

Silent Strike

BENKHALED Mohammed lyad

EDDOUSSI Mohamed

SEMRAOUI Souheil

LY Benjamin

05/12/202

4

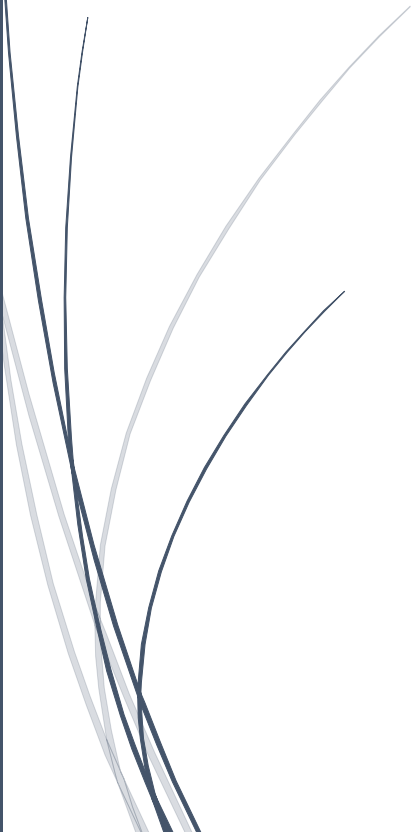
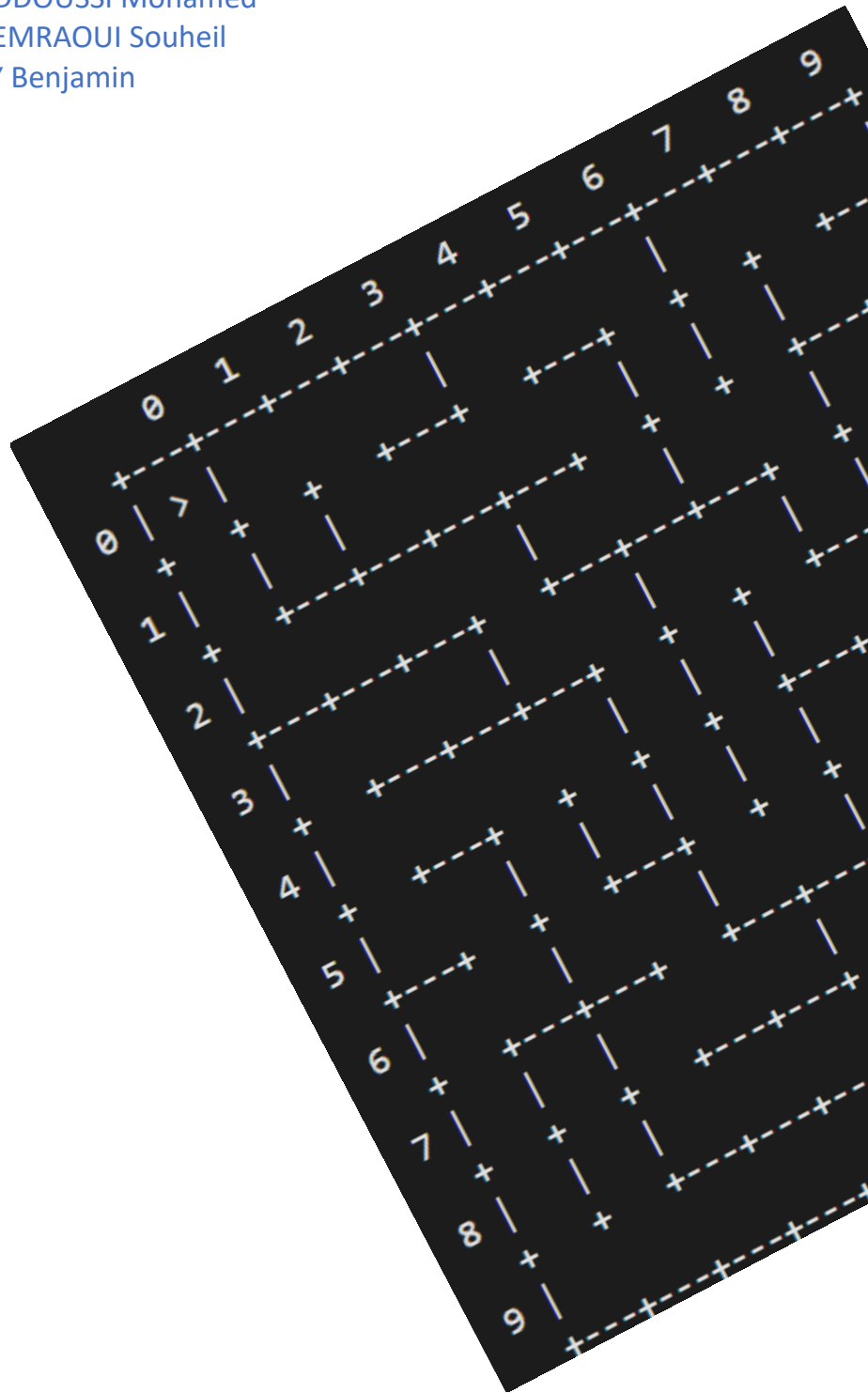


