Set up - Jeu Mystère
Groupe 4

Houda Hannouni
Fatima Nouhi
Julien Beard
Ludovic Merel

Introduction

Pour le projet de **Test Driven Development**, nous avons décidé de mettre en place un jeu où le principe est de trouver un mot grâce à des synonymes, des images ou une phrase.
Pour ce faire, plusieurs choses doivent être mises en place pour démarrer le projet que nous détaillons ci-dessous.

Description

“XXX” est un jeu de devinettes.
Cette application est une manière de passer son temps tout en faisant travailler sa logique et son vocabulaire.

Chaque niveau affiche des images, des phrases ou des synonymes reliés à un mot. Le but du joueur est de trouver ce mot à l’aide d’un ensemble de lettres données au bas de l’écran.

Avant chaque partie de jeu, l’utilisateur peut choisir le niveau de difficulté :

- Facile
- Intermédiaire
- Difficile

La récompense, sous forme de pièces, lorsque le joueur trouve le mot, dépend de cette difficulté (facile : 10 pièces, intermédiaire : 20 pièces et difficile : 30 pièces).
Ces pièces seront utilisées pour acheter des bonus sous trois formes :

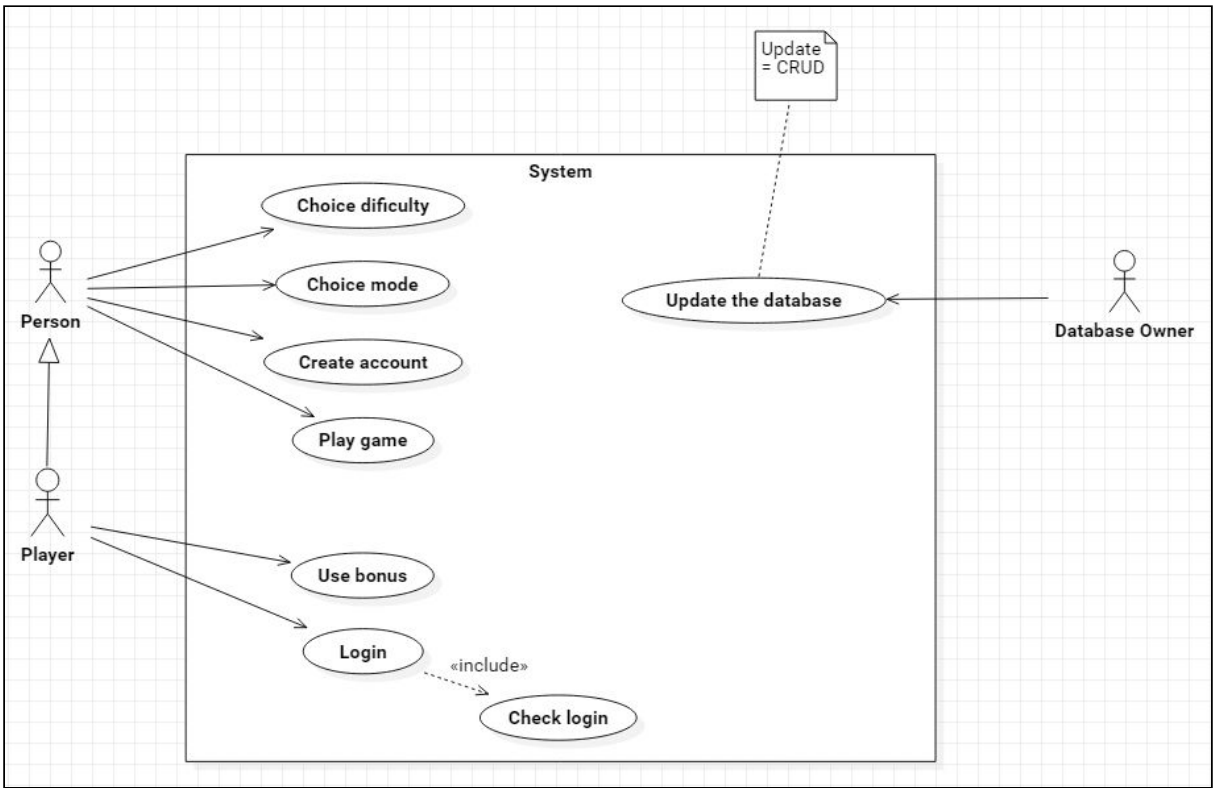
- Retirer des lettres inutiles
- Afficher une lettre du mot à deviner
- Passer le niveau

Si l'utilisateur ne se connecte pas avec un pseudo et un mot de passe à la première connexion, il ne bénéficie pas du système de pièces et de bonus.
Il ne pourra pas non plus jouer en ligne contre des personnes du monde entier.

