Rapport de projet

# Dungeon Master
# –
# Design by Contracts

Vincent Ha Vinh, William Sergeant

Sorbonne Université – Master STL 2018

# Sommaire

# Introduction

Les Design by Contract est réputé pour être une puissante technique d'écriture de logiciels fiables. Les trois éléments clés de ce paradigme de programmation sont préconditions, postconditions et invariants. Ces éléments clés forment des contrats, règles immuables précisant le comportement attendu par un programme. C'est une méthode de programmation semi-formelle dont le but principal est de réduire le nombre de bugs dans les programmes.

## Métaphore du service commercial

Dans la réalité

- **Service :** fourni par un *fournisseur* à des *clients*
- **Contrat :**
    1. Les conditions devant être respectées par le client => **PRÉREQUIS** (précondition)
    2. La description du service fourni => **GARANTIES** (postconditions)

Un contrat peut être:

- **Honoré** => le client reçoit le service tel que défini par les GARANTIES
- **Rompu** par le client (les *PRÉREQUIS* ne sont pas assurés) => le *fournisseur* n'a pas d'obligation
- **Rompu** par le fournisseur (*GARANTIES* ne sont pas assurées) => le client peut demander des compensations

En informatique

- **Contrat du service :** interface avec des annotations (spécification semi-formelle)
- **Fournisseur :** classe(s) d'implémentation de l'interface
- **Client :** code dépendant de l'interface (decorators et contracts)
- **PRÉREQUIS :** tests de préconditons et invariants (pour chaque méthode du service)
- **GARANTIES :** tests de postconditions et invariants (pour chaque méthode du service)
-
- **Rompu:**
    - Par le client => test de précondition invalide (on déclenche une exception, un comportement défini)
    - Par le fournisseur => test de postcondition/invariants invalides

**Un contrat s'établi à partir des spécifications fournies par le client souhaitant recevoir en retour un programme dont il est assuré du bon fonctionnement.**

Nous avons réalisé dans ce projet une implémentation simplifiée du jeu « Dungeon Master » en utilisant le framework Decorator servant de liaison entre une interface de service et un contrat de spécification. Il en résulte une version jouable mais simple du Dungeon Master dont le comportement est garanti par les contrats découlants de la spécification semi-formelle.

# Manuel d'utilisation

## Instructions de compilation et exécution

Afin de compiler les sources lancez la commande :

```
ant create_jar
```

Pour lancer l'exécutable ;

```
java -jar dm.jar
```

## Prise en main et objectifs

### Le donjon

Vous êtes un jeune aventurier venu sauver son compagnon prisonnier des geôles de l'infâme Gargamel. Vous devrez aller au plus profond de son donjon en cherchant à chaque niveau l'escalier menant à l'étage inférieur.

Joueur

A l'initialisation de la partie, le joueur se trouve sur la case IN de la carte ressemblant à ceci :

Entrée

Le but est d'atteindre la case OUT sans se faire tuer par les monstres.

Sortie

Vous parcourez les couloirs du donjon en faisant parfois d'étonnantes rencontres.

Mur Sol

Les monstres possèdent une capacité de frappe circulaire, toutes les cases adjacentes directement (pas les diagonales) recevront des dommages.

Serpent

Chaque niveau est également peuplé avec des créatures pacifiques : les lapins. Celles-ci n'attaquent pas et ne font que se déplacer dans les méandres du donjon. Cependant vous serez amenez à les tuer si elles vous bloquent l'accès à un couloir !

Lapin

Si vous tombez face à une porte fermée, un bon coup de pied est toujours une solution valable (pour les fermer aussi).

DWC DWO DNC DNO

## Le déplacement

L'utilisateur a à sa disposition deux manières possibles de se déplacer :

Utilisation des boutons de l'interface :

Turn L Forward Turn R

Strafe L Backward Strafe R

Action

Inventory

Il peut aussi utiliser les touches selon la carte de contrôle suivante :

Z → forward

S → backward

Q → strafe left

D → strafe right

A → turn left

E → turn right

ENTER → action

La touche action permet de frapper le serpent ou le lapin se trouvant devant vous. Attention toutefois! Aucune de frappe n'est possible de côté !

Si vous vous trouvez face à une porte, l'action opérera l'ouverture ou la fermeture de la porte selon son état.

Vous remarquerez la présence d'un coffre à côté du bouton action, celui-ci devait servir à afficher l'inventaire mais n'ayant pu l'implémenter, il permet désormais de passer au step suivant sans effectuer d'action.

Afin de quitter la partie, cliquez sur la croix située en haut à droite de l'interface pour la clore.



Quitter

# Spécification

Nous avons travaillé à partir des spécifications données dans le sujet du projet et y avons apporté certaines modifications (voir écritures en rouge dans les spécifications ci-dessous.)

