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**

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Jour 2

L’architecture logicielle

L'architecture logicielle représente l'organisation des composants d'un système logiciel ainsi que les interactions entre ces composants. Elle définit les fondations d'un logiciel, à partir desquelles toutes les décisions de conception sont prises. Une architecture bien pensée permet de répondre aux exigences fonctionnelles et non fonctionnelles d'une application.

Pourquoi a-t-on besoin d’une bonne architecture ?

1. **Faciliter la maintenance** :
   Une bonne architecture réduit la complexité du logiciel et facilite la maintenance à long terme. Les systèmes évoluent constamment, et il est essentiel que les nouvelles fonctionnalités puissent être ajoutées sans nécessiter de changements majeurs dans la structure du code existant. Cela permet d'éviter la création de "code spaghetti", c'est-à-dire un code où tout est interconnecté et où une modification dans une partie du système peut entraîner des effets indésirables ailleurs.
2. **Améliorer la lisibilité et la compréhension du code** :
   Une architecture claire et bien définie permet aux développeurs de comprendre rapidement le fonctionnement global du système. Le code devient plus lisible, les intentions sont plus explicites, et il est plus facile de naviguer dans les différentes parties du logiciel. Une bonne architecture impose souvent des règles sur la séparation des responsabilités (par exemple, avec des couches comme l'interface utilisateur, la logique métier, et la gestion des données), ce qui aide à isoler les préoccupations et à organiser le code de manière logique.

L’architecture logicielle

3. **Assurer l’évolutivité** :
   À mesure qu'un logiciel grandit en termes de complexité et de fonctionnalités, l'architecture doit permettre cette évolution sans compromettre les performances ou la stabilité. Une bonne architecture facilite l’ajout de nouvelles fonctionnalités ou modules sans devoir tout réécrire. En revanche, une mauvaise architecture rend l’évolution complexe, coûteuse, et peut même conduire à la nécessité de réécrire tout ou partie du système.

4. **Favoriser la réutilisabilité** :
   Une architecture modulaire et bien conçue permet de réutiliser certaines parties du code dans d'autres projets ou au sein d'autres modules du même projet. Par exemple, en séparant bien la logique métier du reste du système, il devient possible de réutiliser cette logique dans différentes applications (web, mobile, etc.) ou de la remplacer si nécessaire sans impacter l'intégralité du système.

5. **Réduire les risques et les coûts** :
   En ayant une architecture solide dès le début, nous réduisons les risques de voir le projet échouer ou devenir coûteux à maintenir. Une mauvaise architecture peut conduire à des délais importants pour corriger les bugs, intégrer de nouvelles fonctionnalités ou même maintenir la stabilité de l'application. Ces risques peuvent engendrer des surcoûts non prévus et diminuer la satisfaction des clients ou des utilisateurs finaux.

L’architecture logicielle

6. **Permettre une meilleure testabilité** :
   Un des principes fondamentaux du clean code est la testabilité. Une architecture bien définie permet de tester facilement chaque composant de manière isolée. L'isolation des responsabilités au sein du code facilite la création de tests unitaires ou d'intégration, garantissant ainsi que chaque brique du logiciel fonctionne correctement indépendamment du reste du système.

7. **Faciliter la collaboration entre les équipes** :
   Une bonne architecture impose des conventions et des règles claires sur la façon dont les composants interagissent entre eux. Cela aide différentes équipes à collaborer de manière efficace, car elles peuvent travailler sur différents modules ou parties du système sans risquer de créer des conflits ou des dépendances non maîtrisées.

4 notions importantes pour une application propre

1. Couplage faible

Le **couplage** fait référence à la manière dont les différents modules ou classes d'une application dépendent les uns des autres. Un couplage faible signifie que chaque module dépend le moins possible d'autres modules. Cela rend les modules indépendants et facilite leur maintenance ou modification sans affecter les autres parties de l'application.

Pourquoi c’est important :

- Réduit l'impact des modifications sur l'ensemble du système.
- Facilite la réutilisation des modules dans d'autres projets.
- Permet de tester des composants de manière isolée.

4 notions importantes pour une application propre

2. Cohésion

La **cohésion** fait référence à la mesure dans laquelle les responsabilités d'une classe ou d'un module sont étroitement liées. Une classe est dite fortement cohésive lorsqu'elle n'a qu'une seule responsabilité claire et que toutes ses méthodes sont directement liées à cette responsabilité.

Pourquoi c’est important :

- Facilite la lisibilité et la compréhension du code.
- Rend le code plus simple à maintenir.
- Réduit les risques d’introduire des erreurs lors des modifications.

4 notions importantes pour une application propre

3. Changements locaux

Les **changements locaux** signifient que les modifications dans une partie du code ne devraient affecter que des composants spécifiques, et non l'ensemble du système. Cela est souvent le résultat d'une bonne modularité, d'un faible couplage, et d'une forte cohésion.

Pourquoi c’est important :

- Réduit le risque d'introduire des bugs lors des modifications.
- Facilite l'implémentation de nouvelles fonctionnalités.
- Accélère le développement.

4 notions importantes pour une application propre

4. Nommage

Le **nommage** est l'art de donner des noms significatifs et explicites aux variables, méthodes, classes, et autres éléments du code. Un bon nommage rend le code plus lisible et compréhensible sans avoir à parcourir tout le code.

Connaître les principes de programmation objet SOLID

1. Single Responsibility Principle (SRP) — Principe de responsabilité unique

Le **Single Responsibility Principle** stipule qu'une classe ou un module ne doit avoir qu'une seule raison de changer, c'est-à-dire qu'elle doit avoir une seule responsabilité clairement définie.

Pourquoi c’est important :

- **Simplifie la maintenance** : Quand une classe ne fait qu'une seule chose, elle est plus facile à comprendre et à modifier.
- **Réduit les risques d'erreur** : En limitant la portée d'une classe, on minimise les effets de bord et les risques de bugs lorsque cette classe est modifiée.

