Projet de laboratoire Java
Mise en œuvre du jeu « Game Over »

Table des matières

1 En bref

Ce projet vous permettra de mettre en œuvre la plupart des concepts vus pendant cette première année : programmation orientée objet, utilisation du langage Java, tests unitaires et documentation d'une application Java. **Lisez bien et complètement ce document.**

Cette lecture peut vous paraitre longue et demander des efforts, mais ce travail vous tiendra en haleine durant plusieurs semaines.

Ce document ne contient pas que des données relatives à l'exercice. Il vous renseigne également sur les modalités de remises et vous donne une progression dans la résolution de l'exercice.

Nous allons coder une application permettant de gèrer le jeu de mémoire à cartes « Game Over » dont les règles sont présentées ci-dessous.

2 Échéances

Le projet sera évalué en plusieurs phases. Vous veillerez à remettre chacun des travaux en temps et heure sous peine de ne pas être évalué.

Le tableau ci-dessous reprend les différentes semaines pendant lesquelles seront fixées, par votre professeur les dates butoirs de remise. On entend par **date butoir**, un moment défini par date et heure, pour lequel le travail doit être remis au professeur, et au-delà duquel le travail *ne* sera *plus* accepté. Rien ne vous empêche de remettre votre travail plus tôt que la date butoir, la veille, la semaine précédente ...

Partie	Plan de tests	Version 1	Version 2	Défense orale
Semaine du	24 février 2014	31 mars 2014	28 avril 2014	5 mai 2014

Votre professeur fixera et vous communiquera, dans ces semaines, les dates butoirs qui vous concernent spécifiquement.

3 Les phases du projet

Le projet se décompose en plusieurs phases. La première phase consiste à réfléchir aux tests à mettre en œuvre, tandis que les deux autres sont la réalisation des différentes versions de l'application.

Ce n'est qu'au terme de la dernière phase, la défense orale que votre projet reçoit une cote. En d'autres termes, si vous ne défendez pas oralement votre projet, il vous sera attribué la cote zéro pour l'ensemble du projet.

1 Phase 1, le plan de tests

Dans un **premier temps** nous vous demandons de réfléchir à un plan de tests pour une étape du jeu. L'objectif de ceci est de s'assurer que vous avez bien compris les règles. Il peut être nécessaire de mettre en œuvre plusieurs méthodes pour chacun de ces tests.

Si, de manière générale, un plan de tests est une description de tous les cas représentatifs de tests que vous jugez nécessaires pour prouver le bon fonctionnement de votre code, nous vous demandons ici de préciser les enchainements qui montrent que vous comprenez.

Cette description fera l'objet d'une première évaluation sur base du document que vous nous remettrez.

2 Phase 2, GameOver version 1

Dans un **second temps**, vous codez la première version de votre application que vous nous remettrez. Votre professeur fera des commentaires sur cette première version dont vous tiendrez compte pour coder la suite.

3 Phase 3, GameOver version 2

La deuxième version de votre application contiendra des fonctionnalités supplémentaires. Il est inutile de commencer la version 2 si la première version n'est ni fonctionnelle, ni évaluée.

4 Phase 4, défense orale

À la date, c'est-à-dire jour et heure convenue avec votre professeur, vous devez effectuer une défense orale, soutenue par une présentation sur machine, de votre projet. Le projet est présenté et défendu sur l'environnement linux1 qui est le seul environnement de développement considéré comme officiel de votre projet. Ceci signifie que les professeurs considèrent que tout doit être fait, fonctionnel, maitrisable complètement par vous sur cet environnement linux1 à l'école.
Tout ce qui est fait en dehors est de votre ressort personnel et ne sera pas évalué ni même considéré.
Au terme de cette défense orale, une cote finale pour l'ensemble du projet vous sera attribuée. Elle tiendra compte des éléments présentés dans la section 10 Modalités d'évaluation

5 Remises intermédiaires obligatoires

En plus des échéances fixées, une **remise hebdomadaire de votre travail est exigée**. Vous déposerez votre projet dans le casier de votre professeur dont l'acronyme en 3 lettres est «acr» via la commande `casier <acr>`.
Attention : cette remise ne sera pas évaluée, mais elle est **obligatoire** sous peine de ne pas voir votre projet corrigé.

6 Pondération de l'évaluation

À chaque partie évaluée correspond une pondération de la cote finale. Celle-ci est

Partie	Pondération
Plan de tests	1 / 8
Version 1 du projet	3 / 8
Version 2 du projet	1 / 2

4 Règles du jeu

Les règles originales du jeu se trouvent sur le site de l'éditeur du jeu[1], La Haute Roche dans un fichier pdf dont nous vous recommandons la lecture.[2] Voici une présentation que nous espérons éclairante et complémentaire de ces règles originales.
Nous mettrons en œuvre dans les versions 1 et 2 des versions limitées de ces règles, puis complétées. Ceci sera précisé plus loin dans les sections spécifiques.
Comme le précisent les règles originales,

> « Chaque joueur incarne un *Petit Barbare* à la recherche de sa Princesse perdue. Il doit courageusement parcourir un donjon peuplé de blorks (monstres), trouver la clé de la prison de la belle et la libérer.»

Le joueur est donc parfois appelé *barbare*. Le plateau de jeu est lui mentionné comme *le donjon*. Les cartes représentent donc aussi les *pièces* (= des chambres) du donjon.

1 http://www.lahauteroche.eu/gameover/gameover_home.html
2 http://www.lahauteroche.eu/gameover/gameover_GRAFIK/montage%20regles_depliant.pdf

Le jeu GameOver est un jeu de mémoire joué par 2 à 4 joueurs. Le but du jeu est de parcourir un plateau de « cartes inconnues car face cachée » de 5 * 5, le donjon, partant d'une position de départ fixe afin de trouver 2 cartes, la clé et la princesse de sa propre couleur.

Pour ceci, à chaque tour, le joueur part de sa position initiale sur un bord du donjon, choisit une des cartes «arme» et découvre successivement des cartes adjacentes en les retournant et en choisissant une nouvelle «arme» à chaque carte. Suivant la nature de la carte retournée et de l'«arme», plusieurs cas sont possibles.

Le joueur continue son exploration si la carte découverte est :

- un blork qui peut être vaincu avec l'arme choisie (icône correspondante),
- une princesse,
- la clé,
- la porte de transfert. Dans ce dernier cas, le joueur peut continuer son exploration à partir de n'importe quel autre emplacement du plateau de jeu non encore retourné.

