

UNIVERSITE DE NAMUR

ANALYSE ET MODÉLISATION DES SYTÈMES D'INFORMATION IHDCB335

Laboratoire : *World Game*

Laboratoire réalisé de manière individuelle

Etudiant

Hadrien BAILLY

Groupe

HD-38

Professeur

Moussa AMRANI

Assistant

Tony LECLERCQ

12 Décembre, 2020



Table des matières

Introduction	3
Préambule	3
Calendrier	3
Questions	3
I Laboratoire	5
1 Description du jeu	6
1.1 Pré-version	6
1.2 Description détaillée	7
1.2.1 Jeu	8
1.2.2 Joueur	9
1.2.3 Plateau	10
1.2.4 Obstacle	12
1.2.5 Objet	13
1.3 Contraintes d'unicité	13
1.4 Contraintes de règles	14
1.5 Question de stratégies	18
2 Diagrammes d'objet	20
2.1 Jeu	21
2.2 Plateau	22
2.3 Avatar 1	23
2.4 Avatar 2	24
2.5 zombie	25
3 Description des stratégies	26
3.1 Pré-version	26
3.2 Version détaillée	27
3.2.1 Définition	28
3.2.2 Langage	32
3.3 Contraintes d'unicité	36
3.4 Contraintes de règles	37
3.5 Programmation par contrat	38
3.5.1 Expression : type()	38
3.5.2 Avatar : canCross(tile : Tile) : Boolean	40
3.5.3 Avatar : collect() : Void	41
3.5.4 Instruction : isValid()	43
3.6 Questions de priorités	44
4 Diagrammes d'objet	46
4.1 Tile : containsGrail()	46
5 Maturité et modularité	48

6 Conclusion	49
II Annexes	50
Spécifications	51
1 Description du jeu	51
1.1 Game	51
1.2 Series	51
1.3 Match	52
1.4 Player	52
1.5 Avatar	53
1.6 Skin	53
1.7 Inventory	54
1.8 Board	54
1.9 Tile	54
1.10 Zombie	55
1.11 Obstacle	55
1.12 Item	56
2 Description des stratégies	58
2.1 Strategy	58
2.2 Objective	58
2.3 Rule	59
2.4 Action	59
2.5 Module	60
2.6 Types	60
2.7 Expression	61
2.8 Instruction	61
Glossaire	62
Bibliographie	69

Introduction

Préambule

Ce laboratoire a été réalisé dans le cadre du cours d'*analyse et modélisation des systèmes d'information* (IHDCB335). Il s'agit d'un exercice pratique d'analyse et de compréhension de spécifications données par un client. L'objectif est la réalisation d'un ensemble de diagrammes de classes et d'objets modélisant les différents aspects d'un jeu de plateau.

Ce travail se découpe en deux parties : nous aborderons en premier la **description du jeu** d'un point de vue pratique, en décrivant la manière dont une partie se déroule dans le jeu et les éléments qu'elle articule pour ce faire. Ensuite, nous détaillerons la **description des stratégies**, c'est-à-dire le mécanisme qui permet à un personnage du jeu d'être contrôlé par l'ordinateur et de réaliser des actions de manière automatique selon une stratégie définie.

Ce document sera enfin suivi d'annexes : un inventaire des spécifications issues du document reçu du client ainsi qu'un glossaire décrivant les termes employés dans ce travail.

Initialement prévu pour être réalisé en binôme, ce travail a été réalisé de manière entièrement individuelle suite au désistement du deuxième étudiant.

Calendrier

Étape	Date	Temps (h)
Initialisation	24/10/2020	4
Description de jeu - premiers CD	29/10/2020 01/11/2020 07-8/11/2020 11/11/2020	8 8 12 4
Description du jeu - OD	14-16/11/2020	12
Description des stratégies - premiers CD	23-25/11/2020 30/11/2020 01-02/12/2020	12 4 16
Description des stratégies - OD	03-06/12/2020	32
Rédaction et corrections	07-09/12/2020	24
Relecture	10/12/2020	4

Ce projet aura nécessité environ 140h de travail dans sa version actuelle.

Questions

Ce rapport répond à la liste de questions présentée par le document de spécification dans les pages mentionnées ci-après. Le numéro de question est également rappelé au début des sections correspondantes.

Description du jeu	
Question 01 - pré-version	6
Question 02 - version complète	7
Question 03 - contraintes d'unicité	13
Question 04 - contraintes de règles	14
Question 05 - rapprochement avec le diagramme de stratégie	18
Question 06 - diagramme d'objet	20
Description des stratégies	
Question 07 - pré-version	26
Question 08 - type	33
Question 09 - déclaration	32
Question 10 - objectif	29
Question 11 - action	31
Question 12 - expression	34
Question 13 - instruction	35
Question 14 - contraintes d'unicité	36
Question 15 - contraintes de règles	37
Question 16 - Expression : :type()	38
Question 17 - estTraversableGraceAuxItems(...)	40
Question 18 - ramasser(...)	41
Question 19 - Instruction : : estValide()	43
Question 20 - contraintes d'action	44
Question 21 - diagramme d'objets	46
Maturité et modularité	
Question 22 - objets/obstacles	48

Toutes les questions ont été répondues dans l'ordre, hormis la question 10 qui a été déplacée dans la sous-section suivante.

Première partie

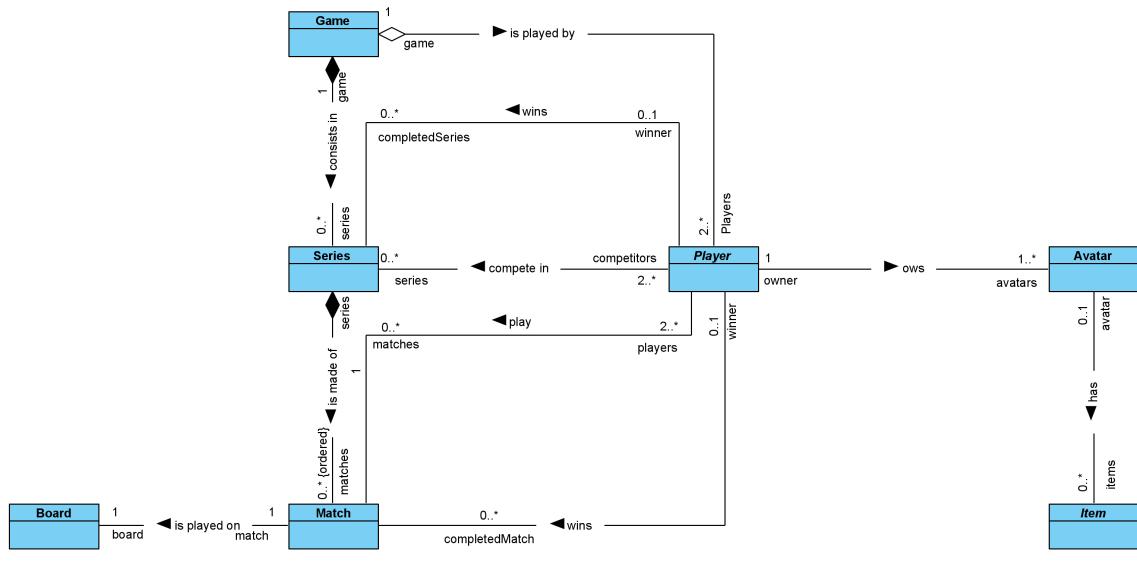
Laboratoire

Chapitre 1

Description du jeu

1.1 Pré-version

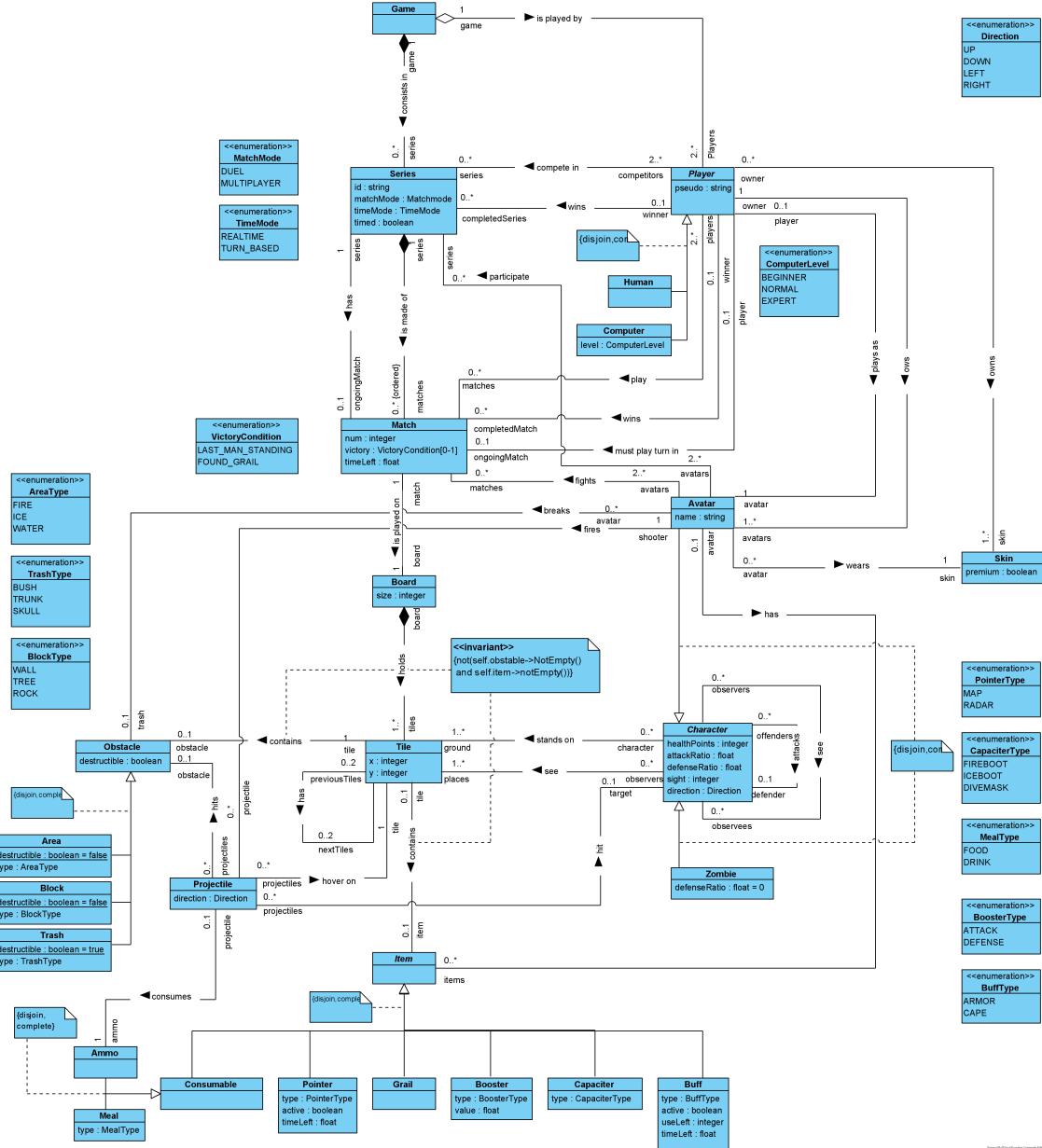
Question 4.1.



Le jeu décrit par les spécifications est un jeu d'exploration et de combat : des joueurs participent à des séries de rencontres. Sur le terrain d'une rencontre, les joueurs contrôlent des avatars : ils explorent un plateau de jeu, ramassent des objets et affrontent d'autres avatars ainsi que des monstres non-contrôlables. Pour gagner, les joueurs doivent soit tuer tous les autres joueurs (*Battle Royale*) soit trouver la sortie en premier (*Graal*).

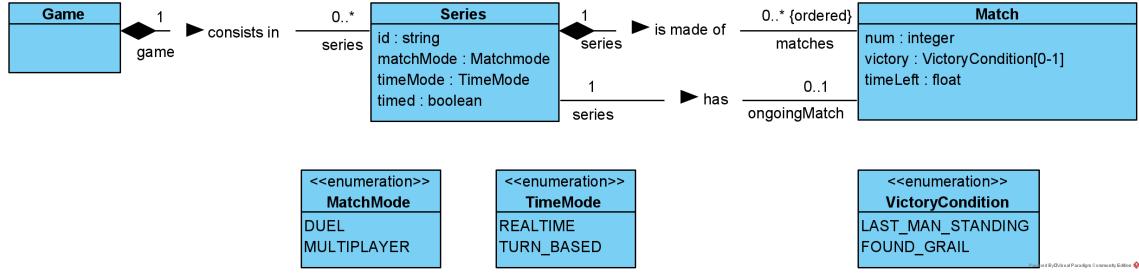
1.2 Description détaillée

Question 4.2.



Dans cette section, nous détaillerons les diagrammes spécifiques aux différentes composantes de la définition du jeu, tel que donnée par les spécifications-client.

1.2.1 Jeu



Le jeu décrit dans ce laboratoire est découpé en séries de rencontres :

- Le jeu est constitué d'un ensemble de séries.
- une série est composée d'au moins une rencontre.

Nous avons ainsi deux relations de *composition* :

- Le jeu est composé de séries, c'est-à-dire que toute série implique nécessairement l'existence d'un jeu dont elle fait partie.
- La série est composée de rencontres et toute rencontre forme, de par son existence, au moins une série de 1 rencontre.

En termes de multiplicité, nous avons

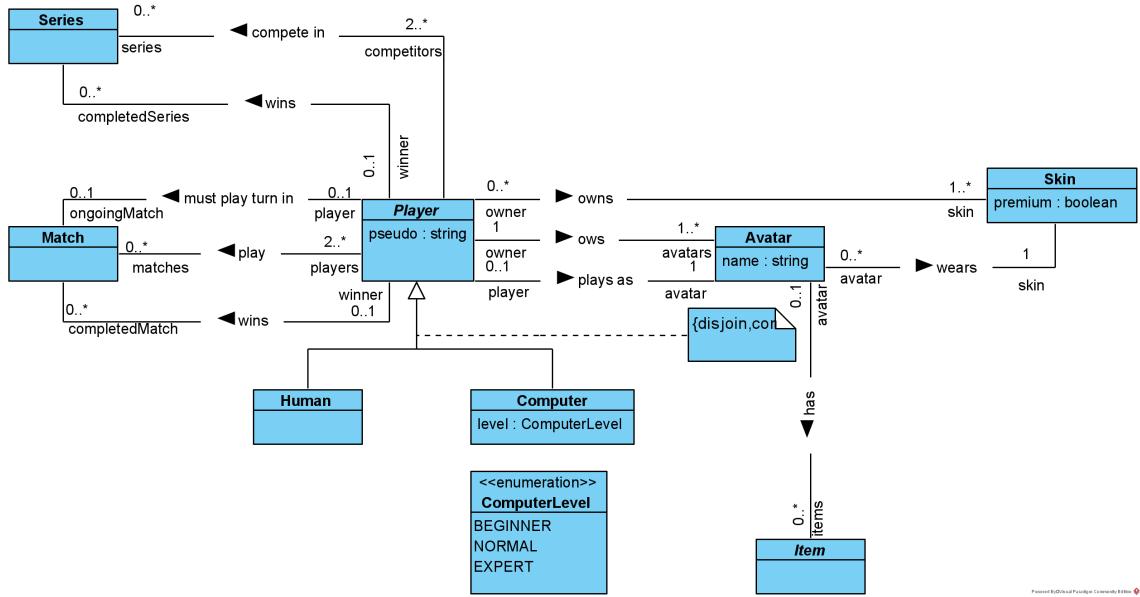
- pour un jeu, un ensemble non-ordonné de suites jouées.
- pour une série, un ensemble ordonné de rencontres, qui sont jouées de manière séquentielle.
- pour une série, il y a au maximum une rencontre en cours (soit il y a une rencontre en cours et la série continue, soit toutes les rencontres ont été jouées et la série est terminée).

Nous avons par ailleurs choisi de représenter les différents choix de mode et formule de jeux que peuvent poser les joueurs par des énumérations. En effet, il s'agit à chaque fois d'une information finie et restreinte, qu'il est facile d'énumérer.

Nous aurions pu utiliser des marqueurs booléens mais ceux-ci nous ont semblé moins porteurs d'information et d'expressivité dans le schéma. De plus, ils se seraient avérés moins flexibles si d'autres modes/formules devaient faire leur apparition par la suite.

Enfin, un match comporte une condition de victoire : il ne s'agit ici pas d'une énonciation à l'instanciation de l'objet mais bien d'un marqueur signifiant (le cas échéant) qu'une rencontre a été remportée et de quelle manière.

1.2.2 Joueur



Le jeu est fait pour être joué : il a donc besoin de joueurs.

Nous distinguons ici deux sous-catégories de joueurs :

- les joueurs humains et
- les joueurs-machine.

Nous avons ici spécifiquement une association de type *spécialisation* : alors que les joueurs normaux (humains) ne possèdent qu'un nom, les joueurs-machine possèdent en plus un niveau de difficulté.

En tant que telle, la classe *Human* est redondante dans la relation de spécialisation. Nous aurions en effet pu nous contenter d'avoir une couverture incomplète pour distinguer les joueurs-machines des autres (qui sont les humains par déduction). Nous avons cependant choisi de garder cette classe sur notre diagramme pour des raisons d'expressivité.

Un joueur est activement impliqué dans le jeu : il concourt dans des séries, qu'il peut gagner ou perdre, et il joue effectivement dans des rencontres, qu'il peut à nouveau gagner ou perdre. Lorsque le mode tour-par-tour est activé, le joueur peut en outre être appelé à jouer ou à attendre que son tour arrive.

Chaque joueur est désigné par un pseudo qui lui est propre.

Nous posons ici que même les joueurs-machine ont un pseudo (par ex. *BOT_1*).

Les joueurs possèdent une galerie de personnages : des avatars utilisés dans le jeu et des apparences pour les personnaliser.

En termes de multiplicité, nous avons posé les choix suivants :

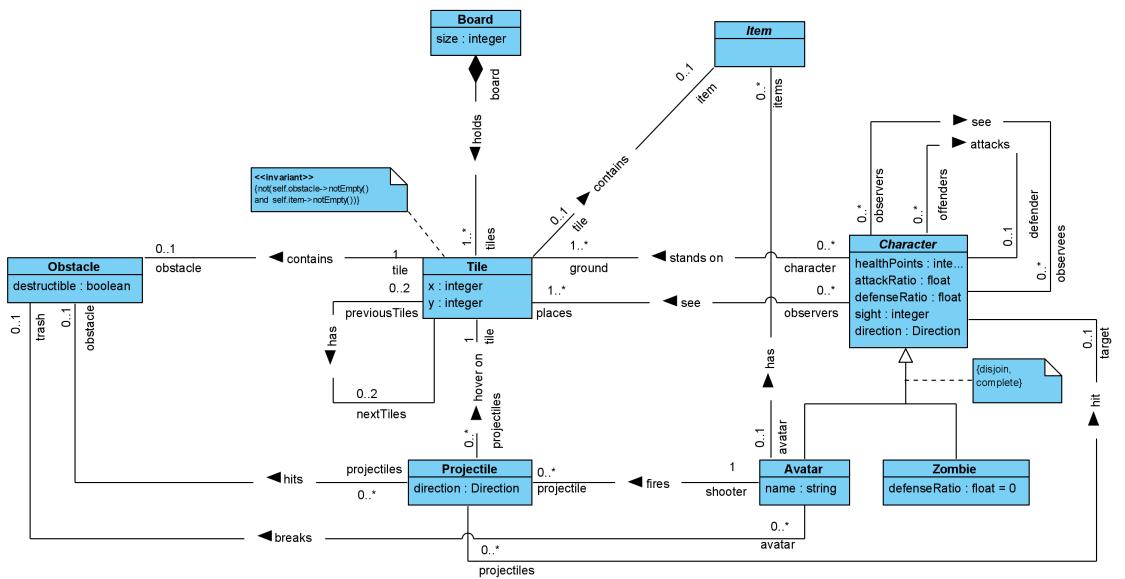
- Un joueur peut avoir joué un nombre indéfini de séries et de rencontre, allant de 0 pour les nouveaux arrivants à un nombre illimité pour les plus accros.
- Toute série et rencontre impliquent nécessairement au moins deux joueurs (aucun mode *joueur vs environnement* n'est prévu). Il peut y en avoir davantage dans un mode multijoueur.
- Un joueur ne peut participer au maximum qu'à une rencontre et série à la fois.
- Il ne peut jamais y avoir plus qu'un gagnant par rencontre et par série.

La spécification du jeu précise que le gagnant d'une série est le joueur qui a remporté le plus de rencontres (raison pour laquelle celles-ci sont en nombre impair). Cette spécification ne tient pas compte du fait que des rencontres peuvent se terminer sans vainqueur parce que le chronomètre est écoulé. Dans ce cas, puisque la spécification ne précise pas le comportement à adopter, le choix sera laissé au développeur de fixer une nouvelle règle :

- personnage ayant trouvé le dernier Graal.
 - personnage ayant tué le plus d'ennemis.
 - ...

- Un joueur dispose d'au moins un avatar, qui présente toujours une apparence (que le joueur a le loisir de changer).
 - Un joueur possède au moins une apparence (par défaut) à appliquer à l'un de ses avatars.

1.2.3 Plateau



Le plateau est l'endroit-clé du jeu : c'est sur celui-ci que les joueurs passent la partie la plus importante du jeu, à explorer, ramasser et attaquer.

Un plateau est constitué de tuiles : des cases numérotées et organisées de manière cardinale. Nous avons ici à nouveau une relation de *composition* : un plateau est nécessairement composé de tuiles, et une tuile forme nécessairement au moins un plateau de 1 tuile.

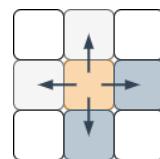
Sur ces tuiles déambulent des personnages : des entités animées, contrôlées par un joueur ou par l'ordinateur et pouvant se mouvoir sur le plateau. Ces personnages sont dotés de caractéristiques et de comportement uniformes : position, attaque et défense, champ de vision, mouvement, ... En raison de cette uniformisation, nous avons choisi de déployer une super classe de *généralisation* disjointe et complète : tout personnage est nécessairement soit un avatar, soit un zombie.

Les avatars des joueurs sont dotés de capacités supplémentaires, par rapport aux zombies : ils peuvent ramasser des objets, tirer des projectiles ou encore tenter de casser des obstacles destructibles. Les objets qu'ils possèdent sont placés dans un inventaire.