Table des matières

1) REGLES DU JEU	2
1) Commandes	2
2) Règles	3
2) CONCEPTION	4
2.1) Conception des ennemis.....	4
2.3) Conception du joueur	6
3) CODE.....	7
3.1) Code en C#	7
3.2.1) La classe Game:.....	7
3.2.2) Les 3 classes Person, Player et Ennemi:	8
3.2.1) La classe Maze et la classe Node:.....	8
3.2.2) La classe Graphics:	9
3.2.2) La classe XMLUtils:	9
3.2) Code en XML, XSD, HTML et XSLT	11
4) Les Diagrammes	14
4.1) Diagramme UML	14
4.2) Diagramme de Séquence	14

1) REGLES DU JEU

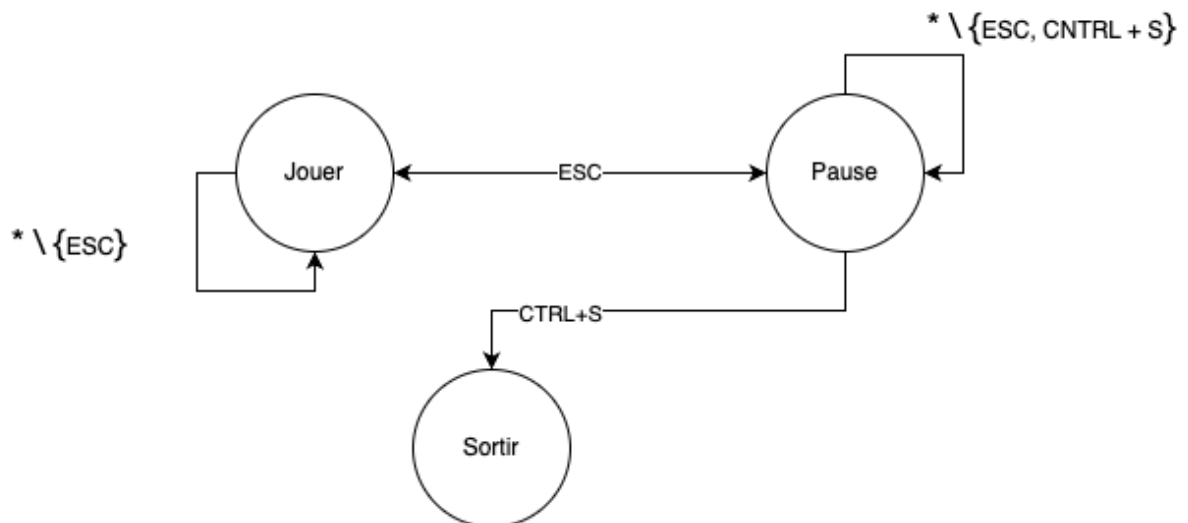
Il s'agit d'un jeu en terminal où l'objectif est de sortir du labyrinthe. Pour y parvenir, il est impératif de tuer tous les ennemis. Une fois cette étape accomplie, une porte s'ouvrira dans la dernière case, permettant ainsi de sortir.

1) Commandes

Pour changer uniquement l'orientation, il suffit d'appuyer sur les touches « ZQSD », alors que pour avoir l'option de se déplacer en plus il faut appuyer sur « shift ». La touche pour tirer sur les ennemis est « P ».

Pour mettre le jeu en pause et le reprendre ultérieurement il faut appuyer sur « échap - ESC ». Pour créer une sauvegarde du jeu en cours il faut faire « Ctrl + S » lors de la pause.

L'automate ci-dessous explique comment les touches influence l'état du jeu.



2) Règles

L'objectif est de tuer tous les gardes par derrière ou sur les côtés, ils ne peuvent nous voir que si nous sommes directement en face d'eux et il n'y a pas de mur entre nous et le garde. De plus la sortie est toujours en bas à droite du terrain qui sera toujours définie par l'absence d'un mur.

- Conditions de victoire :
 - Tuer tous les ennemis,
 - Trouver la sortie,
 - Ne pas mourir car pas de vie supplémentaire disponible.
- Condition de mort :
 - Rentrer dans le champ de vision des gardes.

Attention : Si on trouve la sortie sans tuer les gardes, le jeu ne se termine pas. Il faut obligatoirement tous les éliminer.