Lorsque le joueur continue son exploration, il choisit de nouveau une arme : il peut garder l'arme déjà utilisée ou l'échanger contre une autre. Puis il retourne une carte adjacente par l'un des côtés (jamais en diagonale). S'il peut encore poursuivre son exploration, il laisse cette nouvelle carte visible et continue son chemin en choisissant son arme pour chaque nouvelle carte adjacente retournée.

Le tour d'un joueur prend fin (GameOver) lorsque la carte découverte est :

- un blork qui ne peut pas être vaincu avec l'arme choisie,
- un blork invincible. Dans ce cas, le joueur doit intervertir ce blork invincible et l'une des cartes encore face cachée du plateau, à l'exception des entrées du donjon.

Lorsque le tour d'un joueur prend fin (GameOver), le joueur retourne toutes les cartes visibles et replace l'arme près du plateau. Au tour suivant, il devra repartir de son entrée. C'est alors au joueur suivant d'explorer le donjon avec les informations déjà dévoilées qu'il a, bien sûr, mémorisées....

Si un joueur n'a plus la possibilité de continuer son exploration parce que toutes les cartes adjacentes à sa dernière carte retournée sont visibles, c'est également GameOver. C'est alors au tour d'un autre joueur.

Le jeu prend fin dès qu'un joueur parvient, au cours de son exploration, à avoir simultanément visibles la clé et la princesse de sa couleur. Il remporte alors la victoire.

5 Présentation des classes

Cette présentation est une présentation sommaire des classes qui vont intervenir dans le projet afin de pouvoir travailler sur le plan de tests. Une description détaillée est présentée plus loin.

Nous allons distinguer la partie **métier** (business) de la partie **vue** (view) de l'application. Cette manière de distinguer les classes en fonction de leur rôle est un patron de développement appelé Modèle / Vue / Controleur (en anglais, design pattern Model / View / Controller). Vous allez approfondir celui-ci en deuxième année. Nous ne mettrons en œuvre ici qu'une première approche[3].

La partie vue (view) concerne les classes qui s'occupent de la présentation et de l'interaction avec l'utilisateur.

3 https://en.wikibooks.org/wiki/Computer_Science_Design_Patterns/Model%E2%80%93view%E2%80%93controller

Classes	Rôles
`Game`	Gère le jeu
`Room`	Une des pièces du donjon
`Player`	Un joueur
`DungeonPosition`	Une position dans le donjon
`Dungeon`	Le donjon[4]
`GameView`	La classe qui contient le *main* et l'interface utilisateur

Il y a aussi plusieurs énumérations présentées plus loin.

Les extraits de diagramme des classes sont fournis pour vous donner une autre présentation des classes que la description écrite. Ces diagrammes contiennent aussi des éléments privés qui sont des choix fait par un des auteurs de code. Pour votre code, vous êtes invités à **faire des choix personnels et différents**.

6 Packages

Dans ce projet, vous travaillerez dans 2 *packages*, chacun regroupant les classes d'une des parties définies ci-dessous :

Classes	Package
classes métiers (model)	g12345.gameover.model
classes de présentation (vue)	g12345.gameover.view

7 Plan de tests

Avant de commencer à coder une application, c'est un bon usage de planifier et mettre en place des tests unitaires qui pourront être lancés régulièrement pendant le développement de l'application et montrer que les méthodes du programme font ce qui est attendu.
Dans ce projet-ci, dans un premier temps, avant de prévoir des tests unitaires, nous vous demandons de réfléchir et de décrire les phases du jeu qui sont susceptibles d'apparaitre et devraient être testées.
Pour chaque cas, un point de départ connu et à préciser est la configuration donnée du plateau de jeu, avec des positions définies des cartes et une position initiale définie du joueur.

Nous vous demandons de considérer les tests de la **classe Game** qui représente le moteur de jeu. Il faut tester que les enchainements de coups joués sont corrects. Par exemple, on peut gagner si, lorsque l'on trouve la princesse de sa couleur, on a au préalable trouvé la clé. Les tests que vous écrirez devraient recenser toutes les combinaisons possibles de ces enchainements.

4 Comme déjà dit, le donjon est le plateau de jeu constitué des 25 cartes.

Comme entrainement avant d'écrire le plan de tests de la classe Game, nous vous demandons d'écrire les tests de la classe *DungeonPosition.* Cette classe représente les positions dans le donjon et possède une méthode permettant un déplacement dans une direction (vers le haut, vers le bas ...). Pour écrire les tests Junit de cette classe, il faut s'assurer que la méthode *move(Direction)* retourne bien une nouvelle position lorsque le mouvement est possible et que cette position est la position attendue. Un mouvement possible est un mouvement qui reste dans le donjon. Dans le cas où le mouvement n'est pas possible, une erreur doit être lancée et ceci doit être testé.

Nous pourrions écrire quelque chose comme

Position initiale	Mouvement	Position attendue
(2,3)	HAUT	(1,3)
(0,3)	HAUT	erreur

Les coordonnées des positions du donjon sont définies dans le plateau comme présenté ci-dessous :

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)

8 GameOver, version 1

Dans cette première version de l'application, nous gèrerons l'inscription et le jeu de base. En particulier, nous ne nous occuperons pas des déplacements lorsqu'une carte GATE (porte) ou une carte blork invincible est rencontrée. Dans cette version 1, si une de ces 2 sortes de carte est rencontrée, la carte est retournée mais aucune action spéciale n'est exécutée, le blork invincible est simplement considéré pour le moment comme un blork ne pouvant être combattu par aucune arme.

Nous considérerons dans cette version 1

- que le joueur gagne si il retourne une carte clé et une princesse de sa couleur dans le même tour,
- que le joueur finit son tour (gameover) si il rencontre un blork invincible ou un blork non vaincu par l'arme choisie.

Nous ne gèrons pas le fait que le joueur termine son tour lorsque toutes les cartes autour de lui sont visibles.

Le premier ensemble de classes décrites ci-dessous fait partie du modèle.

Avant de se lancer dans l'écriture : PENSER À ÉCRIRE DU CODE LISIBLE

Il est attendu que l'application que vous écrivez soit facilement lisible et compréhensible, par vous et d'autres, en l'occurrence aussi les professeurs. Dès le début de l'écriture, faites attention à ces aspects.

Vous **devez** respecter les conventions d'écriture de codage en java, que vous trouvez sur poÉSI [5] dans la section du cours de java sous *Aide et références*.

Relisez attentivement ce document pas très long qui vous rappelle comment et quand passer à la ligne, comment et quand introduire des lignes blanches pour mettre en évidence, comment mettre votre code correctement en forme, que les lignes ne peuvent pas avoir plus de 80 caractères, qu'elles sont correctement indentées ...