Connaître les principes de programmation objet SOLID

2. Open-Closed Principle (OCP) — Principe ouvert/fermé

Le **Open-Closed Principle** stipule que les entités logicielles (classes, modules, fonctions, etc.) doivent être ouvertes à l'extension mais fermées à la modification. Cela signifie que l'on doit pouvoir ajouter de nouvelles fonctionnalités sans avoir à modifier le code existant.

Pourquoi c’est important :

- **Réduit le risque d'introduire des bugs** : En ne modifiant pas le code existant, on évite les régressions potentielles.
- **Facilite l'ajout de nouvelles fonctionnalités** : Le système peut évoluer sans perturber les fonctionnalités existantes.

Connaître les principes de programmation objet SOLID

3. Liskov Substitution Principle (LSP) — Principe de substitution de Liskov

Le **Liskov Substitution Principle** stipule que les objets d'une sous-classe doivent pouvoir être remplacés par des objets de leur superclasse sans que cela n'affecte la fonctionnalité du programme. En d'autres termes, les sous-classes doivent respecter les contrats définis par leurs classes parentes.

Pourquoi c’est important :

- **Assure la cohérence** : Si une sous-classe ne peut pas être utilisée à la place de sa superclasse, le polymorphisme est cassé.
- **Améliore la fiabilité** : Les classes dérivées doivent respecter les attentes de la classe parent, sinon des comportements imprévus peuvent survenir.

Connaître les principes de programmation objet SOLID

4. Interface Segregation Principle (ISP) — Principe de ségrégation des interfaces

Le **Interface Segregation Principle** stipule qu'il est préférable de créer plusieurs interfaces spécifiques plutôt qu'une interface générale unique. Cela permet d'éviter que les classes implémentent des méthodes dont elles n'ont pas besoin.

Pourquoi c’est important :

- **Évite la surcharge des classes** : Les classes n'ont pas à implémenter des méthodes inutiles ou non pertinentes.
- **Simplifie le code** : Les interfaces plus petites et plus spécifiques sont plus faciles à comprendre et à utiliser.

Connaître les principes de programmation objet SOLID

5. Dependency Inversion Principle (DIP) — Principe d'inversion des dépendances

Le **Dependency Inversion Principle** stipule que les modules de haut niveau ne doivent pas dépendre de modules de bas niveau. Tous deux doivent dépendre d'abstractions (interfaces). En d'autres termes, le code doit dépendre d'abstractions plutôt que de détails concrets.

Pourquoi c’est important :

- **Facilite la modification et la maintenance** : Le code de haut niveau ne change pas lorsque les modules de bas niveau changent.
- **Favorise l'inversion de contrôle** : Les dépendances sont injectées plutôt que créées directement au sein des modules de haut niveau.

Comprendre la notion de couplage

1. Héritage

L'**héritage** est un mécanisme en programmation orientée objet où une classe dérive d'une autre classe, héritant ainsi de ses attributs et méthodes. La classe dérivée est souvent appelée **sous-classe** ou **classe enfant**, et la classe dont elle hérite est appelée **superclasse** ou **classe parent**.

Avantages de l'héritage :

- Réutilisation du code : La sous-classe peut utiliser les fonctionnalités déjà définies dans la classe parent.
- Organisation hiérarchique : Il permet de structurer les classes de manière hiérarchique, facilitant ainsi la compréhension des relations entre classes.

Inconvénients de l'héritage :

- Couplage fort : La sous-classe dépend fortement de la superclasse, ce qui rend les modifications difficiles. Si nous modifions la superclasse, cela peut avoir des répercussions imprévues sur toutes les sous-classes.
- Difficulté à évoluer : Lorsque la hiérarchie devient trop complexe, l'héritage devient difficile à maintenir et à étendre.

Comprendre la notion de couplage

2. Composition

La **composition** est une technique où une classe inclut une instance d'une autre classe comme attribut. Cela signifie que la classe "composante" fait partie de la classe "contenante". Contrairement à l'héritage, la composition favorise un couplage plus faible et flexible, car la classe contenant ne dépend pas directement de la structure interne de la classe composante.

Avantages de la composition :

- Couplage faible : La classe dépend de l'interface ou de l'implémentation d'une autre classe, mais pas directement de ses détails internes.
- Flexibilité : La composition permet de changer facilement le comportement en remplaçant l'instance utilisée par une autre.
- Favorise la réutilisation : Les objets composés peuvent être réutilisés dans différents contextes sans modifier la structure des classes.

Comprendre la notion de couplage

3. Agrégation

L'**agrégation** est une forme plus faible de composition. Elle indique qu'une classe est liée à une autre, mais l'objet composant peut exister indépendamment de l'objet conteneur. En d'autres termes, une agrégation décrit une relation "a un" (has-a) où la classe contenante peut exister sans la classe composante, et inversement.

Avantages de l'agrégation :

- Couplage encore plus faible : L'objet agrégé peut exister indépendamment de l'objet qui l'agrège.
- Flexibilité : Comme la classe agrégée n'est pas fortement liée à la classe principale, elle peut être partagée entre plusieurs instances sans répercussions.

Comprendre la notion de couplage

Différence entre héritage, composition, et agrégation :

- **Héritage** : C'est une relation "est un" (is-a). La sous-classe hérite des attributs et comportements de la superclasse, mais cela crée un fort couplage entre les classes.
- **Composition** : C'est une relation "a un" (has-a) forte, où la classe contenante ne peut pas exister sans la classe composante. La composition est souvent préférée à l'héritage pour réduire le couplage.
- **Agrégation** : C'est une relation "a un" plus faible, où les objets agrégés peuvent exister indépendamment des objets contenant.

