Fil de pensée

Propos introductif:

Objectif du Programme FizzForm

Le test d'entretien "GestForm" vise à développer un programme, nommé **FizzForm**, capable de générer une liste aléatoire de nombres entiers entre -1000 et 1000. Pour chaque nombre de la liste, le programme détermine si le nombre est divisible par 3, par 5, par les deux, ou par aucun des deux, et affiche respectivement "Geste", "Forme", "Gestform", ou le nombre lui-même.

Choix Technologiques

Pour mener à bien ce projet, plusieurs outils ont été soigneusement sélectionnés :

Le choix d'utiliser **TypeScript** plutôt que JavaScript pur s'inscrit dans une démarche visant à améliorer la maintenabilité et la robustesse du code de FizzForm. offrant ainsi plusieurs avantages significatifs :

- **Meilleure Visibilité des Types** : Grâce à l'annotation de type, les développeurs peuvent comprendre en un coup d'œil les types d'entrées et de sorties des fonctions, facilitant ainsi la lecture et la compréhension du code.
- **Détection Précoce des Erreurs** : TypeScript analyse le code statiquement et signale les erreurs potentielles lors de la compilation, bien avant l'exécution. Cela réduit les risques d'erreurs à l'exécution et améliore la qualité globale du logiciel.
- **Jest**, utilisé comme moteur de tests unitaires, facilite la rédaction de tests précis et la génération de rapports de couverture de code.
- **ts-node** offre la possibilité d'exécuter du code TypeScript directement, sans nécessiter une étape de compilation préalable en JavaScript, accélérant ainsi le cycle de développement.
- **Prettier & ESLint** assurent une base de code propre, formatée de manière cohérente, et suivant les meilleures pratiques de développement.
- **Stryker**, en tant que moteur de test de mutation, permet d'évaluer la qualité des tests unitaires en introduisant des mutations dans le code et en

vérifiant que les tests existants échouent en conséquence.

Modularité et Structure du Code

En l'absence de directives détaillées de la part d'un product owner, le programme est conçu pour être modulaire et adaptable. Par défaut, **FizzForm** génère une liste de 10 nombres aléatoires, un choix stocké dans une variable pour faciliter d'éventuelles modifications futures sans refonte du système. Cette décision stratégique permet une évolution aisée du programme en fonction des besoins.

Le code est organisé en deux fichiers principaux :

- `core.ts` : contient la logique métier du programme, définissant les opérations à effectuer sur chaque nombre de la liste.
- `main.ts` : sert de point d'entrée à l'application, appelant les fonctions métiers définies dans `core.ts` et affichant les résultats à l'utilisateur.

Les tests unitaires, situés dans `core.spec.ts` et `main.spec.ts`, garantissent le bon fonctionnement et la fiabilité du programme, en couvrant respectivement la logique métier et le flux d'exécution principal.

Nota Bene sur le Commentaire du Code:

Comme vous pourrez le constater à la lecture du code source de FizzForm, certaines parties peuvent sembler surcommentées. Cette approche a été intentionnellement adoptée dans le but de faciliter votre compréhension du cheminement logique et des décisions prises tout au long du développement. Il est important de souligner qu'en situation de travail en équipe, ma pratique habituelle en matière de commentaires de code est généralement plus mesurée. Je m'attache à fournir des annotations précises sur les types d'entrées et de sorties des fonctions, accompagnées d'une brève description de leur objectif. Des explications supplémentaires sont ajoutées uniquement en cas de nécessité, par exemple pour clarifier des blocs de code particulièrement complexes ou techniques. Cette méthode vise à maintenir un équilibre entre la lisibilité du code et la surabondance de commentaires, assurant ainsi une maintenance et une compréhension optimales du code pour l'ensemble de l'équipe.

I Développement Coeur du Métier : Fonction `analyzeNumber`

Au cœur du projet FizzForm, la fonction `analyzeNumber` est l'essentiel de la logique métier, déterminant comment chaque nombre de la liste générée est analysé et traité. Cette fonction est conçue pour évaluer si un nombre donné est divisible par 3, par 5, par les deux, ou par aucun des deux, et renvoie une chaîne de caractères appropriée ("Gestform", "Geste", "Forme", ou le nombre lui-même) en fonction du résultat de cette évaluation.

L'approche adoptée commence par examiner le cas où un nombre est divisible à la fois par 3 et par 5. Cette vérification est prioritaire car elle permet d'éviter des vérifications supplémentaires dans les cas où cette condition est vraie. En effet, si un nombre satisfait cette condition, il n'est pas nécessaire d'explorer les autres embranchements : c'est une optimisation logique, réduisant le nombre d'opérations effectuées pour chaque nombre analysé.