Nous avons ici choisi de ne pas représenter conceptuellement l'inventaire sous forme de classe distincte, puisqu'il n'aurait pour seules associations que le fait de contenir des objets et appartenir à un avatar. Fonctionnellement, l'inventaire ne désigne que le regroupement des objets amassés par le joueur, ce qui est déjà signifié par l'association " $[0..1] \text{ player has } [0..*] \text{ items}$ " (un joueur peut amasser un nombre indéfini d'objets).

En termes de multiplicité, nous avons posé les choix suivants :

- Une tuile est située près d'autres tuiles de manière adjacente :
 - Deux tuiles situées à une position inférieure d'un facteur x ou y .
 - Deux tuiles situées à une position supérieure d'un facteur x ou y .

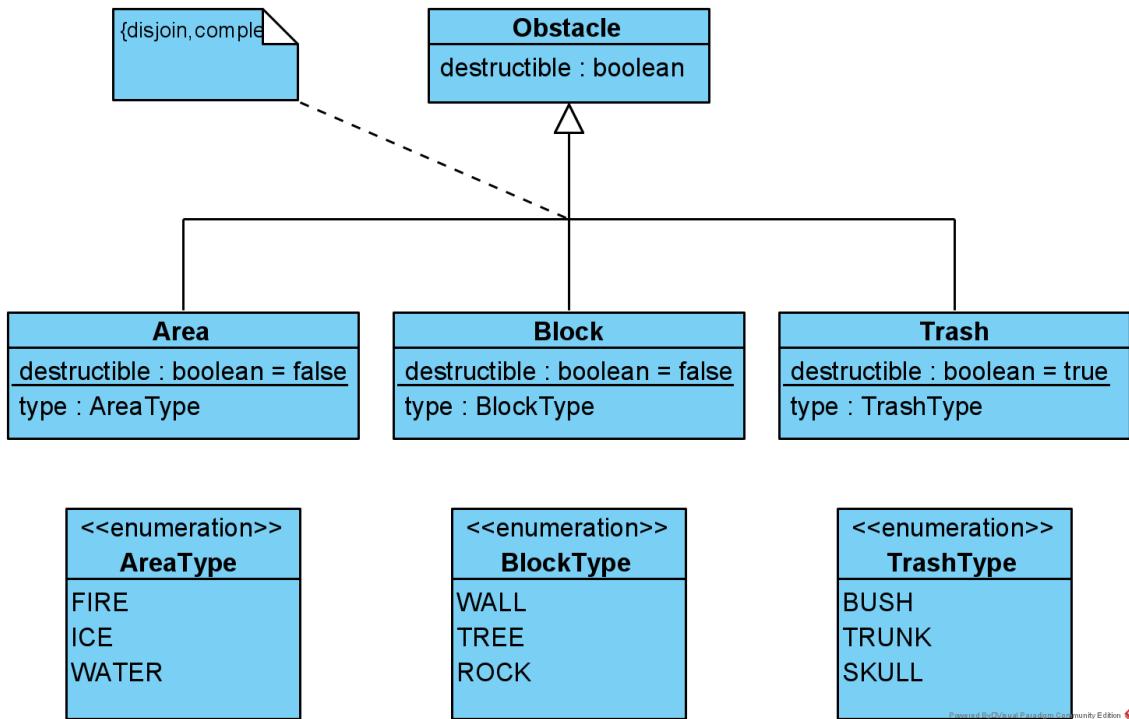


- Une tuile contient au maximum un obstacle ou un objet. La contrainte est ici exprimée en OCL car l'exclusion mutuelle n'est pas descriptible simplement sur le diagramme de classe.

Nous aurions pu gérer ce cas en rassemblant obstacles et objets sous une classe abstraite généralisant les objets *statiques* (par opposition aux personnages dynamiques). Il aurait alors suffi de mettre une contrainte de cardinalité $[0..1]$ pour ne permettre la présence que d'un objet ou d'un obstacle sur une tuile. Nous avons choisi de ne pas le faire car nous estimions qu'ils restaient conceptuellement fort éloignés.

- Une tuile n'est suffisante que pour accueillir au maximum un personnage, qu'il s'agisse d'un zombie ou d'un avatar.
- Un avatar peut ramasser un objet ou tirer un projectile. De manière inverse, tout objet doit être ramassé/tout projectile doit nécessairement être lancé par un avatar.
- Un personnage peut voir autant de personnages que le permet son champ de vision. De même, ce personnage peut être vu par autant de personnages que leur champ de vision le leur permet. Il ne s'agit ici pas d'une relation réflexive : un personnage qui en voit un autre n'est pas forcément aperçu par cet autre personnage, car ils ne partagent pas le même champ de vision.
- Un personnage ne peut jamais attaquer plus d'un seul personnage (situé en face de lui). Il peut en revanche être attaqué par plusieurs personnages/touché par plusieurs projectiles à la fois (s'il est encerclé).

1.2.4 Obstacle



Les obstacles sont des éléments fixes du décor, qui peuvent empêcher ou gêner le personnage.

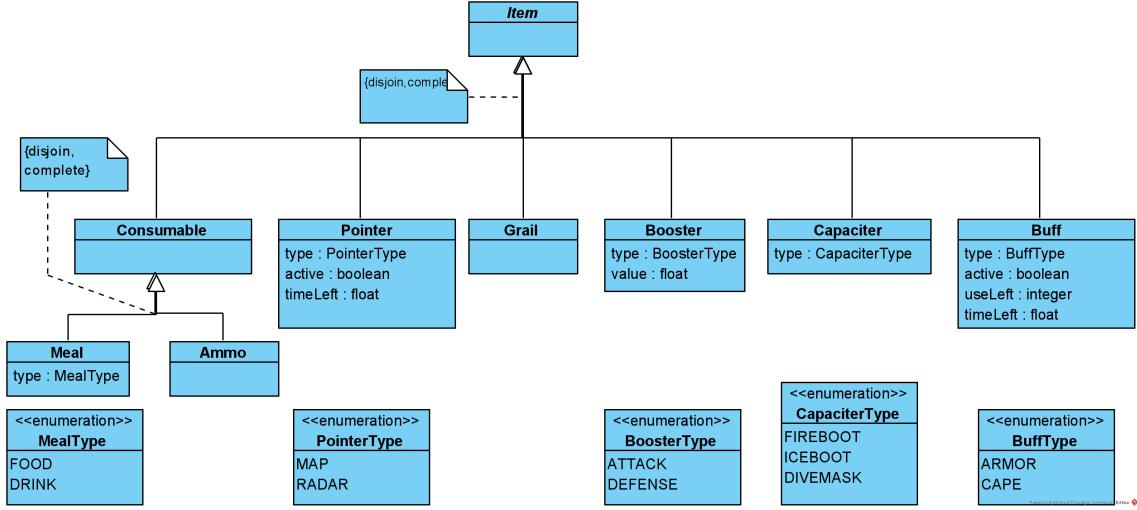
De par les spécifications, nous savons que tout obstacle est traversable par le truchement d'objets. Dès lors, notre catégorisation repose davantage sur des caractéristiques plus pérennes entre les objets :

- Certains obstacles sont déstructibles (les crasses).
- Certains obstacles sont permanents et inactifs (les blocs).
- Certains obstacles sont permanents et offrent un certain niveau d'interaction (les zones).

Ces obstacles sont mutuellement exclusifs, nous avons donc choisi une relation de *spécialisation* disjointe et complète.

Nous avons également choisi de marquer la capacité de *destructibilité* par un booléen défini au niveau des classes et non des instances, puisqu'il s'agit d'un trait spécifique à chaque classe mais immuable.

1.2.5 Objet



Les objets sont des éléments fixes du décor dont le joueur va pouvoir s'emparer pour s'aider dans sa quête.

Comme pour les obstacles, nous avons choisi de dissocier les objets en fonction de leur comportement.

- Les consommables (*Consumable*) sont des objets *jetables* dont la fonction est d'être accumulé puis utilisé par l'avatar.
- Les pointeurs (*Pointer*) sont des objets aidant à l'orientation et non accumulables (seule une unique version est conservée, dont on manipule ensuite le compteur).
- Le Graal (*Grail*) est le graal, simplement.
- Les améliorants (*Booster*) sont des objets instantanés, qui augmentent immédiatement le potentiel de l'avatar.
- Les capaciteurs (*Capaciter*) sont des objets permanents conférant au personnage des nouvelles capacités.
- Les super-pouvoirs (*Buff*) sont des objets puissants, trouvables en faible quantité et conférant à leur porteur un avantage critique. Ils sont caractérisés par une durée limitée d'utilisation.

Il s'agira ici plus d'une association de *spécialisation*, mettant en avant les caractéristiques susmentionnées. Nous avons en outre inclus une sous-spécialisation des consommables, pour signifier l'effet : gain de vie versus munition pour lancer de projectiles.

Pour des raisons d'expressivité et parce que la liste des valeurs possibles est définie et limitée, nous avons choisi d'utiliser des énumérations pour représenter les types finaux possibles des objets.

Il existe par ailleurs deux catégories d'objets qui disposent d'un temps d'utilisation limité : les pointeurs et les super-pouvoirs. Pour représenter l'écoulement du temps, nous avons choisi d'utiliser un seul compteur (*timeLeft*) dont nous faisons varier l'unité en fonction du mode de jeu choisi :

- S'il s'agit d'une rencontre en mode tour-par-tour, le compteur sera décrémenté à chaque fin de tour (via des valeurs entières).
- S'il s'agit d'une rencontre en mode temps-réel, le compteur sera décrémenté au fur et à mesure du temps qui passe (via des valeurs décimales).

1.3 Contraintes d'unicité

Question 4.3.

- les matchs sont identifiés de manière unique au sein du jeu ;

```
context Game inv uniqueSeries :
  self.series->forAll(s1,s2|s1<>s2 implies s1.id<>s2.id)
```

Pour tout jeu donné, toute série différente possède nécessairement un identifiant différent.

- les joueurs ont des pseudos différents au sein du jeu ;

```
context Game inv uniquePlayer :
  self.players->forAll(p1,p2|p1<>p2 implies p1.pseudo<>p2.pseudo)
```

Pour tout joueur donné, s'il existe un autre joueur distinct alors celui-ci aura un pseudo différent.

- les personnages/avatars d'un joueur sont nommés différemment, et ont une représentation physique distincte pour pouvoir les reconnaître visuellement au premier coup d'oeil ;

```
context Player inv distinctAvatars :
  self.avatars->forAll(a1,a2|(a1.id=a2.id or a1.skin=a2.skin) implies a1=a2)
```

Pour tout joueur donné, un même nom ou une même apparence ne peut appartenir qu'à un seul avatar.

- les rencontres d'un match portent un numéro d'ordre unique ;

```
context Series inv distinctMatchNum :
  self.matches->forAll(m1,m2|m1<>m2 implies m1.num <>m2.num)
```

Pour toute série donnée, l'ensemble des rencontres qui la composent possèdent un numéro différent entre elles.

1.4 Contraintes de règles

Question 4.4.

- le potentiel de vie d'un joueur est toujours positif ;

```
context Character inv positiveHealth :
  self.healthPoints > 0
```

Pour un personnage donné, sa santé est toujours positive (sinon, il est mort et enterré †).

- les ratios d'attaque et de défense sont des ratios, c'est-à-dire des valeurs strictement positives et inférieures à 1 ;

```
context Character inv positiveAR :
  self.attackRatio > 0 and self.attackRatio < 1
```

```
context Avatar inv positiveDR :
  self.defenseRatio > 0 and self.defenseRatio < 1
```

```
context Zombie inv zeroDR :
  self.defenseRatio = 0
```

Pour tout personnage donné, le ratio d'attaque est systématiquement compris entre 0 et 1 exclus. Par contre, seuls les avatars ont un ratio de défense non-nul (puisque'il est spécifié que les zombies n'ont pas de défense).

- le nombre de rencontres constituant un match est toujours impair;

```
context Series inv oddNumberOfmatches :
    self.matches->size().mod(2)=1
```

Pour toute série, le reste de la division par deux doit être 1 (ce qui signifie qu'il est impair).

- le numéro identifiant la rencontre au sein d'un match correspond à l'ordre dans lequel il sera joué (ou plus précisément, l'ordre dans la collection qui le stocke, à supposer que les rencontres sont jouées dans l'ordre de stockage);

```
context Series inv matchNumEqualsOrder :
    self.matches->forAll(m|self.matches->at(m.num)=m)
```

Pour toute série, on trouve à chaque indice un match dont le numéro est égal à cet indice.

- l'inventaire d'un joueur contient trois types d'items : les items ramassables (nourriture, boisson ou munitions), les objets d'aide à l'orientation, et les objets d'aide (cotte de maille et cape);

```
context Avatar inv inventory :
    self.items->forAll(i|i.oclIsKindOf(Item) and
                           not(i.oclIsTypeOf(Graal)) and
                           not(i.oclIsTypeOf(Booster)))
```

Pour un avatar donné, tous les éléments de l'inventaire sont des objets de super type *Item*, mais aucun n'est le Graal ni un booster (qui sont consommés immédiatement).

- une seule des cases d'un monde contient un Graal ;

```
context Board inv onlyOneGraal :
    self.tiles.item->one.oclIsTypeOf(Graal))
```

Pour un plateau donné, de tous les objets présents sur toutes les tuiles, seul un est le Graal.

- les coordonnées d'une case ne peuvent excéder la longueur d'un monde ;

```
context Board inv tileNotExceedLimit :
    self.tiles->select(t|t.x<1 or
                           t.x>self.size or
                           t.y<1 or
                           t.y>self.size)->isEmpty()
```

Pour un plateau donné, il n'existe aucune tuile dont l'un des paramètres est hors-limite (en dehors de l'intervalle $[1 : size]$).

- la disposition des cases correspond à leur coordonnées. Par exemple, l'abscisse d'une case à droite d'une case donnée est l'entier successeur de l'abscisse de la case de référence (et similairement pour les autres directions, en tenant compte des bords du plateau);

```
context Board inv isFull :
    self.tiles.count()=(self.size * self.size)
```

```
context Board inv noDuplicateTile :
    self.tiles->forAll(t1,t2|(t1.x=t2.x and t1.y=t2.y)
                           implies t1=t2)
```

```

context Board inv isNextTo :
    self.tiles->forAll(t1,t2|t1.nextTiles->includes(t2)
        implies (t2.x=t1.x+1 and t2.y=t1.y) or
            (t2.x=t1.x and t2.y=t1.y+1))

```

```

context Board inv onlyOneFirstTile :
    self->previousTiles->isEmpty() implies self.x=1 and self.y=1

```

Pour un plateau donné, pour chacune des cases, une case qui est voisine suivante d'une autre a nécessairement soit un x plus grand de 1, soit un y plus grand de 1.

- Une tuile peut avoir deux voisines : 1 au sud, 1 à l'est, par exemple pour la case 1×1 .
- Une tuile peut avoir seulement une voisine (si elle se trouve en extrémité de plateau par un côté)
- Une tuile peut avoir zéro voisine (si elle se trouve en tout fin du plateau, à $size \times size$).
- La seule tuile qui n'a aucune tuile précédente est la tuile 1×1 .

Si on combine les règles *isNexto*, *tileNotExceedLimit*, *isFull* et *onlyOneFirstTile*, on obtient par récurrence que les tuiles sont placées au bon emplacement et que le plateau n'a pas de case manquante : il existe le bon nombre de tuiles, qui sont toutes différentes et comprises dans les limites du plateau, qui sont placées par ordre croissant et dont la première est nécessairement la tuile 1×1 .

- *un joueur se trouve toujours dans la case correspondant à sa position absolue* ;
Pas de contrainte OCL à exprimer ici.

Un personnage est lié au maximum à une et une seule case :

- soit il est utilisé par le joueur dans une partie définie et alors, sa position absolue est fonction de la case (*ground*) sur laquelle il se trouve.
- soit il n'est pas utilisé par le joueur, n'est donc pas présent sur le plateau et n'a dès lors pas de position absolue.

Dès lors que le personnage n'est lié qu'à une seule case, il est possible d'en déduire sa position absolue depuis cette case.

- *la bordure d'un plateau contient toujours des éléments infranchissables (pour éviter aux personnages de « sortir » du monde)* ;

```

context Board inv isWalled :
    self.tiles.forAll(t|(t.x=1 or
        t.x=self.size or
        t.y=1 or
        t.y=self.size)
    implies t.obstacle.oclIsKindOf(Block))

```

Pour toute tuile du jeu, si celle-ci est comprise à une extrémité (càd qu'un des paramètres de sa position est 1 ou la taille), alors elle doit contenir un obstacle de sous-type de **Block**.

Cette règle n'empêchera pas un joueur qui possède une cape de mage de sortir de la cape, puisque celle-ci lui permet de traverser tout obstacle, même ceux de type *Block*. Pour empêcher un joueur de sortir effectivement, il faudra utiliser les coordonnées de la case de destination pour vérifier que le mouvement est valide.

- Le nombre de cases visibles correspond à la valeur de visibilité liée au joueur. Par exemple, dans le plateau à droite dans la Figure 2, la visibilité de 3 rend « découvertes » trois lignes au-dessus, en dessous, et trois colonnes à droite et à gauche ;

```
context Avatar inv canSeeTile :
    self.visibleTiles->forAll(t| t.x <= self.ground.x+self.sight and
                                t.x >= self.ground.x-self.sight and
                                t.y <= self.ground.y+self.sight and
                                t.y >= self.ground.y-self.sight)
```

Pour un personnage donné, les tuiles visibles sont celles dont la position est incluse dans deux intervalles au centre desquels se trouve le personnage :

- [x-sight : x+sight]
- [y-sight : y+sight]

- un personnage ne peut pas se trouver sur une case portant un obstacle infranchissable ;

```
context Tile inv nonCrossableObstacle :
    self.obstacle->notEmpty() and self.character->notEmpty()
implies
    self.character.oclIsTypeOf(Avatar) and
    (self.obstacle.oclIsTypeOf(Area) and
        (self.obstacle.type=AreaType::FIRE
        or (self.obstacle.type=AreaType::ICE and
            self.character->oclAsType(Avatar).items
            ->select(i|i.oclIsTypeOf(Capaciter) and
                    i.type=CapaciterType::ICEBOOT)->notEmpty())
        or (self.obstacle.type=AreaType::WATER and
            self.character->oclAsType(Avatar).items
            ->select(i|i.oclIsTypeOf(Capaciter) and
                    i.type=CapaciterType::DIVEMASK)->notEmpty()))
    or (self.obstacle.oclIsTypeOf(Area) and
        self.character->oclAsType(Avatar).items
        ->select(i|i.oclIsTypeOf(Buff) and
                i.type=BuffType::CAPE)->notEmpty())
    or (self.character.items
        ->select(i|i.oclIsTypeOf(Buff) and
                i.type=BuffType::CAPE and
                i.useLeft>0)->notEmpty()))
```

Si, sur une tuile donnée, on trouve un avatar et un obstacle, alors cela implique que

- soit l'obstacle est une zone de feu (et le joueur peut le franchir au prix de points de vie),
- soit l'obstacle est une zone de glace et le personnage possède un capaciteur "botte à crampons" dans son inventaire,
- soit l'obstacle est une zone d'eau et le personnage possède un capaciteur "kit de plongée" dans son inventaire,
- soit l'obstacle est un obstacle de zone et le personnage possède la cape de mage,
- soit l'obstacle est n'importe quel autre obstacle et le personnage possède la cape de mage avec des points d'utilisation restants.

Nous acceptons ici plusieurs postulats :

- Les obstacles destructibles sont infranchissables tant qu'ils ne sont pas détruits (sauf avec la cape), et disparaissent quand ils sont détruits.
- La cape reste suffisante pour pouvoir traverser les zones *feu-eau-glace*, même si elle n'a plus de point d'utilisation restant : on pourrait ainsi imaginer une situation où le joueur conserve la cape après expiration de son utilisation pour les obstacles infranchissables.
- La cape qui n'a plus de temps (tour) actif restant est retirée de l'inventaire et ne doit donc pas être testée pour son(ses) temps (tours) restant(s).

- deux personnages (y compris les zombies) ne peuvent pas se trouver sur la même case ;
Il n'est pas nécessaire de vérifier par OCL si deux personnages sont sur une même tuile car cela est déjà réglé par la cardinalité de la relation "[0..1] character stands on [0..1] ground". Nous avons ici deux aspects :
 - Un personnage ne se trouve pas nécessairement sur une tuile : cette cardinalité est rendue nécessaire par le fait qu'un joueur peut posséder un avatar sans jouer à une partie. Dès lors qu'il n'est pas en partie, il ne peut y avoir de tuile sur laquelle se trouver. Il faut donc admettre le cas où il n'existe pas de tuile pour un avatar donné.
- Les zombies doivent en revanche nécessairement avoir une tuile, puisqu'ils n'existent que pour un monde à la fois. Cette contrainte doit être exprimée en OCL.

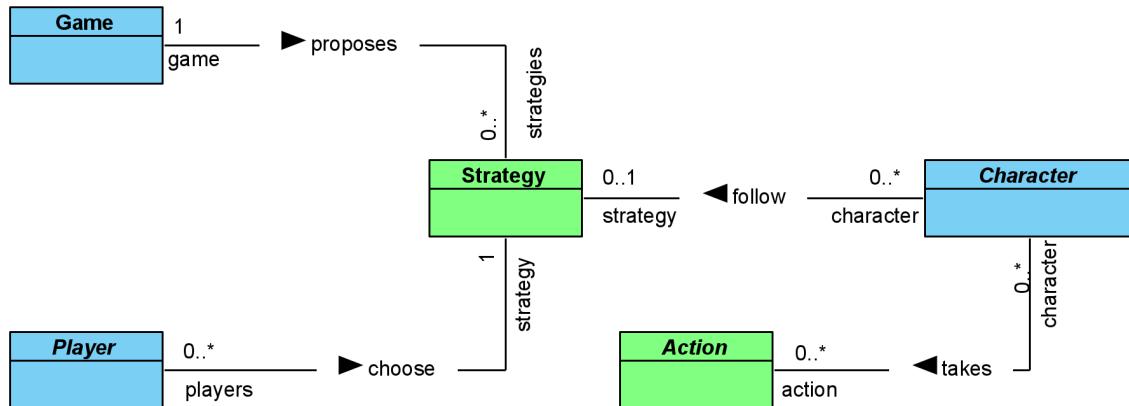
```
context Zombie inv alwaysOnTile:
  self.ground->notEmpty()
```

- Une tuile ne comprend pas nécessairement un personnage : toutes les tuiles ne sont en effet pas occupées (sinon, le plateau serait rempli de joueurs... et le jeu serait injouable).