N'oubliez pas d'écrire *aussi* la javadoc et des commentaires d'implémentation qui permettent de facilement comprendre le code, ce à quoi vous avez pensé durant l'écriture.

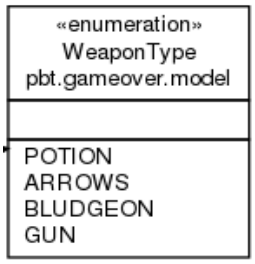
Faire ceci au fur et à mesure de l'écriture est bien plus simple et efficace que de le faire en fin de travail.

1 Classe GameOverException

Cette classe est une exception contrôlée par le compilateur. Cette exception sera lancée dès que l'on demande à la partie model quelque chose d'incohérent.

Méthodes

Cette classe a 1 constructeur : le constructeur à un paramètre de type *String*, qui décrit l'erreur qui s'est produite.



2 Énumération WeaponType

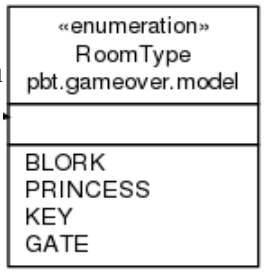
Cette énumération présente les 4 types possibles pour les armes. Elle a les valeurs suivantes :

- *POTION*, une potion,
- *ARROWS*, représentant un arc à flèches,
- *BLUDGEON*, une massue (un bâton noueux)
- *GUN*, une arme à feu.

3 Énumération BarbarianColor

Cette énumération présente les 4 types possibles de couleurs des barbares et des princesses. Elle a les valeurs suivantes :

- *RED*
- *GREEN*
- *BLUE*
- *YELLOW*



4 Énumération RoomType

Cette énumération présente les 4 figures possibles que peuvent prendre les pièces du donjon (dans le jeu original, les cartes), les *Room*.

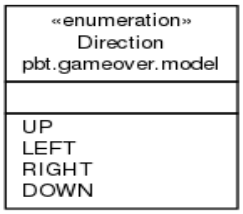
- *BLORK*, un personnage armé,
- *PRINCESS*, une princesse,
- *KEY*, une clé,
- *GATE*, une porte

5 http://elearning.esi.heb.be/main/document/showinframes.php?cidReq=LGJ1&file=%2FAide%2FCodeConventions.pdf en français ou http://elearning.esi.heb.be/main/document/showinframes.php?cidReq=LGJ1&file=%2FAide%2FCodeConventions.pdf en anglais.

5 Énumération Direction

Cette énumération définit les 4 types de déplacements possibles.

- *UP*
- *DOWN*
- *RIGHT*
- *LEFT*



6 Classe Dungeon

Comme nous avons besoin d'un de ses éléments, nous vous invitons à écrire ici une première version de la classe Dungeon. Nous y reviendrons et la terminerons plus loin.

Cette classe Dungeon possède un attribut de classe public

- *N=5*, *int*, une constante de classe publique qui indique la taille d'un côté du plateau de jeu.

7 Classe DungeonPosition

Cette classe représente une position dans le plateau de jeu, le donjon. Elle contient quelques difficultés. Nous allons donc la coder en plusieurs fois.

Dans un premier temps, vous codez une version simplifiée. Pour celle-ci, une position a comme attributs

- *column : int*, un entier positif représentant la colonne, comptée depuis le coin supérieur gauche (0) et dont la valeur maximale est le *N* de la classe Dungeon -1,
- *row* : *int*, un entier positif représentant la ligne, comptée depuis le coin supérieur gauche (0) et dont la valeur maximale est le *N* de la classe Dungeon -1.

Méthodes

Cette classe contient un premier constructeur :

- un constructeur public à 2 paramètres (ligne, colonne). Les valeurs des paramètres sont vérifiées par le constructeur, et une *GameOverException* est lancée si un des paramètres n'est pas dans les bornes autorisées.

Sont aussi présents, les accesseurs *int getRow()* et *int getColumn()*.

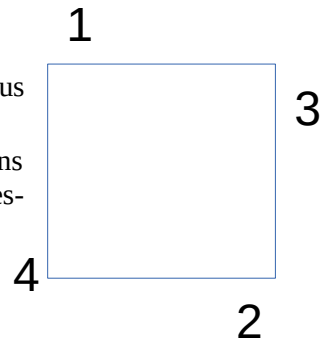
Une méthode *DungeonPosition move (Direction)*[6] qui, en fonction de la direction en paramètre renvoie la nouvelle position dans le sens considéré. Si la position de destination n'est pas autorisée car elle serait en dehors du donjon, une exception sera lancée.

Elle réécrit la méthode *toString()*.

Vous avez maintenant une première version fonctionnelle de la classe.

Vous pouvez compléter l'écriture de cette classe par des éléments plus complexes.

La classe définit aussi quatre constantes de classe publiques, les positions initiales des joueurs. Ce sont P_BARBARIAN_1 à P_BARBARIAN_4. Celles-ci sont créées aux positions hors de la grille comme précisé dans les règles. Nous suggérons que ces positions initiales soient choisies de telle sorte que si 2 joueurs seulement participent, ils partent de positions opposées.



Les positions initiales seraient par exemple respectivement, pour 1 le nord-

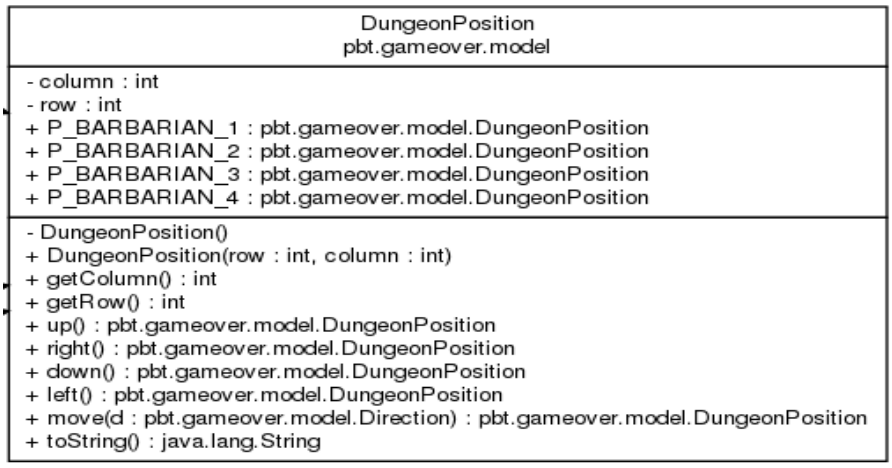
6 Lorsque la signature d'une méthode est donnée, on précise des éléments qui doivent être respecté, le type de retour, le nom de la méthode et les types des paramètres. Vous êtes libres de choisir les noms des variables utilisées en paramètres.

ouest, pour 2 le sud-est, pour 3 le nord-est et pour 4 le sud-ouest.