Les patrons de conceptions

1. Origine des patrons de conception

Les **patrons de conception** ont été popularisés par les membres du **"Gang of Four" (GoF)**, à savoir Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides. Dans leur livre **"Design Patterns: Elements of Reusable Object-Oriented Software"**, publié en 1994, ils ont décrit 23 patrons de conception qui aident à résoudre des problèmes récurrents dans le développement logiciel orienté objet.

L'idée derrière les patrons de conception est de documenter des solutions éprouvées pour faciliter la réutilisation et la communication entre développeurs. En appliquant ces patrons, nous réduisons le risque de réinventer des solutions inefficaces ou maladroites.

Les patrons de conceptions

2. Les catégories de patrons de conception

Les patrons de conception se divisent en trois grandes catégories :

A. Patrons de création

Ces patrons se concentrent sur la manière dont les objets sont créés, en nous aidant à instancier des objets de manière contrôlée et flexible. Ils masquent la logique complexe derrière la création d'objets et permettent de découpler le code de l'instanciation directe.

B. Patrons structurels

Les patrons structurels concernent l'organisation des classes et des objets pour former des structures plus grandes et plus flexibles. Ils facilitent la composition des objets pour résoudre des problèmes tout en minimisant les dépendances entre eux.

C. Patrons comportementaux

Ces patrons se concentrent sur la communication entre les objets et la manière dont ils interagissent. Ils permettent de définir clairement la manière dont les objets collaborent, en simplifiant le flux de communication et les responsabilités.

Les patrons de conceptions

3. Liste des patrons de conception et descriptions

A. Patrons de création

1. **Singleton** :
   Garantit qu'une classe n'a qu'une seule instance, et fournit un point d'accès global à cette instance.
2. **Factory Method** :
   Définit une interface pour créer un objet, mais laisse les sous-classes décider quelle classe instancier. Cela permet de différer l'instanciation à des sous-classes.
3. **Abstract Factory** :
   Fournit une interface pour créer des familles d'objets liés ou dépendants sans spécifier leurs classes concrètes.
4. **Builder** :
   Sépare la construction d'un objet complexe de sa représentation afin que le même processus de construction puisse créer différentes représentations.
5. **Prototype** :
   Crée de nouveaux objets en copiant (clonant) des objets existants plutôt qu'en créant de nouveaux objets à partir de zéro.

Les patrons de conceptions

B. Patrons structurels

1. **Adapter** :
   Permet à des interfaces incompatibles de fonctionner ensemble. Il fait office d'intermédiaire entre deux objets n'ayant pas la même interface.
2. **Bridge** :
   Sépare l'abstraction de son implémentation afin qu'elles puissent évoluer indépendamment.
3. **Composite** :
   Permet de composer des objets en structures arborescentes pour représenter des hiérarchies partielles ou totales. Il traite les objets individuels et leurs compositions de manière uniforme.
4. **Decorator** :
   Permet d'ajouter dynamiquement des responsabilités à un objet, sans modifier son code. Il offre une alternative plus flexible à l'héritage pour l'extension des fonctionnalités.

Les patrons de conceptions

5. **Facade** :
   Fournit une interface simplifiée à un ensemble de classes ou à un système complexe. Il masque la complexité et offre une interface plus accessible.
6. **Flyweight** :
   Réduit la consommation de mémoire en partageant autant que possible les données entre les objets similaires.
7. **Proxy** :
   Fournit un substitut ou un placeholder pour un autre objet afin de contrôler l'accès à cet objet.

Les patrons de conceptions

C. Patrons comportementaux

1. **Chain of Responsibility** :
   Évite de coupler l'expéditeur d'une requête à son destinataire en laissant plusieurs objets avoir la possibilité de traiter la requête. Les objets sont chaînés et la requête passe le long de la chaîne jusqu'à ce qu'un objet la prenne en charge.
2. **Command** :
   Encapsule une requête en tant qu'objet, ce qui permet de paramétrer les clients avec des requêtes, de mettre en file d'attente ou d'annuler des opérations.
3. **Iterator** :
   Fournit un moyen d'accéder séquentiellement aux éléments d'un objet composite sans exposer sa représentation sous-jacente.
4. **Mediator** :
   Définit un objet qui encapsule la manière dont un ensemble d'objets interagit, réduisant ainsi les dépendances directes entre eux.

Les patrons de conceptions

5. **Memento** :
   Capture et externalise l'état interne d'un objet sans violer l'encapsulation, permettant ainsi de restaurer cet état ultérieurement.
6. **Observer** :
   Définit une relation de dépendance entre des objets tels que lorsqu'un objet change d'état, tous ses dépendants en sont informés et mis à jour automatiquement.
7. **State** :
   Permet à un objet de changer son comportement lorsque son état interne change. L'objet semblera changer de classe.
8. **Strategy** :
   Définit une famille d'algorithmes, encapsule chacun d'eux, et les rend interchangeables. Cela permet de choisir dynamiquement un algorithme à utiliser.

Les patrons de conceptions

9. **Template Method** :
   Définit la structure d'un algorithme, mais laisse certaines étapes à des sous-classes. Cela permet de réutiliser du code tout en laissant la flexibilité de l'implémentation des étapes spécifiques.
10. **Visitor** :
    Permet de définir de nouvelles opérations sur des objets sans modifier leurs classes.

Les Values Objects

Les Value Objects (Objets de Valeur)

Les **Value Objects** sont un concept fondamental en conception orientée objet, particulièrement dans le cadre des architectures DDD (Domain-Driven Design). Contrairement aux entités, qui sont définies par leur identité unique, les **Value Objects** sont définis par leurs attributs. Ils représentent des objets simples, sans identité propre, qui encapsulent des valeurs et du comportement.

