Rapport du projet PAF
Dungeon Crawling

Réalisé par :

HERNOUF Mohamed

10 juin 2020

1 L'interface

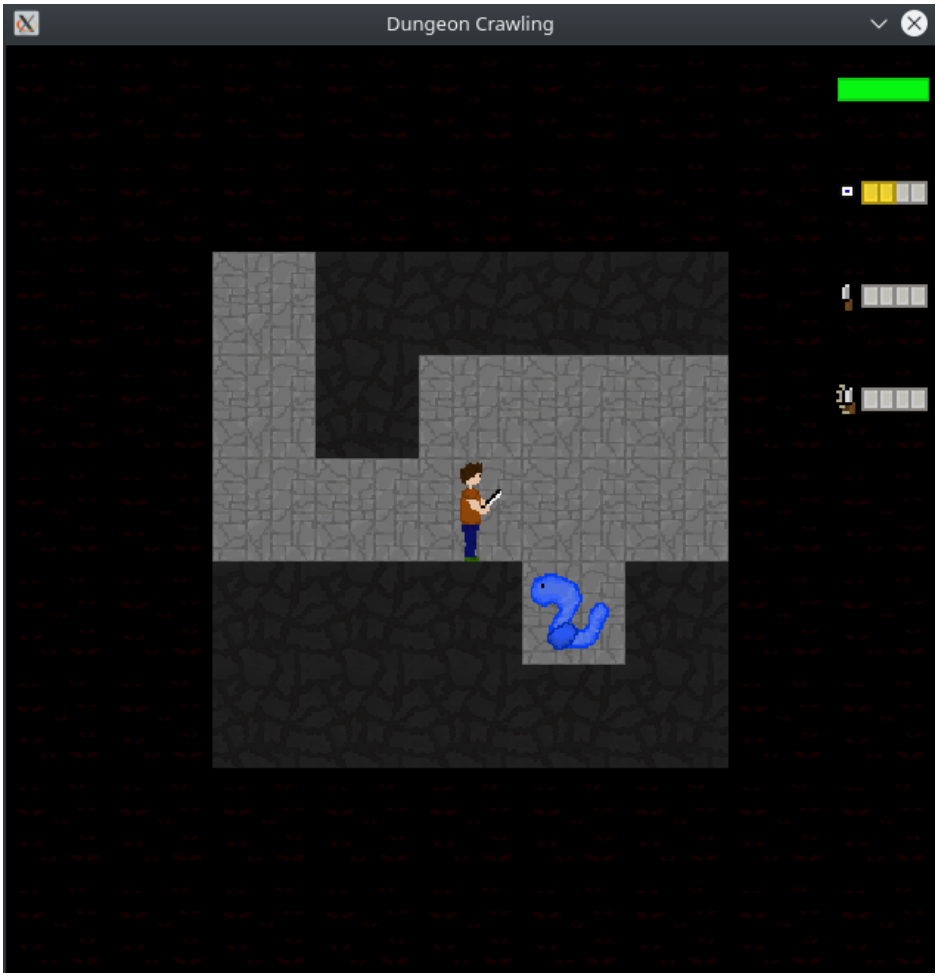

FIGURE 1 – L'interface du jeu

Dans cette section, je vais vous présenter l'interface du jeu, les touches permutants au joueur d'agir et les règles générales du jeu. Le jeu s'affiche sur une fenêtre 630x630 qui permet d'afficher 9 cases en hauteur et 9 cases en largeur au tour du joueur. Les cases qui ne sont pas dans le champ de vision du joueur sont en noir. Tout en haut à droite de la fenêtre se trouve la barre de vie du joueur, qui se met à jour chaque fois quand joueur reçoit un dégât ou obtient un *powerup*. En dessous du barre de vie, se trouvent les indicateurs, permettant de savoir les caractéristiques courants du joueur. En commençant par le haut, ce sont les indicateurs de la vision, du dégât et de la vitesse d'attaque. Indicateur de la vision correspond au nombre de cases visible pour le joueur. Dans la figure 2, ce sont les 2 cases visibles et le restes sont affichées en noir (une sorte de brouillard de la guerre). Indicateur du damage dit quelle est le dégât que joueur peut causer aux monstres. Une trait correspond aux 10 points de dégâts, deux traits correspond aux 10 points de dégâts, etc. Et le dernier indicateur correspond à la vitesse d'attaque. Le joueur possède un *cooldown* (temps de recharge en français) qui doit écouler avant que le joueur puisse effectuer une nouvelle attaque. Voici la correspondance entre l'indicateur et *cooldown* : 0 trait d'indicateur correspond au *cooldown* de 2 seconde, 1 trait - 1.5 seconde, 2 traits - 1.1 seconde, 3 traits - 0.8 seconde, 4 traits - 0.5 seconde.

