

---

Projet *Interface d'un jeu d'exploration*  
(Rapport)

---

Auteur(s) : Florian Legendre, Alexis Louail,  
Vincent Tourenne



# Table des matières

<b>I</b>	<b>Manuel du joueur</b>	<b>2</b>
0.1	Qu'est-ce que Silent In Space ? . . . . .	3
0.2	Lancer le jeu . . . . .	3
0.3	Les contrôles du jeu . . . . .	4
0.4	Carte du jeu et Solution . . . . .	6
0.5	Recommandations pour le confort du jeu . . . . .	7
<b>II</b>	<b>Manuel du développeur</b>	<b>8</b>
0.6	Diagramme de classe du modèle . . . . .	9
0.7	Structuration de l'application . . . . .	10
0.8	Notre démarche dans la construction de cette application . . . . .	11
0.9	Résolution des problèmes de conception . . . . .	12
0.10	Analyse des possibilités d'extension de l'application . . . . .	15
0.11	Erreurs détectées et commentaires . . . . .	16

Première partie

Manuel du joueur

## Qu'est-ce que Silent In Space ?

Silent In Space (abrégé : "SIS") est un jeu Point & Click s'inspirant des Monkey Island. Dans ce jeu, vous vous réveillez dans un vaisseau alien et avez pour but de vous en échapper. Faites-vous des alliés ou des ennemis sur votre chemin et découvrez pourquoi vous êtes ici !

## Lancer le jeu

Le jeu se lance dans un IDE de type IntelliJ IDEA ou NetBeans. Il nécessite **Java 15** et **JavaFX 16**.

Si vous choisissez de le lancer dans un IDE IntelliJ vous devez :

1. Aller dans la structure du projet (File > Project Structure)
2. Ajouter la librairie - ie. le dossier lib/ - de JavaFX (Libraries > + > Java > lib/)
3. Aller dans les paramètres d'exécution (Run > Edit Configurations)
4. Cliquer sur l'application "Main" et ajouter dans le champ *VM options* : `--module-path "../librairies/javafx/lib" --add-modules=``javafx.controls,javafx.fxml`

