Rapport SAE ZELDIABLO

TRITARELLI Adrien
ADAM Tristan
LOGEART Pierre
MILOIKOVITCH Maxime

4 juin 2025

Table des matières

1 Introduction

Ce projet s'inscrit dans le cadre de la SAE 2.01 du BUT Informatique. Il s'agit d'un travail en groupe visant à concevoir et développer une application Java, avec un accent particulier mis sur la qualité de la structure du code et la démarche de conception.

Le TP initial représente le point de départ de cette SAE. Il a pour objectif de nous fournir une première base de code pour l'application à développer, tout en nous familiarisant avec la démarche de projet à suivre.

L'objectif global de la SAE 2.01 est de concevoir une application en suivant une approche rigoureuse, structurée et collaborative. Cela implique plusieurs étapes :

— choisir les fonctionnalités que nous souhaitons implémenter ;
— analyser et adapter le code existant pour les intégrer correctement ;
— produire des diagrammes de classes et de séquences pour expliquer la conception ;
— modifier et étendre le code de manière cohérente ;
— tester et valider chaque nouvelle fonctionnalité.

La SAE est réalisée en groupes de quatre étudiants. Le travail est coordonné à l'aide de l'outil Git, avec un dépôt unique partagé entre les membres de chaque groupe.

2 Organisation du projet

Le projet ZELDIABLO s'est déroulé en quatre itérations de 4 heures chacune. Chaque itération a suivi une structure précise, permettant une progression incrémentale et maîtrisée du développement de l'application.

Voici le déroulement type d'une itération :

— **Choix des fonctionnalités (30 min)** :
  — Identification des nouvelles fonctionnalités à intégrer ;
  — Précision des tâches à réaliser ;
  — Définition des critères de validation et des tests correspondants.
— **Conception (1 h)** :
  — Analyse de la conception existante à partir de l'itération précédente ;
  — Réalisation des diagrammes de classe et de séquence pour les nouveautés ;
  — Mise à jour de la conception globale ;
  — Répartition des tâches à partir de la conception.
— **Réalisation (1 h 30)** :
  — Implémentation des fonctionnalités choisies ;
  — Écriture des tests associés.
— **Intégration et validation (1 h)** :
  — Passage des tests, correction des bugs éventuels ;
  — Mise en forme finale du code, ajout de documentation et création d'un tag Git.

Chaque itération a donné lieu à un rendu partiel du projet, qui sera présenté dans les sections suivantes.

3 Itération 1

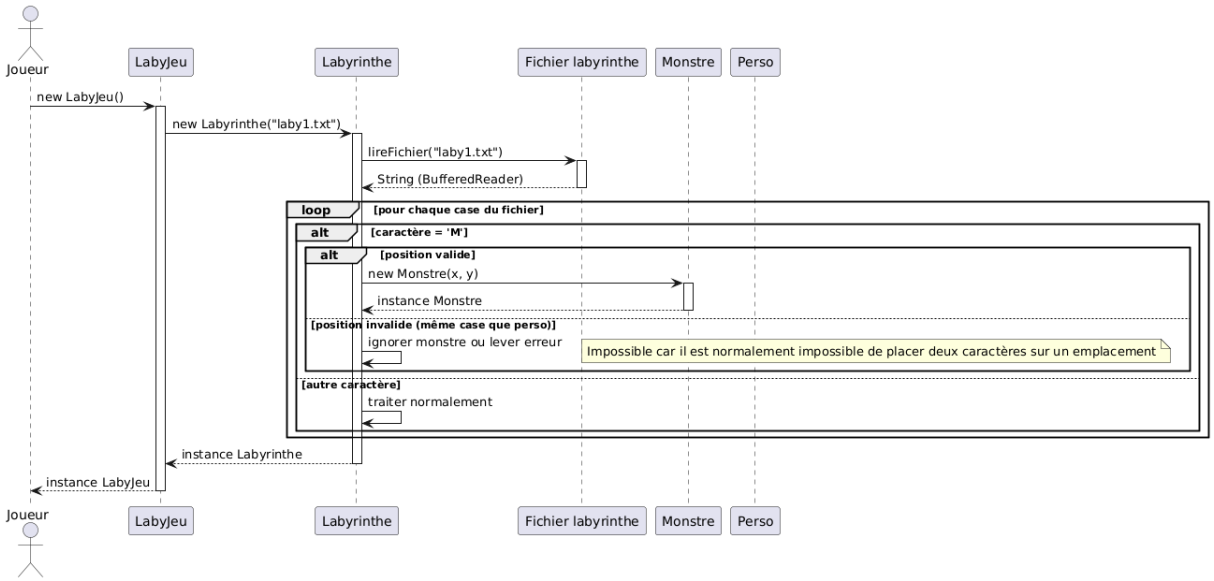
3.1 Choix des fonctionnalités

1. **Donner une position initiale au monstre**
   - *Descriptif :* Le monstre débute sur une case décrite dans le fichier labyrinthe.
   - *Critères de validation :*
     - Le monstre doit avoir une position initiale.
     - Le monstre se trouve sur la case indiquée dans le fichier labyrinthe.
     - Le monstre est représenté par le caractère 'M' dans le fichier labyrinthe.
     - Le monstre ne se trouve pas sur la même case que le personnage.
2. **Afficher le monstre**
   - *Descriptif :* Le jeu doit afficher le monstre à sa position.
   - *Critères de validation :*
     - Le monstre doit être affiché à la bonne position dans le labyrinthe.
     - Le monstre sera représenté sous la forme d'un cercle violet de la taille du personnage.
3. **Considérer le monstre dans les déplacements du personnage**
   - *Descriptif :* Lorsque le jeu évolue, le personnage ne peut pas se déplacer sur la case du monstre.
   - *Critères de validation :*
     - Le monstre constitue un obstacle pour le personnage.
     - Le monstre et le personnage ne peuvent pas se trouver sur la même case.
     - Le personnage ne peut pas traverser la case du monstre.
4. **Déplacer le monstre (optionnel en fonction avancée)**
   - *Descriptif :* Lorsque le jeu évolue, le monstre choisit une case adjacente de manière aléatoire et tente de s'y déplacer.
   - *Critères de validation :*
     - Le monstre doit se déplacer sur une case adjacente. Il considère les 4 directions de déplacement possibles.
     - Le monstre ne peut pas se déplacer sur un mur. S'il tente de se déplacer sur cette case, il ne bouge pas.
     - Le monstre ne peut pas se trouver sur la même case que le personnage.

3.2 Conception

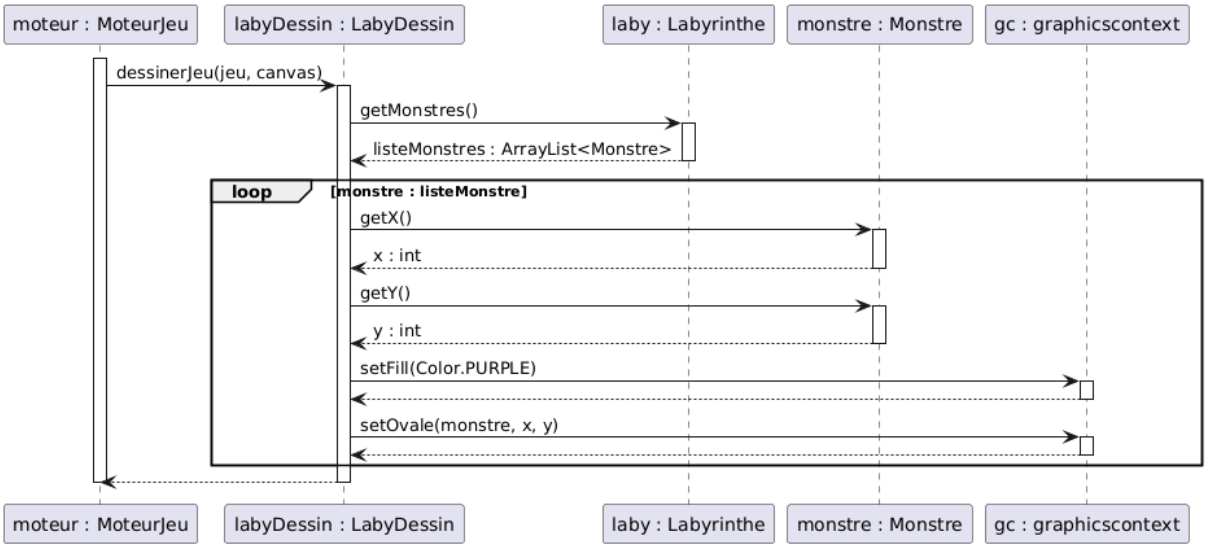
3.2.1 Diagramme de séquence : Initialisation du jeu et du monstre

Ce diagramme illustre la séquence d'initialisation du jeu, mettant en avant la création de l'objet `LabyJeu`, la lecture du fichier labyrinthe, et la création du monstre. Lors de la lecture du fichier, chaque case est analysée : si un caractère 'M' est rencontré et que la position est valide (pas sur la même case que le personnage), une instance de `Monstre` est créée et placée. Ce processus garantit que le monstre a une position initiale correcte, respectant les contraintes du fichier.



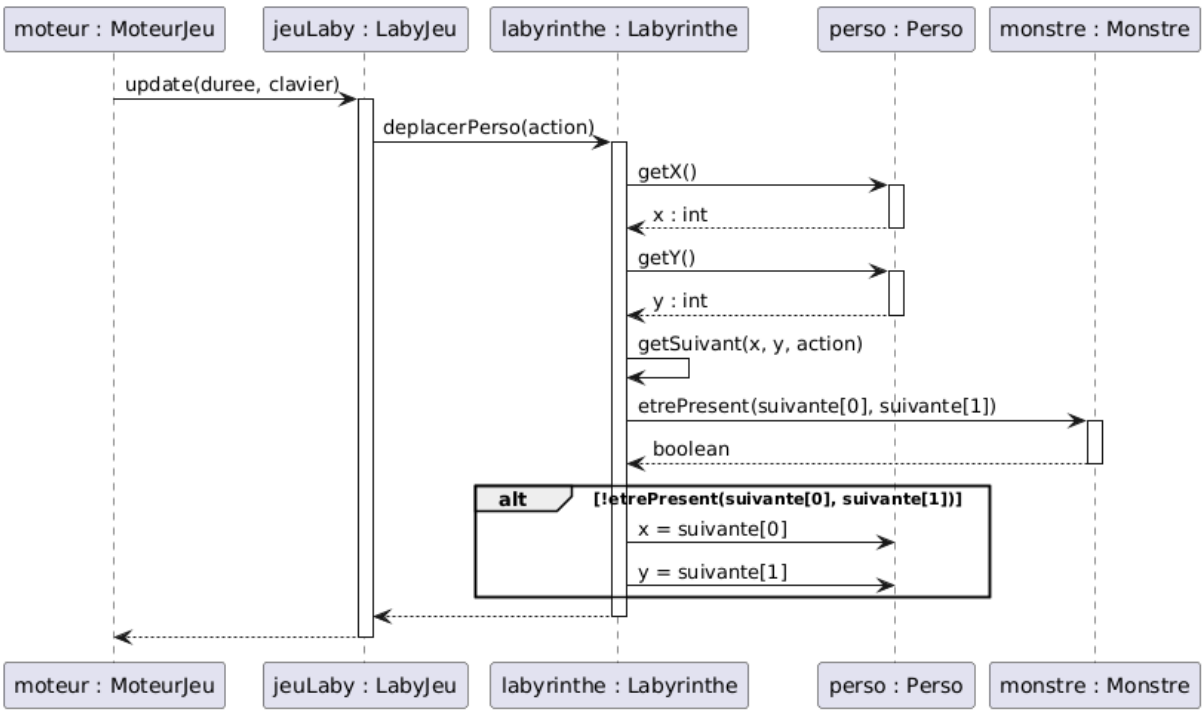
3.2.2 Diagramme de séquence : Affichage du monstre

Ce diagramme décrit comment le moteur de jeu sollicite le composant d'affichage (`LabyDessin`) pour dessiner le monstre. Le dessin récupère la liste des monstres depuis le labyrinthe, puis pour chaque monstre, obtient ses coordonnées, définit la couleur violette, et dessine un ovale à la position correspondante. Ce mécanisme permet d'afficher visuellement le monstre dans le labyrinthe à la bonne position.



