manière fluide et efficace à long terme.

"Clean Code" : Philosophie du code

Bénéfices du *Clean Code* pour un développeur

1. **Facilite la lecture et la compréhension du code** :
   Le premier avantage d'écrire du *Clean Code* pour un développeur est la lisibilité. Un code propre utilise des noms de variables, de méthodes et de classes qui reflètent clairement leur rôle. Cela permet au développeur (et à d'autres membres de l'équipe) de comprendre rapidement ce que fait le code sans avoir à le déchiffrer. Ainsi, les développeurs peuvent passer moins de temps à essayer de comprendre le code et plus de temps à l'améliorer.
2. **Réduit les erreurs et les bugs** :
   Un code propre est organisé et structuré de manière à minimiser les erreurs. En appliquant des principes comme le **Single Responsibility Principle** (une méthode ou classe ne fait qu'une seule chose), il est plus facile de tester chaque composant du système et de s'assurer qu'il fonctionne correctement. Cela permet également de réduire les bugs car chaque partie du code est isolée et indépendante.

"Clean Code" : Philosophie du code

3. **Facilite la maintenance** :
   Un autre avantage est que le *Clean Code* facilite grandement la maintenance. Lorsque le code est clair et bien structuré, il est plus facile d'y revenir plus tard pour l'améliorer ou corriger des problèmes. Si vous revenez sur un projet quelques mois plus tard, un code propre sera toujours facile à comprendre, réduisant le temps nécessaire pour se réacclimater.

4. **Améliore la collaboration en équipe** :
   Travailler en équipe devient beaucoup plus fluide lorsque tout le monde écrit et lit un *Clean Code*. Comme le code est plus facile à comprendre et à suivre, il est plus simple de prendre en main le travail d'un autre développeur sans avoir à passer trop de temps à comprendre ses intentions. Cela améliore la productivité globale de l'équipe.

"Clean Code" : Philosophie du code

Bénéfices du *Clean Code* pour le logiciel

1. **Améliore la maintenabilité du logiciel** :
   Un logiciel basé sur du *Clean Code* est plus facile à maintenir sur le long terme. Comme chaque partie du code est isolée et facile à comprendre, les développeurs peuvent y apporter des modifications sans risquer de casser d'autres parties du système. Cela permet de faire évoluer le logiciel de manière agile, en ajoutant de nouvelles fonctionnalités ou en corrigeant des bugs avec moins de risques.

   Exemple :
   Si vous avez un logiciel avec plusieurs fonctionnalités bien séparées dans des classes ou méthodes distinctes, il est facile de modifier une fonctionnalité sans affecter les autres parties du système. Cela est beaucoup plus difficile si le code est spaghetti et mal structuré.

2. **Améliore la performance à long terme** :
   Bien que le *Clean Code* ne garantisse pas directement des performances accrues, il facilite l'optimisation du logiciel à long terme. Un code bien structuré permet d'identifier rapidement les goulots d'étranglement et les parties à améliorer. Par exemple, vous pouvez facilement optimiser une méthode isolée si elle est bien séparée du reste du système.

"Clean Code" : Philosophie du code

3. **Réduit les risques d'introduction de bugs lors des modifications** :
   Lorsqu'un logiciel est bien organisé en suivant les principes du *Clean Code*, les risques de casser une fonctionnalité en modifiant une autre sont considérablement réduits. Chaque module, classe ou fonction étant indépendant, les modifications locales sont moins susceptibles de causer des effets de bord non désirés ailleurs dans le système.

4. **Facilite l'ajout de nouvelles fonctionnalités** :
   Un logiciel structuré de manière claire et modulaire est plus simple à faire évoluer. Si vous devez ajouter une nouvelle fonctionnalité, vous pouvez souvent le faire sans avoir à réécrire une grande partie du code existant. Grâce à la séparation des responsabilités, il est possible d'étendre les capacités du logiciel de manière incrémentale et structurée.