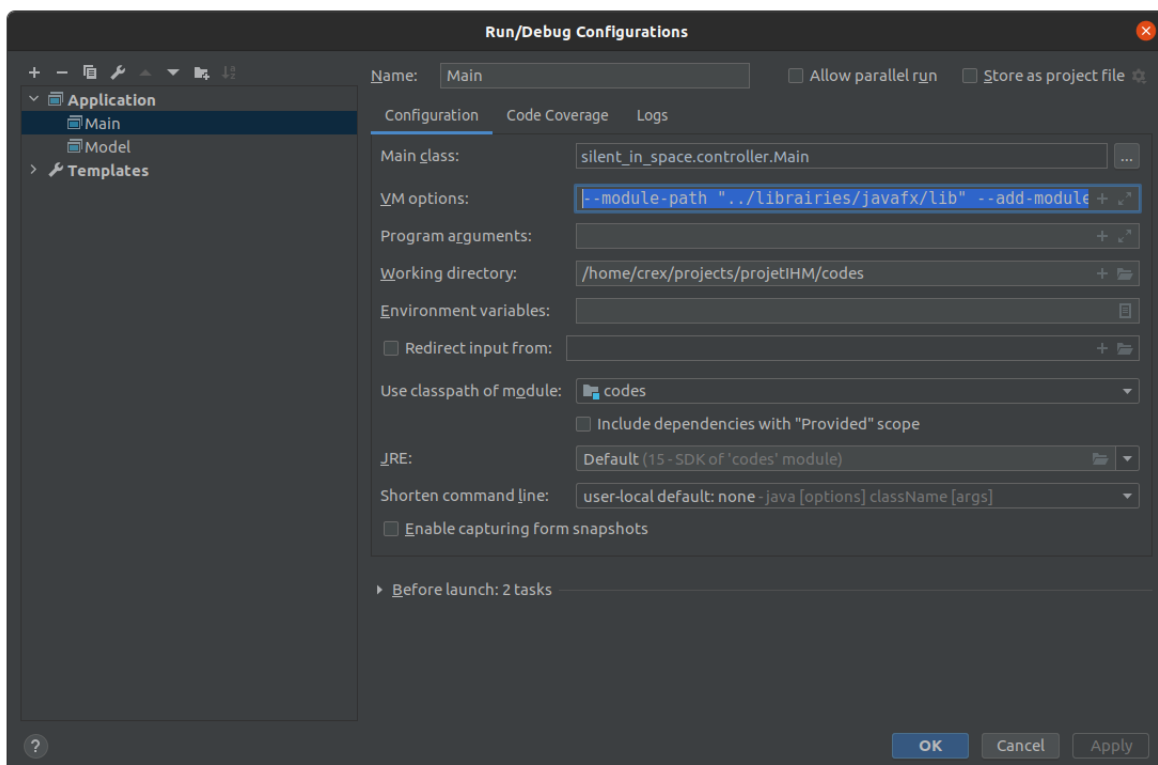


FIGURE 1 – Configuration du Runner d'IntelliJ pour JavaFx

## Les contrôles du jeu

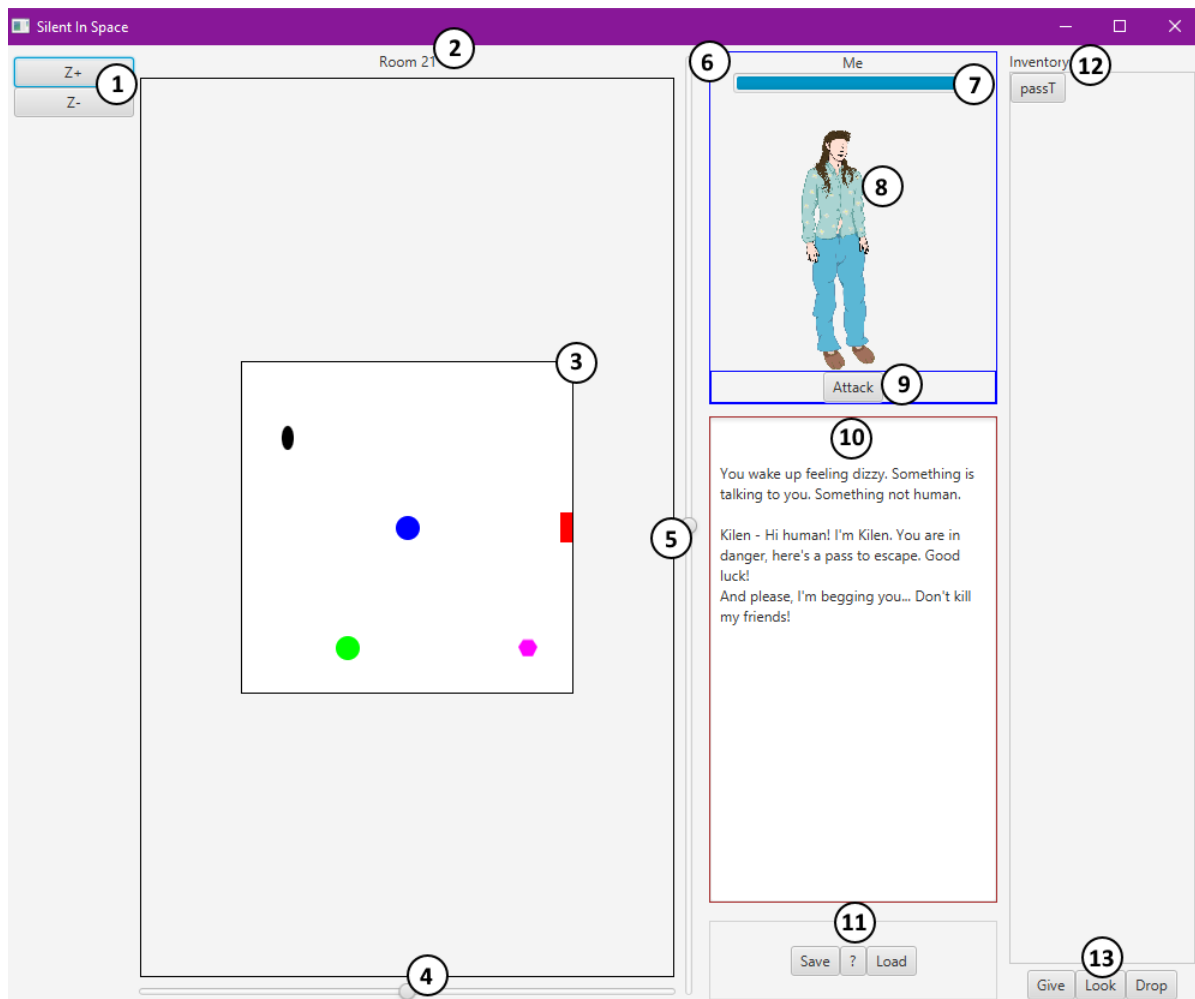


FIGURE 2 – L’interface graphique de l’application (avec annotations)

1. Fonctions de Zoom sur la carte
2. Numéro de la pièce actuelle
3. Panneau de la Carte de Jeu. C’est ici qu’évoluera la vue des pièces et leurs contenus
4. Slider permettant de déplacer la vue sur l’axe horizontal
5. Slider permettant de déplacer la vue sur l’axe vertical
6. Panneau des personnages
7. Barre de vie du personnage sélectionné
8. Illustration du personnage sélectionné
9. Bouton d’Attaque
10. Panneau de Dialogues. C’est ici que sera affiché le texte en jeu
11. Boutons de Sauvegarde et de Chargement. Appuyer sur “?” fera apparaître le manuel du jeu.
12. Panneau de l’Inventaire. Chaque nouvel item ramassé par le joueur apparaît ici.
13. Boutons relatifs à l’Inventaire. “Give” donne un item à un personnage, “Look” donne une description d’un item, et “Drop” lâche un item sélectionné au sol.

## Carte de Jeu

La carte du jeu **(3)** affiche le contenu de chaque pièce. Un clic droit sur n'importe quel élément (item, personnage, porte, etc.) en affichera la description dans le panneau des dialogues **(10)**.

Utiliser les sliders **(4 et 5)** permet de déplacer la vue de la pièce et cliquer sur les boutons *Z+* et *Z-* **(1)** permettent de zoomer et dézoomer.



## Interactions avec les portes

Les portes sont symbolisées par un rectangle noir ou rouge placé contre un mur. Leurs localisations dépendent de l'emplacement de la pièce à laquelle cette porte conduit. Si la porte est à droite sur la carte du jeu **(3)**, la pièce à laquelle elle mène se trouve à droite.

Les portes noires sont déverrouillées et les portes rouges sont verrouillées. Cliquer sur une porte déverrouillée permet de passer à la pièce à laquelle elle mène. Une porte verrouillée ne peut pas être empruntée si elle n'a pas été déverrouillée au préalable avec un pass ou un ordinateur.

## Interactions avec les objets

Il existe deux types d'objets avec lesquels il est possible d'interagir :

- ceux représentés par un hexagone rose (stations) 
- ceux représentés par une ellipse noire (items) 

## Stations

Les stations sont des objets que le joueur ne peut pas ramasser. Faire un clic gauche dessus permet de les utiliser suivant la fonction qui leur a été assignée préalablement (obtenir une information en lisant un panneau, se soigner, etc.).

## Items

Les items peuvent être ramassés par le joueur avec un simple clic gauche. Un item ramassé est ensuite affiché dans l'inventaire, dans la partie droite de l'écran **(12)**.

## Interactions avec un personnage

Vous pouvez sélectionner un personnage en faisant un clic gauche sur son icône sur la carte **(3)**. Lorsque vous sélectionnez un personnage, son portrait apparaît dans le panneau des personnages **(6)** et vous lui parlez. Une fois cela fait, vous pouvez l'attaquer (bouton *Attack*, **9**) ou lui donner un objet (bouton *Give*, **13**).

## Interactions avec l'Inventaire

L'inventaire **(12)** affiche tous les objets obtenus par le joueur. Avec les boutons *Give*, *Look* et *Drop* **(13)**, le joueur peut manipuler ceux-ci de plusieurs manières.

## Utiliser un item sur un objet

Pour cela il faut cliquer sur l'item dans l'inventaire **(12)**, puis cliquer sur l'objet ciblé dans la vue **(3)**. Cela permet par exemple de déverrouiller une porte.

## Donner un item à un personnage

Il faut sélectionner le personnage à qui donner l'objet, sélectionner l'objet puis cliquer sur le bouton *Give* **(13)**.

## Boutons Look et Drop

Pour utiliser ces boutons, il suffit de sélectionner l'item dans l'inventaire **(12)** puis de cliquer sur le bouton voulu **(13)**.