### Types

**type** Cell {**IN, OUT, EMP, WLL, DNO, DNC, DWO, DWC**}
**type** Dir {**N, S, W, E**}
**type** Command {**FF, BB, RR, LL, TL, TR, HIT**}
**type** Option[T] {**No, So(T)**}
**type** Set[T]
**type** Array[T1,...,TN]

## Service: Map

**Types**: bool, int, Cell

**Observators**:
**const** Height: [Map] → int
**const** Width: [Map] → int
CellNature: [Map] × int × int → Cell
**pre** CellNature(M,x,y) **requires** 0 ≤ x < Width(M) **and** 0 ≤ y < Height(M)

**Constructors**:
init: int × int → [Map]
**pre** init(w,h) **requires** 0 < w **and** 0 < h

**Operators**:
OpenDoor: [Map] × int × int → [Map]
**pre** OpenDoor(M,x,y) **requires** CellNature(M,x,y) ∈ {**DNC**, **DWC**}
CloseDoor: [Map] × int × int → [Map]
**pre** CloseDoor(M,x,y) **requires** CellNature(M,x,y) ∈ {**DNO**, **DWO**}

**Observations**:
[invariant]
- T

[init]
- Height(init(h,w)) = h
- Width(init(h,w)) = w

[OpenDoor]
- CellNature(M,x,y) = **DWC implies** CellNature(OpenDoor(M,x,y),x,y) = **DWO**
- CellNature(M,x,y) = **DNC implies** CellNature(OpenDoor(M,x,y),x,y) = **DNO**
- **forall** u ∈ [0; Width(M)-1] **forall** v ∈ [0; Height(M)-1],
  (u ≠ x **or** v ≠ y) **implies** CellNature(OpenDoor(M,x,y),u,v) = CellNature(M,u,v)

[CloseDoor] : : :
Similaire pour CloseDoor

# Service: EditMap

**refines** Map
**Types**: bool, int, Cell

**Observators**:
isReachable: [EditMap] × int × int × int × int → bool
**pre** isReachable(M,x1,y1,x2,y2) **requires**
CellNature(M,x1,y1) ≠ **WLL and** CellNature(M,x2,y2) ≠ **WLL**
isReady: [EditMap] → bool

**Constructors**:
Ø

**Operators**:
SetNature: [EditMap] × int × int × Cell → [EditMap]
**pre** SetNature(M,x,y) **requires** 0 ≤ x < Width(M) **and** 0 ≤ y < Height(M)

**Observation**:
[invariant]:

- isReachable(M,x1,y1,x2,y2) =
  **exists** P **in** Array[int,int], P[0] = (x1,y1) **and** P[size(P)-1] = (x2,y2)
  **and forall** i **in** [1;size(P)-1], (P[i-1]=(u,v) **and** P[i]=(s,t)) **implies** $(u - s)^2 + (v - t)^2 = 1$
  **and forall** i **in** [1;size(P)-2], P[i-1]=(u,v) **implies** CellNature(M,u,v) ≠ **WLL**
- isReady(M) =
  **exists** xi,yi,xo,yo **in** $int^4$,
  CellNature(M,xi,yi) = **IN and** CellNature(M,xi,yi) = **OUT**
  **and** isReachable(M,xi,yi,xo,yo)
  **and forall** x,y **in** $int^2$, x ≠ xi **or** y ≠ yi **implies** CellNature(M,x,y) ≠ **IN**
  **and forall** x,y **in** $int^2$, x ≠ xo **or** y ≠ yo **implies** CellNature(M,x,y) ≠ **OUT**
  **forall** x,y **in** $int^2$, CellNature(M,x,y) ∈ {**DNO**, **DNC**} **implies**
  CellNature(M,x+1,y) = CellNature(M,x-1,y) = **EMP and**
  CellNature(M,x,y-1) = CellNature(M,x,y+1) = **WLL**
  **forall** x,y **in** $int^2$, CellNature(M,x,y) ∈ {**DWO**, **DWC**} **implies**
  CellNature(M,x+1,y) = CellNature(M,x-1,y) = **WLL and**
  CellNature(M,x,y-1) = CellNature(M,x,y+1) = **EMP**

[SetNature]:

- CellNature(SetNature(M,x,y,Na),x,y) = Na
- **forall** u,v **in** $int^2$, u ≠ x **or** v ≠ y **implies** CellNature(SetNature(M,x,y),u,v) = CellNature(M,u,v)

## Service: Environment

**includes** Map
**Types**: bool, int, Cell, Mob

**Constructor:**
init: Map → [Environment]

**Observators** :
CellContent: [Environment] × int × int → Option[Mob]

**Operators**:
CloseDoor: [Environment] × int × int → [Environment]
**pre** CloseDoor(M,x,y) **requires** CellContent(M,x,y) = **No**