Joueur peut rencontrer les différents monstres dans labyrinthe. Dans ce jeu il y a 3 types du monstre implémenté :

- Un monstre pacifique, qui se déplace aléatoirement, et ne peut pas attaquer le joueur
- Un monstre agressif, qui commence à chasser rapidement le joueur quand il est dans son champ de vision. Dès qu'il rattrape le joueur, il commence l'attaquer.
- Un monstre neutre, qui se comporte comme un monstre pacifique avant que le joueur lui frappe. Après avoir frappé un monstre neutre, il commence à se comporter comme un monstre agressif.

Quand le joueur tue un monstre, il peut (avec une certaine probabilité) obtenir un *powerup* aléatoire. Le *powerup*

cause la modification d'une de ces caractéristiques. Il y a *powerup* implémenté dans le jeu :

— *Powerup*, qui augmente le nombre des points de vie courant.
— *Powerup*, qui augmente le nombre des points de vie maximale d'un joueur, sans augmenter le nombre des points de vie courant.
— *Powerup*, qui augmente le nombre des points de dégâts.
— *Powerup*, qui augmente la vitesse d'attaque.
— *Powerup*, qui améliore la vision (le plus rare).

Le but du jeu est de passer 5 niveaux du labyrinthe qui sont générés aléatoirement, en ne mourant pas. Chaque niveau du labyrinthe est plus difficile que précédent. Sur le premier niveau, on rencontre que les monstres pacifiques, sur le deuxième niveau - la mélange entre les monstres pacifiques et les monstres neutres, sur le troisième niveau - que les monstres neutres, sur le quatrième niveau - la mélange entre les monstres neutres et les monstres agressives et sur le cinquième niveau que les monstres agressifs.

Ici, je vous présente les touches permettant au joueur d'agir :

— Les touches **Z**, **S**, **Q** et **D** permettant de déplacer le joueur.
— Le touche **Espace** permettant d'ouvrir/fermer la porte.
— Les touches **Up**, **Down**, **Left** et **Right** permettant de frapper la case (éventuellement avec un monstre) à côté.

2 Implémentation

Dans cette section, je vais vous présenter brièvement l'implémentation du jeu, repartie en 11 modules du jeu dans la répertoire `src` et en 10 modules de tests dans le répertoire `test`. Pour commencer, j'aimerais dire que les propositions (post/pré conditions et invariant) et les tests (*hspec* ou *quickcheck*) ont était implémenté pour chaque fonction/type du jeu. Sauf les fonctions des modules Graphics, Keyboard et TextureMap. La raison en est que les modules Keyboard et TextureMap n'ont pas d'intérêt à être testé. Keyboard est un simple ensemble de Keycodes, TextureMap est la correspondance entre la texture et le nom de cette texture. La raison pourquoi module Graphics n'as pas été testé est le fait que tous les fonctions du ce module produisent un IO, ce que n'est pas évident à tester et à faire des propositions dessus. Malgré cela, le projet entièrement est assez bien commenté, donc pour savoir plus les détaille de chaque type / fonction / proposition / test n'hésitez pas à regarder le code source.

2.1 Extensions implémenté

Dans le premier rendu, le jeu ne nous a permis que de se déplacer dans la carte créée "à la main", ouvrir des portes et rencontrer les monstres (au début, il n'y avait que les monstres pacifiques). Aujourd'hui, le jeu a était étendu avec ces extensions :

