Université de Namur

2020 — 2021

Amsi
INFOB313 / IHDCB335

Sujet de Labo

Table des matières

1 Introduction

Ce document constitue la première partie d'un Labo commun entre le cours d'Analyse et Modélisation des Systèmes d'Information (INFOB313/IHDCB335) d'une part, et le cours de Syntaxe & Sémantique (INFOB314/IHDCB332) d'autre part.

L'idée est de montrer comment le développement effectif d'une solution à un problème met en action les acquis conceptuels et techniques de chacune des matières : l'analyse du problème et sa modélisation d'une part, et les techniques de compilation (analyse syntaxique et sémantique, génération de code) d'autre part.

1.1 Langages de Modélisation Dédiés

Un Langage de Modélisation Dédié (Lmd, ou *Domain-Specific Language* en anglais, terme que l'on utilisera dans le reste du document) se focalise sur un domaine restreint : il peut être purement informatique (comme la définition de bases de données), ou métier (comme la définition de polices d'assurances), contrastant ainsi avec l'usage traditionnel des langages de programmation classiques (comme Java ou C) qui permettent d'effectuer n'importe quel calcul (on dit alors qu'ils sont *Turing-complets*).

Un Dsl cherche à capturer les possibles variations d'un domaine afin d'en faciliter la définition ; souvent d'ailleurs, un Dsl va de pair avec une infrastructure plus conséquente : les artéfacts définis et/ou produits à partir du Dsl vont venir s'intégrer au sein d'une infrastructure logicielle plus conséquente. Par exemple, un opérateur de téléphone va utiliser un Dsl pour définir l'ensemble des forfaits qu'il propose à ses clients, lui permettant d'ajuster leur durée, leur spécificité (quand ont lieu les périodes de gratuité des appels et/ou Sms, etc.) et leur tarif ; ce Dsl va alors modifier le comportement d'une lourde infrastructure gérant, entre autres, l'ensemble des antennes, le système de facturation..., sans pour autant devoir réécrire ces parties de logiciel à chaque nouveau forfait.

L'avantage d'un Dsl réside dans cette spécificité : plus restreint, il permet d'automatiser de nombreuses tâches allant de la vérification de la consistance des modèles durant les premières étapes du cycle de développement, jusqu'à la génération complète d'un code fonctionnel, en passant par l'analyse approfondie et la remontée d'erreurs, favorisant leur capture au plus tôt dans le cycle de développement, contribuant à une réduction drastique des coûts liés à leur maintenance.

Un Dsl reste un langage au sens informatique du terme : il faut donc en définir ses syntaxes (i.e., les phrases du langage que l'on considère comme valides) et sa sémantique (i.e., le sens, l'interprétation que l'on attache à ces phrases).

La syntaxe *abstraite* constitute la représentation interne du langage, celle que l'ordinateur (et le programmeur) utilise afin d'opérer les calculs nécessaires au traitement du langage (en particulier, elle est parcourue exhaustivement pour produire le code final). La syntaxe *concrète* s'adresse aux utilisateurs pour manipuler le langage : elle peut être purement textuelle, mais peut aussi avoir une forme plus graphique (souvent hybride, car elle mélange du texte à des formes géométriques), comme nous le montrons dans ce document. Les deux formes ne sont pas exclusives, elles dépendent du public auquel le langage s'adresse (typiquement, une syntaxe purement textuelle est traditionnellement réservée aux informaticiens).

L'opération permettant d'inférer la syntaxe abstraite à partir de la syntaxe concrète s'appelle le *parsing* : il est généralement plus facile pour des formes textuelles, et son étude est l'objet des cours de Syntaxe & Sémantique.

L'objectif général dans le cours d'Amsi consiste à définir des langages de modélisation suffisamment précis pour pouvoir modéliser en détail l'exemple proposé. Le même exemple sera repris dans le cours de Syntaxe & Sémantique pour étudier la génération de code à partir d'une grammaire.

1.2 Objectifs & Organisation

Ce document décrit un World Game *fictif*, proposé à des fins d'apprentissage de la programmation. Le but est d'apprendre la programmation à l'aide d'un langage simplifié, où le scénarion d'apprentissage met en jeu un jeu de rôles où :

— deux joueurs s'affrontent pour remporter un Graal, garant de l'accès au monde suivant ;

— le joueur qui remporte la partie est celui qui termine (ou va le plus loin dans) le jeu.

Le langage de programmation utilisé pour décrire les mouvements des personnages mêle deux paradigmes connus :

Programmation Impérative où chaque instruction décrit un pas de calcul effectué par la plateforme d'exécution ;

Programmation Déclarative où le programmeur-joueur décrit un objectif que la plateforme va essayer de satisfaire en utilisant des heuristiques.

Bien que le jeu lui-même contient des éléments de programmation, les solutions ne doivent pas être *implémentatoires* ; nous souhaitons une **modélisation conceptuelle**.

Dans ce Labo, nous visons la modélisation *complète* des concepts du jeu, composée de deux éléments majeurs :

Décrire les *Plateaux de Jeu* (§2) et ses composantes, où il s'agit de modéliser les éléments constituant les leçons et niveaux.

Définir un *Langage de Stratégie* (§3) permettant de traduire en instructions précises les étapes guidant le personnage vers la sortie.

Pour chacune des deux parties, *Plateau de Jeu* & *Langage de Stratégie*, votre travail consiste en la production de trois artéfacts :

1. Un **Diagramme de Classes** capturant les concepts présents dans chaque partie ;
2. Un ensemble d'**expressions OCL** énonçant les contraintes et contrats portant sur les diagrammes précédents ;
3. Un **Diagramme d'Objets** illustrant une utilisation des diagrammes et expressions sur chaque partie ;

Pour vous aider, une série de Questions (§4) vous guident dans la réalisation du Labo : nous vous demandons de suivre les questions pour faciliter **notre** lecture.

1.3 Critères d'Évaluation

La réalisation du Labo a lieu **par binôme**. L'évaluation tient explicitement compte des points suivants :

1. La **modélisation** au niveau *conceptuel* : en particulier, toute solution qui reposerait sur des « astuces » d'implémentation sont sanctionnées ;
2. La **cohérence interne** à chaque partie (*Plateau de Jeu* & *Langage de Stratégie*) : en particulier, les Diagrammes d'Objets doivent être *conformes* aux Diagrammes de Classe au sein d'une même partie ; et les expressions OCL doivent respecter les déclarations du Diagramme de Classe (en particulier, noms de classes, d'attributs, de rôles d'association).
3. La **cohérence externe** entre les parties : en particulier, les (noms de) classes nécessaires à la modélisation de la partie *Langage de Stratégie* doivent respecter les déclarations de classe dans la partie *Plateau de Jeu*.

