



"Roll Through the Ages est bien plus qu'un simple jeu de dés. C'est un voyage passionnant à travers l'histoire, où les joueurs peuvent forger leur propre destinée et construire des civilisations qui dureront dans le temps."

Projet Informatique



ENSTA Bretagne
2 rue F. Verny
29806 Brest Cedex 9,
France

GANTIN Johan
DERMOUCHE Alexandre

Sommaire

Introduction	3
1. Présentation du problème.....	3
2. Définition des objectifs.....	4
2.1. Objectifs de la première partie	4
2.2. Pistes de recherches	4
3. Description des classes	4
3.1. Classes utilisées.....	4
3.2. Description des méthodes	6
3.2.1. La méthode <i>tirage()</i>	6
3.2.2. La méthode <i>nb_monument_accessible()</i>	7
3.2.3. La méthode <i>gestion()</i>	7
3.2.4. Les méthodes <i>construire()</i> et <i>test_construction()</i>	7
3.2.5. La méthode <i>unTour()</i>	7
3.2.6. La fonction <i>question()</i>	8
3.2.7. La fonction <i>creer_compte_rendu_joueur ()</i>	8
3.3. Description des tests.....	8
4. Analyse des résultats	8
4.1. Présentation.....	8
Conclusion	9
Table des figures	9

Introduction

Dans le contexte du projet informatique de second semestre à l'ENSTA Bretagne, nous avons choisi de réaliser un programme permettant de jouer au jeu de plateau « Roll Through The Ages ». Ce jeu bien connu des adeptes, permet de s'affronter entre amis. L'objectif étant de faire évoluer une civilisation est de l'emmener le plus loin possible. Ce projet a pour but de nous faire appliquer les connaissances que nous avons acquises en CM et en TD, orientées principalement sur la Programmation Orientée Objet (POO) en Python. Ce projet se décompose en 2 parties. La première partie se concentre sur la modélisation du jeu tandis que la seconde se focalise sur la mise en place d'une IHM et d'autres améliorations.

1. Présentation du problème

Le jeu à simuler comporte différents objets à manipuler, ce qui rend la POO pratique pour approcher le problème.

Au début de chaque tour, les joueurs lancent un certain nombre de dés pour collecter des ressources telles que la nourriture, la pierre, le bois et les outils. Ensuite, ils peuvent utiliser ces ressources pour construire des structures, nourrir leur population, développer leur commerce et leur culture, et améliorer leur civilisation. Les joueurs peuvent lancer un nombre de dé égale au nombre de leurs cités. Les dés peuvent fournir différents résultats tels que : de la nourriture, des marchandises, des ouvriers, des pièces et des crânes. Tous les dés peuvent par la suite être relancé hormis les crânes qui au bout d'un nombre total de 2 imposent des désastres au joueur. Ces désastres lui font perdre des marchandises et des points.

Afin de limiter les dégâts des désastres et de perdre un minimum de points, les joueurs peuvent acquérir avec leurs ressources des développements. Ces développements leur permettent aussi de gagner des points en fin de jeu, en leur offrant des facilités de jeu durant la partie.

Afin de gagner des points, les joueurs peuvent aussi construire avec leurs ouvriers des monuments ou des cités.

Le jeu se termine après un certain nombre de tours et les joueurs marquent des points en fonction de leur niveau de développement dans différentes catégories. Le joueur avec le plus de points à la fin du jeu est déclaré vainqueur.

2. Définition des objectifs

2.1. Objectifs de la première partie

Les objectifs définis dans la première partie sont :

- Mise en place des lancers de dé et du choix des dés à relancer par le joueur.
- Gestion de la nourriture (et des famines) et des constructions de ville.
- Gestion des constructions de merveilles, et l'adaptation des merveilles disponibles selon le nombre de joueurs.
- Contrôle des ressources via les amphores, la dépense des ressources et l'utilisation des pièces pour acheter des développements, la défausse des ressources en excédent en fin de tour.
- Action des fléaux et prise en compte des développements et merveilles liés aux fléaux.
- Prise en compte des règles spécifiques aux différents développements.
- Identifier que la partie est finie et compter les points.

2.2. Pistes de recherches

Plusieurs pistes de recherches ont été envisagées principalement au niveau de la structure des classes. Nous avons, au départ, fait l'erreur de créer une classe pour chaque objet. C'est-à-dire des classes : Player, Monuments, Cites, Développements, De, Désastres, Ressources, Partie, Cartes de civilisation, Gain. Notre démarche était alors plus adaptée pour la construction d'une base de données plutôt qu'une structure de classe.