1.5 Question de stratégies

Question 4.5.

La définition des deux DSLs, pour spécifier les mondes et pour définir les stratégies, est séparée en deux morceaux. Cependant, pour que le jeu soit utilisable, ces deux éléments doivent être harmonieusement connectés. Comment, et où, apparaît la notion de stratégie dans votre Diagramme de Classes pour spécifier les mondes ?



La notion de stratégie doit apparaître à quatre endroits :

- Dans la définition même du jeu, au sein d'une liste ayant été élaborée par les développeurs pour permettre de jouer en mode tour-par-tour.
- Dans le choix que peuvent faire les joueurs de la stratégie à suivre au cours d'une rencontre tour-par-tour.
- Dans le comportement des personnages, qui suivent une stratégie dans une rencontre tour-par-tour : les avatars qui suivent la stratégie choisie par leur propriétaire et les zombies qui appliquent une stratégie par défaut, que l'on suppose être long-terme=RANDOM, court-terme=ATTACK (les zombies errrent sans but jusqu'à apercevoir un avatar qu'ils vont tenter d'occire prestement).
- Dans les actions concrètes réalisées par les personnages, en fonction de la stratégie suivie.

En termes de cardinalité, il est possible que le jeu, dans une version alpha, ne comprenne que le mode temps-réel et puisse donc se jouer sans stratégie. Pour celui-là, la présence de stratégie n'est pas une condition nécessaire à son existence.

Pour les stratégies existantes, il peut y avoir certains stratégies rencontrant plus de succès que d'autres : une stratégie peut donc avoir été choisie par un ou plusieurs joueurs, mais aussi n'avoir été sélectionnée par aucun. Elles appartiennent en revanche toutes nécessairement à une et une seule instance du jeu.

Enfin, un personnage peut (ou peut ne pas) avoir de stratégie. Ceci dépend de son sous-type et du mode de jeu. Ainsi, dans une partie en temps réel, les avatars suivent les commandes directes des joueurs plutôt que d'une stratégie. En revanche, ils en suivent une et une seule (qui peut changer) dans les parties en tour-par-tour. Les zombies étant contrôlés par ordinateur, on suppose qu'ils suivront toujours une seule et même stratégie, qui sera adaptée aux conditions du jeu.

En suivant les stratégies, les personnages entreprennent des actions concrètes : ils vont se déplacer, ramasser des objets, attaquer des ennemis.

Chapitre 2

Diagrammes d'objet

Question 4.6.

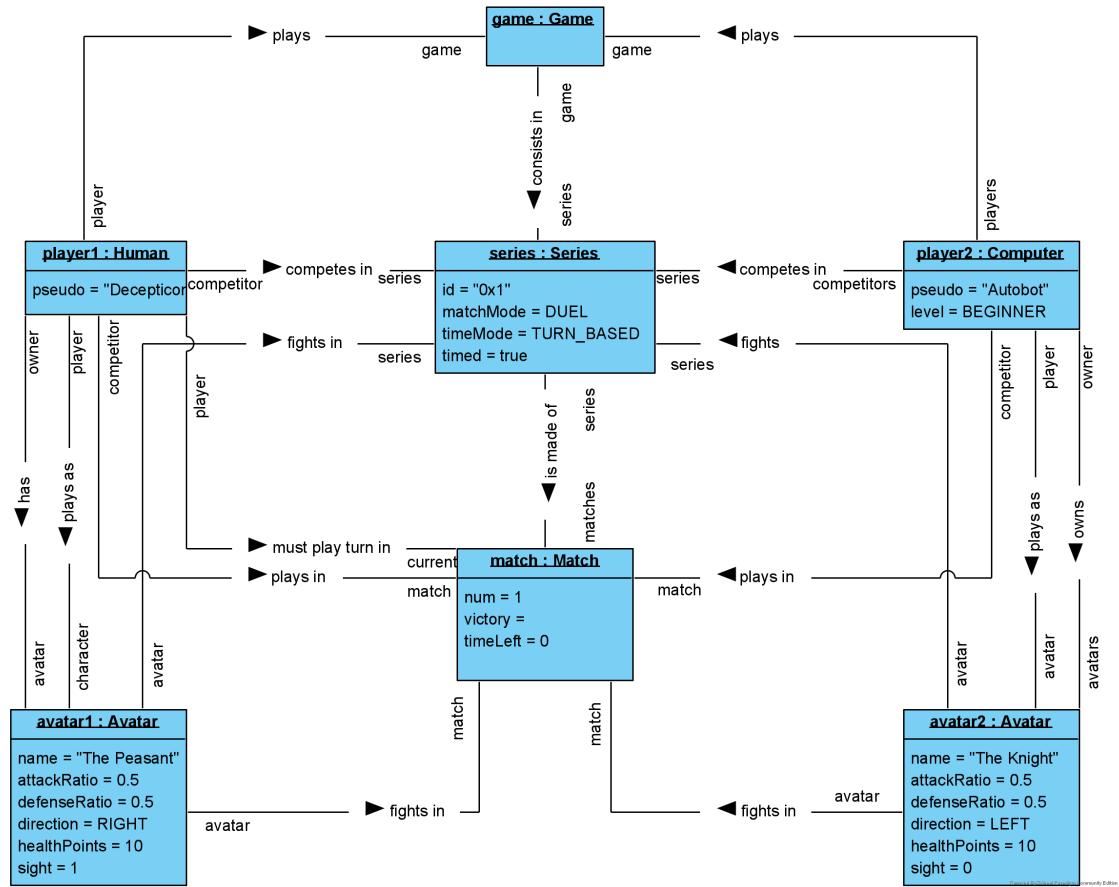
Définir un diagramme d'objets pour un monde de taille 4 (càd. 2×2) dont les bordures sont délimitées par des murs, et où les coins en bas à gauche et à droite sont occupés par les deux joueurs ; et les coins en haut à gauche et à droite sont occupés par le Graal et un zombie. On donnera au joueur et au zombie les mêmes potentiel et caractéristiques, et on supposera que le joueur possède un item de chaque sorte. Les autres valeurs nécessaires pour compléter le modèle peuvent être choisies arbitrairement.



Pour la réalisation de ces diagrammes, nous avons décidé de diviser le diagramme en cinq sous-diagrammes mettant chacun l'accent sur un aspect du monde ci-dessus :

- Le jeu actuellement en cours.
- Le plateau de jeu en lui-même.
- Le monde tel qu'il est perçu par l'avatar 1 (en vert).
- Le monde tel qu'il est perçu par l'avatar 2 (en bleu).
- Le monde tel qu'il est perçu par le zombie.

2.1 Jeu



Le jeu actuellement en cours rassemble deux joueurs :

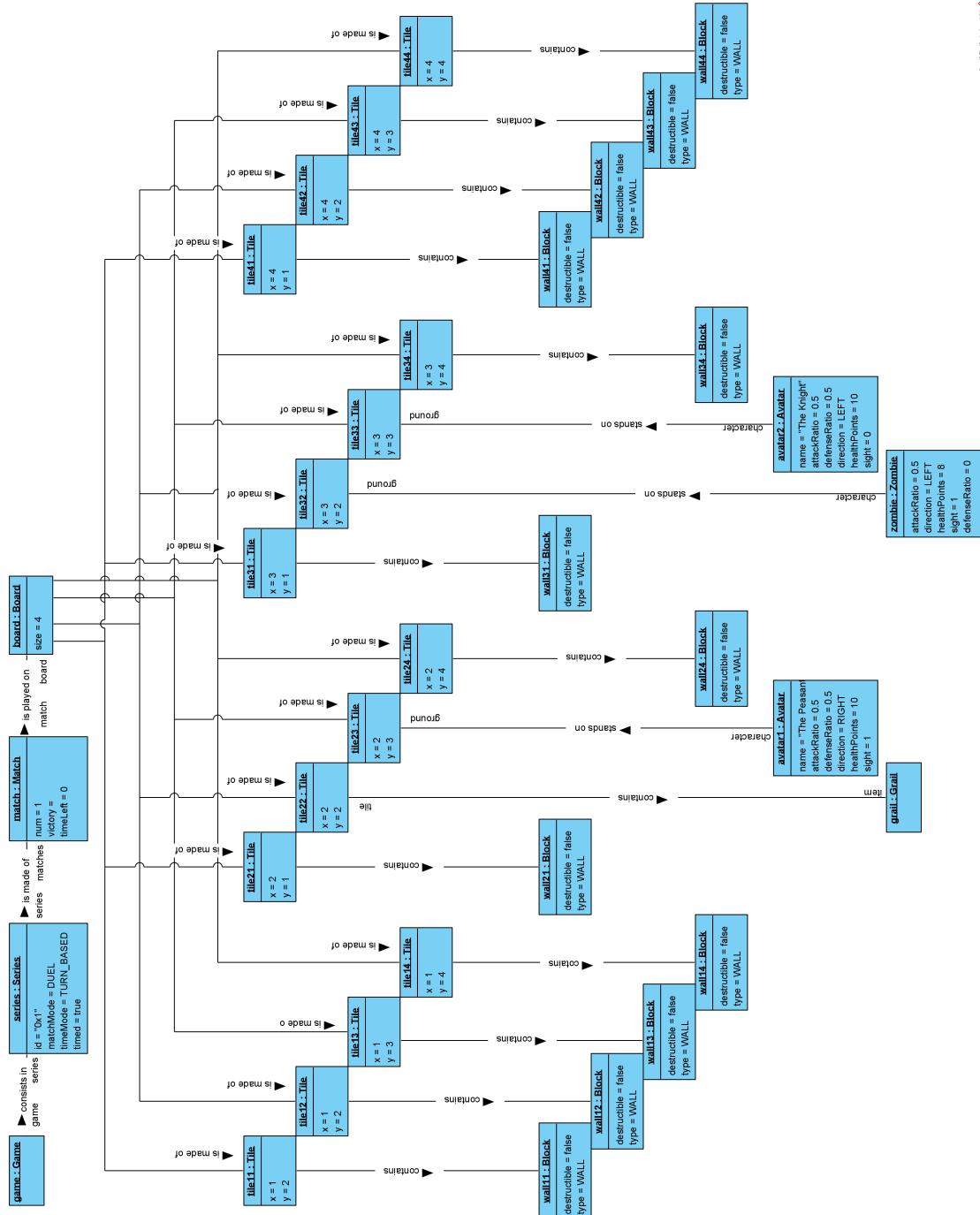
- Un joueur humain - le joueur 1.
 - Un joueur artificiel, contrôlé par l'ordinateur - le joueur 2.

Ces deux joueurs sont engagés dans une suite de rencontres en mode DUEL, en tour-par-tour avec une limite de temps. Ils ont pour cela chacun choisi un avatar pour disputer les rencontres de cette suite. L'identifiant unique de la série est 0x1.

La suite est composée d'un unique match, dans lequel sont engagés les joueurs et leurs avatars. Ce match porte le numéro unique 1, n'a pas encore été gagné (par mort subite ou découverte du Graal) et ne compte plus que 3 tours avant de s'achever.

C'est au tour du joueur 1 de choisir son action pour le tour.

2.2 Plateau

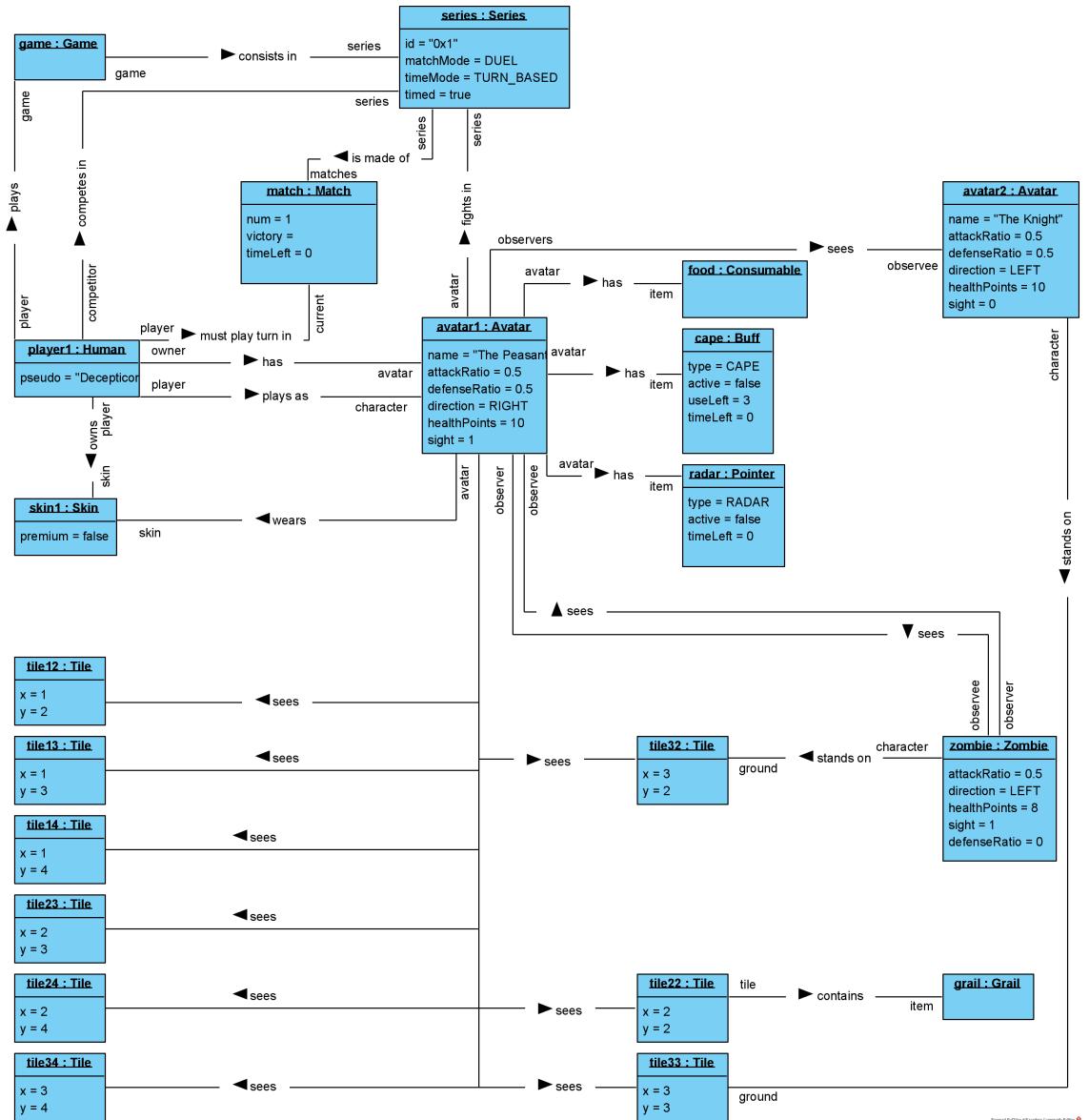


Le plateau présente 16 tuiles, dont 4 servent de plateau réel de jeu.

- Les bords comprennent tous un obstacle de type *Block* (c'est-à-dire infranchissable et indestructible).
 - Trois tuiles comprennent des personnages.
 - Une tuile comprend un objet, le Graal.

Nous avons volontairement omis ici de préciser les éléments des relations "*[1] board is made of [1..*] tiles*", afin de ne pas surcharger le diagramme.

2.3 Avatar 1



Le monde perçu par l'avatar 1 dépend de son indice de vision : celui-ci étant de 1, ce personnage "voit" sa tuile plus les 8 tuiles qui l'entourent. Ce faisant, il peut apercevoir l'avatar 2, le zombie et le Graal.

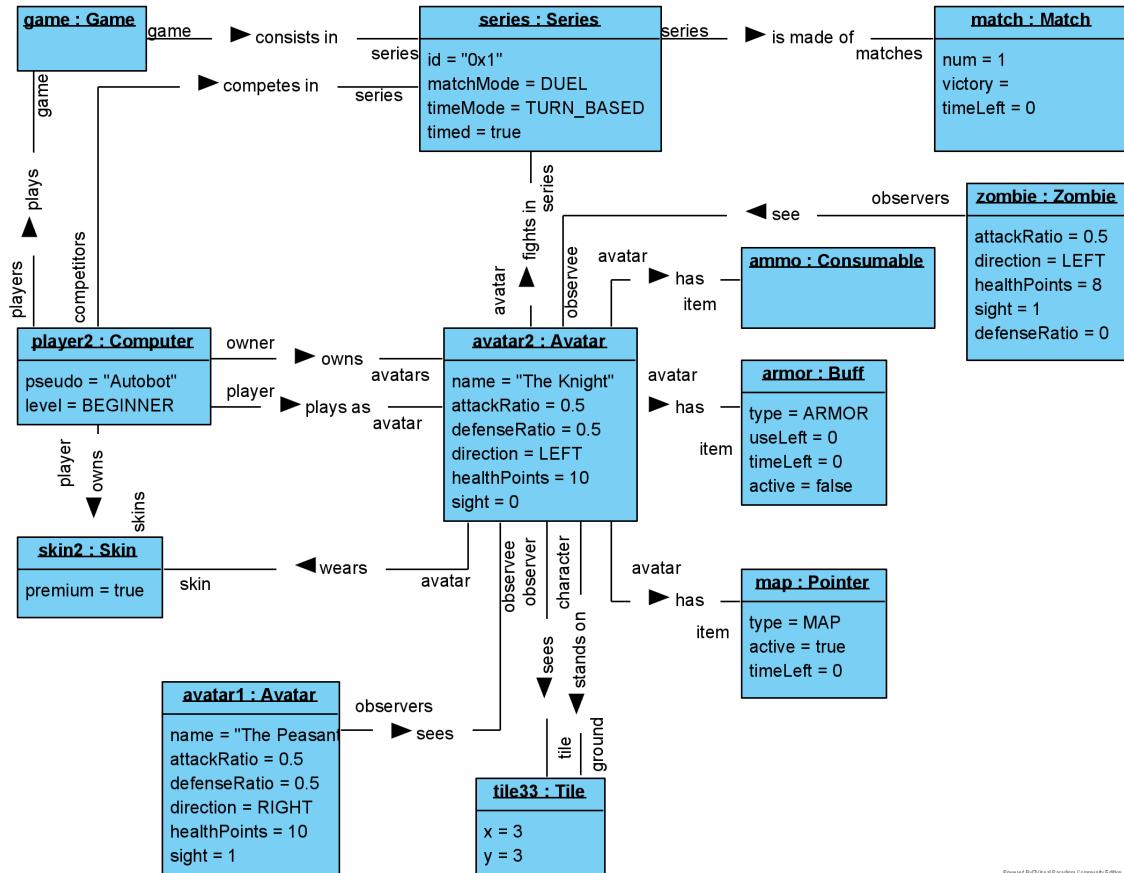
L'avatar utilise l'apparence par défaut, non-payante et a un inventaire constitué de 1 nourriture, une cape de mage avec 3 points d'utilisation et 9 tours restants et encore 10 points de vie. Il possède également un objet-radar dont il reste 1 tour.

Il porte en outre le nom de *The Peasant* et présente les ratios AP=0.5 et DP=0.5 et un solde de points de vie de 10. N'ayant pas encore pu faire de déplacement, sa direction a été arbitrairement initialisée à *Direction.RIGHT*.

Ce personnage est contrôlé par le joueur 1, dont c'est le tour de jouer.

Nous avons volontairement omis ici de préciser les éléments des relations "*[0..*] observers sees [1..*] tiles*", afin de ne pas surcharger le diagramme.

2.4 Avatar 2



Le monde perçu par l'avatar 2 est beaucoup plus restreint, puisque son indice de vision est 0 : il ne voit par conséquent que la tuile sur laquelle il est positionné. Ceci ne l'empêche pas d'être vu à la fois par le zombie et l'avatar 1 (qui possèdent une plus grande portée de vue).

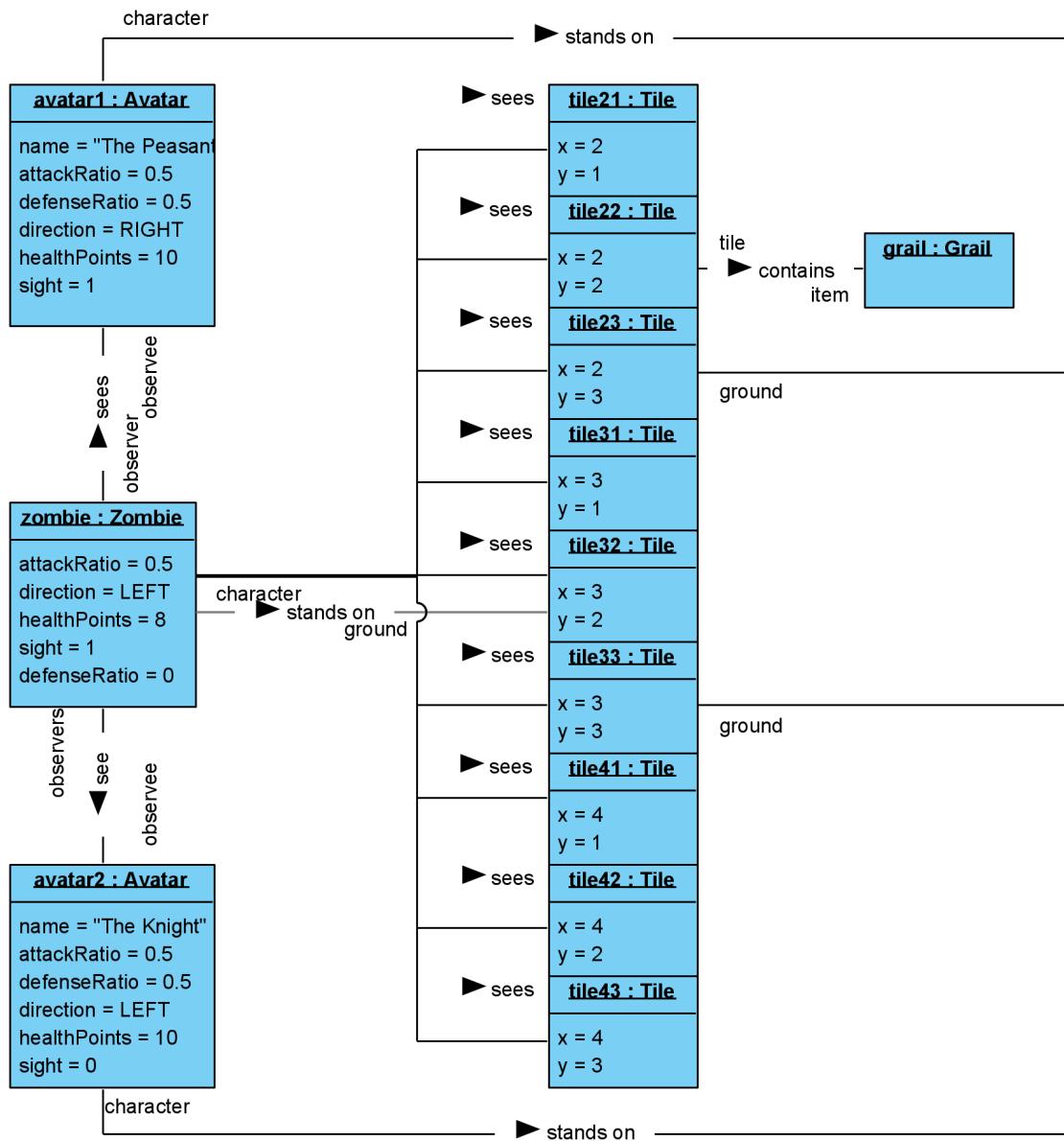
Ce personnage, dénommé *The Knight*, porte une apparence *Premium*, possédé par le joueur 2. Il possède en outre dans son inventaire une munition, une cotte de maille encore non-utilisée et un pointeur vers le Graal automatiquement activé (et sans limite d'utilisation).