La taille du labyrinthe n'est pas prédéfinie, chaque partie se déroule dans un nouveau labyrinthe. Il y a un algorithme qui génère le labyrinthe au début de chaque partie. La logique de cet algorithme est la suivante:

Choisissez la cellule initiale, marquez-la comme visitée et ajoutez-la à la pile.
Tant que la pile n'est pas vide :

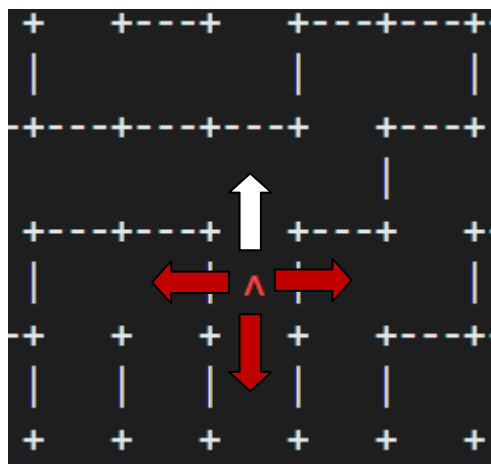
- Retirez une cellule de la pile et faites-en la cellule courante.
- Si la cellule courante a des voisins non visités :
 - Ajoutez la cellule courante à la pile.
 - Choisissez un des voisins non visités.
 - Enlevez le mur entre la cellule courante et la cellule choisie.
 - Marquez la cellule choisie comme visitée et ajoutez-la à la pile.

On le trouve implémenté dans la classe Maze, c'est la fonction generate().

2) CONCEPTION

2.1) Conception des ennemis

Les gardes ont un champ de vision rectiligne.



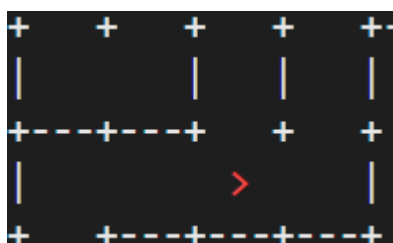
On peut donc être derrière ou à côté sans se faire remarquer.

Attention : On meurt directement une fois dans le champ de vision du garde, ou si on est dans la même case que lui.

Le garde va d'une manière prédéfinie choisir d'aller à gauche, tout droit ou à droite. S'il n'y a pas d'obstacle à gauche, il choisira d'abord d'aller à gauche, s'il y a un obstacle il ira tout droit, et si c'est bloqué, il ira à droite. Dans le cas d'un cul de sac, il fait demi-tour après s'être cogné dans un mur à chaque orientation.

Exemple :

Ici le garde ira à gauche bien qu'il puisse continuer à avancer tout droit.



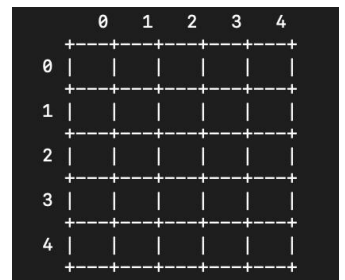
Information complémentaire :

- Ici le seul objet avec lequel on peut rentrer en contact est le mur. C'est-à-dire que les gardes peuvent se passer dessus et continuer leurs chemins.
- Si le joueur et le garde sont sur la même case car le joueur est allé trop vite en arrivant par derrière, alors le joueur est considéré comme mort.
- Les gardes ne peuvent pas apparaître, au debut de la partie, dans un certain rayon par rapport au joueur, sinon la partie peut se terminer avant même de commencer.

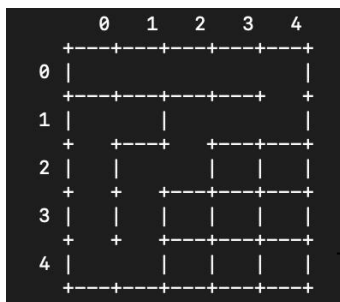
2.2) Conception des murs

- Etape 1 :

Pour l'affichage du terrain on commence avec un terrain complet.



	0	1	2	3	4
0					
1					
2					
3					
4					



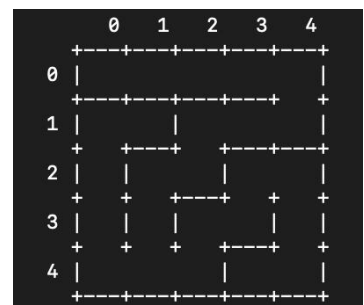
	0	1	2	3	4
0					
1					
2					
3					
4					

- Etape 2 :

L'algorithme de conception du labyrinthe se met en route pour tracer le labyrinthe.

- Etape 3 :

On obtient un labyrinthe complet avec le point de départ et un chemin vers la sortie. On fera apparaître les gardes à des endroits aléatoires juste après.



	0	1	2	3	4
0					
1					
2					
3					
4					

2.3) Conception du joueur

Le joueur apparaît toujours à la case (0.0) et le labyrinthe commence toujours sa création ici.

Si le joueur entre en collision avec un mur, il ne pourra pas bouger jusqu'à changement de l'orientation.