Diagrammes

Cas d'utilisation

Ce diagramme permet de visualiser les rôles qui entrent en jeux dans l'application ainsi que leurs actions possibles.



Trois rôles nous semblent pertinents:

- **Person**: une personne lambda peut jouer au jeu sans se connecter mais certaines fonctionnalités ne seront pas disponibles.
- **Player**: un joueur a les mêmes droits qu'une simple personne mais comme elle a un compte, il lui est possible d'utiliser des suppléments.
- **Database Owner**: la base de données doit pouvoir être maintenue par une personne extérieure.

Une **personne** peut :

- Choisir la difficulté qui définira les mystères proposés ainsi que l'argent gagné et dépensé (cfr. partie diagramme de classe).
- Choisir le mode de jeux (cfr. description).
- Créer un compte pour profiter des bonus et des gains d'argent.
- Jouer au jeu.

Un **joueur** peut:

- Connecter son compte pour avoir accès aux suppléments.

- Utiliser des bonus comme: passer un mystère, enlever des lettres, afficher des lettres.
    - Lorsqu'une personne se connecte, un check est automatiquement fait pour vérifier que le compte est bien existant.

Un **database owner** peut:

- Ajouter des mystères.
- Modifier des mystères.
- Supprimer des mystères.

Diagramme de classes

Vous pouvez voir le diagramme en annexe.

Le diagramme de classe possède 14 classes.
« Game » est la classe centrale, elle permet principalement de récolter les données liées au jeu, de les traiter s'il le faut, et de les afficher correctement. Elle stocke donc le type de difficulté qui sera le même tout au long de la partie de jeu, les données du joueur, le type de jeu mystère en cours (soit PictureMistery, SentenceMistery ou SynonymMystery) et le mot associé à deviner.
A l'ouverture de l'application, le joueur est amené à d'abord choisir la difficulté (après s'être connecté). Ce choix restreindra alors les énoncés de chaque type de jeu. Ces derniers possèdent chacun des énoncés de difficulté facile, intermédiaire et difficile et sont triés selon cette logique dans la base de données.
La difficulté gère donc :

- Les énoncés de chaque type de jeu
- L'argent gagné
- L'argent dépensé dans les bonus

La classe « Game » utilise la classe « Bonus » pour avoir accès aux différents bonus possibles. Mais pour les utiliser, un joueur a besoin d'être connecté ; la classe « Bonus » utilise donc la classe « Game » qui stocke les données du joueur pour effectuer cette vérification.
Un design pattern est utilisé dans ce diagramme de classe.
La classe « Game », la classe abstraite « TypeMystery » et les classes qui l'implémentent forment le design pattern « State ».

Pattern state

Notre application comporte 3 modes de jeux différents. Chacun d'eux possède sa propre logique par rapport à l'affichage ou à la recherche de mystères dans la base de données.
Le principe de ce pattern est simple: déléguer le comportement aux **états** (dans notre cas les états sont les différents modes de jeux).
Lorsque la partie centrale de l'application « Game » désire lancer un mystère, elle va demander au type de jeu sélectionné de lancer la recherche dans la base de données et de lui envoyer tout ce qui est nécessaire pour le traitement: les images, les synonymes, les devinettes ainsi que les lettres à afficher sur l'interface utilisateur.