Tout comme l'avatar 1, il présente les ratios AP=0.5 et DP=0.5 et un solde de points de vie de 10. Sa direction a également été posée arbitrairement comme *Direction.LEFT*.

Cet avatar est contrôlé par l'ordinateur et ne devrait pas poser un gros challenge (son niveau est débutant).

Nous avons volontairement omis ici de préciser les éléments des relations "*[0..*] observers sees [1..*] tiles*", afin de ne pas surcharger le diagramme.

2.5 zombie



Le zombie est pour sa part doté des caractéristiques suivantes : AP=0.5, DP=0;5, direction arbitraire vers la gauche. Il voit les deux personnages contrôlés par les joueurs, ainsi que 9 tuiles dont l'une contient le Graal.

Nous avons volontairement omis ici de préciser les éléments des relations "*[0..*] observers sees [1..*] tiles*", afin de ne pas surcharger le diagramme.

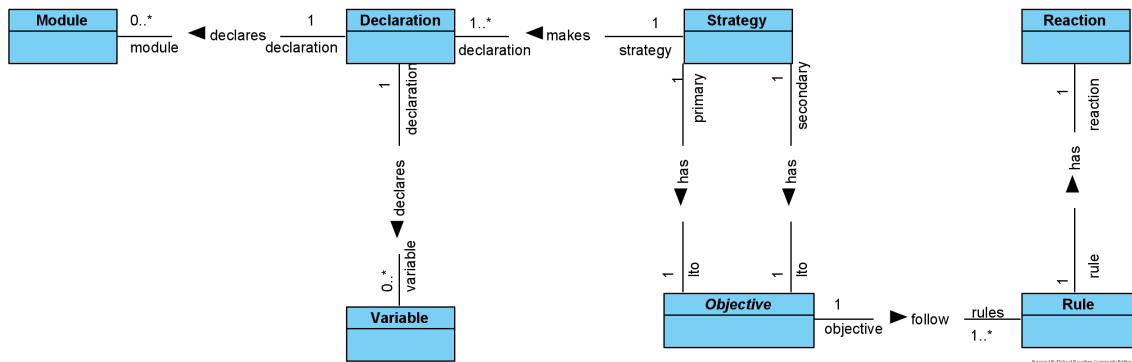
Powered By Object Paradigm Community Edition

Chapitre 3

Description des stratégies

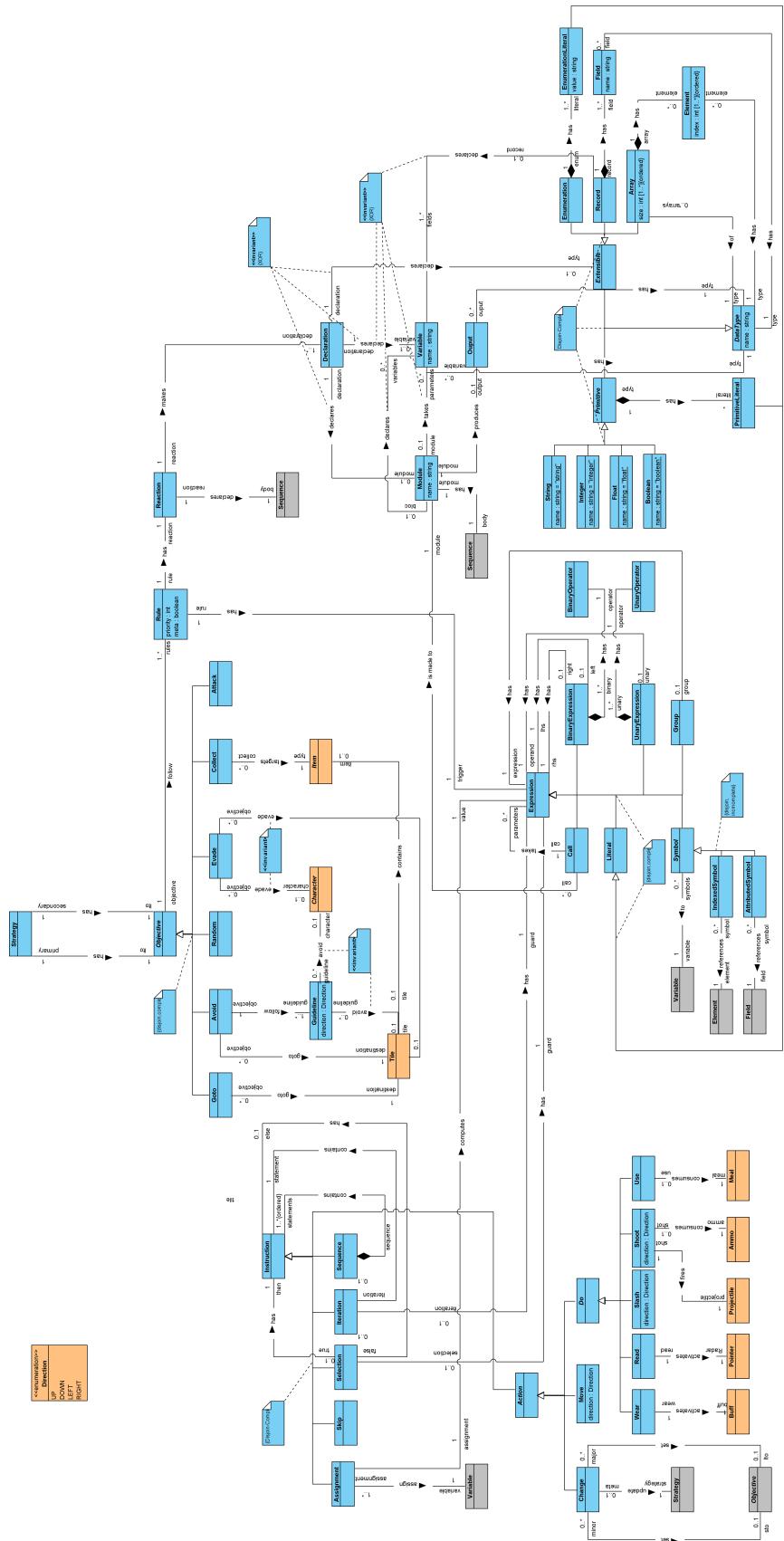
3.1 Pré-version

Question 4.7.



Les stratégies décrivent les mécanismes d'instructions et de réalisation de comportements automatisés au sein du jeu. Un personnage se voit ainsi attribuer une stratégie qui va guider le sens de ses actions et lui donner une série d'outils pour les mener à bien. On distingue ainsi les objectifs (primaires et secondaires) qui vont prescrire un comportement général et donner une série de directives pour réagir à certaines circonstances, et les modules qui sont des descriptions de sous-comportements.

3.2 Version détaillée

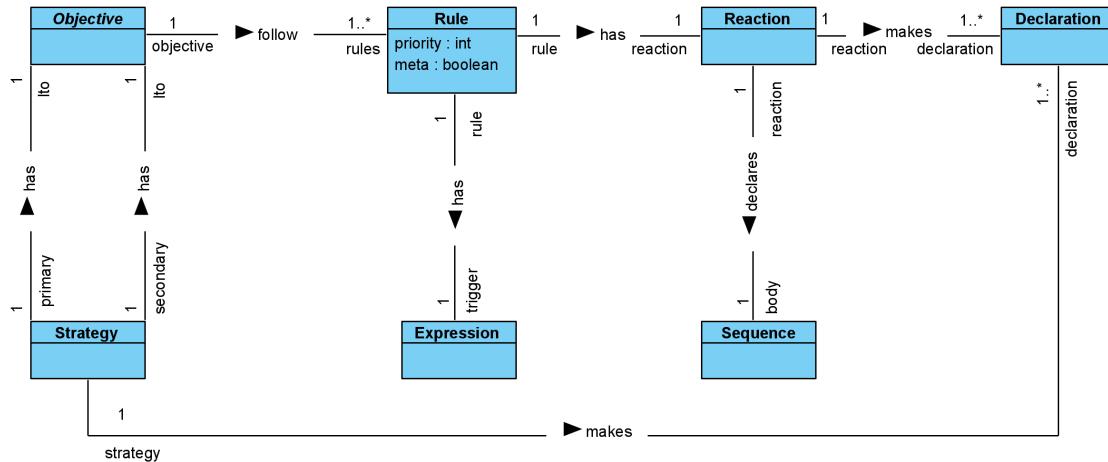


(En orange sont les classes issues du modèle de jeu, et en gris sont les classes qui ont été dupliquées pour permettre la réalisation d'un diagramme lisible).

Dans cette section, nous détaillerons les diagrammes spécifiques aux différentes composantes de la définition des stratégies.

3.2.1 Définition

Stratégie et directives



Les stratégies permettent de décrire le comportement prescrit à un personnage au cours d'une rencontre durant laquelle ce personnage est contrôlé automatiquement. Elles se composent de deux aspects :

- Un objectif principal (*Long-Term Objective - LTO*), qui sert de fil rouge et reste globalement stable.
 - Un objectif secondaire (*Short-Term Objective - STO*), déclaré en fonction des conditions avoisinantes.

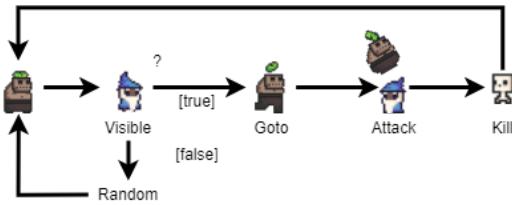
Au moment d'agir, le personnage fait l'état de ses objectifs : l'objectif secondaire étant présumé plus rapide à exécuter, celui-ci sera prioritaire dans l'exécution par rapport à l'objectif principal. Si l'objectif secondaire est atteint, alors le personnage peut soit en changer, soit se concentrer sur son objectif principal.

Pour réaliser un objectif, un personnage suivra un ensemble de directives : un ensemble de comportements spécifiques à activer lorsque les conditions le permettent. Nous avons ainsi les trois éléments suivants :

- Le déclencheur / la garde : une expression permettant de vérifier si les conditions sont réunies pour l'exécution.
 - La réaction / le corps : une suite d'instructions permettant de réaliser le comportement.
 - La priorité : un marqueur permettant de déterminer la directive à appliquer en cas de conflit entre plusieurs directives activées simultanément.

La directive choisie permet alors au personnage de réaliser un ensemble de comportements qu'elle documente dans une déclaration : elle fournit ainsi des variables significatives et des actions concrètes qui prendront le parti de ces variables.

Décrivons ici la stratégie suivie par les zombies à titre d'exemple.

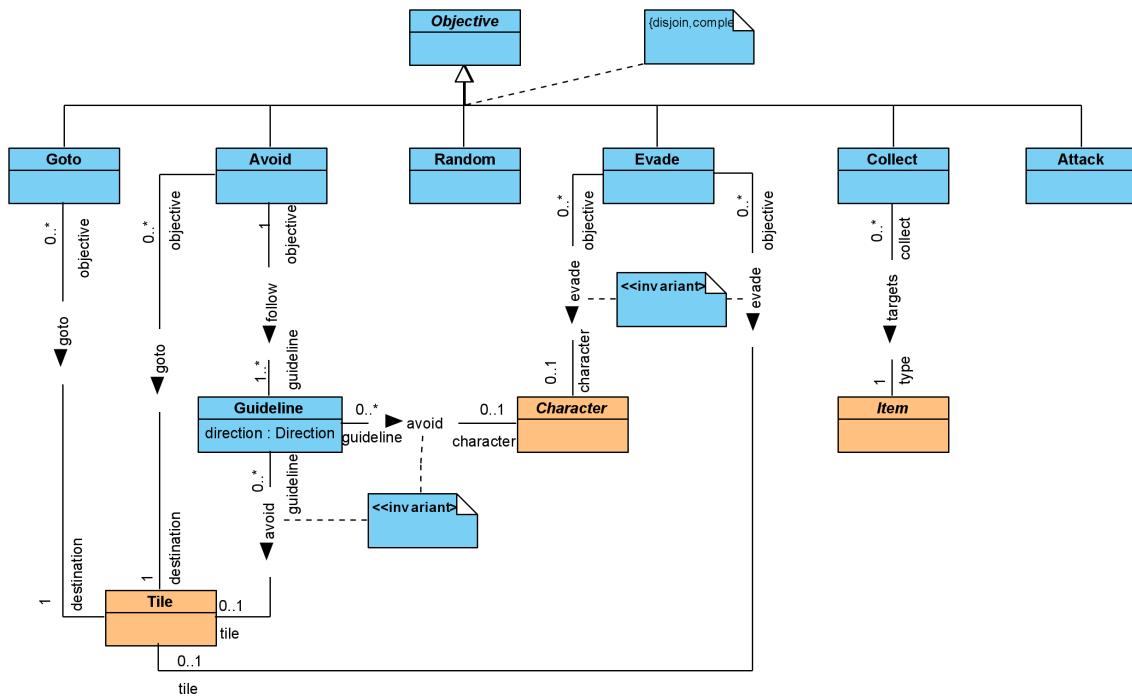


Les zombies poursuivent un objectif principal clair : tuer tous les humains (qu'ils soient contrôlés par un joueur ou un ordinateur). Il s'agit d'un objectif de *long-terme*, qui ne peut être réalisé si aucun humain n'est à proximité. Face à cela, les zombies adoptent un objectif secondaire de trouver les humains. Ils errent de manière aléatoire sur le plateau jusqu'à ce qu'ils croisent un personnage. Dès cet instant, l'objectif secondaire d'errance change pour intimider aux zombies de se rapprocher du personnage (de manière à pouvoir le mordre). Ce nouvel objectif s'accompagne de variables et d'actions très concrètes : information sur la position du personnage, choix de l'itinéraire le plus approprié pour s'en rapprocher, comment réagir face à un obstacle, etc. Les zombies se mettent alors en chasse du joueur jusqu'à l'atteindre.

Dès le moment où les zombies ont atteint le personnage, tous les objectifs secondaires sont remplis et les zombies peuvent alors appliquer leur objectif principal : *kill all humans*. Ils retournent enfin à leur phase d'errance et se remettent à chercher une nouvelle victime.

Objectif

Question 4.10.



Les objectifs dirigent les comportements des personnages de la manière suivante :

- L'objectif *Néant* laisse le personnage errer de manière aléatoire et ne nécessite aucun paramètre.
- L'objectif *AllerVers* donne à un personnage une case de destination et lui permet ainsi de déterminer à un chemin à suivre.

La spécification n'indique pas si l'objectif *AllerVers* peut prendre un personnage plutôt qu'une case (comme pour la stratégie d'évitement). Nous avons donc choisi de ne pas représenter cette possibilité.

- L'objectif *Contourner* donne à nouveau à un personnage une case du destination, mais également un ensemble de contraintes à respecter pour tracer son chemin jusqu'à cette case : un ensemble d'obstacles et/ou de personnages à éviter, ainsi que le côté cardinal par lequel les dépasser.

Les consignes de contournement prennent chacune en paramètre soit une case, soit un personnage (contrainte XOR).

- L'objectif *Éviter* est le pendant inverse d'un objectif *AllerVers* : le personnage reçoit à nouveau une case, mais celle-ci est cette fois-ci la case dont il faut s'éloigner. Le personnage peut donc à nouveau déterminer un chemin.

Les consignes d'évitement peuvent prendre une case ou un joueur (au moins un et jamais deux) (contrainte XOR).

- L'objectif *Combattre* ne prend pas de paramètre car il instruit au personnage non pas d'attaquer un ennemi en particulier, mais de choisir de lui-même selon une logique indéfinie les ennemis à attaquer : ennemi le plus proche, préséance de cible vers les autres joueurs plutôt que les zombies, etc.
- L'objectif *CollecterMax* donne à l'avatar un exemple d'objet à ramasser autant que possible.

Cet objectif diffère des autres objectifs en ce sens qu'il fait appel au méta-modèle : plus qu'un objet, c'est une classe d'objet telle que décrite dans la description de jeu que cet objectif prend en compte. Nous avons cependant choisi de procéder de manière assez pragmatique en donnant un *exemple* d'objet à ramasser plutôt qu'une abstraction issue du modèle.

Les objectifs sont dans une logique de spécialisation **disjinte** et **complète** : un objectif est nécessairement d'un et un seul sous-type. Le personnage peut cependant combiner deux objectifs grâce à la stratégie qui accepte un objectif de court terme et de long terme.

Par souci de cohérence linguistique, nous avons traduit les types d'objectifs comme suit :

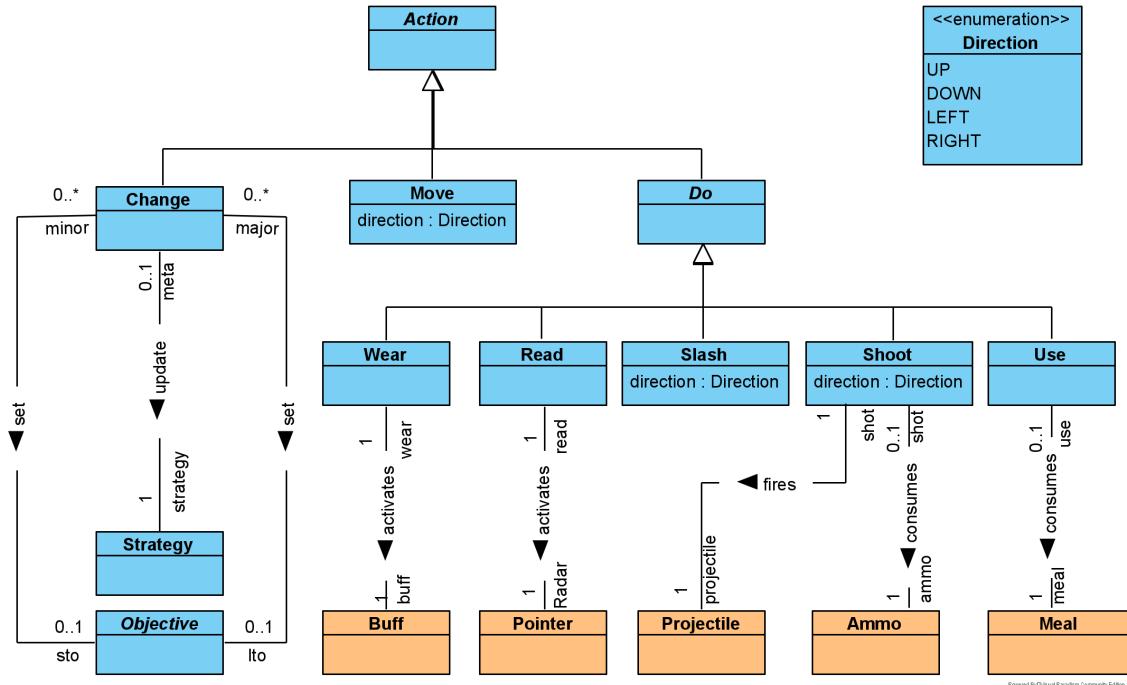
Terme	Traduction
Néant	<i>Random</i>
AllerVers	<i>Goto</i>
Contourner	<i>Avoid</i>
Éviter	<i>Evade</i>
Combattre	<i>Fight</i>
CollecterMax	<i>Collect</i>

Décrivons ici comment l'enchaînement d'objectif cumulé à leur priorité peut influencer leur comportement en prenant deux objectifs et en les réorganisant :

Stratégie	Cas 1	Cas 2
STO	Goto(tile)	Collect(Consumable)
LTO	Collect(Consumable)	Goto(Tile)
Réalisation	Le personnage tente de rejoindre une tuile et fera des détours si nécessaires pour collecter des objets consommables. (trajet avec détour)	Le personnage se rend vers une tuile pour collecter les objets alentours une fois sur place. (trajet direct)

Action

Question 4.11.



Les actions sont des types d'instruction spécifiques permettant au joueur de contrôler les actions de son personnage. Elles se décomposent en trois catégories :

- Les actions de type *mouvement* (MOVE) permettent au personnage de se déplacer sur le plateau de jeu dans une des directions cardinales.
- Les actions de type *activité* (DO) permettent au personnage d'interagir avec le monde qui l'entoure.
- Les actions de type *méta* (META) permettent au joueur de provoquer un changement de comportement de leur personnage.

Par souci de cohérence linguistique, nous avons traduit les types d'activités comme suit :

Terme	Traduction
Frapper	Slash
Tirer	Shoot
UtiliserItem	Use
Revêtir	Wear
ConsulterRadar	Read

Les actions nécessitent différents paramètres :

- L'action *Revêtir* nécessite logiquement d'avoir un objet enfilable (cape de mage ou cotte de maille).
- L'action *Lire* nécessite tout aussi logiquement d'avoir un objet d'aide à l'orientation à disposition.
- L'action *Frapper* ne prend pas de paramètre en entrée et consiste à faire tournoyer l'épée dans une certaine direction.

