Journal de Bord du Refactoring de TennisGame

Introduction

Nous avons été confrontés au projet de refactoriser un morceau de code pour le jeu de tennis, `tennis1.py`, avec pour objectif d'en améliorer la lisibilité, la maintenabilité et la flexibilité. Le code original souffrait de plusieurs maux : utilisation de nombres magiques, chaînes de caractères littérales, et une duplication de code importante. Notre mission était de clarifier le code et de le préparer pour l'ajout facile de nouvelles fonctionnalités, comme la prise en charge de plusieurs langues.

Étape 1 : Analyse du code original

commit:

- 027e2a93ff3f308e0be0701e924381034f81d99a
- d31c5f89a829cc1719573cabcbd1d33a93c9dd5b

Le début de notre travail a consisté à analyser le code original pour identifier les principaux problèmes :

- Des variables et une logique de score éparpillées et mal structurées
- Une utilisation excessive de conditions imbriquées
- Une duplication significative de code, notamment dans la gestion des scores
- L'emploi de nombres magiques et de chaînes de caractères en dur

Odeur	Gravité	Récurrence	Risque
Noms des joueurs dans des variables séparées	Moyenne	Courante	4
Points des joueurs dans des variables séparées	Moyenne	Courante	4
Méthode score() trop longue et difficile à comprendre	Forte	Rare	5
Utilisation de valeurs magiques pour les points	Moyenne	Fréquente	3
Duplication de code dans la méthode score()	Moyenne	Courante	4
Strings non stockées dans des variables	Faible	Fréquente	2
Méthode score() pourrait utiliser des dictionnaires	Faible	Fréquente	2
Méthode score() pourrait être décomposée en méthodes plus petites	Forte	Courante	5
Switch Statements	Moyenne	Fréquente	3
Data Clumps	Moyenne	Courante	4
Primitive Obsession	Moyenne	Courante	4
Feature Envy	Moyenne	Courante	4
Inappropriate Intimacy	Moyenne	Courante	4
Missing Encapsulation	Moyenne	Courante	4
Speculative Generality	Faible	Rare	2
Hard-coded Logic	Forte	Rare	5

Étape 2 : Planification du refactoring

Suite à cette analyse, nous avons planifié nos actions de refactoring :

- Centraliser la gestion des joueurs et des scores au moyen de structures de données appropriées.
- Remplacer les nombres magiques et les chaînes littérales par des constantes et des énumérations pour plus de clarté.
- Simplifier la logique conditionnelle grâce à des méthodes plus spécifiques.

- Introduire un système de traduction pour faciliter la gestion des différentes langues.

Étape 3 : Refactoring

Introduction de TennisTranslation

Notre première grande modification a été l'ajout de la classe `TennisTranslation`, qui gère la traduction des termes de score dans différentes langues, rendant le code prêt pour l'internationalisation et améliorant sa flexibilité.

Refonte de TennisGame

commit:

- 65d008f10651331ac804148313e01c0521362f22

La classe `TennisGame` a été entièrement remaniée pour utiliser un dictionnaire pour les scores des joueurs, ce qui a rendu le code plus intuitif et a réduit les risques d'erreurs lors de la manipulation des scores.

Simplification de la logique de score

Nous avons décomposé la logique complexe de détermination du score en plusieurs méthodes plus petites et plus claires, comme `_is_early_game_all`, `_is_deuce`, et `_get_regular_score`. Cela a considérablement simplifié la compréhension du flux logique.

Étape 4 : Tests et validation

commit:

- 5bfcbd6230e02d3d86f345aedc029f28d7f50164

Après le refactoring, les tests ont été cruciaux pour valider nos changements. Nous avons utilisé `golden_master_test.py` pour nous assurer que le comportement du jeu restait identique après nos modifications.

Golden Master Testing

commit:

- 04e6de27ec4ca3b8dee5e07fd10665a4016852d3
- e7231e95e624c94a451af50a42ba9660e77c76bd

Le test Golden Master a comparé les résultats des scores entre l'ancienne et la nouvelle version pour de nombreux scénarios de jeu, ce qui nous a permis d'identifier et de résoudre rapidement les divergences.

Test d'indépendance du nom des joueurs

Un test supplémentaire a vérifié que les scores restent cohérents quel que soit le nom des joueurs. Ce test a souligné l'importance d'une gestion adéquate des noms dans notre version refactorisée.

Pourquoi avons-nous réalisé ces refactorings ?

Les refactorings réalisés abordent non seulement les points spécifiques mentionnés dans les TODO de la version originale, mais ils visent également à améliorer la clarté, la flexibilité, et la maintenabilité du code. En adoptant ces changements, le code devient plus aligné avec les principes de bonne pratique en développement logiciel, facilitant ainsi sa compréhension, son extension, et sa maintenance pour les développeurs actuels et futurs.

Ce que nous avons décidé de faire :

1. Centralisation des Chaînes de Caractères pour Faciliter les Traductions

Pourquoi ? La centralisation des chaînes dans `TennisTranslation` répond au TODO sur l'utilisation de chaînes littérales. Cela évite les chaînes en dur, facilitant les modifications et l'ajout de traductions.

2. Utilisation d'une Structure de Données pour les Scores

Pourquoi ? En passant à `self.player_scores`, on répond au TODO suggérant l'utilisation d'une structure de données pour les joueurs et leurs points. Cela améliore la clarté, réduit la complexité et facilite la gestion des scores.

3. Réduction de la Duplication du Code

Pourquoi ? En extrayant les fonctionnalités répétitives comme `_get_score_difference` et `_get_lead_player`, on répond au TODO sur la duplication de code. Cette approche diminue les risques d'erreurs et simplifie les modifications futures.

4. Remplacement des Nombres Magiques par des Constantes

Pourquoi ? Les modifications apportées utilisent des méthodes et des structures de données qui éliminent l'usage de nombres magiques (comme les scores 0, 1, 2, 3 transformés en termes de tennis dans `TennisTranslation`). Ceci répond au TODO sur les nombres magiques, rendant le code plus lisible et maintenable.

5. Simplification des Blocs Conditionnels

Pourquoi ? En réorganisant la logique de détermination des scores et des situations de jeu (comme le "Deuce" ou "Advantage"), on aborde le TODO concernant la complexité des méthodes. Cela simplifie la compréhension et la maintenance du code en décomposant la logique complexe en sous-parties plus gérables.