Monica MORTELETTE
Lucile PERBET

Rapport
PLANNING POKER

Présentation du projet

Notre projet consiste en une application de planning poker implémentée en Python avec une interface graphique utilisant Tkinter. L'objectif est de permettre à une équipe de développement de faire des estimations sur les tâches du backlog de manière collaborative.

Règle du jeu

Pour pouvoir jouer à une partie de planning poker, il faut commencer par entrer le nombre de joueurs puis entrer le pseudo de chacun des joueurs. Une fois le mode choisi, chaque joueur vote. Les règles vont ensuite dépendre du mode de jeu sélectionné.
Etant donné que notre jeu se fait en local et que les participants seront dans la même pièce, il n'y a pas de chat, mais il est possible de prendre des notes dans la zone bloc note dédiée à cet effet.

Les modules nécessaires

Une partie des modules devrait déjà être installée sur votre ordinateur, mais si ce n'est pas le cas, voici lignes de commande pour les installer à partir d'un terminal. Les modules nécessaires sont également présents dans le readme.md du github.

- **tkinter :** pip install tkinter
- **json** : pip install json
- **ctypes** : pip install ctypes
- **os** : pip install os
- **collections** : pip install collections
- **statistics** : pip install statistics

Choix des patterns

Singleton Pattern

La classe Application est conçue pour être la classe principale (classe mère) pour l'ensemble du code. Comme elle gère l'interface utilisateur avec la saisie du nombre de joueurs, des pseudos et du mode de jeu, il nous semble important de s'assurer qu'une seule instance de cette classe soit en cours d'exécution dans toute l'application. Nous utilisons des sous-classes pour créer et/ou modifier les objets.

Decorator Pattern

Dans les tests unitaires, les décorateurs @patch et @classmethod sont utilisés afin d'éviter de déclarer des fonctions. Nous les initions une fois puis nous appelons cette

fonction autant de fois que nécessaire. Le décorateur @patch sert à remplacer les appels de certaines fonctions (Par exemple, messagebox.showinfo) pendant les tests.

Factory Method Pattern

La mise en place d'un pattern Factory Method est utile dans le développement d'une application de planning poker. En effet, puisqu'il permet de fournir une interface commune pour les différents modes de jeu tout en gardant la possibilité que chaque classe garde des spécificités pour certaines méthodes notamment celle de calcul.