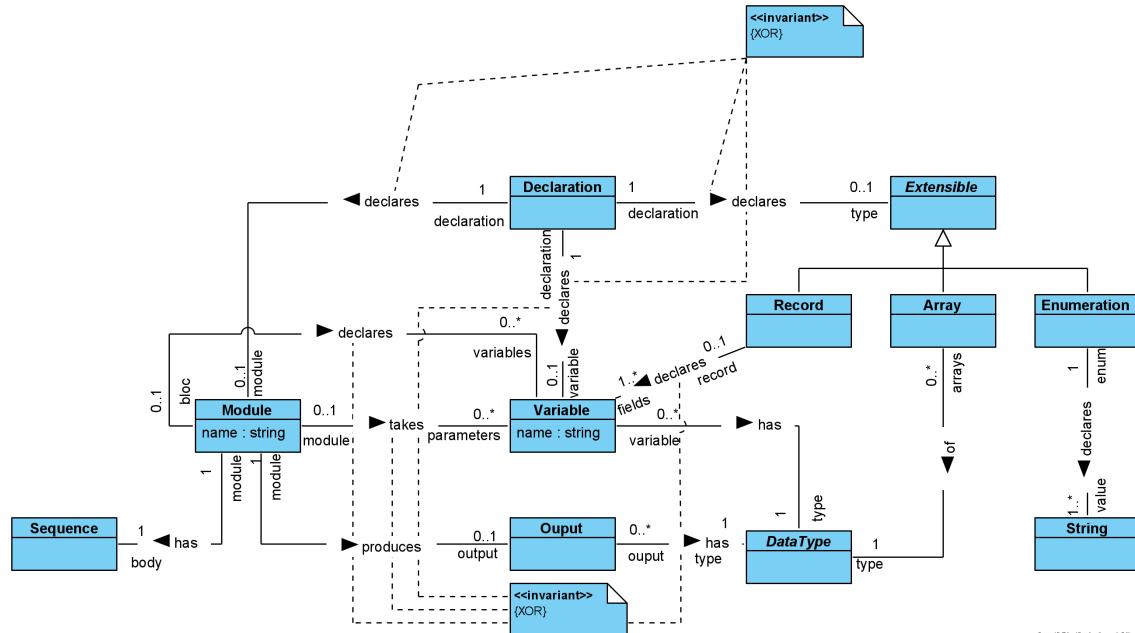
Nous aurions pu requérir que cette action nécessite d'avoir un ennemi ou un obstacle destructible en face. Cependant, il nous a semblé qu'un personnage pouvait jouer les Don Quichotte et de frapper dans le vide, sans cible, ou s'acharner à escrimer un mur infranchissable par frustration.

- L'action *Tirer* nécessite de pouvoir consommer une munition depuis l'inventaire et envoie alors un projectile dans la direction demandée.
- L'action *UtiliserItem* permet enfin de regagner quelques points de vie en mangeant un repas bien mérité (câd nourriture et boisson).

3.2.2 Langage

Déclaration

Question 4.9.



Nous avons dénombré trois types distincts de déclarations :

- Les déclarations de modules ;
- Les déclarations de variables ;
- Les déclarations de types extensibles (tableau, enregistrement, énumération).

Variables

```
var total : integer;
```

Modules

```
module shortestDistance
param start, stop : Cell
produces integer
begin
var count : integer
...
end
```

Types extensibles

```
type Cell : record
  x : integer;
  y : integer;
endrecord

type Sight = array[1..MAX] of integer;

type majuscule = 'A'..'Z';
```

Une variable est simplement une entité-mémoire nommée et associée à un type de données. Elles sont présentes à plusieurs niveaux dans le langage de programmation des stratégies :

- En tant qu'objet explicitement déclaré.
- en tant que paramètre d'un module
- en tant que champ d'un enregistrement.

Un module est quant à lui un objet plus complexe, qui encapsule une logique de programmation. Il possède lui aussi un nom et déclare lui-même une série de paramètres et une valeur de retour si nécessaire. Il possède en outre un corps d'instructions permettant de réaliser effectivement sa

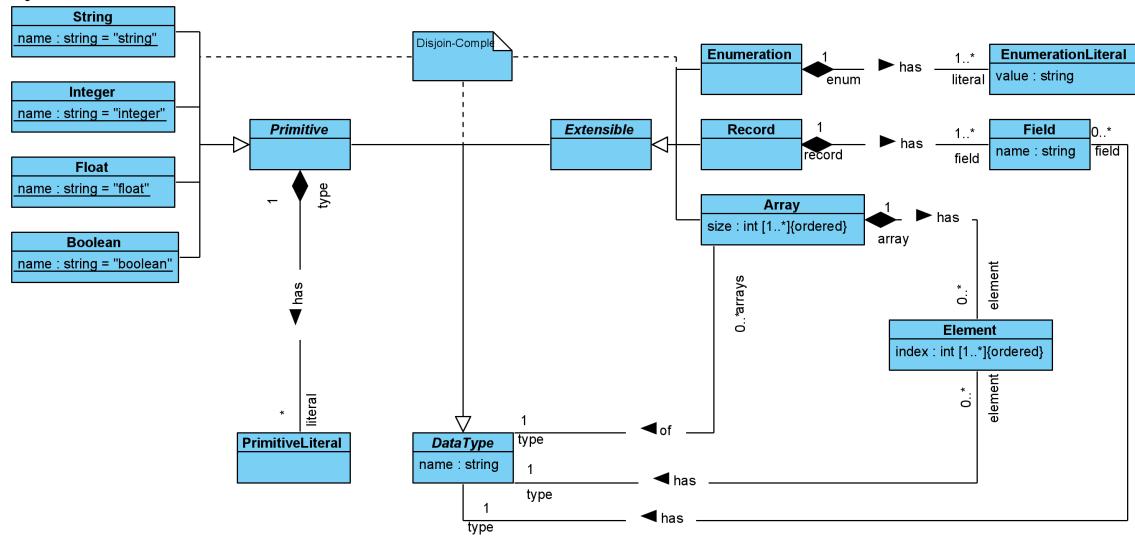
logique. Outre ses paramètres, le module peut également faire ses propres déclarations internes (de variables uniquement)

Enfin, un type extensible n'existe pas en tant que tel dans le langage : ils doivent nécessairement avoir été déclarés et définis à un moment ou l'autre dans le développement.

- Un enregistrement doit avoir été nommé et la liste de ses champs doit avoir été explicitée (nom et type).
 - Un tableau doit avoir été nommé, typé et dimensionné.
 - Une énumération doit avoir été nommée, et ses valeurs énoncées.

Type de données

Question 4.8.



Les types de données peuvent être divisés en deux catégories :

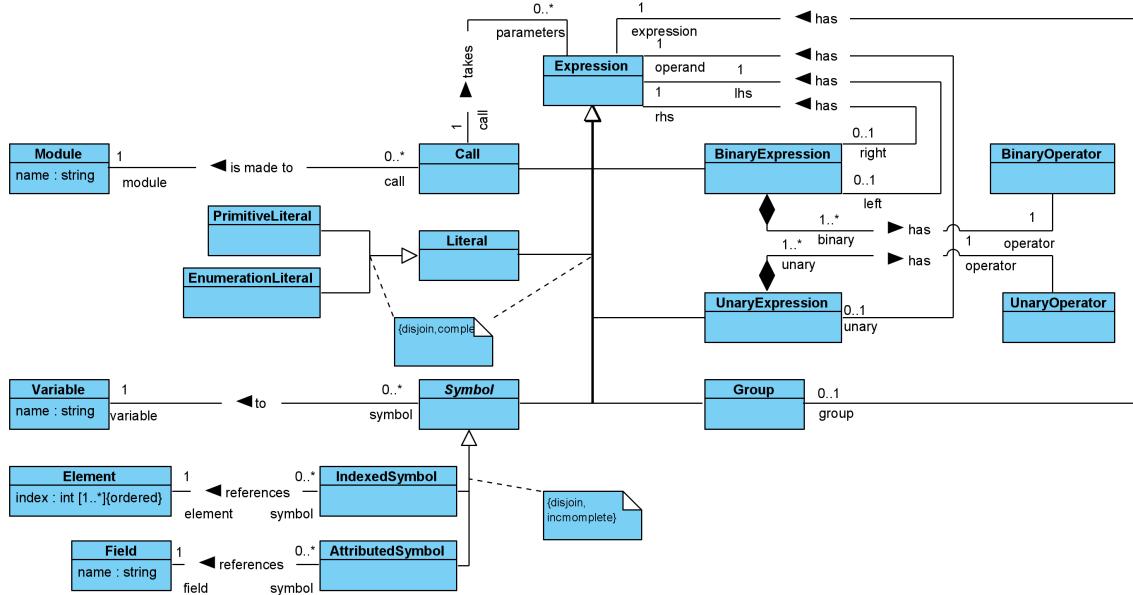
- D'une part, les types "*finis*" qui sont déclarés d'entrée de jeu de manière absolue et définitive :
 - types primitifs (entiers, décimaux, booléens, chaînes de caractères)
 - D'autre part, les types "*extensibles*", qui peuvent être ajoutés par les développeurs :
 - tableaux
 - enregistrements
 - énumérations

Tout type de données est ensuite associé à un ensemble de valeurs possibles.

- Les valeurs des types primitifs et des énumérations sont *immédiates*, car leur signification n'est pas sujette à interprétation (ils n'ont qu'une seule valeur et celle-ci est exprimée directement).
 - Les valeurs des types tableau et enregistrement ne sont en revanche pas aussi claires, ceux-ci étant avant tout des entités d'agrégation de données. Par récursion, ces types finissent toujours par être associés à des valeurs primitives ou des énumérations et peuvent ainsi recevoir une valeur *immédiate*.

Expression

Question 4.12.



Nous pouvons distinguer les expressions sous trois catégories :

- Les regroupements : expressions unaires, binaires et regroupées.
- Les appels : une expression structurée faisant appel à une séquence d'instruction.
- Les représentations : littéraux et symboles (tableaux, enregistrement, etc.).

Les expressions de **regroupement** sont marquées par une certaine forme de récursivité : elles peuvent être enchaînées de manière illimitée pour construire des expressions plus ou moins complexes et combiner des valeurs entre elles. Nous avons remarqué, pour les opérations unaires et binaires, la fonction structurelle des opérateurs : c'est la présence même de ces opérateurs qui constitue l'expression.

Les **appels de module** font également preuve d'une certaine récursivité, puisqu'ils acceptent des expressions comme des valeurs comme paramètres. Ils font appel à un et un seul module.

Les **représentations** sont la partie la plus complexe de ce diagramme : nous les avons réparties en deux groupes :

- Les valeurs *littérales*, dont le sens est immédiatement perceptible (comme précisé dans la spécification de ce projet et rappelé au point Datatype).
- Les valeurs *interprétables*, dont il faut évaluer la référence.

Les valeurs littérales font directement référence aux valeurs qui sont associées aux types primitifs et aux énumérations : il y a une bijection entre l'ensemble des valeurs concrètes et celui de leur représentation littérale dans les expressions. A l'inverse, les éléments symbolisés (*LHS* dans la spécification) sont des éléments qui peuvent représenter une même information d'autant de manières qu'il y a de déclarations. Nous avons cependant à nouveau distingué trois catégories :

- Les symboles *purs* font directement référence à une entrée en mémoire par le nom qui lui a été assigné au cours d'une déclaration.
- Les symboles *indiqués* font référence à un élément au sein d'une entrée en mémoire regroupée et rangée (tableau). Dans ce cas, le symbole fait référence à la variable de regroupement et indique ensuite l'emplacement de la valeur à récupérer.

- Les symboles *attribués* font référence à un élément au sein d'une entrée-mémoire structurée (enregistrement). Dans ce cas, le symbole fait référence par la variable à la structure générale et indique ensuite le nom de la valeur à récupérer au sein de cette structure.

La modélisation que nous avons choisi ici ne permet pas de gérer l'imbrication de symboles. Par exemple, il n'admettrait pas un symbole tel que `monTableau[1].monChamp`. Pour pouvoir traiter ce cas, notre modèle nécessite de passer par plusieurs opérations d'assignation successives :

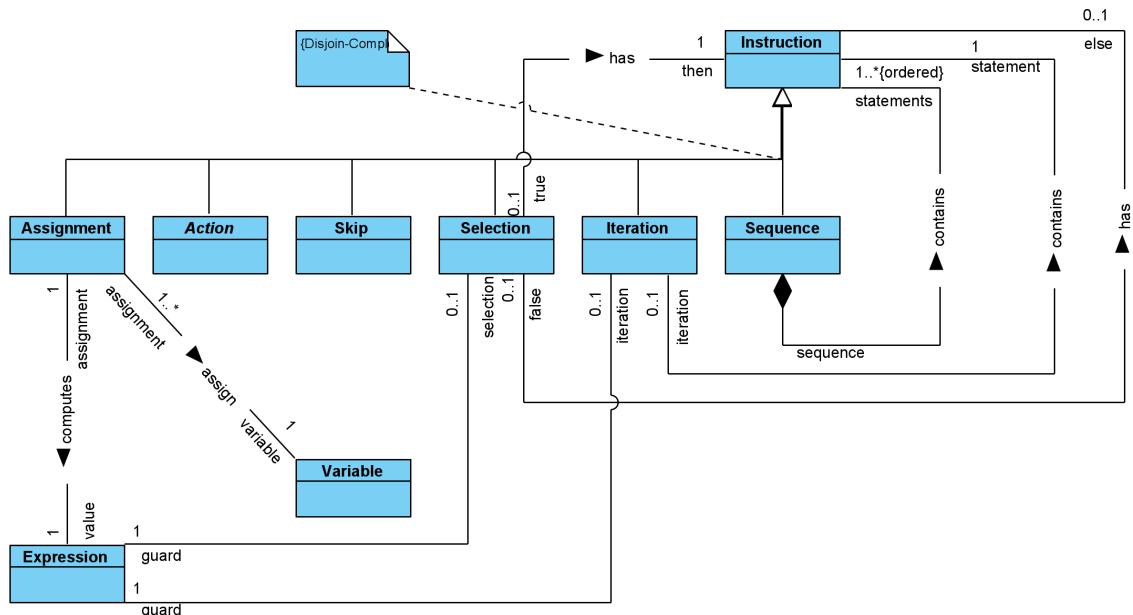
- Récupérer l'enregistrement dans une variable intermédiaire.
- Récupérer le champ souhaité depuis la variable intermédiaire.

```
var monEnregistrement : Record = monTableau[1];
var monChamp : string = monEnregistrement.monChamp;
```

Pour les symboles, nous avons choisi d'utiliser une relation de spécialisation disjointe, mais pas complète : un symbole n'est en effet pas forcément indicé ou attribué.

Instruction

Question 4.13.



Séquences, itérations et sélections sont des instructions qui n'existent que pour permettre l'occurrence d'autres instructions.

- Les séquences forment le gros de ces instructions :
 - toute instruction forme à elle seule une séquence de 1 instruction.
 - toute déclaration de module implique la déclaration d'une séquence d'instructions pour former son corps.
 - tout réaction déploie l'action d'un personnage à travers une série d'instructions et de tests pour déterminer l'action immédiate à prendre.

Une séquence d'instruction est, comme son nom l'indique, une suite ordonnée d'instructions. Prises dans le désordre, elles perdraient tout leur sens.

- Les itérations sont des instructions qui appellent la même instruction (qui peut être une séquence ou une sélection) encore et encore, sur base d'une expression de contrôle (la garde).

- Les instructions de sélection (ou branchement) permettent de choisir entre une (séquence d')instruction(s) gauche(s) et une (séquence d')instruction(s) droite(s), en fonction du résultat du test de la garde. Nous avons choisi de modéliser cette instruction par deux associations distinctes : cela nous permet de mettre en avant la contrainte d'avoir exactement deux branches distinctes, dont l'une est facultative.

On retrouve donc, comme pour les expressions, un principe fort de récursion dans les instructions. Pour terminer la récursion, nous avons trois instructions terminales :

- *Je pause (skip)*, qui vise à signifier l'absence d'action.
- *Je réfléchis (Assignment)*, qui vise à préserver en mémoire le résultat d'une procédure de calcul.
- *J'agis (Action)*, qui vise à réaliser le comportement (sur base d'un arbre d'instructions et des éléments qui auront été mémorisés).

3.3 Contraintes d'unicité

Question 4.14.

Spécifier les contraintes d'unicité suivantes :

- Les modules ont un nom unique au sein d'une même stratégie ;

```
context Strategy inv uniqueModules :
  self.declarations->selectByType(Module)
    ->forAll(m1,m2|m1<>m2 implies m1.name <> m2.name)
```

Nous vérifions ici que toutes les déclarations de module d'une même stratégie utilisent un nom différent.

- Les variables globales ont un nom unique au sein d'une même stratégie ;

```
context Strategy inv uniqueGlobalVars :
  self.declarations->selectByType(Variable)
    ->forAll(v1,v2|v1<>v2 implies v1.name<>v2.name)
```

De la même manière, nous vérifions ici que, pour une même stratégie, toutes les déclarations qui concernent des variables utilisent un nom différent.

- Les variables locales au sein d'un module possèdent un nom unique ;

```
context Module inv uniqueLocalVars :
  self.parameters->forAll(v1,v2|v1<>v2 implies v1.name<>v2.name)
```

Nous regroupons ici les variables locales déclarées par le module, qui doivent être toutes nommées différemment.

Nous ne vérifions pas ici si les variables locales sont des redéfinitions de variables globales.

- Les noms des paramètres d'un module sont tous différents ;

```
context Module inv uniqueParameters :
  self.parameters->forAll(p1,p2|p1<>p2 implies p1.name<>p2.name)
```

Nous vérifions ici que les paramètres sont tous nommés différemment. Nous ne vérifions pas contre pas si les paramètres et les variables déclarés au sein d'un même module ne sont pas en conflit pour un même nom.

- Pour ce faire, il aurait fallu joindre les deux collections de type *bag* puis comparer à nouveau les valeurs entre elles :

```
context Module inv uniqueVariables :
    self.parameters->union(self.variables)->forAll(e1,e2|e1<>e2 implies e1.->
        name<>e2.name)
```

- Les noms des types déclarés sont globalement uniques (en particulier, on ne peut pas nommer une énumération et un tableau de manière identique) ;

```
context DataType inv uniqueTypeName :
    Datatype.AllInstances()->forAll(dt1,dt2|dt1<>dt2 implies dt1.name<>dt2.name)
```

Pour toutes les instances de type de données existantes simultanément (indépendamment de leur localisation), il n'existe aucun autre type qui porte le même nom.

Nous posons ici le choix fort d'utiliser la méthode *AllInstances* pour s'assurer qu'il ne puisse jamais y avoir de redéfinition d'un type à aucun moment dans le code. Cette démarche nous a semblé acceptable dans la mesure où le nombre de types déclarés devrait rester dans un ordre de grandeur simple.

3.4 Contraintes de règles

Question 4.15.

Spécifier une contrainte OCL permettant de vérifier qu'une déclaration de type est bien formée :

- La liste des littéraux est unique au sein d'une énumération ;

```
context Enumeration inv uniqueLiterals :
    self.literals->forAll(l1,l2|l1<>l2 implies l1.value<>l2.value)
```

- liste des champs d'un enregistrement est non-vide ;

Ce point est déjà couvert par une contrainte de multiplicité sur l'association "*/1/record has [1..*] fields*".

- les noms des champs sont uniques au sein d'un même enregistrement ;

```
context Record inv uniqueFields :
    self.fields->forAll(f1,f2|f1<>f2 implies f1.name<>f2.name)
```

- un tableau comporte au moins une dimension ;

Il n'est pas nécessaire de préciser cette contrainte en OCL : celle-ci est déjà présente dans la cardinalité de l'attribut multiple *dimensions* du tableau : celui-ci doit être strictement supérieur ou égal à 1.

- Chaque dimension de tableau doit être strictement positive.

```
context Array inv positiveDimension :
    self.dimensions->forAll(dimension|dimension>0)
```

3.5 Programmation par contrat

3.5.1 Expression : :type()

Question 4.16.

Supposons l'existence d'une opération Expression :: type() : Type définie dans la classe Expression. Spécifier le contrat sur le résultat produit par cette opération pour vérifier que le type retourné correspond :

Pour répondre à cette question, nous allons d'abord détailler les différents cas de figures de manière individuelle. Une contrainte OCL complète unifiant tous les cas sera ensuite proposée en fin de réponse.

- au type du littéral (par exemple, le type de la valeur true doit d'être Boolean , celui de la chaîne "123" doit d'être String et celui de l'entier 122 Integer, etc.);

```
context PrimitiveLiteral::type(): DataType
  pre nothing      : --none
  post primitiveType: result = self.type
```

- **Précondition** : aucune précondition spécifique n'est à déclarer pour cette opération.
- **Postcondition** : Le résultat est le type primitif associé à ce littéral.
- à l'opérateur unary, à condition que sa sous-expression corresponde à ce type. Par exemple, les types de expressions unaires -123 et not isVisible doivent respectivement d'être Integer (puisque 12 est entier) et Boolean (à condition que la variable isVisible soit déclarée comme une variable booléenne) ;

```
context UnaryOperator::type(): DataType
  pre nothing      : --none
  post unaryType: result = self.operand.type()
```

- **Précondition** : aucune précondition spécifique n'est à déclarer pour cette opération.
 - **Postcondition** : Le résultat est le type déduit depuis la sous-expression.
- Nous prenons ici le parti de la récursivité : tant que l'expression contient une sous-expression, l'opération type() continue de s'appeler, jusqu'à atteindre une expression littérale ou un symbole qui renvoie vers un type de données modélisé.
- le type correspondant à l'opérateur binaire, à condition que les types des sous-expressions soient cohérentes. Par exemple, 12 + total utilise un opérateur entier + sur deux sous-expressions entières, mais 12 + isVisible est incohérent ;

```
context BinaryOperator::type(): DataType
  pre isValidBinary: self.operands->forAll(e1,e2|e1.type=e2.type)
  post binaryType   : result = self.operands->first().type()
```

- **Précondition** : les deux membres de l'expression binaire doivent être de même type.
 - **Postcondition** : le résultat est le type de la première expression.
- Nous savons par précondition que les deux expressions sont de même type : l'une ou l'autre peut donc être choisie arbitrairement et renvoyer tout de même le bon résultat. Nous utilisons en outre à nouveau la récursion sur les types d'expression.
- le type de la sous-expression dans une expression parenthésée ;

```
context Group::type(): DataType
  pre nothing      : --none
  post exprType: result = self.expression.type()
```

- **Précondition** : aucune précondition spécifique n'est à déclarer pour cette opération.
- **Postcondition** : Le résultat doit être le type déduit de la sous-expression.

Une fois encore, nous nous appuyons sur la récursivité présente dans le super-type expression.