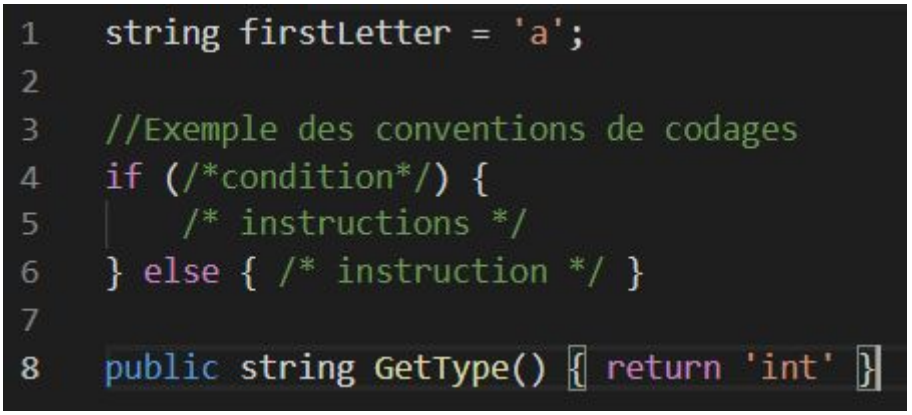
Conventions

Lorsque plusieurs personnes travaillent sur un même projet, il est conseillé d'adopter des conventions de codage. En effet, celles-ci permettent une lecture plus aisée du code et une meilleure compréhension des parties des autres membres.

Pour ce projet, nous allons vous présenter les conventions de codage que nous avons choisi de respecter :

1. Utiliser l'écriture UpperCase pour les méthodes (ex : GetType()).
   Et l'écriture lowerCase pour les noms de variables (ex : firstLetter)
2. Mettre l'accolade directement après toutes les instructions qui en demandent telles que les conditions, les boucles, ….
   Cette accolade doit être séparée par un espace de ce qui la précède.
3. Les conditions et boucles qui ne possèdent qu'une instruction, s'écrivent en 1 ligne.
4. Pour écrire une fonction, un espace doit être présent entre la parenthèse finale des paramètres de la fonction et l'accolade.
5. Dans les conditions, la condition entre parenthèses doit être isolée par des espaces.
6. En ce qui concerne les commentaires, ils doivent toujours se trouver au-dessus de ce qu'ils expliquent et être séparés du reste du code par une ligne vide au-dessus de ceux-ci.

Vous trouverez, ci-dessous, un exemple des conventions expliquées précédemment.



```
1   string firstLetter = 'a';
2
3   //Exemple des conventions de codages
4   if (/*condition*/) {
5       /* instructions */
6   } else { /* instruction */ }
7
8   public string GetType() { return 'int' }
```

Critères de qualités

Pour savoir comment améliorer la qualité d'un logiciel, nous devons d'abord définir certains critères de qualité. C'est pourquoi nous en avons choisis cinq.
Il faut ensuite décrire comment mesurer ces critères, comme Tom DeMarco a dit "Tu ne peux pas contrôler ce que tu ne peux mesurer".
Nous avons choisi ces critères de manière à balayer le plus d'aspect possible, que ça soit au niveau du codage, de l'interface ou de l'utilisation.

Densité des commentaires

Les commentaires jouent un rôle essentiel dans la qualité d'un code source. Il s'agit d'un des premiers aspects qu'un développeur passe en revu pour avoir une idée de la qualité général du logiciel.
Il faut un certains équilibre pour les commentaires. S'il y'en a trop, le code sera noyé par des informations inutiles et s'il y'en a trop peu, nous risquons de ne pas comprendre le rôle de certaines parties du code.

Notre but est donc de commenter de manière intelligente pour que nous puissions revenir sur le code sans devoir le relire entièrement.

Pour mesurer ce critère, nous allons utiliser un outil qui calcule la densité des commentaires suivant la "formule" :
DC = Source line of code / comment line of code

Nous avons décidé qu'un pourcentage acceptable serait de 30%.
Nous n'avons malheureusement pas trouvé de module permettant de faire cette vérification.

Couverture des tests

Les tests unitaires sont limité à un certain module. L'objectif est de vérifier le comportement d'un module isolé, et par conséquent de voir si le contrat de l'interface est rempli.
L' avantages des tests unitaires est que cela permets de détecter les erreurs plus vite ainsi que leurs localisation. En effet, lorsqu'un test échoue, son périmètre est délimité et donc nous pouvons facilement trouver l'origine de l'erreur et la corriger rapidement.
La couverture de code est un critère qui permet de voir le pourcentage de code couvert par les test et ainsi détecter les failles de notre système de test. Cependant une couverture de code parfaite n'élimine pas le risque de nouveaux bugs.

Des tests intelligents sont mieux qu'une couverture de 100%, c'est pourquoi nous voulons atteindre un seuil de couverture de code de 80% pour le code coeur et 30% pour l'interface.
Nous avons actuellement une couverture de 62% sur nos fichiers.