2 Description d'un Match

Deux joueurs s'affrontent dans un match constitué d'un nombre impair de rencontres qui ont lieu dans une série de *mondes*, ou *plateau de jeu*.

Une rencontre est remportée soit par le joueur dont le personnage trouve le Graal, objet magique ouvrant l'accès au monde suivant ; soit par le joueur dont le personnage reste vivant le plus longtemps sans se faire tuer par son adversaire, ou part les nombreuses créatures peuplant les mondes. Le joueur remportant le plus grand nombre de rencontres gagne le match.

Selon un commun accord entre les participants, un match peut se dérouler suivant deux modes :

Mode Éducatif Les rencontres du match sont jouées automatiquement : chaque joueur sélectionne une *stratégie de jeu*, exprimée à l'aide du *Langage de Stratégies* proposé par le jeu (et qu'il faut modéliser, cf. §3), et automatiquement chargée par le jeu. La stratégie guide les mouvements d'un personnage qui sert d'avatar du joueur au sein du jeu. La rencontre donne la main à chaque joueur à tour de rôle, jusqu'à ce qu'un joueur meurre, trouve le Graal ou qu'une limite de tour ou de temps soit atteinte.

Mode Interactif Les rencontres sont jouées en *live*, en réagissant aux interactions des joueurs à partir d'un clavier ou d'une manette, de la même manière que dans un jeu d'arcades ou sur une console.

Les deux modes utilisent les mêmes aspects, mais diffèrent fondamentalement dans la manière dont le jeu exécute les mouvements : en *éducatif*, ce sont les stratégies qui guident les mouvements des avatars ; en *interactif*, ce sont les joueurs qui pilotent directement leurs avatars.

Le jeu propose aussi des matches en *multi-joueurs* : au lieu de laisser s'affronter seulement deux joueurs, le jeu permet à plusieurs joueurs de se rencontrer. Les règles de victoire ne changent pas, mais peuvent donner lieu à des égalités suivant le nombre de joueurs et de rencontres choisies. Finalement, il est aussi possible de jouer contre l'ordinateur : les concepteurs du jeu utilisent eux-mêmes le Langage de Stratégies pour concevoir des stratégies que l'ordinateur utilisera, et le joueur souhaitant jouer ainsi peut sélectionner le niveau de difficulté de l'ordinateur (débutant, avancé, expert).

2.1 Joueurs & Personnages

Un joueur possède un personnage (ou plusieurs), aussi appelé *avatar*, qui le représente physiquement dans le monde. Le joueur peut changer l'aspect de son avatar en choisissant parmi une liste de figurines fixé à l'avance. Moyennant paiement, il est possible de rajouter de nouvelles figurines au jeu et ainsi accéder à des personnages plus loufoques. Chaque personnage possède les caractéristiques suivantes :

Potentiel de Vie. Un personnage possède un *potentiel de vie* : ce potentiel indique la « résistance » du joueur aux attaques. Si le potentiel tombe à zéro, le joueur meurt. Il n'y a pas de valeur maximale.

Inventaire. Un joueur possède un *inventaire* composé de nourriture, d'eau, de munitions, ainsi que des items qu'il aurait pu ramasser. La nourriture et la boisson regonfle le potentiel de vie d'une valeur prédéfinie pour chacun ; les munitions permettent d'attaquer les ennemis durant les phases de combat ; les différents items et leurs utilisations sont décrits plus bas.

Caractéristiques de combat. Un joueur possède deux caractéristiques : le *ratio d'attaque* indique la sévérité des attaques infligées, en termes de potentiel de vie perdu par celui qui les subit ; et le *ratio de défense* indique la sévérité des attaques reçues. Les bonus, décrits plus bas, permettent de faire évoluer ces caractéristiques tout au long du jeu.

Chaque monde est parsemé de zombies, créatures calmes qui déambulent dans le monde à la recherche de nourriture : lorsqu'un joueur passe à proximité, les zombies l'attaquent pour pouvoir le dévorer. Les zombies parviennent à détecter des humains à une distance prédéfinie lié à leur ratio d'attaque : lorsqu'un joueur entre dans ce rayon d'action, le zombie ne peut s'empêcher de se diriger vers lui pour tenter de le dévorer. Comme tout joueur, un zombie possède un potentiel de vie qu'il faut rendre nul pour le tuer (auquel cas, il disparaît du monde) et de caractéristiques de combat.

2.2 Monde / Plateau de Jeu

Un monde est représenté par un plateau de jeu carré de dimension fixe, sur lequel les personnages peuvent se déplacer. La plupart des cases sont vides, mais certaines contiennent des obstacles ou des items :

Obstacles. Les obstacles comme un mur, un arbre ou un menhir, sont infranchissables : un personnage ne peut jamais accéder à cette case, même s'il cherche à détruire l'obstacle. Au contraire, des ronces, des buissons, un tronc d'arbre ou une tête de mort peuvent être détruits, permettant ensuite au personnage d'accéder à la case où ils se trouvaient.

Les mondes sont aussi parsemés de zones d'eau, de feu ou de glace. Rester sur une case dans une zone de feu sans les bottes pare-feu retire un point de vie par tour ; il n'est possible de traverser une zone d'eau et de glace qu'en possédant le kit de plongée et les bottes à crampons dans son inventaire (le jeu empêche simplement le passage sur la zone correspondante).

Items. Plusieurs items sont distribués dans les mondes pour aider les personnages dans leur avancée : la nourriture et les boissons remontent le potentiel de vie du personnage qui les utilise ; les recharges de munition permettent de tirer à distance sur un ennemi ; les bottes pare-feu ou à crampons, et les kits de plongée permettent de franchir les zones de feu, glace et eau respectivement. Des bonus agrémentent certains mondes : ils permettent de faire évoluer les caractéristiques de combat d'un personnage en améliorant ses capacités d'attaque ou de défense.

Orientation. Deux items sont d'une aide importante pour l'orientation et les décisions stratégiques : un radar et une carte permettent de déterminer respectivement la position de l'autre personnage sur le plateau, et celle du Graal. Cette position est signalée par une flèche donnant la direction à vol d'oiseau vers la cible : pour la carte, cette position reste affichée jusqu'à la fin de la rencontre ; pour le radar, la position reste affichée durant n tours, nombre qui est doublé à chaque fois que le personnage ramasse un nouveau radar.