- *le type de sa déclaration pour une expression d'accès à la valeur d'une variable (cf. exemples plus haut avec isVisible) ;*

```
context Symbol::type(): DataType
  pre nothing : --none
  post symbolType: result = self.variable.type
```

- **Précondition** : aucune précondition spécifique n'est à déclarer pour cette opération.
- **Postcondition** : le résultat est le type déclaré de la variable.

Cette opération est permise grâce à la partition incomplète des sous-types de *Symbol* : certains symboles ne sont en effet que des références vers des variables *normales* et peuvent donc renvoyer que le type de cette variable. Pour les symboles à indices ou à champs, une méthode surchargée sera appliquée (cf. ci-dessous).

- *le type de l'énumération pour une expression d'un accès à un littéral d'énumération. Par exemple, ‘Direction.Up’ doit renvoyer le type énumération ‘Direction’;*

```
context EnumerationLiteral::type(): DataType
  pre nothing : --none
  post enumType: result = self.enum
```

- **Précondition** : aucune précondition spécifique n'est à déclarer pour cette opération.
- **Postcondition** : le type renvoyé sera l'énumération à laquelle appartient le littéral d'énumération.
- *le type de la déclaration du champ dans une expression d'accès à un champ. Par exemple, origine.x doit retourner Integer puisque le champ x est déclaré comme tel ;*

```
context AttributedSymbol::type(): DataType
  pre isValidField : self.variable->oclAsType(Record)->one(field=self.field)
  post symbolType : result = self.field.type
```

- **Précondition** : la variable passée en paramètre doit être un enregistrement contenant le champ demandé par le symbole d'attribut.
- **Postcondition** : le résultat renvoyé est le type du champ.
- *le type du contenu du tableau pour une expression d'accès à un élément de tableau. Par exemple, à partir des exemples déclarés en section Types , les expressions visible[0,0] et area[0] doivent respectivement retourner Integer et VisibleLine .*

```
context IndexedSymbol::type(): DataType
  pre isContained: self.element.index->size()
    = self.variable->oclAsType(Array).dimensions->size() and
    self.element.index
    ->forAll(i|i>=0 and
      i<self.variable->oclAsType(Array)
      .dimensions->at(indexOf(index)))
  post symbolType : result = self.variable.type
```

- **Précondition** : l'indice/les indices de l'élément recherché par le symbole indicé doivent être compris dans les limites de dimension du tableau correspondant.
- **Postcondition** : Le type renvoyé est le type déclaré du tableau.

Il s'agit ici de s'assurer que l'élément recherché appartient bien au tableau demandé. On présume ici que l'élément est du même type que le tableau.

Une fois toutes ces fonctions spécifiées, il suffit au moment de la réception d'une expression d'interpréter son sous-type et d'appliquer l'opération surchargée qui lui correspond.

- **Précondition** : aucune précondition spécifique n'est nécessaire ici.
 - **Postcondition** : le résultat correspond au résultat propre au sous-type correspondant.

La norme UML précise qu'une opération peut être redéfinie par un sous-type :

An Operation may be redefined in a specialization of the featuringClassifier. This redefinition may add new preconditions or postconditions, add new raisedExceptions, or otherwise refine the specification of the Operation. [4, p. 117]

Il s'agit ici en effet d'appliquer le principe de *substitution de Liskov* [1].

Nous n'avons malheureusement pas pu trouver d'exemple concret permettant d'illustrer cette redéfinition et de confirmer notre notation : il y a fort à parier que l'opération décrite ci-dessus soit redondante en raison du principe de substitution, voire jamais appelée (puisque Expression est abstraite).

3.5.2 Avatar ::canCross(tile :Tile) : Boolean

Question 4.17.

Supposons l'existence d'une opération estTraversableGraceAuxItems(...) : Boolean, qui rend vrai si et seulement si, pour un personnage donné, une case passée en paramètre est traversable grâce aux items que le personnage possède sans l'intervention du joueur, c'est-à-dire que la case est sur une zone de feu, d'eau ou de glace et que le personnage possède respectivement des bottes pare-feu, un kit de plongée et des bottes à crampon.

- Quelle classe de votre Diagramme de Classe pourrait contenir cette opération ?

La classe la plus appropriée est celle de l'avatar : c'est lui dont on veut vérifier la capacité à traverser une tuile donnée, en fonction des objets qu'il possède dans son inventaire.

- Définir cette opération en précisant quel(s) seraient ses paramètre(s), et quel serait le contrat OCL sur cette opération.

```

context Avatar::canCross(tile:Tile): Boolean
  pre validTile : tile->notEmpty()
  post isCrossable: result =
    tile.obstacle->isEmpty()
  or (tile.obstacle.oclIsTypeOf(Area) and
        (tile.obstacle.type=AreaType::FIRE
        or (tile.obstacle.type=AreaType::ICE and
              self.items->select(i|i.oclIsTypeOf(Capaciter) and
                i.type=CapaciterType::ICEBOOT)->notEmpty())
        or (tile.obstacle.type=AreaType::WATER and
              self.items->select(i|i.oclIsTypeOf(Capaciter) and
                i.type=CapaciterType::DIVEMASK)->notEmpty()←
                  ))
        or (tile.obstacle.oclIsTypeOf(Area) and
              self.items->select(i|i.oclIsTypeOf(Buff) and
                i.type=BuffType::CAPE)->notEmpty())
        or (self.items->select(i|i.oclIsTypeOf(Buff) and
          i.type=BuffType::CAPE and
          i.useLeft>0)->notEmpty())
)

```

- **Précondition** : la tuile doit être non-nulle.
- **Postcondition** : la tuile est traversable selon plusieurs cas de figure.
 - La tuile ne contient aucun obstacle.
 - La tuile contient une zone de feu et l'avatar peut la franchir sans condition. Le fait d'avoir des *bottes de feu* ne lui permet que d'éviter de perdre des PV.
 - La tuile contient une zone d'eau ou de glace et dans ce cas, l'avatar doit posséder l'objet-capaciteur correspondant.
 - La tuile contient une zone et le personnage possède la cape de mage dans ses objets.
 - La tuile contient n'importe quel obstacle et le personnage possède la cape de mage avec des points d'utilisation restants.

Il s'agit ici de vérifier uniquement si la tuile concernée est traversable par le joueur : rien ne nécessite de vérifier si l'avatar du joueur est placé à côté ni même si la tuile fait partie du plateau de jeu sur lequel se trouve l'avatar.

Nous appliquons ici le *principe de responsabilité unique* [2].

Nous ne vérifions pas non plus que la tuile ne contient pas d'ennemi, puisque ce n'est pas l'objet de la méthode et que cette contrainte est déjà exprimée sur le diagramme de classe par les cardinalités.

3.5.3 Avatar : :collect() : Void

Question 4.18.

Supposons l'existence d'une opération ramasser(...) : Void dont l'effet est le suivant : si la case passée en paramètre contient un item ramassable, alors préciser le contexte (càd. sur quelle classe est défini l'opération) et la signature (càd. les paramètres) de l'opération ramasser et définir le contrat sur son résultat.

La classe à laquelle s'applique cette opération est bien entendu *Avatar* : le contexte est défini par le personnage, puisque c'est lui qui bénéficie de l'opération de ramassage.

- Si l'item est un bonus de défense ou d'attaque, le ratio correspondant du personnage se trouve modifié en multipliant l'ancienne valeur par la valeur du bonus ;

```
context Avatar::collectBooster(booster:Booster): Void
  pre itemPresent: booster->notEmpty()
  post apBooster :
    if booster.type=BoosterType::ATTACK
      then self.attackRatio = self.attackRatio@pre * booster.value
    else self.attackRatio = self.attackRatio@pre
    endif
  post dpBooster :
    if booster.type=BoosterType::DEFENSE
      then self.defenseRatio = self.defenseRatio@pre * booster.value
    else self.defenseRatio = self.defenseRatio@pre
    endif
```

- **Précondition** : l'objet doit être non-nul.
- **Postcondition** : en fonction du type de Booster ramassé, le ratio d'attaque ou de défense de l'avatar est modifié proportionnellement à la valeur du booster.
- Si l'item est un radar, il double la visibilité de la position de l'adversaire ;

```
context Avatar::collectPointer(pointer:Pointer): Void
  def packedRadar :
    Pointer = self.items->selectByType(Pointer)
      ->select(i|i.oclAsType(Pointer).type=Pointer::RADAR)
      ->first()
  def packedMap :
    Pointer = self.items->selectByType(Pointer)
      ->select(i|i.oclAsType(Pointer).type=Pointer::MAP)
      ->first()
  pre itemPresent: pointer->notEmpty()
  post addRadar : if pointer.type=Pointer::RADAR
    then if packedRadar->notEmpty()
      then packedRadar.timeLeft = packedRadar.timeLeft@pre * 2
      else self.items.append(pointer)
      endif
    else self.items=self.items@pre
    endif
  post addMap : if pointer.type=Pointer::MAP
    then if packedMap->isEmpty()
      then self.items.append(pointer)
      else self.items = self.items@pre
      endif
    else self.items = self.items@pre
    endif
```

- **Précondition** : l'objet doit être non-nul.
- **Postcondition** : Si l'avatar possède déjà un radar dans son inventaire, sa durée restante est doublée, sinon le nouveau radar est ajouté à l'inventaire.
- Si l'item est de la nourriture, de l'eau, des munitions, des bottes, des palmes, une cape ou une cotte de maille, l'item est simplement rajouté dans l'inventaire.

```
context Avatar::collectConsumable(consumable:Consumable): Void
  pre itemPresent: consumable->notEmpty()
  post isPacked: self.items.append(consumable)
```

```
context Avatar::collectBuff(buff:Buff): Void
  pre itemPresent: buff->notEmpty()
  post isPacked: self.items.append(buff)
```

```

context Avatar::collectCapaciter(capaciter:Capaciter): Void
  pre itemPresent: capaciter->notEmpty()
  post isPacked: self.items.append(capaciter)

```

- **Précondition** : l'objet doit être non-nul.
- **Postcondition** : l'objet est simplement ajouté à l'inventaire de l'avatar.

Maintenant que nous avons défini les comportements propres à chaque type d'objet, nous pouvons établir le comportement général au ramassage d'objet :

```

context Avatar::collect(tile : Tile): Void
  def item : Item = tile.item@pre
  pre isOnTile : self.ground = tile
  post tileLooted: tile.item->isEmptypost itemPacked:
    if item->notEmptythen if item.oclIsTypeOf(Booster)
        then self.collectBooster(item->oclAsType(Booster))
      else if item.oclIsTypeOf(Consumable)
        then self.collectConsumable(item->oclAsType(Consumable))
      else if item.oclIsTypeOf(Pointer)
        then self.collectPointer(item->oclAsType(Pointer))
      else if item.oclIsTypeOf(Buff)
        then self.collectBuff(item->oclAsType(Buff))
      else if item.oclIsTypeOf(Capaciter)
        then self.collectCapaciter(item->oclAsType(Capaciter))
      else self = self@pre
    endif
  endif
  endif
endif

```

- **Précondition** : l'avatar doit se trouver sur la tuile correspondante.
- **Postcondition** : si la tuile contient un objet, alors elle est *pillée*, tandis que l'effet spécifique de l'objet est appliqué à l'avatar en fonction des règles énoncées ci-avant.

3.5.4 Instruction : :isValid()

Question 4.19.

Spécifier le contrat OCL sur une opération ‘Instruction : :isValid() : Boolean‘ qui vérifie qu'une instruction est valide :

- L'instruction Skip est toujours valide ;

```

context Skip::isValid(): Boolean
  pre nothing: --none
  post isTrue : result = true

```

- **Précondition** : aucune précondition n'est applicable pour cette opération.
- **Postcondition** : vrai dans tous les cas.
- La garde d'une conditionnelle ou d'une itération doit être de type booléen ;

```

context Selection::isValid(): Boolean
  pre nothing : --none
  post booleanGuard: result = self.guard.type().oclIsTypeOf(Boolean)

```

```

context Iteration::isValid(): Boolean
  pre nothing : --none
  post booleanGuard: result = self.guard.type().oclIsTypeOf(Boolean)

```

- **Précondition** : aucune précondition n'est applicable pour cette opération.
- **Postcondition** : vrai si la garde de l'instruction est une expression dont le type est *Boolean*.
(cf. Question 4.16.).
- *La partie gauche et droite d'une affectation doivent être de même type.*

```

context Assignment::isValid(): Boolean
  pre nothing : --none
  post sameType: result = self.variable.type = self.expression.type()

```

- **Précondition** : aucune précondition n'est applicable pour cette opération.
- **Postcondition** : vrai si le type associé au symbole de destination correspond au type retourné par l'expression.
(cf. Question 4.16.)

Maintenant que tous ces opérations sont décrites pour chaque sous-type, nous pouvons détailler l'opération générale.

```

context Instruction::isValid(): Boolean
  pre nothing : --none
  post sameType:
    if self.oclIsTypeOf(Skip)
    then result = true
    else if self.oclIsTypeOf(Selection)
      then result = self->oclAsTypeOf(Selection).isValid()
    else if self.oclIsTypeOf(Iteration)
      then result = self->oclAsTypeOf(Iteration).isValid()
    else if self.oclIsTypeOf(Assignment)
      then result = self.statements->forAll(s|s.isValid())
    else if self.oclIsTypeOf(Action)
      then result = true
      else return invalid
    endif
  endif
  endif
  endif

```

A nouveau, nous utilisons le principe de redéfinition des opérations autorisé par la spécification UML [4], avec la réserve sur la formulation de l'opération au niveau du super type.

3.6 Questions de priorités

Question 4.20.

Spécifier une contrainte OCL vérifiant la cohérence des règles d'une stratégie par rapport à son objectif :

- *Toutes les règles d'un même objectif (à court ou long terme) ont des priorités différentes pour assurer leur déclenchement déterministe.*

```

context Rule inv hasPriority :
    self.priority->notEmpty()

```

```

context Objective inv uniquePriorities :
    self.rules.priorities->forAll(p1,p2|p1<>p2
                                implies p1.priority<>p2.priority)

```

On établit ici deux invariants

- Un invariant sur *Rule* qui statue que toute directive a nécessairement une priorité.
- Un invariant sur *Objective* qui s'assure que toutes les directive différentes ont une priorité différente.
- *Une réaction de règle ne contient qu'une seule action 'seDéplacer'.*

```

context Reaction inv uniqueMovement :
    self.sequence->closure(statements)->forAll(statement|statement.oclIsKindOf( $\leftrightarrow$ 
        Action)) and
    self.sequence->closure(statements)->selectByType(Move).size()=1

```

Une réaction ne peut être composée que d'une séquence d'instruction d'action, et celles-ci ne peuvent contenir qu'une seule action de mouvement.

- *Seules les métarègles peuvent contenir l'action 'changer'.*

```

context Rule inv changeOnlyMeta :
    self.sequence->closure(statements)->exists(statement|statement.oclIsTypeOf( $\leftrightarrow$ 
        Change))
    implies self.meta=true

```

S'il existe une action de type changement parmi l'une des actions appartenant à une directive, alors cette dernière doit avoir le type "*meta*".

Il s'agit d'une implication : cela signifie qu'une règle méta n'a pas nécessairement d'action de type *Change*.

Chapitre 4

Diagrammes d'objet

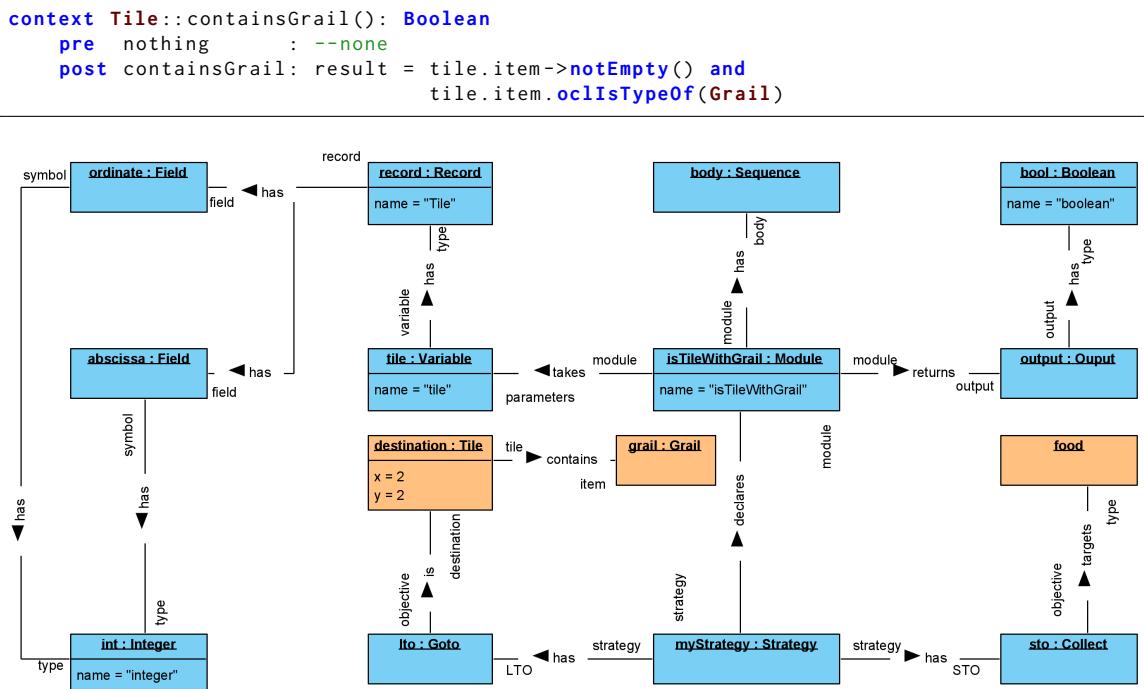
4.1 Tile ::containsGrail()

Question 4.21.

À partir de la situation définie en question 4.6, définir une stratégie constituée des visions suivantes :

- à long terme, se rendre vers le Graal ;
- à court terme, ramasser un maximum de nourriture.

On ne définira pas les règles correspondantes, mais on déclarera en plus dans la stratégie un module estCaseGraal(...) : Boolean qui renvoie vrai si et seulement la case contient le Graal. On supposera une variable réservée result dont le type correspond au type de retour du module, et qui sera affectée du résultat.



La stratégie est composée de deux objectifs :

- Un objectif de long-terme de type GOTO (*AllerVers*) - se rendre à une destination.
- Un objectif de court-terme de type COLLECT (*CollecterMax*) - ramasser le plus d'objets possible sur le chemin.

Pour réaliser ces objectifs, la stratégie a déclaré un module permettant de tester si une tuile contient le Graal. Ce module accepte un paramètre en entrée (une tuile *tile*), produit une sortie de type booléen, et est composé d'une séquence indéterminée d'instructions.

```
module isTileWithGrail
param tile : Tile
produces boolean
begin
...
end
```

Le type tuile est un symbole de type *Record* composé lui-même de deux champs (*Field*) dont le type est *Integer*. Ce sont ces champs qui ont la position de la tuile.

```
type Tile : record
  x : integer;
  y : integer;
endrecord
```

Chapitre 5

Maturité et modularité

Question 4.22

Dans l'état actuel du jeu, une case ne peut pas superposer un item sur un obstacle. Pourtant, ce serait parfois utile de donner au joueur un gros bonus à condition qu'il puisse traverser une zone de feu.

- *Comment proposeriez-vous de modifier votre diagramme de définition du monde pour intégrer cette possibilité ?*

Dans le diagramme de base, un invariant spécifie qu'il ne peut y avoir soit qu'un obstacle soit qu'un objet sur une tuile. Il suffit de retirer cet invariant pour permettre à une tuile de contenir simultanément un obstacle et un objet.

- *Quel impact cela a-t-il sur votre modélisation du langage de stratégie, et sur les contraintes OCL définies de part et d'autre ?*

Puisque la contrainte était exprimée par un invariant, il n'y a pas de contrainte OCL à ré-écrire ou à supprimer ou de changement à apporter au DSL.

Chapitre 6

Conclusion

La réalisation de ce laboratoire nous a permis de comprendre la difficulté qu'il peut y avoir à modéliser correctement un phénomène lorsqu'il est spécifié de manière informelle : nous avions commencé ce travail en construisant les objets au fur et à mesure de la spécification, sans analyse préparatoire. Ceci nous a conduit à de nombreuses erreurs et incompatibilités et a rendu nécessaire de revoir le modèle à de nombreuses reprises.

Notre compréhension du sujet a réellement évolué à partir du moment où nous avons réalisé une liste exhaustive et précise de l'ensemble des spécifications ainsi qu'un glossaire. Cela nous a en effet permis de clarifier les concepts, de réduire les ambiguïtés et d'effectuer des recouplements. De cette analyse, nous avons pu dresser la liste des différentes classes nécessaires et commencer la réalisation des diagrammes.

Nous nous sommes ensuite attaqués à la réalisation des associations et avons pu nous confronter à la seconde difficulté qu'entraîne un exercice de modélisation : la rapidité avec laquelle un certain modèle peut prendre forme et sembler complet vis-à-vis de notre sujet. Pour s'assurer que notre modèle est correct, il faut régulièrement le remettre à l'épreuve, que ce soit à la suite de changements dans d'autres sections du diagramme ou simplement pour s'assurer qu'il répond aux conditions énoncées. Nous avons ainsi dû revenir plusieurs fois sur une partie du modèle pour la reformuler entièrement, parce qu'un aspect manquait dans la conceptualisation (importance du *rubber duck debug* ).

En conclusion, nous retiendrons l'importance du systématisme et de la documentation dans la réussite d'une modélisation : découpe claire des concepts, choix des termes et harmonisation.

Deuxième partie
Annexes

Spécifications

1 Description du jeu

1.1 Game

Le **jeu** est la raison principal de ce laboratoire : il s'agit d'une construction complexe permettant à des êtres humains de se divertir en s'opposant de manière ludique.

1.2 Series

La **série** est le cœur même du jeu : c'est au cours de séries que les joueurs peuvent effectivement *jouer* les uns avec les autres.

- Une série possède un identifiant unique..
- Une série est composée d'un ensemble de rencontres.
 - Une série de rencontres comprend au moins une rencontre.
 - Le nombre de rencontres au sein d'une série est impair.
 - Les rencontres sont jouées en ordre successif.
 - Une seule rencontre peut être jouée à la fois.
 - Chaque rencontre ne peut être jouée qu'une seule fois.
- Une série est gagnée par le joueur qui a remporté le plus de rencontres durant la série.

Il se peut que la série ne compte pas de gagnant.