   Exemple :
   Si votre logiciel est composé de classes bien définies avec des responsabilités claires, vous pouvez ajouter une nouvelle classe pour introduire une fonctionnalité sans toucher au reste du système, rendant l'ajout plus sécurisé.

"Clean Code" : Philosophie du code

5. **Réduction de la dette technique** :
   Le *Clean Code* aide à minimiser la dette technique, c'est-à-dire les compromis faits à court terme qui rendent le logiciel plus coûteux à maintenir à long terme. En suivant les pratiques de *Clean Code*, le logiciel devient plus facile à maintenir et à améliorer, ce qui réduit la dette technique et rend le projet plus durable.

6. **Facilite le testing** :
   Un code propre est plus facile à tester. Les unités de code étant petites, indépendantes, et ayant une responsabilité unique, il est facile de les isoler pour les tests unitaires ou les tests d'intégration. Cela garantit que chaque fonctionnalité fonctionne correctement avant même d'être intégrée au reste du logiciel.

Les bonnes raisons de coder « proprement »

1. Les bonnes raisons de coder « proprement »

Coder proprement est essentiel pour garantir la qualité, la maintenabilité et la longévité d'un projet logiciel.

- **Lisibilité** : Un code propre est beaucoup plus facile à lire, non seulement pour soi-même à l'avenir, mais aussi pour les autres développeurs qui pourraient être amenés à travailler sur le projet. Cela réduit le temps nécessaire pour comprendre ce que fait le code, améliorant ainsi la productivité de toute l'équipe.

Exemple :

```
// Mauvais code : difficile à comprendre
public void proc(String p) {
    if (p != null && !p.isEmpty()) {
        // Process the input
    }
}

// Code propre : noms explicites et intention claire
public void processInput(String input) {
    if (input == null || input.isEmpty()) {
        throw new IllegalArgumentException("Input cannot be null or empty");
    }
```

Les bonnes raisons de coder « proprement »

- **Maintenance** : Lorsque le code est propre et bien structuré, il devient plus facile de le maintenir et de le faire évoluer. La maintenance inclut à la fois la correction des bugs et l'ajout de nouvelles fonctionnalités. Un code propre facilite la localisation des erreurs et minimise les risques de créer de nouveaux problèmes lors de la modification.
- **Évolutivité** : Un code propre permet de faire évoluer le projet de manière plus flexible. Si le code est bien structuré avec des classes, méthodes et modules clairement séparés par leurs responsabilités, il sera plus facile d'ajouter de nouvelles fonctionnalités sans perturber le reste du système.
- **Collaboration** : Dans les projets d'équipe, un code propre améliore la collaboration. Tous les membres de l'équipe peuvent comprendre rapidement le code écrit par les autres et y apporter des modifications sans passer trop de temps à l'analyser. Cela améliore l'efficacité globale du travail d'équipe.
- **Prévenir la dette technique** : La dette technique est le coût que l'on accumule lorsqu'on adopte des solutions rapides et sales dans le code. Coder proprement dès le départ permet d'éviter cette dette, qui pourrait rendre le projet beaucoup plus coûteux à long terme en termes de maintenance et d'amélioration.

Les bonnes raisons de coder « proprement »

2. Pourquoi coder proprement ?

Coder proprement n'est pas simplement une question de style ou d'esthétique. Cela impacte directement la qualité, la productivité et la rentabilité d'un projet. Voici les principales raisons pour lesquelles il est important de coder proprement :

- **Faciliter la compréhension du code** : Le code est souvent lu et modifié bien plus souvent qu'il n'est écrit. Coder proprement facilite la compréhension, même pour un développeur qui n'a jamais vu ce code auparavant. Cela signifie que la courbe d'apprentissage pour un nouveau membre de l'équipe est plus rapide.