## Carte du jeu et Solution

### Légende de la carte du jeu

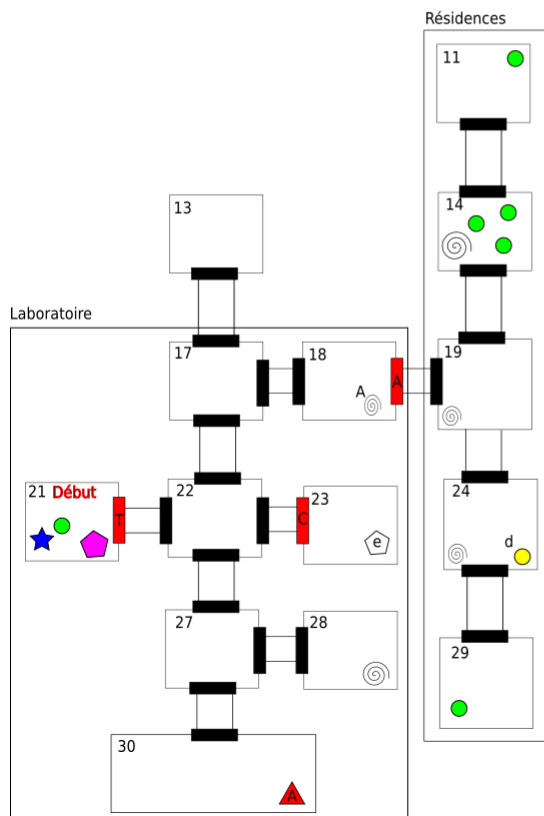
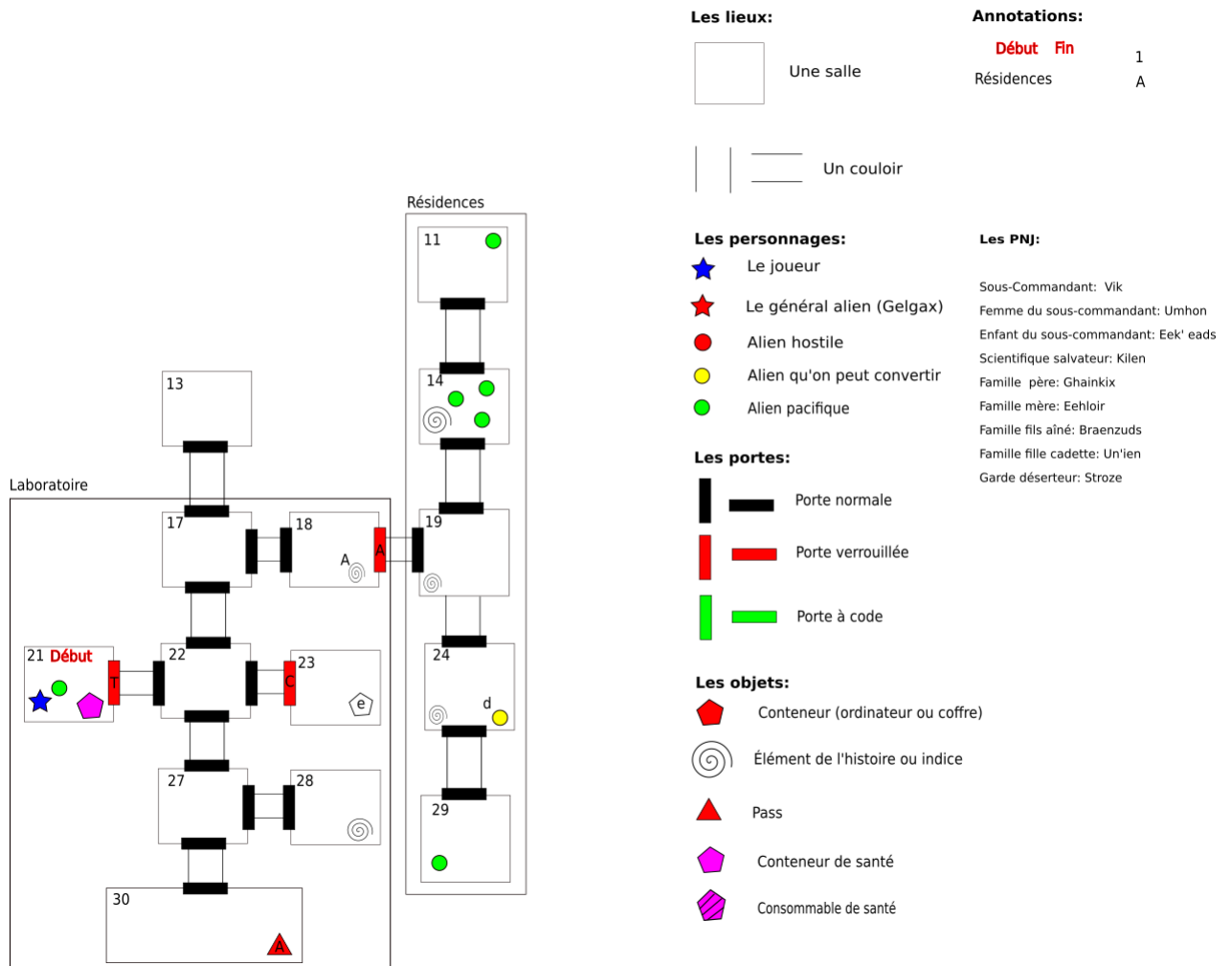


FIGURE 3 – Carte du jeu

Commencez par utiliser le passT sur la porte T et sortez de la pièce de départ. Rendez-vous en salle 30, puis récupérez le pass A. Vous pouvez maintenant vous rendre en pièce 18, et débloquent l'aile résidentielle du vaisseau.

Pour finir le jeu, vous aurez besoin de récupérer le code de l'ordinateur du capitaine. Il se trouve que la personne qui le possède n'est autre qu'Umhon, en salle 24. Cependant elle vous l'échangera uniquement contre une preuve des expériences faites sur les humains sur le vaisseau. Vous avez alors deux options pour l'obtenir :

- Tuer Umhon et récupérer le code sur son cadavre.
- Aller en salle 23. La porte se fermera automatiquement derrière vous. Utilisez l'ordinateur pour la débloquent, ainsi que pour imprimer le document "doctorLog". Retournez voir Umhon et donnez-lui ce document.