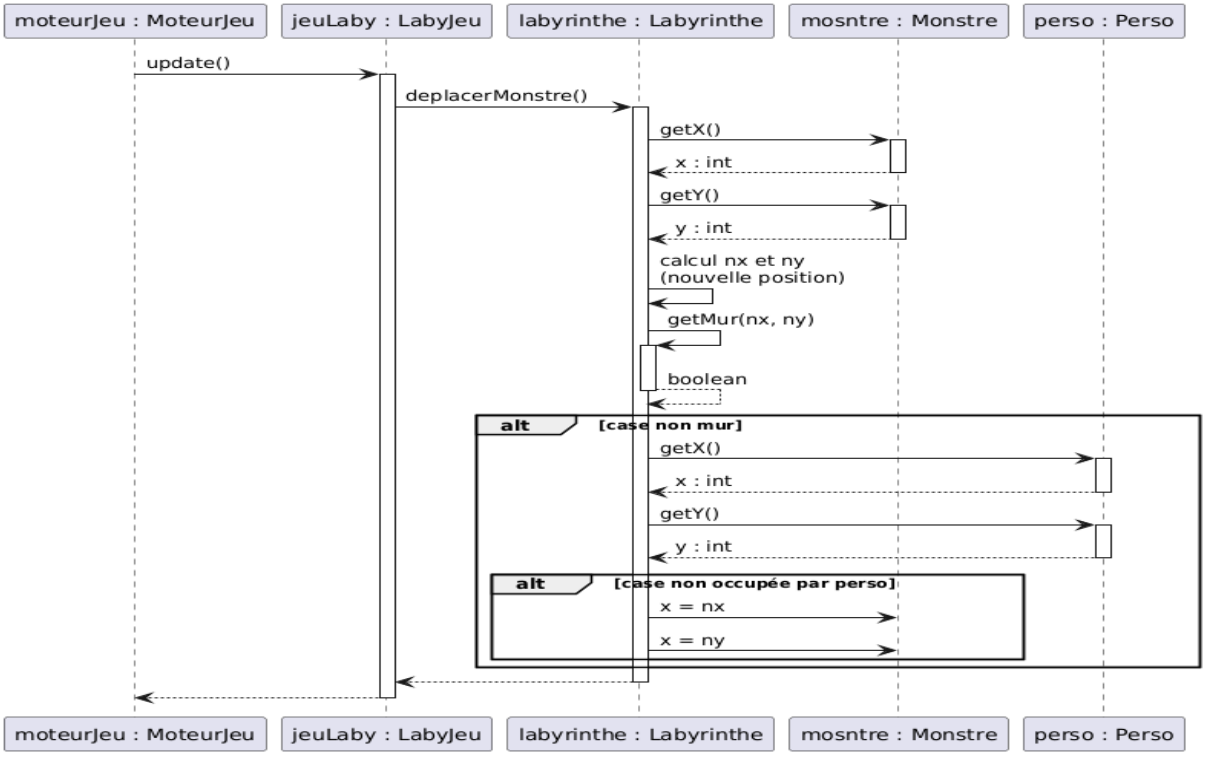
3.2.3 Diagramme de séquence : Gestion du déplacement du personnage

Ce diagramme met en lumière la procédure lors d'un déplacement du personnage. Le moteur met à jour le jeu, qui demande au labyrinthe de déplacer le personnage selon l'action effectuée. Le labyrinthe vérifie la position cible, notamment en interrogeant le monstre pour s'assurer que la case n'est pas occupée. Si la case est libre (monstre absent), le personnage peut se déplacer. Sinon, le déplacement est bloqué, garantissant que le personnage ne peut pas occuper la même case que le monstre.



3.2.4 Diagramme de séquence : Déplacement du monstre

Ce diagramme décrit comment le monstre est déplacé de façon aléatoire par le moteur de jeu. Le labyrinthe calcule une nouvelle position candidate pour le monstre, puis vérifie que cette position n'est pas un mur et qu'elle n'est pas occupée par le personnage. Si la case est libre, le monstre est déplacé ; sinon, il reste à sa position actuelle. Ce contrôle assure que le monstre ne traverse pas les murs ni n'entre en collision avec le personnage.



3.3 Réalisation

3.3.1 Donner une position initiale au monstre

Le constructeur de la classe `Labyrinthe` lit le fichier de labyrinthe ligne par ligne. Pour chaque caractère, il crée les éléments correspondants : murs, case vide, personnage ou monstre. Pour le monstre, il instancie un objet `Monstre` avec les coordonnées lues, et l'ajoute à la liste des monstres. Cela permet de définir sa position initiale conformément au fichier.

```
public Labyrinthe(String nom) throws IOException {
        // ouvrir fichier
        FileReader fichier = new FileReader(nom);
        BufferedReader bfRead = new BufferedReader(fichier);

        int nbLignes, nbColonnes;
        // lecture nblignes
        nbLignes = Integer.parseInt(bfRead.readLine());
        // lecture nbcolonnes
        nbColonnes = Integer.parseInt(bfRead.readLine());

        // creation labyrinthe vide
        this.murs = new boolean[nbColonnes][nbLignes];
        this.pj = null;

        // lecture des cases
        String ligne = bfRead.readLine();

        // stocke les indices courants
        int numeroLigne = 0;

        // parcours le fichier
        while (ligne != null) {

            // parcours de la ligne
            for (int colonne = 0; colonne < ligne.length();
               colonne++) {
                char c = ligne.charAt(colonne);
                switch (c) {
                    case MUR:
                        this.murs[colonne][numeroLigne] = true;
                        break;
                    case VIDE:
                        this.murs[colonne][numeroLigne] = false;
                        break;
                    case PJ:
                        // pas de mur
                        this.murs[colonne][numeroLigne] = false;
                        // ajoute PJ
                        this.pj = new Perso(colonne, numeroLigne
```

```
                        );
                    break;
                case MONSTRE:
                    this.murs[colonne][numeroLigne] = false;
                    if (this.pj == null || !this.pj.
                        etrePresent(colonne,numeroLigne)) {
                        // permet de s'assurer qu'un
                        personnage n'est pas d j  pr sent
                        sur la position que va occuper le
                        monstre
                         this.monstres.add(new Monstre(
                            colonne, numeroLigne));
                    }
                    break;
                default:
                    throw new Error("caractere inconnu " + c
                        );
                }
            }

            // lecture
            ligne = bfRead.readLine();
            numeroLigne++;
        }

        // ferme fichier
        bfRead.close();
    }
```

3.3.2 Afficher le monstre

Pour afficher le monstre, la méthode de dessin utilise une boucle qui parcourt la liste des monstres, puis dessine un ovale violet à la position de chacun, ajusté à la taille de la case.

```
// Dessin des monstres
gc.setFill(Color.PURPLE);
for (Monstre monstre : laby.getMonstres()) {
    gc.fillOval(monstre.getX() * tailleCase + tailleCase / 4,
                monstre.getY() * tailleCase + tailleCase / 4,
                tailleCase / 2,
                tailleCase / 2);
}
```

3.3.3 Déplacer le monstre

La méthode `deplacerMonstre` choisit aléatoirement une direction parmi les quatre possibles pour chaque monstre, calcule la case suivante, et effectue le déplacement uni-

quement si la case n'est pas un mur. Si la case est un mur, le monstre reste à sa position actuelle.

```
public void deplacerMonstre() {
        for (int i = 0; i < this.monstres.size(); i++) {
            // case courante
            int[] courante = {this.monstres.get(i).getX(), this.
               monstres.get(i).getY()};
            String[] action= {HAUT, BAS, GAUCHE, DROITE};

            // calcule case suivante
            int[] suivante = getSuivant(courante[0], courante
               [1], action[(int)(Math.random()*4)]);

            // si ce n'est ni un mur ni un personnage, on
               effectue le deplacement
            if (!this.murs[suivante[0]][suivante[1]] && !this.pj
               .etrePresent(suivante[0],suivante[1])) {
                // on met a jour le monstre
                this.monstres.get(i).x = suivante[0];
                this.monstres.get(i).y = suivante[1];
            }
        }
    }
```

3.4 Intégration et validation

L'intégration consiste à rassembler toutes les fonctionnalités développées au cours des différentes itérations dans une base de code commune, en s'assurant que leur interaction est correcte. Dans notre projet, cela passe par la coordination via Git, et la vérification que le déplacement et l'affichage du monstre fonctionnent comme prévu dans le cadre du jeu.

Le code suivant illustre un test unitaire automatisé écrit avec JUnit, qui vérifie la validité du déplacement du monstre. Ce test est répété plusieurs fois pour couvrir la nature aléatoire du déplacement :

```
    @RepeatedTest(10)  // Test repete plusieurs fois a cause du
       hasard
    public void testDeplacerMonstre_DeplacementValide() {
        int width = 800;
        int height = 600;
        int fps = 60;

        // Creation des objets jeu et dessin
        LabyJeu jeuLaby = new LabyJeu();
        LabyDessin dessinLaby = new LabyDessin();

        // Configuration du moteur
        MoteurJeu.setTaille(width, height);
```

```
        MoteurJeu.setFPS(fps);

        List<Monstre> monstresAvant = jeuLaby.getLabyrinthe().
            getMonstres();
        int xAvant = monstresAvant.get(0).getX();
        int yAvant = monstresAvant.get(0).getY();

        jeuLaby.getLabyrinthe().deplacerMonstre();

        int xApres = monstresAvant.get(0).getX();
        int yApres = monstresAvant.get(0).getY();

        // Le monstre doit soit tre rest sur place, soit s'
            tre d plac d'une case adjacente
        boolean deplacementValide =
                (xApres == xAvant && yApres == yAvant) || //
                    pas d plac (mur ou hasard)
                (xApres == xAvant + 1 && yApres == yAvant) ||
                (xApres == xAvant - 1 && yApres == yAvant) ||
                (xApres == xAvant && yApres == yAvant + 1) ||
                (xApres == xAvant && yApres == yAvant - 1);

        assertTrue(deplacementValide, "Le monstre s'est
            d plac sur une case invalide");

        // La case d'arrivee ne doit pas etre un mur
        assertFalse(jeuLaby.getLabyrinthe().getMur(xApres,
            yApres), "Le monstre s'est d plac sur un mur");
    }
```

Ce test permet de valider que le déplacement du monstre respecte les règles définies : - Le monstre ne sort pas des cases adjacentes. - Il ne se déplace pas sur un mur.

La répétition du test plusieurs fois est nécessaire en raison de l'aspect aléatoire du déplacement. Cette validation automatique assure la fiabilité de la fonctionnalité, facilite le débogage, et garantit que les modifications futures ne cassent pas le comportement attendu.