  Exemple :

```
// Mauvaise pratique : le code est difficile à comprendre
public void calc(int x, int y) {
   int z = x + y;
   System.out.println(z);
}

// Bonne pratique : noms explicites et intentions claires
public void calculateAndPrintSum(int firstNumber, int secondNumber) {
   int sum = firstNumber + secondNumber;
```

Les bonnes raisons de coder « proprement »

- **Réduire les erreurs** : Un code propre suit des principes qui encouragent des pratiques saines, telles que la séparation des responsabilités et l'encapsulation. En suivant ces principes, le risque d'introduire des erreurs ou des bugs est considérablement réduit.
- **Améliorer la maintenabilité** : Un code propre est plus facile à maintenir. Si des modifications sont nécessaires à l'avenir (par exemple, pour corriger des bugs ou ajouter des fonctionnalités), elles peuvent être faites de manière efficace sans risque d'introduire de nouveaux problèmes. Cela rend également les revues de code plus simples et plus efficaces.
- **Gagner du temps à long terme** : Bien que coder proprement puisse parfois prendre un peu plus de temps au départ, cela permet de gagner du temps à long terme. Un code clair et organisé est plus facile à adapter et à déboguer, ce qui réduit les efforts nécessaires pour corriger des bugs ou ajouter des fonctionnalités.
- **Améliorer la qualité du logiciel** : Un code propre produit un logiciel plus robuste, fiable, et de meilleure qualité. Les utilisateurs finaux ressentent directement cet effet, car le logiciel est plus stable et plus performant.

Les bonnes raisons de coder « proprement »

3. Apports de "Clean Code"

Le *Clean Code* apporte de nombreux avantages non seulement pour le développeur, mais aussi pour l'équipe de développement, le produit final, et même l'entreprise.

- **Lisibilité et simplicité** : Le *Clean Code* encourage la clarté et la simplicité. Chaque méthode, fonction ou classe doit être facile à comprendre et à lire. Cela permet de s'assurer que tout le monde, même ceux qui n'ont pas participé à l'écriture du code, peut rapidement comprendre ce qu'il fait.

Les bonnes raisons de coder « proprement »

Exemple :

```
// Mauvais code : la méthode fait trop de choses
public void handleOrder(Order order) {
   if (order.getItems().isEmpty()) {
         throw new IllegalArgumentException("Empty order");
   }
   double total = 0;
   for (Item item : order.getItems()) {
         total += item.getPrice();
   }
   System.out.println("Total: " + total);
}
// Clean Code : méthode décomposée en petites parties
public void handleOrder(Order order) {
   validateOrder(order);
   double total = calculateTotal(order);
   printTotal(total);
}
private void validateOrder(Order order) {
   if (order.getItems().isEmpty()) {
         throw new IllegalArgumentException("Empty order");
   }
}
private double calculateTotal(Order order) {
   return order.getItems().stream().mapToDouble(Item::getPrice).sum();
}
```

Les bonnes raisons de coder « proprement »

- **Réutilisabilité** : Le *Clean Code* encourage la création de composants modulaires et réutilisables. En suivant les principes comme la séparation des responsabilités (chaque classe ou méthode ne fait qu'une seule chose), le code devient plus modulaire, ce qui facilite sa réutilisation dans d'autres parties du projet ou dans des projets futurs.
- **Facilité de test** : Un code propre est généralement plus facile à tester. Les petites méthodes indépendantes, bien nommées et bien structurées, permettent d'écrire des tests unitaires efficaces. Les développeurs peuvent ainsi vérifier le bon fonctionnement de chaque unité de code de manière isolée, ce qui réduit les risques d'introduire des régressions lors des modifications futures.

  Exemple de méthode facile à tester :

```
public double calculateTotal(Order order) {
   return order.getItems().stream().mapToDouble(Item::getPrice).sum();
}
```

Cette méthode est concise et se concentre sur une seule tâche, ce qui facilite son test avec des cas de test clairs et directs.