Cotte de Maille & Cape de Mage Deux items confèrent aux personnages des capacités hors norme. Un personnage peut les ramasser, mais doit les rendre actifs en les enfilant : lorsque un personnage porte une cotte de maille, il devient insensible à toutes les armes du jeu ; revêtir la cape de mage permet de traverser n'importe quelle zone sans les items correspondants, et de traverser des obstacles infranchissables trois fois (gare à ne pas se faire enfermer à l'intérieur d'une enceinte !). Chaque item a un effet qui dure durant un nombre de tours égal à un dixième des points de vie de l'avatar qui l'utilise en mode éducatif, ou durant une minute en mode interactif.

Une seule case dans chaque monde contient le *Graal*, objet dont il faut se saisir pour s'assurer d'accéder au monde suivant.

La carte de la Figure 1 donne un exemple de monde de taille 27x27. Le monde est bordé de murs infranchissables (lignes et colonnes 0 et 26) empêchant les joueurs, placés ici en positions $(1, 1)$ et en $(22, 14)$ (on nommera le second "le roux" à cause de sa barbe rousse), de sortir autrement qu'en trouvant le Graal placé en $(25, 25)$. D'autres obstacles parsèment le monde : par exemple, en $(8, 2)$, et verticalement entre $(24, 5)$ à $(24, 7)$ se trouvent d'autres murs infranchissables ; tandis qu'en $(4, 21)$ et $(7, 21)$ se trouvent des obstacles qu'un joueur peut éliminer ; le premier obstacle en $(4, 21)$ avec la tête de mort nécessitant d'être frappé plusieurs fois avant de disparaître (et a donc une valeur d'item plus importante). Le premier joueur, en bas à gauche, est chanceux : il trouvera à proximité une carte en $(2, 3)$ et un radar en $(2, 5)$ ou $(4, 4)$. Le second joueur, lui, trouvera de la nourriture en $(23, 12)$ et une boisson $(23, 17)$. Il n'y a par contre aucun bonus ni aucune recharge de munition dans ce monde : les joueurs devront mener leurs combats au corps à corps, sauf s'ils ont accumulé des munitions dans les mondes précédents.

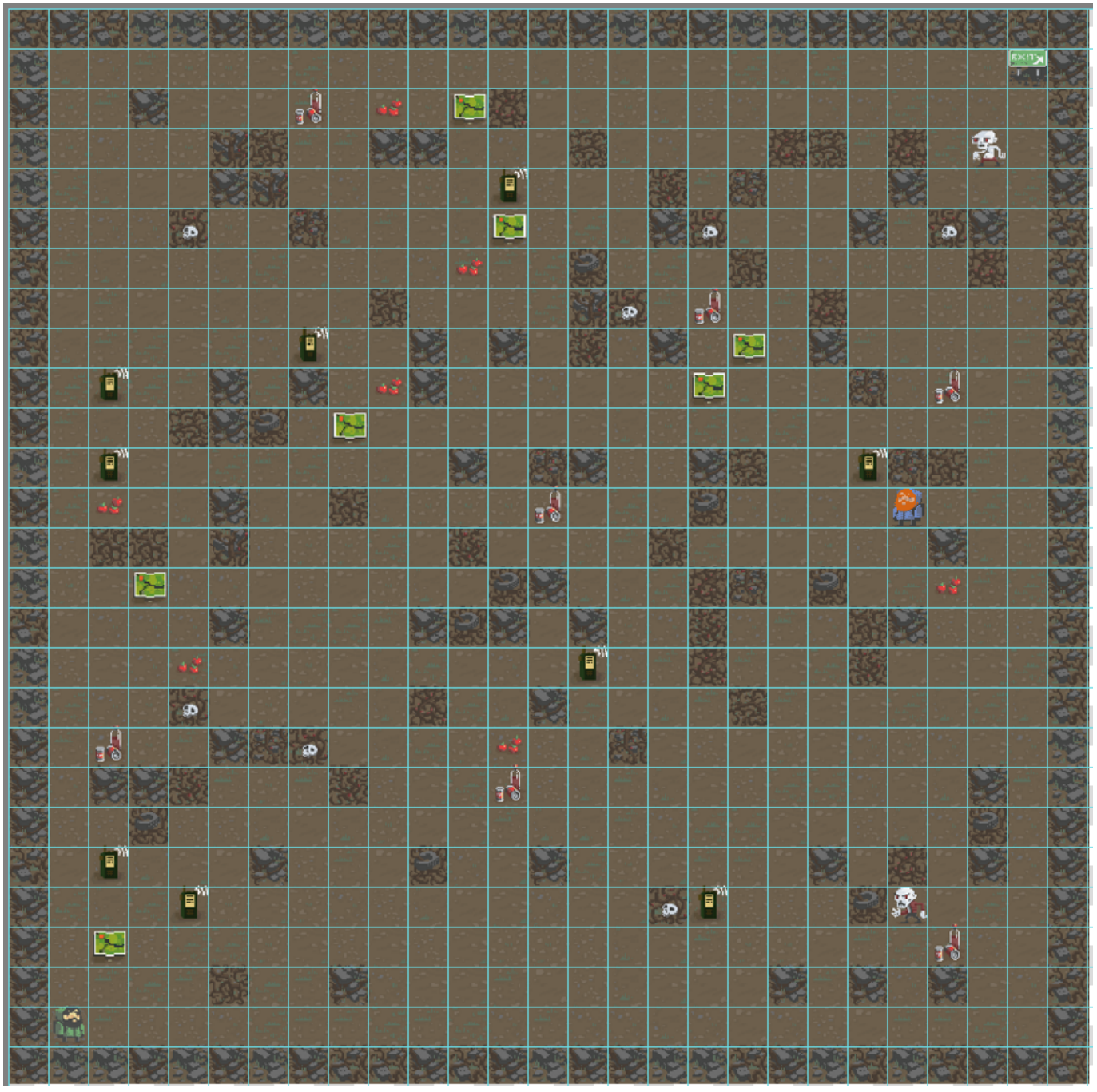

FIGURE 1 – Un exemple de monde de taille 27, sans brouillard d'invisibilité. On remarque les deux personnages en positions $(1, 1)$ et $(22, 14)$, et le Graal dans le coin supérieur droit (si on ne tient pas compte de la bordure). Une multitude d'items (boissons, nourriture, cartes et radars) pourraient être ramassés par les personnages pour les aider dans leur quête.

2.3 Déroulement du Jeu

Le jeu est conçu pour pouvoir être joué suivant plusieurs modes, comme expliqué en introduction de cette Section. Pour rappel, lorsqu'un joueur lance le jeu, il doit choisir :

— s'il veut jouer en mode éducatif ou interactif ;

— s'il veut jouer contre un seul joueur, ou plusieurs ;

— de combien de rencontre se constitue le match qu'il s'apprête à jouer.