1. Principes des Value Objects

Les **Value Objects** suivent plusieurs principes clés :

- **Absence d'identité** : Contrairement à une entité, un objet de valeur n'a pas d'identité propre. Ce qui signifie que deux objets de valeur contenant les mêmes données sont considérés comme égaux.
- **Encapsulation des valeurs** : Un objet de valeur encapsule un ou plusieurs attributs, et son comportement est centré sur la manipulation de ces valeurs.
- **Invariance** : Un **Value Object** ne change pas d'état. Au lieu de modifier les valeurs internes, on crée de nouveaux objets de valeur pour refléter des modifications, ce qui favorise la sécurité et la prévisibilité du code.
- **Egalité basée sur les valeurs** : Deux **Value Objects** sont égaux si et seulement s'ils contiennent les mêmes valeurs, indépendamment de leur position en mémoire.

Les Values Objects

2. Immutabilité des Value Objects

Un principe fondamental des **Value Objects** est leur **immutabilité**. Cela signifie qu'une fois créé, un **Value Object** ne peut pas être modifié. Cela permet de garantir l'intégrité des objets de valeur et de faciliter le raisonnement sur le code.

Pourquoi l'immutabilité est importante :

- **Sécurité des données** : Puisque les objets de valeur ne peuvent pas être modifiés, il n'y a pas de risque de voir un objet changer d'état de manière inattendue.
- **Simplicité** : Les objets immuables éliminent une classe entière de bugs liés aux modifications d'état indésirées.
- **Facilité dans les environnements concurrents** : Dans les contextes multithreadés ou parallèles, les objets immuables sont intrinsèquement thread-safe, car leur état ne change jamais après création.

Les Values Objects

3. Effectuer des opérations en parallèle avec les Value Objects

Les **Value Objects**, en tant qu'objets immuables, sont particulièrement adaptés aux **opérations en parallèle**. Dans des environnements multithread ou lorsqu'on effectue des calculs parallèles, le fait qu'un **Value Object** ne puisse pas être modifié garantit qu'il peut être partagé en toute sécurité entre plusieurs threads.

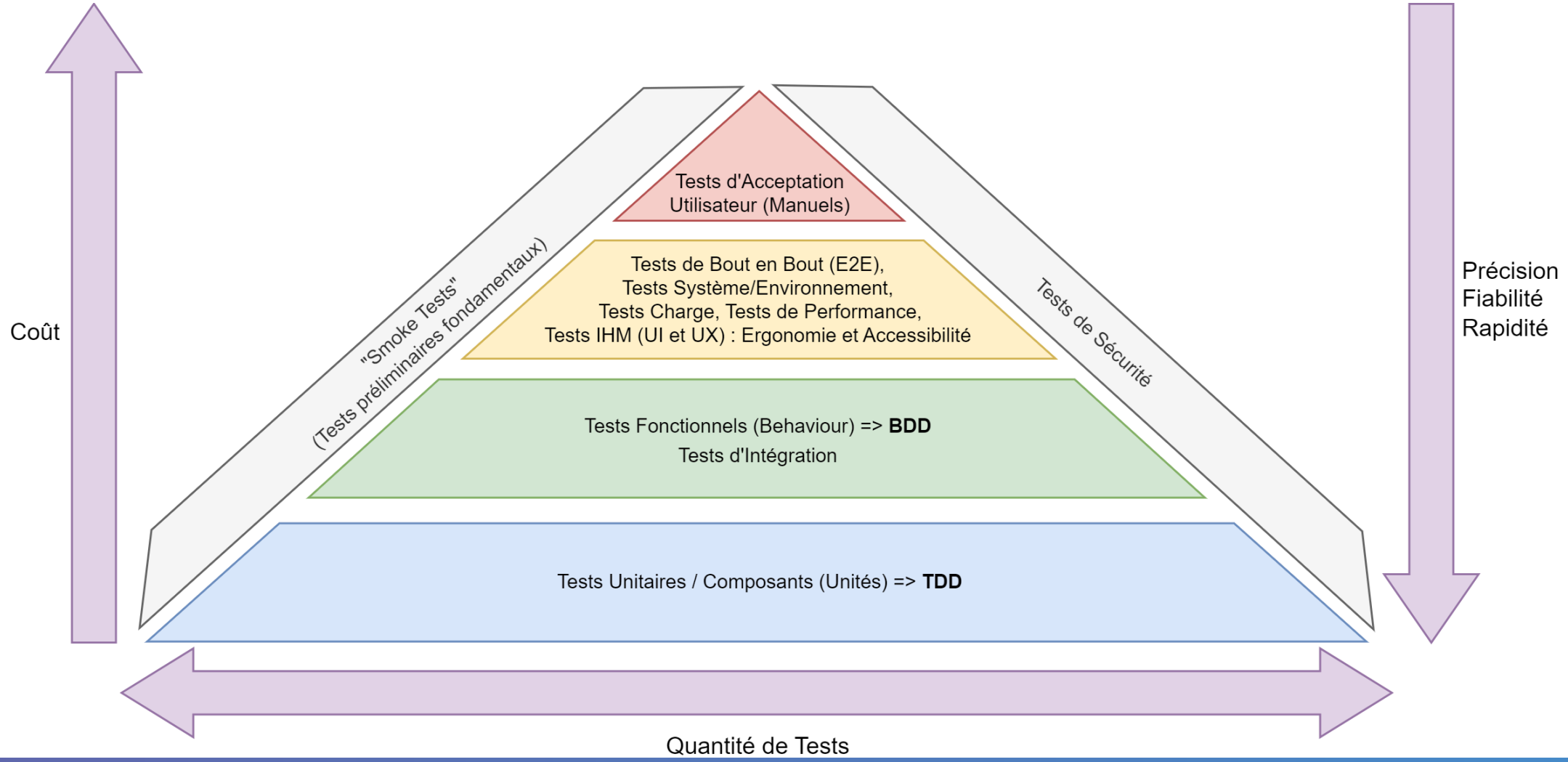
Avantages des Value Objects dans le traitement parallèle :