Vous devez ensuite vous rendre dans la salle 13 pour déclencher la fin du jeu.

## Recommandations pour le confort du jeu

Une carte du jeu sans les solutions vous est fournie. Vous pouvez partager votre écran en deux parties comme dans l'image ci-dessous. Ainsi il vous sera plus facile de vous orienter dans le vaisseau spatial et de compléter les objectifs!

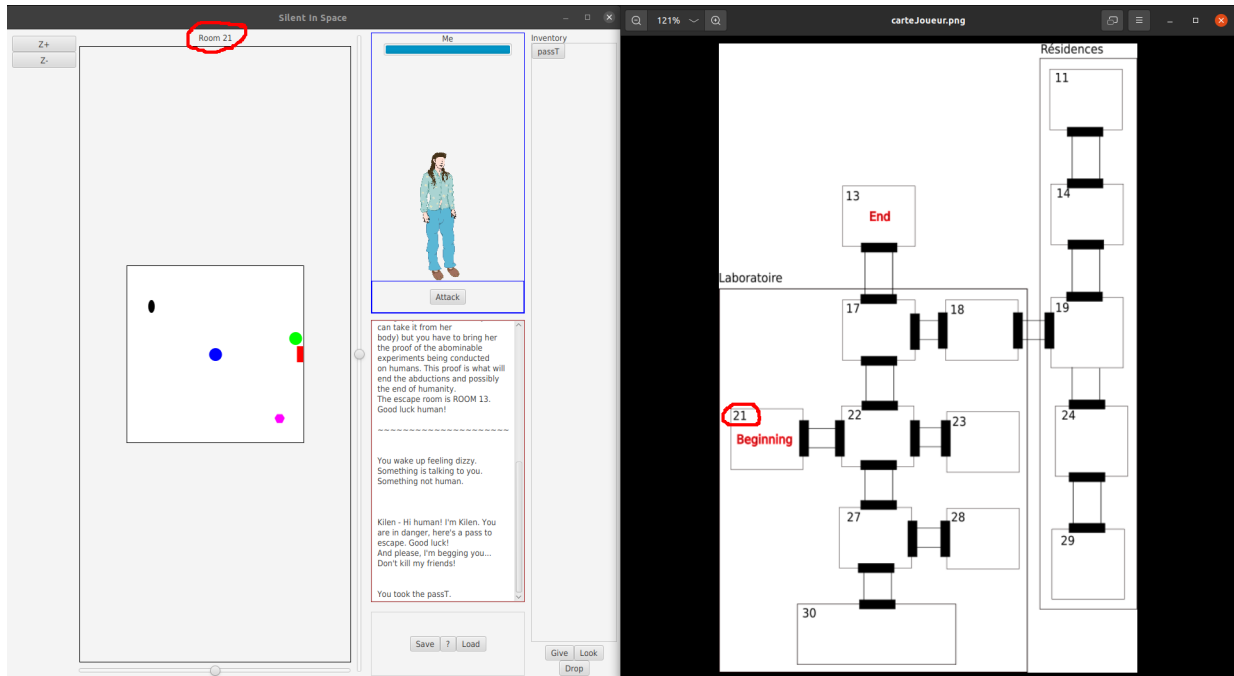


FIGURE 4 – Les indicateurs entourés en rouge vous aideront à vous orienter à tout moment dans le jeu

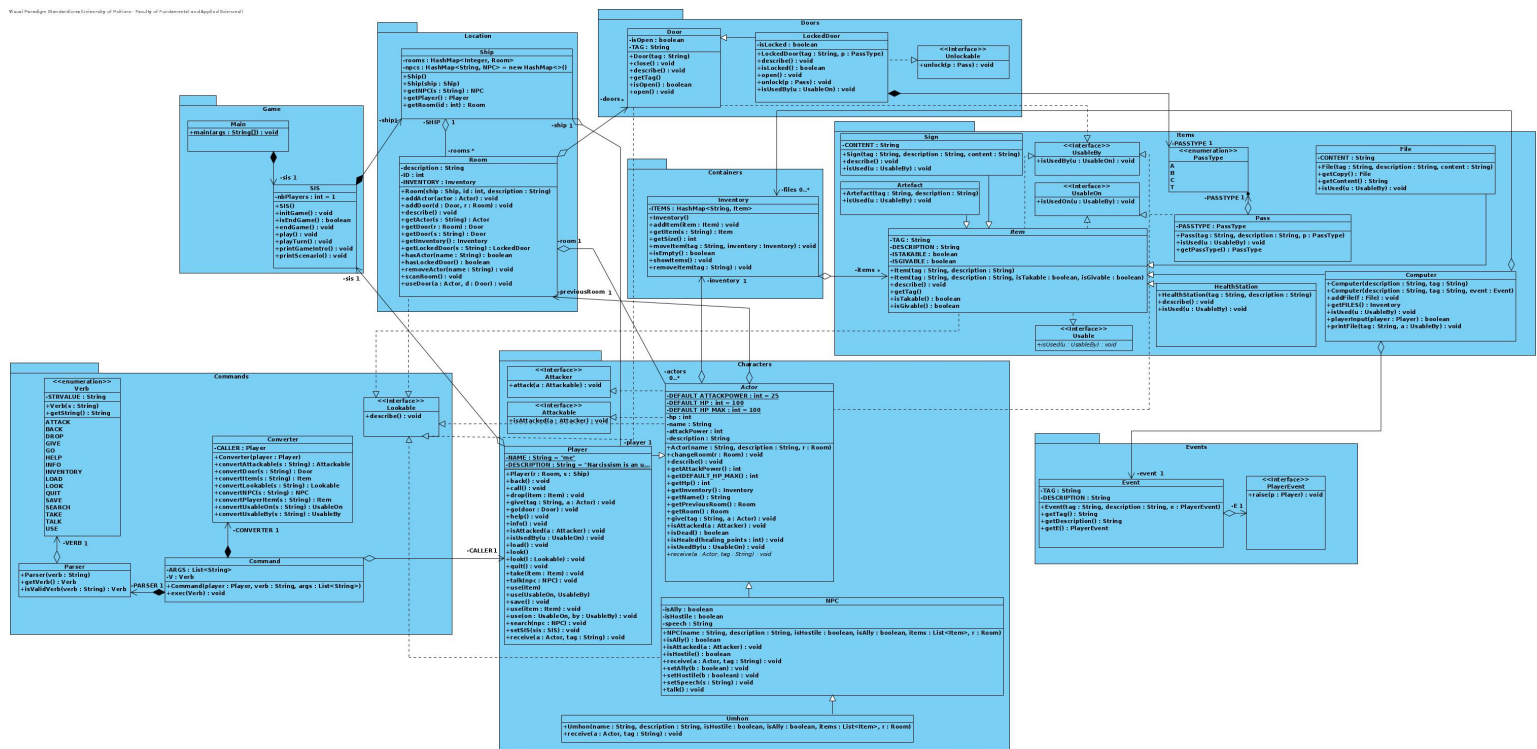


Deuxième partie

Manuel du développeur

Bien que le modèle ne soit pas le coeur de ce projet nous pensons que présenter ce-dernier vous permettra de mieux comprendre les quelques adaptations que nous avons dû y apporter pour l'application actuelle.