Nous avons ensuite cherché des méthodes et des attributs en commun entre les différents objets afin de pouvoir définir une structure de classe adaptée. Nous avons établi une structure générale (méthodes et fonctions d'implémentation) afin de permettre un fonctionnement théorique du jeu.

3. Description des classes

3.1. Classes utilisées

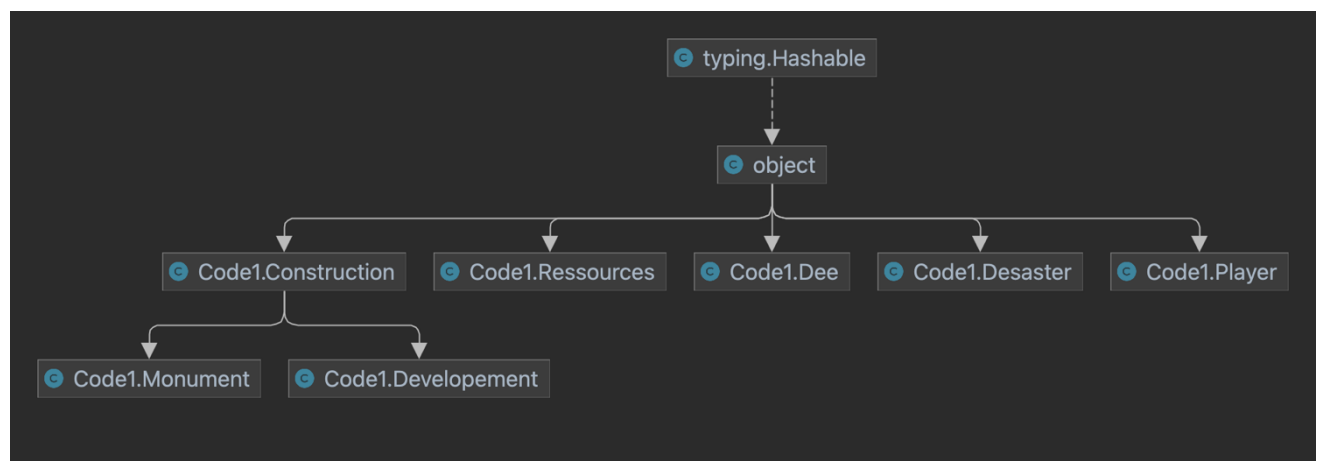


Figure-1 Diagramme de classe

Nous avons défini une structure générale des classes et de leurs héritages. Les classes principales Construction, Ressources, Dee, Désastre, Player correspondent aux objets utilisés dans le jeu. La classe

construction permet de généraliser le fonctionnement des cités, monuments et développements. Les monuments et développement possédant des attributs supplémentaires par rapport aux cités, nous avons choisi de faire des classes supplémentaires héritant de Construction. Le reste des objets ne possédant pas de point commun ils possèdent chacun une classe distincte. De plus, les classes héritent de Object afin de pouvoir appliquer les méthodes des objets classique sur l'ensemble des objets du jeu.