SetContent: [Environment] × int × int × Mob → [Environment]
**pre** SetContent(M,x,y,C) **requires** CellNature(M,x,y) ∉ {**WLL**, **DNC**, **DWC**}
**and** C ≠ No **implies** CellContent(M,x,y) = No

**Observations**:
[SetContent]

- CellContent(SetContent(M,x,y,C),x,y) = C
- **forall** u,v **in** $int^2$,
  u ≠ x **or** v ≠ y **implies** CellContent(SetContent(M,x,y, C),u,v) = CellContent(M,u,v)

## Service: Mob

**Types**: bool, int, Cell

**Observators** :
Env: [Mob] → Environment
Col: [Mob] → int
Row: [Mob] → int
Face: [Mob] → Dir

**Constructors**:
init: Environment × int × int × Dir → [Mob]
**pre** init(M,x,y,D) **requires** Environment::CellNature(M,x,y) ≠ **IN**
**and** Environment::CellContent(M,x,y) = **No**

**Operators**:
Forward: [Mob] → [Mob]
Backward: [Mob] → [Mob]
TurnL: [Mob] → [Mob]
TurnR: [Mob] → [Mob]
StrafeL: [Mob] → [Mob]
StrafeR: [Mob] → [Mob]

**Observations**:
[invariant] :
- 0 ≤ Col(M) < Environment::Width(Envi(M))
- 0 ≤ Row(M) < Environment::Height(Envi(M))
- Environment::CellNature(Envi(M),Col(M),Row(M)) ∉ {**WLL**, **DNC**, **DWC**}

[init] :
- Col(init(E,x,y,D)) = x
- Row(init(E,x,y,D)) = y
- Face(init(E,x,y,D)) = D
- Envi(init(E,x,y,D)) = E

[Forward]:
- Face(M)=**N implies**
  Environment::CellNature(Envi(M),Col(M),Row(M)+1) ∈ {**EMP**, **DNO**}
  **and** Row(M)+1 < Environment::Height(Envi(M))
  **and** Environment::CellContent(Envi(M),Col(M),Row(M)+1) = **No**
  **implies** Row(Forward(M)) = Row(M) + 1
  **and** Col(Forward(M)) = Col(M)
- Face(M)=**N implies**
  Environment::CellNature(Envi(M),Col(M),Row(M)+1) ∉ {**EMP**, **DNO**}
  **or** Row(M)+1 ≥ Environment::Height(Envi(M))
  **or** Environment::CellContent(Envi(M),Col(M),Row(M)+1) ≠ **No**
  **implies** Row(Forward(M)) = Row(M)
  **and** Col(Forward(M)) = Col(M)
- Face(M)=**E implies**
  Environment::CellNature(Envi(M),Col(M)+1,Row(M)) ∈ {**EMP**, **DWO**}
  **and** Col(M)+1 < Environment::Width(Envi(M))
  **and** Environment::CellContent(Envi(M),Col(M)+1,Row(M)) = **No**
  **implies** Row(Forward(M)) = Row(M)

**and** Col(Forward(M)) = Col(M) + 1

- Face(M)=**E implies**
  Environment::CellNature(Envi(M),Col(M)+1,Row(M)) ∉ {**EMP**, **DWO**}
  **or** Col(M) ≥ Environment::Width(Envi(M))
  **or** Environment::CellContent(Envi(M),Col(M)+1,Row(M)) ≠ **No**
  **implies** Row(Forward(M)) = Row(M)
  **and** Col(Forward(M)) = Col(M)
- Face(M)=**S** implies
  Environment::CellNature(Envi(M),Col(M),Row(M)-1) ∈ {**EMP**, **DNO**}
  **and** Row(M)-1 ≥ 0
  **and** Environment::CellContent(Envi(M),Col(M),Row(M)-1) = **No**
  **implies** Row(Forward(M)) = Row(M) - 1
  **and** Col(Forward(M)) = Col(M)
- Face(M)=**S implies**
  Environment::CellNature(Envi(M),Col(M),Row(M)-1) ∉ {**EMP**, **DNO**}
  **or** Row(M)-1 < 0
  **or** Environment::CellContent(Envi(M),Col(M),Row(M)-1) ≠ **No**
  **implies** Row(Forward(M)) = Row(M)
  **and** Col(Forward(M)) = Col(M)
- Face(M)=**W implies**
  Environment::CellNature(Envi(M),Col(M)-1,Row(M)) ∈ {**EMP**, **DWO**}
  **and** Col(M)-1 ≥ 0
  **and** Environment::CellContent(Envi(M),Col(M)-1,Row(M)) = **No**
  **implies** Row(Forward(M)) = Row(M)
  **and** Col(Forward(M)) = Col(M) - 1
- Face(M)=**W implies**
  Environment::CellNature(Envi(M),Col(M)-1,Row(M)) ∉ {**EMP**, **DWO**}
  **or** Col(M)-1 < 0
  **or** Environment::CellContent(Envi(M),Col(M)-1,Row(M)) ≠ **No**
  **implies** Row(Forward(M)) = Row(M)
  **and** Col(Forward(M)) = Col(M)