- **Thread-safety** : Comme les **Value Objects** ne peuvent pas être modifiés, il n'y a pas de risques de conditions de course ou d'accès concurrentiel inapproprié.
- **Réduction des bugs liés au partage d'état** : L'immutabilité élimine les erreurs causées par des modifications concurrentes sur un même objet partagé.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Jour 3

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

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Développement en TDD

Rappel des frameworks java pour les Test

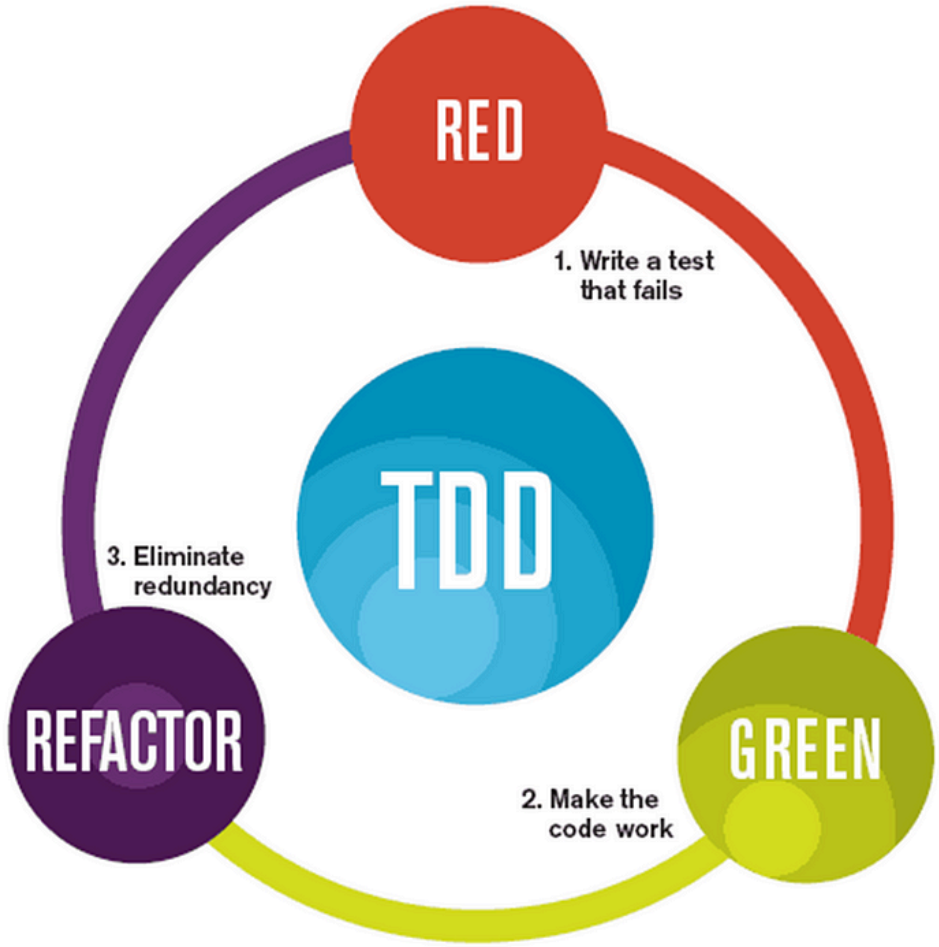
En Java, plusieurs frameworks sont disponibles pour vous aider à créer et à gérer des tests unitaires, d'intégration et fonctionnels.

- **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 :

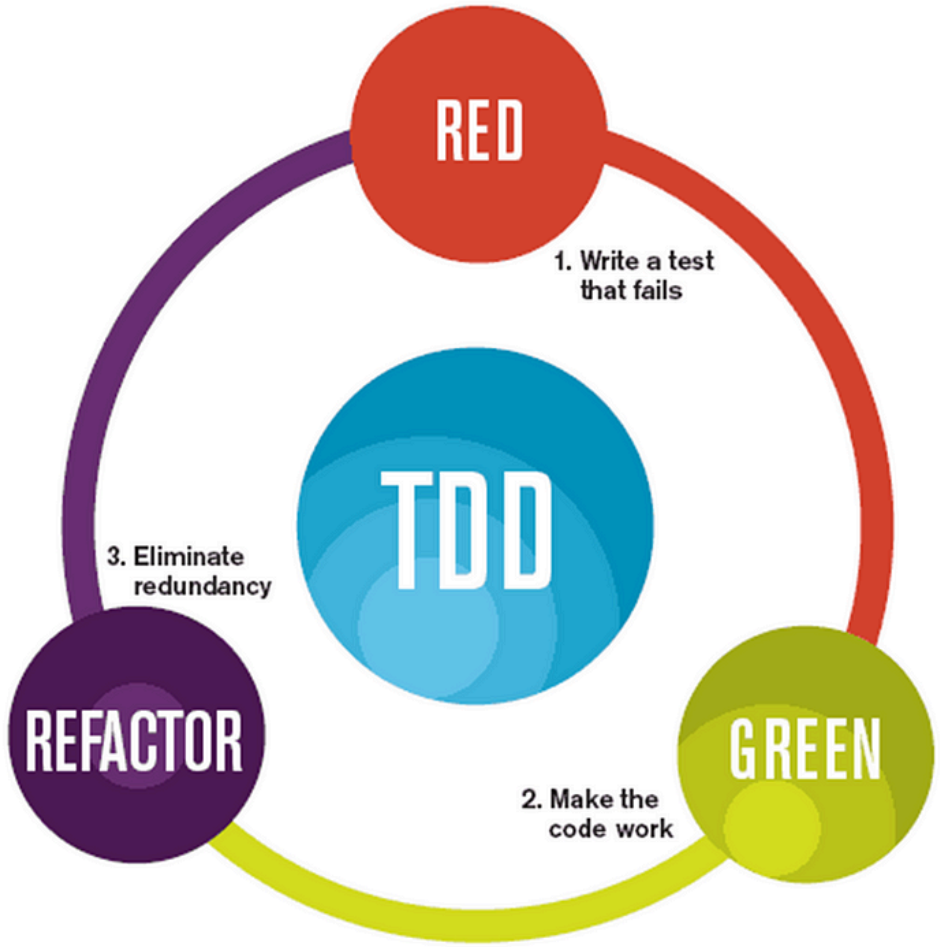
- **Écrire un test** : Vous commencez par écrire un test pour la nouvelle fonctionnalité que vous voulez implémenter. À ce stade, le test échouera, car vous n'avez pas encore écrit le code de production.
- **Exécuter tous les tests et voir si le nouveau test échoue** : Ceci est fait pour s'assurer que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

