UNIVERSITÉ DE SHERBROOKE
Faculté de génie
Département de génie électrique et génie informatique

Méthodes de développement avancées

Architecture et construction d'un logiciel

Présenté à
Ruben Gonzalez-Rubio

Présenté par
Benjamin Roy - royb2006
Marc-Antoine Beaudoin – beam2039
Maxime Dupuis – dupm2216
Soukaina Nassib – nass2801

Sherbrooke – 16 septembre 2018

Table des matières

1. Réunion de la planification de la réalisation du projet

Pour identifier les fonctionnalités du jeu et bien saisir le fonctionnement de ce dernier, la visualisation des récits utilisateurs s'avère une nécessité. Ainsi, dans la première réunion de l'équipe, nous avons commencé un remue-méninge pour pouvoir identifier les acteurs du jeu et leurs besoins. Ensuite et après l'identification des éléments clés, nous avons développé chaque point à part.

Le résultat de ce travail est synthétisé dans les figures 1, 2 et 3. Dans la première figure, nous avons décrit le scénario du côté joueur. Puis et pour avoir une vue de l'intérieur du jeu, nous avons focussé sur les acteurs Pac-Man et fantômes (figure 2 et 3).

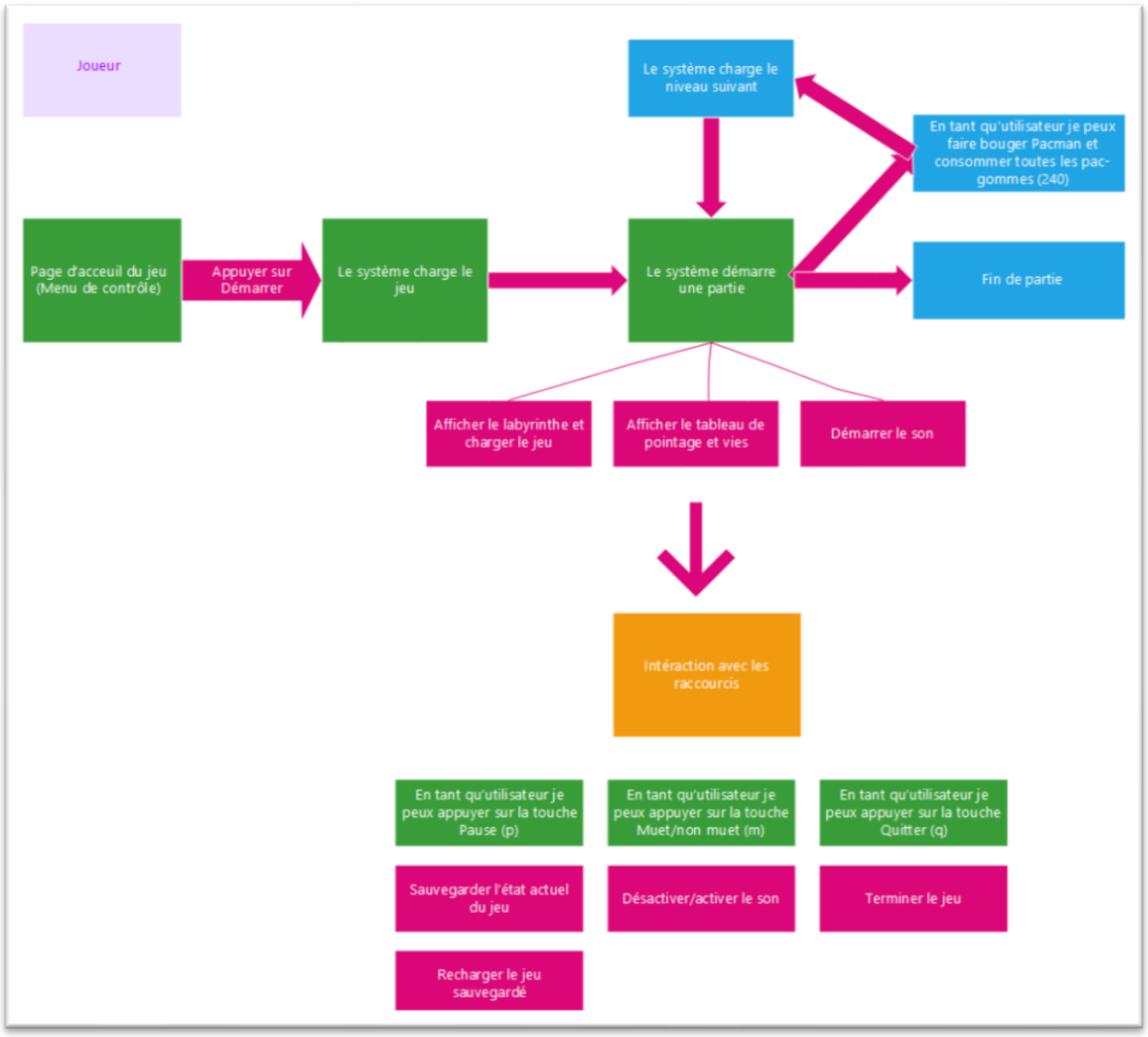

Figure 1: Récits utilisateur - Joueur

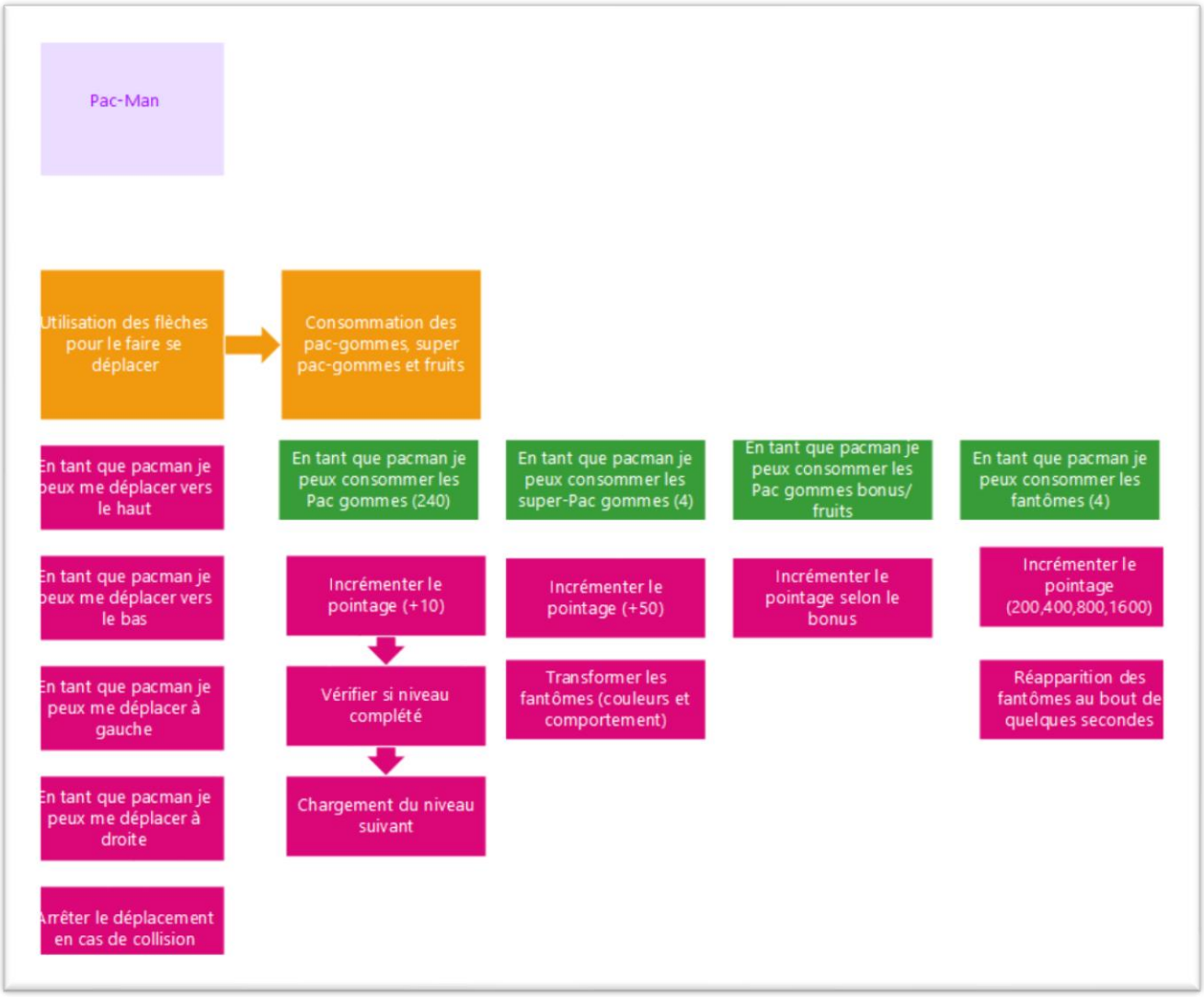

Figure 2: Récits utilisateur – PacMan

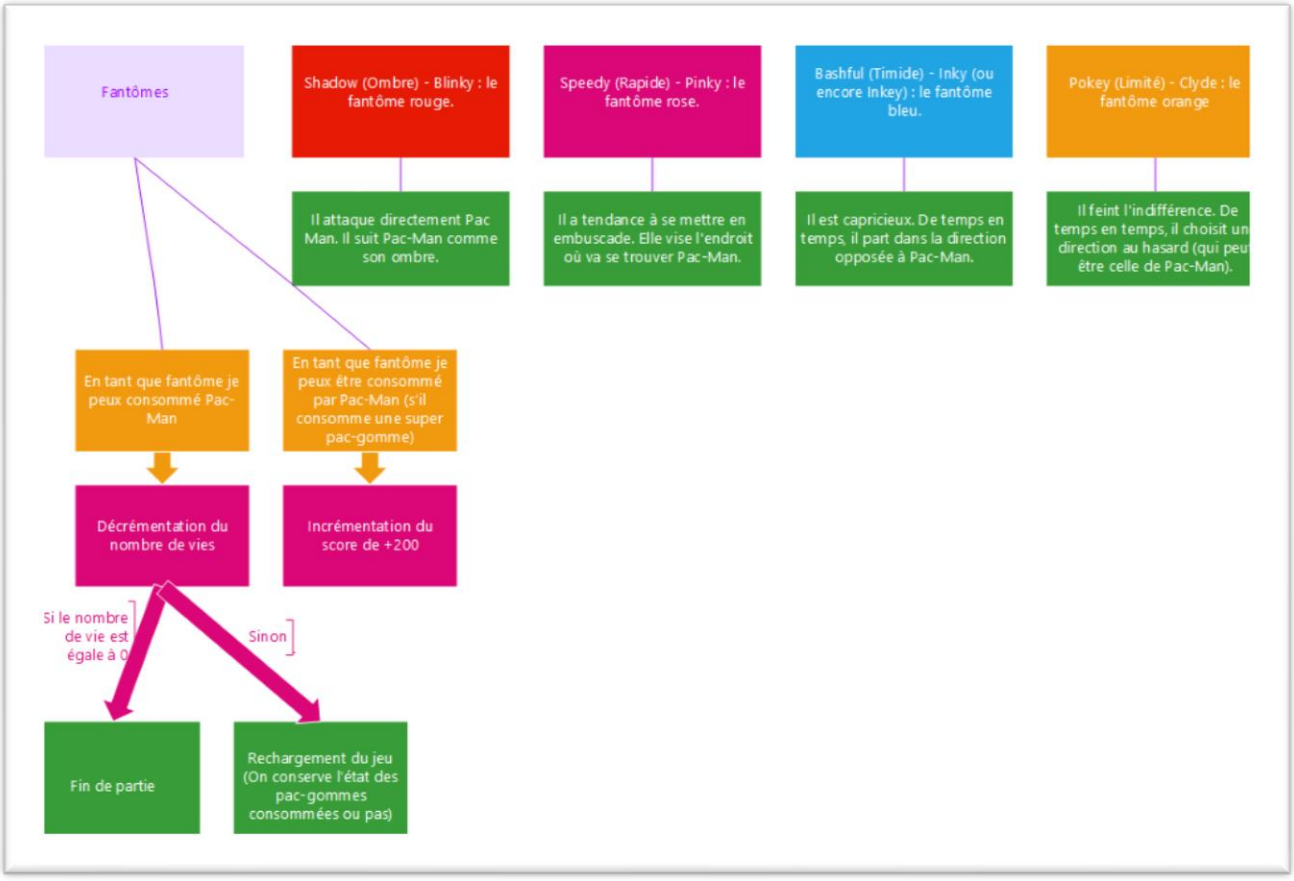

Figure 3: Récits utilisateur – Fantômes