Duplication de code

Un code de qualité est un code qui n'est pas dupliqué. Nous tâchons de garder en tête cet aspect lors de la phase d'implémentation.
Pour éviter la duplication, il n'y a pas de solution miracle : nous devons refactorer.
Nous voulons éviter la duplication de code car lors d'un changement, elle nous oblige de reporter le changement dans toutes les parties de codes dupliquées.

Pour mesurer ce critère nous allons utiliser un logiciel qui détecte la duplication de code.
Nous estimons que nous ne devons avoir aucune duplication pour avoir un bon code.
Nous n'avons pas réussi à faire fonctionner le module (PMD) permettant cette mesure.

Ergonomie

Le design d'une application et l'ergonomie sont devenus parmi les principaux critères recherchés par l'utilisateur. Une interface non-attrayante est abandonnée au profit d'une autre application concurrente à l'aspect plus joli.
En effet, nous avons été habitués à un certain niveau d'esthétique ainsi qu'un à certain niveau d'ergonomie.
Ces critères étant impossibles à mesurer à l'aide d'un logiciel, nous constituons un panel d'utilisateurs qui donnera une note à l'aspect et à l'ergonomie de notre application.
Le score en dessous duquel nous ne voulons pas être est de 8/10.

Confort visuel	../10
Confort d'utilisation	../10
Enigmes respectent bien la difficultée choisie	../5
Compréhension du mode jeu intuitif	../10
A quel point avez-vous aimé l'application?	../10

Temps de réponse

Une application se doit d'être réactive de nos jours. Si l'utilisateur doit attendre trop longtemps lorsqu'il appuie sur un bouton nul doute qu'il se lassera et qu'il utilisera une autre application plus rapide.
C'est pourquoi nous avons décidé de prendre comme critère de qualité le temps de réponse.
Ce dernier est le temps entre une action sur l'interface user et la réponse du programme à celle-ci, mais du point de vue de l'utilisateur il s'agit d'un temps d'attente.
Lorsque nous calculerons le temps de réaction des actions de notre jeu, à l'aide d'un outil, nous ne voulons pas qu'il dépasse une seconde, mais surtout, il ne faut pas que l'utilisateur ressente une certaine latence.
Rien n'est fait sur Jenkins pour ce métrique.

Librairies utilisées

javatuples-1.2:

Cette librairie nous permet d'utiliser des tuples en java.
Voici la documentation: https://www.javatuples.org/using.html

org.json:

Grâce à ce paquet nous pouvons lire des JSON en java.
Voici la documentation: https://stleary.github.io/JSON-java/

org.apache.commons.io:

On peut lire et écrire facilement dans des fichiers
Voici la documentation:
https://commons.apache.org/proper/commons-io/javadocs/api-release/index.html

Installation du projet avec Intellij IDEA CE :

- Télécharger le logiciel Intellij IDEA CE
- Télécharger Java Development Kit 8 (selon votre OS) depuis le site Oracle
- Télécharger le projet "TestDrivenDevelopement" depuis le gitHub
- Ouvrir depuis Intellij le sous dossier "InterfaceMysteryGame" (attention il ne ne faut pas ouvrir le dossier principal)
- Aller dans File → Project Structure
    - Dans l'onglet Project, sélectionner pour
        - Projet SDK : new → JDK → sélectionner le dossier Java → "1.8" qui fait référence au jdk 8.
        - Project language level : "8- Lambdas, type annotations, ect."
        - Project compiler output : Un dossier par défaut est sélectionné mais si cette section est vide, ajoutez un dossier "out" que vous devez créer vous même dans "InterfaceMysteryGame" et le choisir.
        - Appuyer sur "Apply"
    - Sortir de la fenêtre en appuyant sur "OK"
- Dans l'arborescence sélectionner le fichier pom.xml, s'il est bleu ne rien faire sinon faire un clique-droit et appuyer sur "Add as Maven Project".
- Ouvrir dans l'arborescence le fichier "App.java" qui se trouve dans InterfaceMysteryGame → src→ main→ java→ inter , faire un clique-droit et appuyer sur "Run App.main()"
- Pour faire tourner le programme les fois suivantes, il suffit d'appuyer sur le bouton play/run en haut à droite