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

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

```
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
```

```
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
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Jour 4

Refactorer une application legacy

Refactorer une Application Legacy

Une application legacy est un logiciel hérité qui peut être difficile à maintenir ou à faire évoluer en raison de sa complexité, de son ancienneté ou de son manque de tests. Refactorer ce type d'application est essentiel pour améliorer sa qualité et faciliter son évolution.

1. Tester

Avant de commencer le refactoring, il est crucial d'avoir une suite de tests automatisés pour s'assurer que les modifications n'introduisent pas de régressions. Si l'application n'a pas de tests, il faut en écrire pour les parties critiques du code.

2. Refactorer

Appliquez des techniques de refactoring pour améliorer la structure du code sans modifier son comportement observable. Le refactoring doit être effectué par petites étapes, en vérifiant après chaque modification que les tests passent toujours.

3. Apporter de la Valeur

Le refactoring doit faciliter l'ajout de nouvelles fonctionnalités, améliorer les performances ou rendre le code plus maintenable. Il doit toujours avoir un objectif clair qui apporte de la valeur à l'application.

Refactorer une Application Legacy

Exemple de Refactoring

Code Avant Refactoring :

```
public class ClientService {
    public void processClientData(Client client) {
        // Validation des données client
        if (client.getName() != null && client.getAge() > 0) {
            // Calcul du score de crédit
            int creditScore = calculateCreditScore(client);
            // Envoi d'un email de bienvenue
            EmailService emailService = new EmailService();
            emailService.sendWelcomeEmail(client.getEmail());
            // Mise à jour de la base de données
            Database database = new Database();
            database.updateClientRecord(client);
        } else {
            throw new IllegalArgumentException("Données client invalides");
        }
    }
}
```

Refactorer une Application Legacy

Problèmes Identifiés :

- **Long Method** : La méthode `processClientData` effectue plusieurs tâches distinctes.
- **Violation du Principe de Responsabilité Unique** : La classe gère la validation, le calcul du score, l'envoi d'emails et la mise à jour de la base de données.
- **Couplage Fort** : La classe crée directement des instances d'autres classes (`EmailService`, `Database`), ce qui rend difficile le test unitaire.

Refactorer une Application Legacy

Code Après Refactoring :

```
public class ClientService {
    private Validator validator;
    private CreditScoreCalculator creditScoreCalculator;
    private EmailService emailService;
    private Database database;
    public ClientService(Validator validator, CreditScoreCalculator creditScoreCalculator,
                         EmailService emailService, Database database) {
        this.validator = validator;
        this.creditScoreCalculator = creditScoreCalculator;
        this.emailService = emailService;
        this.database = database;
    }
    public void processClientData(Client client) {
        validateClient(client);
        int creditScore = creditScoreCalculator.calculate(client);
        emailService.sendWelcomeEmail(client.getEmail());
        database.updateClientRecord(client);
    }
    private void validateClient(Client client) {
        if (!validator.isValid(client)) {
            throw new IllegalArgumentException("Données client invalides");
        }
```

Refactorer une Application Legacy

Améliorations Apportées :

- **Extraction de Méthodes** : Les différentes responsabilités sont maintenant dans des méthodes ou classes séparées.
- **Injection de Dépendances** : Les dépendances sont injectées via le constructeur, facilitant ainsi les tests unitaires.
- **Responsabilité Unique** : Chaque classe a une responsabilité clairement définie.

Savoir ce qu’on Appelle un Code Smell

Un code smell est un indicateur de problèmes potentiels dans le code source. Ce n'est pas nécessairement un bug, mais plutôt un symptôme de mauvaise conception ou de mauvaise implémentation qui peut rendre le code difficile à maintenir ou à faire évoluer.

Il existe plusieurs types de code smells, classés en différentes catégories. Nous allons détailler les principaux, avec des exemples concrets pour chacun.

Bloaters

Les **bloaters** sont des éléments du code qui ont "gonflé" au fil du temps, devenant trop grands ou complexes.

Savoir ce qu'on Appelle un Code Smell

Long Method (Méthode Longue)

Une méthode qui est trop longue et qui fait trop de choses, ce qui la rend difficile à comprendre et à maintenir.

Exemple de Code Avant Refactoring :

```
public void generateReport(List<Transaction> transactions) {
    double total = 0;
    for (Transaction t : transactions) {
        // Calcul du total
        total += t.getAmount();
        // Affichage des détails de la transaction
        System.out.println("Transaction ID: " + t.getId());
        System.out.println("Amount: " + t.getAmount());
        System.out.println("Date: " + t.getDate());
        // Vérification de fraudes potentielles
        if (t.isSuspicious()) {
            alertFraud(t);
        }
    }
    // Affichage du total
    System.out.println("Total Amount: " + total);
```

Savoir ce qu'on Appelle un Code Smell

Problèmes Identifiés :

- La méthode effectue plusieurs tâches : calcul du total, affichage des détails, vérification de fraudes, etc.
- Difficile à tester et à maintenir.

Exemple de Code Après Refactoring :