1. Génération de la carte aléatoire, avec un environnement aléatoire. La carte générée ressemble au maximum à labyrinthe qu'on imagine. En plus, toutes les cases vides sont réunies (il n'existe pas une case vide entouré par des murs dans la carte), et sont accessibles par le joueur. L'entrée est générée quelque part au milieu du carte, et la sortie dans une des 4 coins de la carte. Cela est fait pour que joueur ne sachant pas depuis le début dans quelle direction il puisse avancer. Les portes sont générées de façon efficace (on ne génère pas les deux portes qui sont à coté). L'environnement se génère aléatoirement selon la carte. Les monstres ne sont pas générés à coter du joueur depuis le début. En plus, les monstres sont générés uniformément dans la carte (par exemple, il n'y a pas une partie de la carte avec 4 monstres et l'autre avec 0 monstre). Pour plus de détail, regardez la sous-section Generators.
2. **Statistique du joueur**. On y inclut les affichages des barres et le brouillard de la guerre (les cases visibles au tour du joueur).
3. **Différents types de monstres**. J'ai ajouté ici les monstres agressive et le monstre neutre, ainsi que les mécanismes permettant de déterminer pour ces monstres leur comportement selon l'état du jeu actuelle. En plus, j'ai implémenté l'algorithme de la détection du joueur, l'algorithme permettant de chasser le joueur quand il est dans le champ de vision du monstre et l'algorithme permettant d'accélérer le monstre quand il détecte le joueur et de remettre la vitesse propre à ce monstre quand le joueur s'échappe.
4. **Combat**. Cette extension a été repartie en gestion d'attaque de joueur et en gestion d'attaque du monstre. Dans les deux parties, j'ai travaillé sur l'implémentation du mécanisme permettant de gérer le *cooldown* d'attaque grâce à la liste des événements (cf. sous-section Events) et d'animer cette attaque. Maintenant quand joueur attaque, on pourrait visualiser cela dans l'interface graphique. L'attaque du monstre peut être

vue comme un "saut" rapide sur le joueur. En plus, j'ai implémenté le mécanisme permettant de mettre à jour les points de vie d'entité et de mettre à jour l'environnement si monstre a été tué.

5. **PowerUp**. Maintenant après avoir tuer le monstre, le joueur pourrait obtenir un *powerup* (la probabilité est fixée à 40%). Dans cette extension, j'ai travaillé sur : la génération du *powerup* éventuelle, mise à jour des caractéristiques de joueur, la notification dans l'interface graphique que le *powerup* a été bien obtenu.
6. **Le labyrinthe multi-niveaux**. Ici, on a ajouté les différents monstres reparti en trois types décrit plus haut. En plus, j'ai modifié le générateur d'environnement pour qu'il puisse générer l'environnement selon la difficulté qu'on lui indique. Comme il y a 5 niveaux du labyrinthe, on a décidé d'avoir 5 niveaux de difficultés (très facile, facile, moyenne, difficile, très difficile).

2.2 Modules

Dans le rapport, on va brièvement décrire chaque type/fonction/proposition. La description des tests se trouve dans le fichier du test (ils sont trop particuliers pour le décrire ici). Pour chaque type, on a certains nombres des propriétés et un invariant qui réunit ces propriétés (et éventuellement ajoute les conditions en plus). Pour chaque propriété d'un type et pour chaque invariant, on a un test, qui vérifie les cas intéressants. Si le type instancie un *typeclass* (comme Eq ou Show), il y a une teste pour cette instanciation. Pour chaque fonction je choisis les arguments intéressants (pour tester les différents résultats que fonction pourrait produire) et on teste la pré et la post condition pour chaque choix effectué. Les cartes (module `CarteQuickCheckSpec.hs`), les environnements, les entités et ces fonctions sont testés avec *quickcheck*. La carte est aussi testée avec *hspec* (module `CarteSpec.hs`). J'ai décidé de regrouper la description de chaque module dans une section à part.

2.3 Carte

Dans ce module, on implémente la carte ainsi que les coordonnées de la carte et les cases possibles. Voici les types de ce module :

— **Coord** représentant une coordonnée. Les coordonnées peuvent être comparées entre eux.
— **Case** représentant différents types de cases. Ça peut être une case vide, un mur, une porte, une entrée ou une sortie. La case peut être convertie vers la chaîne de caractères et vice-versa.
— **Carte** représentant la carte d'un niveau du jeu. La carte contient les associations entre coordonnée et sa case associe. La carte pourrait être convertie vers la chaîne de caractère et vice-versa. Elle possède 6 propriétés correspondant à ceux qu'ont été décrit dans le guide. L'invariant du carte vérifie toutes ces propriétés.