C'est la même dynamique que pour les ennemis sauf que les mouvements sont automatisés.

3) CODE

3.1) Code en C#

3.2.1) La classe Game:

C'est le coeur de notre jeu. La classe qui retient les information sur les jouer, le labyrinthe et les ennemies. C'est aussi la classe qui mis a jour les information et permet au ennemi de bouger automatiquement, et d'appeler fonction graphics qui permet d'afficher le jeux. Aussi, c'est la classe qui interagit avec le joueur.

La gestion simultanée des mises à jour de l'état du jeu (déplacement des ennemis, vérification des conditions de victoire/défaite) et des entrées utilisateur est délicate. Le jeu doit rester réactif tout en respectant un certain rythme (délai entre les mises à jour).

- La méthode **Update** doit vérifier en boucle les conditions de victoire/défaite, gérer les déplacements des ennemis, et déverrouiller la sortie si tous les ennemis sont tués.
- La méthode **Run** gère une boucle infinie pour capturer les entrées utilisateur sans interrompre **Update**.
- La méthode **Show**, qui s'exécute sur un autre thread et affiche le jeu toutes les 10 ms, a été mise en place pour garantir une expérience fluide.

Synchroniser ces deux tâches sans créer de conflits dans l'état du jeu a été un défi, car nous n'avions jamais fait la programmation concurrente. Nous l'avons découverte a travers notre projet.

Solution :

- Utilisation de **async** et de **Task.Delay** pour gérer les mises à jour du jeu en arrière-plan.

3.2.2) Les 3 classes Person, Player et Ennemi:

- La classe Person:

La classe Person est une classe abstraite qui sert de base pour les entités mobiles du jeu, telles que les joueurs et les ennemis. Elle gère les coordonnées (X, Y) de la personne, ainsi que son orientation dans le labyrinthe.

Gestion des déplacements : La méthode Move est responsable du déplacement en vérifiant si la cellule dans la direction donnée (selon l'orientation) est accessible. Si la cellule est libre (c'est-à-dire qu'il n'y a pas de mur), la personne se déplace.

Affichage de la personne : La méthode ToString retourne un caractère représentant l'orientation de la personne, ce qui permet de l'afficher visuellement dans le jeu.

- La classe Player (Joueur) :

La classe Player hérite de la classe Person et représente le joueur dans le jeu. Elle utilise les mêmes mécanismes que la classe Person pour gérer les coordonnées et l'orientation, mais elle ne comporte pas de logique spécifique autre que celle héritée de la classe de base.

- La classe Enemy (Garde) :

La classe Enemy représente un ennemi dans le jeu. Elle hérite de la classe Person et ajoute une propriété booléenne, Alive, pour suivre si l'ennemi est vivant ou non. L'ennemi peut être tué grâce à la méthode Kill, qui met à jour son état de vie à false. La méthode Move permet à l'ennemi de se déplacer sur le labyrinthe, avec les règles décrites dans la partie 2.1 conception des ennemis.

3.2.1) La classe Maze et la classe Node:

- La classe Maze :

La classe Maze représente le labyrinthe dans lequel le joueur et les ennemis évoluent. Elle est responsable de la génération, de la gestion des dimensions, et de la structure interne du labyrinthe. Le labyrinthe est constitué d'un tableau de nœuds (Node), chaque nœud représentant une cellule du labyrinthe avec ses coordonnées et ses murs.

La classe dispose de deux propriétés, Width et Height, qui définissent respectivement la largeur et la hauteur du labyrinthe. Ces dimensions sont contraintes pour

être strictement positives et inférieures à 40 (c'est choix que nous avons fait car après ça devient trop grand pour l'afficher dans le terminal).

- La classe Node :

La classe Node représente une cellule du labyrinthe. Chaque nœud a des coordonnées (x, y) et quatre murs (haut, droite, bas, gauche) qui peuvent être activés ou désactivés pour créer des passages entre les cellules.

Les propriétés X et Y définissent les coordonnées du nœud dans le labyrinthe. Ces coordonnées sont vérifiées pour s'assurer qu'elles sont positives.

Chaque nœud possède des murs définis par les propriétés booléennes LEFT, TOP, RIGHT, et BOTTOM, qui indiquent si un mur existe dans la direction correspondante. Par défaut, tous les murs sont actifs (true), mais ils peuvent être modifiés lorsque des passages sont créés entre les nœuds voisins (la génération du labyrinthe).

3.2.2) La classe Graphics:

La classe Graphics est responsable de l'affichage graphique du jeu dans le terminal. Elle fournit des méthodes statiques pour afficher l'état actuel du labyrinthe, les entités, c'est à dire le joueur et les ennemis. On l'utilise aussi pour afficher des erreurs dans le cas des exceptions.

