SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Sujet Devoir CPS : *Dungeon Master*

Projet

Le but du projet est de donner une spécification d'un jeu similaire à *Dungeon Master* dont le cahier des charges est partiellement décrit dans le sujet d'Examen Réparti 1, d'implémenter cette spécification selon la méthode *Design-by-Contract* vue en cours, et d'écrire une implémentation pour le jeu.
Le but du projet est de proposer des défis de spécification et d'implémentation de contrats. L'implémentation du jeu a une importance moindre. Toute fonctionnalité implémentée mais non spécifiée a de fortes chances d'être négligée dans l'évaluation.
Le but du projet n'est pas de se rapprocher au maximum de jeux existants. Les initiatives personnelles et originales sont encouragées.

Rendu

Le rendu est composé d'une unique archive contenant:

- la spécification semi-formelle des services utilisés par le projet sous forme d'interface **Java** (e.g. une interface `Player`),
- l'implémentation de spécification par des contrats selon le motif *ecorator* vu en cours (e.g. deux classes `PlayerDecorator` et `PlayerContract`),
- (au moins) deux implémentations des services spécifiés, une implémentation correcte et une implémentation buggée (e.g. deux classes `PlayerImpl` et `PLayerBugImpl`).
- une série de test **JUnit** pertinents, élaborés selon la méthode MBT.
- un fichier `build.xml` permettant la compilation, l'execution et le test des implémentations.
- un rapport en `pdf`.

Etape 1: Compléter l'examen

La spécification et l'implémentation des services donnée dans les cinq exercices de l'examen doivent être entièrement terminée dans le projet.

Etape 2: Fonctionnalités imposées

Les fonctionnalités suivantes doivent être présentes pour que le projet obtienne une évaluation correcte:

- **Jeu**: Le programme du jeu doit intégrer un éditeur, et permettre le lancement d'une partie et l'intialisation d'un moteur de jeu, qui vérifie des conditions de victoire et de défaite.

- **Monstres et Combat**: Le jeu doit contenir des monstres qui peuvent suivre et attaquer le joueur quand ils se trouvent à proximité de lui. Le joueur doit pouvoir effectuer une action de combat contre un monstre devant lui. Le moteur doit maintenir la cohérence du combat (supprimer un monstre quand ses points de vie sont négatifs ou nuls).

- **Trésor**: La grille doit contenir un trésor accessible dans une pièce accessible depuis l'entrée. Le but pour le joueur est de ramasser le trésor puis de trouver la sortie.

- **Affichage et Interface**: Cette fonctionnalité est la seule pour laquelle il n'est pas demandé de spécification. L'affichage peut être à la première personne (préférentiellement): la nature et le contenu des cases visibles parmi les neuf cases devant le joueur est représentée à l'écran. Alternativement, si le projet est ambitieux sur d'autres points, l'affichage peut se faire en vue du dessus (en représentant les 15 cases proches du joueur), y compris en ASCII. Les commandes de l'interface peuvent être entrées au clavier ou en cliquant sur des boutons (l'une ou l'autre méthode, ou les deux).

- **Gestion des grilles** Le joueur doit pouvoir charger une grille existante ou éditer une grille à l'aide d'une interface simple. Le jeu est capable de vérifier qu'une grille est licite. Deplus, le jeu doit intégrer des fonctions basiques de génération aléatoire de grilles.

Etape 3: Extensions possibles

Pour obtenir l'évaluation maximale, le projet doit en plus spécifier et implémenter plusieurs extensions parmi les suivantes[1]:

- **Clefs**: Certaines portes peuvent être verrouillées et nécessitent de trouver la bonne clef pour être ouverte.

- **Butin**: Le joueur peut trouver du butin (au sol ou en triomphant des montres) qui améliorent ses statistiques de combat, ou augmentent un éventuel score final.

- **Combat amélioré**: Le joueur peut utiliser différents types d'attaques, d'armes, de sortilèges et de protection pour se battre contre des monstres variés, disposant de compétences spéciales (faire passer son tour au joueur, le forcer à reculer, . . .)

- **Enigme**: Le joueur doit résoudre des énigmes à base de leviers, interrupteurs, pièges pour parvenir à la sortie ou au trésor.

- **Survie**: Le joueur doit collecter de la nourriture, s'alimenter et dormir (à l'abri de tout monstre) régulièrement.

- **Lumière**: Le champ de perception du joueur dépend de la force de la source de lumière qu'il porte (torch, lanterne) et des sources de lumière présentes sur la grille.

- **Multijoueur**: Plusieurs joueurs peuvent jouer en même temps, en se partageant l'écran.

- **Génération améliorée**: Le jeu intègre une génération automatique de niveau selon des paramètres de difficulté, et peut générer des "donjons" composés de plusieurs grilles (la sortie de la grille précédente amène à l'entrée de la grille suivante).

Rapport

Le rapport se compose de 3 parties:

1. Un manuel d'utilisation succint de l'implémentation.

[1] Les extensions issues d'idées originales, ou de jeux vidéo existants sont bien sûr encouragées.

2. La spécification formelle complète du projet.

3. Un rapport de projet proprement dit, se concentrant sur les choix et les difficultés de spécification, d'implémentation, de tests, exhibant des exemples choisis pour leur pertinence.