Les fonctions du ce module sont :

— **getCase** qui renvoies la case associé aux coordonnées spécifié. Sa pré-condition vérifie que la carte spécifiée dans l'argument respecte son invariant. Sa post-condition vérifie que le résultat est bien ce qu'on a attendu.
— **isTraversable** qui dit si les coordonnées sont traversables (ce n'est pas un mur ou porte fermée). Sa pré-condition vérifie que la carte spécifiée dans l'argument respecte son invariant. Sa post-condition vérifie que le résultat est bien ce qu'on a attendu.
— **editCase** qui permet de modifier une case dans la carte, si la nouvelle carte ne contredit pas l'invariant, sinon l'ancienne carte est retournée. Sa pré-condition vérifie que la carte spécifiée dans l'argument respecte son invariant et que la case passé n'est pas un entré et une sortie. Sa post-condition vérifie que la carte obtenue respecte son invariant et que si les modifications ont été fait alors la nouvelle carte contient bien cette case sur les cordonnées spécifié.
— **actWithDoor** qui ouvre/ferme la porte, qui se trouve aux coordonnées spécifiées, et renvoie la nouvelle carte. Sa pré-condition vérifie que la carte spécifiée dans l'argument respecte son invariant. Sa post-condition vérifie que la carte obtenue respecte son invariant et que si les modifications ont été fait alors la nouvelle carte contient bien une porte modifiée sur les cordonnées spécifié, sinon il n'y a aucune porte.
— **getEntreeCoord** qui renvoie les coordonnées du entrée dans la carte. Sa pré-condition vérifie que la carte spécifiée dans l'argument respecte son invariant. Sa post-condition vérifie que la coordonnée retournée contient bien l'entrée.
— **portesAround** qui prends une coordonnée et renvoi la liste des coordonnées voisins contenant des portes. Sa pré-condition vérifie que la carte spécifiée dans l'argument respecte son invariant. Sa post-condition vérifie que toutes les coordonnées du liste retournée contiennent des portes.

2.4 Entite

Dans ce module, on implémente les entités du jeu, ainsi que leurs statistiques. Voici les types de ce module :

— **Stat** représentant une information sur la statistique d'attaque (points des dégâts, *cooldown* d'attaque et la vision). Stat ne peut pas contenir un damage < 10, une *cooldown* < 0.1 ou une vision < 1. On vérifie cela dans son invariant.
— **Entite** représentant différents types des entités présent dans le jeu. Ça peut être soit un joueur, soit un monstre pacifique, soit un monstre neutre, soit un monstre agressive. Toutes les entités possèdent un identifiant et le nombre de points de vie courant. Les monstres possèdent le nom, le temps du dernier mouvement et le *cooldown* du mouvement. Cela permet de bouger les monstres indépendamment du joueur et des autre montres. Le joueur, les monstres agressifs et les monstres neutres possèdent la statistique d'attaque. Les entités sont comparables selon leur identifiant (deux entités sont égales quand leur identifiant sont égaux.). L'invariant d'entité vérifie chaque type d'entité à part. Pour joueur, l'invariant vérifie que son identifiant est $= 0$ et le nombre des points de vie maximal est > 0 et que le nombre de points de vie courant ne peut pas être supérieur au maximal.

2.5 Environnement

Dans ce module, on implémente l'environnement du jeu. Le type **Environnement** garde en soi la correspondance entre la coordonnée et la liste des entités présentes dessus. Il possède 4 propriétés :

1. Il n'y pas des correspondances avec la liste des entités vide
2. Toutes les entités d'environnement sont différentes (il n'existe pas 2 entités avec le même identifiant.)
3. Toutes les entités d'environnement respectent ses invariants.
4. Toutes les coordonnées d'environnement sont positives.