Il convient de rajouter un second constructeur, privé et sans paramètre *DungeonPosition()*. Celui-ci sera utilisé uniquement pour définir les constantes publiques P_BARBARIAN_1 à P_BARBARIAN_4.

Pour chaque position initiale, selon la description ci-dessus, il y a encore une ambigüité qui aura des implications sur le premier mouvement autorisé lors du début du tour de chaque joueur. Soyez clairs dans vos choix et indiquez par une présentation précise du plateau de jeu ou par des instructions précises à l'utilisateur ce qu'il convient pour que l'utilisateur sache quelle est cette position initiale que vous avez choisie et quel premier mouvement est autorisé.

Ces constantes publiques P_BARBARIAN_1 à P_BARBARIAN_4 sont initialisées dans un ***bloc statique***[7]. Un tel bloc statique est un ensemble d'instructions qui sont rassemblées dans un bloc, précédé du mot-clé *static*. Ce bloc sert à initialiser des variables de classe, statiques. Contrairement au constructeur, il est exécuté une seule fois, à l'initialisation de la classe. Ces blocs ne peuvent pas contenir de return, this ou super. Ils sont exécutés séquentiellement dans l'ordre dans lequel ils sont écrits.



8 Classe Player

Cette classe représente un joueur (player). Un joueur a comme attributs

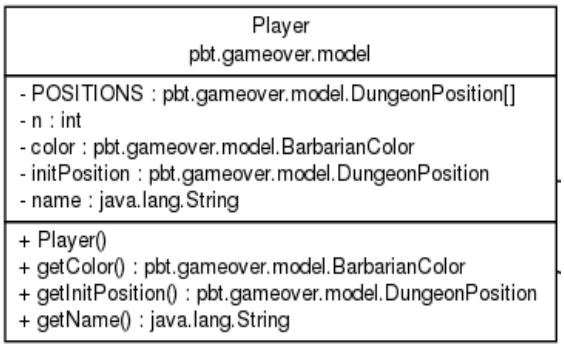
- *n : int*, un numéro unique (entier positif de 0 à 3),
- *name* : *String*, un nom,
- *color* : *BarbarianColor*, la couleur du personnage,
- *initPosition* : *DungeonPosition*, la position qui représente l'emplacement initial du joueur. Celle-ci est en dehors du donjon et est une des constantes définie dans la classe *DungeonPosition*. Cet attribut ne pourra qu'être lu.

Méthodes

Cette classe a un constructeur à 1 paramètre : le nom. La position initiale et la couleur sont définies automatiquement au départ, en utilisant le numéro *n*.

La méthode possède aussi les accesseurs *BarbarianColor getColor()*, *String getName()*, *DungeonPosition getInitPosition()*.

7 Block static : http://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.7



9 Classe Room

Cette classe représente un élément du donjon, une carte dans le jeu original.
Elle a comme attribut :

- *type* : *RoomType*, le type de figures que l'élément du donjon peut éventuellement porter,
- *weapon* : *WeaponType*, le type d'armes que l'élément du donjon peut éventuellement porter,
- *color* : *BarbarianColor*, qui représente la couleur de la carte,
- *hidden* : *boolean*, un paramètre qui indique si la carte est cachée (vrai) ou si elle a été retournée et est donc visible (faux). Ce paramètre est donc vrai au début de la partie.

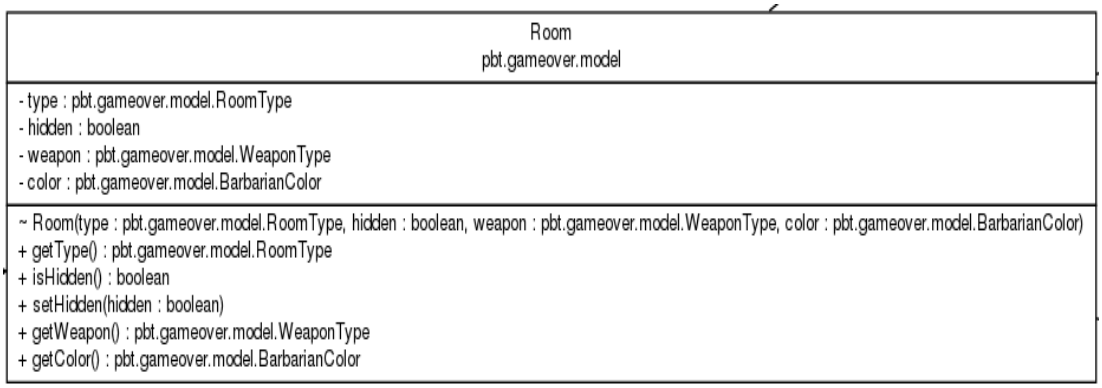
Lors de l'usage des objets instanciés de cette classe, certains attributs seront à *null* si ils ne sont pas pertinents. Par exemple, si le type de la carte est une clé, *color* et *weapon* sont à *null*. Si c'est un blork, son attribut *color* est à null. Si la valeur de RoomType est BLORK et *weapon* est à *null*, ceci décrira un blork invincible.

Méthodes

Cette classe contient

- un constructeur à 4 paramètres *Room(RoomType, WeaponType, BarbarianColor, boolean)* qui définit les 4 attributs,
- les accesseurs *RoomType getType()*, *WeaponType getWeapon()*, *boolean isHidden()* et *BarbarianColor getColor()*, et
- le mutateur *void setHidden(boolean)*.

Cette classe contient aussi une méthode *boolean equals(Object)* qui permet de comparer l'égalité de 2 objets sur base de leurs états. Comme la spécification le précise, il convient alors de réécrire aussi la méthode *int hashCode()*. Pour que le contrat soit satisfait, celle-ci doit être définie avec les mêmes champs que la méthode *equals()*.



10 Classe Dungeon (bis)

Nous revenons à la classe *Dungeon*[8]. Cette classe représente le donjon, c'est-à-dire le plateau de jeu. Elle a comme attributs :

- *N=5*, une constante publique qui indique la taille d'un côté du plateau de jeu. Nous l'avons déjà décrit précédemment car nous en avons eu besoin dans la classe *DungeonPosition*,
- *roomss : Room[][]*, un tableau à 2 dimensions qui contiendra les cartes mélangées.

Pour être sûr de ne créer qu'une seule instance de donjon, nous allons mettre en œuvre le *design pattern singleton*[9] qui consiste en une classe contenant une méthode qui crée une instance uniquement s'il n'en existe pas encore. Dans le cas contraire, elle renvoie une référence vers l'objet qui existe déjà.