En plus du test aléatoire du déplacement du monstre, nous avons ajouté plusieurs tests unitaires afin de valider la bonne position initiale du monstre et de garantir qu'il ne se trouve jamais sur la même case que le personnage.

testMonstrePositionInitiale : vérifie qu'au moins un monstre est bien présent à l'initialisation

```
        @Test
    public void testMonstrePositionInitiale() {
        LabyJeu jeu = new LabyJeu();
        List<Monstre> monstres = jeu.getLabyrinthe().getMonstres
            ();

        assertFalse(monstres.isEmpty(), "Le monstre n'a pas de
```

```
            position initiale");
    }
```

testMonstreSurBonneCase : s'assure que le monstre est positionné aux coordonnées attendues dans le fichier labyrinthe

```
        @Test
        public void testMonstreSurBonneCase() {
            LabyJeu jeu = new LabyJeu();
            List<Monstre> monstres = jeu.getLabyrinthe().
                getMonstres();

            Monstre m = monstres.get(0);

            assertEquals(4, m.getX(), "Le monstre n'est pas
                sur la colonne attendue");
            assertEquals(1, m.getY(), "Le monstre n'est pas
                sur la ligne attendue");
        }
```

testFichierLabyrintheContientM : vérifie que le caractère `M` est bien présent dans le fichier de construction du labyrinthe

```
    @Test
    public void testFichierLabyrintheContientM() throws
        IOException {
        BufferedReader br = new BufferedReader(new
            FileReader("labySimple/laby1.txt"));
        br.readLine();
        br.readLine();

        String ligne;
        boolean monstreTrouve = false;
        while ((ligne = br.readLine()) != null) {
            if(ligne.contains("M")) {
                monstreTrouve = true;
                break;
            }
        }

        br.close();
        assertTrue(monstreTrouve, "Le fichier Labyrinthe ne
            contient pas de monstre");
    }
```

testMonstrePasSurPersonnage : garantit qu'aucun monstre ne se trouve sur la même case que le personnage au chargement du labyrinthe

```
    @Test
    public void testMonstrePasSurPersonnage() {
        LabyJeu jeu = new LabyJeu();
```

```
        List<Monstre> monstres = jeu.getLabyrinthe().
            getMonstres();
        int x = jeu.getLabyrinthe().pj.getX();
        int y = jeu.getLabyrinthe().pj.getY();
        boolean memeCase;

        for (Monstre m : monstres) {
            memeCase = (m.getX() == x && m.getY() == y);
            assertFalse(memeCase,"Le monstre est positionn
                sur la m me case que le personnage");
        }
    }
```

4 Itération 2

4.1 Choix des fonctionnalités

1. **Donner une position initiale au monstre**
   - *Descriptif :* Le monstre débute sur une case décrite dans le fichier labyrinthe.
   - *Critères de validation :*
     - Le monstre doit avoir une position initiale.
     - Le monstre se trouve sur la case indiquée dans le fichier labyrinthe.
     - Le monstre est représenté par le caractère 'M' dans le fichier labyrinthe.
     - Le monstre ne se trouve pas sur la même case que le personnage.
2. **Affichage les attaques du monstre**
   - *Descriptif :* Quand un monstre attaque, il change de couleur.
   - *Critères de validation :*
     - La couleur d'affichage du monstre change au moment où il attaque (passe de rouge à noir).
     - Après l'attaque, le joueur et le monstre reprennent une couleur normale.
3. **Fin du jeu : mort du héros**
   - *Descriptif :* Lorsque le héros n'a plus de points de vie, le jeu s'arrête et la partie est perdue. Critères de validation
   - *Critères de validation :*
     - Le héros par défaut dispose de 5 points de vie.
     - Chaque monstre fait 1 point de dégât lorsque le héros se fait attaquer.
     - Quand le héros meurt, un message de fin est affiché sur la console et le jeu s'arrête.

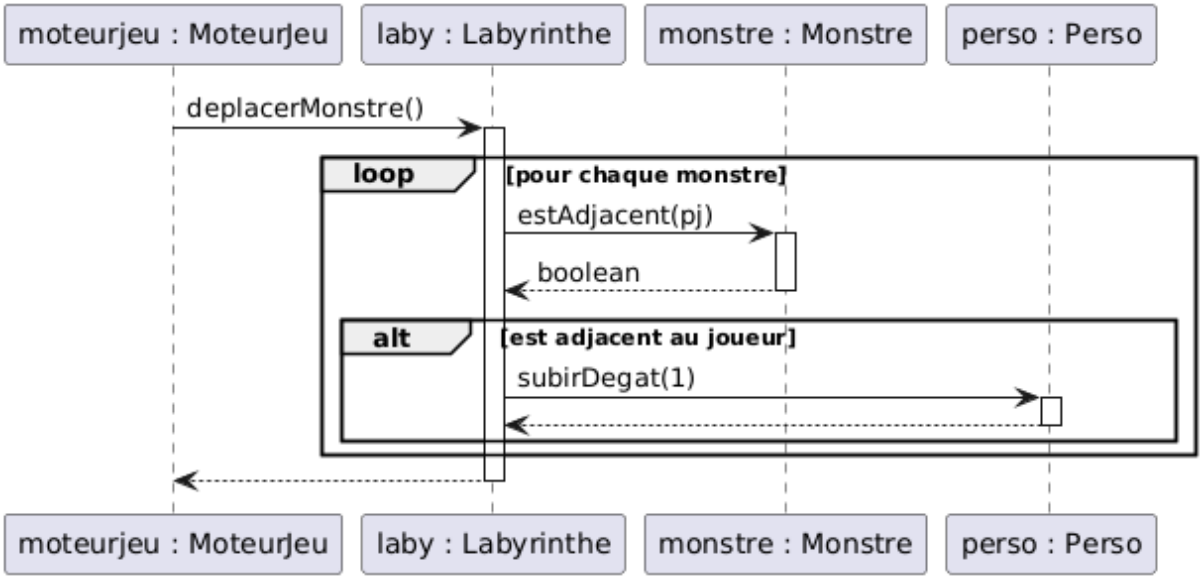
4.2 Conception

4.2.1 Diagramme de séquence : Attaque des monstres

Le diagramme de séquence ci-dessous modélise le comportement des monstres lorsqu'ils sont adjacents au personnage joueur (`Perso`) durant l'évolution du jeu. Ce processus est déclenché par le moteur de jeu (`MoteurJeu`) via l'appel à la méthode `deplacerMonstre()` de la classe `Labyrinthe`.

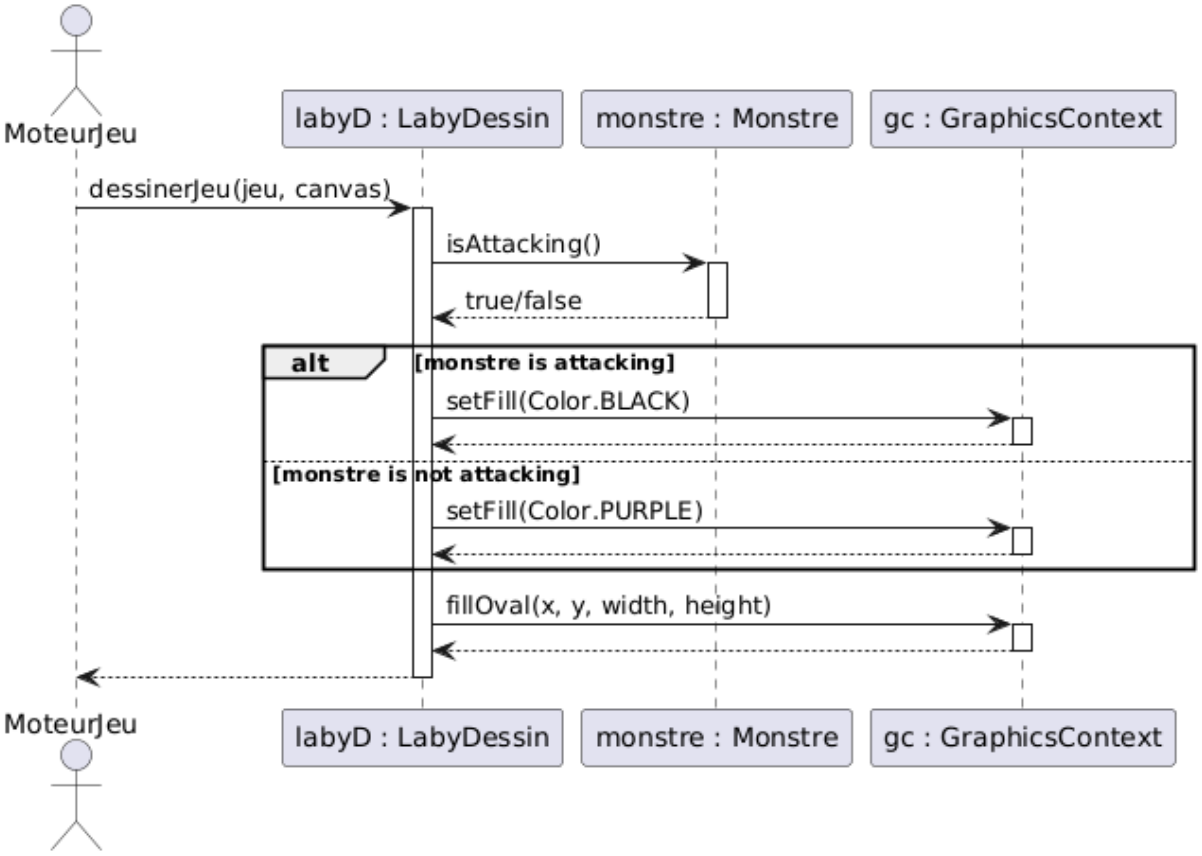
À l'intérieur de cette méthode, une boucle parcourt chaque monstre présent dans le labyrinthe. Pour chaque monstre, la méthode `estAdjacent(pj)` est invoquée pour vérifier si le monstre se trouve sur une case adjacente à celle du personnage. Si c'est le cas, le monstre attaque le personnage en appelant la méthode `subirDegat(1)`, qui fait perdre un point de vie au héros.

Ce comportement permet d'introduire une interaction offensive entre les monstres et le joueur, renforçant l'aspect dynamique et dangereux du jeu. Cette logique fait désormais partie intégrante de la boucle de jeu et permet de détecter des situations critiques pouvant mener à la fin de la partie.



4.2.2 Diagramme de séquence : Affichage des attaques du monstre

Ce diagramme décrit comment le moteur de jeu choisis la couleur du monstre (`LabyDessin`). C'est un ajout à dessinJeu, une méthode déjà éxistante. Le dessin récupère la liste des monstres depuis le labyrinthe, puis pour chaque monstre, regarde si il attaque ou non. Si le monstre attaque, il est dessiné en noir, sinon il est dessiné en rouge. Cela permet de visualiser l'état d'attaque du monstre dans le labyrinthe.