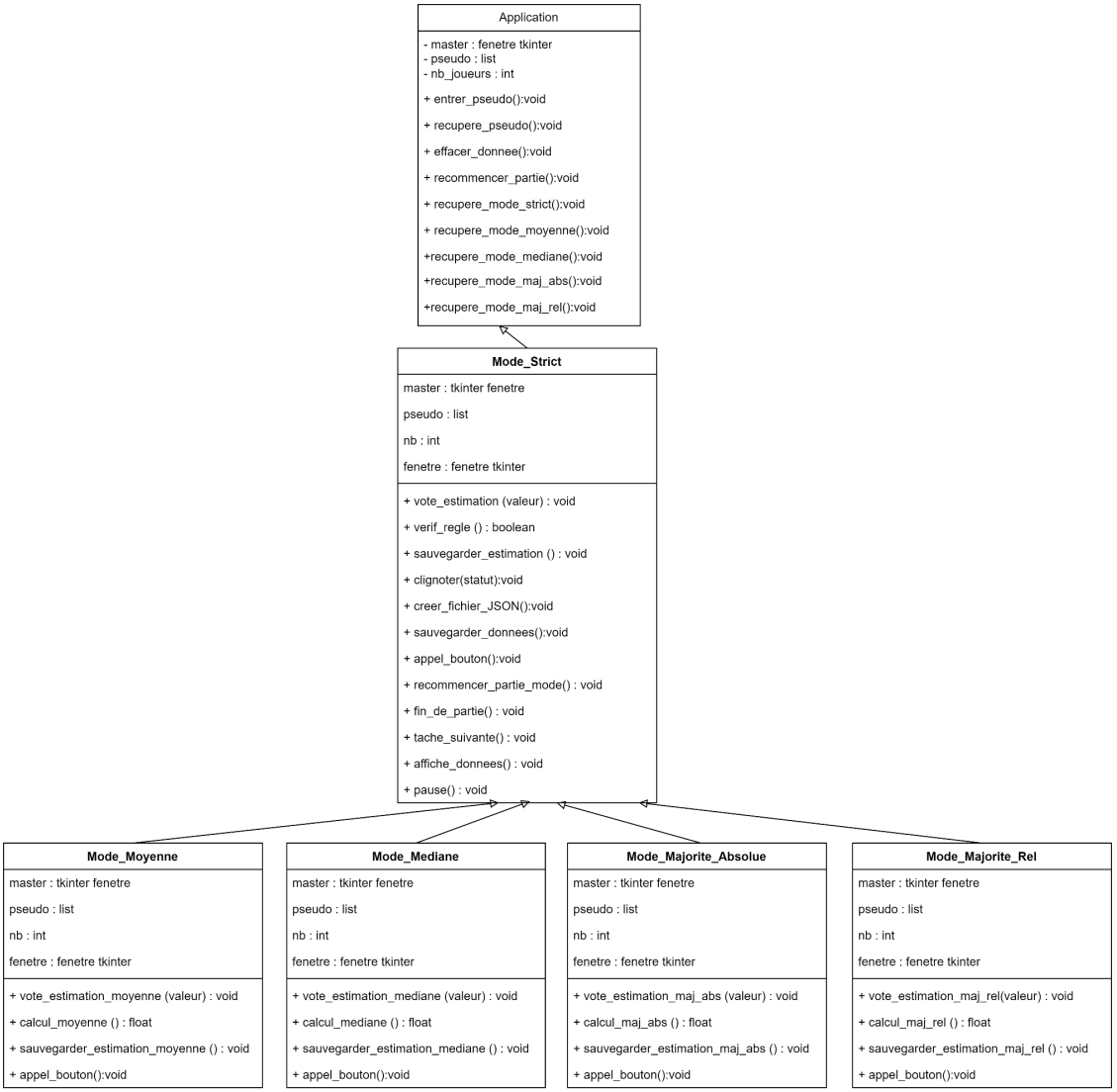
Ainsi, dans le code, il y aurait une classe abstraite contenant une interface commune pour tous les modes de jeu et les méthodes pour calculer les estimations. Suite à cela, il y aurait des classes concrètes héritant de la classe abstraite. Une classe concrète pour chaque mode de jeu qui redéfinit sa propre méthode pour calculer les estimations. Pour finir, il y aurait, une classe factory qui définirait des méthodes qui retourneraient une instance des classes concrètes définis précédemment.

Choix techniques

Architecture des classes

Nous avons opté pour une architecture orientée objet pour favoriser la réutilisabilité du code. La structure est divisée en six classes. Elles seront explicitées plus tard dans le rapport. Tout d'abord voici comment sont agencées les classes et les méthodes qu'elles contiennent.

Diagramme de classe



Sur ce diagramme, il est facile de remarquer que l'architecture des classes dépend de la relation d'héritage. Nous avons décidé de mettre en place l'héritage puisque cette approche permet de partager des fonctionnalités communes tout en laissant la possibilité de garder des spécificités pour les différents modes de jeu.

Classes

Dans le code, nous avons au total 6 classes qui sont reliées par des liens d'héritages. Par la suite, l'utilité et les spécificités de chaque sont détaillées.

Application (mère)

La première classe et étant la classe mère, s'appelle Application. Cette classe gère toute la partie interface de l'application hormis la gestion des différents modes de jeu. Elle

comprend les zones de saisie permettant d'entrer le nombre de joueurs et les pseudos, de sauvegarder ces informations. Mais également de choisir les modes de jeu.

Par ailleurs, cette classe définit une méthode qui est de recommencer une partie. Ainsi, un bouton est présent sur l'interface à différents endroits de la partie comme lors du choix des pseudos ou lors du vote des joueurs.

Cependant, en fonction de la page sur laquelle se situe le bouton, le fonctionnement de la méthode *recommencer_partie* n'est pas exactement la même. En effet, dans la classe Application, la méthode se contente d'effacer les données du JSON et de lancer une nouvelle instance de la classe. Alors que dans la classe *Mode_Strict*, la méthode *recommencer_partie_mode* appel la méthode de la classe mère, mais également appel la méthode *tache_suivante* qui a pour but de réinitialiser l'interface de vote à son état d'origine, c'est-à-dire sans modification de la part des joueurs.