Mode d'emploi du jeu

1) Choisissez un niveau de jeu : Facile - Moyen - Difficile et sélectionnez "Valider"
2) Choisissez un mode de jeu : pour l'instant il n'y a que "Devinette" et appuyez sur "Jouer"
3) Si vous avez un pseudo et un mot de passe, veuillez les entrer dans les champs correspondants sinon appuyez sur "Commencer". Lorsque vous vous connectez un message sur la fenêtre de sortie précise si la connexion a réussi ou échoué.
   Pour l'instant, il n'y a que deux joueurs dans la base de données : Ludo (psw: 678910) et Houda (psw : cool)
4) Sur l'interface vous pouvez voir le nombre de lettres du mot à trouver ainsi que le nombre de pièces disponibles si vous vous êtes connecté avec un pseudo.
5) Pour répondre à la devinette veuillez cliquer sur les lettres au fur et à mesure pour former le mot.
   Si vous avez saisi une mauvaise lettre ou que votre réponse n'est pas correcte, vous pouvez cliquer sur le bouton "clear", ainsi les lettres reviendront à leurs places.
   Dès que la bonne réponse a été donnée, l'énigme suivante apparaît.

Amélioration de l'application

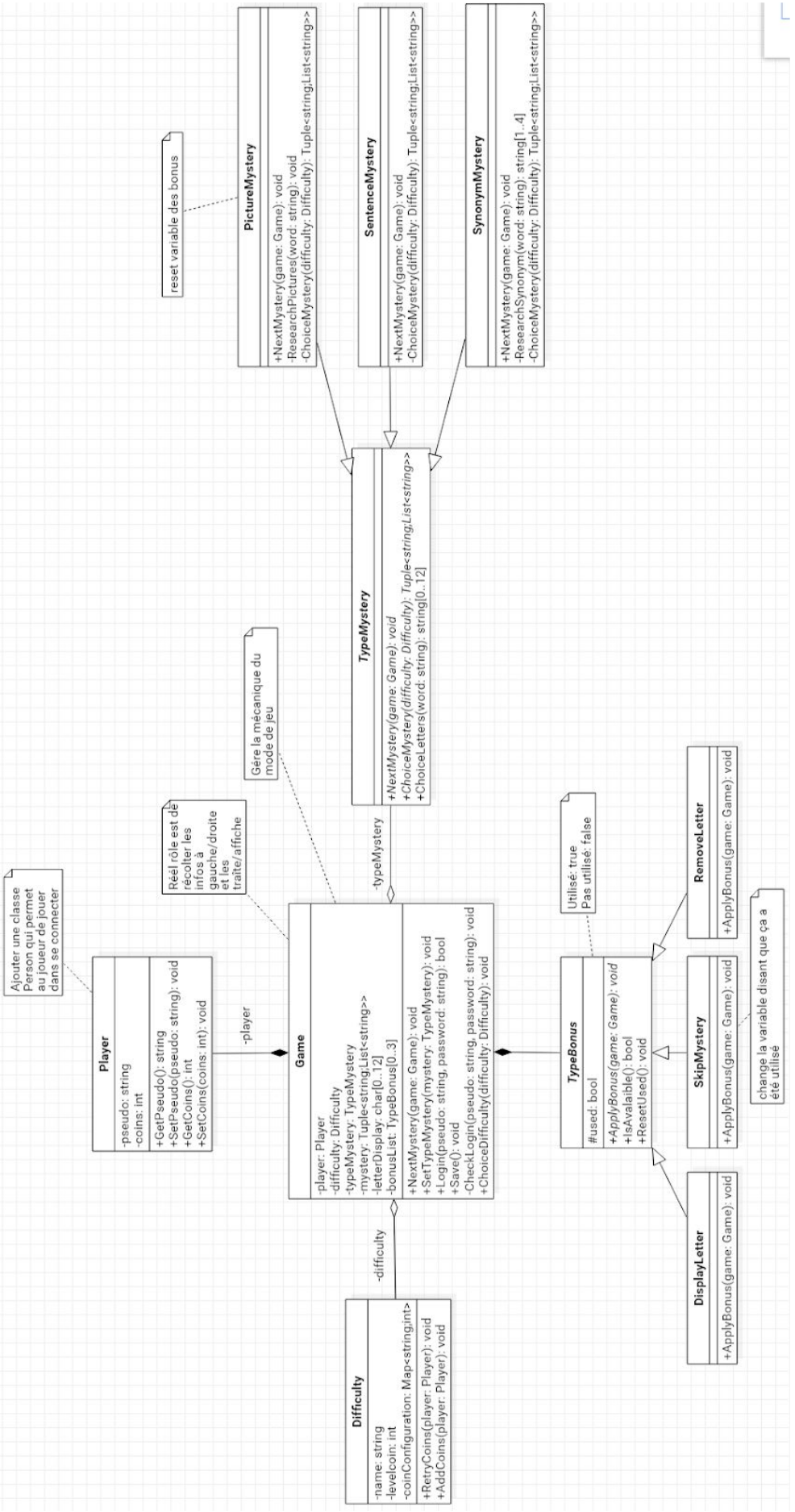
Voici les points qui doivent être faits pour terminer le projet :