La méthode principale de la classe, Print, est utilisée pour afficher le labyrinthe à chaque mise à jour. Elle commence par effacer l'écran avec Console.Clear et affiche un message personnalisé passé en paramètre (text). Ensuite, elle itère sur chaque cellule du labyrinthe pour afficher les murs et les entités qui se trouvent dans chaque case.

L'affichage des murs est conditionné par la présence ou l'absence de murs dans chaque cellule, et les entités sont affichées à leur position respective. Le joueur est affiché avec son orientation actuelle, et les ennemis sont affichés en rouge s'ils sont vivants, sinon ils sont marqués d'un point (.) si ils sont morts (du sang après leur mort).

Affichage des jeux sauvegardés : La méthode PrintSavedGamesTable est utilisée pour afficher une liste des jeux sauvegardés dans un répertoire donné. Elle prend un tableau de fichiers et affiche un tableau avec l'index et le nom du fichier pour chaque jeu sauvegardé.

Gestion des erreurs : La méthode PrintError permet d'afficher un message d'erreur en rouge dans la console. Cela permet de signaler des problèmes ou des erreurs dans le jeu de manière claire et visible. Elle est utilisée surtout dans l'exception-handling pour afficher des messages à l'utilisateur lorsqu'il entre des valeurs fausses ou lorsqu'une erreur arrive.

3.2.2) La classe XMLUtils:

La classe XMLUtils gère la sérialisation, la validation, la conversion et la manipulation des fichiers XML relatifs au jeu. Elle fournit des méthodes pour sauvegarder et charger les données du jeu, valider les fichiers XML contre un schéma XSD, convertir les fichiers XML en HTML et gérer un menu des jeux sauvegardés.

Cette classe est capitale pour le jeu, car c'est ici que nous utilisons les technologies apprises dans le module. Nous allons donc passer plus de temps à l'expliquer en détail.

1. La méthode **Save(Game game, string? filename)**

- **But** : Sérialise un objet Game (représentant l'état du jeu) dans un fichier XML.

- **Détails** :

- Crée un fichier XML contenant les données du jeu et le sauvegarde dans un répertoire spécifié (./data/saved_games/).

- Utilise XmlSerializer pour sérialiser l'objet du jeu dans le fichier.

- Après avoir sauvegardé le fichier XML, la méthode appelle **RunXSLT** pour convertir l'XML qu'on vient de créer en fichier HTML en utilisant une transformation XSLT, créant ainsi une version HTML du jeu sauvegardé.

2. La méthode **Load(string filename):**

- **But** : Charge un jeu sauvegardé à partir d'un fichier XML et le désérialise.

- **Détails** :

- Valide le fichier XML avec le schéma XSD en utilisant la méthode **Validate** pour vérifier qu'il n'est pas corrompu.

- Désérialise le fichier en un objet **Game** à l'aide de **XmlSerializer**.

- Définit l'état du jeu sur **Pause** et appelle les méthodes Update(), Show() et Run() pour mettre à jour et exécuter le jeu.

3. La méthode **Validate(string XSDFilePath, string XMLFile):**

- **But** : Valide un fichier XML contre un schéma XSD.

- **Détails** :

- Charge le schéma XSD et utilise **XmlReader** pour valider le fichier XML.

- Lève une exception avec un message d'erreur si le fichier est corrompu ou ne respecte pas le schéma.

4. La méthode **ConvertXMLtoHTML(string XSDFilePath, string xmlFilePath, string outputHtmlPath):**

- **But** : Convertit un fichier XML validé en un fichier HTML représentant l'état du jeu. Elle fait le travail qu'un script XSLT fait. Nous avons le choix entre utiliser cette fonction pour convertir les fichier XML qui sauvegarde l'état d'un jeu et ou d'utiliser

XSLT. Ici on emploie le DOM pour lire le document XML et créer un HTML en meme temp.

- **Détails :**

- Valide le fichier XML en utilisant le schéma XSD. Pour être certain qu'on est entrain de convertir un fichier non corrompu.
- Charge le fichier XML à l'aide de **XmlDocument** et extrait les informations sur le joueur, les ennemis et la grille.
- Génère un fichier HTML qui représente visuellement l'état du jeu avec des classes CSS pour les murs, le joueur et les ennemis.
- Écrit le HTML généré dans le fichier de sortie spécifié.

5. La méthode `RunXSLT(string xsltFilePath, string xsdFilePath, string xmlFilePath, string outputHtmlPath)`:

- **But :** Applique une transformation XSLT pour convertir un fichier XML en un fichier HTML.

- **Détails :**

- Valide d'abord le fichier XML à l'aide du schéma XSD fourni. Encore une fois, avant de faire quoi que se soit avec nos fichier XML on vérifie qu'il sont conforme a la définition du schéma.
- Charge le fichier XSLT et applique la transformation pour générer le fichier HTML à l'emplacement spécifié.