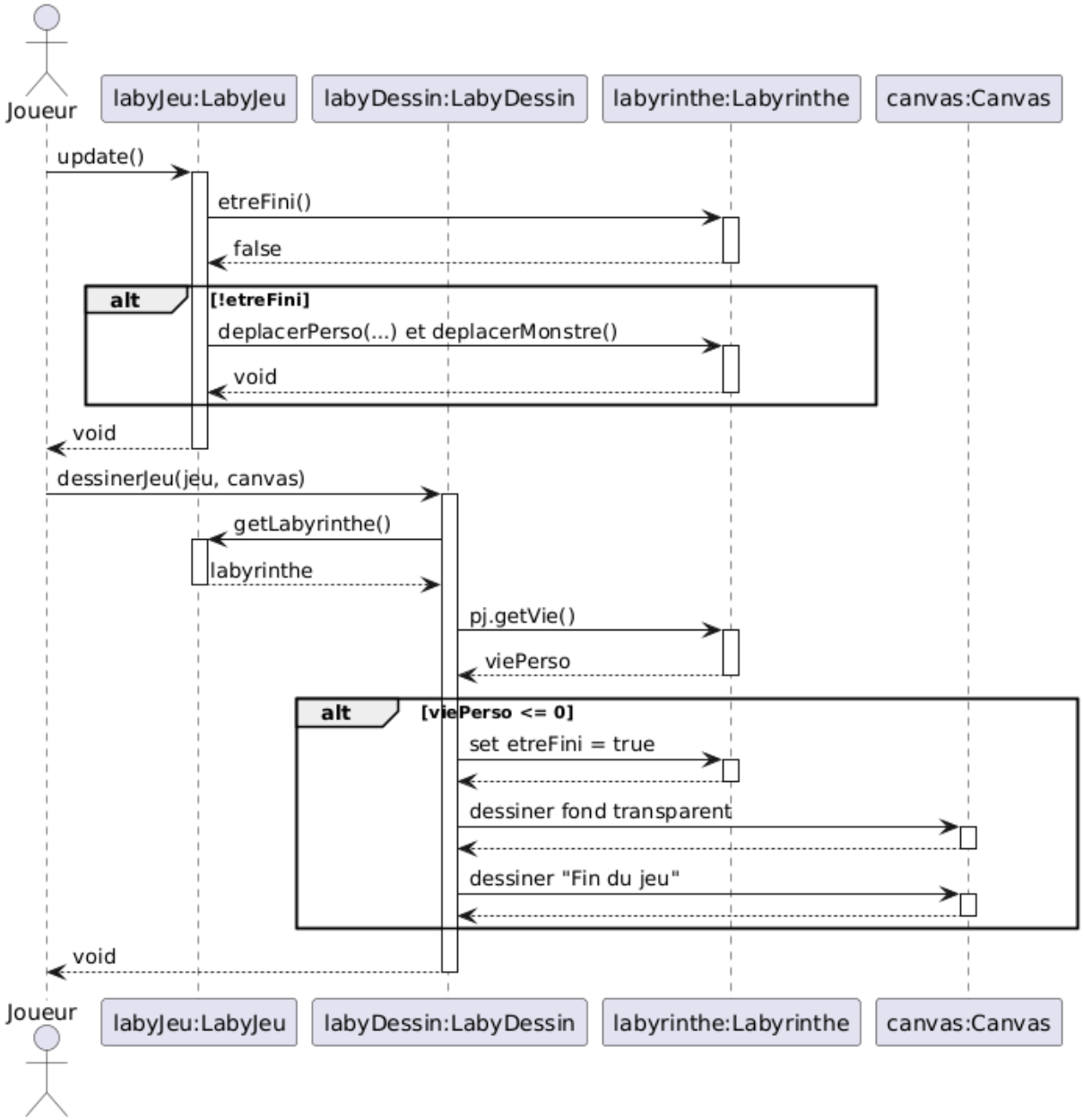
4.2.3 Diagramme de séquence : Fin du jeu : mort du héros

Dans cette itération, la conception s'appuie sur un nouveau diagramme de séquence mettant en évidence la logique de fin de jeu. Celui-ci illustre comment le programme détecte que la partie est terminée lorsque le personnage principal (le *Perso*) n'a plus de points de vie.

Le diagramme commence par une mise à jour (`update()`) déclenchée par le joueur. Le moteur du jeu (`LabyJeu`) interroge alors le labyrinthe (`Labyrinthe`) pour vérifier si la partie est déjà finie via la méthode `etreFini()`. Si ce n'est pas le cas, le jeu effectue les déplacements du personnage et du monstre.

Ensuite, la vue (`LabyDessin`) est sollicitée pour dessiner le jeu. Elle récupère le labyrinthe, puis consulte le nombre de points de vie du personnage via `pj.getVie()`. Une condition est alors testée : si le nombre de vies est inférieur ou égal à 0, cela signifie que le personnage est mort. Dans ce cas, la variable `etreFini` est mise à `true`, et l'écran affiche un fond transparent avec le message « Fin du jeu ».

Cette séquence permet ainsi de garantir que l'état de fin de partie est bien détecté à chaque rafraîchissement du jeu, et que l'information est correctement transmise à l'interface graphique pour être affichée à l'utilisateur.



4.3 Réalisation

4.3.1 Attaque des monstres

Pour intégrer l'attaque des monstres lorsque le personnage est à proximité, plusieurs éléments ont été ajoutés au code. Chaque partie est décrite brièvement ci-dessous.

Méthode `subirDegat`

```
public void subirDegat(int degats){
    this.vie -= degats;
}
```

Cette méthode permet à un personnage (notamment le héros) de perdre un nombre donné de points de vie lorsqu'il est attaqué.

Classe `Monstre`

```
public class Monstre extends Personnage {

    private boolean attaque;

    public Monstre(int dx, int dy, int v) {
        super(dx, dy, v);
    }

    public void attaquer(Personnage cible) {
        cible.subirDegat(1);
        this.attaque = true;
    }

    public boolean estAdjacent(Personnage p) {
        return (Math.abs(this.x - p.x) <= 1 && Math.abs(this.y -
            p.y) <= 1)
                && (this.x == p.x || this.y == p.y);
    }

    public void stopAttaque(){
        this.attaque = false;
    }

    public boolean isAttacking() {
        return this.attaque;
    }
}
```

Cette classe ajoute le comportement d'attaque au monstre. Si un héros est détecté sur une case adjacente (horizontale ou verticale), le monstre peut lui infliger des dégâts. Le booléen `attaque` permet de ne pas répéter l'attaque à chaque frame tant que le monstre reste collé au héros.

Méthode `deplacerPerso`

```
public void deplacerPerso(String action) {
    int[] courante = {this.pj.x, this.pj.y};
    int[] suivante = getSuivant(courante[0], courante[1], action
        );

    if (!this.murs[suivante[0]][suivante[1]] && (!monstrePresent
        (suivante[0],suivante[1]))) {
        this.pj.x = suivante[0];
        this.pj.y = suivante[1];

        for (Monstre m : this.monstres) {
            if (m.estAdjacent(this.pj)) {
                m.attaquer(this.pj);
            } else {
```

```
                m.stopAttaque();
            }
        }
    }
}
```

Le déplacement du personnage est désormais suivi d'une vérification de proximité avec les monstres. Si un monstre est adjacent, il attaque immédiatement.

Méthode `deplacerMonstre`

```
public void deplacerMonstre() {
    for (Monstre m : this.monstres) {
        if (m.estAdjacent(this.pj) && !m.isAttacking()) {
            m.attaquer(this.pj);
        } else if (!m.isAttacking()) {
            int[] courante = {m.getX(), m.getY()};
            String[] action = {HAUT, BAS, GAUCHE, DROITE};
            int[] suivante = getSuivant(courante[0], courante
                [1], action[(int) (Math.random() * 4)]);

            if (!this.murs[suivante[0]][suivante[1]] && !this.pj
                .etrePresent(suivante[0], suivante[1])) {
                m.x = suivante[0];
                m.y = suivante[1];
            }
        }
    }
}
```

Cette méthode complète le comportement du monstre. S'il est adjacent au héros et ne l'a pas encore attaqué, il inflige des dégâts. Sinon, il peut se déplacer aléatoirement sur une case vide non occupée.

4.3.2 Affichage des attaques du monstre

Pour que le monstre change de couleur, on vérifie si le monstre est en train d'attaquer avec isAttacking(), si oui, il sera rouge, sinon il sera violet.

```
    // Changement de couleur
    for (Monstre monstre : laby.getMonstres()) {
            //Si le monstre attaque on change ca couleur, sinon
                on met celle de base
            if (monstre.isAttacking()){
                gc.setFill(Color.BLACK);
            }else {
                gc.setFill(Color.RED);
            }
            gc.fillOval(monstre.getX() * tailleCase + tailleCase
                / 4, monstre.getY() * tailleCase + tailleCase /
                4, tailleCase / 2, tailleCase / 2);
```

```
    }
```

4.3.3 Fin du jeu : mort du héros

Pour implémenter la détection de fin de jeu lorsque le personnage n'a plus de vie, plusieurs ajouts ont été réalisés dans les classes principales.

Ajout d'un attribut `etreFini` dans `Labyrinthe` Nous avons ajouté un attribut de type `boolean` nommé `etreFini`, initialisé à `false`. Il permet de savoir si la partie est terminée ou non.

```
public class Labyrinthe {
    public boolean etreFini = false;
    ...
}
```

Cet attribut pourra être modifié et consulté par les différentes classes du jeu pour déclencher le bon comportement à l'affichage ou au niveau des interactions.

Modification de `LabyDessin` Nous avons ajouté une condition pour afficher un écran de fin lorsque le personnage n'a plus de points de vie. Dans ce cas, la variable `etreFini` est mise à `true`, puis un message est dessiné.

```
if (laby.pj.getVie() <= 0) {
    laby.etreFini = true;
    // Dessin du message de fin de jeu
    gc.setFill(new Color(1, 1, 1, 0.7));
    gc.fillRect(0, 0, canvas.getWidth(), canvas.getHeight());
    gc.setFill(Color.YELLOW);
    gc.fillText("Fin du jeu", canvas.getWidth()/2 - 100, canvas.
       getHeight()/2);
}
```

Cela permet d'alerter visuellement le joueur que la partie est terminée et empêche le redessin du jeu en arrière-plan.

Ajout de la vérification dans `LabyJeu` Enfin, dans `LabyJeu`, une condition a été ajoutée pour que les actions de déplacement (du joueur ou du monstre) ne soient exécutées que si le jeu n'est pas terminé.

```
if (!labyrinthe.etreFini()) {
    cooldown -= secondes;
    if (cooldown <= 0) {
        if (clavier.haut) {
            labyrinthe.deplacerPerso(Labyrinthe.HAUT);
        }
        if (clavier.bas) {
            labyrinthe.deplacerPerso(Labyrinthe.BAS);
        }
        if (clavier.gauche) {
```

```
            labyrinthe.deplacerPerso(Labyrinthe.GAUCHE);
        }
        if (clavier.droite) {
            labyrinthe.deplacerPerso(Labyrinthe.DROITE);
        }
        labyrinthe.deplacerMonstre();
        cooldown = DELAI;
    }
}
```

Ainsi, une fois que la condition de fin de partie est remplie, plus aucun mouvement n'est traité, ce qui garantit un arrêt complet de l'évolution du jeu.

4.4 Intégration et validation

Après avoir codé la gestion des attaques des monstres et la détection de fin de jeu lorsque le héros perd tous ses points de vie, nous avons intégré cette logique dans la boucle principale du jeu (`LabyJeu.update`) ainsi que dans le rendu graphique avec `LabyDessin`. L'attribut `etreFini` dans la classe `Labyrinthe` permet de figer le jeu une fois la condition de fin atteinte.

Pour vérifier la robustesse de cette fonctionnalité, une suite de tests unitaires a été mise en place :

```
@Test
public void testFinDeJeuQuandVieAtteintZero() {
    pj.setVie(0);
    if (pj.getVie() <= 0) {
        labyrinthe.etreFini = true;
    }
    assertTrue(labyrinthe.etreFini, "Le jeu doit tre termin
        si la vie est 0");
}

@Test
public void testFinDeJeuQuandVieNegatif() {
    pj.setVie(-5);
    if (pj.getVie() <= 0) {
        labyrinthe.etreFini = true;
    }
    assertTrue(labyrinthe.etreFini, "Le jeu doit tre termin
        si la vie est n gative");
}

@Test
public void testJeuContinueSiViePositive() {
    pj.setVie(3);
    if (pj.getVie() <= 0) {
        labyrinthe.etreFini = true;
    }
```

```
    assertFalse(labyrinthe.etreFini, "Le jeu ne doit pas  tre
       termin  si la vie est positive");
}

@Test
public void testMonstreAttaqueSiAdjacent() {
    Monstre monstre = labyrinthe.getMonstres().get(0);
    monstre.x = pj.getX() + 1;
    monstre.y = pj.getY();
    int vieAvant = pj.getVie();
    labyrinthe.deplacerMonstre();
    assertEquals(vieAvant - 1, pj.getVie(), "Le h ros doit
       perdre 1 PV si le monstre est adjacent");
    assertEquals(pj.getX() + 1, monstre.getX());
    assertEquals(pj.getY(), monstre.getY());
}

@Test
public void testMonstreSeDeplaceSiNonAdjacent() {
    Monstre monstre = labyrinthe.getMonstres().get(0);
    monstre.x = pj.getX() + 3;
    monstre.y = pj.getY() + 3;
    int vieAvant = pj.getVie();
    labyrinthe.deplacerMonstre();
    assertEquals(vieAvant, pj.getVie(), "Le h ros ne doit pas
       perdre de PV si le monstre est  loign  ");
    assertFalse(monstre.isAttacking());
}

@Test
public void testMonstreSeDeplaceSiNonAdjacent() {
    Monstre monstre = labyrinthe.getMonstres().get(0);
    monstre.x = pj.getX() + 3;
    monstre.y = pj.getY() + 3;
    int vieAvant = pj.getVie();
    labyrinthe.deplacerMonstre();
    assertEquals(vieAvant, pj.getVie(), "Le h ros ne doit pas
       perdre de PV si le monstre est  loign  ");
    assertFalse(monstre.isAttacking());
}
```

Ces tests ont permis de valider :

- que la condition de fin de jeu se déclenche bien lorsque les points de vie du personnage sont à 0 ou en dessous,
- que le jeu continue normalement si les points de vie sont strictement positifs,
- que le monstre attaque correctement lorsqu'il est adjacent au héros,
- et qu'il se déplace de manière aléatoire lorsqu'il est éloigné.

