TP 2 — Bulls, cows… and unit tests

On reprend le code source du TP 1 *Bulls and Cows*. Placez-vous dans le répertoire de votre projet et vérifier l'état de votre dépôt Git avec la commande `git status`. Vérifiez également l'historique avec `git log`. On y retrouve les commits et fusions de branches réalisées lors du TP 1.

En particulier, on retrouve trace des tests réalisés. En principe, votre branche `dev` et votre branche `master` sont au même niveau. Si ce n'est pas le cas, cela signifie que vous avez encore des fonctionnalités en cours de développement [1].

Dans ce TP, nous allons retravailler les tests, programmés dans le fichier `partieBC_tests.py`. L'objectif est d'améliorer la robustesse des tests effectués, en utilisant le framework `unittest`.

Travail à rendre

Vous devez déposer sur EPREL les fichiers `partieBC.py`, `partieBC_tests.py` et un troisième fichier `gitlog.txt` que l'on construira à la fin du TP.

Exercice 1 (Mise en place).

1) Placez-vous sur la branche Git `dev` et créez une sous-branche `refonte_tests_unitaires`. On travaillera exclusivement sur cette branche.

2) Videz le contenu du fichier `partieBC_tests.py` pour reprendre sa conception à partir du début.

Retrouver le fichier `partieBC_tests.py`

On retrouve ici un des intérêts majeurs de Git. Nous venons de supprimer le contenu du ficher `partieBC_tests.py` sur la branche `refonte_tests_unitaires`. Mais sur les branches `dev` et `master`, la version antérieure est toujours présente et n'est pas perdue.

On créera une sous-branche de `refonte_tests_unitaires` pour chaque fonction du fichier `partieBC.py` qui est testée. Sur chaque branche, on créera autant de commits que nécessaire. À chaque fois qu'un test est réussi, on fusionne la sous-branche de test dans la branche `refonte_tests_unitaires`.

Si un test unitaire échoue, cela peut signifier deux choses :

1) le test unitaire lui-même contient un bug.

2) le bug se situe dans le code de `partieBC.py`.

La première situation ne devrait pas arriver car le code des tests unitaire *doit être structurellement très simple*. Si l'on détecte un bug grâce aux tests unitaire, *on le corrige immédiatement*. Pour ce faire, on crée un commit sur la branche courante, puis on *crée une nouvelle branche* pour débuguer. Lorsque le débugage est terminée, on revient sur la branche de tests unitaires, puis on reprend.

1. Ou que vous n'avez pas géré votre dépôt correctement, auquel cas sollicitez votre enseignant·e

Le *workflow* en situation de débugage

Il est important, à ce stade, d'avoir une bonne visibilité de *l'arbre des branches* Git. On suppose que l'on vient de détecter un bug dans la fonction `histo(configuration1, configuration2)`. On doit alors débuguer cette fonction, et on a la structure Git suivante :

- La branche `master` contient le code stable.
- La branche `dev` contient le code en développement.
- La branche `refonte_tests_unitaires` contient le code des tests unitaires qui ont été réussis.
- La branche `test_histo` contient le test en développement sur la fonction `histo(configuration1, configuration2)`.
- Comme un test a échoué, on a créé une branche `fix-histo` sur laquelle on corrige le bug détecté.

Lors que bug est corrigé, on bascule sur `test_histo`, que l'on fusionne avec `fix-histo`. Puis on fusionne `test_histo` dans `refonte_tests_unitaires`.

3) Importer le module `unittest` et le fichier `partieBC.py` dans le code `partieBC_tests.py`.

4) Créer la classe `TestPartieBC` :

partieBC_tests.py

```
1 import unittest
2 import partieBC
3
4 class TestPartieBC:
5     """
6     Classe de tests permettant de valider le fonctionnement du Bulls and Cows
7     """
```

Exercice 2 (Test de la fonction `combinaison_secrete()`). **1)** Écrire une méthode de test appelée `combinaison_secrete_test(n)` qui appelle `n` fois la fonction `combinaison_secrete()` et vérifie que les valeurs générées sont des entiers entre `0` et `9`. *Indication — On pourra utiliser* ***`isinstance`****, fournie par le framework* ***`unittest`****.*

Exercice 3 (Test de la fonction `histo(configuration)`). Programmer un test unitaire pour la fonction `histo(configuration)`.

1) Quel doit être la valeur retournée par l'instruction `histo([3, 6, 3, 1])` ?

2) Vérifier à l'aide d'un test unitaire que la fonction que vous avez programmée renvoie le résultat attendu.

3) On se donne deux entiers $m, n \in [\![0, 9]\!]$. Quel doit être la valeur retournée par `histo[m, n, n, m]` ?

4) Proposer un test unitaire associé à cette configuration et l'exécuter 1 000 fois pour des valeurs aléatoires de m, n.

Exercice 4 (Test de la fonction `nombre_bien_places(configuration1, configuration2)`). **1)**

Quelle doit être la valeur de retour lorsque les deux configurations passées en paramètre sont égales ?

2) Générer des tests automatisés permettant de représenter toutes les situations intermédiaires (« de 0 valeurs bien placées » à « toutes valeurs bien placées »).

3) On souhaite maintenant effectuer des tests plus élaborés sur cette fonction. Soient $\mathtt{m}, \mathtt{n} \in [\![0, 9]\!]$. Proposer un test unitaire permettant de vérifier que la fonction renvoie le résultat attendu dans tous les cas suivants :

partieBC_tests.py

```
1 nombre_bien_places([m, 1, 1, 1], [n, 2, 2, 2])
2 nombre_bien_places([1, m, 1, 1], [2, n, 2, 2])
3 nombre_bien_places([1, 1, 1, m], [2, 2, 2, n])
4 nombre_bien_places([m, m, m, m], [n, n, n, n])
```

Exercice 5 (Test de la fonction `nombre_bulls_cows(configuration1, configuration2)`)**.** Proposer de la même manière des tests unitaires automatisés (au moins 100) pour cette fonction.

Exercice 6 (Nettoyage et finalisation)**.**

- Lorsque tous les tests ont été codés et validés, fusionner le code dans la branche `dev`, puis dans la branche `master`.
- Procéder au nettoyage du dépôt Git en supprimant les branches devenues inutiles.
- Exécuter la commande suivante :

Terminal

```
1 user@host:~/Documents/R203_QDev/TP2 $ git log --oneline > gitlog.txt
```

Cette commande écrit le log de Git dans le fichier `gitlog.txt`.

- Déposer sur EPREL votre travail (sans le dépôt Git, mais avec le fichier `gitlog.txt`), placé dans une archive ZIP nommée `r203_tp2_<nomDeFamille>.zip`.