Chaque joueur est identifié par un pseudo unique sur la plateforme, et peut sauvegarder le profil de plusieurs avatars, conservant ainsi les informations suivantes :

— les caractéristiques de l'avatar, comme détaillés en §2.1 ;

— le détail des rencontres et matchs remportés, et l'identifiant de l'opposant, ainsi que le mode dans lequel se sont joués ces matchs ;

— optionnellement, une série de stratégies, identifiées par un libellé descriptif.

Les matchs sont identifiés de manière unique afin de pouvoir rapidement en retrouver les détails (participants, modes, palmarès, etc.) Lorsque les conditions du match sont fixés, le joueur doit sélectionner un avatar parmi ceux qu'il possède, retrouvant ainsi l'avatar dans les conditions où il l'avait laissé. Pour chaque rencontre, le joueur est libre d'eventuellement adapter sa stratégie, en fonction de l'aperçu que le jeu donne du Plateau de Jeu où se déroulera la rencontre. Cependant, pour éviter les parties infinies, une rencontre peut être paramétrée de façon à imposer un chronomètre ou un compteur de tours au terme desquels la rencontre s'arrête. Seuls les matchs joués en mode éducatifs contre l'ordinateur permettent des parties « infinies », çàd. sans limite de temps ni de tours.

L'aperçu offert avant une rencontre permet aux joueurs d'apprécier la dimension du Plateau de Jeu ainsi que la distribution des items en son sein, afin de sélectionner (ou définir) une stratégie adéquate. À tout moment, un joueur :

— connaît la *positon absolue* de son personnage sur la carte, c'est-à-dire les coordonnées exactes de sa position dans le carré représentant le monde ;

— voit autour de lui, suivant une *portée de visibilité*, un espace carré où il peut savoir tout ce qui s'y trouve (ressources et bonus, ainsi que la position éventuelle du Graal s'il est visible).

En mode interactif, le jeu est joué dynamiquement en fonction des mouvements commandés par les joueurs. En mode éducatif, la main est donnée à chaque joueur à tour de rôle, appliquant les stratégies de chacun. Les actions à la disposition des personnages sont les suivantes :

SeDéplacer. Un joueur peut déplacer son personnage dans l'une des directions possibles (**H**aut, **Bas**, **G**auche ou **D**roite), à condition que la case où il souhaite aller ne soit pas infranchissable (un mur ou le bord du monde), auquel cas, le déplacement n'est pas effectué ; si la case sur laquelle il arrive contient un item, le joueur le ramasse automatiquement.

UtiliserItem. Un joueur peut utiliser les éléments consommables de son inventaire (nourriture ou boisson) afin de regagner du potentiel de vie, ou recharger ses munitions. Les autres items (bottes pare-feu, à crampons ou kit de plongée) ne nécessitent pas l'intervention explicite du joueur.

Revêtir. Un joueur peut vouloir revêtir sa Cape ou sa Cotte de Mailles afin de profiter de son effet.

Frapper. Un joueur peut frapper le contenu de la case adjacente dans le sens de son dernier déplacement (par exemple, si son dernier mouvement était vers la droite, il frappera la case à droite). Il peut frapper un autre joueur ou un zombie, pour tenter de le tuer ; mais il peut aussi tenter de frapper sur un buisson ou des ronces afin de les éliminer et libérer le passage.

Tirer. Lorsqu'un joueur voit un ennemi à proximité (donc, *a priori*, pas sur une case adjacente), il peut lui tirer dessus en consommant une munition de son inventaire, toujours dans le sens de son dernier déplacement. Le tir ne s'arrête que sur un obstacle infranchissable et assène un coup sur le premier ennemi sur son passage.

FIGURE 2 – Un exemple de carré de visibilité autour des personnages : à gauche, la portée de visibilité est de 2, conduisant à 9 cases visibles parce que le personnage est dans le coin ; à droite, la portée est de 3, conduisant à 49 cases visibles.

ConsulterRadar. Cette action permet de connaître précisément la position de l'autre joueur dans le monde pendant n tours.

À chaque tour, un personnage ne peut pas effectuer plus d'un mouvement ; par contre, il peut combiner un mouvement avec une ou plusieurs autres actions de cette liste.

Une attaque peut être portée soit sur un ennemi (un autre joueur, ou un zombie), soit sur un obstacle (s'il est infranchissable, l'attaque n'a pas d'effet) de deux manières différentes :

Courte portée (ou combat rapproché). Cela correspond à l'action Frapper, le joueur donne un coup d'épée sur la case immédiatement contigüe à celle où il se trouve, dans le sens de son déplacement précédent ;

Longue portée (ou combat distant). Cela correspondant à l'action Tirer, le joueur tire une munition dans la direction de son déplacement précédent.

Les zombies ne s'engagent que sur des combats rapprochés (ils ne peuvent pas "tirer" à distance) lorsqu'ils "sentent" un humain.

L'effet d'une attaque à courte ou longue portée est défini à chaque début de match dans une variable effetAttaque : pour simplifier, cette valeur est supposée être la même pour les deux types d'attaque. Cependant, l'effet est mitigé par les ratios d'attaque et de défense des personnages : avec un ratio d'attaque élevé, une attaque est amplifiée et provoque donc des dommages plus importants ; avec un ratio de défense plus élevé, l'effet d'une attaque reçue est diminué, provoquant des dommages moins importants. On suppose aussi que

les zombies n'ont pas de ratio de défense : une attaque sur un zombie dépend to ujours du personnage qui la porte.

Chaque joueur voit sur son écran le monde relativement à son propre personnage (*a priori*, ils ne jouent pas sur le même ordinateur, mais en réseau) : il voit le plateau de jeu ainsi qu'une barre d'état verte affichant certaines données. La Figure 2 montre en surimposition deux exemples de telles vues écran sur la même carte que celle présentée en Figure 1, avec les personnages aux mêmes places. Cette fois, le plateau est recouvert d'un brouillard noir masquant l'ensemble du plateau, sauf dans le carré immédiatement autour du personnage, dont la taille dépend des caractéristiques du joueur : le personnage dans le coin a une portée de visibilité de 2 tandis que le personnage roux a une portée de visibilité de taille 3. La taille du carré peut éventuellement augmenter en ramassant les bonus adéquats.

La barre d'état indique que les joueurs ont le même inventaire : une carte, un radar, aucune munition, trois doses de nourriture et deux de boisson. Par contre, ils n'ont pas le même potentiel de vie : celui dans le coin a 100 tandis que le roux a 76.