L'utilisation de tests JUnit nous a donc permis de garantir la validité fonctionnelle de l'attaque, de la perte de vie et de la gestion de la fin de partie dans un cadre automatisé.

5 Itération 3

5.1 Choix des fonctionnalités

1. **Mise en place de l'amulette**
   - *Descriptif :* Au lancement du jeu, une amulette est placée sur une case vide du labyrinthe. Le placement de l'amulette est toujours le même et dépend du niveau.
   - *Critères de validation :*
     - L'amulette ne peut être placée que sur une case vide.
     - Les monstres et le héros peuvent se situer sur la case de l'amulette.
     - L'amulette est affichée dans le jeu sous la forme d'un cercle jaune sur la case vide où elle se trouve.
2. **Inventaire**
   - *Descriptif :* Des objets (sans utilité pour le moment) sont disposés dans des cases vides du labyrinthe. Lorsque le héros appuie sur la touche d'utilisation ("E" par défaut), il récupère les objets de la case où il se trouve et les ajoute dans son inventaire.
   - *Critères de validation :*
     - L'inventaire est de taille infinie.
     - Lorsque le joueur cherche à prendre un objet alors qu'il n'y a rien sur la case où il se trouve, rien ne se passe.
     - Lorsqu'un joueur prend un objet, l'objet disparaît du labyrinthe (et n'est plus affiché) mais apparaît dans son inventaire.
     - À chaque évolution du jeu, le jeu affiche dans la console l'inventaire du héros.
     - Les objets sont affichés dans le labyrinthe tant qu'ils sont présents (sous la forme d'un cercle noir).
3. **Attaque à distance du joueur**
   - *Descriptif :* Le joueur dispose d'une touche particulière pour tirer à l'arc. Lorsqu'il appuie sur cette touche ("F" par défaut), le héros lance une flèche qui traverse l'écran dans la direction suivie par le héros.
   - *Critères de validation :*
     - La flèche se déplace de manière rectiligne à partir du héros et selon la direction de son dernier déplacement.
     - La flèche se déplace à la vitesse des monstres et du héros.
     - Lorsqu'elle rencontre un obstacle (monstre ou mur), la flèche disparaît et fait 1 point de dégât si c'est un monstre.
     - Tant qu'elle n'est pas détruite, la flèche se déplace d'une case par évolution du jeu.
     - La flèche est affichée dans la fenêtre de jeu sous la forme d'un petit point rouge centré dans la case où elle se trouve.

5.2 Conception

5.2.1 Placement de l'amulette dans le labyrinthe

Lors du chargement du fichier texte du labyrinthe, chaque caractère est lu pour construire la structure du niveau. Lorsque le caractère lu correspond à une amulette (''A''), le labyrinthe l'interprète comme une case vide (pas un mur) puis crée une instance

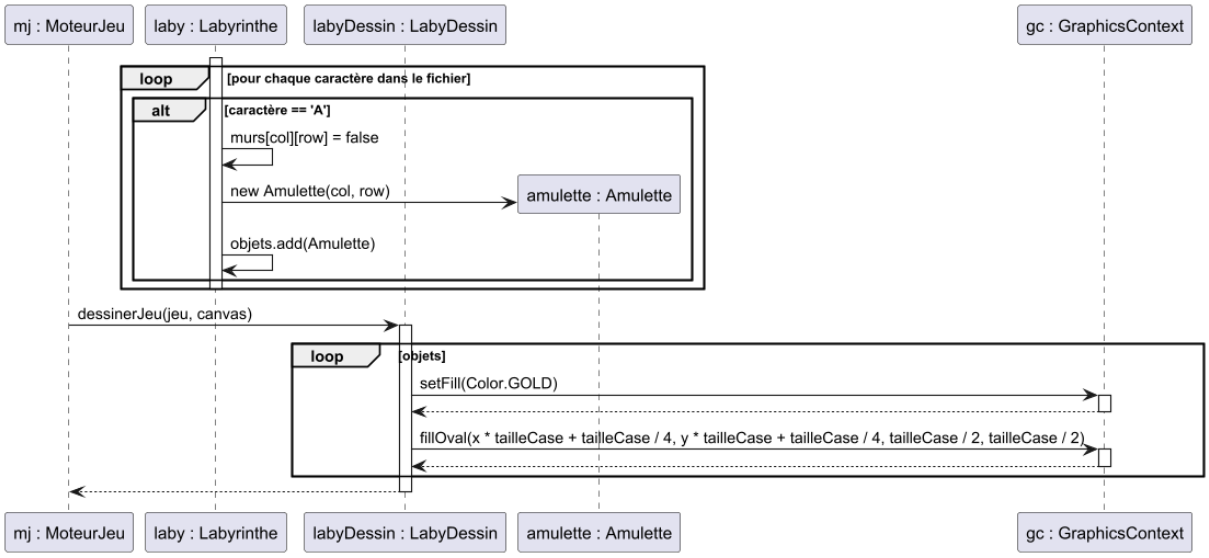
de l'objet `Amulette` à cette position. L'objet est ensuite ajouté à la liste des objets du labyrinthe.

Cette logique est décrite dans le diagramme de séquence suivant :

— Le `Labyrinthe` lit chaque caractère du fichier via un `BufferedReader`.
— Si le caractère correspond à une amulette, la position est enregistrée comme vide.
— Une instance d'`Amulette` est créée à la position détectée.
— L'amulette est ajoutée à la liste des objets du labyrinthe.

Lors de l'affichage :

— La méthode `dessinerJeu` de `LabyDessin` parcourt tous les objets du labyrinthe.
— Si un objet est une amulette, elle est dessinée comme un cercle doré à sa position via le `GraphicsContext`.



5.2.2 Tir d'une flèche par le héros

Lorsqu'un joueur appuie sur la touche dédiée au tir (par exemple "F"), une flèche est créée :

— `LabyJeu` appelle la méthode `tir()` du `Labyrinthe`.
— Le labyrinthe récupère la direction du dernier déplacement du héros via `getDerniereDirection()`
— Une nouvelle instance de `Projectile` est créée à la position actuelle du héros, dans cette direction.
— Cette flèche est ajoutée à la liste des projectiles du labyrinthe.

Le diagramme de séquence montre ce processus en partant de l'interaction du joueur jusqu'à la création et l'enregistrement du projectile.

5.2.3 Déplacement et gestion des flèches (tir)

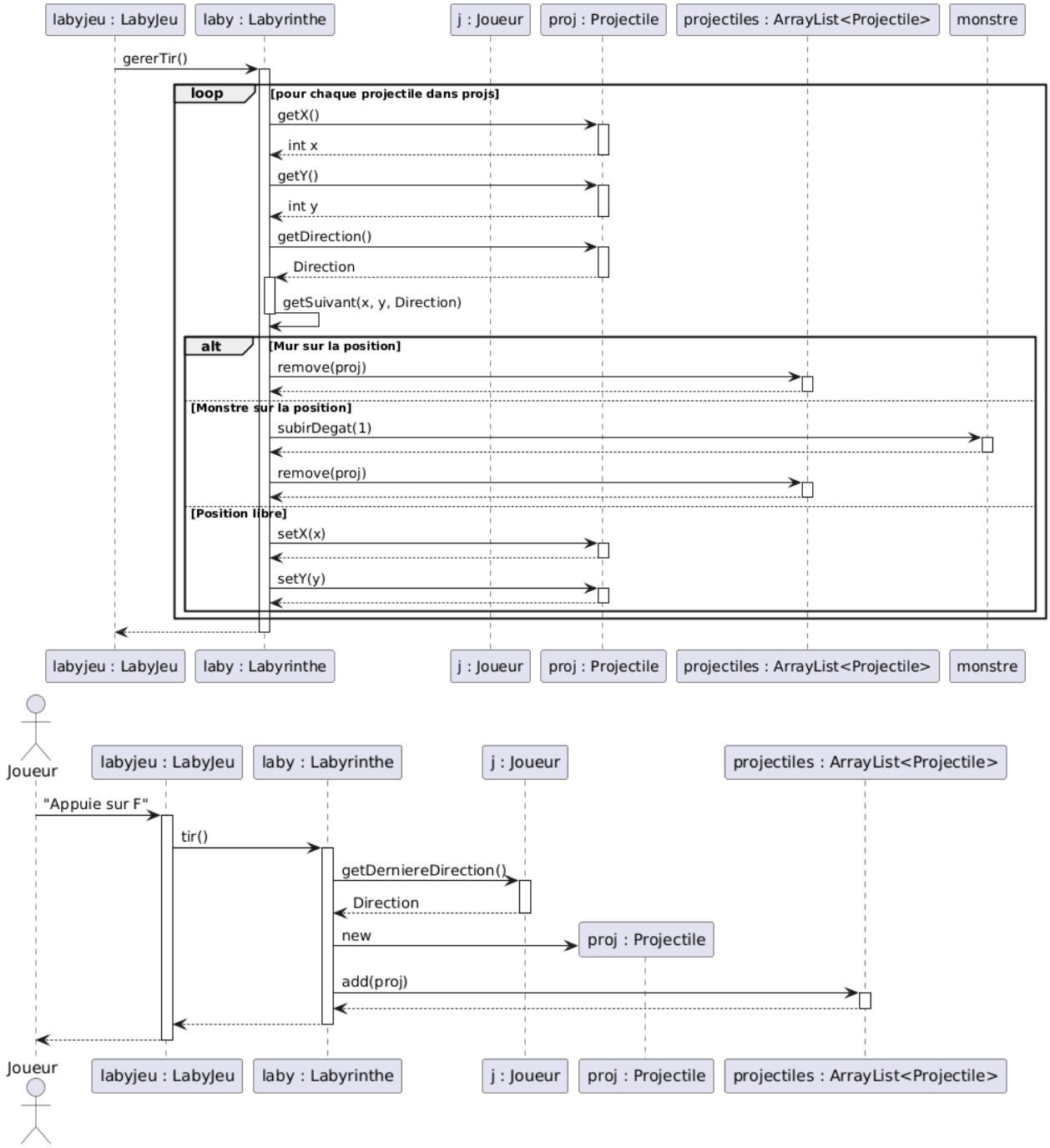
À chaque évolution du jeu (appelée par le moteur), la méthode `gererTir()` du `Labyrinthe` est appelée pour gérer le comportement des projectiles existants :

— Pour chaque projectile :
  — Sa position actuelle et sa direction sont récupérées.
  — La position suivante dans la direction indiquée est calculée.
  — Trois cas sont gérés :
    — **Mur** : le projectile est supprimé.