- **Modularité** : Le *Clean Code* favorise la modularité. Un système modulaire est un système dans lequel chaque composant est indépendant des autres et peut être modifié ou remplacé sans affecter les autres parties du code. Cela rend le projet plus flexible et plus facile à faire évoluer

Comment identifier un code sale

1. Comment identifier un code sale

Un **code sale** est un code qui présente des signes de mauvaise qualité, ce qui le rend difficile à lire, à comprendre, à maintenir ou à étendre.

- **Duplication de code** : Lorsque le même code est copié-collé à plusieurs endroits, cela introduit des difficultés de maintenance. Si une modification est nécessaire, elle doit être répétée partout où le code est dupliqué, ce qui est source d'erreurs.
- **Méthodes ou classes trop longues** : Les méthodes ou les classes qui font trop de choses sont difficiles à comprendre et à maintenir. Elles doivent être refactorisées en petites unités qui ont des responsabilités claires.
- **Noms vagues et non explicites** : Les noms de variables, méthodes, ou classes qui ne reflètent pas leur rôle rendent le code difficile à lire.

Quelle est votre méthode ?

1. **Séparation des responsabilités** : Chaque méthode ou classe doit faire une seule chose bien définie. Cela permet d'éviter que le code devienne trop complexe.
2. **Utilisation de noms explicites** : Je m'assure que les noms de mes méthodes, variables, et classes décrivent leur intention. Cela aide à la compréhension rapide du code.
3. **Refactorisation régulière** : Je refactorise régulièrement le code pour le simplifier, supprimer la duplication et le rendre plus lisible, même après qu'il fonctionne correctement.
4. **Testabilité** : J'écris du code en pensant aux tests. Cela signifie que chaque méthode doit être suffisamment petite et isolée pour être testée facilement.
5. **Respect des principes SOLID** : J'essaie de suivre les principes SOLID, qui aident à structurer le code de manière modulaire, évolutive et maintenable.

Les 7 péchés capitaux : Rigidité, Fragilité, etc.

Les **7 péchés capitaux** du développement logiciel, selon Robert C. Martin, sont les erreurs courantes qui rendent le code difficile à maintenir et à évoluer :

1. **Rigidité** : Le code est difficile à changer. Une modification dans une partie du système nécessite des changements dans plusieurs autres parties.
2. **Fragilité** : Le code est facilement cassé. Une petite modification entraîne des régressions ou des bugs dans d'autres parties du système.
3. **Immobility** : Le code est difficile à réutiliser dans d'autres parties du projet ou dans d'autres projets, car il est fortement couplé à des éléments spécifiques.
4. **Viscosité** : Il est plus facile de faire les choses de manière incorrecte que de les faire correctement.
5. **Complexité inutile** : Le code est plus complexe que nécessaire pour résoudre le problème. Cela rend le système difficile à comprendre.
6. **Duplication** : Le même code ou la même logique est répété à plusieurs endroits, ce qui rend le projet difficile à maintenir.
7. **Opacité** : Le code est difficile à lire et à comprendre. Cela se produit souvent avec des noms vagues ou des structures de code trop complexes.

Rappel des critères de qualité d'une application (ISO)

Les critères de qualité d’une application définis par les normes **ISO/IEC 25010** incluent les aspects suivants :

1. **Fonctionnalité** : Le logiciel doit répondre aux besoins fonctionnels spécifiés.
2. **Fiabilité** : Le logiciel doit fonctionner de manière stable et sans erreurs dans des conditions normales d'utilisation.
3. **Performance** : Le logiciel doit avoir des temps de réponse acceptables et gérer efficacement les ressources.
4. **Maintenabilité** : Le logiciel doit être facile à comprendre, à corriger et à faire évoluer.
5. **Portabilité** : Le logiciel doit pouvoir être transféré d'un environnement à un autre avec un minimum de modifications.
6. **Testabilité** : Le logiciel doit être facilement testable, avec des unités de code isolées et bien définies.
7. **Sécurité** : Le logiciel doit protéger les données des utilisateurs et assurer la confidentialité et l’intégrité.

Qui est responsable de la dette technique ?