[Backward]

- *post-conditions similaires à Forward, les post-conditions pour Face(M)=**N** sont celles de Face(M)=**S** et inversement, idem pour **W** ⇔ **E***

[StrafeR]

- *post-conditions similaires à Forward, mais on change: les post-conditions pour Face(M)=**N** deviennent celles de Face(M)=**W**, celles de **E** deviennent celles de **N**, celles de **S** deviennent celles de **E**, celles de **W** deviennent celles de **S**.*

[StrafeL]

- *post-conditions similaires à Forward, mais on change: les post-conditions pour Face(M)=**N** deviennent celles de Face(M)=**E**, celles de **E** deviennent celles de **S**, celles de **S** deviennent celles de **W**, celles de **W** deviennent celles de **N**.*

[TurnLeft]:

- Face(M)=**N implies** Face(TurnLeft(M))=**W**
- Face(M)=**W implies** Face(TurnLeft(M))=**S**
- Face(M)=**S implies** Face(TurnLeft(M))=**E**
- Face(M)=**E implies** Face(TurnLeft(M))=**N**

[TurnRight]:

- Face(M)=**N implies** Face(TurnLeft(M))=**E**
- Face(M)=**W implies** Face(TurnLeft(M))=**N**
- Face(M)=**S implies** Face(TurnLeft(M))=**W**
- Face(M)=**E implies** Face(TurnLeft(M))=**S**

## Service: Entity

**includes** Mob

**Observator**:
Hp: [Entity] → int

**Constructor**:
init: Environment × int × int × Dir × int → [Entity]
**pre** init(E,x,y,D,h) **requires** h > 0

**Operator**:
step: [Entity] → [Entity]
hit: [Entity] → [Entity]
takeHit: [Entity] → [Entity]
**pre** takeHit(E) **requires** Hp(E) > 0

**Observations**:
[init]:
- Hp(init(E,x,y,D,h)) = h

[takeHit]:
- Hp(takeHit(E)) = Hp(E) – 1

## Service: Cow

**includes** Entity

**Constructor**:
init: Environment × int × int × Dir × int → [Entity]
**pre** init(E,x,y,D,h) **requires** 3 ≤ h ≤ 4

**Observations**:
[step]:
- Col(M) - 1 ≤ Col(step(M)) ≤ Col(M) + 1
- Row(M) - 1 ≤ Row(step(M)) ≤ Row(M) + 1

[hit]:
- Ø

## Service: Player

**includes** Entity

**Observator**:
LastCom: [Player] → Option[Command]
Content: [Player] × int × int → Option[Mob]
**pre** Content(P,x,y) **requires** x ∈ {-1,0,1} and y ∈ {-1,+3}
Nature: [Player] × int × int → Cell
**pre** Nature(P,x,y) **requires** x ∈ {-1,0,1} and y ∈ {-1,+3}
Viewable: [Player] × int × int → bool
**pre** Nature(P,x,y) **requires** x ∈ {-1,0,1} and y ∈ {-1,+3}

**Operators**:
SetLastCom: [Player] × Option[Command] → [Player]

**Observations**:
[invariant]:

- Face(P) = **N implies** Content(P,u,v) = Environment:CellContent(Envi(P),Col(P)+u,Row(P)+v)
- Face(P) = **N implies** Nature(P,u,v) = Environment:CellNature(Envi(P),Col(P)+u,Row(P)+v)
- Face(P) = **S implies** Content(P,u,v) = Environment:CellContent(Envi(P),Col(P)-u,Row(P)-v)
- Face(P) = **S implies** Nature(P,u,v) = Environment:CellNature(Envi(P),Col(P)-u,Row(P)-v)
- Face(P) = **E implies** Content(P,u,v) = Environment:CellContent(Envi(P),Col(P)+v,Row(P)-u)
- Face(P) = **E implies** Nature(P,u,v) = Environment:CellNature(Envi(P),Col(P)+v,Row(P)-u)
- Face(P) = **W implies** Content(P,u,v) = Environment:CellContent(Envi(P),Col(P)-v,Row(P)+u)
- Face(P) = **W implies** Nature(P,u,v) = Environment:CellNature(Envi(P),Col(P)-v,Row(P)+u)
- **forall** u,v **in** [-1,1] × [-1,1], Viewable(P,u,v)
- Viewable(P,-1,2) = Nature(P,-1,1) ∉ {**WALL**, **DWC**, **DNC**}
- Viewable(P,0,2) = Nature(P,0,1) ∉ {**WALL**, **DWC**, **DNC**}
- Viewable(P,1,2) = Nature(P,1,1) ∉ {**WALL**, **DWC**, **DNC**}
- Viewable(P,-1,3) = Nature(P,-1,2) ∉ {**WALL**, **DWC**, **DNC**} **and** Viewable(P,-1,2)
- Viewable(P,0,3) = Nature(P,0,2) ∉ {**WALL**, **DWC**, **DNC**} **and** Viewable(P,0,2)
- Viewable(P,1,3) = Nature(P,1,2) ∉ {**WALL**, **DWC**, **DNC**} **and** Viewable(P,1,2)