- **Monstre** : le monstre subit un dégât, et le projectile est supprimé.
- **Case libre** : le projectile avance d'une case dans sa direction.

— Les suppressions de projectiles sont différées pour éviter les conflits lors de l'itération.

Ce mécanisme assure un comportement fluide et réaliste des flèches, tout en respectant la mécanique du tour par tour.



5.3 Realisation

Nous avons modifié les attributs X et Y en privé pour mettre en place des getters. De plus pour clarifier la compréhension des classe, Perso a été renommé en Joueur.

5.3.1 Gestion de l'amulette

L'amulette est un objet spécial ajouté au labyrinthe. Lors de chaque dessin du jeu, une vérification est faite pour savoir si elle a été récupérée. On parcourt la liste des objets et si une instance de l'amulette y est encore présente, cela signifie qu'elle n'a pas été prise. Une fois l'amulette récupérée, et si le joueur atteint la case de sortie, le jeu est marqué comme terminé, avec un message de victoire affiché. Si au contraire la vie du joueur atteint 0, un message de défaite est affiché.

— **Vérification de l'amulette** :

```
boolean amuletteRecup = true;
for (Objet objet : laby.objets) {
    if (objet instanceof Amulette) {
        amuletteRecup = false;
    }
}
```

5.3.2 Système d'inventaire

— **Attribut inventaire dans le héros** :

```
private List<Objet> inventaire = new ArrayList<>();
```

5.3.3 Attaque à distance du héros

Le joueur peut désormais tirer des flèches dans la direction de son dernier déplacement. Une flèche est représentée par un objet `Projectile` contenant sa position et sa direction. À chaque évolution du jeu, la méthode `gererTir()` déplace les projectiles, vérifie les collisions avec les murs ou les monstres, inflige des dégâts, et supprime les projectiles concernés.

— **Classe Projectile** :

```
public class Projectile extends Element {
    private final String direction;
    ...
    public String getDirection() {
        return direction;
    }
}
```

— **Création d'un projectile** :

```
public void tir(){
    Projectile p = new Projectile(pj.getX(), pj.getY(),
                                  pj.getDerniereDirection()
                                     );
    listProjectile.add(p);
}
```

— **Déplacement et collision** :

```
public void gererTir(){
    ArrayList<Projectile> aEnlever = new ArrayList<>();
    for (Projectile p : listProjectile) {
        int[] suivante = getSuivant(p.getX(), p.getY(), p.
            getDirection());
        if (this.murs[suivante[0]][suivante[1]]) {
            aEnlever.add(p);
        } else if ((m = monstrePresent(suivante[0],
            suivante[1])) != null) {
            m.subirDegat(1);
            aEnlever.add(p);
        } else {
            p.setX(suivante[0]);
            p.setY(suivante[1]);
        }
    }
    listProjectile.removeAll(aEnlever);
}
```

— **Affichage graphique** :

```
gc.setFill(Color.BLUE);
gc.setStroke(Color.BLACK);
for (Projectile p : laby.listProjectile) {
    gc.fillOval(p.getX() * tailleCase + tailleCase / 4,
                p.getY() * tailleCase + tailleCase / 4,
                tailleCase / 4, tailleCase / 4);
}
```

5.4 Intégration et validation

Afin de valider le bon fonctionnement des nouvelles fonctionnalités développées à l'itération 3 (amulette, tir à distance), des tests unitaires ont été réalisés avec JUnit. Ces tests permettent de garantir la robustesse du comportement du jeu ainsi que l'intégration cohérente des objets et des projectiles.

Tests liés à l'amulette

— **testAmuletteEstPlaceeSurCaseVide** : Vérifie que l'amulette est bien placée sur une case vide (non murée) et bien ajoutée à la liste des objets du labyrinthe.
— **testAmulettePeutPartagerCaseAvecPJ** : Confirme que le joueur peut occuper la même case que l'amulette sans provoquer de conflit de logique.
— **testAmulettePeutPartagerCaseAvecMonstre** : Vérifie que l'amulette peut coexister avec un monstre sur une même case, ce qui est conforme aux règles de conception.

Tests liés à l'attaque à distance

- **testTirInitialiseDansLaBonneDirection** : Vérifie que la flèche est créée à la position du joueur et orientée selon sa dernière direction de déplacement.
- **testProjectileAvanceCorrectement** : Vérifie qu'une flèche se déplace bien d'une case par évolution de jeu dans la bonne direction.
- **testProjectileDisparaitQuandMur** : Teste la disparition de la flèche lorsqu'elle rencontre un mur dès sa première évolution.
- **testProjectileDisparaitEtBlesseMonstre** : Vérifie que la flèche supprime un point de vie à un monstre s'il est touché, puis disparaît.
- **testProjectileContinueTantQuePasObstacle** : Assure que la flèche continue son mouvement si aucune collision avec mur ou monstre n'est détectée.

```
@Test
public void testAmuletteEstPlaceeSurCaseVide() {
Amulette amulette = new Amulette(2, 2);
laby.objets.add(amulette);

// On marque la case comme vide
laby.murs[2][2] = false;

assertFalse(laby.getMur(2, 2), "La case doit tre vide.
    ");
assertTrue(laby.objets.contains(amulette), "L'amulette
    doit tre pr sente dans les objets.");
}

@Test
public void testAmulettePeutPartagerCaseAvecPJ() {
laby.objets.add(new Amulette(2, 2));

assertEquals(17, laby.pj.getX());
assertEquals(12, laby.pj.getY());
assertNotNull(laby.objetPresent(2, 2), "L'amulette doit
    tre sur la m me case que le PJ.");
}

@Test
public void testAmulettePeutPartagerCaseAvecMonstre() {
Monstre monstre = new Monstre(3, 3, 100);
laby.monstres.add(monstre);
laby.objets.add(new Amulette(3, 3));

assertEquals(monstre, laby.monstrePresent(3, 3));
assertNotNull(laby.objetPresent(3, 3), "L'amulette doit
    tre sur la m me case que le monstre.");
}
```

```
@Test
public void testTirInitialiseDansLaBonneDirection() {
laby.pj.setDerniereDirection(Labyrinthe.DROITE);
laby.tir();

assertEquals(1, laby.listProjectile.size());
Projectile p = laby.listProjectile.get(0);
assertEquals(17, p.getX());
assertEquals(12, p.getY());
assertEquals(Labyrinthe.DROITE, p.getDirection());
}

@Test
public void testProjectileAvanceCorrectement() {
laby.pj.setDerniereDirection(Labyrinthe.GAUCHE);
laby.tir();
laby.gererTir();  // avance d'une case

Projectile p = laby.listProjectile.get(0);
assertEquals(16, p.getX());
assertEquals(12, p.getY());
}

@Test
public void testProjectileDisparaitQuandMur() {
laby.murs[6][5] = true;  // mur     droite du joueur
laby.pj.setDerniereDirection(Labyrinthe.DROITE);
laby.tir();

laby.gererTir();
assertTrue(laby.listProjectile.isEmpty(), "Le projectile
    devrait dispara tre en touchant un mur.");
}

@Test
public void testProjectileDisparaitEtBlesseMonstre() {
Monstre monstre = new Monstre(6, 5, 3);  // monstre avec
    3 HP     droite
laby.monstres.add(monstre);

laby.pj.setDerniereDirection(Labyrinthe.DROITE);
laby.tir();
laby.gererTir();  // tir vers le monstre

assertTrue(laby.listProjectile.isEmpty(), "Le projectile
    devrait dispara tre apr s avoir touch  le monstre
   .");
assertEquals(3, monstre.getVie(), "Le monstre devrait
```

```
        perdre 1 point de vie.");
    }

    @Test
    public void testProjectileContinueTantQuePasObstacle() {
    laby.pj.setDerniereDirection(Labyrinthe.GAUCHE);
    laby.tir();

    // avance 3 fois
    laby.gererTir(); // x = 6
    laby.gererTir(); // x = 7
    laby.gererTir(); // x = 8

    Projectile p = laby.listProjectile.get(0);
    assertEquals(14, p.getX());
    assertEquals(12, p.getY());
    }

    }
```

6 Itération 4

6.1 Choix des fonctionnalités

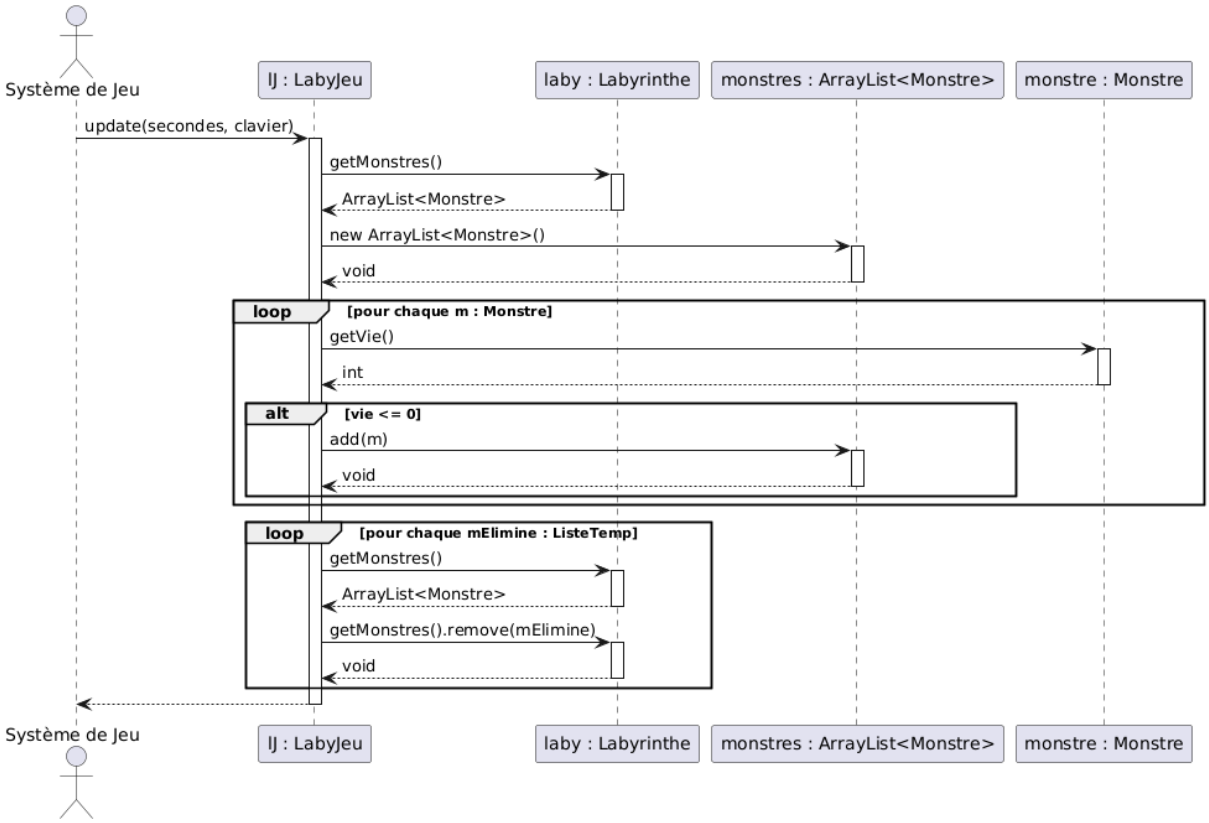
1. **Mort des monstres**
   — *Descriptif :* Lorsqu'un monstre n'a plus de points de vie, il meurt et est retiré du jeu.
   — *Critères de validation :*
      — Les monstres ont 2 points de vie par défaut.
      — Les points de vie ne peuvent pas être négatifs.
      — Un monstre meurt après avoir été touché deux fois.
      — Le héros attaque en premier : si le monstre meurt, il ne peut pas riposter.
2. **Acquisition de l'amulette**
   — *Descriptif :* Lorsque le héros entre sur la case contenant l'amulette, il la ramasse automatiquement.
   — *Critères de validation :*
      — L'amulette disparaît du plateau.
      — Le héros possède l'amulette.
      — Un monstre ne peut pas récupérer l'amulette.
3. **Fin du jeu : victoire du héros**
   — *Descriptif :* Une fois l'amulette en sa possession, le héros peut retourner à l'entrée du labyrinthe pour remporter la partie.
   — *Critères de validation :*
      — Si le héros possède l'amulette et revient à l'entrée, le jeu se termine et un message est affiché dans la console.
      — Si le héros revient à l'entrée sans amulette, le jeu continue.