Enfin, cette classe vérifie que des erreurs ne soient pas commises par les joueurs avec l'entrée du nombre de joueurs par exemple. En effet, les utilisateurs sont obligés d'entrer un entier qui soit compris entre 1 et 6 sinon ils recommencent.

Mode_Strict

Cette classe hérite de la classe Application. Elle définit toute l'interface du jeu en mode strict ainsi que les méthodes nécessaires pendant le déroulement d'une partie comme par exemple la sauvegarde des tâches du backlog dans le JSON ainsi que l'estimation votée. Elle a également la possibilité de recommencer la partie, ce qui a pour conséquence de tous réinitialiser les données du JSON, le nombre de joueurs, les pseudos et le mode de la partie. Elle permet aussi de mettre fin à la partie et de visualiser les données qui ont été enregistrées dans le JSON.

Cette classe reprend les informations récoltées grâce à la classe mère comme la liste des pseudos et le nombre de joueurs.

Pour l'enregistrement du vote dans le JSON, chaque vote des joueurs est enregistré dans une liste. Uniquement, la valeur pause n'est pas ajoutée à la liste puisque cette dernière met le jeu en pause et n'est pas une estimation. Une fois que tous les joueurs ont voté, alors la méthode pour vérifier la règle stricte est appelée si c'est respectée la valeur finale est enregistrée dans le JSON sinon la liste qui contient les estimations est remise à vide et les joueurs doivent recommencer.

Les quatre classes suivantes héritent chacune de la classe *Mode_Strict*. En effet, comme chaque mode de jeu à une interface similaire avec les mêmes fonctionnalités. Toutes ont été définies dans *Mode_Strict* pour éviter de réécrire les mêmes fonctions pour chaque classe. Cependant, il y a tout de même une différence qui réside dans le calcul de l'estimation. Ainsi, chaque classe possède sa propre méthode de calcul de l'estimation et de sauvegarde dans le JSON.

Pour le calcul des estimations pour les autres modes de jeu, le principe est le même. Les votes sont enregistrés dans une liste puis la règle de calcul dépend ensuite du mode de jeu. La différence avec la règle stricte est que les joueurs n'ont pas besoin de recommencer.

Mode_Moyenne

La spécificité de la classe réside dans la méthode *vote_estimation_moyenne* qui calcul la moyenne des estimations. Si la valeur *Ne Sait Pas* est sélectionnée par un joueur alors elle n'est pas enregistrée dans la liste des estimations.

Mode_Mediane

La spécificité de cette classe réside dans la méthode *vote_estimation_mediane* qui calcul la médiane des estimations. Elle a la même gestion de la valeur *Ne Sait Pas* que pour la classe moyenne.

Mode_Majorite_Absolue

La spécificité de cette classe réside dans la méthode *vote_estimation_maj_abs*. Elle a la même gestion de la valeur *Ne Sait Pas* que pour les classes précédentes.

Mode_Majorite_Relative

La spécificité de cette classe réside dans la méthode vote_estimation_maj_rel. Elle a la même gestion de la valeur Ne Sait Pas Pas que pour les classes précédentes.

Langage

Au départ, nous avions choisi d'effectuer le code en JS, HTML et CSS pensant que nous aurions plus de facilité pour implémenter l'interface. Nous avons vite changé d'avis et nous nous sommes orienté pour du Python, en effet ce langage était pour nous deux celui où nous sommes le plus à l'aise. Par ailleurs, Python présente un large choix de bibliothèques pour le développement d'interfaces graphiques. Nous avons choisi Tkinter puisque cette librairie possède une forte documentation sur Internet et elle assez simple d'utilisation.

Documentation

Pour chaque fichier de code, une documentation a été générée. Cela nous a permis de s'organiser et de suivre correctement l'avancement du code pour le reprendre plus facilement après. La documentation est structurée de manière à refléter l'architecture du projet. Chaque classe, méthode et fonction est documentée individuellement avec des balises Doxygen d'une part et avec des commentaires classiques d'autre part.

De plus, la sortie souhaitée avec Doxygen est sous format HTML. On y retrouve notamment les informations des différentes balises ainsi que les graphiques générés pour les fonctions.