L'invariant d'environnement vérifie ces propriétés.
Les fonctions du ce module sont :
— **franchissable_env** qui dit si la case est franchissable (il n'existe pas les autres entités en dessus.). Sa pré-condition vérifie que l'environnement respecte son invariant. Sa post-condition vérifie que le résultat est bien ce qu'on a attendu.
— **trouve_id** qui prends un identifiant et environnement, et renvoie l'entité avec cet identifiant et coordonnes de sa case s'il existe, sinon elle renvoie Nothing. Sa pré-condition vérifie que l'environnement respecte son invariant. Sa post-condition vérifie que si le résultat obtenu est Nothing, alors dans l'environnement, ils n'existent pas des coordonnés contenants des entités avec cet identifiant. Sinon dans la liste des entités de cette case, il existe une entité avec un identifiant pareil.
— **rm_env_id** qui retire l'entité avec identifiant spécifié d'environnement. Sa pré-condition vérifie que l'environnement respecte son invariant. Sa post-condition vérifie que nouvel environnement respecte son invariant et que si la modification a été fait alors dans le nouvel environnement il n'y a plus d'entité spécifié, sinon il n'existe pas une entité pareil dans l'environnement.
— **bouge_id** qui déplace une entité dans l'environnement. Sa pré-condition vérifie que l'environnement respecte son invariant et les coordonnées spécifiées ne sont pas négatives. Sa post-condition vérifie que nouvel environnement respecte son invariant et que si la modification a été fait alors entité spécifiée se trouve à la tête du liste des entités du coordonnée spécifiée dans le nouvel environnement, sinon soit entité pareille n'existe pas, soit les coordonnées source et coordonnées destination du déplacement sont égaux, soit les coordonnées destinations ne sont pas franchissable et l'entité n'est pas un joueur (un NPC).
— **get_pos_joueur** qui renvoie la position (coordonnées) du joueur dans l'environnement. Sa pré-condition vérifie que l'environnement respecte son invariant. Sa post-condition vérifie que si le résultat obtenu est Nothing alors tous les entités dans l'environnement sont des mobs (pas de joueur), sinon le résultat est une coordonnée sur laquelle il existe un joueur.
— **info_env** permettant de parcourir l'environnement et séparer le joueur avec tous les monstres. Sa pré-condition vérifie que l'environnement respecte son invariant et qu'il y existe au moins un joueur. Sa post-condition vérifie que l'information produite est à jour avec environnement et vice-versa.
— **remplace** permettant de remplacer une entité dans l'environnement par lui-même (sert pour faire à jour les caractéristiques). Sa pré-condition vérifie que l'environnement et les entités respecte ses invariants et environnement contient bien l'entité qu'on veut remplacer. Sa post-condition vérifie que le nouvel environnement respecte son invariant et nouvelle entité y présent.

2.6 PowerUp

Dans ce module on implémente les *powerups* dont joueur peut obtenir en tuant un monstre. Le type **PowerUp** est une somme de tous les *powerups* possible. Dans ce module, on définit aussi la probabilité d'obtention pour chaque *powerup*. Dedans, on trouve qu'une seul fonction *apply_powerUp*, qui est appelé chaque fois quand le joueur tue un monstre. Elle prend un joueur et lui applique un *powerup*, s'il est généré, et renvoie le joueur avec les nouvelles caractéristiques. Si *powerup* n'a pas été généré elle renvoi Nothing. Sa pré-condition vérifie que l'entité spécifiée respecte son invariant. En plus, ça doit être un joueur. Sa post condition vérifie que si fonction a renvoyé une entité alors une de ces caractéristiques ont été augmentés.

Dans ce module, on implémente les événements du jeu. Cette information sert pour le module Graphics, pour afficher les notifications, ou animer les attaques du joueur/monstre. L'événement aide au jeu de compter le temps écoulé depuis la dernière attaque du mob/joueur, notamment, pour la gestion de *cooldown* d'attaque. Le type **GameEvent** représente un événement du jeu. Chaque événement garde en soi le temps quand cet événement a été produit. Il existe 3 types d'événements :

1. **JoueurAttack**, représentant l'attaque du joueur. Ce type d'événement garde en plus la direction d'attaque.
2. **MonsterAttack**, représentant l'attaque du monstre. Ce type d'événement garde l'identifiant du monstre et la direction d'attaque.
3. **PowerUpEvent**, représentant l'obtention du *powerup* par joueur.

L'invariant d'événement vérifie que le temps du début d'événement n'est pas négatif et MonsterAttack ne possède pas l'identifiant inférieure à 1.

Ce module possède une seul fonction **filterEvents** permettant de filtrer la liste des événements et ne garder que les événements récents. Chaque événement a la durée pendant laquelle il peut rester dans la liste des événements. Pour PowerUpEvent, c'est 2 secondes (pour pouvoir afficher PowerUp dans l'interface graphique du jeu). Pour un événement d'attaque du joueur/monstre, c'est *cooldown* de ce joueur/monstre (pour interdire à cette entité d'attaquer avant que son *cooldown* ne soit pas écoulé). Sa pré-condition vérifie ces conditions :

1. Le temps courant doit être supérieure au 0
2. Tous les événements de liste doivent respecter ces invariants
3. Pour chaque joueur/monstre il y a au plus un événement d'attaque
4. L'entité passé doit être un joueur
5. L'ensemble des entités spécifié doit contenir que des monstres