6. La méthode `UpdateSavedGamesMenu(string savedGamesDirectory, string outputFilePath)`:

- **But :** Génère un fichier XML contenant une liste de tous les jeux sauvegardés dans un répertoire spécifié, ici on lit le contenu d'un répertoire et on l'écrit dans un fichier xml en utilisant le DOM.

- **Détails :**

- Collecte des informations sur chaque jeu sauvegardé, y compris le nom du fichier, la date de dernière modification et le chemin du fichier HTML correspondant.
- Crée un document XML avec ces informations et le sauvegarde dans le fichier de sortie spécifié.

3.2) Code en XML, XSD, HTML et XSLT

Nous avons deux types de fichiers, un qui sert a définir les jeux sauvegardé, définit avec le schema saved_game.xsd. Il est important dans la serialisation de la classe game. Il définit la structure des données utilisées pour représenter un jeu sauvegardé, essentiel pour la sérialisation et la désérialisation de la classe Game. Ce schéma décrit l'organisation

des éléments clés du jeu, comme l'état actuel, les informations sur le joueur, les ennemis et la configuration du labyrinthe. L'élément racine, `Game`, regroupe toutes ces informations et s'assure que chaque section respecte un format précis. Par exemple, le joueur est caractérisé par ses coordonnées et son orientation, tandis que les ennemis sont décrits individuellement avec leur position, leur orientation et leur état (vivant ou non). Le labyrinthe est structuré en une séquence de lignes (rows) qui sont eux même compose d'une sequence de nœuds, où chaque nœud contient des informations sur la présence de murs autour de lui. En plus de cela, le schéma impose une validation stricte pour garantir la cohérence des données, comme les états possibles du jeu (Over, Paused, Running). Cette définition claire et précise permet de transformer un objet complexe en un fichier XML lisible.

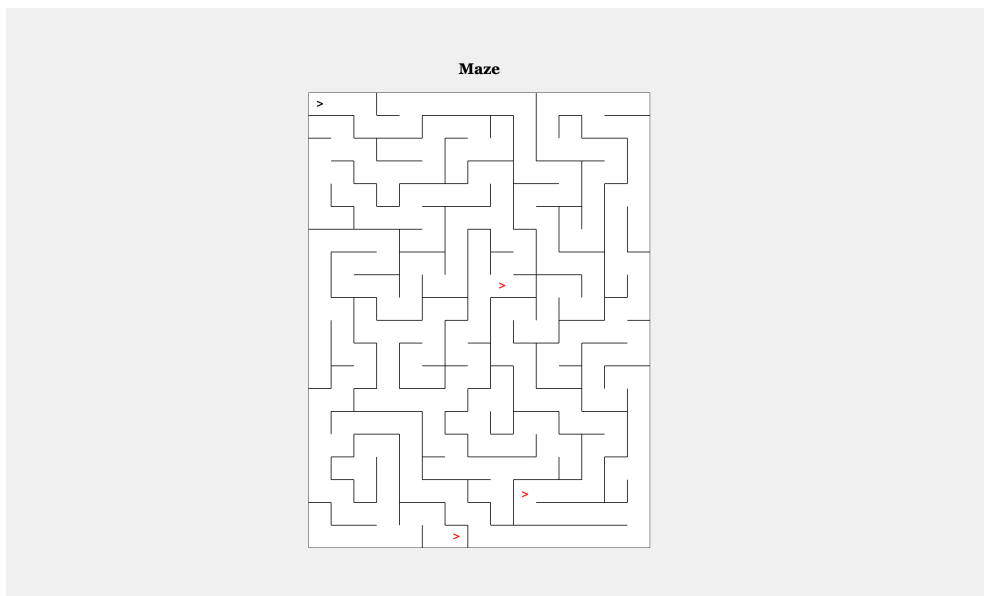
L'autre fichier sert à stocker les informations sur les jeux sauvegardés, que nous avons nommé **menu.xml** . Le fichier `menu.xsd` définit le schéma de ces données et se trouve dans le répertoire `data/xsd`. Par ailleurs, le fichier `menu.xml` contient les informations concrètes sur les jeux sauvegardés.

Un fichier XSLT, nommé `convertMenuToHTML.xslt`, permet de transformer ce menu en une page HTML que l'utilisateur peut ouvrir pour visualiser tous ses jeux sauvegardés, accompagnés de la date de leur dernière sauvegarde, c'est-à-dire la dernière fois qu'ils ont été joués. Cette page HTML offre également la possibilité de cliquer sur un jeu pour accéder à ses détails et voir dans quel état il se trouve, comme illustré ci-dessous.

Pour voir le menu en HTML il faut aller dans le répertoire *data*, si vous avez au moins une partie sauvegardé il devrait apparaître sous le nom *menu.html*.