[step]:

- LastCom(P)=**FF implies** step(P) = Forward(P)
- LastCom(P)=**BB implies** step(P) = Backward(P)
- LastCom(P)=**LL implies** step(P) = StrafeLeft(P)
- LastCom(P)=**RR implies** step(P) = StrafeRight(P)
- LastCom(P)=**TL implies** step(P) = TurnLeft(P)
- LastCom(P)=**TR implies** step(P) = TurnRight(P)
- LastCom(P)=**HIT implies** step(P) = Hit(P)

[SetLastCom]

- LastCom(SetLastCom(P,C)) = C

[Hit]

- Face(M)=**N implies**
  - Environment::CellNature(Envi(M),Col(M),Row(M)+1) = **DWO implies** CloseDoor(Envi(M),Col(M),Row(M)+1)
  - Environment::CellNature(Envi(M),Col(M),Row(M)+1) = **DWC implies** OpenDoor(Envi(M),Col(M),Row(M)+1)

- Environment::CellContent(Envi(M),Col(M),Row(M)+1) ≠ **No** **implies** takeHit(CellContent(Envi(M),Col(M),Row(M)+1))
- Environment::CellNature(Envi(M),Col(M),Row(M)+1) ∉ {**DWO**, **DWC**} **and** Environment::CellContent(Envi(M),Col(M),Row(M)+1) = **No** **implies** Envi(Hit(P)) = Envi(P)

- *De même pour les autres directions (en changeant les coordonnées) (W et E fonctionnent avec DNO et DNC)*

## Service: Engine

**Observator**:
Envi: [Engine] → Environment
Entities: [Engine] → Array[Entity]
getEntity: [Engine] × int → Entity

**Constructor**:
init: Environment → [Engine]

**Operator**:
removeEntity: [Engine] × int → [Engine]
**pre** removeEntity(E,i) **requires** 0 ≤ i < size(Entities(E))
addEntity: [Engine] × Entity → [Engine]
step: [Engine] → [Engine]
**pre** step() **requires forall** i **in** [0;size(Entities(E))-1], Entity::Hp(getEntity(E,i))>0

**Observations**:
[invariant]:
- **forall** i **in** [0;size(Entities(E))-1], Entity::Envi(getEntity(E,i))=Envi(E)
- **forall** i **in** [0;size(Entities(E))-1], Entity::Col(getEntity(E,i))=x **and** Entity::Row(getEntity(E,i))=y **implies** Environment::CellContent(Envi(E),x,y) = getEntity(E,i)

[removeEntity]:
- size(Entities(removeEntity(E,i))) = size(Entities(E)) - 1
- **forall** k **in** [0,i-1], getEntity(removeEntity(E,i),k)) = getEntity(E,k)
- **forall** k **in** [i,size(Entities(E))-2], getEntity(removeEntity(E,i),k)) = getEntity(E,k+1)

[addEntity]:
- size(Entities(addEntity(E,e))) = size(Entities(E)) + 1
- **forall** k in [0,size(Entities(E))-1], getEntity(addEntity(E,e),k)) = getEntity(E,k)
- getEntity(addEntity(E,e),size(Entities(E))) = e

*En cherchant une observation pour step on peut être tenté d'écrire:*
*forall i in size(Entities(E)), getEntity(step(E),i) = Entity::step(getEntity())*
*Mais cette observation ne serait pertinente que si les méthodes step des différentes entitées étaient indépendantes les unes des autres. Mais ce n'est pas le cas (les déplacements sont dépendants de l'ordre des entitées, plus tard on ajoutera le combat).*

# Travail effectué

## Modifications remarquables dans les spécifications:

EnvironmentService:

Le constructeur du service Environnement prend un service Map en argument.
Environnement, qui étend Map, déléguera donc les appels d'OpenDoor et getNature au service Map passé en argument à l'initialisation.