Sa post-condition vérifie ces conditions :

1. Tous les événements de liste doivent respecter ces invariants
2. Pour chaque joueur/monstre il y a au plus un événement d'attaque
3. Il n'y a pas des événements des monstres morts
4. Il n'y a pas des événements anciens (ceux dont la durée est écoulée)

2.7 Modèle

Dans ce module, on implémente le modèle du jeu ainsi que les ordres appliqués aux monstres. Voici les types de ce module :

- **Ordre** représentant des ordres qu'on utilise pour générer le déplacement aléatoire des mobs. Cela est utilisé pour les monstres pacifiques, et les autres types des monstres quand ils ne s'agressent pas sur le joueur. Il existe 5 ordres : Rien_faire, Aller_Nord, Aller_Sud, Aller_Ouest, Aller_Nord.
- **Modèle** représentant le modèle du jeu. Réunit en soi la carte, environnement, générateur standard (pour générer les mouvements des mobs), l'état du clavier (pour pouvoir déplacer le joueur dans la bonne direction), dernière touche tapée pour se déplacer ou pour frapper (pour pouvoir graphiquement dessiner dans quelle direction regarde le joueur), et la liste des événements permettant de gérer de façon efficace le *cooldown* d'attaque des entités. Le modèle possède 2 propriétés :
  - L'environnement n'est pas en contradiction avec la carte. Toutes les cases où se trouvent les entités sont traversables (les coordonnées existantes, pas d'entités dans les murs, dans les portes fermées).
  - Chaque événement de la liste des événements du modèle doit respecter son invariant, et pour chaque entité ne contenir qu'un seul événement d'attaque. Le temps de début du chaque événement ne peut pas être négatif.

L'invariant vérifie ces propriétés et en plus vérifie que la carte et l'environnement respecte ses invariants.
Les fonctions du ce module sont :

— **bouge** qui bouge une entité au sein d'un modèle. Sa pré-condition vérifie que modèle spécifié respecte son invariant et les coordonnées spécifié sont traversables sur la carte. Sa post-condition vérifie que nouveau modèle respecte son invariant et si la modification a été fait alors l'entité spécifié se trouve à la tête du liste des entités du coordonnes spécifié dans nouvelle environnement, sinon soit entité pareille n'existe pas, soit la coordonnées source et coordonnées destination du déplacement sont égaux, soit la coordonnées destination n'est pas franchissable et l'entité n'est pas un joueur (un NPC), soit les coordonnées ne sont pas traversables.

— **decide** qui permet de choisir quel ordre on va appliquer sur un mob, quand il ne s'agresse pas au joueur (ou quand le mob est pacifique). Cette fonction prend la liste pondérée des ordres (les poids doivent être $>$ 0), tire un ordre au hasard du cette liste et applique cet ordre sur mob spécifié. Sa pré-condition dit que le modèle doit respecter son invariant, entité doit exister dans l'environnement, la liste pondérée ne doit pas être vide, les poids des ordres doivent être strictement positive et l'ordre Rien_faire doit être présent dans la liste. Sa post-condition vérifie que le modèle résultat respecte son invariant. En plus, il construit la liste des coordonnées à partir d'ancien coordonnes et la liste des ordres, représentant toutes les positions possibles d'entité dans le nouveau modèle, et vérifie que les nouvelles coordonnées d'entité sont présent dans la liste des positions possibles.

— **prevoit** qui génère la liste pondérée des ordres possibles pour une entité dans le modèle. Sa pré-condition vérifie que modèle respecte son invariant, entité existe dans l'environnement et entité n'est pas un joueur. Sa post-condition vérifie que la liste obtenue respecte les conditions suivantes :

1. Premier élément du liste doit être un ordre "Rien faire" avec un poids égale au nombre de déplacements possibles (ou 1 si aucun déplacement n'est possible). Cela nous dit que la probabilité d'obtenir un ordre Rien_faire pour un mob est toujours 50% (sauf dans les cas quand le mob est coincé).
2. Pour chaque élément de la liste à partir du deuxième élément : le poids du déplacement est égal à 1 et la coordonnée du déplacement doit être traversable et franchissable.