3 Langage de Stratégie

Avant chaque rencontre, les joueurs voient pendant quelques secondes le Plateau de Jeu dans son entièreté. En mode éducatif, cet aperçu permet aux joueurs de sélectionner une stratégie adéquate pour faire évoluer leur personnage. Une stratégie est conçue pour guider les déplacements et actions d'un personnage : elle définit quand et comment un avatar se déplace, quand il fait usage de quels items disponibles dans son inventaire, quand et comment il s'engage dans des combats, s'il privilégie de tuer son adversaire ou de trouver le Graal, etc.

Cette section définit un mini-DSL typé permettant aux joueurs de spécifier des stratégies de jeu. Une (instance de) stratégie s'articule autour d'*objectifs* qui sont réalisés par un ensemble de *règles*. Une stratégie peut faire usage d'éléments réutilisables nommés *modules*, qui peuvent définir un comportement arbitraire.

3.1 Types

Afin de rester compatible avec de multiples plateformes de jeu, le DSL de stratégie se base sur des types primitifs simples (booléens, entiers, réels, chaîne de caractère) et des énumérations, ainsi que des combinaisons simplifiées de ces types :

Tableau définit une séquence de valeurs de même type, de longueur finie. Un même tableau peut être déclaré de différentes manières, mais sera manipulé de manière uniforme. Par exemple, le code suivant définit une Matrix carrée d'entiers de trois manières distinctes :

```
type VisiblePlayground1 = array [1 .. MAX] of array [1 .. MAX] of integer;

type VisibleLine = array [1 .. MAX] of integer;
type VisiblePlayground2 = array [1 .. MAX] of VisibleLine;

type VisiblePlayground3 = array [1 .. MAX, 1 .. MAX] of integer;
```

Une variable de type entier peut être utilisée comme bornes (par exemple, en calculant une taille à partir de la portée de visibilité, ou en comptant le nombre d'items stockés dans l'inventaire).

Notons la manière dont VisiblePlayground est définie de manière identique avec trois syntaxes différentes : sous la forme d'un tableau de tableau (VisiblePlayground1) ; sous la forme d'un tableau d'un type tableau (VisiblePlayground2) ; et sous la forme d'un tableau bi-dimensionnel (VisiblePlayground3). Ces différentes formes de déclarations donneront lieu à differentes formes d'affectation :

```
var total : integer;
var line : VisibleLine;
var visible, enemy : VisiblePlayground1;
var area : VisiblePlayground2;
var bigArea : VisiblePlayground3;
visible ⇐ enemy // Affectation correcte
bigArea[0,0] ⇐ 0; // Correcte car bigArea : VisiblePlayground3
bigArea[0] ⇐ line; // Incorrect : bigArea est bi-dimensionnel
visible[0] ⇐ line; // Correcte
visible[0,0] ⇐ total // Incorrect : visible : VisiblePlayground1 n'est pas bi-dimensionnel !
visible[0][0] ⇐ total // Correct
area ⇐ enemy; // Correct : VisibleLine définit la même chose que la partie of de VisiblePlayground1
```

Un tableau du Langage de Stratégie peut se traduire nativement dans la structure correspondante dans un langage de programmation (par exemple, C), ou faire l'objet d'une abstraction sous la forme d'une collection appropriée (par exemple en Java).

Enregistrement définit un tuple de valeurs accessibles au travers d'un champ nommé à l'aide de la notation pointée. Par exemple, le code suivant définit un (type) enregistrement Case, et y affecte une valeur 0.

```
type Cell : record
      x : integer ;
      y : integer ;
endrecord
var position : Cell
...
position.x ⇐ 0 ;
position.y ⇐ 0 ;
```

Un enregistrement peut être traduit nativement si le langage dispose de cette construction (par exemple en Pascal/Delphi), ou correspondre à une classe (en Java).

3.2 Déclarations globales : Modules & Variables

Un *module* est un élément réutilisable au sein d'une stratégie qui peut être arbitrairement complexe et paramétrable : il peut définir une série de déplacements-type, une procédure de calcul (du chemin le plus court vers une destination, ou d'une heuristique d'évitement de zombie), ou simplement un ensemble d'actions fréquentes, comme la manière de combattre un ennemi particulièrement récalcitrant (un « boss »). Un module est identifié par un nom unique au sein d'une instance de stratégie, comporte un ensemble de paramètres typés et retourne une valeur typée.

Il est possible de définir des variables dites *globales*, c'est-à-dire accessibles à partir de n'importe quelle entité au sein d'une stratégie (vision, objectif ou règle), ou *locales*, c'est-à-dire accessible seulement au sein d'un module. Une variable est simplement la donnée d'un nom, unique parmi les variables globales, et du type assigné à la variable.

```
var position : Cell ;
...
module shortestDistance
      param start, end : Cell
      produces integer
begin
var sum : integer
...
end
```

Le fragment de code ci-dessus montre deux déclarations : une variable globale position, et un module shortestDistance, qui définit deux paramètres start et end, et une variable locale sum. Dans cette situation, position pourrait être utilisée dans le corps du module, mais sum n'existe pas en dehors de celui-ci.

3.3 Objectifs

Pour gagner des rencontres, et donc le match, il faut soit trouver le Graal, soit éliminer son adversaire. Pour ce faire, il faut pouvoir les repérer tout en contournant les obstacles, en se défendant contre les zombies tout en restant vivant le plus longtemps possible.

Le langage de stratégie est centré autour de deux types de vision : à *long* et à *court terme*. La vision à long terme définit l'objectif général du personnage, celui vers lequel il va tendre tout au long du jeu. La vision à court terme permet de réagir aux évènements directs que le personnage rencontre : devoir réagir à une attaque ; devoir contourner des murs pour suivre une direction ; trouver des ressources pour ne pas mourir. Ces visions à court et long termes vont pouvoir évoluer en fonction du terrain couvert par le personnage et des priorités que le joueur privilégie dans sa manière de jouer.

Chaque vision poursuit un objectif particulier, choisi dans la liste suivante (la liste pourra évoluer par la suite, quand les développeurs auront fait le tour de tous les objectifs possibles) :

Néant est l'objectif vide par défaut, pour lequel les déplacements sont choisis aléatoirement.

AllerVers. Cet objectif, paramétré par une case, indique que le joueur souhaite atteindre cette case le plus rapidement possible (c'est-à-dire, en jouant le minimum de coups possibles, en fonction des obstacles présents).

Contourner. Cet objectif, paramétré par un ensemble de cases (symbolisant par exemple un obstacle), indique que le joueur veut atteindre une case se trouvant "au-delà" de son paramètre. Cet objectif est dans l'esprit similaire à AllerVers une case pré-spécifiée au-delà de l'obstacle, mais admet en paramètre optionnel un procédé de contournement guidant les déplacements : on veut pouvoir spécifier, pour éviter certains pièges comme la présence de zombies, de contourner un obstacle par la droite ou la gauche, ou par le haut ou le bas (exprimé en direction absolue par rapport à l'orientation du monde).