Experimental and Applied Polymer Symp.



**FIGURE 5** – Le diagramme en taille originale est fournie dans le dossier docs/ de l'archive du projet

Ainsi, nous avons rajouté un nouveau package : le package *Utils*, qui n'est pas représenté sur ce schéma et dont l'unique classe *Scalar2D* est composée dans les classes *Actor*, *Item* et *Door*. Cette classe sert à indiquer des positions proportionnelles (aucune unité de mesure ne leurs sont associées) des objets/acteurs/portes les uns par rapport aux autres.

Une interface et une classe abstraite statique ont été ajoutées au package *Game* : l'interface *MessageListener* et la classe abstraite statique *Message*.

La classe abstraite statique *Message* a été conçue en nous inspirant de la fonction *System.out.print()* : accessible partout et non instantiable. L'interface *MessageListener* est inspirée des écouteurs de javaFX. Nous avons ainsi notre propre système d'écoute des messages du modèle. Les classes qui le souhaitent s'y abonnent peu importe si elles sont dans un modèle, une vue ou encore ce qu'elles font de ces messages.

## Structuration de l'application

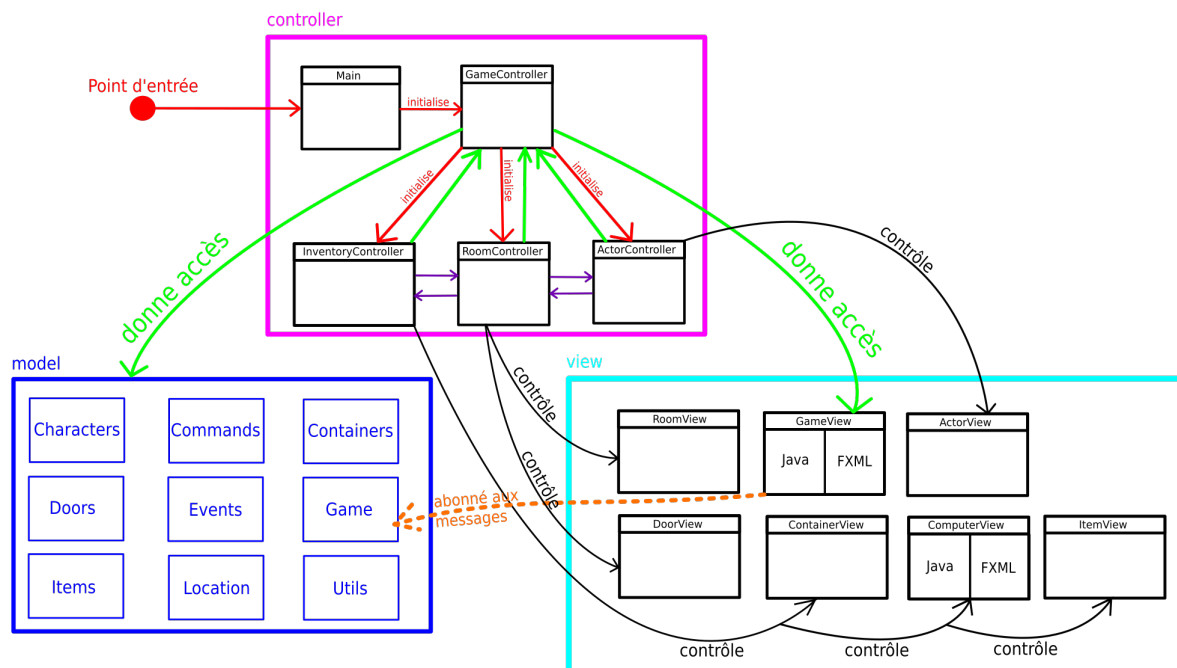


FIGURE 6 – Schéma de conception de la structure de l'application

Le jeu se lance avec le *Main* du package *controller*. Ce *Main* initialise le contrôleur général du jeu, appelé *GameController*. Ce contrôleur lui-même lance trois autres contrôleurs spécialisés : le *ActorController*, le *InventoryController* et le *RoomController*.

Le but du *GameController* est d'initialiser les gestionnaires d'événements globaux du jeu (manuel d'aide et fin de la partie) et d'offrir une porte d'accès au modèle et à la vue du jeu pour les trois autres contrôleurs plus spécialisés.

Aussi, les trois autres contrôleurs ont tous la main sur la vue du jeu (*GameView*) et sur d'autres éléments visuels dont ils sont spécialistes (comme la vue des pièces et des portes - respectivement *RoomView* et *DoorView* - pour le *RoomController*)

Enfin, ces trois contrôleurs peuvent avoir besoin de communiquer entre eux. Ceci est le cas, par exemple, pour les fonctions *drop()* et *take()* de l'*InventoryController* qui doivent nécessairement connaître la pièce dans laquelle il doivent ajouter ou retirer un élément visuel. Quand ils communiquent ils le font toujours en passant par le *GameController* afin d'avoir les données les plus à jour du modèle et de la vue.<sup>1</sup>

L'objectif de cette structuration était de recréer des mini-modèles MVC au sein d'un modèle MVC plus général. Cette structuration devait permettre d'éviter un contrôleur de jeu trop imposant et d'améliorer la lisibilité du code.

Pour terminer, nous noterons l'abonnement de la vue du jeu (*GameView*) aux messages du jeu par le biais d'une interface *MessageListener* présente dans le package *Game*. La classe abstraite statique *Message* du modèle envoie les chaînes de caractère du jeu au *MessageListener* enregistré. Ceci nous permettait de continuer de tester le modèle dans la commande (pour y trouver d'éventuels bugs) tout en ne changeant rien au fonctionnement général du jeu.