— **tour** qui prend un modèle et entité et applique une action sur cette entité en produisant le nouveau modèle. Si l'entité est un joueur alors l'action à appliquer est choisi selon la touche du clavier pressé (champ 'keyboard' du modèle permet de savoir ça). Les actions possibles pour joueur sont : se déplacer, frapper une case (potentiellement avec un monstre) à coté, utiliser (ouvrir/fermer la porte) et rien faire. Si l'entité est un NPC (non jouer) alors plusieurs réglés d'application sont possibles, selon le type de monstre et selon l'état d'environnement. Comme c'est une fonction centrale du jeu, il se repousse sur 150 lignes, donc je vous invite à lire les commentaires qui détaillent très bien le comportement de cette fonction. Sa pré-condition vérifie que modèle respecte son invariant et que si l'entité est un jouer alors il doit exister dans l'environnement. Sa post-condition vérifie que nouvelle modèle respecte son invariant et effectue plusieurs testes (sur 120 lignes) de ce qu'il attend de nouveau modèle en fonction du type d'entité, d'état du clavier, d'environnement actuelle, des événements, des *cooldown* des monstres, etc.

— **level_finished** qui dit si le niveau est terminé. Le niveau est terminé quand le joueur est sur la case du sortie. Sa pré-condition vérifie que le modèle respecte son invariant. Sa post-condition vérifie que le résultat est bien ce qu'on a attendu.

— **perte** qui dit si le modèle est en situation perdante. La situation est perdante quand le joueur ne possède plus des points de vie. Sa pré-condition vérifie que le modèle respecte son invariant. Sa post-condition vérifie que le résultat est bien ce qu'on a attendu.

— **frapper** permettant à l'entité de frapper la case à côté (ou aussi la même case dans le cas du monstre). Sa pré-condition vérifie que l'entité spécifiée n'est pas un monstre pacifique, modèle et l'entité doivent respecter ses invariants, la case à frapper doit être traversable et doit être à coté de l'entité spécifiée. Sa post-condition vérifie les conditions suivantes :

1. Si le joueur frappe une case sur laquelle se trouve un monstre alors la mis à jour de points de vie du ce monstre est effectué. Si joueur tue un monstre alors l'environnement a bien été mis à jour.
2. Sa pré-condition vérifie que l'entité spécifiée n'est pas un monstre pacifique, modèle et l'entité doivent respecter ses invariants, la case à frapper doit être traversable et doit être à coté de l'entité spécifiée. Le monstre ne peut pas frapper les autres monstres et ne peut pas attaquer la case vide.

2.8 Etat

Dans ce module, on implémente l'état du niveau. L'état réunit en soi le numéro du niveau, le modèle du niveau, le joueur, et l'ensemble des monstres de niveau. Il y a trois états possible :

— **Tour**, représentant l'état avec modèle qui est en cours du traitement (la situation n'est ni gagnant ni perdante.)
— **Gagne** représentant le niveau qui a été passé (joueur s'est déplacé vers la sortie.)
— **Perdu**, indiquant que le joueur ne possède plus des points de vie.

L'état possède propriétés qui vérifie si tous les monstres et joueur dans les champs 'monstres' et 'joueur' existent dans l'environnement et inversement (toutes les entités d'environnement sont présentes dans ces champs). L'invariant d'état vérifie cette propriété et en plus vérifie ces conditions :

1. Le numéro du niveau doit être strictement positive
2. Le champs 'joueur' doit contenir un joueur
3. Joueur respecte son invariant
4. Modèle d'état doit respecter son invariant
5. Toutes les monstres doivent respecter ses invariants.
6. Toutes les entités du champ 'monstre' doivent être un NPC (pas de joueur)

Ce module possède une seule fonction qui est **etat_tour**. C'est une autre fonction centrale du jeu, qui prend le temps actuel, l'état actuel, l'état du clavier actuel et calcule le nouvel état du jeu. Le nouvel état est obtenu après avoir calculé la nouvelle modèle du jeu en appelant la fonction 'tour' sur chaque entité du jeu (toutes les entités sont présentant dans les champs 'joueur' et 'monstres'). Après avoir calculé le modèle, la fonction teste si le modèle est en situation gagnante pour le niveau actuel. Si une de ces tests donne le résultat positif alors fonction retourne état correspondant (Gagne ou Perdu). Sinon elle construit le nouvel état Tour avec tous les mises a jours prises en compte. À chaque tour, la liste des événements du modèle est filtrée selon le temps actuelle. Sa pré-condition ne permet pas d'appeler cette fonction pour l'état Gagne ou Perdu. En plus, elle vérifie si l'état spécifié respecte bien l'invariant. Sa post-condition vérifie que l'état produit respecte son invariant et vérifie les règles suivantes :