Les commits 3 à 6 reflètent un processus de développement itératif, mettant en lumière la réflexion derrière l'implémentation de cette fonction. Il est crucial de noter que cette séquence démontre non seulement l'ajout progressif de la logique métier mais aussi l'intégration des tests unitaires associés, assurant ainsi que chaque comportement attendu de la fonction `analyzeNumber` est rigoureusement validé. Les tests couvrent exhaustivement les scénarios possibles, garantissant une fiabilité et une robustesse élevées de la fonction.Cela permet également par ces tests de documenté le code.

II. Génération de la Liste de Nombres Aléatoires : Fonction `generateRandomList`

La capacité à générer une liste de nombres aléatoires constitue un pilier central de l'application FizzForm, facilitée par la fonction `generateRandomList`. Cette fonction incarne l'aspect dynamique et adaptable du programme, permettant la création de listes de tailles variées, avec des nombres s'étendant sur un large intervalle. Les variables `boundaryMin` et `boundaryMax` définissent respectivement les limites inférieure (-1000) et supérieure (1000) pour les nombres générés, établissant un cadre clair et modifiable pour la génération des nombres.

Flexibilité et Adaptabilité

L'introduction de `listSize`, `boundaryMin`, et `boundaryMax` sert plusieurs objectifs critiques. Premièrement, cela évite l'usage de nombres magiques à travers le code, rendant les modifications futures plus simples et plus intuitives. Par exemple, si un besoin émerge de modifier l'intervalle de génération des nombres ou la taille de la liste, ces ajustements peuvent être réalisés rapidement en modifiant les valeurs de ces variables sans nécessiter de fouiller dans la logique de la fonction. Cela augmente considérablement la maintenabilité du code.

Validation et Tests

Les commits 7 et 8 illustrent le développement minutieux de `generateRandomList` ainsi que l'intégration des tests unitaires conçus pour valider son comportement. Ces tests assurent que la fonction répond aux spécifications requises, comme la génération d'une liste de la taille demandée et la production de nombres dans l'intervalle spécifié. De plus, les tests vérifient la robustesse de la fonction face à des paramètres non conventionnels, tels que des tailles de liste négatives ou des intervalles mal définis où le minimum est supérieur au maximum.

III. Traitement de la Liste : Fonction `processList`

La fonction `processList` représente une étape cruciale dans l'application FizzForm, orchestrant la transformation de la liste de nombres aléatoires générés en une série de résultats conformément aux règles définies. Cette fonction incarne le pont entre la génération des données et leur interprétation finale, illustrant ainsi l'intégration parfaite de la logique métier au sein de l'application.

Fonctionnalité et Flexibilité

En appliquant la fonction `analyzeNumber` à chaque élément de la liste fournie, `processList` convertit chaque nombre en une chaîne de caractères descriptive ("Gestform", "Geste", "Forme", ou la valeur numérique elle-même sous forme de chaîne). L'utilisation de la méthode `map` garantit que cette transformation est réalisée de manière efficace et élégante, produisant une nouvelle liste de chaînes qui reflète directement le résultat de l'analyse.

Design Intentionnel

La conception de `processList` souligne un choix délibéré de maintenir une séparation claire entre la génération des données (via `generateRandomList`) et leur traitement (via `processList`). Cette séparation des préoccupations permet non seulement une plus grande clarté et maintenabilité du code, mais offre également la flexibilité nécessaire pour adapter ou étendre l'application à l'avenir. Les variables telles que `listSize`, `boundaryMin` et `boundaryMax` jouent un rôle essentiel dans cette architecture en évitant l'utilisation de nombres magiques et en facilitant les ajustements aux besoins changeants sans intervention profonde dans la logique du programme.

Validation et Tests

Les commits 9 et 10 mettent l'accent sur la validation de la fonction `processList` à travers un ensemble de tests unitaires rigoureux. Ces tests vérifient non seulement que `processList` produit une liste de la taille attendue et respecte les plages de valeurs spécifiées, mais ils examinent également des cas particuliers tels que la gestion de tailles de liste négatives ou nulles, ainsi que des situations où les valeurs minimales et maximales sont mal configurées.