Pour cela, le constructeur de la classe doit être privé, afin de s'assurer que la classe ne puisse être instanciée autrement que par la méthode de création contrôlée. On crée un attribut de classe privé *instance* et une méthode de classe publique *Dungeon getInstance()* seule autorisée à créer le donjon si l'instance n'existe pas. Cette méthode doit donc être appelée dès que l'on veut obtenir le donjon. Elle fournira son instance unique (et le créera si besoin).

Nous introduisons donc un attribut de classe

- *instance : Dungeon* initialement mis à null lors de l'initialisation.

Méthodes

La classe possède un constructeur *privé* sans paramètre *Dungeon()*. Le fait que le constructeur soit privé fait qu'il ne peut pas être utilisé par un autre objet. On introduit la méthode de classe *Dungeon getInstance()* qui est la seule par laquelle il est possible de créer l'objet, et uniquement si l'attribut *instance* était préalablement à null.

8 L'écriture d'un programme n'est pas une opération linéaire. On écrit un élément, une classe. Plus loin, il faut revenir au code écrit, l'adapter pour tenir compte de nouveaux éléments. Nous essayons de vous fournir à la fois une lecture linéaire mais aussi de vous apprendre ces pratiques qui permettent par exemple (le cas illustré ici), des références circulaires entre des éléments de 2 classes distinctes.

9 Vous trouverez quelques informations complémentaires sur ce patron de conception (design pattern) à http://www.oodesign.com/singleton-pattern.html

Ce constructeur crée explicitement les 25 pièces. Elles sont toutes initialement cachées. Celles-ci sont :

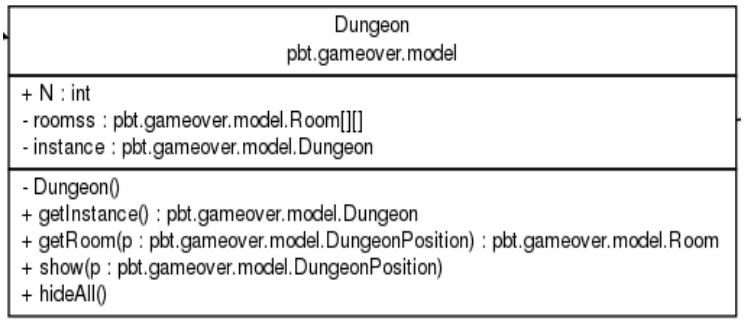
- 16 cartes de type BLORK séparées en 4 fois 4 avec chaque arme, sans couleur.
- 1 de type GATE, sans couleur.
- 2 de type KEY, sans couleur.
- 4 de type PRINCESS, chacune d'une couleur.
- 2 de type BLORK sans arme ni couleur qui seront les blorks invincibles.

Les 25 pièces sont créés avec les paramètres adéquats, mélangées et ajoutées une à une dans *roomss*. Il est alors possible que un blork qu'aucune arme ne peut vaincre (invincible) soit présent dans un coin.

Pour les tests, la classe possède aussi un constructeur dont la visibilité est *package*, qui permet de définir un plateau de jeu de configuration définie. Ce constructeur a la forme *Dungeon(Room[][] configuration)*. La visibilité *package* est utilisée pour mettre les tests dans le même paquet que la classe testée.

Il y a encore les méthodes

- *Room getRoom(DungeonPosition)* qui renvoie la *Room* à la position en paramètre,
- *void show(DungeonPosition)* qui modifie la valeur de l'attribut *hidden* d'une pièce pour qu'elle devienne visible. La valeur devient donc *faux*.
- *void hideAll()* qui modifie pour toutes les cartes du donjon les valeurs du paramètre *hidden* pour qu'elles soient à nouveau cachées.



11 Classe Game

C'est la classe qui contient le plus d'«intelligence». Elle sera la principale avec laquelle échangeront les classes de présentation de la partie *view*.

Cette classe contient les attributs :

- *dungeon : Dungeon*, le donjon
- *players : List*<`Player`>, la liste des joueurs
- *idCurrent : int*, le numéro du joueur courant,
- *lastPosition : DungeonPosition*, la dernière position du joueur courant. Si c'est son premier mouvement du tour, sa position est sa position de départ.
- *keyFound : boolean*, et
- *princessFound : boolean*, 2 valeurs qui sont mises à vrai lorsque, respectivement une clé et la princesse de la bonne couleur sont trouvées,
- *idWinner : int*, le numéro du joueur gagnant. Ce numéro est initialement mis à -1 par le constructeur, indiquant qu'il n'y a pas encore de vainqueur. Il est mis à l'id du gagnant lorsque celui-ci est défini.

Méthodes

Cette classe a les méthodes suivantes :

- un constructeur *Game(String... names)* où *names* sont les noms des joueurs. Le constructeur vérifie qu'il y a 2, 3 ou 4 joueurs et envoie une GameOverException si ce n'est pas le cas. Le constructeur va créer les joueurs et les ajouter à la liste *players*. Il crée un donjon unique par appel à la méthode *getInstance()* de Dungeon. Le joueur courant est le premier, la dernière position est la première du premier joueur, les variables *keyFound* et *princessFound* sont mises à faux.
- des getters :
  - *Dungeon getDungeon()*,
  - *Player getCurrentPlayer()*, qui renvoie le joueur courant
  - une méthode *boolean isOver()* qui permet de savoir si la partie est finie. C'est la valeur de idWinner qui indique cela[10].
  - une méthode *Player getWinner()* qui renvoie null si aucun joueur n'est encore gagnant et si la partie n'est pas finie, renvoie le joueur dont l'idWinner est donné.
- un setter de visibilité *package* pour les tests *void setDungeon(Dungeon)*
- une méthode *boolean play(Direction, WeaponType).* Cette méthode décrit un coup en fonction de la direction et de l'arme choisies. Elle est jouée par le joueur qui commence son tour ou par le joueur courant qui retourne une carte supplémentaire. Elle renvoie vrai si le joueur continue et faux si il a rencontré un blork d'une arme différente à la sienne ou invincible (sans arme).