La **dette technique** est une métaphore qui désigne le coût supplémentaire que l'on va payer à l'avenir en raison des choix techniques rapides ou de mauvaise qualité que l'on fait aujourd'hui.
Tout le monde dans l'équipe de développement est responsable de la dette technique. Cela inclut :

- **Les développeurs** : Ils doivent écrire du code propre et éviter de prendre des raccourcis qui introduiraient de la dette technique.
- **Les chefs de projet** : Ils doivent planifier suffisamment de temps pour que les développeurs puissent écrire du code de qualité, au lieu de les pousser à terminer rapidement au détriment de la qualité.
- **Les architectes techniques** : Ils doivent s'assurer que les solutions techniques choisies sont évolutives, maintenables, et ne créent pas de dette technique excessive.

La gestion proactive de la dette technique est essentielle pour garantir la pérennité du projet.

Qui est responsable de la dette technique ?

La **dette technique** est un concept qui désigne le compromis pris par une équipe de développement pour accélérer le processus de livraison d'un produit en sacrifiant la qualité du code, ce qui entraîne des coûts supplémentaires à long terme pour la maintenance et l'amélioration du système. Calculer la dette technique n'est pas toujours un processus simple ou linéaire, car elle peut inclure à la fois des aspects qualitatifs et quantitatifs du code. Cependant, il existe plusieurs méthodes et outils pour **évaluer** et **quantifier** la dette technique.

1. Méthodes qualitatives pour évaluer la dette technique

Ces méthodes reposent sur des observations et des évaluations subjectives pour déterminer où se trouve la dette technique et son ampleur :

- **Revue de code** : Un processus où plusieurs développeurs passent en revue le code pour identifier les endroits où le code est mal structuré, non conforme aux bonnes pratiques ou nécessite une refactorisation. Ces revues permettent d'identifier les sources potentielles de dette technique.
- **Feedback des développeurs** : Les développeurs eux-mêmes peuvent souvent signaler où se trouve la dette technique dans le code, car ce sont eux qui travaillent avec et qui ressentent la difficulté d'apporter des modifications ou des améliorations.
- **Rapport de refactorisation** : Identifier des parties du code qui doivent être refactorisées ou restructurées régulièrement peut être un indicateur de dette technique.

Qui est responsable de la dette technique ?

2. Méthodes quantitatives pour calculer la dette technique

Les méthodes quantitatives consistent à utiliser des métriques de code et des outils d'analyse pour mesurer la dette technique en termes concrets.

a) Métriques de qualité du code

Ces métriques sont utilisées pour mesurer la complexité, la duplicité et la maintenabilité du code. Voici quelques métriques importantes à considérer :

- **Complexité cyclomatique** : Cette métrique mesure la complexité d'une fonction ou d'une méthode en calculant le nombre de chemins indépendants dans le code. Un score élevé indique un code plus difficile à maintenir et à tester.
- **Couverture des tests** : Un faible pourcentage de couverture de tests unitaires et fonctionnels indique que le code n'est pas bien testé, ce qui augmente la probabilité d'introduire des bugs et donc la dette technique.
- **Duplication du code** : Les outils peuvent mesurer combien de fois une portion de code est dupliquée dans le projet. Plus il y a de duplication, plus il y a de dette technique, car chaque changement doit être répliqué dans plusieurs endroits du code.

Qui est responsable de la dette technique ?

b) Utilisation d'outils d'analyse de la dette technique

Il existe des outils qui permettent de quantifier la dette technique et de fournir des rapports détaillés sur les zones problématiques du code. Ces outils utilisent des règles prédéfinies pour détecter les "code smells", les faiblesses structurelles, et estimer les efforts nécessaires pour corriger ces problèmes. Voici quelques outils populaires :

- **SonarQube** : SonarQube est l'un des outils les plus populaires pour analyser la qualité du code. Il fournit une estimation de la dette technique en calculant le temps nécessaire pour corriger les problèmes détectés (bugs, vulnérabilités, code smells). La dette technique est souvent exprimée en heures ou en jours de travail.