IV. Test d'Intégration pour `main.ts`

Maintenant que le cœur métier du programme FizzForm est achevé et que les tests unitaires indiquent une couverture de code de 100% grâce à Jest, il est temps de porter notre attention sur le module `main.ts`. Bien que l'ajout de tests unitaires pour `main.ts` puisse sembler superflu, étant donné que ce module se limite principalement à appeler des fonctions métier et à afficher leurs résultats, un test d'intégration s'avère néanmoins essentiel pour s'assurer de l'intégrité et du bon fonctionnement de l'ensemble du système.

Le test d'intégration pour `main.ts` a été conçu pour valider que les fonctions `generateRandomList` et `processList` sont correctement invoquées et que leurs résultats sont appropriés affichés à l'utilisateur. Grâce à l'utilisation de `jest.mock`, nous remplaçons les fonctions importées de `core` par des versions simulées (`mocks`), ce qui nous permet de contrôler les valeurs retournées par ces fonctions et de vérifier les interactions avec elles sans dépendre de leur implémentation interne.

Pour le test, nous définissons une liste de nombres aléatoires simulés à retourner par `generateRandomList` et une liste de résultats attendus à retourner par `processList`. L'exécution de `main.ts` devrait entraîner l'appel de ces fonctions

mockées avec les arguments attendus et l'affichage des résultats. En remplaçant `console.log` par une version mockée, nous pouvons également vérifier que les bonnes valeurs sont affichées à l'utilisateur.

Ce test d'intégration, réalisé au cours des commits 11 et suivants, confirme non seulement que `main.ts` fonctionne comme prévu en intégrant les composants métiers, mais aussi valide l'interaction entre les différentes parties du programme

V. Tests de Mutation avec Stryker

Après avoir atteint une couverture de tests de 100% avec Jest, ce qui suggère théoriquement que chaque ligne de code est vérifiée par au moins un test, il est crucial de s'assurer de la pertinence et de l'efficacité de ces tests. C'est ici qu'interviennent les tests de mutation, et plus spécifiquement l'utilisation de **Stryker**, un outil de test de mutation puissant et flexible.

Fonctionnement des Tests de Mutation

Les tests de mutation fonctionnent en apportant de petites modifications (mutations) au code source pour créer des "mutants". Ces mutants sont ensuite soumis aux tests unitaires existants. Si un test échoue, cela signifie qu'il a "tué" le mutant, ce qui est le résultat souhaité car il démontre que le test est capable de détecter une anomalie introduite dans le code. Inversement, si un mutant survit, cela révèle une faiblesse dans la suite de tests, indiquant que certains comportements du code ne sont pas suffisamment testés.

Résultats avec Stryker

Dans le cadre du projet FizzForm, l'application des tests de mutation avec Stryker a donné un score de **86,85%**. Sur **40 mutants générés** par l'outil, **6 mutants ont survécu**. Ce résultat est significatif car, malgré une couverture de tests unitaires de 100% indiquée par Jest, le score de mutation inférieur à 100% met en lumière des scénarios ou des chemins dans le code qui ne sont pas entièrement validés par les tests actuels.

Analyse et Actions

La survie de 6 mutants suggère des pistes d'amélioration pour la suite de tests. Pour chaque mutant survivant, il convient d'examiner le code correspondant et de comprendre pourquoi les tests existants n'ont pas détecté la modification.

Cela peut impliquer d'ajouter de nouveaux tests ou de renforcer ceux qui existent déjà pour couvrir des cas de bord ou des comportements spécifiques non pris en compte précédemment.

5.1 Élimination du Mutant : Cas des Bornes Inversées

Un des mutants survivants identifiés par Stryker concerne le cas où les bornes minimales et maximales sont inversées dans la fonction `generateRandomList`. Le mutant en question modifie la chaîne de caractères dans l'appel de `console.log` qui avertit de l'inversion des bornes :

- console.log("les valeurs de min et max sont inversées");
- console.log("");

Ce mutant a survécu car, bien que le test vérifie que `console.log` soit appelé avec le message attendu lorsque `min` est supérieur à `max`, il ne détecte pas la modification spécifique de la chaîne de caractères. Cela révèle une faiblesse dans mon test : il ne s'assure pas que le message d'erreur spécifique est correct, seulement qu'un appel à `console.log` a lieu.

Pour "tuer" ce mutant, il faut renforcer ce test en s'assurant que le message spécifique concernant les bornes inversées est bien vérifié. C'est l'action menée dans le commit 12.

Suite à l'ajustement apporté pour éliminer le mutant concernant les bornes inversées dans `generateRandomList`, un nouveau test donne 5 mutant et un sore de 88,37 %.