  La méthode commence par tester que la partie n'est pas finie et lance une exception si elle l'est . La méthode teste si la carte à la position est déjà visible et si oui, lance une exception. Ensuite, une nouvelle position est déterminée en fonction de la direction en paramètre. La pièce du donjon à cette nouvelle position est mise à visible. En fonction des paramètres, cette méthode effectue le coup : selon le type de la pièce *room* à la nouvelle position, les cas sont évalués et on examine le cas échéant si le joueur gagne (clé et princesse de la bonne couleur trouvées) ou selon l'arme du blork qui apparait si la méthode renvoie faux.
- Comme dit plus haut, dans cette version 1, on ne traite pas le cas où l'on rencontre une porte (GATE). Ce cas sera laissé pour la version 2. Ici, on ne fait rien de spécial si cette carte est identifiée : elle devient visible. Et l'on perd contre un blork invincible, comme contre un blork d'une autre arme que l'arme choisie, et rien de plus n'est fait : il n'est pas déplacé.
- une méthode *void nextPlayer()*. Celle-ci incrémente le numéro du joueur (en revenant au premier si l'on dépasse le nombre de joueurs), remet les variables *keyFound* et *princessFound* à leurs valeurs initiales, redéfinit la position à la position initiale du nouveau joueur courant et remet toutes les cartes à l'état caché.

10 La partie est finie lorsqu'un gagnant est désigné. C'est le joueur qui le premier trouve une clé et la princesse de sa couleur. Ceci doit être distingué de la fin d'un tour dont il est souvent question. La fin d'un tour est mentionnée comme GameOver.

12 Classe GameView

Cette classe fait partie du package «view». Elle est destinée à l'interface utilisateur qui sera dans notre cas en mode **console**. Ceci signifie que pour faire une interface graphique, il ne faudrait modifier que les classes de ce package. C'est cette classe qui contient la méthode ***main***.

Cette classe gère les affichages et les entrées au clavier par l'utilisateur. Elle fait aussi appel aux classes qui interviennent dans la logique métier pour faire évoluer le jeu.

Cette classe instancie le jeu. Tant que le jeu n'est pas fini, c'est-à-dire qu'un joueur n'a pas trouvé la princesse de sa couleur et une clé, il faut demander au joueur de choisir et introduire un mouvement et une arme. Et bien sûr afficher le tableau de jeu et des questions. Il faut aussi passer d'un joueur à l'autre.

C'est sans doute une bonne idée d'ajouter une classe *Display* qui gère les présentations à l'utilisateur. Celle-ci typiquement possède des méthodes pour présenter une pièce, le donjon et du texte. Elle utilisera par exemple la classe java.io.Console et ses méthodes.

Écrire une interface texte n'est pas le plus amusant. Faites attention cependant à rendre l'interface conviviale et facilement utilisable, informative mais pas trop chargée.

13 Conclusion de la version 1

Vous devriez ici avoir une application fonctionnelle dans sa version 1. Vous préparez sa remise. Bien que vous ayez fait attention durant l'écriture elle-même, vérifiez que le code est correctement mis en forme : que les lignes ne font pas plus de 80 caractères, qu'elles sont correctement indentées, que les passages à la ligne respectent les conventions, que la javadoc et des commentaires d'implémentation qui permettent de facilement comprendre le code sont écrits ...

Rappel : Vous **devez** respecter les conventions d'écriture de codage en java, que vous trouvez sur poÉSI [11] dans la section du cours de java sous *Aide et références.*

Conservez précieusement des copies de vos codes et prenez une pause avant d'attaquer la version 2.

11 http://elearning.esi.heb.be/main/document/showinframes.php?cidReq=LGJ1&file=%2FAide%2FCodeConventions.pdf en français ou http://elearning.esi.heb.be/main/document/showinframes.php?cidReq=LGJ1&file=%2FAide%2FCodeConventions.pdf en anglais.

9 GameOver, version 2

Dans cette seconde version, nous serons moins directif que dans la version 1. Plus de liberté et de créativité vous seront donc laissés ... et demandés.

Nous commencerons par compléter le code afin qu'il réponde à toutes les règles :

- gestion de la carte GATE. Le joueur peut continuer son exploration à partir de n'importe quelle autre pièce du donjon (non encore découverte) ;
- gestion du blork invincible. Le joueur peut intervertir la carte blork invincible avec une des cartes encore face cachée, à l'exception des entrées du donjon ;
- gestion du cas où le petit barbare se retrouve coincé, c'est-à-dire entouré de pièces déjà découvertes. Dans ce cas, le tour du joueur prend fin, c'est GameOver.

Vous ajouterez aussi les 2 fonctionnalités suivantes :

- définir un statut débutant pour un joueur. Si un joueur est défini comme débutant, il profite d'un privilège par tour à choisir parmi :
  - un joker. S'il se trompe d'arme, il peut réessayer une fois;
  - le joueur gagne (quand même) contre un blork invincible et a le droit de le déplacer.
- passer en paramètre un fichier contenant un profil de joueurs.

Pour répondre aux règles de base, une refactorisation du code est nécessaire. En résumé, il faudra :

- ajouter une classe *BarbarianState* précisant l'état du barbare courant.
- ajouter un attribut *BarbarianState stateCurrent* dans la classe *Game*
- modifier ou ajouter des méthodes de jeu qui dépendent de cet état dans la classe *Game*
  - modifier la méthode *play(Direction, Weapontype)* qui retournera maintenant un *BarbarianState.*
  - ajouter deux méthodes qui décrivent les actions lorsque les cartes GATE ou BLORK invincible seront rencontrées, respectivement
    - une méthode *BarbarianState playGate(DungeonPosition, WeaponType)*
    - une méthode *BarbarianState playBlorkInvincible(DungeonPosition)*
- ajouter dans la classe *Dungeon* une méthode *void swap(DungeonPosition, DungeonPosition)* permettant d'échanger 2 positions,
- ajouter dans la classe Dungeon une méthode *boolean isSurrounded(DungeonPosition)* indiquant si une pièce est entourée de pièces toutes visibles.
- ajouter dans la classe *DungeonPosition* une méthode *boolean isCorner()* précisant si une position se situe dans un coin ou pas.
- modifier la classe *GameView* puisque les échanges avec l'utilisateur seront différents et les possibilités plus nombreuses.
- enfin, il faudra revoir les tests de la classe *Game* car la méthode play ne retourne plus un booléen mais un *BarbarianState*.

Avec plus de détails, voici la description du travail et de ces classes.

1 Classe BarbarianState

Cette classe est une énumération qui décrit les différents états possibles du joueur dans son tour. Ce sont :

- GAMEOVER, état qui marque la fin du tour d'un joueur. C'est alors au joueur suivant d'introduire son mouvement,
- READY_TO_GO, au début du tour, avant d'entrer dans le donjon,
- CONTINUE, à tous les moments où le joueur peut poursuivre son exploration car son tour n'est pas fini,
- MOVE_BLORK, lorsqu'un joueur a identifié un blork invincible et peut le déplacer,
- BEAM_ME_UP, lorsqu'un joueur tombe sur la porte GATE et peut poursuivre de toute autre position[12],
- WIN, lorsque le joueur a gagné.