6.2 Conception

6.2.1 Suppression des monstres morts

Lors de l'évolution du jeu, les monstres ayant une vie inférieure ou égale à zéro doivent être supprimés du labyrinthe. Pour cela, la classe `LabyJeu` récupère la liste des monstres depuis le labyrinthe, puis construit une nouvelle liste contenant uniquement ceux qui sont morts.

Cette logique est décrite dans le diagramme de séquence suivant :
— `LabyJeu` récupère la liste des monstres via `getMonstres()`.
— Pour chaque monstre, la méthode `getVie()` est appelée.
— Si la vie est inférieure ou égale à zéro, le monstre est ajouté à une liste temporaire.
— Tous les monstres de cette liste sont ensuite supprimés du labyrinthe.



6.2.2 Acquisition de l'amulette

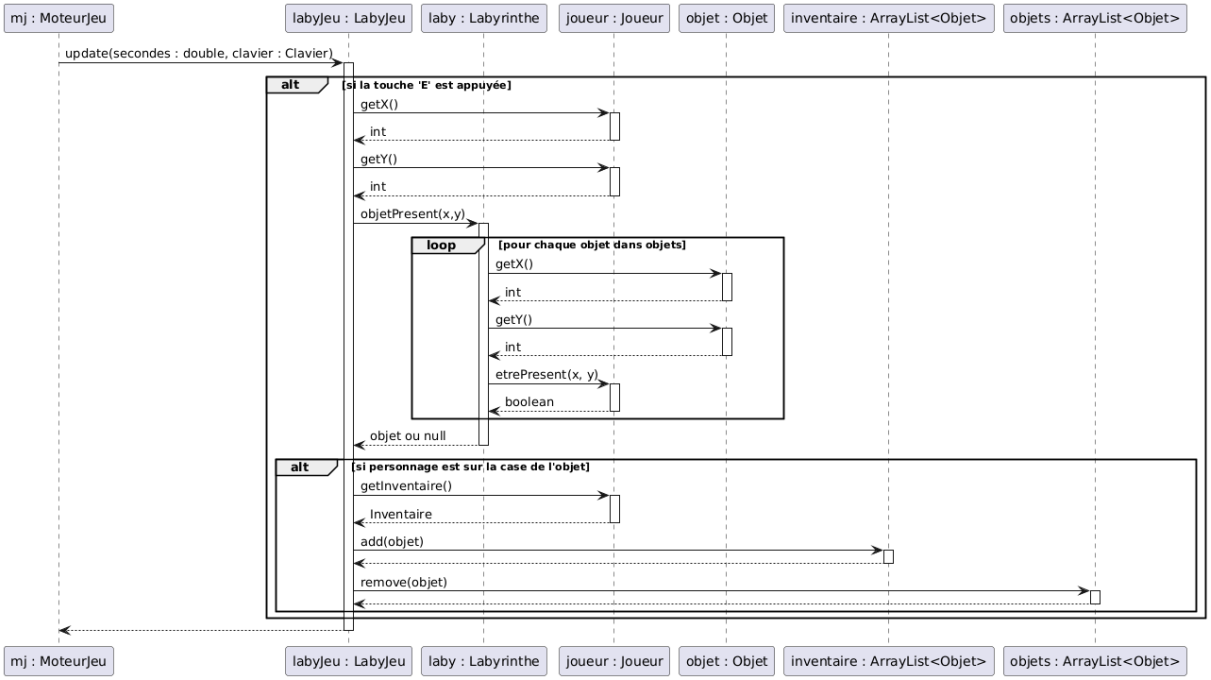
Le héros peut ramasser une amulette présente sur sa case en appuyant sur la touche d'interaction ("E"). La vérification de présence d'un objet à la position du joueur est effectuée à chaque mise à jour.

Fonctionnement :

— Lors de l'appel à `update`, si la touche "E" est appuyée, les coordonnées du joueur sont récupérées.

— La méthode `objetPresent(x, y)` du labyrinthe est appelée pour vérifier s'il y a un objet sur la case.

— Si un objet est trouvé, il est ajouté à l'inventaire du joueur, puis retiré de la liste des objets du labyrinthe.



6.2.3 Fin du jeu : victoire du héros

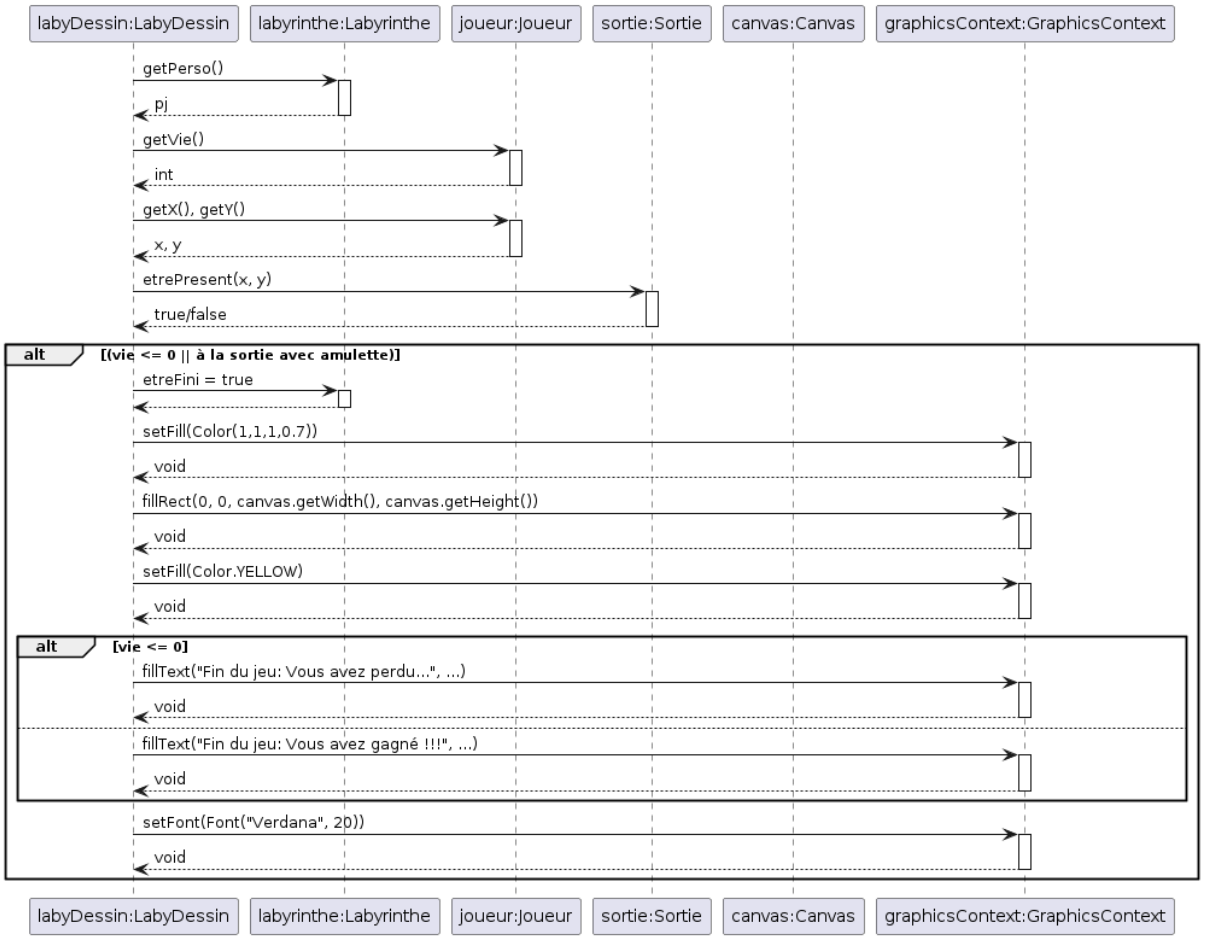
Le jeu se termine lorsque le héros possède l'amulette et atteint la sortie du labyrinthe. Cette vérification est effectuée lors du dessin de la scène par la classe `LabyDessin`.

Logique appliquée :

— Les coordonnées du joueur sont comparées à celles de la sortie.

— Si le joueur est sur la sortie et possède l'amulette, le jeu est déclaré comme terminé.

— Un message de victoire est affiché à l'écran.



6.3 Réalisation

6.3.1 Suppression des monstres morts

Pour éviter que des monstres morts restent dans la liste principale du labyrinthe, une liste temporaire `aElim` est utilisée pour collecter les monstres dont la vie est inférieure ou égale à zéro. Cette double boucle permet d'éviter les erreurs liées à la suppression directe d'éléments d'une collection en cours d'itération.

```
ArrayList<Monstre> aElim = new ArrayList<>();
for (Monstre m : labyrinthe.getMonstres()) {
    if (m.getVie() <= 0) {
        aElim.add(m);
    }
}
for (Monstre m : aElim) {
    labyrinthe.getMonstres().remove(m);
}
```

6.3.2 Acquisition de l'amulette (ou d'un objet)

Lorsque le joueur appuie sur la touche d'interaction ('E'), on vérifie si un objet est présent à sa position. Si oui, l'objet est ajouté à l'inventaire du joueur, puis retiré de la liste d'objets du labyrinthe.

```
if (clavier.prendre) {
    Objet o;
    if ((o = labyrinthe.objetPresent(labyrinthe.pj.getX(),
       labyrinthe.pj.getY())) != null){
        labyrinthe.pj.getInventaire().add(o);
        labyrinthe.objets.remove(o);
    }
}
```

6.3.3 Fin du jeu : condition de victoire ou de défaite

Lors du dessin de la scène, si la vie du héros tombe à zéro ou si ce dernier atteint la sortie avec l'amulette, la variable `etreFini` du labyrinthe est activée. Un message de fin est alors affiché graphiquement via le `GraphicsContext`.

```
if (laby.pj.getVie() <= 0 ||
    (laby.sortie.etrePresent(pj.getX(), pj.getY()) &&
       amuletteRecup)) {

    laby.etreFini = true;

    gc.setFill(new Color(1, 1, 1, 0.7));
    gc.fillRect(0, 0, canvas.getWidth(), canvas.getHeight());
    gc.setFill(Color.YELLOW);
    String message = "Fin du jeu: ";
    if (laby.pj.getVie() <= 0) {
        message += "Vous avez perdu...";
    } else {
        message += "Vous avez gagn  !!!";
    }
    gc.fillText(message, canvas.getWidth() / 2 - 100, canvas.
       getHeight() / 2);
    gc.setFont(new Font("Verdana", 20));
}
```

6.4 Intégration et validation

Fonctionnalité 5.5 – Mort des monstres

Des tests unitaires ont été écrits pour valider que les monstres ne peuvent pas avoir de points de vie négatifs, qu'ils disparaissent correctement après avoir subi deux attaques, et qu'ils ne peuvent plus attaquer s'ils sont déjà morts.