1. Par souci de lisibilité du schéma, on a représenté cette communication par des flèches violettes directes sur la figure 6

## Notre démarche dans la construction de cette application

Nous avons commencé le développement dès le lancement du projet le 20 Mars. Ayant un modèle que nous avons beaucoup testé le semestre précédent et qui ne comportait pas de bugs à notre connaissance, nous avons débuté sur le développement de la vue générale du jeu. Nous avons choisi FXML parce que ce choix répondait au cahier des charges et aussi parce que nous jugions que c'était l'outil le plus simple pour gérer la création d'une vue aussi complexe.

Nous avons ensuite longuement hésité sur la marche à suivre. Nous pensions que le contrôleur associé au document FXML était un "véritable" contrôleur. Aussi nous avons commencé à diviser la vue en 6 FXML différents (un pour la carte du jeu, un autre pour le panneau des acteurs, etc.)

En fin de compte il nous a paru plus simple de concevoir le contrôleur associé au FXML comme un moyen d'accéder dans Java aux éléments de la vue FXML. Il s'agissait donc moins d'un contrôleur et bien plus de la "partie Java" de la vue. Nous avons adopté ce point de vue suite à un entretien avec M. Bergey.

### Répartition du travail

Suite à ce travail de conception en collectif nous nous sommes répartis le travail entre nous :

- Florian devait réaliser le chargement automatique des pièces (afin d'éviter de coder en dur les vues des 30 pièces du jeu final). Il devait également réaliser la fonction de zoom sur les pièces, les translations avec les sliders et le *binding* du label (cf. élément **(2)** de l'UI présentée ci-dessus).
- Alexis Louail et Vincent Tourenne devaient s'occuper des interactions du joueur avec les aliens et lui-même. Ils devaient également s'occuper de l'implantation en GUI de la fonction de sauvegarde et de chargement du jeu.
- L'implantation de l'UI de l'inventaire devait être réalisé collectivement à la fin de l'implantation des points précédemment évoqués. Cette implantation comprenait les fonctions de prise d'un objet sur le sol, d'affichage de la description des objets et de *give()*, *use()*, *drop()*.

Cependant, certaines difficultés dont nous faisons la liste ci-après, nous ont amené à revoir cette répartition :

- Florian a implanté ses fonctionnalités et celles de l'inventaire.
- Alexis Louail et Vincent Tourenne ont implanté leur part.

### Obstacles et difficultés du développement

Durant le développement de ce projet nous avons dû faire face à de nombreuses difficultés :

- Les examens de fin d'année et les autres projets universitaires dont les dates de rendu étaient très proches
- Un début de stage en 35h dès le 12 Avril pour Florian Legendre
- Des difficultés personnelles pour Florian Legendre (deuil dans son entourage)
- Des déplacements réguliers ainsi que des difficultés personnelles et matérielles pour Alexis Louail
- Des difficultés liées au partage du code sur l'IDE IntelliJ et aux différentes versions de Java/JavaFx avec Git (fichiers de configuration .idea/)
- Nos incompréhensions et fausses routes propres au travail d'un étudiant

### Outils utilisés pour le développement et la gestion de ce projet

L'objectif dans le choix de ces outils était de nous permettre de travailler le plus efficacement possible. Aussi nous avons choisi :

1. De versionner notre code sur Git
2. De nous le partager via un dépôt privé GitHub
3. De communiquer régulièrement et en temps réel par des salons vocaux ou par messages sur Discord
4. De nous répartir les bugs détectés via la fonctionnalité des tickets de GitHub

## Résolution des problèmes de conception

Au cours du développement de cette application nous avons dû résoudre un nombre important de problèmes de conception. Certains d'entre eux ont déjà été détaillé ci-dessus et nous ne faisons donc que les mentionner ici :

- La conception du modèle (semestre précédent).
- L'adaptation du modèle (positions relatives des objets/acteurs/portes du modèle et abonnements aux messages du jeu).
- La structuration globale de l'architecture de l'application en nous basant sur l'architecture MVC tout en essayant d'avoir un code le plus modulaire possible. Cela nous a permis d'éviter d'avoir un seul gros contrôleur et plusieurs sous-contrôleurs spécialisés dont les méthodes sont des briques réutilisables pour construire des méthodes plus complexes.

Parmi les autres problèmes de conception les plus notables que nous avons eus à résoudre et que nous n'avons pas encore expliqués, il y a :

1. Le chargement automatique des pièces.
2. La gestion de l'utilisation des objets.
3. La gestion du don d'objet.
4. La gestion de l'ordinateur du jeu.

### Chargement automatique des pièces

```
1 public void updateRoomView(int nbCol, int nbLignes) {
2     //On récupère le modèle:
3     currentRoomModel = gameController.getPlayerModel().getRoom();
4
5     //On met à jour la vue:
6     gameController.getGameView().getMapPane().getChildren().remove(
currentRoomView);
7     currentRoomView = new RoomView(nbCol, nbLignes);
8     gameController.getGameView().getRoomLabel().setText("Room " +
currentRoomModel.getID());
9     loadDoors();
10    loadItems();
11    loadPlayer();
12    loadNPCs();
13    loadHandlers();
14    gameController.getGameView().getMapPane().getChildren().add(
currentRoomView);
15
16    //On signale à l'inventaire de mettre à jour la taille du tableau
17    //de gestionnaires d'événements utilisé pour la gestion de la
18    //fonction use() des objets (quand on clique gauche sur un objet):
19    gameController.getInventoryController().resetUseItemHandlersArray(
nbCol, nbLignes);
20
21    // "Éteint" l'ordinateur si le joueur quitte la pièce sans appuyer
22    // sur le bouton 'quitter':
23    gameController.getActorController().resetActorPanel();
24
25    //À chaque nouvelle pièce chargée on vérifie si le jeu est terminé:
26    gameController.isGameOver();
27 }
```

**Listing 1** – Méthode de mise à jour de la vue des pièces extraite du *RoomController*

On peut remarquer plusieurs problèmes à cette conception du chargement automatique :

- La vue n'est pas vraiment mise à jour : elle est détruite par l'initialisation d'une toute nouvelle vue.
- Certaines méthodes normalement du ressort de l'*InventoryController*, de l'*ActorController* ou du *GameController* sont laissées à la charge du *RoomController* dans cette méthode de mise à jour de la vue. Bien qu'il s'agissait de solutions simples pour résoudre certains problèmes, nous avons conscience que nous ne pouvons pas résoudre tous les problèmes du jeu en réinitialisant tout à chaque changement de pièce.