Ainsi, on se sert de la classe Environnement comme d'une Map qui :

- a un opérateur CloseDoor plus restrictif (ne peut fermer une porte sur un Mob)
- qui possède en plus une notion de contenu (Mob ou pas dans une cellule) par rapport à une Map classique.

Nous avons donc ajouté un opérateur setContent permettant de manipuler le contenu d'une cellule (à la manière dont EditMap manipule la nature d'une cellule).

A noter:

Le service passé en argument du constructeur d'Environnement est une simple Map, pas une EditMap. Environnement ne peut donc pas modifier librement la nature d'une cellule autrement qu'avec OpenDoor/CloseDoor.
Si Environnement permet un contrôle total du contenu d'une cellule, seule EditMap permet la même chose sur la nature d'une cellule.
Ceci dans l'optique d'empêcher des changements accidentels de terrain dû à une mauvaise implémentation lors du parcours du labyrinthe.

Hit, takeHit:

Nous avons ajouté le mécanisme d'interaction entre une entité et son environnement (au sens général du terme). Ce mécanisme se traduit par la nouvelle option HIT dans l'énumération Commande et les opérateurs hit et takeHit du service Entité.
Dans le cas général (monstres et joueur), une entité peut Hit, et l(es) entité(s) touchée(s) déclenche son opérateur takeHit pour subir la conséquence de l'interaction.

La portée, le champ et le nombre d'entités touchées par le Hit d'une entité varie selon le type de l'entité. Nous avons choisi de fixer dans la spécification le Hit des vaches pour que celui-ci ne fasse...rien. Les vaches sont bien entendues gentilles et herbivores, elles ne blessent donc pas les gens.

Si un développeur veut un autre comportement pour Hit, il est libre de modifier la méthode Hit d'une implémentation d'EntityService, qui n'a pas de postcondition à Hit.
Par exemple, dans notre implémentation fonctionnelle, le Hit d'EntityImpl affecte toutes les autres entités adjacentes à l'entité actrice. Il déclenche donc 4 takeHit (au maximum).

Enfin, nous avons fixé dans la spécification le comportement du Hit du joueur :

- Le joueur peut Hit ce qui est en face de lui seulement.
- si c'est une porte, le Hit ouvre ou ferme la porte (actionne OpenDoor/CloseDoor sur les coordonnées de la case de devant)
- Si c'est une entité, actionne le takeHit de cette entité.

## Difficultés dans l'écriture des contrats:

Step de Player:

Le comportement de l'opérateur Step de PlayerService varie entre Forward, Backward,... (méthodes de déplacement), Hit, selon la dernière commande du joueur.
C'est donc un service "s'utilise" lui-même, dans le sens où step appelle une autre méthode de son service.
Cela pose problème pour brancher le contrat et tester, car PlayerContract.step() va déléguer à l'implémentation branchée, qui elle va exécuter sa fonction Forward/Backward/etc (selon le cas). Cela "court-circuite" le contrat puisqu'on aurait voulu que ce soit le Forward/Backward/etc de PlayerContract qui soit exécuté pour tester ses pré et postconditions.

Pour que le contrat vérifie les postconditions de step,

[step]:

- LastCom(P)=**FF implies** step(P) = Forward(P)
- LastCom(P)=**BB implies** step(P) = Backward(P)
- LastCom(P)=**LL implies** step(P) = StrafeLeft(P)
- LastCom(P)=**RR implies** step(P) = StrafeRight(P)
- LastCom(P)=**TL implies** step(P) = TurnLeft(P)
- LastCom(P)=**TR implies** step(P) = TurnRight(P)
- LastCom(P)=**HIT implies** step(P) = Hit(P)

Nous avons donc recopié l'ensemble des captures de toutes ces autres méthodes du contrat, puis après la délégation, l'ensemble des postconditions de toutes ces méthodes, dans la fonction step de PlayerContract.
Si à première vue, le code semble fastidieux, il reste en fait assez efficace : Les captures des méthodes de déplacement étant les mêmes, elles ne sont recopiées (donc effectuées) qu'une seule fois. Les postconditions sont structurées dans un switch : En effet, une seule implication peut être vraie à la fois.

Au final, l'exécution du step de PlayerContract est aussi rapide qu'une exécution des méthodes de placements ou que Hit.

# Remarques dans l'écriture des tests:

Structure :

La structure pour un test de service, par exemple MapService, est la suivante :

AbstractMapTest.java, contient :

- l'attribut Service à tester, MapService.
- les attributs Service utilisés, EditMapService.
- les getters/setters, le @AfterTest qui nettoie les données après chaque test.
- les méthodes @Test testant les couvertures des préconditions, des transitions, et des cas d'utilisation.