- Une série peut se jouer selon deux modes :
 - DUEL : mode duel (2 joueurs uniquement)
 - MULTIPLAYER : mode multi-joueurs (3+ joueurs)
- Une série est jouée selon deux formules :
 - REALTIME : soit en formule temps-réel (également appelée *interactive*).
 - TURN-BASED : soit en formule tour-par-tour (également appelée *éducative*).
- En formule éducative, les joueurs ne jouent pas directement, mais choisissent des stratégies.
- Pour initier une nouvelle série, les joueurs humains doivent :
 - Choisir le mode de jeu.
 - Choisir la formule de jeu et la limite de temps.Selon le type d'adversaire, une rencontre peut ou ne peut pas être chronométrée.
 - Contre un ordinateur : FREE est autorisé.
 - Contre un ou plusieurs joueurs humains : FREE est interdit.
- S'ils jouent contre un ordinateur, choisir un niveau de difficulté.
- Choisir le nombre de rencontres que comptera la série.
- Choisir l'avatar qui le représentera au cours de cette partie

1.3 Match

La **rencontre** est le composant d'une série au cours de laquelle les joueurs s'affrontent effectivement par le biais de leurs avatars.

- Un rencontre est identifié de manière unique.
- Toute rencontre se déroule sur un plateau propre.
- Une rencontre ne peut pas être rejouée.
- Un rencontre voit se rencontrer les avatars des joueurs.
- Un match peut être libre ou limité :
 - FREE : sans limite
 - TIMER/TURN : soit par chronomètre, soit par compte-tours, selon la formule de jeu choisie.
- Selon le type d'adversaire, une rencontre peut ou ne peut pas être chronométrée.
 - Contre un ordinateur : FREE est autorisé.
 - Contre un ou plusieurs joueurs humains : FREE est interdit.
- La limite d'une rencontre ainsi choisie s'exprime soit :
 - en temps restant.
 - en tours restants.
- Une rencontre peut être remporté de deux manières :
 - LAST-MAN-STANDING : seul un dernier avatar est encore en vie.
 - FOUND-GRAIL : un avatar a trouvé le Graal.

Une rencontre peut ne pas compter de vainqueur, si le temps restant est écoulé/le nombre de tours autorisés dépassés, ou si les deux derniers avatars meurent au même moment.

- Au cours d'une rencontre, les avatars suivent les directives de stratégies choisies pas les joueurs.

1.4 Player

Le **joueur** est un acteur principal dans le jeu : c'est lui qui détermine et participe aux séries de rencontres, donne des instructions aux avatars et remportent les victoires.

- Un joueur possède un nom unique qui l'identifie parmi tous les joueurs.
- Un joueur est soit un être humain, soit un ordinateur.
- Un ordinateur a trois niveaux de difficulté possibles
 - BEGINNER
 - NORMAL
 - EXPERT
- Un joueur possède des apparences, dont au moins une apparence par défaut.
- Un joueur possède au moins un avatar et peut en avoir davantage.
- Un joueur participe à des séries pour jouer.
- Un joueur possède un historique des séries et rencontres jouées et remportées, ainsi que des opposants dans ces parties.
- Au cours d'une rencontre, les joueurs voient :
 - Le plateau de jeu, avec
 - Les limites du plateau.
 - Le brouillard de guerre.

- La tuile sur laquelle leur avatar est positionné.
- Un ensemble de tuiles du plateau dans un périmètre autour de l'avatar, défini par la ligne de mire de celui-ci
- Leur avatar.
- Une barre de menu avec
 - Les caractéristiques de cet avatar.
 - L'inventaire courant de cet avatar.

1.5 Avatar

L'**avatar** est le composant qui s'affronte effectivement sur les plateaux de jeu.

- Un avatar est possédé par un et un seul joueur.
- Un avatar possède un identifiant unique parmi les avatars d'un joueur.
- Un avatar porte une et une seule apparence (qui peut être changée).
- Un avatar se trouve au maximum sur une et une seule tuile.
- L'avatar possède un inventaire.
- Un avatar possède différentes caractéristiques :
 - Un potentiel de vie, exprimé en points.
 - Des niveaux d'attaque et de défense, exprimés en ratios.
 - Une portée de vue
- Un avatar dont les points de vies sont à zéro meurt. Il perd la rencontre en cours et est retiré du plateau.
- Un avatar possède un inventaire rassemblant les objets collectés par l'avatar au cours des différentes rencontres.
- Un avatar peut utiliser différents objets pour s'aider au cours d'une rencontre.
- Un avatar est la cible des attaques des zombies à proximité.
- Un avatar suit les directives d'une stratégie en mode tour-par-tour.
- Un avatar est positionné sur 0 ou 1 tuile (suivant qu'il est utilisé dans une partie ou non).
- Un avatar possède des coordonnées absolues, correspondant à la tuile sur laquelle il se trouve.

La description du jeu ne précise pas explicitement ce qu'il se passe lorsqu'un avatar meure au cours d'une rencontre en milieu de série :

- Est-ce que le joueur doit choisir un remplaçant pour continuer la série ?
- Est-ce que le joueur perd immédiatement la série, sans jouer les rencontres restantes ?
- Est-ce que l'avatar est *ressuscité* avec des caractéristiques par défaut et un inventaire donné ?

1.6 Skin

L'**apparence** est une représentation graphique d'un avatar.

- Une apparence est soit libre, soit payant (*premium*).
- Une apparence est disponible mais pas forcément encore acquise par un des joueurs.
- Une apparence peut être porté par n'importe quel avatar, pour autant que son joueur-propriétaire l'ait acheté préalablement.
- Il existe au moins une apparence non-payante.

1.7 Inventory

L'**inventaire** représente un ensemble d'objets appartenant à un même avatar.

- Au cours des rencontres, les avatars accumulent des objets, qu'ils stockent dans leur inventaire.
- À la fin d'une rencontre et d'une série, certains objets sont conservés, d'autres sont défaussés.
- L'inventaire d'un avatar peut contenir
 - Un stock de munitions/boissons.
 - Des objets lui conférant de nouvelles aptitudes permanentes.
 - Des objets à activer qui confèrent des aptitudes temporaires.

1.8 Board

Le **plateau** de jeu est l'endroit sur lequel s'affrontent les joueurs au cours d'une rencontre.

- Au cours de la rencontre, les avatars des joueurs se déplacent sur et interagissent avec les éléments du plateau.
- Un plateau est de dimension figée et carrée.
- Un plateau est découpé selon une grille bi-dimensionnelle.
- Un plateau est composé de tuiles, en nombre spécifié par ses dimensions.
- Les tuiles du plateau sont posées les unes à côtés des autres de manière ordonné et bidimensionnelle.
- Il n'existe pas de "trou" au sein d'un plateau.
- Un plateau est habité par des personnages qui sont soit les avatars des joueurs, soit des zombies contrôlés par l'ordinateur.
- Un plateau est toujours délimité par une rangée d'obstacles infranchissables sur son contour.
- Au début d'une rencontre, le plateau de jeu est entièrement révélé aux joueurs pour leur permettre d'établir une stratégie. Le plateau est ensuite recouvert d'un brouillard de guerre, permettant aux avatars de voir les tuiles dans un périmètre défini par leur ligne de mire.

1.9 Tile

La **tuile** est le composant principal, atomique, du plateau de jeu.

- Une tuile possède des coordonnées absolues qui renseignent sa position sur le plateau.
- Une tuile peut être positionnée à côté de 0 à 4 tuiles adjacentes, selon les quatre directions cardinales.
- Une tuile peut de manière facultative contenir différents éléments :
 - Un personnage (avatar ou zombie).
 - Un obstacle.
 - Un objet.
- Une tuile ne peut avoir au maximum qu'un seul personnage présent à la fois.
- Une tuile ne peut avoir au maximum qu'un seul obstacle présent à la fois.
- Une tuile ne peut avoir au maximum qu'un seul objet présent à la fois.
- Une tuile ne peut contenir simultanément un obstacle et un objet.

1.10 Zombie

Le **zombie** est le personnage antagoniste aux avatars.

- Un zombie se trouve nécessairement sur une tuile sur un plateau donné.
- Un zombie erre dans le monde sans but, jusqu'à apercevoir un avatar.
- La distance à partir de laquelle un zombie aperçoit un avatar est fonction de son ratio d'attaque.
- Lorsqu'il aperçoit un avatar, le zombie se lance à sa poursuite : il essaye de s'en rapprocher, puis d'effectuer une attaque au corps-à-corps.
- Un zombie effectue une attaque au corps-à-corps lorsqu'il frappe une tuile adjacente dans la direction observée par le zombie.
- Un zombie n'attaque pas les autres zombies.
- Un zombie ne collecte pas les objets présents sur le plateau.
- Un zombie est présent sur une et une seule tuile.
- Un zombie a, comme les avatars, un certain nombre de caractéristiques.
 - Un potentiel de vie, exprimé en points
 - Un niveau d'attaque, exprimé sous forme de ratio.

Un zombie n'a pas de niveau de défense, contrairement aux avatars.

- Un zombie dont le potentiel de vie est à zéro meurt et disparaît du plateau.
- Un zombie observe une direction parmi les directions cardinales.
- Un zombie adapte sa direction à la suite de chaque mouvement : elle prend la valeur de la direction du mouvement effectué.

1.11 Obstacle

Les **obstacles** sont des éléments de décor qui opposent une résistance aux personnages.

- Un obstacle se trouve nécessairement sur une tuile sur un plateau donné.
- Un obstacle est parfois franchissable, parfois destructible.
 - Un obstacle est *franchissable* si un avatar peut le franchir sous certaines conditions :
 - par la possession d'un objet spécifique.
 - au prix d'un nombre défini de points de vie.
 - Un obstacle est *destructible* si un avatar peut le détruire au moyen d'une attaque directe ou indirecte.
 - Un obstacle qui est détruit disparaît de la case et du plateau de jeu.
- Les obstacles destructibles sont :
 - THORN : les ronces
 - BUSH : les buissons
 - TRUNK : les troncs d'arbre
 - SKULL : les têtes de mort
- Les obstacles indestructibles sont répartis en deux catégories :
 - Les obstacles permanents.
 - WALL : mur
 - TREE : arbre
 - ROCK : menhir
 - Les obstacles de zone.
 - FIRE : zone de feu

- ICE : zone de glace
- WATER : zone d'eau
- Les conditions pour qu'un avatar puisse franchir un obstacle sont les suivantes :
 - l'avatar possède la cape de mage et
 - L'obstacle est une zone.
 - L'obstacle est un obstacle permanent ou destructible et la cape de mage possède des points d'utilisation restants.
 - L'obstacle est une *zone de feu* et
 - l'avatar possède une cape de mage et/ou des bottes de feu.
 - autrement, l'avatar perd un nombre défini de points de vie.
Le nombre de points de vie perdu est fonction de la durée de séjour sur la zone de feu : 1 point par tour ou un nombre indéfini par minute.
 - L'obstacle est une *zone d'eau* et l'avatar possède la cape de mage et/ou un kit de plongée
 - L'obstacle est une *zone de glace* et l'avatar possède la cape de mage et/ou des bottes à crampons.

1.12 Item

Les **objets** sont des éléments disséminés sur le plateau et pouvant être collectés par les joueurs pour leur conférer un avantage/une aptitude temporaire ou permanent.

- Il existe plusieurs types d'objets :
 - GRAIL : Graal.
 - CONSUMABLE : consommables.
 - BOOSTER : améliorants.
 - CAPACITER : capaciteurs.
 - BUFF : super-pouvoirs.
 - POINTER : pointeurs.
- Le *Graal* est l'un des deux objectifs pour remporter une rencontre.
 - Il existe nécessairement un et un seul Graal sur le plateau.
 - Il est activé automatiquement lorsque ramassé.
 - Il ne rentre jamais dans l'inventaire.
- Les *consommables* permettent au joueur de récupérer des points de vie ou de lancer des projectiles.
 - Les consommables sont de trois types :
 - FOOD : Nourriture
 - DRINK : Boisson
 - AMMO : Munitions
 - La nourriture et les boissons permettent de restaurer les points de vie de l'avatar.
 - Les munitions permettent à l'avatar de lancer un projectile.
 - Un consommable doit être activé pour réaliser son objet.
 - Lorsqu'un consommable est activé, il est défaussé de l'inventaire.
 - Les consommables non-utilisés sont permanents : ils restent dans l'inventaire de l'avatar après la fin de la rencontre et de la série.
- Les *améliorants* augmentent de manière permanente les caractéristiques de l'avatar qui les collecte.
 - Les améliorants sont de deux types :
 - ATTACK : Boosteur d'attaque

- DEFENSE : Boosteur de défense
- Les améliorants sont automatiquement activés lorsque ramassés par l'avatar.
- Les améliorants sont immédiatement défaussés après activation.
- Les *capaciteurs* confèrent une nouvelle aptitude permanente à l'avatar qui les collecte.
 - Les capaciteurs sont de trois types :
 - FIREBOOT : bottes de feu.
 - ICEBOOT : bottes à crampons.
 - DIVEMASK : kit de plongée.
 - Les bottes de feu permettent de franchir une zone de feu sans perdre de points de vie.
 - Les bottes à crampons permettent de franchir une zone de glace.
 - Le kit de plongée permet de franchir une zone d'eau.
 - Les capaciteurs sont automatiquement activés lorsque ramassés par l'avatar.
 - Les capaciteurs sont permanents : ils restent dans l'inventaire de l'avatar après la fin de la rencontre et de la série.
- Les *super-pouvoirs* donnent à l'avatar une capacité temporaire puissante.
 - Les super-pouvoirs sont de deux types :
 - ARMOR : cotte de mailles
 - CAPE : cape de mage
 - La cotte de mailles rend l'avatar qui la porte insensibles aux attaques des autres avatars et zombies.
 - La cape de mage permet à l'avatar qui la porte de traverser zones et obstacles sans dommages.
 - La cape de mage ne permet de traverser des obstacles infranchissables que trois fois. Un compteur sous forme de points d'utilisation permet de mesurer combien d'utilisations restantes subsistent.
 - Un super-pouvoir doit être activé pour réaliser son objet.
 - Un super-pouvoir n'est actif que pour une durée limitée, exprimée en termes de temps ou nombre de tours restants, selon la formule de jeu retenue :
 - TURN-BASED : la durée correspond à [$\frac{1}{10}$ e des points de vie de l'avatar].
 - REALTIME : la durée correspond à 1 minute.
 - Lorsqu'un super-pouvoir est activé, un compteur est enclenché et réduit progressivement la durée restante d'activité du super-pouvoir.
 - Lorsqu'un super-pouvoir n'a plus de durée d'activité restante, il est défaussé de l'inventaire et l'avatar perd son pouvoir.
 - Les super-pouvoirs non-utilisés sont permanents : ils restent dans l'inventaire de l'avatar après la fin de la rencontre et de la série.
- Les *pointeurs* aident l'avatar (et le joueur) à s'orienter vers le plateau, en indiquant la direction à vol d'oiseau vers le(s) autre(s) avatar(s) ou le Graal.
 - Les pointeurs sont de deux types :
 - MAP : carte.
 - RADAR : radar.
 - La carte indique la direction à vol d'oiseau vers le Graal du plateau.
 - La carte n'a pas de durée limitée.
 - La carte est automatiquement activée lorsqu'elle est ramassée.
 - La carte n'est pas permanente : elle est défaussée à la fin de la rencontre.
 - Le radar indique la direction à vol d'oiseau vers le/les adversaire(s).
 - Le radar doit être activé par l'avatar pour fonctionner.

- Le radar a une durée limitée, exprimée en terme de temps/tours restants.
- A chaque fois qu'un nouveau radar est ramassé,
 - si l'avatar ne possède pas encore de radar, il est simplement ajouté à l'inventaire.
 - si le joueur possède déjà un radar, celui-ci voit son temps restant d'utilisation doublé (qu'il soit actif ou non). Le nouveau radar est immédiatement défaussé.
- Les radars non-utilisés sont permanents : ils restent dans l'inventaire de l'avatar après la fin de la rencontre et de la série.
- Un objet peut être ramassé par l'avatar présent sur la même tuile.
- Un objet ramassé disparaît du plateau et apparaît dans l'inventaire du joueur selon les conditions décrites ci-dessus.

2 Description des stratégies

2.1 Strategy

Les **stratégies** sont des mécanismes de jeu automatique permettant au joueur de faire des rencontres en tour-par-tour.

- Une stratégie permet aux avatars d'effectuer des actions et des déplacements sans input direct des joueurs.
- Une stratégie est composée :
 - d'objectifs ;
 - de variables globales ;
 - de modules.
- Une stratégie comporte toujours deux objectifs :
 - Un objectif de long-terme, à poursuivre en toutes circonstances.
 - Un objectif de court-terme, à réaliser de manière opportuniste.
- L'ordre de réalisation de ces objectifs est implicite :
 - L'objectif de court-terme prime sur l'objectif de long-terme.
 - Un objectif est réalisable tant qu'une de ses directives est activable.
 - Un objectif néant est nécessairement précédé par l'autre objectif.
 - Il ne peut pas y avoir deux objectifs néant au sein d'une stratégie.
- Tous les modules au sein d'une stratégie sont nommés différemment.
- Toutes les variables globales au sein d'une stratégie sont nommées différemment.

2.2 Objective

Les **objectifs** permettent de donner une direction et une liste d'actions possibles à des avatars pour remporter une rencontre.

- Un objectif est caractérisé par une cible :
 - **Néant (Random)** - déplacements aléatoires.
 - **AllerVers (Goto)** - déplacements en direction d'une tuile.
 - **Contourner (Avoid)** - déplacement en direction d'une tuile avec contournement.
 - **Éviter (Evade)** - déplacement d'éloignement d'un personnage ou d'une tuile.
 - **CollecterMax (Collect)** - priorisation du ramassage d'un certain type d'objet.
 - **Combattre (Attack)** - attaque de personnages avoisinants.
- Les objectifs *Néant* et *Combattre* n'acceptent aucun paramètre.
- L'objectif *AllerVers* prend en paramètre une tuile de destination.

- L'objectif *Contourner* prend en paramètre une tuile de destination, ainsi qu'un ensemble de consignes d'évitements constituées d'un couple
 - élément à éviter et
 - direction à emprunter pour l'évitement.
- L'objectif *Éviter* prend en paramètre un personnage ou une tuile dont s'éloigner.
- L'objectif *CollecterMax* prend en paramètre un type d'objet à rechercher et ramasser.
- Un objectif peut uniquement être changé au moyen d'une action Changer.
- Un objectif définit un ensemble de directives.
- Un objectif a nécessairement une directive par défaut. La priorité de cette directive a la valeur zéro.
- Lorsque plusieurs directives sont activées simultanément, seule celle dont la valeur de priorité est la plus proche de 0 est appliquée.
- Les priorités des directives d'un objectif sont toutes différentes.
- La directive par défaut n'a pas de pré-requis d'exécution.

2.3 Rule

Les **directives** décrivent les actions à entreprendre pour la réalisation d'un objectif et les conditions dans lesquelles elles s'appliquent.

- Il existe deux catégories de directives :
 - les directives normales.
 - les métadirectives.
- Une directive est composée de trois éléments
 - une priorité explicite (*priority*).
 - un déclencheur (*trigger*)
 - une réaction (*reaction*)
- La priorité indique le niveau de préséance d'une directive parmi les directives d'un objectif.
 - La priorité est un entier positif.
 - Le niveau de priorité d'une directive est inversement proportionnel à sa valeur.
 - La règle par défaut d'un objectif a la priorité zéro.
- Le déclencheur est une expression booléenne qui, si elle est évaluée à vraie, signifie que la directive peut être appliquée.
- La réaction est un ensemble d'instructions élémentaires et de paramètres permettant la réalisation de la directive.
 - Une réaction est composée de :
 - des variables ;
 - une séquence d'instructions.
 - Dans la séquence d'instructions, il peut y avoir plusieurs actions.
 - Dans la séquence d'instructions, il ne peut y avoir qu'une seul action de type *mouvement*.
 - Seule une métadirective peut inclure une action de type *Changer*.

2.4 Action

Les **actions** décrivent les interactions qu'un joueur peut entretenir avec le jeu et son avatar.

- Il existe différentes catégories d'action :
 - **Mouvement** : action permettant au joueur de déplacer son personnage.
 - **Activité** : action permettant au joueur d'interagir avec l'environnement direct du personnage.

- **Méta** : action permettant au joueur d'influer sur le comportement de son personnage.
- L'action de mouvement est *SeDéplacer* et consiste à modifier l'emplacement du personnage vers une tuile adjacente (en vérifiant les conditions de franchissement).
- Les actions de type *activité* sont de cinq types :
 - **Frapper** (*slash*)
 - **Tirer** (*fire*)
 - **Revêtir** (*wear*)
 - **UtiliserItem** (*use*)
 - **ConsulterRadar** (*read*)
- L'action de frapper consiste

2.5 Module

Les **modules** sont les éléments déclaratifs de la programmation et renseignent sur les calculs pouvant être réalisés et leur conditions d'exécution.

- Un module est composé
 - de variables locales ;
 - de paramètres optionnels en entrée ;
 - d'un retour facultatif ;
 - d'un corps.
- Un module possède un nom.
- Un module peut utiliser les variables globales de la stratégie à laquelle il appartient.
- Toutes les paramètres au sein d'un module sont nommées différemment.
- Toutes les variables au sein d'un module sont nommées différemment.
- Une variable locale ne peut avoir le même nom qu'une variable globale définie par la stratégie à laquelle le module appartient.

2.6 Types

Les **types** sont les éléments informatifs de la programmation et renseignent sur le format des données manipulées.

- Il existe des types "*finis*", prédéfinis et figés.
- Il existe des types "*extensibles*", qui peuvent être ajoutés selon les besoins.
- Il existe différentes natures, ou formats :
 - Les types primitifs
 - Les énumérations
 - Les tableaux
 - Les enregistrements
- Les types *primitifs* sont au nombre de 4 : string, integer, float, boolean.
- Les *énumérations* établissent un ensemble fini de valeurs auto-expressives.
 - Une énumération établit au moins une valeur.
 - Les valeurs établies par une énumération porte un nom.
 - Tous les noms de valeur d'une énumération doivent être différents.
- Les *tableaux* sont des séquences de valeurs, de longueur finie et organisées de manière multi-dimensionnelle.
 - Un tableau possède un type (qui peut être un autre tableau).
 - Un tableau définit une séquence de valeurs.