### Gestion de l'utilisation des objets

Pour la gestion de l'utilisation des objets, deux options ont été considérées :

1. Un bouton *Use* présent aux côtés du bouton *Give*, *Drop* et *Look*. Quand le joueur sélectionne un des *ToggleButton* de l'inventaire, le *string* du label de ce dernier est stocké dans un tableau. Le joueur aurait ensuite cliqué sur un élément visuel de la pièce ce qui aurait ajouté un second *string* dans le tableau. Lorsque le joueur aurait ensuite cliqué sur le bouton *Use*, ce bouton aurait appelé la fonction *use()* du modèle en cherchant les éléments du modèle associés aux *strings* stockés dans le tableau
2. Nous n'utilisons pas de bouton *Use*. Lorsque le joueur sélectionne un des *ToggleButton* de l'inventaire, des gestionnaires d'événements sont attachés à tous les éléments visuels présents dans la pièce. Ces gestionnaires d'événements appellent la fonction *use()* du modèle avec le *string* de l'objet sélectionné en paramètre. Afin de gérer ces gestionnaires d'événements temporaires nous avons utilisé un tableau nommé *fireHandlers* qu'on nettoyait dès que l'objet était désélectionné ou utilisé ou que le joueur changeait de pièce.

La dernière solution, certe plus complexe, a été choisie car elle nous semblait plus ergonomique. Cependant, elle a aussi introduit de la complexité dans notre code et une gestion supplémentaire d'un tableau que l'on retrouve notamment dans la méthode de mise à jour de la vue des pièces présentées ci-dessus.

```
public class InventoryController {  
    private final GameController gameController;  
    private final ToggleGroup invTG;  
    private EventHandler<MouseEvent>[] fireHandlers;
```

FIGURE 7 – Le tableau *fireHandlers* permettant de gérer les gestionnaires d'événements temporaires

### Gestion du don d'objet

La gestion du don d'objet utilise deux informations :

1. L'acteur sélectionné (cf. élément (6) de l'UI, figure 2).
2. L'objet sélectionné (le *ToggleButton* dans le panneau de l'inventaire, élément (12) de l'UI).

L'étiquette du personnage est récupérée dans la vue de l'acteur sélectionné et l'étiquette de l'objet dans le *ToggleButton* sélectionné de l'inventaire. Un gestionnaire d'événement est alors associé au bouton *Give* qui fait appel à la méthode *give()* du modèle avec ces deux informations.

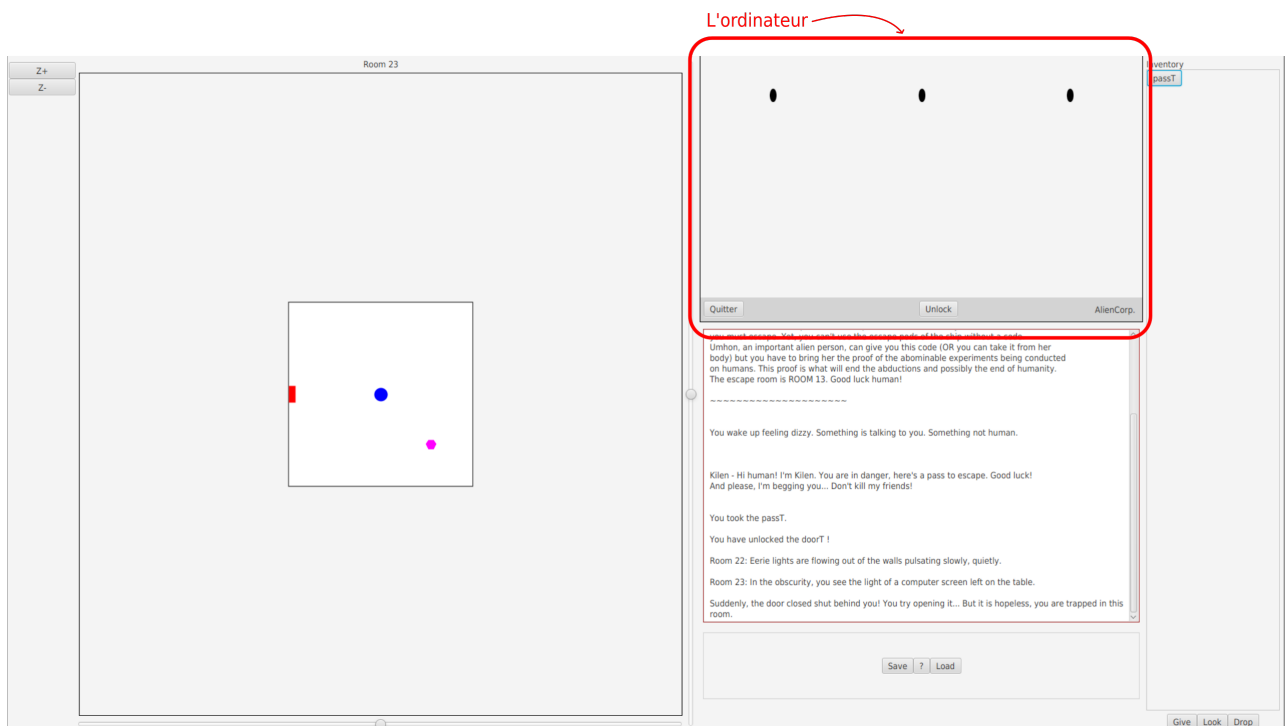
Aussi notre solution ressemble beaucoup à la première solution envisagée pour l'utilisation des objets. Le tableau de stockage des étiquettes en moins car n'étant pas nécessaire à la résolution de ce problème.

## Gestion de l'ordinateur du jeu

Enfin, notre jeu présentait un élément spécial : un ordinateur. Dans le modèle, cet ordinateur se comportait comme un terminal de commande dans le terminal de commande. L'ordinateur contient des fichiers que le joueur peut consulter et/ou en emporter des copies. Il contient également des "programmes" permettant de déclencher des événements dans le jeu (comme l'ouverture de la porte dans la pièce 23).

Étant donné que dans le modèle la "vue" choisie était celle d'un terminal dans le terminal, il nous a paru naturel de lui donner une traduction graphique "d'ordinateur dans l'ordinateur" pour cette application.