Soutenance

Une soutenance de 15 minutes se compose d'environ:

1. 10 minutes de rapport: présentation (rapide) du projet, puis exploration de différents cas de spécification, contrats ou tests pertinents.

2. 5 minutes de démonstration.

Le contenu de la soutenance doit se concentrer sur quelques points précis de spécifications ou de tests. Il est conseillé à chaque groupe de trouver des aspects originaux et spécifiques à présenter et de ne pas chercher à couvrir l'intégralité du projet.

Spécifications

Les spécifications présentées ici sont des suggestions, qui peuvent servir de base au projet. Il n'y a pas une unique "correction" du partiel.

En outre, ces spécifications peuvent contenir des erreurs ou des imprécisions[2].

type Cell {**IN**, **OUT**, **EMP**, **WLL**, **DNO**, **DNC**, **DWO**, **DWC** }
type Dir {**N**, **S**, **W**, **E** }
type Command {**FF**, **BB**, **RR**, **LL**, **TL**, **TR** }
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`: [Mat] × int × int → Cell
pre `CellNature(M,x,y)` **requires** $0 \leq$ x $<$ `Width(M)` **and** $0 \leq$ 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** }
Observation:
[invariant]: ⊤
[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`
...

---

[2]Dans ce cas, écrire rapidement aux encadrants.

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: ∅

Operators: `SetNature`: [EditMap] × int × int × Cell → [EditMap]
pre `SetNature(M,x,y)` **requires** $0 \leq$ x < `Width(M)` **and** $0 \leq$ 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, `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, `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

Observators : `CellContent`: int × int → Option[Mob]

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

Service:	Mob
Types:	bool, int, Cell
Observators :	`Env`: [Mob] $\rightarrow$ Environment
	`Col`: [Mob] $\rightarrow$ int
	`Row`: [Mob] $\rightarrow$ int
	`Face`: [Mob] $\rightarrow$ Dir
Constructors:	`init`: Environment $\times$ int $\times$ int $\times$ Dir $\rightarrow$ [Mob]
	pre `init(E,x,y,D)` **requires** $0 \leq$ `x` $<$ `Environment::Width(E)`
	and $0 \leq$ `y` $<$ `Environment::Height(E)`
Operators:	`Forward`: [Mob] $\rightarrow$ [Mob]
	`Backward`: [Mob] $\rightarrow$ [Mob]
	`TurnL`: [Mob] $\rightarrow$ [Mob]
	`TurnR`: [Mob] $\rightarrow$ [Mob]
	`StrafeL`: [Mob] $\rightarrow$ [Mob]
	`StrafeR`: [Mob] $\rightarrow$ [Mob]
[invariant] :	$0 \leq$ `Col(M)` $<$ `Environment::Width(Envi(M))`
	$0 \leq$ `Row(M)` $<$ `Environment::Height(Envi(M))`
	Environment::CellNature(Envi(M),Col(M),Row(M)) $\notin$ {**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`

On ne donne qu'une version partielle des postconditions (à compléter de manière systématique).

[Forward]: Face(M)=**N implies**
Environment::CellNature(Envi(M),Col(M),Row(M)+1) $\in$ {**EMP**, **DWO**}
and Row(M)+1 $<$ Environment::Width(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) $\notin$ {**EMP**, **DWO**}
or Row(M)+1 $\geq$ Environment::Width(Envi(M))
or Environment::CellContent(Envi(M),Col(M),Row(M)+1) $\neq$ **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)) $\in$ {**EMP**, **DNO**}
and Col(M)+1 $<$ Environment::Height(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)) $\notin$ {**EMP**, **DWO**}
or Row(M) $\geq$ Environment::Width(Envi(M))
or Environment::CellContent(Envi(M),Col(M)+1,Row(M)) $\neq$ **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) $\in$ {**EMP**, **DWO**}
and Col(M)-1 $\geq$ 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) $\notin$ {**EMP**, **DWO**}
or Col(M)-1 $<$ 0
or Environment::CellContent(Envi(M),Col(M),Row(M)-1) $\neq$ **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)) $\in$ {**EMP**, **DNO**}
and Row(M)-1 $\geq$ 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)) $\notin$ {**EMP**, **DNO**}
or Row(M)-1 $<$ 0
or Environment::CellContent(Envi(M),Col(M),Row(M)-1) $\neq$ **No**
implies Row(Forward(M)) = Row(M)
and Col(Forward(M)) = Col(M)

[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**

[...]: les autres opérateurs ont des post-conditions similaires.

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

Service: Cow **includes** Entity
Constructor: `init`: Environment × int × int × Dir × int → [Entity]
pre `init(E,x,y,D,h)` **requires** 4 ≥ h ≥ 3
Observations
[step]: `Col(M) - 1` ≤ `Col(step(M))` ≤ `Col(M) + 1`
`Row(M) - 1` ≤ `Row(step(M))` ≤ `Row(M) + 1`

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 → Cell
pre `Nature(P,x,y)` **requires** x ∈ {-1,0,1}**and** y ∈ {-1,+3}
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], **not** `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)`

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** $0 \leq$ `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 entités étaient indépendantes les unes des autres. Mais ce n'est pas le cas (les déplacements sont dépendants de l'ordre des entités, plus tard on ajoutera le combat).