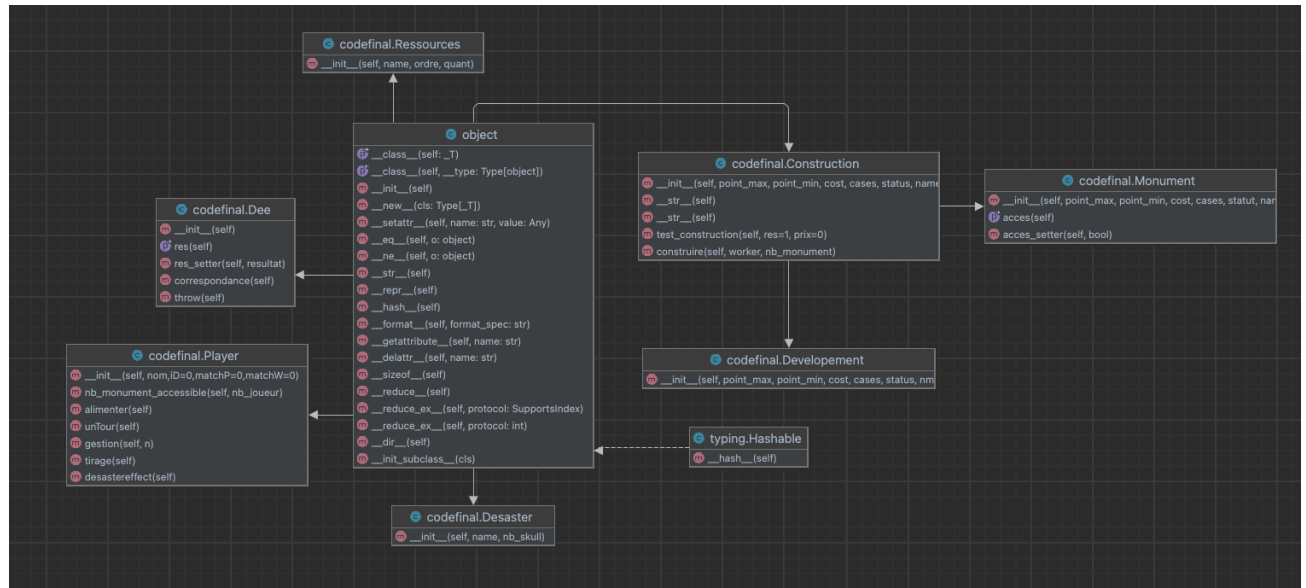


Figure-2 Structure des méthodes et attributs

La classe Player correspond à la classe permettant de modéliser les actions et attributs d'un joueur.

La classe Construction est la classe mère des classes Monuments et Développement. Les objets de Construction sont les cités du jeu.

Les classes Ressources, Désastre, Dee sont respectivement les classes définissant les ressources, désastres et dé du jeu.

Les objets de la classe Construction ont pour attributs :

- `self.point_max` : (int) nombre de points maximal accordés pour l'acquisition d'une construction
- `self.point_min` (int) nombre de points minimal accordés pour l'acquisition d'une construction
- `self.cost` (int) cout de l'acquisition d'une construction
- `self.cases` (int) nombre de cases à cocher pour obtenir une construction(égale au nombre d'ouvrier nécessaire
- `self.status` (boolean) statut de la construction acquis ou non acquis
- `self.nm` (str) nom de la construction, pour les identifier individuellement.

Pour les classes héritières, des attributs viennent s'ajouter :

Pour Monument :

- `self.acces = acces` (boolean) statut d'un monument accessible ou non en fonction du nombre de joueur

- `self.fisrt_buy = first_buy` : (boolean) devient vrai si le joueur est le premier à acquérir un monument, vient modifier le nombre de point attribuer au joueur pour l'acquisition d'un monument.

Pour Développement :

- `self.effet = effet` (str) correspond à l'effet du développement.
- `self.point_min= self.point_max`

Afin de permettre l'héritage nous avons veiller à ce que `point_max` et `point_min` soient égaux pour les cités et les développements.

Les attributs de la classe Ressources sont :

- `self.quant = quant` (int) correspond à la quantité de la ressource
- `self.nm = name` (str) nom de la ressource
- `self.decalage = 0` (int) permet la gestion des ressources pour suivre le plateau du jeu
- `self.ordre = ordre` (int) permet la gestion des ressources pour suivre le plateau du jeu

Les objets de la classe Player ont pour attributs l'ensemble des cités, monuments et développements et ressources disponibles dans le jeu. C'est ici que l'attribut *status* prend tout son sens, il permet lors du décompte final de savoir si une cité, monument ou développement a été acquis par le joueur. Se rajoute aussi les attributs :

- `self.nm = nom` (str) nom du joueur pour l'identifier
- `self.f_point = 0` (int) nombre de points total joueur
- `Self.point=[]` (list) liste qui donne le détail des points
- `self.skull = 0` (int) nombre de crâne obtenu lors d'un lancer
- `self.nb_city = 3` (int) nombre de cité du joueur
- `self.max_relance = 2` (int) nombre de relance de dé maximal possible
- `self.souhait_nb_relance = self.nb_city` (int) nombre de relance le joueur souhaite faire.

Les attributs de la classe Dee sont :

- `self.__res = 0` (int) résultat du lancer de dé
- `self.effect = "rien"` (str) effet du dé lors du tirage

3.2. Description des méthodes

Nous présentons dans la section suivante les principales méthodes de notre programme, qui permettent à notre sens d'avoir une compréhension globale du fonctionnement.