Aussi l'ordinateur a son propre mini-MVC. Lorsque l'ordinateur est utilisé par le joueur, le contrôleur de ce mini-MVC est appelé et remplace le panneau de l'acteur comme illustré dans la figure 8 :



**FIGURE 8** – Illustration de la vue de l'ordinateur dans le jeu, les fichiers réutilisent la vue des items du jeu et sont représentés par des ellipses noires

La solution de la fenêtre popup a été considérée et aurait permis de simplifier la gestion de la fermeture de l'ordinateur lorsque le joueur quitte une pièce sans avoir cliqué sur le bouton 'quitter'. Cependant, nous tenions à éviter l'utilisation des fenêtres popups car nous jugions qu'elles cassaient un peu le rythme du jeu.

Cette gestion est donc actuellement assurée par la méthode *updateRoomView* du *RoomController* présenté précédemment.

La gestion complète de l'ordinateur est inachevée et fait partie des axes d'amélioration et des possibilités d'extension de l'application dont nous voulons maintenant faire l'état dans la section suivante de ce rapport.

# Analyse des possibilités d'extension de l'application

## Les axes d'amélioration

Le présent rendu est un travail inachevé. Les premiers points d'achèvement que nous souhaiterions aborder sont :

- Davantage clarifier la responsabilité de chaque contrôleur
- Renforcer la modularité du code
- Éliminer les quelques données laissées en dur dans le code
- Optimiser certaines méthodes en évitant de mettre à jour toute la pièce mais seulement quelques éléments de la pièce par le biais de méthodes *load()* et de méthodes *unload()*
- Revoir le système de conversion des objets du modèle en leurs vues respectives

Dans l'état actuel des choses nous pensons pouvoir produire le jeu final que nous espérons. En effet :

- Le chargement des pièces est automatique, aussi l'ajout d'une pièce dans le modèle provoque l'ajout d'une pièce dans la vue
- La vue est abonnée aux messages du jeu, aussi l'ajout de dialogues dans le jeu est automatiquement répercuté dans la vue
- 100% des fonctions du modèles sont implantées et ont leurs équivalents dans la présente interface

Développer l'application finale sans retravailler les points énoncés ci-dessus nous amènerait probablement à produire un code lourd (notamment si on ajoute de nouveaux objets) dans son exécution et peu lisible.

De plus, il faudrait renforcer le nombre de cas possibles : il est prévu dans le modèle des pièces qui ne comportent pas nécessairement 4 portes. Pour l'ordinateur nous n'avons pas prévu de comportement dans le cas où le nombre de fichiers excéderaient la capacité de la vue, etc.

## Les fonctionnalités futures envisagées

En dernier lieu, quand tous ces points trouveront leurs implantations dans l'application, nous pensions améliorer le visuel du jeu en utilisant de nouveaux *assets*, en utilisant des services qui modifieraient la couleur des pièces selon un gradient et qui donneraient ainsi un effet de pulsation lumineuses au vaisseau, peut-être même en préchargeant les pièces voisines pour aider le joueur à s'orienter, etc.

D'autres éléments d'ergonomie peuvent être reconsidérés : le bouton *Give* n'est pas très pratique à utiliser et il aurait pu être placé à côté du bouton *Attack* sous le personnage avec lequel le joueur interagit. Les boutons ZOOM+ et ZOOM- pourraient être remplacés (ou complétés) par un *binding* sur la molette de la souris du joueur. Les sliders pourraient aussi être remplacés (ou complétés) par un drag & drop de la pièce avec la souris, etc.



## Erreurs détectées et commentaires

Au cours du développement nous avons détecté de nombreuses erreurs que nous avons corrigé en nous répartissant le travail par le biais de tickets (figure 9).

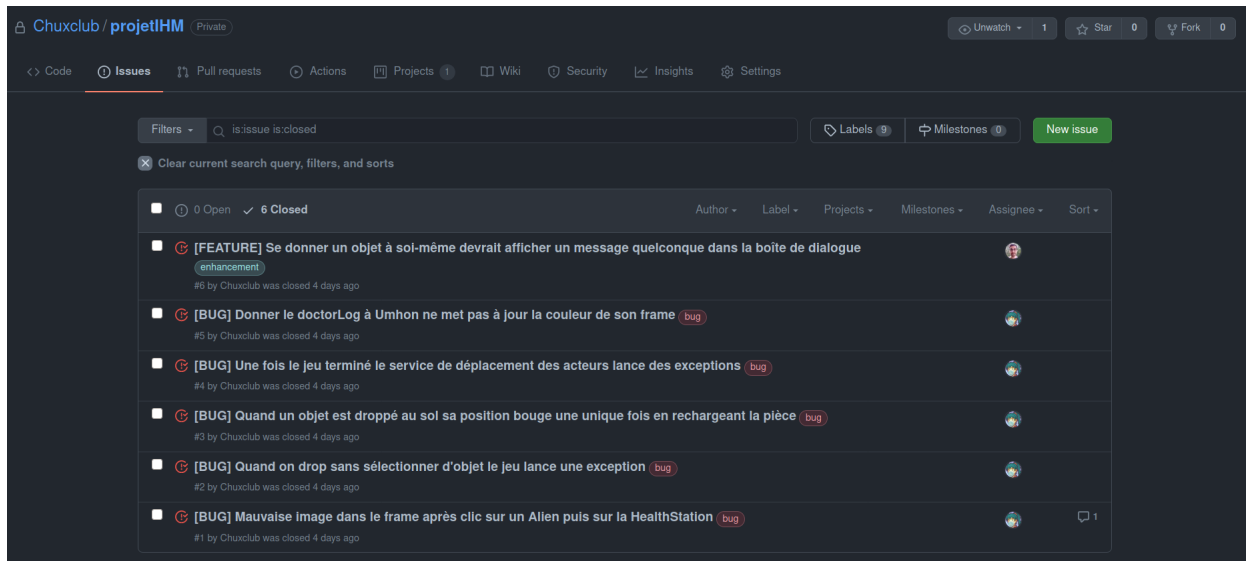


FIGURE 9 – Des tickets de bug sous GitHub

Ces erreurs étaient détectées soit au cours de nos propres tests de l'application, soit en laissant l'application aux mains de nos proches. Ces-derniers sont en remerciement dans la fenêtre popup de fin de partie.

Au moment du rendu de cette application nous n'avons pas détecté de nouveaux bugs.