Saved Games	
1.	smallGame.xml Last modified on: 17-12-2024 on 14:47
2.	mygame15x15.xml Last modified on: 17-12-2024 on 14:41
3.	game10101.xml Last modified on: 17-12-2024 on 14:45
4.	notSoBig.xml Last modified on: 17-12-2024 on 14:46
5.	game1.xml Last modified on: 17-12-2024 on 14:22
6.	weirdGame.xml Last modified on: 18-12-2024 on 17:34
7.	gg.xml Last modified on: 17-12-2024 on 14:21
8.	bigGame.xml Last modified on: 18-12-2024 on 17:35
9.	game111.xml Last modified on: 18-12-2024 on 17:34
10.	10x5.xml Last modified on: 18-12-2024 on 17:33
11.	mediumGame.xml Last modified on: 17-12-2024 on 14:48

Le menu des jeux.

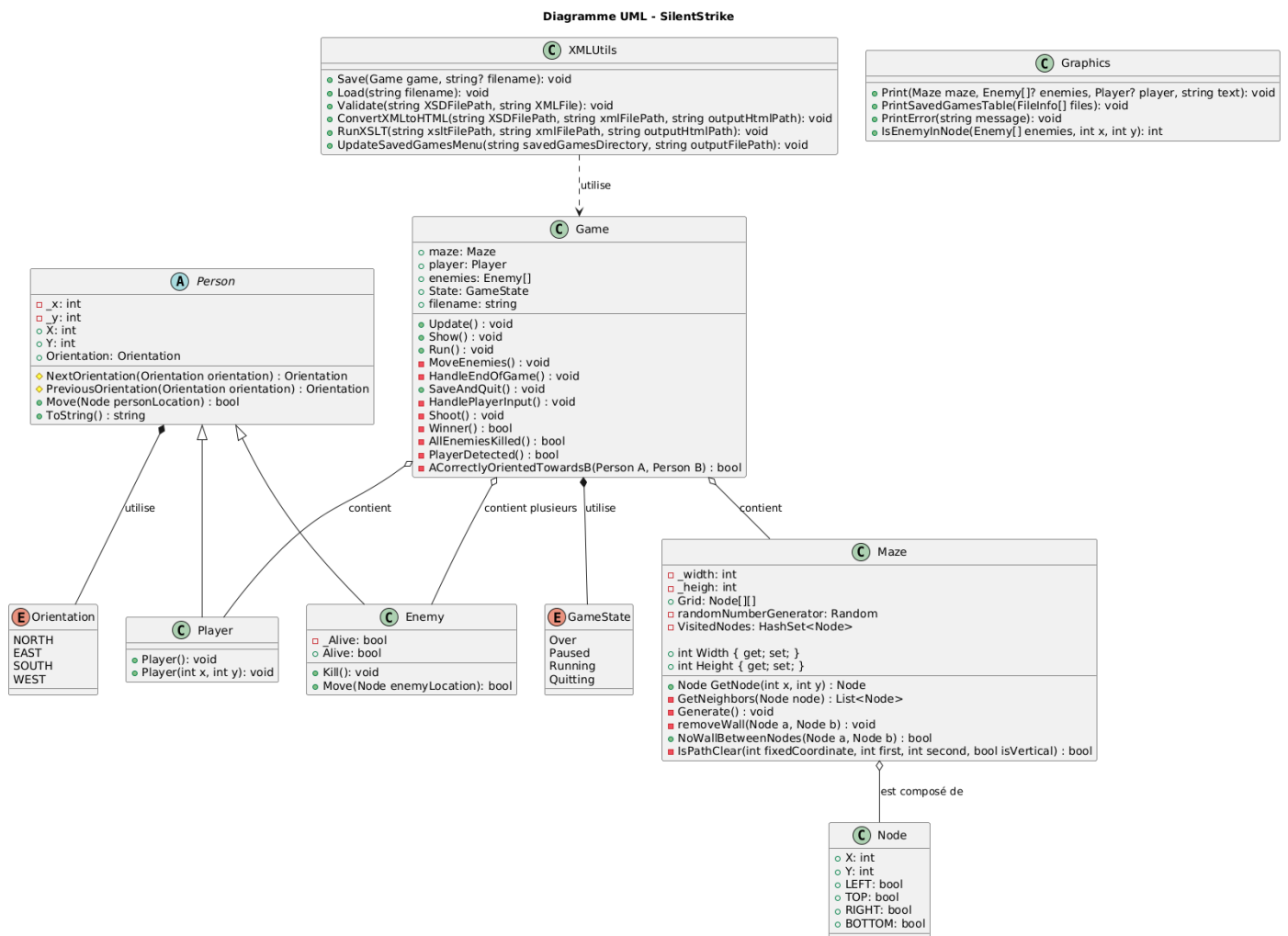


Une partie que l'utilisateur peut voir sur le navigateur

4) Les Diagrammes

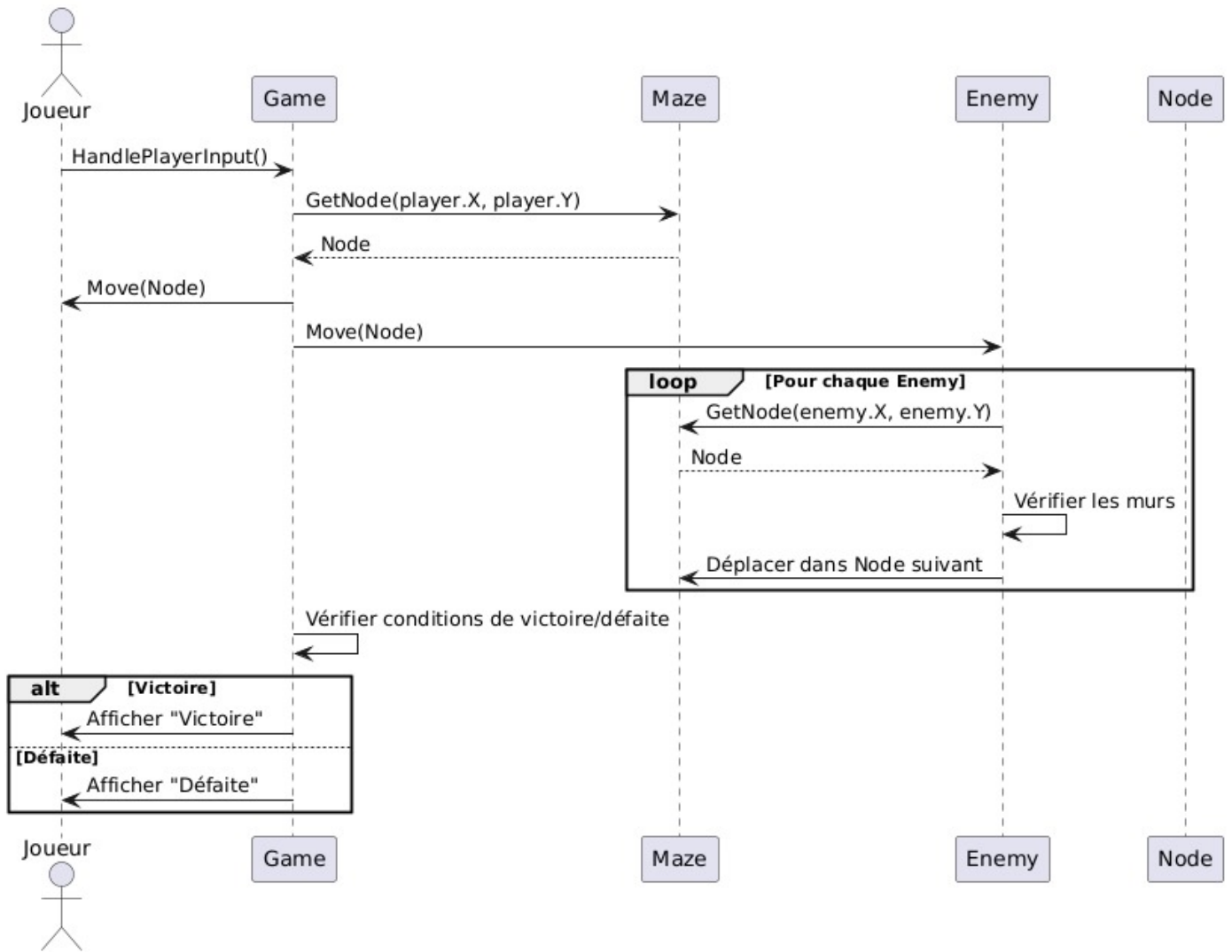
4.1) Diagramme UML

Le diagramme UML présenté ci-dessous est un diagramme de classes qui modélise la structure logicielle du jeu. Il décrit les principales classes, telles que Game, Player, Enemy et Maze, ainsi que leurs relations et interactions.



4.2) Diagramme de Séquence

Diagramme UML de séquence - Interaction de jeu



Contexte du diagramme :

Le diagramme représente une interaction typique dans le jeu , où :

1. Le joueur effectue une action (comme se déplacer ou tirer).
2. La fonction Update est appelé et la reaction adéquate est faite.
3. Refait la même chose jusqu'à ce que le joueur gagne ou perd.

Acteurs et participants :

1. Joueur (Player) : L'utilisateur contrôlant le personnage.
2. Jeu (Game) : La classe principale qui orchestre les interactions.
3. Labyrinthe (Maze) : Structure représentant les nœuds et murs.
4. Ennemi (Enemy) : Entité autonome qui agit selon des règles prédéfinies.
5. Nœud (Node) : Cellule du labyrinthe utilisée pour les déplacements.