  Exemple de rapport SonarQube :

```
Dette technique estimée : 20 jours
- 5 jours pour corriger les duplications
- 10 jours pour améliorer la couverture des tests
- 5 jours pour réduire la complexité cyclomatique
```

- **CAST (Computer Aided Software Testing)** : Cet outil offre également une estimation de la dette technique en analysant la complexité du code, les violations des bonnes pratiques et la maintenabilité.

Qui est responsable de la dette technique ?

c) Formule pour calculer la dette technique

Certains modèles mathématiques permettent de calculer la dette technique en fonction de divers facteurs de qualité. Voici un modèle couramment utilisé pour estimer la dette technique en fonction de la **maintenabilité** et des coûts de correction :

```
Dette technique = (Coût d'amélioration * Facteur d'impact) / Valeur ajoutée par la correction
```

- **Coût d'amélioration** : Le temps ou les ressources nécessaires pour corriger ou refactoriser le code.
- **Facteur d'impact** : L'importance de la dette technique dans une partie spécifique du projet (par exemple, un module critique).
- **Valeur ajoutée** : Le bénéfice attendu de la correction (amélioration de la qualité, réduction des bugs).

Qui est responsable de la dette technique ?

3. Estimer la dette technique en termes de coût

Une autre approche consiste à **estimer la dette technique en termes de coût**. L'idée est de calculer le temps supplémentaire que les développeurs passent à gérer un code sale par rapport à un code propre, puis de convertir ce temps en coûts financiers.

- **Temps supplémentaire** : Si les développeurs passent en moyenne 20% de temps en plus à corriger des bugs ou à ajouter des fonctionnalités dans une zone du code à cause de la dette technique, cela peut être mesuré en heures supplémentaires.

  Exemple de calcul :
  Si une tâche qui aurait pris 10 heures dans un code propre prend 12 heures à cause de la dette technique, la dette technique représente un surcoût de 2 heures.

  En supposant que le taux horaire d'un développeur est de 50 euros, la dette technique pour cette tâche est de :

```
Dette technique = 2 heures * 50 euros = 100 euros
```

- **Calcul à grande échelle** : En multipliant ce coût supplémentaire par toutes les tâches affectées par la dette technique dans un projet, on peut estimer l'impact total sur le budget de développement.

Qui est responsable de la dette technique ?

4. Suivi continu de la dette technique

Le suivi de la dette technique doit être un processus continu. Voici comment intégrer le suivi dans le flux de développement :

- **Rapports réguliers** : Utiliser des outils comme SonarQube pour générer des rapports de dette technique après chaque cycle de développement ou chaque sprint dans une approche Agile.
- **Refactorisation planifiée** : Inclure des cycles de refactorisation réguliers dans le plan de développement pour réduire progressivement la dette technique.
- **Mise en place de seuils** : Définir des seuils de dette technique acceptables pour que le projet ne devienne pas ingérable (par exemple, limiter la duplication à 5% du code ou fixer une limite de complexité cyclomatique).

Les principes "Clean Code"

Devenir un **"Clean Coder"** (développeur qui pratique les principes du *Clean Code*) est un processus continu d'apprentissage, d'amélioration, et de discipline. Voici un procédé en plusieurs étapes pour devenir un *Clean Coder*, basé sur l'apprentissage de bonnes pratiques de développement, la pratique régulière, et l'application des principes du *Clean Code* dans les projets réels.

Les principes "Clean Code"

1. Étudier les principes du *Clean Code*

La première étape pour devenir un *Clean Coder* est d'étudier et de comprendre les principes fondamentaux du *Clean Code*. Ces principes ont été largement développés par Robert C. Martin dans son livre *Clean Code* et d'autres ouvrages connexes. Il est important de comprendre pourquoi ces principes sont importants et comment ils aident à produire du code de haute qualité.