1. Si l'état est Gagne alors, si on appelle la fonction tour sur le joueur avec le clavier spécifié et le modèle d'ancien état, alors la nouvelle modèle est bien en situation gagnante
2. Si l'état est Perdu alors le modèle est en situation perdante
3. Si l'état est Tour alors le modèle de cet état n'est ni en situation gagnante ni en situation perdante

2.9 Generators

Dans ce module, on définit les générateurs des cartes et d'environnements, ainsi que les fonctions auxiliaires qui correspondent aux différentes étapes du génération. Dans le jeu, on utilise les cartes qui ont une largeur et une hauteur maximale égale à 25. Pour la génération du carte, on utilise la notion du group. C'est sont toutes les cases du même nature réunit ensemble.

La génération du carte se passe en 7 étapes (donc en appelant 7 fonctions consécutivement) :

— Appel de la fonction **create_bordure** qui gère la création des bordures en tour de la carte, on lui passe hauteur et largeur aléatoires. La fonction renvoie l'association entre les coordonnées et une case correspondante. Sa pré-condition vérifie que la hauteur et largeur sont supérieures strictement à 2. Sa post-condition vérifie que le *map* obtenu contient toutes les coordonnées des bordures et leur case associés sont des mures.
— Appel de la fonction **fill_randomly** qui prend le résultat d'étape 1, et remplit aléatoirement les coordonnes centrales soit par une case normale soit par un mur. Sa pré-condition vérifie que le *map* a bien passé l'étape 1. Sa post-condition vérifie que toutes les cases centrales présentent dans la *map* et que ces cases sont soient des murs soient des cases normaux.
— Appel de la fonction **fix_walls** permettant de réunir tous les murs en groupes de 4 ou plus, de façon aléatoire. Sa pré-condition vérifie que la *map* a bien passé l'étape 2. Sa post-condition vérifie qu'il n'existe pas des groupes de murs avec le nombre d'éléments < 4.
— Appel de la fonction **join_ways** permettant de réunir toutes les cases normales du *map*, de façon aléatoire. Sa pré-condition vérifie que le *map* a bien passé l'étape 3. Sa post-condition vérifie qu'il existe un seul group de cases normales (ou bien que à partir du n'importe quelle case normale on pourrait accéder à toutes les autres cases normales du *map*).
— Appel de la fonction generate_doors permettant de générer des portes à la place de certaines cases normaux. Les portes sont générés avec la distante minimal entre eux qui est égale à 3. Les portes nord-sud sont générées dans les cases où les cases voisines au nord et au sud sont les cases normales et les cases voisins à l'est et à l'ouest sont des murs. Idem. pour les portes est-ouest. Sa pré-condition vérifie que la *map* a bien passé l'étape 4. Sa post-condition vérifie que les règles décrit tout à l'heure ont été bien respecté.

— Appel de la fonction **generate_entry** qui génère l'entrée à la place d'une case normale la plus centrale. Sa pré-condition vérifie que la *map* a bien passé l'étape 5, et qu'il existe au moins une case normal dans la *map*. Sa post-condition vérifie que l'entrée a bien été placée quelque part.
— Appel de la fonction **generate_exit** qui génère la sortie à la place du case normale de coins aléatoires. Sa pré-condition vérifie que la *map* a bien passé l'étape 6, et qu'il existe au moins une case normal dans la *map*. Sa post-condition vérifie que la sortie a bien été placée quelque part.

Ces fonctions sont appelées dans la fonction **genCarte**, qui encapsule le *map* obtenu après l'étape 7 et l'encapsule dans la carte. Sa pré-condition vérifie que la largeur et hauteur sont supérieures à 10 (pour être sûr que le générateur nous génère une *map* contenant ou moins 2 cases normales pour l'entrée et pour la sortie). Sa post-condition vérifie que la carte construite respecte son invariant.

La génération d'environnement se passe juste après la génération de la carte, on y ajoute le joueur à la case d'entrée et on génère les monstres sur des cases normales avec la distance minimal entre chaque monstre qui est égale à 5. Les monstres sont générés selon la difficulté spécifiée dans l'argument. L'environnement généré à la fin respecte son invariant. En plus, il correspond à la carte, qui a été généré.