```
public void generateReport(List<Transaction> transactions) {
    double total = calculateTotal(transactions);
    displayTransactions(transactions);
    System.out.println("Total Amount: " + total);
}

private double calculateTotal(List<Transaction> transactions) {
    return transactions.stream()
            .mapToDouble(Transaction::getAmount)
            .sum();
}

private void displayTransactions(List<Transaction> transactions) {
    for (Transaction t : transactions) {
        displayTransactionDetails(t);
        checkForFraud(t);
    }
}

private void displayTransactionDetails(Transaction t) {
    System.out.println("Transaction ID: " + t.getId());
    System.out.println("Amount: " + t.getAmount());
```

Savoir ce qu'on Appelle un Code Smell

Améliorations Apportées :

- La méthode `generateReport` est maintenant plus concise et lisible.
- Les différentes tâches sont déléguées à des méthodes spécifiques.

Savoir ce qu'on Appelle un Code Smell

Large Class (Classe Trop Grande)

Une classe qui accumule trop de responsabilités, ce qui la rend complexe et difficile à maintenir.

Exemple de Code Avant Refactoring :

```
public class UserAccount {
    private String username;
    private String password;
    private List<Order> orders;
    private List<Message> messages;
    private Settings settings;

    public void addOrder(Order order) { /* ... */ }
    public void sendMessage(Message message) { /* ... */ }
    public void updateSettings(Settings settings) { /* ... */ }
    public void resetPassword() { /* ... */ }
    // Beaucoup d'autres méthodes
}
```

Savoir ce qu'on Appelle un Code Smell

Problèmes Identifiés :

- La classe gère les commandes, les messages, les paramètres, etc.
- Violation du principe de responsabilité unique.

Exemple de Code Après Refactoring :

- **Création de Classes Spécifiques :**

```
public class UserAccount {
    private String username;
    private String password;
    private Settings settings;

    public void resetPassword() { /* ... */ }
    public void updateSettings(Settings settings) { /* ... */ }
}

public class OrderService {
    private List<Order> orders;

    public void addOrder(Order order) { /* ... */ }
    // Autres méthodes liées aux commandes
}
```

Savoir ce qu'on Appelle un Code Smell

Améliorations Apportées :

- Chaque classe a une responsabilité claire.
- Le code est plus modulaire et plus facile à maintenir.

Savoir ce qu'on Appelle un Code Smell

Object-Oriented Abusers: Cette catégorie concerne les abus ou les mauvais usages des principes de la programmation orientée objet.

Switch Statements (Utilisation Excessive de Switch ou If)

L'utilisation répétée de structures conditionnelles pour contrôler le flux du programme en fonction du type ou de l'état d'un objet.

Exemple de Code Avant Refactoring :

```
public double calculateDiscount(Product product) {
    if (product.getType() == ProductType.ELECTRONICS) {
        return product.getPrice() * 0.1;
    } else if (product.getType() == ProductType.CLOTHING) {
        return product.getPrice() * 0.2;
    } else if (product.getType() == ProductType.FOOD) {
        return product.getPrice() * 0.05;
    } else {
        return 0;
    }
}
```

Savoir ce qu'on Appelle un Code Smell

Problèmes Identifiés :

- Difficile à maintenir lorsque de nouveaux types de produits sont ajoutés.
- Violation du principe ouvert/fermé.

Exemple de Code Après Refactoring :

- **Utilisation du Polymorphisme :**

```
public abstract class Product {
    protected double price;
    public abstract double calculateDiscount();
    public double getPrice() {
        return price;
    }
}

public class Electronics extends Product {
    public double calculateDiscount() {
        return price * 0.1;
    }
}

public class Clothing extends Product {
    public double calculateDiscount() {
        return price * 0.2;
```

Savoir ce qu'on Appelle un Code Smell

Améliorations Apportées :

- Ajout de nouveaux types de produits sans modifier le code existant.
- Le calcul de la remise est délégué aux sous-classes.

Savoir ce qu'on Appelle un Code Smell

Temporary Field (Champ Temporaire)

Description :

Des attributs d'une classe qui ne sont initialisés que dans certaines circonstances, ce qui peut rendre le code confus.

Exemple de Code Avant Refactoring :

```
public class ReportGenerator {
    private String title;
    private String content;
    private String footer;
    private Connection databaseConnection; // Utilisé uniquement pour certains rapports

    public ReportGenerator(String title, String content, String footer) {
        this.title = title;
        this.content = content;
        this.footer = footer;
    }

    public void generate() {
        // Génération du rapport
```

Savoir ce qu'on Appelle un Code Smell

Problèmes Identifiés :

- L'attribut `databaseConnection` n'est pas toujours pertinent.
- Rend la classe plus complexe qu'elle ne devrait l'être.

Exemple de Code Après Refactoring :

- **Extraction de Sous-Classes ou Utilisation de Patrons de Conception Appropriés :**

```
public class ReportGenerator {
    protected String title;
    protected String content;
    protected String footer;

    public ReportGenerator(String title, String content, String footer) {
        this.title = title;
        this.content = content;
        this.footer = footer;
    }

    public void generate() {
        // Génération du rapport standard
    }
}

public class DatabaseReportGenerator extends ReportGenerator {
    private Connection databaseConnection;

    public DatabaseReportGenerator(String title, String content, String footer, Connection databaseConnection) {
```

Savoir ce qu'on Appelle un Code Smell

Améliorations Apportées :

- Séparation des responsabilités.
- Le champ temporaire est déplacé vers une sous-classe où il est toujours pertinent.

Savoir ce qu'on Appelle un Code Smell

Change Preventers

Les **change preventers** sont des code smells qui rendent le code difficile à modifier ou à étendre.

Divergent Change (Changement Divergent)

Une classe qui doit être modifiée pour différentes raisons, à chaque fois qu'un type particulier de changement est nécessaire.

Exemple de Code Avant Refactoring :

