**Master**

Sciences et Technologies du Logiciels

---

# Devoir CPS : Dungeon Master

---

**Réalisé par :**
**Thamazgha SMAIL**
**Joseph RASCAR**

# Contents

# List of Figures

# 1 Introduction

Le but de ce projet est de mettre en oeuvre les connaissances acquises en ce qui concerne la spécification et programmation par contrats dans le module CPS en spécifiant d'un jeu similaire à "Dungeon Master"

# 2 Manuel d'utilisation

## 2.1 La touche "UP" du clavier

quand l'utilisateur appuie "UP" le joueur avance dans sa direction courante, Nord initiallement.

## 2.2 La touche "DOWN" du clavier

Quand l'utilisateur appuie sur "DOWN" le joueur recule.

## 2.3 La touche "RIGHT" du clavier

quand l'utilisateur appuie sur "RIGHT" le joueur fait un pas chassé à droite et garde sa direction actuelle.

## 2.4 La touche "LEFT" du clavier

dès que l'utilisateur appuie "LEFT" le joueur fait un pas chassé à gauche et garde également sa direction actuelle.

## 2.5 La touche "CTRL" du clavier

Quand l'utilisateur appuie sur "CTRL" le joueur tourne à gauche, sa position ne change évidement pas, mais sa direction change en fonction de sa direction actuelle, par exemple si sa direction courante est "Nord", sa direction après avoir appuyer sur "CTRL" sera "Ouest"

## 2.6 La touche "ALT" du clavier

Dès que l'utilisateur appuie sur "ALT" le joueur tourne à droite, sa position ne change également pas, mais sa direction change en fonction de sa direction actuelle, par exemple si sa direction courante est "Nord", sa direction après avoir appuyer sur "ALT" sera "Est".

## 2.7 La touche "ENTER" du clavier

Si le joueur et un des monstres sont assez proches, le joueur peut attaquer le monstre en appuyant sur "ENTER", sinon le clique sur cette touche n'a aucun effet.

# 3 La spécification formelle du projet

## 3.1 Service Mob

L'environnement d'un Mob peut être facilement connu à partir de Environnement, mais nous avons préférer garder l'observateur qui permet de récuperer ce dernier dans le cas d'une extention du jeu où celui ci peut avoir plusieurs environnements, et ainsi le Mob peut éventuellement changer d'environnement. Dans les méthodes Forward et Backward, nous avons ajouté une post-condition qui vérifie que le Mob reste dans la même direction lorsqu'il avance ou recule.

## 3.2 Service Environnement

Ce service contient un observateur qui sert à récupérer la liste des Mobs qu'il y à dans l'environnement. Nous pouvons ainsi ajouter un Mob facilement à ce dernier.

## 3.3 Service Combat

C'est le service qui gère les combats entre le joueur et les monstres

## 3.4 Service Player

Nous avons choisit d'ajouter un observateur à ce service qui est `TresorFound` : [Player] → Boolean, il permet d'observer si le joueur a trouvé le trésor. le choix de l'avoir rajouter dans ce service est pour le cas du même jeu en multi-joueurs, cela permettrai de connaître le joueur qui a trouvé le trésor.

## 3.5 Service Clef

Nous avons choisit d'ajouter un service Clef. La clef est inserée dans une case accessible depuis la position du joueur ( pas de mur et pas de porte fermée ).

**Service**: Clef
**Types**: bool, int
**Observators**: **const** `i`: [Clef] → int
**const** `j`: [Clef] → int
**const** `getEnv`: [Clef] → [Environment]
**Constructors**: `init`: Environment × int × int → [Clef]
**pre** `init(env,i,j)` **requires** i >= 0 **and** j>= 0
**and** i <= Environnement :: Height(getEnv())
**and** j <= Environnement :: Width(getEnv())
**Operators**: ⊤
**Observation**:
[invariant]: ⊤
[init]: `getI(init(env,i,j))` = `i`
`getJ(init(env,i,j))` = `j`
`getEnv(init(env,i,j))` = `env`

## 3.6 Service Trésor

Nous avons choisit d'ajouter un service Trésor afin de permettre une future perspective qui est d'avoir plusieurs trésor.