5.2 Correction du Mutant : Décalage de 2 Unités dans les Nombres Générés

Un autre mutant identifié lors des tests de mutation avec Stryker concernait un décalage de deux unités dans la plage de nombres générés par la fonction `generateRandomList`. Ce mutant modifiait l'opérateur arithmétique utilisé pour calculer la plage de nombres aléatoires, réduisant effectivement la plage de deux unités :

- () ⇒ Math.floor(Math.random() * (max - min + 1)) + min
- () ⇒ Math.floor(Math.random() * (max - min - 1)) + min

Cette mutation a révélé une faiblesse dans la suite de tests : bien qu'ils vérifient que les nombres générés étaient dans la plage spécifiée, ils ne contrôlent pas

suffisamment que la plage exacte était respectée, permettant ainsi à ce mutant de survivre.

Pour remédier à cela, j'ai renforcé le test pour s'assurer non seulement que les nombres générés étaient compris dans la plage spécifiée, mais aussi que la plage de génération était exactement conforme aux spécifications. Ceci implique de vérifier que les valeurs minimales et maximales possibles sont effectivement générées par la fonction. En ajustant le test pour qu'il échoue en l'absence d'exactitude dans la plage de génération, j'ai pu "tuer" ce mutant. (... désolé Wolverine 🤣)

Après cette correction apportée dans le commit 13, seulement 2 mutants ont survécu, portant le score de mutation à 95,35%

5.3 Correction du Mutant : Gestion du Cas d'Égalité entre les Bornes Min et Max

Un aspect initialement négligé dans mon implémentation de `generateRandomList` était la gestion du cas où les bornes minimales et maximales sont égales. Cela a été mis en évidence par un mutant survivant lors des tests de mutation avec Stryker, signalant l'absence d'une vérification spécifique pour ce scénario. Reconnaissant l'importance de traiter tous les cas d'usage possibles pour assurer la robustesse de l'application, une correction a été apportée dans le commit 14 pour adresser cette lacune.

La correction impliquait l'ajout d'une condition supplémentaire dans `generateRandomList` pour vérifier si `min` est égal à `max`. Si cette condition est vraie, la fonction retourne immédiatement un tableau vide et enregistre un message d'erreur spécifique via `console.log`, informant de l'impossibilité de générer une liste dans ces conditions :

Pour valider cette correction, un test unitaire dédié à la manière de celui verifiant que les bornes `min` et `max` sont inversées, a été ajouté, vérifiant que :

1. Un tableau vide est retourné lorsque `min` et `max` sont égaux.
2. Le message d'erreur approprié est enregistré dans la console, confirmant que la condition d'égalité est correctement gérée.

Après l'introduction de cette correction et la validation via le test unitaire correspondant, j'ai réussi à éliminer tous les mutants

survivants, atteignant ainsi un score de mutation de 100%. Ce résultat illustre non seulement l'exhaustivité de la suite de tests à couvrir tous les aspects du comportement de `generateRandomList`, mais confirme également l'efficacité de l'approche de tests de mutation pour améliorer continuellement la qualité du code. Cette ultime correction marque un point crucial dans le développement de FizzForm, garantissant que le programme est robuste et fiable face à une variété de conditions d'entrée.

Conclusion et Perspectives

Dans le cadre de cet exercice, atteindre un score de mutation de 100% a été une démarche à la fois enrichissante et instructive, principalement en raison de la simplicité et de la concision du code métier de FizzForm, qui facilitait grandement l'effort de couverture exhaustive par les tests. Cet objectif, bien qu'ambitieux, a permis d'explorer en profondeur les capacités des tests de mutation à révéler des faiblesses subtiles dans la suite de tests, renforçant ainsi la robustesse globale du programme.

Il est important de souligner, cependant, que dans le cadre de projets réels, où la logique métier peut être significativement plus complexe et étendue, viser un score de mutation de 100% peut s'avérer extrêmement difficile, voire contre-productif. Dans de tels contextes, l'effort nécessaire pour traquer et éliminer les derniers pourcentages de mutants survivants peut devenir disproportionné par rapport aux bénéfices réels en termes d'amélioration de la qualité du code. De plus, il peut y avoir des cas où atteindre les derniers points de pourcentage ne serait pas pertinent ou justifié, à moins que des besoins spécifiques du projet ou des engagements contractuels n'exigent une telle exhaustivité.

En résumé, bien que la quête d'un score de mutation parfait puisse être motivée par le plaisir de l'exercice et la volonté d'affiner notre approche de test, il est crucial d'évaluer judicieusement l'investissement nécessaire à l'aune des gains de qualité et de fiabilité attendus, surtout dans le cadre de projets à grande échelle ou à haute complexité.