2 Classe DungeonPosition

La carte d'un blork invincible, lorsque celui-ci est identifié, ne peut pas aller dans un coin. Il faut donc rajouter une méthode *boolean isCorner()* qui renvoit vrai si une position se situe dans un coin et faux autrement.

3 Classe Dungeon

Il faut ajouter dans cette classe une méthode *void swap(DungeonPosition, DungeonPosition)* . Cette méthode permet d'échanger dans le donjon 2 positions. Ceci permet de déplacer le blork invincible.

Une seconde méthode, *boolean isSurrounded(DungeonPosition)* doit être ajoutée. La méthode retourne vrai si la position en paramètre est entourée de cartes visibles. La logique de cette méthode demande quelque réflexion : il faut tenir compte du fait que la position est près d'un bord ou non par exemple.

4 Classe Game

Cette classe nécessite sans doute le plus d'adaptation.

- Un attribut *BarbarianState stateCurrent* est ajouté. Il décrit l'état du joueur courant. Les valeurs de cet état seront définies et testées dans les méthodes de jeu de la classe *Game* .

Méthodes

- Le constructeur *Game(String... names)* définit maintenant en plus, au départ que *stateCurrent* est dans l'état CONTINUE.
- La méthode *BarbarianState play(Direction, WeaponType)* est modifiée. Elle teste qu'elle ne peut-être effectuée que si le joueur est dans l'état CONTINUE. Si ce n'est pas le cas, elle lance une exception. Elle teste aussi qu'aucun gagnant n'a déjà été trouvé.

  Les différences par rapport à la version 1 apparaissent dans les différents cas selon les types de cartes. Si les cartes KEY ou PRINCESS sont trouvées, c'est comme pour la version 1. Dans les autres cas, la valeur de *stateCurrent* est déterminée à BEAM_ME_UP si la porte est rencontrée, à MOVE_BLORK si un blork invincible est rencontré ou à GAMEOVER si un blork d'une autre arme est rencontré. Elle retourne la valeur de stateCurrent. Il peut être intéressant de factoriser une partie de cette méthode dans une méthode privée *BarbarianState play(DungeonPosition, WeaponType)* qui sera aussi utilisée lorsque l'on effectue le déplacement dans *playGate()* (voir ci-dessous).

12 « Beam me up » lorsque le petit barbare bénéfice du passage par la porte dérobée fait référence à une phrase classique pour les fans de la série StarTrek. Voir https://en.wikipedia.org/wiki/Beam_me_up,_Scotty

- La méthode *BarbarianState playGate(DungeonPosition, WeaponType)* est ajoutée. Elle permet de jouer le déplacement du joueur lorsque la room GATE est rencontrée. La méthode teste successivement si l'état est bien celui attendu (BEAM_ME_UP) et si un gagnant n'a pas été trouvé et si ce n'est pas le cas, lance une exception. La position *lastPosition* est cette fois celle passée en paramètre plutôt qu'une position calculée d'après le mouvement. Puis, comme dans la méthode *play()* la méthode teste tous les cas de type de carte et selon chacun d'eux, définit et retourne la valeur de *stateCurrent*.
- Une méthode *BarbarianState playBlorkInvincible(DungeonPosition)* décrit les actions si un blork invincible a été identifié. Elle teste successivement si le joueur est bien dans l'état MOVE_BLORK, si un gagnant n'a pas été trouvé et si la position n'est pas un coin et dans chaque cas lance une exception si c'est le cas. Dans le cas contraire, les positions sont échangées et les cartes à chacune de ces positions sont montrées. Puis l'état stateCurrent est mis à GAMEOVER avant d'être retourné.

5 Classe GameView

Les interactions avec l'utilisateur doivent cette fois tenir compte des nouvelles entrées possibles et des nouveaux états possibles. Il convient donc d'adapter les questions posées à l'utilisateur et l'affichage. Cela demande travail et réflexion que nous vous laissons.

6 Pour les 2 nouvelles fonctionnalités

Nous vous conseillons de commencer par implémenter la notion de joueur débutant. Pour ce faire, il faut

- ajouter un attribut *boolean beginner,* son getter et son setter dans la classe *Player.*
- ajouter un état JOKER dans la classe *BarbarianState*
- ajouter un attribut *boolean jokerUsed* dans la classe *Game.* Celui-ci mémorisera si le joker a été utilisé dans un tour du jeu et sera mis à *false* dans le constructeur *Game()* et sera remis à false dans la méthode *nextPlayer().*
- ajouter une méthode *BarbarianState playJoker(WeaponType)* dans la classe *Game*. Cette méthode s'inspire fort de *playGate()* en testant les cas selon les types de cartes , mais cette fois définit la valeur de jokerUsed à vrai avant ces tests. Elle retourne la valeur de stateCurrent.
- adapter les méthodes de la classe *Game*, *play(), playGate()* et *playBlorkInvincible()* en conséquence, en utilisant les états.
- il faut aussi bien sûr adapter la classe *GameView* pour tenir compte de ces modifications.

Dans un deuxième temps, ajouter la fonctionnalité de lecture du fichier dont le nom est passé en paramètre. Le fichier contiendra au maximum 4 noms de joueurs et définira si ils ont les privilèges de débutant, et aura l'allure suivante :

```
Colas débutant
Aglaé
Abraham débutant
Zeus
```

Vous devrez, dans la classe *GameView*, prendre en charge le « passage de paramètres » au programme via la variable *args*, argument de la méthode *main()*. S'il y a un paramètre qui convient, chercher le fichier correspondant et le traiter. Pour la gestion du fichier, il suffira simplement de l'ouvrir et de le parcourir ligne par ligne. À vous de choisir si vous accepter les noms composés de plusieurs mots …

C'est probablement une bonne idée d'ajouter une classe spécifique qui prend en charge la gestion de ce fichier. Nous vous en laissons la création.

7 Conclusion de la version 2

Vous voici arrivés au bout du projet. Il vous reste, comme pour la version 1, à le relire attentivement pour que la mise en page et la documentation soient présentes, et à **respecter les dates de remise.** Ce serait dommage de rendre votre travail en retard.

Il convient aussi de préparer la défense orale (voir le point 4 Défense du projet de la section Modalités d'évaluation, page 20) .

10 Modalités d'évaluation

Cette section précise certains éléments d'évaluation du projet.