Map1Test.java, contient :

- la méthode @BeforeTest qui instancie l'implémentation à tester et les implémentations des services à utiliser.

Tous les tests n'ont pas pu être créés par manque de temps.
Actuellement, nous avons branché les contrats pour nos tests.
Avec JUnit, Nous avons utilisé des objets **Failure** pour signifier le cas où le test rate (oracle échoue), montrant un problème au niveau du respect du contrat par l'implémentation, ou de cohérence dans la spec. Un objet **Error** est levé quand il y a une erreur inattendue (un bug dans l'implémentation/le contrat, levant une exception au Runtime. Ou bien une Precondition/PostCondition/InvariantError dans un contrat inattendu par le test, comme une PreconditionError lors d'un test de transition).

Initialiser la Map avant un test :

La spécification du service Map ne possède aucun invariant. Cela signifie que nous n'avons aucune certitude de l'existence d'une porte, sortie, ou entrée dans la map.

Alors, pour tester les opérateurs OpenDoor/CloseDoor, nous avons procédé de la façon suivante :

Dans Map1Test (qui instancie les services attributs):

```
EditMapService editmap = new EditMapImpl();
setEditMap(editmap);
setMap(new MapContract(editmap));
```

Dans le @Test de précondition d'init() de AbstractMapTest, cas négatif:

```
editmap.init(14, 35);
editmap.setNature(10, 10, Cell.IN);
...
//operation
map.openDoor(10, 10);
...
```

On crée une instance d'EditMapImpl, qu'on encapsule à la fois dans le MapService à tester et dans le EditMapService utilisé.
Comme ça, avec "l'interface d'édition" EditMapService, on initialise notre map pour le test,

puis on test cette même map avec “l’interface de visualisation” MapService. Sur l’aspect fonctionnel, cette façon de faire est conforme à la méthode MBT, puisqu’on teste le comportement du service MapService en boîte noire.

De manière similaire, pour tester tout service qui nécessite de la configuration de map (placer des portes et des murs, placer une cellule vide pour y mettre un monstre), nous aurons un attribut EditMapService dans le test, et un attribut Environment qui sera initialisé avec notre EditMapService.

Tout cela est possible car EditMapService **raffine** MapService, il peut donc remplacer tout MapService nécessaire dans un assemblage de Services composants, et offrira à la fois la possibilité de faire la même chose qu’un MapService et des fonctionnalités supplémentaires d’édition que le MapService ne possède pas.

Par exemple, dans AbstractMobTest, nous avons besoin de modifier la natures de cellules pour rendre déterministe notre @Test d’initialisation, cas négatif (On ne peut se reposer sur la génération de map aléatoire offerte par notre implémentation) :

```
//init
editmap.init(14,35);
editmap.setNature(7, 23, Cell.DNC);
env.init(editmap);
...
//operation
mob.init(env, 7, 23, Dir.E);
```

# Génération de cartes aléatoires

La carte est générée aléatoirement avec l'algorithme du "marcheur ivre".

Celui-ci prend deux valeurs numériques en paramètres :

- Le nombre maximum de couloirs (max_cor)
- La longueur maximale d'un couloir (max_len)

Ensuite une matrice de la taille désirée (on peut éventuellement rajouter deux paramètres pour définir sa taille) est créée avant de la remplir entièrement de murs.
Une case est choisie aléatoirement et constitue l'entrée du niveau.

Puis le "marcheur" se déplace de la manière suivante :

1. Il choisit une distance à parcourir en ligne droite entre 0 et max_len
2. Une fois arrivé au bout, il décrémente max_cor de 1 puis choisit une nouvelle direction aléatoirement
3. Si max_cor = 0, le "marcheur ivre" s'arrête et cette case est désignée comme sortie sinon, reprendre à l'étape 1.

Une fois la carte générée, il faut la remplir avec des monstres, des créatures pacifiques, des portes, et autres. On tire alors une case au hasard parmi les cases valides (cellules vides, sans aucune entité dessus) et on place soit une porte soit une créature.

Le ratio optimal du nombre de portes sur nombre de cases a été défini après plusieurs tests. Ce ratio est de 2 portes pour 100 cases (tout type confondu).
Nous n'avons pas réussi à dégager de paramètres optimaux pour max_len et max_cor mais avons pu faire quelques observations avec des études de sensibilités :

- Plus max_len se rapproche de la largeur / hauteur, plus les risques qu'un couloir faisant le tour complet de la carte se forme sont élevés
- Plus max_cor est élevé, plus la carte comprend de grands espaces

Nous avons apporté quelques modifications à l'algorithme du marcheur afin de générer des cartes plus adaptées au contexte de donjon, comme lui faire faire plus de demi-tours pour créer des culs-de-sac.