3.2.1. La méthode *tirage()*

La méthode *tirage()* est une méthode de la classe Player, son but est de simuler un lancer de dé par un joueur ainsi que l'ensembles de événements succédant se lancer (affichage du résultat du lancer, relancer, effet du lancer sur les ressources du joueur).

La méthode cherche d'abord à créer un objet de la classe Dee, elle effectue ensuite le lancer de ce dé en appelant une méthode de Dee, *throw()*, elle associe le résultat du lancer à son effet à travers d'une seconde méthode de Dee, *correspondance()*. Une fois l'effet déterminé, *tirage()* modifie les ressources du joueur selon l'effet du lancer. Les résultats sont affichés à l'utilisateur qui peut alors choisir de relancer les dés ne correspondant pas à un crâne. On veille bien évidemment à inverser les résultats du dé en cas de relance afin

de ne pas réduire les ressources du joueur injustement. La méthode prend de plus en compte la possibilité pour le joueur de posséder le développement « Leadership » qui lui permet alors de relancer un autre dé.

3.2.2. La méthode *nb_monument_accessible()*

La méthode *nb_monument_accessible()* est une méthode de la classe *Player*. Elle permet de définir les monuments qui peuvent être potentiellement accessible pour les joueurs. Le nombre de joueur de la partie influence en effet les monuments qui peuvent, ou ne peuvent pas être acheté. On utilise pour cela l'attribut *self.__acces* et l'accesseur *acces_setter()* de la classe *Monument*. Le booléen passe en True si le nombre de joueur requit pour le monument est suffisant.

3.2.3. La méthode *gestion()*

Une des spécificités de ce jeu est la récolte des ressources qui se fait de manière "pyramidale". Nous avons commencé par modéliser sous forme de suites le plateau:

$U_{n+1} = U_n + k \cdot n$ avec $k \in \{1,2,3,4,5\}$ selon la ressource

Ainsi il est tout naturel d'avoir créé une fonction *suite* qui calcule de manière récursive la valeur du plateau. Maintenant la méthode *gestion* doit retenir la position de dernier mouvement et ajuster si nous allons dans le sens positif (ajout de ressources) ou sens négatif (suppression de ressources). Cette mécanique permet au joueur d'avoir en temps réel la quantité de ressources qu'il possède et décider de sa stratégie de relance ou d'achat.

3.2.4. Les méthodes *construire()* et *test_construction()*

Afin de construire une cité ou un monument, on effectue un test d'acquisition à l'aide de la méthode *test_construction()* de la classe *Construction*. On vérifie que le joueur ne possède pas déjà la construction qu'il souhaite acheter et qu'il possède de même les fonds nécessaires pour l'acquisition.

La méthode *construire()* permet alors, une fois le test vérifié, de construire une construction en demandant à l'utilisateur le nombre d'ouvrier qu'il souhaite utiliser pour la construction. Si la construction est construite, on passe l'attribut *self.status* à True.

3.2.5. La méthode *unTour()*

La méthode *unTour()* représente en quelque sorte le « cœur » de l'algorithme. Elle permet d'automatiser l'ensemble des événements consécutifs se produisant lors d'un tour pour un joueur. La première étape est alors d'appeler la méthode *tirage()* afin de permettre au joueur de lancer ses dés. On alimente ensuite les cités du joueur à travers de la méthode *alimenter()*, en veillant à appliquer des pénalités en cas de manque de nourriture. On impose ensuite les dégâts liés aux désastre grâce à la méthode *desastereffect()*.

Dans le cas où le joueur possède des ouvriers, on lui propose d'acquérir des cités, monuments ou développements. Dans ce cas, on affiche la liste des monuments, cités, ou développements au joueur afin de lui permettre de faire un choix. On veille à garantir une robustesse de l'algorithme en s'assurant que le nom rentré par l'utilisateur est bien un nom de construction du jeu. On acquiert ensuite la construction grâce aux méthodes *test_construction()* et *construire()*. Pour l'acquisition d'un monument, on vérifie que le joueur est, ou non, le premier acquéreur afin de lui adresser le nombre de point adapté grâce à l'attribut *self.fisrt_buy*. Dans le cas de l'achat de développements, on fait appel aux attributs *self.gestionressource* et *self.decalage* afin d'assurer une gestion des ressources correcte pour acheter le développement et respecter le plateau du jeu. On fait ainsi appel à une méthode récursive, la

méthode *suite()*, simulant les suites numériques utilisé par le plateau de jeu pour la gestion des ressources.