Lisez cette section attentivement pour ne pas être surpris par la manière dont le projet est évalué ou par les contraintes de remise

1 Attribution de la cote de base

Nous allons d'abord vous attribuer une cote en fonction de ce qui a été réalisé et testé (le projet est accompagné d'une liste des fonctionnalités attendues avec leur pondération). **Une fonctionnalité dont on demande les tests, ne sera considérée comme réalisée que si vous fournissez un programme de test qui montre que cela fonctionne effectivement**. C'est donc à vous de prouver que quelque chose a été fait et pas à votre professeur à le découvrir à partir de votre code. En pratique, vous devez montrer que votre code passe une série de tests Junit que vous écrirez.

2 Critères modifiant la cote

Un programme qui fait ce qu'on lui demande est loin d'être suffisant. Nous sommes également attentifs au style, à la documentation, ... Votre **cote sera donc réduite** si les différents critères décrits ci-dessous ne sont pas rencontrés.

La javadoc (jusqu'à 8 points de pénalité)

Une javadoc complète est essentielle. Est-ce que tout est décrit ? Est-ce que les cas particuliers sont documentés ? Est-ce que les balises sont utilisées correctement (return, param, throws, ...) ?

La lisibilité du code (jusqu'à 8 points de pénalité)

Un code qui fonctionne mais n'est pas facilement lisible n'est pas un bon code. On juge ici

- l'indentation,
- les noms judicieux de variables et de méthodes,
- la non redondance de code,
- l'utilisation de constantes littérales,
- l'utilisation judicieuse de commentaires, ...

Nous vous invitons encore une fois à relire et mettre en œuvre les conventions de codage que vous pouvez trouvez sur Poesi dans la section Aide et références du cours (document fourni en anglais et français).

Utilisation des éléments adaptés du langage (jusqu'à 5 points de pénalité)

Nous jugeons ici si vous utilisez à bon escient les différents éléments du langage.

Par exemple : vous serez sanctionné si, pour initialiser un petit tableau avec des constantes, vous le faites en assignant case par case plutôt que d'utiliser la possibilité offerte par le langage de spécifier toutes les valeurs via une liste entre accolades lors de la déclaration du tableau.

Écart par rapport à l'analyse (jusqu'à 8 points de pénalité)

L'énoncé reprend déjà une analyse plus ou moins développée du problème (classes et méthodes à fournir). Il est impératif de suivre cet énoncé. Tout écart sera sanctionné. Vous pouvez discuter avec votre professeur d'une proposition et trouver un accord sur une modification.

Il y a de nombreux endroits dans lesquels vous devez faire preuve de recherches personnelles et compléter l'analyse. N'oubliez pas de documenter convenablement vos mises en œuvre.

Interface utilisateur (jusqu'à 2 points de pénalité)

Nous accordons peu d'importance à l'interface utilisateur du programme. Il faut néanmoins qu'on puisse comprendre **sans effort et recherche** ce qu'il écrit et qu'on puisse interagir avec lui. Votre interface doit être fonctionnelle et utilisable. N'oubliez pas d'inclure et afficher les informations pour que le professeur – utilisateur soit informé par votre programme de ce qu'il doit entrer comme éléments.

Orthographe et grammaire (jusqu'à 5 points de pénalité)

Vous devez soigner le style de votre javadoc. Nous sanctionnerons les **fautes d'orthographe**, les problèmes de **grammaire**, les phrases incompréhensibles. Relisez attentivement votre programme et ses commentaires, avec un dictionnaire à portée de main. Demandez à une autre personne (ami, camarade de classe) de jouer et vous faire des remarques sur les textes.

3 Modalités de remise

Date de remise butoir

Votre professeur vous aura indiqué une date (jour et heure) ultime de remise du projet, pour chaque phase. Celui-ci doit être remis en main propre au plus tard à cette date. Ceci signifie que la remise est faite au laboratoire, en présence du professeur et prendra un certain temps. Votre professeur fera avec vous un certain nombre de vérifications et vous demandera des précisions ou démonstrations éventuelles.

Il est inutile de remettre une version 2 si vous n'avez pas remis de version 1.

Tout projet remis après la date ultime ne sera pas corrigé. Aucune excuse ne sera tolérée. Et si vous êtes malade ? Arrangez-vous pour le faire remettre par quelqu'un d'autre avant la date ultime. **Comme cela a déjà été signalé, tout projet remis en retard ne sera pas évalué**.

Contenu et support du rapport

Voyez avec votre professeur sur quel support informatique vous devez rendre le rapport. La plupart des professeurs demandent une remise avec la commande `casier <acr>` sur linux1. Il doit contenir :

- tous les sources,
- les versions compilées,
- la javadoc

Tout doit être directement utilisable. Nous ne procéderons à aucune compilation ou génération de javadoc pour vous. Aucun document ne doit être imprimé sauf si demandé explicitement.

Remise des versions intermédiaires

Vous avez l'obligation de nous remettre une version intermédiaire de votre travail **une fois par semaine** (dans le casier électronique sur linux1 via la commande `casier <acr>` sur linux1). Cela nous permet de suivre votre évolution.

- Cette remise est obligatoire sous peine de non correction de votre projet.
- Ces versions intermédiaires ne seront pas évaluées. Elles n'interviendront donc pas dans l'établissement de votre cote.
- Votre professeur ne doit pas vous rappeler de faire ce dépôt, même si il le fera plus que probablement quelques fois. Si vous n'avez pas beaucoup avancé ? Remettez le peu que vous avez déjà fait.

4 Défense du projet

La défense du projet clôture la dernière phase du projet et n'a qu'un seul but : **vérifier que vous êtes bien l'auteur personnel du travail remis**.

Pour ce faire, nous vous poserons des questions sur ce que vous avez fait, pourquoi vous l'avez fait de telle ou telle manière. Nous vous demanderons de petites modifications ou adaptations du code.

La participation au laboratoire intervient également dans la manière de procéder. Il est évident que nous vous avons vu travailler et développer et que ceci influencera les questions que nous vous poserons.

Comme décrit précédemment, un **projet non défendu oralement est considéré comme non remis et sera évalué à 0**. Toutes les cotes intermédiaires sont perdues.

5 Le problème de la tricherie

Bien sûr nous vous encourageons à vous entraider. Mais cela doit rester un échange : discussion sur l'énoncé, comparaison de logiques, entraide pour corriger du code, ... **Aucune copie de code ne sera tolérée.** Si nous trouvons des projets qui sont trop proches (changer le nom des variables ou modifier les commentaires ne suffit pas !) nous mettrons 0 à tous ces projets. Aussi bien celui qui a laissé copier que celui qui a copié. Soyez donc vigilants : sur la machine linux1, **adaptez les permissions de votre dossier** contenant le projet pour empêcher les copies par les autres étudiants et ne laissez rien trainer.