Eviter. Cet objectif, paramétré par une case, un joueur ou un zombie, indique que le joueur souhaite éviter l'élément en paramètre en adaptant ses mouvements pour s'en éloigner. Il peut être combiné avec SeDirigerVers ou Contourner pour spécifier des déplacements plus complexes.

CollecterMax. Cet objectif, paramétré par un type d'item, indique que le joueur veut collecter un maximum d'items d'une catégorie donnée se trouvant autour de lui, en adaptant ses déplacements pour couvrir la plus grande aire possible, se donnant ainsi la chance de maximiser le nombre d'items collectés.

Combattre. Cet objectif permet au joueur d'engager rapidement le combat avec le joueur adverse ou les zombies à proximité, si possible avant que ceux-ci n'aient le temps de riposter. Cet objectif doit être implémenté par des règles réalisant un subtil équilibre entre l'utilisation des armes à courte et longue portées.

Les visions à court et long termes sont modifiées par une action spécifique nommée changer (paramétrée par la vision concernée). Le comportement général est le suivant : si la vision à long terme est Néant, mais que celle à court terme est spécifié, cette dernière est considérée comme la vision à long terme.

3.4 Règles

Définir des règles pour réaliser une stratégie est une tâche souvent complexe : en effet, il est très facile, si on ne fait pas attention, de tomber dans des comportements cycliques qui ne réalisent pas du tout les objectifs fixés (par exemple, un personnage tourne en rond sur le plateau parce que les règles s'activent toujours suivant la même séquence). Les développeurs réfléchissent encore à une approche pédagogique pour l'écriture des règles, et à intégrer un système d'analyse de règles pour détecter des comportements non désirés, comme les comportements cycliques. Cependant, elle cherche quand même à avoir une première mouture de langage de stratégie pour pouvoir expérimenter l'écriture de stratégie et tester le comportement des joueurs.

Un objectif est réalisé par un ensemble de règles, et l'objectif acquiert une priorité implicite suivant la vision qu'il poursuit : les règles réalisant un objectif poursuivant une vision à long terme sont toujours considérées comme moins prioritaires que celles rattachées à la vision à court terme. Ceci permet de réagir prioritairement aux situations particulières qu'un personnage rencontre, indépendamment de la vision à long terme qu'il poursuit. La réalisation d'un objectif suppose l'existence d'une règle « par défaut » qui se déclencherait dans l'éventualité où aucune autre ne s'applique.

En outre, les règles sont séparées en deux catégories : les règles normales réalisent un objectif, tandis que les *métarègles* permettent de changer l'objectif poursuivi par une vision. Seules les métarègles peuvent contenir l'action changer. Les règles possèdent en outre les caractéristiques suivantes :

Priorité. Une règle possède une priorité explicite, exprimée par un entier positif, qui permet de contrôler le déclenchement des règles et de lever les sources d'indéterminisme ; la priorité de la règle par défaut est toujours 0.

Déclencheur. Le déclencheur, constitué d'une expression évaluée comme un booléen, est la partie qui, si elle est évaluée à vraie, lance l'exécution immédiate de la règle, à condition qu'il n'y ait pas d'autre règle plus prioritaire ;

Réaction. La réaction est la partie de la règle qui définit les actions à réaliser, au moyen d'un ensemble de déclarations, et d'une séquence d'instructions dont la définition est l'objet de la Section 3.6. Ces actions sont celles spécifiées en Section 2.3 : une séquence d'action ne peut comporter qu'un seul mouvement,

mais peut combiner arbitrairement les autres types d'action ; et seule une métarègle peut contenir une action changer.

Lorsqu'elles sont évaluées durant le jeu, des règles dont les déclencheurs s'activent en même temps peuvent mener à des comportements indéterministes : la priorité de la règle est alors utilisée pour n'en sélectionner qu'une. Cela impose que les priorités d'un ensemble de règles réalisant un même objectif sont toutes différentes, pour éviter des conflits sur les priorités aussi.

3.5 Expressions

Les expressions sont utilisées à la fois dans les déclencheurs, où elles constituent une garde régulant l'exécution d'une règle, et dans les instructions, où elles permettent des calculs dont l'évaluation contribue à la décision. Structurellement, une expression peut être d'un des types suivants :

Litéral recouvre l'ensemble des expressions litérales, c'est-à-dire dont l'interprétation correspond à elle-même.

Expression Binaire désigne l'ensemble des expressions composées à partir d'un opérateur binaire, à partir de deux sous-expressions.

Expression Unaire désigne l'ensemble des expressions composées à partir d'un opérateur unaire et d'une sous-expression.

Appel de Module désigne l'expression permettant d'invoquer le comportement spécifié par un module à partir de son nom et d'une liste d'expressions définissant les paramètres « *effectifs* » appropriés ;

Expression Gauche (« *Left-Hand Side* », Lhs) désigne un « réceptacle » de valeur, c'est-à-dire une expression qui peut être stocker une valeur, et donc être affectée (cf. la définition des Instructions en Section 3.6). Une Lhs peut désigner soit (l'accès à la valeur d') une variable, soit (l'accès à la valeur d') un paramètre, soit (l'accès au contenu d') une case d'un tableau (possiblement multidimensionnel), soit (l'accès à la valeur d') un champ d'un enregistrement.

Expression Parenthésée désigne une expression de groupage (typiquement, à l'aide de parenthèses, ou d'un symbole d'encapsulation graphique) constitué d'une sous-expression.

	Catégorie	Type	Exemples	Sous-Expressions
Expressions	**Literal**	Entier	0 1	
		Booléen	true false	
		String	“super” “abcd”	
		Valeur d'Enumération	Direction.Up	
	Lhs	Variable	position, total, line	
		Paramètre	start end	
		Cellule(s)	visible[0, 0] area[0]	
		Champ	position.x	
		Tableau	bigArea	
		Enregistrement	position	
	Unaire	Réel	-1.2	
		Boolean	**not** v	v
	Binaire	Numérique	total + 1, total = 1	total
		Boolean	isVisible **or** isFarAway	isVisible, isFarAway
	Appel de Module		shortestDistance(position, enemyPos)	position, enemyPos
	Parenthésée	Booléen	(**not** isVisible) or isFarAway	a, b

Table 1 – Illustration d'expressions en syntaxe textuelle.

En Table 1 sont présentés quelques exemples d'expressions classiques sur la base d'une syntaxe textuelle proche de ce qu'on pourrait trouver dans les langages C ou Pascal.