- **Livres recommandés :**
  - *Clean Code: A Handbook of Agile Software Craftsmanship* de Robert C. Martin.
  - *The Pragmatic Programmer* d'Andrew Hunt et David Thomas.
- **Principes clés à étudier :**
  - **Simplicité** : Le code doit être aussi simple que possible pour accomplir la tâche.
  - **Nommer clairement** : Les noms de classes, méthodes et variables doivent être explicites et refléter leur rôle.
  - **Pas de duplication** : DRY (Don't Repeat Yourself) est un principe clé ; ne répétez pas le même code à différents endroits.
  - **Séparation des responsabilités** : Une classe ou une méthode ne doit faire qu'une seule chose.
  - **Testabilité** : Le code doit être facile à tester (tests unitaires, tests d'intégration).

Les principes "Clean Code"

2. Pratiquer régulièrement avec des katas de code

Une fois les principes appris, il est essentiel de les pratiquer régulièrement pour intégrer les bonnes habitudes. Les **katas de code** sont des exercices de programmation conçus pour répéter et perfectionner l'écriture de code propre.

- **Qu'est-ce qu'un code kata ?**
  Un kata est un exercice court et répétitif qui permet d'améliorer ses compétences. L'idée est de résoudre un problème simple en appliquant les principes du *Clean Code* et de refactorer le code jusqu'à obtenir une solution claire et maintenable.
- **Exemple de katas populaires :**
  - **FizzBuzz** : Un exercice classique qui consiste à écrire un programme qui imprime des nombres, mais remplace certains par "Fizz" ou "Buzz" selon des règles spécifiques.
  - **Roman Numerals Kata** : Un exercice consistant à convertir des chiffres en nombres romains en appliquant une logique simple tout en respectant les principes du *Clean Code*.
  - **Bowling Game Kata** : Un kata plus complexe qui simule le calcul du score d'une partie de bowling.

  Conseil pratique : Faire régulièrement ces exercices, seul ou en groupe, en essayant de refactorer à chaque itération pour améliorer la lisibilité et la simplicité du code.

Les principes "Clean Code"

3. Appliquer les principes du *Clean Code* dans vos projets réels

Une fois que vous avez acquis une certaine maîtrise des katas, l'étape suivante consiste à **appliquer ces principes dans des projets réels**. Cela peut inclure des projets personnels ou professionnels. Voici comment commencer à appliquer les pratiques du *Clean Code* au quotidien :

- **Revue de code** : Intégrer les revues de code dans votre flux de travail et encourager l'application des principes du *Clean Code* dans les projets d'équipe. Cela permet de partager des bonnes pratiques et de corriger les mauvaises habitudes.
- **Refactorisation continue** : Toujours chercher à améliorer et à simplifier le code existant. Ne pas attendre qu'un code devienne obsolète ou ingérable pour le refactorer.
- **Écrire des tests** : Pour chaque nouvelle fonctionnalité, écrire des tests unitaires et des tests d'intégration. Cela garantit que votre code est non seulement fonctionnel, mais aussi testable et maintenable à long terme.

Les principes "Clean Code"

4. Refactoriser régulièrement

Le *Clean Code* ne consiste pas seulement à écrire du code propre dès le départ, mais également à **refactoriser** continuellement pour améliorer la qualité du code existant. Refactoriser consiste à restructurer le code sans en modifier le comportement externe, afin de le rendre plus lisible, maintenable et modulaire.

- **Refactorisation guidée par les tests** : Avant de refactoriser, assurez-vous que votre code est bien couvert par des tests. Cela garantit que vous ne cassez rien en cours de refactorisation.
- **Petits pas** : La refactorisation doit se faire par petites étapes. Chaque modification doit améliorer une petite partie du code, plutôt que de faire une refactorisation massive qui risque d'introduire des bugs.
- **Identifier et corriger les *code smells*** : Apprendre à repérer les *code smells* (signes indiquant des problèmes dans le code, comme les méthodes trop longues ou les classes trop complexes) et les corriger en appliquant les principes du *Clean Code*.