**Service**: Trésor
**Types**: bool, int, Cell
**Observators**: **const** `i`: [Trésor] → int
**const** `j`: [Trésor] → int
**const** `getEnv`: [Trésor] → [Environment]
**Constructors**: `init`: Environment ×int × int → [Trésor]
**pre** `init(env,i,j)` **requires** i >= 0 **and** j>= 0
**and** i <= Environnement :: Height(getEnv())
**and** j <= Environnement :: Width(getEnv())
**Operators**: ⊤
**Observation**:
[invariant]: ⊤
[init]: `getI(init(env,i,j))` = `i`
`getJ(init(env,i,j))` = `j`
`getEnv(init(env,i,j))` = `env`

## 3.7 Service Cellule

Vu que la possibilité d'accéder au contenu d'une cellule est nécessaire a la fois dans EditMap, Map et Environnement, nous avons décider d'ajouter cette méthode dans le service Cellule afin que ceci soit possible pour les trois derniers services
Nous n'avons pas besoin de vérifier que les coordonnées de la cellule sont comprises entre 0 et la largeur/hauteur de l'environnement, dans ce service car cela est déjà vérifié à l'initialisation de l'environnement.
Nous n'avons pas d'opérations pour modifier les coordonnées d'une cellule car on a pas besoin de ça dans notre cas.
L'option Nourriture est ajoutée dans cellule, quand le joueur tue le monstre, ce dernier se transforme en nourriture, il est bien sûr possible de rajouter de la nourriture dans des cases aléatoirement ou en les choisissant, ce qui explique

| | |
|---|---|
| **Service**: | Cellule |
| **Types**: | bool, int, Cell, OptionContentNo,So, OptionFoodFo,No |
| **Observators**: | **const** `i`: [Cellule] → int |
| | **const** `j`: [Cellule] → int |
| | `getNature`: [Cellule] → Cell |
| | `getContent`: [Cellule] → OptionContent |
| | `ContainsFood`: [Cellule] → OptionFood |
| **Constructors**: | `init`: int × int × Cell → [Cellule] |
| **Operators**: | `setContent`: [Cellule]× OptionContent → [Cellule] |
| notre choix. | `setFood`: [Cellule] → [Cellule] |
| | **pre** `setFood(C)` **requires** ContainsFood(C) = No |
| | `setNature` : [Cellule] × Cell → [Cellule] |
| **Observation**: | |
| [invariant]: | ⊤ |
| [init]: | `getI(init(i,j,c))` = `i` |
| | `getJ(init(i,j,c))` = `j` |
| | `getNature(init(i,j,c))` = `c` |
| [setContent]: | `getContent(setContent(C,content))` = **content** |
| [setNature]: | `getNature(setNature(C,cell)` = **cell** |

### 3.8 Service Combat

Le nombre de points de vie des monstres et des joueurs est initialisé à 3, quand le joueur frappe un monstre le nombre de points du monstre est décrementé de 1 et celui du joueur incrementé de 1 et c'est l'inverse si c'est le monstre qui frappe le joueur.
la méthode "frapper" du monstre s'excute aléatoirement (dans step)
Quand le joueur réussit à tuer le monstre ( nombre de points de vie du monstre = 0 ), il se transforme en nourriture, le joueur peut alors manger et gagner un point de vie.
Dans le cas contraire, si le monstre tue le joueur, alors la partie est finie, le joueur peut évidement relancer une partie ou quitter.
Le combat est initialisé uniquement avec le joueur, ensuite quand il est proche du monstre, ce dernier est ajouté au combat et debutCombat est initialisé à vrai.