3.6 Instructions

Les instructions constituent le cœur de la définition des règles, et permettent d'à la fois stocker des informations sur l'environnement de jeu (par exemple, la position détectée du Graal ou d'un ennemi, la mémoire sur les mouvements précédents pouvant conduire à contourner un obstacle), et à spécifier les mouvements d'un joueur. Comme à l'accoutumé, les instructions utilisent des expressions (cf. Section 3.5) pour définir des tests et calculer des valeurs. Les instructions sont de cinq types :

Skip est l'instruction vide, qui ne produit aucun effet visible dans l'état ou la situation d'un joueur (elle est utilisée pour des raisons de compatibilité, et pour permettre d'éventuellement sauter des tours) ;

Conditionnelle est une instruction spécifiant un choix conditionné par une garde booléenne exprimée par une expression ; une Conditionnelle se décline en deux versions, dont l'une permet de ne réaliser aucune action si la garde est fausse ;

Itération est une instruction spécifiant un calcul itératif, comme dans une boucle ; une Itération possède un corps composé lui-même d'instructions qui sont exécutées tant que la garde de l'Itération reste vraie ;

Affectation permet d'assigner une valeur à un réceptacle (Lhs), et se compose donc structurellement de deux parties :

- la partie gauche (*left-hand side*), apparaissant traditionnellement à gauche d'un symbole d"affectation (:= en Pascal ou simplement = en Java, on utilisera dans la Table 1 le symbole ⇐ comme dans les exemples précédents) désigne un réceptacle : soit une variable, soit une case, une colonne d'un tableau ou encore un tableau complet, ou bien encore un champ d'enregistrement ;
- la partie droite (*right-hand side*) est une expression permettant de calculer la valeur à affecter à cette partie gauche.

Élement affecté	Exemples
Variable	enemyDist ⇐ distance(current, enemyPos) lastDir ⇐ Direction.Up square ⇐ pos.x * pos.x + pos.y * pos.y
Tableau	enemy[1, 1] ⇐ MAX visible[MAX] ⇐ lineAbove
Champ	origin.x ⇐ 0 position ⇐ origin

Le tableau ci-dessus illustre les possibles réceptacles affectés avec des expressions variées, mais dont les types correspondent (les déclarations correspondent sont fictives, ou correspondent aux exemples mentionnés précédemment).

Action définit une séquence d'actions réalisables par le joueur, comme spécifié en Section 2.3 : une séquence d'action ne peut comporter qu'un seul mouvement, mais peut combiner arbitrairement les autres types d'action ; et seule une métarègle peut contenir une action changer.

4 Questions

Ce Labo vise à modéliser de manière la plus complète possible le jeu décrit en Sections 2 et 3, de manière *conceptuelle*. Afin d'accompagner dans la réalisation de ce travail, la liste de questions qui suit jalonne les étapes menant à l'obtention de modèles fonctionnels et complets.

Les questions sont groupées suivant les deux DSLs présentés précédemment : le DSL pour décrire les éléments du jeu (Section 2) et celui pour définir des stratégies (Section 3). Notez l'approche *top/down* : on modélise d'abord grossièrement et de manière très abstraite certains concepts, avant de les raffiner dans des questions ultérieures pour obtenir une modélisation complète. Pour faciliter la lecture, nous vous demandons de :

1. **Répondre aux questions *dans l'ordre***;
2. **Faire figurer *toutes* les questions, même si vous n'y apportez pas de réponse**.
3. **Donner des titres aux questions (et éviter de recopier son numéro, peu informatif!)**

4.1 Description du Jeu (DJ)

Dans cette section, on cherche à définir un Diagramme de Classes (UML) et ses contraintes permettant de capturer les éléments constituant un match, comme décrit dans la Section 2. Pour simplifier, on pourra considérer que dans la version réactive du jeu, l'ordinateur est un joueur comme un autre, avec un personnage propre évoluant sur le plateau : ceci permet de s'exercer contre un joueur fictif.

▻ **Question 4.1** ◅
Établir une première version d'un diagramme de classes UML qui fixe les éléments principaux : le jeu rassemble des joueurs qui sauvegardent des personnages/avatars, qu'ils utilisent pour jouer des matchs, eux-mêmes constitués d'un certain nombres de rencontres ayant lieu dans un monde.

▻ **Question 4.2** ◅
Enrichir cette première version avec les détails nécessaires concernant les joueurs et leurs personnages; les matchs, les rencontres et les mondes. Les joueurs et personnages sont décrits dans la Section 2.1; les rencontres et les mondes dans la Section 2.2. La modélisation doit tenir compte des éléments indispensables pour décrire le déroulement d'une partie, comme spécifié en Section 2.3.

On prêtera une attention particulière aux éléments UML suivants :

- la multiplicité des associations doit correspondre aux contraintes de l'énoncé;
- le type des attributs choisis doit permettre l'expression des contraintes;
- les associations portant des agrégations / compositions doivent être choisies avec soin;
- si besoin, la nature des collections (*bag*, *set*, *sequence*, *ordered set*) doit être justifiée;
- les éventuelles hiérarchies doivent être pleinement spécifiées (complétude et couverture).

Puisqu'il s'agit d'une modélisation conceptuelle, les détails de visibilité de propriétés de classe (attributs / associations), ainsi que la présence d'opérations, sont à proscrire.

▻ **Question 4.3** ◅
Spécifier les contraintes d'unicité suivantes :

1. Les matchs sont identifiés de manière unique au sein du jeu;
2. Les joueurs ont des pseudos différents au sein du jeu;
3. 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;
4. Les rencontres d'un match portent un numéro d'ordre unique.

▻ **Question 4.4** ◅
Les contraintes énumérées ci-après expriment des règles permettant d'obtenir des modèles bien formés. Spécifier ces contraintes à l'aide d'éléments structurels dans le Diagramme de Classes, éventuellement complétés par des expressions OCL (dépendamment de la manière dont votre Diagramme de Classes est construit).

1. le potentiel de vie d'un joueur est toujours positif ;
2. les ratios d'attaque et de défense sont des ratios, c'est-à-dire des valeurs strictement positives et inférieures à 1 ;
3. Le nombre de rencontres constituant un match est toujours impair ;
4. 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).
5. 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).
6. Une seule des cases d'un monde contient un Graal ;
7. Les coordonnées d'une case ne peuvent excéder la longueur d'un monde ;
8. 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).
9. Un joueur se trouve toujours dans la case correspondant à sa position absolue ;
10. La bordure d'un plateau contient toujours des éléments infranchissables (pour éviter aux personnages de « sortir » du monde).
11. 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 « découverte » trois lignes au-dessus, en dessous, et trois colonnes à droite et à gauche.
12. Un personnage ne peut pas se trouver sur une case portant un obstacle infranchissable.
13. Deux personnages (y compris les zombies) ne peuvent pas se trouver sur la même case.