Ainsi, lorsque l'on atteint la sortie d'un niveau, un autre est généré donnant potentiellement au jeu une durée de vie infinie.

Afin de mieux vous y retrouver dans le donjon, une carte vous sera proposée en sus de l'interface de base. Attention si vous la fermez cela quittera le jeu également.

# Interface

L'interface est réalisée en **Java**. On utilise la librairie **JavaFX** ainsi que des ressources externes (images, layouts) pour les vues avec **FXML** permettant l'implémentation de scènes et de feuilles **CSS** afin de définir des styles pour les groupes d'éléments.
**JavaFX** fût choisi car nous disposons d'un socle de connaissances quant à son utilisation ce qui nous a permis d'être plus efficaces et innovants lors du développement de cette version de Dungeon Master. L'introduction de l'utilisation de documents **FXML** et **CSS** en adéquation avec le code fonctionnel est cependant une nouveauté pour nous. Cela nous a permis de mieux utiliser les composants graphiques de **JavaFX** ainsi que de faciliter les interactions entres les vues et les contrôleurs tout en élargissant notre cercle de compétences. Nous utilisons donc un framework simili MVC, le Modèle et la Vue étant sensiblement entrelacés.

## Vues

### Liaisons

Les interactions entre l'interface et son contrôleur se font majoritairement par le biais de boutons, lesquels se sont vu attribuer une fonction spécifique lors de leur appel.

En outre au lieu de devoir créer, instancier, personnaliser le bouton et par la suite lui affecter un écouteur d'événement (event listener) définissant le comportement à avoir (en cas de clique du bouton par exemple) celui-ci est simplement défini par une ligne dans le fichier FXML correspondant :

<Button fx:id="**buttonForward**" onAction="**#forward**" text="Submit word" id="**forward**"/>

Le champ ***fx:id*** est l'identité du bouton dans le fichier FXML, celui-ci peut être utilisé par le contrôleur java associé. L'instruction dans le code du contrôleur **@FXML Button forward** permet de récupérer l'instance de ce bouton (après le chargement du fichier FXML, moment où les instanciations des composants sont effectuées). Ainsi celui-ci sera directement utilisable et référera au même composant dans l'interface et dans le code.

L'autre champ ***onAction*** défini la fonction associée en cas d'activation du contrôleur. Ici, lors du clique sur le bouton, la fonction **forward()** sera exécutée. Ces fonctions se comportant tels les event Listeners et Handlers, il est possible de passer un argument de type **Event**. Il est également possible de paramétrer un champ user-data dans le composant FXML et de récupérer cette valeur lors de l'appel.

Enfin, le champ ***id*** est une référence interne au composant et servira notamment à définir les références aux fichiers **CSS** afin d'appliquer les styles appropriés aux composants de manière automatique. Par exemple, le fichier forward.gif est associé au bouton sus-cité. Si le fichier est changé, le nouveau design sera chargé sans devoir modifier le code du jeu.

### Fonctionnalités

La vue principale est décomposée en deux parties :

- La carte qui est un affichage des cases visibles par le joueur
- Les contrôles permettant d'interagir avec le jeu

Pour plus de confort, des touches ont été définies afin de ne pas avoir à cliquer sur les boutons:

- ZQSD pour les déplacements
- A E pour les rotations
- ENTER pour les actions (frapper / ouvrir ou fermer une porte)

## Threads

Ce contrôleur communique efficacement avec l'interface afin de mettre à jour celle-ci avec les informations correspondantes. Afin de passer la main au thread JavaFX pour changer les données de l'écran on utilisera la méthode **Platform.runlater( ()→ {runnable*}) ;***
Cela nous permettra d'effectuer des modifications mineures de l'affichage tel que le changement de point de vie lorsque le thread sera disponible.

# Conclusion

Nous avons implémenté avec succès une version revisitée du Dungeon Master selon la méthode du Design by Contract suivant le framework Decorator.

Malgré certaines difficultés pour appliquer à plus grande échelle les notions apprises ce semestre, nous avons surmonté les problèmes en cherchant des solutions innovantes et respectueuses des enjeux du Design by Contract.

Toutefois des améliorations nous apparaissent évidentes. Pour exemple la rotation de la vue correspondant aux cases visibles par le personnage en fonction de son orientation ou encore l'implémentation d'autres dynamiques de jeu telles que les clés et coffres ou encore les boss.

Ce projet revêtant un certain aspect de Game Design, même si ce n'était pas l'enjeu principal, fût agréable à mener de bout en bout et enrichissant pour chacun des membres du groupe.

*Remerciements distingués à StackOverflow.*

*Le lien Github de notre projet :*
*https://github.com/Hephixor/CPS_DUNGEON_MASTER_PROJECT*