```
public class NotificationService {
    public void sendEmailNotification(User user, String message) { /* ... */ }
    public void sendSMSNotification(User user, String message) { /* ... */ }
    public void sendPushNotification(User user, String message) { /* ... */ }
    // À chaque nouveau type de notification, la classe doit être modifiée.
}
```

Problèmes Identifiés :

- Violation du principe de responsabilité unique.
- Difficile à maintenir et à étendre.

Savoir ce qu'on Appelle un Code Smell

Exemple de Code Après Refactoring :

- **Utilisation du Patrons Stratégie ou Commande :**

```
public interface NotificationStrategy {
    void send(User user, String message);
}

public class EmailNotification implements NotificationStrategy {
    public void send(User user, String message) { /* ... */ }
}

public class SMSNotification implements NotificationStrategy {
    public void send(User user, String message) { /* ... */ }
}

public class PushNotification implements NotificationStrategy {
    public void send(User user, String message) { /* ... */ }
}

public class NotificationService {
    private NotificationStrategy strategy;

    public NotificationService(NotificationStrategy strategy) {
        this.strategy = strategy;
    }
```

Savoir ce qu’on Appelle un Code Smell

Améliorations Apportées :

- Ouvert pour extension, fermé pour modification.
- Facilite l'ajout de nouveaux types de notifications sans modifier le code existant.

Savoir ce qu'on Appelle un Code Smell

Shotgun Surgery Un changement mineur nécessite de modifier de nombreuses classes ou méthodes disséminées dans le code.

Exemple de Code Avant Refactoring :

- Supposons que le format de la date doit changer dans l'application, et que le format est utilisé directement dans plusieurs classes.

```
public class Order {
    public String getFormattedDate() {
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
        return sdf.format(orderDate);
    }
}

public class Invoice {
    public String getFormattedDate() {
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
        return sdf.format(invoiceDate);
    }
}
```

Savoir ce qu'on Appelle un Code Smell

Problèmes Identifiés :

- Duplications de code.
- Difficulté à effectuer des changements globaux.

Exemple de Code Après Refactoring :

- **Centralisation du Code Commun :**

```
public class DateUtil {
    private static final SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");

    public static String formatDate(Date date) {
        return sdf.format(date);
    }
}

public class Order {
    public String getFormattedDate() {
        return DateUtil.formatDate(orderDate);
    }
}
```

Savoir ce qu’on Appelle un Code Smell

Améliorations Apportées :

- Le format de la date est centralisé.
- Un changement du format n'affecte qu'une seule classe.

Savoir Refactorer les Principaux Code Smells

1. **Identifier le Code Smell :** Utiliser des outils d'analyse statique ou faire une revue de code pour repérer les problèmes.
2. **Comprendre le Code :** Avant de modifier, assurez-vous de bien comprendre le fonctionnement du code.
3. **Écrire des Tests :** Si des tests n'existent pas, en écrire pour couvrir les fonctionnalités existantes.
4. **Appliquer le Refactoring :** Utiliser les techniques appropriées pour améliorer le code.
5. **Vérifier les Tests :** S'assurer que les tests passent toujours après les modifications.

La Loi de Demeter

La Loi de Demeter, également connue sous le nom de "Principe du moindre savoir", stipule qu'une méthode d'un objet ne devrait interagir qu'avec :

- Ses propres méthodes.
- Ses attributs directs.
- Les objets créés par la méthode.
- Les paramètres de la méthode.

Violation de la Loi de Demeter :

Exemple de Code Avant Refactoring :

```
public class OrderService {
    public double getCustomerBalance(Order order) {
        return order.getCustomer().getAccount().getBalance();
    }
}
```

La Loi de Demeter

Problèmes Identifiés :

- L'accès en chaîne aux objets (`order.getCustomer().getAccount().getBalance()`) crée un couplage fort entre les classes.
- Si la structure interne change, le code doit être modifié.

Exemple de Code Après Refactoring :

- **Encapsulation des Détails Internes :**

```
public class Customer {
    private Account account;

    public double getBalance() {
        return account.getBalance();
    }
}

public class Order {
    private Customer customer;

    public double getCustomerBalance() {
        return customer.getBalance();
```

La Loi de Demeter

Améliorations Apportées :

- Réduction du couplage entre les classes.
- Les détails internes sont encapsulés, ce qui facilite les modifications futures.

Autre Exemple :

Avant Refactoring :

```
public class Car {
    private Engine engine;

    public void start() {
        engine.getFuelInjector().injectFuel();
        engine.getSparkPlug().ignite();
    }
}
```

La Loi de Demeter

Après Refactoring :

```
public class Engine {
    private FuelInjector fuelInjector;
    private SparkPlug sparkPlug;

    public void start() {
        fuelInjector.injectFuel();
        sparkPlug.ignite();
    }
}

public class Car {
    private Engine engine;

    public void start() {
        engine.start();
    }
}
```

La Loi de Demeter

Améliorations Apportées :

- La classe `Car` ne connaît plus les détails internes de `Engine`.
- Respect de la Loi de Demeter.

Avantages du Respect de la Loi de Demeter :

- **Faible Couplage :** Les classes sont moins dépendantes des structures internes des autres classes.
- **Facilité de Maintenance :** Les modifications internes d'une classe n'affectent pas les classes qui l'utilisent.
- **Meilleure Lisibilité :** Le code est plus clair et plus facile à comprendre.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Jour 5

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.