3.2.6. La fonction *question()*

La fonction *question()* est une fonction global qui nous permet d'augmenter la robustesse de notre algorithme. Prenant en argument : (*type*, *question*, *listereponses_attendues*) on peut vérifier à chaque *input()* de l'utilisateur que la réponse fournie correspond à une réponse possible, définie dans l'argument *listereponses_attendues*. On s'assure ainsi que l'utilisateur ne pourra créer une erreur lors de l'exécution de l'algorithme.

3.2.7. La fonction *creer_compte_rendu_joueur()*

Cette fonction permet de répondre à la figure imposée d'écriture d'un fichier. Tout au long de la partie, le programme enregistre les différents détails comme le nombre de ressources, de pièces ou de crânes. A la fin, le joueur a la possibilité d'obtenir un fichier qui récapitule tous ces éléments sous forme d'un tableau.

3.3. Description des tests

Le fichier *unittest.py* contient des tests de vérifications de notre programme, nous avons fait le choix de tester :

- 1) Méthode ressource et sa fonction récursive qui modélise bien le plateau.
- 2) Lors d'un achat, l'effet d'un développement a bien lieu (dans l'exemple il nous évite un désastre)
- 3) Si la méthode *construire* liée à la classe *Construction* édifie bien une ville lors d'un achat du joueur.
- 4) Si le résultat du dé est bien protégé dans la classe *Dee*

4. Analyse des résultats

4.1. Présentation

Dans cette première partie, nous n'avons que la console python pour jouer au jeu. Nous avons fait le choix d'une disposition sous forme de tableau et des couleurs pour modéliser le plateau.

```

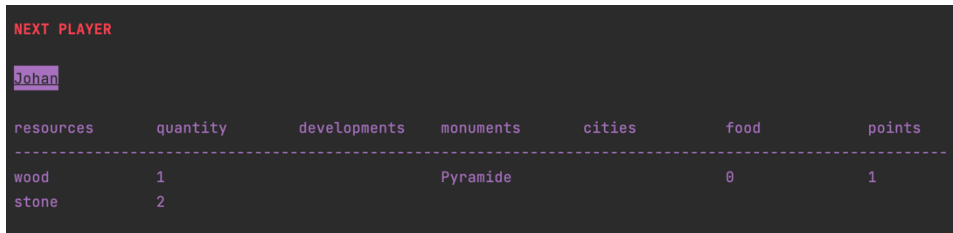
You are throwing dice

7 Coins
1 Good
1 Good
In total you got

effects          die    resources quantity skulls  coins  workers  number of throw remaining
-----
3 Food           0      food      3         0      7       0         2
1 Good           2      wood      1
2 Goods and 1 Skull 0      stone     2
3 Workers         0      pottery   0
2 Food or 2 Workers 0      cloth     0
7 Coins          1      arrow     0
-----
Do you want to roll at least one die again: yes/no |

```

Ici le joueur tire 3 dés et vient d'obtenir 7 pièces, 1 Marchandise et 1 Marchandise. Nous lui affichons l'état de son plateau pour qu'il puisse prendre sa décision. Nous affichons aussi au début de chaque début de tour l'état du plateau du joueur.



A screenshot of a game interface with a dark background. At the top left, the text 'NEXT PLAYER' is displayed in red. Below it, the name 'Johan' is shown in a purple box. A table with a dashed line separator lists player statistics. The table has seven columns: resources, quantity, developments, monuments, cities, food, and points. The data rows show 'wood' with a quantity of 1 and 'stone' with a quantity of 2. The 'monuments' column contains the word 'Pyramide', the 'food' column contains the number '0', and the 'points' column contains the number '1'.

resources	quantity	developments	monuments	cities	food	points
wood	1		Pyramide		0	1
stone	2					

Conclusion

Notre algorithme nous permet de modéliser de façon correcte le jeu Roll Through The Ages, en respectant son fonctionnement, ses règles ainsi que son système de gestion des ressources. Les tests unitaires nous permettent de garantir un fonctionnement nominal, il serait en effet dommage d'arrêter brutalement une partie endiablée en raison d'un bug du système. Les sécurités placées sur les inputs empêchent toutes les erreurs de saisie. Les améliorations à apporter reposent principalement sur la mise en place d'une base de données fonctionnelle en utilisant un hébergeur. La mise en place d'une IHM relevant de la seconde partie du projet, son implémentation sera effectuée par la suite.

Table des figures

Figure-1 : Diagramme de classe

Figure-2 : Structure des méthodes et attributs