**Service**: Combat

**Types**: bool, int, Cell, OptionContentNo,So, OptionFoodFo,No

**Observators**:
`getPlayer`: [Combat] → Player
`getVache`: [Combat] → Cow
`getEnv`: [Combat] → Environnement
`isFini`: [Combat] → boolean
`debutCombat`: [Combat] → boolean

**Constructors**:
`init`: Player × Vache → [Combat]
**pre** `init(p,v)` **requires** Player :: getHp(p) > 0 and
Cow :: getHp(p) > 0 and Environnement :: getEntities().size() >= 2

**Operators**:
`PlayerfrappeMonstre`: [Combat]→ [Combat]
**pre** `PlayerfrappeMonstre()` **requires** Player :: isEnvie(getPlayer()) = true
and Cow :: isEnvie(getCow()) = true and proche(getPlayer() , getVache())
`VachefrappePlayer`: [Combat]→ [Combat]
**pre** `VachefrappePlayer()` **requires** Player :: isEnvie(getPlayer()) = true
and Cow :: isEnvie(getCow()) = true and proche(getPlayer() , getVache())
`proche`: [Combat] × Player × Cow → [Combat]
**pre** `proche(C,p,v)` **requires**
Player :: getRow(p) == Cow :: getRow(v) and
| Player :: getCol(p) - Cow :: getCol(v | == 1 ) OR
Player :: getCol(p) == Cow :: getCol(v) and
| Player :: getRow(p) - Cow :: getRow(v) | == 1 )
and debutCombat(C)==true
`touché`: [Combat] × Entity × Entity → [Combat]
**pre** `touché(C,e1,e2)` **requires**
Entity :: getFace(e1) =(N,S) and Entity :: getCol(e1) = Entity :: getCol(e2)
and Entity :: getRow(e1) − Entity :: getRow(e2)== (1,-1) OR
Entity :: getFace(e1) =(E,W) and Entity :: getRow(e1) =Entity :: getRow(e2)

# 4 Implementation

L'interface graphique a été réalisée en utilisant JavaFX. Ci dessous un aperçu de cette dernière :

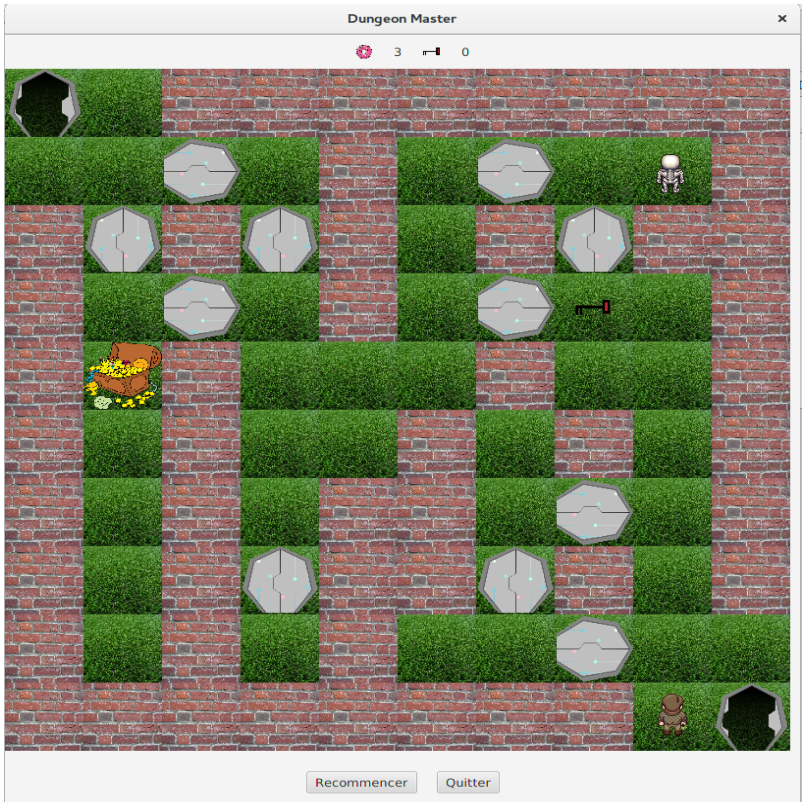


Figure 1: Aperçu sur l'interface graphique.

# 5 Tests

Les tests ont été réalisés pour les services Map, EditMap et Cellule.

# 6 Conclusion

Ce projet nous a permi d'appliquer les connaissances acquises pendant les cours, TD et TME du module CPS, la spécification et programmation par contrats nous a aidé à poser les bases du projet, bien cadrer chaque service de ce dernier avant de l'implémenter mais aussi à réduire le nombre de bugs grace aux tests.