1) Ajouter les 2 autres modes de jeux, avec les images et les synonymes (PictureMystery et SynonymMystery)
2) Créer les interfaces qui y correspondent
3) Ajouter les deux autres bonus : Donner la première lettre et passer le mot
4) Implémenter le fait d'ajouter un nouveau joueur dans la base de données
5) Créer une interface permettant de créer un nouveau joueur
6) Avoir une interface pour gérer les bases de données (des logins et des énigmes)

Conclusion

Ce projet nous aura donc permis d’imaginer la manière d'implémenter un jeu en Java, chose que nous n’avions jamais faite auparavant.
Par le biais de ce rapport, nous avons documenté notre travail afin d’avoir une trace de nos choix et de pouvoir le consulter en cas de doute face à certaines interrogations.
Nous avons essayé de fournir une description claire et concise de ce que nous voulons produire, pour cela, des diagrammes de classes et de cas d’utilisation sont nécessaires. Une mise au point de nos conventions de codage était aussi recommandée car elle permet une uniformisation du code. Enfin, nous avons détaillé nos critères de qualité ainsi que nos attentes afin de pouvoir y parvenir.
Nous avons également décrit les fonctionnalités attendues par notre application afin que chacun garde en idée ce à quoi nous devons arriver en fin de projet.

Annexe


Player
-pseudo: string
-coins: int
+GetPseudo(): string
+SetPseudo(pseudo: string): void
+GetCoins(): int
+SetCoins(coins: int): void
Ajouter une classe Person qui permet au joueur de jouer dans se connecter
-player
Game
-player: Player
-difficulty: Difficulty
-typeMystery: TypeMystery
-mystery: Tuple<string,List<string>>
-letterDisplay: char[0..12]
-bonusList: TypeBonus[0..3]
+NextMystery(game: Game): void
+SetTypeMystery(mystery: TypeMystery): void
+Login(pseudo: string, password: string): bool
+Save(): void
-CheckLogin(pseudo: string, password: string): void
+ChoiceDifficulty(difficulty: Difficulty): void
Réel rôle est de récolter les infos à gauche/droite et les traite/affiche
Gère la mécanique du mode de jeu
-typeMystery
TypeMystery
+NextMystery(game: Game): void
+ChoiceMystery(difficulty: Difficulty): Tuple<stringList<string>>
+ChoiceLetters(word: string): string[0..12]
PictureMystery
+NextMystery(game: Game): void
-ResearchPictures(word: string): void
-ChoiceMystery(difficulty: Difficulty): Tuple<stringList<string>>
reset variable des bonus
SentenceMystery
+NextMystery(game: Game): void
-ChoiceMystery(difficulty: Difficulty): Tuple<stringList<string>>
SynonymMystery
+NextMystery(game: Game): void
-ResearchSynonym(word: string): string[1..4]
-ChoiceMystery(difficulty: Difficulty): Tuple<stringList<string>>
-difficulty
Difficulty
-name: string
-levelcoin: int
-coinConfiguration: Map<string,int>
+RetryCoins(player: Player): void
+AddCoins(player: Player): void
TypeBonus
#used: bool
+ApplyBonus(game: Game): void
+IsAvalaible(): bool
+ResetUsed(): void
Utilisé: true
Pas utilisé: false
RemoveLetter
+ApplyBonus(game: Game): void
SkipMystery
+ApplyBonus(game: Game): void
DisplayLetter
+ApplyBonus(game: Game): void
change la variable disant que ça a été utilisé