Mise en place de l'intégration continue

Pour l'intégration continue nous avons fait le choix d'utiliser GitHub puisque c'est l'outil que nous connaissons le mieux parmi les différents outils d'intégration continue existant. A chaque nouvel ajout de fonctionnalités dans le code, la nouvelle version était poussée sur la branche principale du répertoire Git.

Les tests unitaires ont été écrits en même temps que le code. Ainsi, à chaque nouvelle version du code principal (ApplicationPoker.py) les tests étaient lancés.

Tests unitaires

Il y a une classe test pour chaque classe implémentée dans l'application. Dans les 6 classes tests, on y trouve un test Application, un test ModeStrict, un test ModeMediane, un test ModeMoyenne, un test ModeMajoriteAbsolue et un test ModeMajoriteRelative. Chaque test comprend différentes fonctions qui permettent de vérifier le bon fonctionnement du code.

TestApplication :

- test_entrer_pseudos_valid : tester si lors de la saisie d'un entier, on peut continuer sur l'application
- test_entrer_pseudos_invalid : tester si lors de la saisie de caractères (hors entiers), l'application s'arrête

TestModeStrict :

- test_sauvegarder_donnees : tester si les données entrées sont bien enregistrées dans le JSON
- test_recommencer_partie_mode : tester si les données sont supprimées quand on clique sur le bouton "recommencer partie"

TestModeMoyenne :

- test_vote_estimation_moyenne : tester si l'estimation est récupérée
- test_calcul_moyenne : tester si la moyenne est calculée correctement
- test_sauvegarder_estimation_moyenne : tester si la moyenne est bien enregistrée dans le JSON

TestModeMediane :

- test_vote_estimation_mediane : tester si l'estimation est récupérée
- test_calcul_mediane : tester si la médiane est calculée correctement
- test_sauvegarder_estimation_mediane : tester si la médiane est bien enregistrée dans le JSON

TestModeMajoriteAbsolue :

- test_vote_estimation_majorite_abs : tester si l'estimation est récupérée
- test_calcul_majorite_abs : tester si la majorité absolue est calculée correctement
- test_sauvegarder_estimation_majorite_abs : tester si la majorité absolue est bien enregistrée dans le JSON

TestModeMajoriteRelative :

- test_vote_estimation_majorite_relative : tester si l'estimation est récupérée
- test_calcul_majorite_rel : tester si la majorité relative est calculée correctement
- test_sauvegarder_estimation_majorite_rel : tester si la majorité absolue est bien enregistrée dans le JSON

Génération de la documentation

Pour générer la documentation, nous avons utilisé Doxygen. Chaque méthode de la classe est commentée avec les balises de Doxygen et explique brièvement ce qu'elle fait et à quoi elle sert. Les balises que nous avons principalement utilisées sont les suivantes :

- brief
- param
- return
- class

De plus, pratiquement, chaque ligne du code est commentée pour expliquer ce que nous avons essayé de faire à chaque étape. Cela s'applique pour le fichier *ApplicationPocker.py* et tous les autres fichiers contenant du code.

Par ailleurs, nous avons essayé de créer un workflow pour que la documentation soit générée automatiquement. Malheureusement, le workflow se marque en fail.

Concernant la configuration du fichier doxyfile, nous avons repris celui sur Moodle. Il est possible de le voir sur GIT dans le fichier nommé *config*.

Ajouts possibles ou que l'on aurait pu ajouter

Durant le projet, nous avons pensé à ajouter des fonctionnalités, mais soit nous n'avons pas eu le temps soit nous n'avons pas eu tous les moyens pour pleinement les effectuer. Cependant, il est possible de voir certaines fonctions qui pourraient être ajoutées au code dans le fichier python suivant Fonctionnalites_supplementaires.py, notamment le compte à rebours et la classe pour avoir des cartes jouables. Ci-dessous, voici l'ensemble des améliorations pour notre application :

- Un compte à rebours qui à la fin de temps annonce aux joueurs qu'il est temps de voter.
- Restreindre les pseudos en termes de longueur et de caractères possibles.
- Ajouter la possibilité de jouer à distance avec notamment une zone de chat pour que les joueurs puissent communiquer.

- La possibilité de jouer avec des cartes directement au lieu de cliquer sur les boutons.
- Faire en sorte que peu importe le mode, le premier tour de vote soit en mode strict.