Les principes "Clean Code"

5. Participer à des revues de code et des échanges avec d'autres développeurs

L'un des meilleurs moyens d'apprendre et d'améliorer ses compétences en *Clean Code* est de **participer à des revues de code** avec d'autres développeurs. Ces revues permettent d'identifier les erreurs, les zones à améliorer, et de partager des bonnes pratiques.

- **Code reviews** : Impliquez-vous activement dans des revues de code. Cela permet non seulement de recevoir des retours constructifs, mais aussi d'apprendre des approches et des idées d'autres développeurs.
- **Communautés et échanges** : Participez à des groupes de développeurs, des forums en ligne, ou des conférences. Apprendre de l'expérience des autres est un excellent moyen de progresser dans la pratique du *Clean Code*.

Les principes "Clean Code"

6. Tester constamment votre code

Un *Clean Coder* doit avoir une approche de **développement piloté par les tests** (TDD - Test Driven Development). Le TDD encourage les développeurs à écrire des tests avant d'écrire le code et à utiliser ces tests pour guider l'implémentation de la solution.

- **Écrire des tests avant le code** : En suivant le processus TDD, vous écrivez d'abord un test qui échoue (car le code n'existe pas encore), puis vous écrivez juste assez de code pour passer le test, et enfin vous refactorisez.
- **Métriques de tests** : Utilisez des outils pour surveiller la couverture des tests, en veillant à ce que toutes les parties importantes du code soient couvertes par des tests automatisés.

Les principes "Clean Code"

7. Utiliser des outils pour garantir la qualité du code

Des outils d'analyse de la qualité du code peuvent vous aider à **repérer les problèmes** dans le code et à mesurer son état. Ces outils permettent d'identifier les *code smells*, les duplications, la complexité, etc.

- **Outils recommandés :**
  - **SonarQube** : Analyse la qualité du code, détecte les bugs, les vulnérabilités et les *code smells*, et mesure la dette technique.
  - **Lint** : Outil de vérification statique qui détecte les erreurs de syntaxe et les problèmes potentiels dans le code.
  - **Junit** pour les tests unitaires.

8. Rester à jour

Le développement logiciel évolue constamment, tout comme les bonnes pratiques. Pour rester un *Clean Coder*, il est important de se tenir au courant des **nouvelles pratiques** et des **nouveaux outils**.

- **Continuer à apprendre** : Suivre des blogs, lire des livres récents sur le développement logiciel, participer à des formations et des conférences.
- **Expérimenter** : N'hésitez pas à tester de nouvelles approches, méthodologies ou langages. Les pratiques évoluent, et il est important d'adapter votre manière de coder en fonction des nouvelles connaissances.

TP 1

Sujet du Kata : Refactorisation et Clean Code

Vous avez une classe `PrimePrinter` qui génère et imprime les 1000 premiers nombres premiers. Cependant, ce code est complexe, mélange plusieurs responsabilités (génération de nombres premiers, gestion de la pagination, et affichage), et manque de clarté. Votre mission est d'améliorer ce code en suivant les principes du *Clean Code*, sans changer son comportement.

Objectifs :

1. **Améliorer la lisibilité**
2. **Séparer les responsabilités**
3. **Simplifier la logique**
4. **Respecter les bonnes pratiques du Clean Code**

TP 2

Sujet du Kata : Refactorisation et Clean Code

Vous avez un jeu de Tic-Tac-Toe écrit en Java, mais le code présente plusieurs problèmes qui affectent sa lisibilité, sa maintenabilité et sa modularité. Le code contient également beaucoup de commentaires qui peuvent être éliminés si le code est refactorisé correctement. Votre mission est d'améliorer ce code pour en faire un exemple de Clean Code en suivant les principes de lisibilité, de séparation des responsabilités, et de simplicité.

Objectifs :

1. **Améliorer la lisibilité**
2. **Séparer les responsabilités**
3. **Simplifier la logique**
4. **Respecter les bonnes pratiques du Clean Code**