- Toutes les valeurs d'un tableau ont le même type.
- Un tableau possède au moins une dimension.
- La dimension d'un tableau doit être strictement positive.
- Les *enregistrements* sont des ensembles non-ordonnés et finis de valeurs de type variable.
 - Un enregistrement déclare un ensemble de champs avec un certain type et un nom.
 - Un enregistrement doit avoir au moins un champ.
 - Les champs d'un enregistrement peuvent être de différents types.
 - Les champs déclarés au sein d'un enregistrement possèdent un nom unique entre eux.
- Les types possèdent un nom unique permettant de les identifier.

2.7 Expression

Les **expressions** sont l'élément de représentation textuelle des données dans le système.

- Un expression peut être de six types :
 - **Littéral** - un objet auto-exprimé.
 - **Symbol** - un élément de la mémoire pouvant être assigné (LHS) ou récupéré (RHS).
 - **Unaire** - Un opérateur unaire lié à un expression.
 - **Binaire** - Un opérateur binaire lié à deux expressions.
 - **Parenthésée** - Un groupe d'expressions
 - **Appel de module** - un appel à un module prenant (ou non) des paramètres et produisant (ou non) un retour.
- Un *littéral* est une expression comprenant uniquement un élément auto-suffisant, c'est-à-dire un élément de type primitif ou une énumération.
- Un *symbol* est une expression référençant un objet en mémoire pouvant être plus ou moins complexe.
- Une expression *unaire* est une expression composée d'un opérateur unaire et d'un opérande. Cet opérande est lui-même une expression.
- Une expression *binaire* est une expression composée d'un opérateur binaire et de deux opérandes : un à gauche et un à droite. Ces opérandes sont eux-mêmes des expressions.
- Une expression *parenthésée* est une expression contenant une sous-expression entre deux marqueurs.

2.8 Instruction

Les **instructions** sont l'élément effectif de programmation qui permet de manipuler effectivement les données.

- Une instruction peut être de cinq différents types :
 - **Saut (Skip)** - une instruction sans effet.
 - **Conditionnelle (Selection)** - une instruction de branchement, composée d'une garde et de 1 ou 2 branches (*if...then...else...*).
 - **Itération (Iteration)** - une instruction de répétition, composée d'une garde de sortie et d'un corps d'instruction.
 - **Séquence (sequence)** - une instruction composée d'une suite d'instructions.
 - **Affectation (Assignment)** - une instruction composée d'une partie gauche et d'une partie droite et consistant à assigner la valeur de la partie droite dans la partie gauche.
 - **Action (Action)** - une instruction réalisant une action.
- Elle permet
 - de manipuler les données et
 - d'effectuer des opérations : mouvements, interactions.

Glossaire

Action Une instruction permettant à un joueur d'agir sur son personnage.

Ammo Un objet collectable et consommable qui augmente le nombre de projectiles qu'un personnage peut envoyer.

synonymes : (recharge de) munition(s), *ammunition*.

Area Un type d'obstacle ne pouvant pas être détruit, mais qui est franchissable soit au prix de points de vie, soit par la possession d'un capaciteur.

synonymes : Zone.

Armor Un objet qui confère à son détenteur une immunité temporaire contre toutes les armes du jeu (projectile et épée).

synonymes : cotte de maille, armure, protection, (pouvoir d') invincibilité.

Array Un type de données composé d'une suite ordonnée de données de même type.

synonymes : tableau, table, séquence/suite de données.

Assignment Une instruction permettant d'associer une donnée à une variable.

synonymes : affectation, assignation.

Attack Ratio (AR) Le niveau d'attaque indique le potentiel de dégâts causés à la cible d'une attaque par l'agresseur.

synonymes : (Ratio d') attaque, (ratio de) force, (Ratio de) puissance.

Attacker Un personnage est dit attaquant s'il donne un coup d'épée.

synonymes : attaquant.

Avatar Un personnage contrôlé par un joueur sur un plateau de jeu.

synonymes : personnage, character.

Avoid Action de se déplacer en direction d'une tuile en évitant un obstacle/un personnage.

synonymes : contourner, éviter.

Binary Expression Une expression composée d'un opérateur et de deux membres.

synonymes : expression binaire.

Binary Operator Un connecteur acceptant deux expressions (une à droite et une à gauche) et permettant de combiner deux données. Forme une expression binaire.

synonymes : opérateur binaire.

Bite Un zombie mord un personnage lorsqu'il effectue une frappe à courte distance sur ce personnage. Enlève un nombre de points de vie au personnage proportionnel au ratio d'attaque du zombie et de défense du personnage.

synonymes : mordre, attaquer.

Block Un type d'obstacle ne pouvant pas être détruit et franchissable uniquement si en possession d'une cape de mage avec des points d'utilisation restants. Traverser l'obstacle *Block* au moyen de la cape réduit son potentiel d'utilisation de 1.

Board L'endroit sur lequel les joueurs voient s'affronter leurs avatars au cours d'une rencontre. Est composé d'un nombre défini et carré de tuiles et a son pourtour entièrement composé d'obstacles *Block*.

synonymes : plateau, monde, terrain, carte, *World, Map*.

Body Se dit d'une séquence d'instructions associée à un module et décrivant une suite d'opérations à mener.

synonymes : corps.

Booster Un type d'objet qui, une fois ramassé, améliore de façon permanent les caractéristiques du personnage.

synonymes : amélioration, objet améliorant, bonus.

Booster AP Un collectable qui augmente de manière permanente le ratio d'attaque.

synonymes : bonus d'attaque, amélioration d'attaque.

Booster DP Un collectable qui augmente de manière permanente le ratio de défense.

synonymes : bonus d'attaque, amélioration de défense.

Buff Un type d'objet conférant à son détenteur une capacité spéciale temporaire puissante. Sa limite s'exprime en termes de temps ou de tours restants en fonction du mode de jeu, à compter du moment où son détenteur l'active.

* Son nombre de tours restants initial est défini comme 10% des points de vie du joueur.

* Son temps restant initial est défini comme 1 minute.

synonymes : super-objet, super-pouvoir.

Call Une expression dénotant de l'appel à un module et à l'exécution d'instructions. déclare des arguments en entrée et le nom du module à appeler.

synonymes : appel.

Capaciter Un type d'objet conférant à son détenteur une capacité spéciale permanente.

synonymes : capaciteur.

Cape Un objet qui confère à son détenteur la capacité de franchir tous les obstacles de zone (feu, glace, eau) sans l'objet capaciteur correspondant, et tout autre obstacle 3 fois.

synonymes : cape (de mage), manteau.

Change Action de mettre à jour la stratégie suivie en adaptant les objectifs primaires et secondaires.

synonymes : changer, adapter.

Close-Range Situation de combat durant laquelle les personnages impliqués sont l'un à côté de l'autre.

synonymes : courte portée, combat rapproché, corps-à-corps.

Collect Action de prendre un objet présent sur une tuile : l'objet disparaît alors de la tuile et applique l'effet propre à son type.

synonymes : collecter, Ramasser.

Combat Situation où au moins un personnage tente de toucher un second au moyen de coups d'épée ou de lancers de projectile.

synonymes : affrontement, échauffourée, escarmouche, *skirmish*.

Computer Un joueur fictif contrôlé par l'ordinateur.

synonymes : ordinateur, bot, *Artificial Intelligence (AI)*, intelligence artificielle (IA).

Consumable Se dit d'un objet à usage unique, qui est défaussé après utilisation.

synonymes : consommable, jetable, à usage unique, *one-time*.

Crossable Se dit d'une tuile qui peut être traversée par un avatar.

synonymes : franchissable, traversable.

Crosser Se dit d'un avatar qui traverse une tuile.

synonymes : franchisseur.

Damage Les dommages sont un nombre calculés dans le cas d'une attaque, en fonction des ratios d'attaque de l'agresseur et de défense de l'agressé, et qui vont diminuer le potentiel de vie de l'agressé. Si le nombre de points de vie de l'agressé est inférieur à ce nombre, alors ce personnage meurt et disparaît du plateau.

synonymes : dommages.

Defender Un personnage est dit défendant s'il reçoit un coup d'épée.

synonymes : défenseur, Défendant, attaqué, victime, cible.

Defense Ratio (DR) Le ratio de défense indique le potentiel de dégâts que l'agressé reçoit de la part de l'agresseur.

synonymes : (Ratio de) défense, (Ratio de) résistance.

Destructible Un obstacle est destructible si le joueur peut le faire disparaître à coup d'épée. La tuile qu'il occupe est alors rendue vide et franchissable.

Direction Sens vers lequel le personnage est orienté et peut frapper avec son épée ou tirer un projectile.

synonymes : direction, orientation.

Direction: Down Direction sud (X+1).

synonymes : bas, sud, south.

Direction: Left Direction ouest (X-1).

synonymes : Gauche, ouest, West.

Direction: Right Direction est (X+1).

synonymes : droite, est, east.

Direction: Up Direction nord (Y-1).

synonymes : haut, supérieur, nord, North.

Divemask Un objet collectable qui permet à son détenteur de franchir une zone d'eau.

synonymes : masque/kit de plongée.

DP-Booster Un collectable qui augmente de manière permanente le ratio de défense.

synonymes : bonus de défense, amélioration de défense.

Drink Un objet collectable et consommable, qui augmente les points de vie (HP) du joueur.

synonymes : boisson, eau.

Evade Action d'éviter (les coups d')un adversaire et de fuir le combat.

synonymes : éviter, esquiver.

Expression Une expression est un élément de programmation des stratégies servant à exprimer une valeur ou une combinaison de valeurs.

Fire Un obstacle franchissable en toute circonstances, mais qui inflige des dommages si le franchisseur ne possède pas des bottes de feu.

synonymes : (Zone de) feu.

Fireboot Un objet collectable qui permet à son détenteur de franchir une zone de feu sans souffrir de dommages.

synonymes : bottes de feu, bottes pare-feu.

Food Un objet consommable, qui augmente les points de vie (HP) du joueur.

synonymes : nourriture.

Free Limit Un mode de jeu pour des parties qui se déroulent sans limite de temps ou de tour.

synonymes : temps libre.

Game Le jeu décrit dans ce laboratoire.

synonymes : Jeu, World Game.

Grail Un objet qui, une fois ramassé, déclenche la fin de partie et donne le joueur courant comme gagnant. Il ne rentre jamais dans l'inventaire.

synonymes : Graal.

Group Une expression composée d'une, ou plusieurs, sous-expressions.

synonymes : parenthèse, expression parenthésée, groupe, expressions groupées.

Health Points (HP) Les points de vie indique le niveau restant de vie du défenseur et le nombre de dommages qu'il peut encore supporter avant de mourir et disparaître du plateau.

synonymes : points de vie, potentiel de vie, indicateur de vie.

Hit Action d'utiliser son épée pour frapper - soit un personnage ou un monstre adjacent, - soit un obstacle destructible. L'élément frappé doit se trouver dans la direction du regard du joueur.

synonymes : frapper, slash.

Human Un joueur (réel) incarné par un être humain.

synonymes : Humain.

Ice Un obstacle uniquement franchissable si le franchisseur possède les bottes de glace.
synonymes : (Zone de) glace.

Iceboot Un objet qui permet à son détenteur de franchir une zone de glace.
synonymes : botte de glace, bottes à crampons.

Instruction Un fragment de programme permettant de manipuler des données et d'influer sur le jeu.

Inventory Ensemble des objets ramassés par le personnage au cours des différentes parties jouées.
synonymes : inventaire, sac, barda.

Item Un élément pouvant être ramassé par un avatar.
synonymes : Objet.

Iteration Une instruction composée d'une expression de garde et d'un corps d'instructions imbriquées qui s'exécutent tant que la garde n'est pas activée.
synonymes : itération, boucle.

Left Hand Side (LHS) Se dit d'une expression située à gauche du symbole d'affectation.
synonymes : expression gauche.

Level Le niveau d'expertise de l'ordinateur définit sa capacité à jouer efficacement face aux autres joueurs..
synonymes : niveau, difficulté.

Level Beginner Un niveau associé à un joueur contrôlé par l'ordinateur pendant un tournoi. Est le niveau le plus faible permettant à un joueur humain de facilement gagner la rencontre.
synonymes : débutant, facile.

Level Expert Le niveau le plus fort. Offre au joueur un challenge relevé.
synonymes : avancé, Difficile.

Level Normal Le niveau intermédiaire. Offre au joueur un challenge modéré.

Literal Une expression strictement équivalente à une valeur primitive ou à une valeur d'énumération.
synonymes : littéral.

Long-Range Situation de combat durant laquelle les personnages impliqués ne sont pas à proximité.
synonymes : longue portée, combat distant, combat à distance.

Map Un objet qui, une fois collecté, indique de manière permanente la direction absolue vers le Graal sur le plateau. Est retiré de l'inventaire du personnage à la fin de la rencontre
synonymes : carte, plan.

Match Une composante des séries permettant à deux joueurs (ou plus) de faire affronter leurs avatars sur un plateau de jeu.
synonymes : rencontre, partie, affrontement.

Meal Un objet comestible qu'un avatar peut manger pour restaurer des points de vie.
synonymes : repas.

Module Élément de programmation réutilisable, acceptant une série de paramètres (optionnels) et produisant un résultat (optionnel). Possède un nom unique.
synonymes : procédure, fonction, méthode.

Move Action de se déplacer sur le plateau de jeu : le personnage se déplace de la tuile courante vers une tuile adjacente. Cette action est contrainte par l'accessibilité de la tuile avoisinante : il n'est pas possible de se déplacer sur une tuile infranchissable. Doit suivre l'une des quatre directions autorisées.
synonymes : bouger, se mouvoir, se déplacer.

Objective Entité donnant une orientation au comportement automatique d'un personnage en visant une situation donnée. Peut être de court ou de long terme.
synonymes : objectif, vision.

Objective: Long-Term (STO) Objectif constant guidant l'action d'un personnage tout au long d'une rencontre.

synonymes : objectif de long-terme/primaire, stratégie, *primary, strategy*.

Objective: Short-Term (STO) Objectif impromptu, défini par les circonstances.

synonymes : objectif de court-terme/secondaire, tactique, *secondary, tactics*.

Observee Entité qui est visible aux yeux d'un personnage alentours.

synonymes : observé.

Observer Personnage en mesure d'apercevoir une entité présente sur une tuile adjacente.

synonymes : observateur.

Obstacle Élément de décor se trouvant sur une tuile et pouvant bloquer ou nuire aux personnages.

Output Valeur retournée par un module en fin d'exécution. Doit correspondre au type déclaré et répondre aux postconditions de la spécification du module.

synonymes : résultat, retour, sortie, sortant, produit, production, *output, extrant*.

Parameter Élément déclaré par un module et devant être fourni par un code appelant pour permettre au module de fonctionner. Doit respecter les spécifications du module.

synonymes : paramètre, *input, entrant, intrant, argument*.

Player Acteur non nécessairement humain participant à une série et contrôlant un avatar dans un match.

synonymes : joueur.

Pointer Un type d'objet permettant à son détenteur de s'orienter plus facilement sur le plateau de jeu.

synonymes : (aide à l')orientation, pointeur.

Position Les coordonnées exactes de l'emplacement d'un personnage ou d'une tuile, par rapport aux axes horizontal et vertical.

synonymes : emplacement, position absolue, coordonnées.

Premium Un skin premium est un skin disponible uniquement aux joueurs l'ayant acheté.

synonymes : payant, exclusif.

Priority Élément d'une directive permettant de déterminer, lors d'un conflit entre plusieurs directives, celle à appliquer en premier.

synonymes : priorité.

Procedure *synonymes* : procédure, méthode, processus.

Projectile Élément que le joueur peut envoyer dans une direction. Suit ensuite la trajectoire définie jusqu'à rencontrer un autre joueur, un monstre ou un obstacle.

* si un joueur ou un monstre est percuté par cet élément, il permet une partie de ses points de vie (partie qui est fonction du ratio de défense).

* si un obstacle destructible est percuté par cet élément, il est détruit et supprimé du plateau.
synonymes : balle, lance, flèche.

Pseudo Un nom unique identifiant un joueur du jeu.

synonymes : nom, identifiant, *username*.

Radar Un objet qui, une fois collecté, indique de manière temporaire la direction absolue vers les autres personnages du plateau. Sa limite s'exprime en termes de temps ou de tours restants en fonction du mode de jeu, à compter du moment où son détenteur l'active.

Reaction Composante d'une directive constituée d'une suite d'instructions, qui se déclenche lorsque le déclencheur de la directive est activé.

synonymes : réaction, réponse.

Read Action de déclencher une carte ou un radar présent dans l'inventaire. Déclenche le décompte d'utilisation du radar. Si épuisé, le radar disparaît de l'inventaire.

synonymes : lire, activer.

Real Time Un mode de jeu où les personnages se déplacent et effectuent des actions simultanément.

synonymes : mode interactif, temps réel.

- Record** Un type de donnée complexe, constitué d'un ensemble d'autres données.
synonymes : enregistrement.
- Right-Hand Side (RHS)** Se dit d'une expression située à gauche du symbole d'affectation.
synonymes : expression droite.
- Rock** Un objet indestructible.
synonymes : menhir, Rocher, caillou.
- Rule** Composante d'un objectif, dotée d'un déclencheur et d'une réaction. Prescrit un comportement face à une situation donnée.
synonymes : directive, règle.
- Selection** Une instruction permettant d'exécuter de manière conditionnelle du code.
synonymes : conditionnelle, branchement.
- Sequence** Une instruction composée d'une suite ordonnée d'instructions.
synonymes : séquence, suite d'instructions.
- Series** L'ensemble des rencontres disputées entre des joueurs.
synonymes : tournoi, série de rencontres.
- Series ID** Un numéro unique identifiant la série.
synonymes : numéro de série, tag, identifiant de série.
- Shoot** Action d'utiliser un objet consommable de type *Ammo* de l'inventaire pour envoyer un projectile dans une direction. Le nombre de munitions est alors diminué de 1.
synonymes : tirer.
- Shooter** Un personnage est dit tireur lorsqu'il effectue un tir au moyen d'une munition.
synonymes : tireur, archer, lanceur.
- Sight** La portée de vue exprime le rayon visible autour du joueur : tuiles avoisinantes et objets/-monstres/joueurs présents sur ces tuiles.
synonymes : vision, portée de vue.
- Skin** L'image représentant l'avatar du joueur.
synonymes : figurine, apparence.
- Skip** Une action qui consiste à passer, ne rien faire.
synonymes : pause, Rien, nothing, idle, passer.
- Skull** Un objet destructible.
synonymes : crâne, tête de mort, squelette, os, ossements, *bones*.
- Slash** Action de donner un coup d'épée devant soi (frappe à courte portée).
synonymes : frapper, trancher, estoquer.
- Strategy** Méthode d'appréhension d'une rencontre permettant à un avatar de choisir automatiquement les actions à effectuer en fonction de son contexte environnant et dans la poursuite d'objectifs.
synonymes : stratégie, approche.
- Symbol** Un symbole est une expression permettant de désigner une donnée stockée en mémoire.
synonymes : symbole, référence, *reference*.
- Target** Un personnage est appelé cible lorsqu'il reçoit un tir de munition.
synonymes : cible.
- Thorn** Un objet destructible.
synonymes : Ronces.
- Tile** Un élément constitutif du plateau de jeu, sur lequel vont se trouver les éléments de jeu.
synonymes : tuile, case, cellule, emplacement, tuile, *ground*, *place*.
- Time Limit** Un mode de jeu qui délimite la durée d'une partie en fonction d'un chronomètre défini.
synonymes : limite de temps, temps limité.

Trash Un type d'obstacle pouvant être détruit par un coup d'épée et franchissable uniquement si en possession d'une cape de mage avec des points d'utilisation restants.
synonymes : débris.

Tree Un objet indestructible.
synonymes : arbre.

Trigger Une expression de garde d'une directive, dont l'évaluation positive déclenche l'exécution de la réaction associée à cette directive.
synonymes : déclencheur, condition.

Trunk Un objet destructible.
synonymes : tronc.

Turn Limit Un mode de jeu pour des parties qui ont une limite exprimée en termes de tours restants.
synonymes : limite de tours, tours limités.

Turn-Based Un mode de jeu où les personnages jouent au tour-par-tour en suivant des stratégies. Les personnages effectuent alors des actions de manière automatique, en suivant les directives des stratégies.
synonymes : mode éducatif.

Type Une valeur dénotant de la qualité intrinsèque d'une donnée.

Unary Un connecteur acceptant à sa droite une seule expression et appliquant une opération de conversion (négation, incrément,...). Forme une expression unaire.
synonymes : unaire.

Use Action d'utiliser un objet consommable de type *Food* ou *Drink* présent dans l'inventaire. L'objet consommé effectue l'action de son type, puis décompté.
synonymes : utiliser, activer, employer.

Use Left Un indicateur du nombre d'utilisation possibles restantes pour un objet donné dans l'inventaire. Une fois que ce nombre atteint zéro, l'objet ne peut plus être utilisé.
synonymes : point d'utilisation.

Variable Une entité nommée, associée à un type de données et à laquelle on va pouvoir affecter une valeur.

Variable: Global Se dit d'une variable déclarée par une stratégie et qui peut être utilisée par un module déclaré sous cette même stratégie.
synonymes : variable globale.

Wall Un objet indestructible, servant notamment à démarquer la frontière du plateau.
synonymes : mur.

Water Un obstacle uniquement franchissable si le franchisseur possède le masque de plongée.
synonymes : (Zone d')eau, bassin.

Wear Action d'utiliser un objet de type *Buff* présent dans l'inventaire.
Déclenche le décompte d'utilisation. Une fois le décompte achevé, l'objet est décompté de l'inventaire et le personnage retiré.
synonymes : porter, enfiler, activer.

Zombie Un personnage non-jouable, présent sur le plateau de jeu et cherchant à attaquer et tuer les joueurs indistinctement. Il ne possède ni inventaire ni points de défense.
synonymes : zombie, mob, ennemi, monstre.

Bibliographie

- [1] B. LISKOV et Guttag J. *Program Development in Java : Abstraction, Specification, and Object-Oriented Design*. 6^e éd. Addison-Wesley, 2000.
- [2] R.C. MARTIN et al. *Agile Software Development : Principles, Patterns, and Practices*. Alan Apt series. Pearson Education, 2003. ISBN : 9780135974445.
- [3] OBJECT MANAGEMENT GROUP (OMG). *Object Constraint Language (OCL)*. 2.4. Accessed : 2020-12-09. Fév. 2014. URL : <https://www.omg.org/spec/OCL/2.4>.
- [4] OBJECT MANAGEMENT GROUP (OMG). *Unified Modeling Language (UML)*. 2.5.1. Accessed : 2020-12-09. Déc. 2017. URL : <https://www.omg.org/spec/UML/2.5.1/>.