- Un monstre ayant 1 point de vie subit plusieurs dégâts : sa vie ne descend jamais en dessous de 0.
- Après deux attaques, la vie du monstre atteint 0, ce qui valide sa mort.
- Un monstre tué avant de commencer son attaque ne doit pas exécuter l'action de riposte.

Fonctionnalité 6.3 – Acquisition de l'amulette

L'acquisition de l'amulette est testée par la simulation d'un appui sur la touche `E`. On vérifie que l'amulette est bien retirée du plateau et ajoutée à l'inventaire du joueur uniquement lorsque le joueur est sur la même case qu'elle et que la touche est pressée.

- Si la touche `E` est enfoncée et que le joueur est sur la case de l'amulette, elle est ajoutée à l'inventaire et supprimée du plateau.
- Si la touche `E` n'est pas pressée, aucun objet n'est ramassé.
- Si le joueur n'est pas sur la même case que l'amulette, elle reste sur le plateau.

Fonctionnalité 6.4 – Fin du jeu : victoire du héros

Les tests valident que le jeu se termine lorsque le joueur atteint la sortie en possession de l'amulette, et qu'il ne se termine pas si l'un des deux critères n'est pas rempli.

- Si le joueur est sur la sortie et que l'amulette a été ramassée, le jeu se termine (`etreFini` est à `true`).
- Si l'amulette est encore sur le plateau, le jeu ne se termine pas même si le joueur est sur la sortie.
- Si le joueur n'est pas sur la sortie, il ne peut pas gagner même s'il possède l'amulette.

7 Itération 5

7.1 Choix des fonctionnalités

9. **Génération automatique de labyrinthe**
   — *Descriptif :*
   Au lieu de charger un niveau à partir d'un fichier texte prédéfini, le labyrinthe est généré dynamiquement à chaque partie. Cela garantit des configurations différentes à chaque exécution, ce qui améliore la rejouabilité du jeu. La génération repose sur un algorithme qui construit un labyrinthe connexe, sans murs bloquants entre la position de départ et la sortie.
   — *Critères de validation :*
   — Le labyrinthe généré est toujours connexe : le joueur peut atteindre la sortie.
   — Le héros est placé sur une case vide valide, non murée.
   — L'amulette est également placée sur une case vide différente de celle du héros.
   — Les monstres sont positionnés aléatoirement, uniquement sur des cases accessibles.
   — Aucun élément (joueur, objet ou monstre) ne se retrouve bloqué ou inaccessible.
   — La disposition des murs respecte une structure cohérente (pas de murs flottants ou de chemins totalement bloqués).

7.2 Conception

La génération automatique du labyrinthe repose sur un algorithme de parcours en profondeur récursif (DFS – Depth First Search), initialisé dans la classe `Labyrinthe` lorsque le paramètre aléatoire est activé. L'objectif de cet algorithme est de créer dynamiquement une structure de labyrinthe connexe avec des chemins accessibles.

7.2.1 Initialisation des structures de génération

Lorsque le mode aléatoire est activé :
— Un tableau de booléens nommé `visite` est créé pour mémoriser les cellules déjà explorées.
— Toutes les cellules de la grille sont initialement marquées comme des murs (`murs[x][y] = true`).
— Le tableau `visite[x][y]` est rempli à `false`, indiquant qu'aucune cellule n'a encore été visitée.

7.2.2 Parcours récursif en profondeur (DFS)

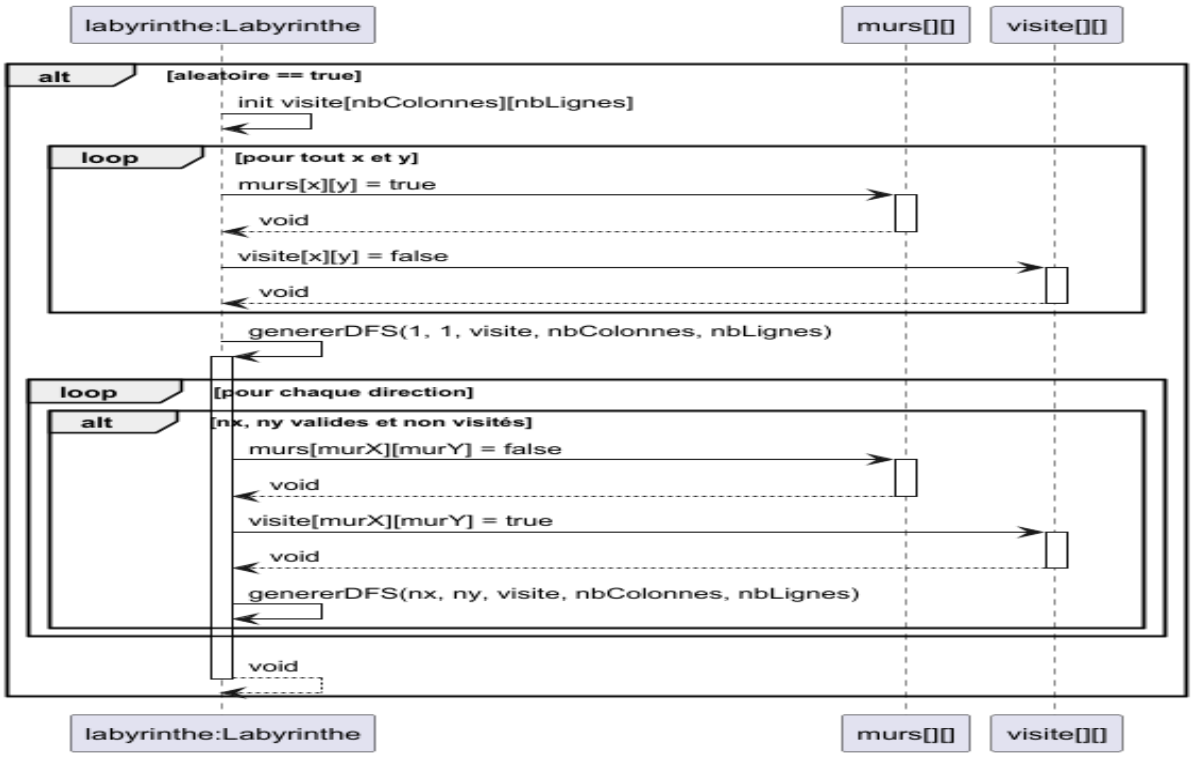
L'algorithme démarre depuis une cellule de départ (typiquement `(1,1)`). Il procède de manière récursive comme suit :
— Pour chaque direction possible (haut, bas, gauche, droite), on calcule la position de la cellule voisine.
— Si cette cellule est dans les bornes et non encore visitée, on retire le mur entre la cellule actuelle et la cellule voisine.

— On marque la cellule voisine comme visitée, puis on applique récursivement le même processus à partir de cette cellule.

Ce processus assure que toutes les cellules accessibles forment un graphe connexe (pas de pièces isolées), ce qui garantit la générabilité du niveau.

7.2.3 Avantages de cette conception

— Simplicité d'implémentation : l'algorithme DFS est simple, efficace et bien adapté à la génération de labyrinthes.
— Rejouabilité élevée : à chaque partie, un nouveau labyrinthe est généré, créant ainsi une expérience unique.
— Maîtrise de la complexité : la profondeur du labyrinthe peut être ajustée en contrôlant les dimensions et la densité initiale.



7.3 Réalisation

Simplification des objets du jeu

Dans cette itération, la classe `Objet` a été supprimée afin de simplifier la gestion des objets dans le labyrinthe. En effet, l'unique objet manipulé dans le jeu étant l'amulette, il a été décidé de remplacer la hiérarchie d'objets par une gestion directe des `Amulette`. Ainsi, l'inventaire du joueur contient désormais directement des instances de `Amulette`, et la liste d'objets du labyrinthe (`labyrinthe.objets`) devient une liste d'amulettes. Cette refonte permet de clarifier le code et d'éviter une abstraction inutile, tout en gardant une structure évolutive si de futurs objets devaient être introduits.

Pour permettre la génération automatique d'un labyrinthe, une nouvelle logique a été intégrée à la classe `Labyrinthe`. Lorsqu'un niveau est chargé avec le mode aléatoire activé, le fichier texte n'est pas utilisé pour placer les murs. À la place, un algorithme récursif de génération est invoqué.

7.3.1 Generation aléatoire du labyrinthe

Initialisation

Lorsque le mode `aleatoire` est activé :

— Un tableau de booléens `visite` de taille `[nbColonnes][nbLignes]` est créé pour suivre les cellules déjà explorées.
— Toutes les cellules du labyrinthe sont initialisées comme des murs en positionnant `murs[x][y] = true`.
— La génération commence à la cellule `(1,1)`, choisie volontairement pour être une cellule impaire afin de respecter la structure des murs entre cases.

Méthode `genererDFS`

La méthode `genererDFS(int x, int y, boolean[][] visite, int nbColonnes, int nbLignes)` implémente un parcours récursif en profondeur (DFS). Voici les principales étapes :

— Marquer la cellule `(x,y)` comme visitée et la rendre accessible en supprimant le mur.
— Définir les directions possibles de déplacement (haut, bas, gauche, droite) avec un pas de 2 cases pour sauter les murs.
— Mélanger aléatoirement les directions pour créer un labyrinthe différent à chaque exécution.
— Pour chaque direction, si la cellule voisine est dans les bornes et non visitée :
  — Supprimer le mur entre la cellule courante et la voisine.
  — Appeler récursivement `genererDFS` sur la cellule voisine.

Cette méthode garantit que l'ensemble du labyrinthe est connecté, tout en assurant un haut degré de variété entre les parties.

Modifications de code

— Modification de la classe `Labyrinthe` pour inclure un constructeur ou une logique conditionnelle permettant de basculer entre lecture de fichier et génération.
— Ajout de la méthode privée `genererDFS()`.
— Suppression de la lecture classique du fichier dans le cas où le mode aléatoire est activé.

7.3.2 Modification général du code

— Changement de la gestion des couleurs du monstre. Le monstre à maintenant un attribut couleur qui est modifié lorsqu'il attaque et arrête d'attaquer. Ca couleur est ensuite mise à jour graphiquement grâce à dessinerJeu().

7.4 Intégration et validation

Pour valider la génération automatique du labyrinthe, un test unitaire a été mis en place dans la classe `TestIteration5`. Ce test vérifie que deux labyrinthes générés successivement à partir du même fichier source en mode aléatoire produisent des résultats différents.

Test

- Deux instances de `Labyrinthe` sont créées à partir du même fichier texte avec le mode aléatoire activé.
- Chaque instance génère un labyrinthe différent en appelant l'algorithme `genererDFS`.
- Le test vérifie que l'appel à `toString()` sur chaque labyrinthe renvoie une représentation différente, prouvant que la génération est bien aléatoire.

```
@Test
public void testGeneration() throws IOException {
    Labyrinthe l1 = new Labyrinthe("labySimple/laby0.txt", true)
        ;
    Labyrinthe l2 = new Labyrinthe("labySimple/laby0.txt", true)
        ;
    assertNotEquals(l1.toString(), l2.toString());
}
```

Bugs rencontrés et corrections

- **Accès aux bornes du tableau** : Une erreur hors-limite pouvait survenir si les coordonnées étaient mal vérifiées. Des conditions de type `nx > 0 && ny > 0 && nx < nbColonnes - 1 && ny < nbLignes - 1` ont été ajoutées.
- **Cellules inaccessibles** : Sans respecter un pas de 2 cases lors du déplacement, certaines cellules n'étaient jamais reliées. Le pas a été corrigé pour garantir une connectivité correcte.
- **Labyrinthe vide** : En oubliant de supprimer les murs à la position de départ, le labyrinthe pouvait rester entièrement fermé. Un appel explicite à `murs[x][y] = false` a été ajouté au début de `genererDFS`.

Ce test simple mais efficace assure que la génération produit des labyrinthes uniques et correctement connectés à chaque lancement.