▻ **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 ?

▻ **Question 4.6** ◅
Définir un diagramme d'objets pour un monde de taille 4 (çàd. 2x2) 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.

4.2 Description de Stratégies (DS)

Dans cette section, on cherche à définir un Diagramme de Classes (UML) et ses contraintes permettant de modéliser le langage de stratégies comme décrit dans la Section 3.

La Question 5 précisait déjà que les deux Diagrammes sont en réalité liés, même si pour des besoins de notations et pour faciliter votre progression, leur modélisation est séparée. Il est donc normal de devoir faire référence à des classes du Diagramme de Classe précédent. Dans ce cas,

— Ne répéter que la classe référencée, sans reprendre les détails de celle-ci (attributs et/ou associations) ;
— Rester cohérent dans les noms choisis : le nom de classe ***doit*** correspondre à une classe effectivement présente dans le Diagramme de Classes précédent.

La cohérence entre les deux diagrammes est un critère prépondérant de notation.

▻ **Question 4.7** ◅
Établir une première version d'un diagramme de classe UML qui fixe les éléments principaux : une stratégie comporte une série de déclarations de modules et de variables, et est composée d'une vision

à court et long terme articulées autour d'objectifs, qui sont réalisés par des règles (dont une règle par défaut). Chaque règle comporte une priorité, un déclencheur et une réaction.

▻ **Question 4.8** ◅
Modéliser le concept de Type comme décrit dans la Section 3.1, en s'assurant de pouvoir déclarer tous les exemples donnés pour les énumérations, les tableaux et les enregistrements, sur la base des types primitifs classiques (on utilisera une classe abstraite TypePrimitif / PrimitiveType que l'on n'instanciera pas).

▻ **Question 4.9** ◅
Modéliser le concept de Declaration de variables et de modules ; puis les relier avec leurs éléments constitutifs.

▻ **Question 4.10** ◅
Modéliser explicitement le concept d'Objectif (Objective) réalisant les visions, sans oublier leur(s) paramètre(s), comme déjà modélisés dans le Diagramme décrivant les mondes.

▻ **Question 4.11** ◅
Modéliser ensuite explicitement le concept d'Action, comme décrit en Section 2.3, en veillant à en *respecter le vocabulaire*.

▻ **Question 4.12** ◅
Modéliser le concept d'Expression comme défini en Section 3.5, en s'assurant de pouvoir représenter tous les exemples de la Table 1.

▻ **Question 4.13** ◅
Modéliser le concept d'Instruction (Statement) comme défini en Section 3.6, en s'assurant de pouvoir décrire tous les exemples fournis.

▻ **Question 4.14** ◅
Spécifier les contraintes d'unicité suivantes :

- Les modules ont un nom unique au sein d'une même stratégie ;
- Les variables globales ont un nom unique au sein d'une même stratégie ;
- Les variables locales au sein d'un module possèdent un nom unique ;
- Les noms des paramètres d'un modules sont tous différents ;
- Les noms des types déclarés sont globalement uniques (en particulier, on ne peut pas nommer une énumération et un tableau de manière identique) ;

▻ **Question 4.15** ◅
Spécifier une contrainte OCL permettant de vérifier qu'une déclaration de type est bien formée :

- La liste des litéraux sont uniques au sein d'une énumération ;
- La liste des champs d'un enregistrement est non-vide ;
- Les noms des champs sont uniques au sein d'un même enregistrement ;
- Un tableau comporte au moins une dimension ;
- Chaque dimension de tableau doit être strictement positive.

Pour les questions qui suivent et qui demandent la spécification d'un **contrat**, il s'agit de définir des **pre-/post conditions** sur l'exécution de l'opération associée.

▻ **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 à :

- le type du littéral qui correspond au littéral (par exemple, le type de la valeur true doit être Boolean, celui de la chaîne "123" doit être String et celui de l'entier 122 Integer, etc.)
- le type correspondant à l'opérateur unaire, à condition que sa sous-expression corresponde à ce type. Par exemple, les types de expressions unaires - 123 et **not** isVisible doivent respectivement être Integer (puisque 12 est entier) et Boolean (à condition que la variable isVisible soit déclarée comme une variable booléenne).

— 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.

— le type de la sous-expression dans une expression parenthésée ;

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

— le type de l'énumeration pour une expression d'un accès à un littéral d'énumération. Par exemple, Direction.Up doit renvoyer le type énumération Direction.

— 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.

— le type du contenu du tableau pour une expression d'accès à un élement de tableau. Par exemple, à partir des exemples déclarés en Section 3.1, les expressions visible[0,0] et area[0] doivent respectivement retourner Integer et VisibleLine.

▻ **Question 4.17** ◅

Supposons l'existence d'une opération estTraversableGraceAuxItems(...) : Boolean, qui rend vrai ssi pour un personnage donné, une case passée en paramètres 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 ?

— Définir cette opération en précisant quel(s) serai(en)t ses paramètre(s), et quel serait le contrat OCL sur cette opération.

▻ **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

— 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 ;

— Si l'item est un radar, il double la visibilité de la position de l'adversaire ;

— 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.

Préciser le contexte (çàd. sur quelle classe est défini l'opération) et la signature (çàd. les paramètres) de l'opération ramasser et définir le contrat sur son résultat.

▻ **Question 4.19** ◅

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

— L'instruction Skip est toujours valide ;

— La garde d'une Conditionelle ou d'une Itération doit être de type booléen ;

— La partie gauche et droite d'une Affectation doivent être de même type.

▻ **Question 4.20** ◅

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

1. Toutes les règles d'un même objectif (à court ou long terme) ont des priorités différentes pour assurer leur déclenchement déterministe.
2. Une réaction de règle ne contient qu'une seule action SeDéplacer.
3. Seules les métarègles peuvent contenir l'action changer.

▻ **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 ssi 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 résulat.

Note : Il faut donc utiliser une instruction *affectation* (cf. Section 3.6) avec la bonne expression en partie droite !

4.3 Mâturité & Modularité

Dans cette section, on teste la robustesse des Diagrammes de Classes définis au terme des Sections précédentes, en introduisant des variantes dans le jeu.

▻ **Question 4.22** ◅

Dans l'état actuel du jeu, une case ne peut pas supperposer 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 ?

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

A Exemple de Carte

Dans cette Annexe sont présentées la carte explicative utilisée dans le sujet, mais sans la grille facilitant le calcul des coordonnées.