2. Description de l'architecture

2.1 Diagramme de classes UML complet

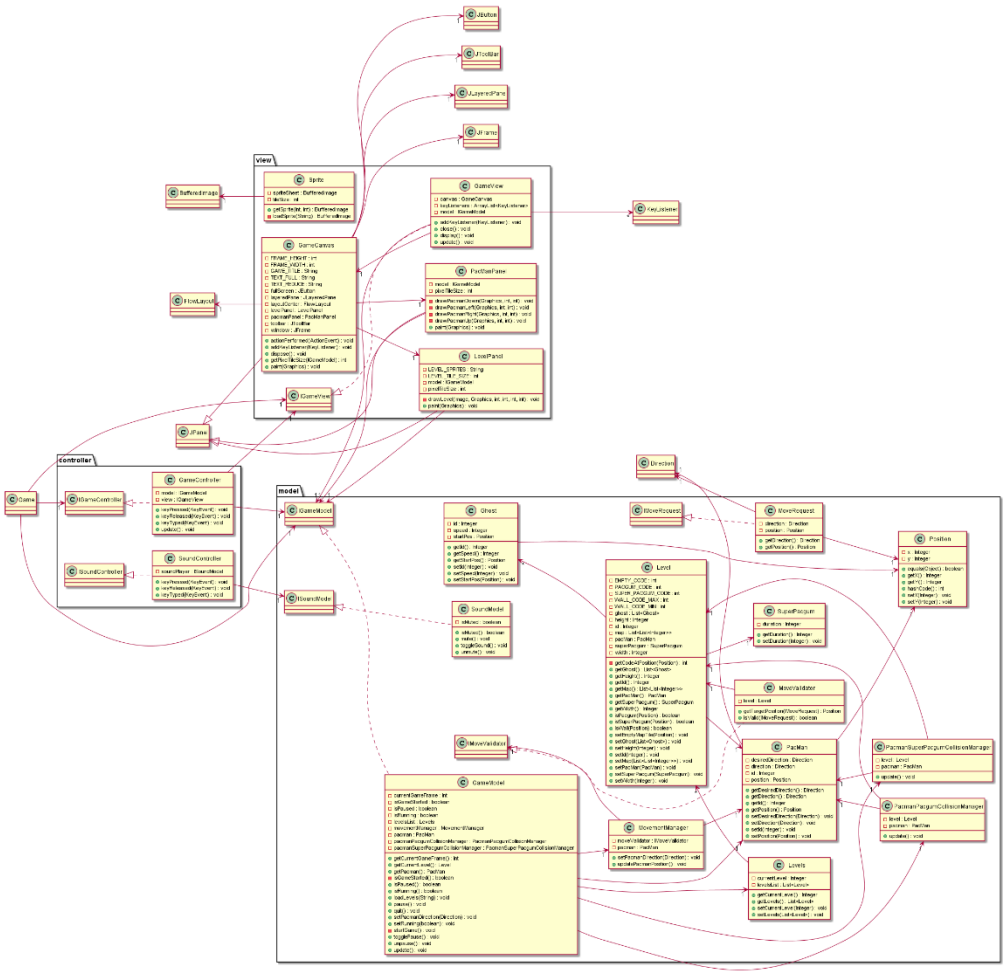

Figure 4: Diagramme UML de l'application

2.2 Diagrammes UML du modèle seulement

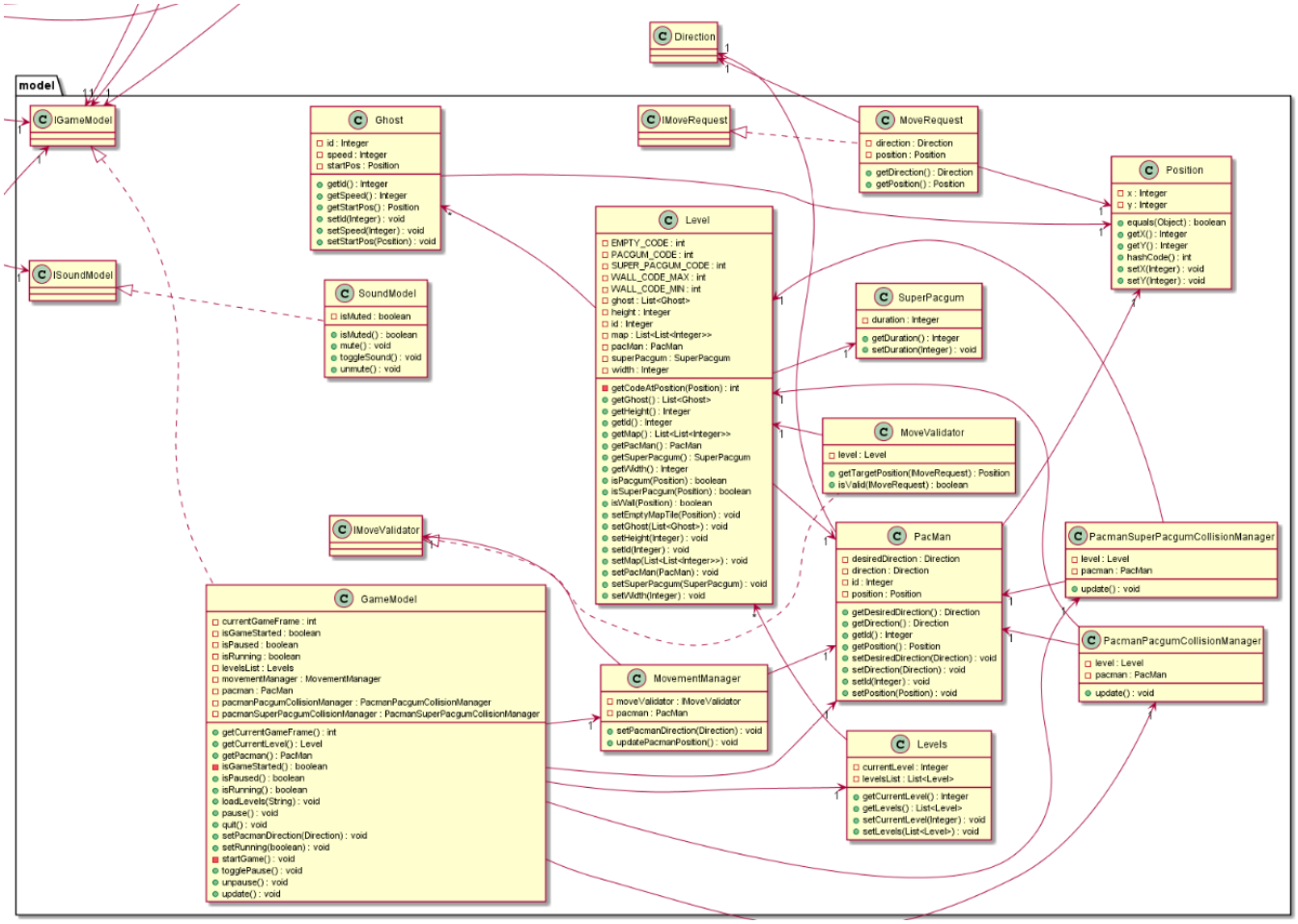

Figure 5: Diagramme UML du modèle

2.3 Diagramme UML de la vue seulement

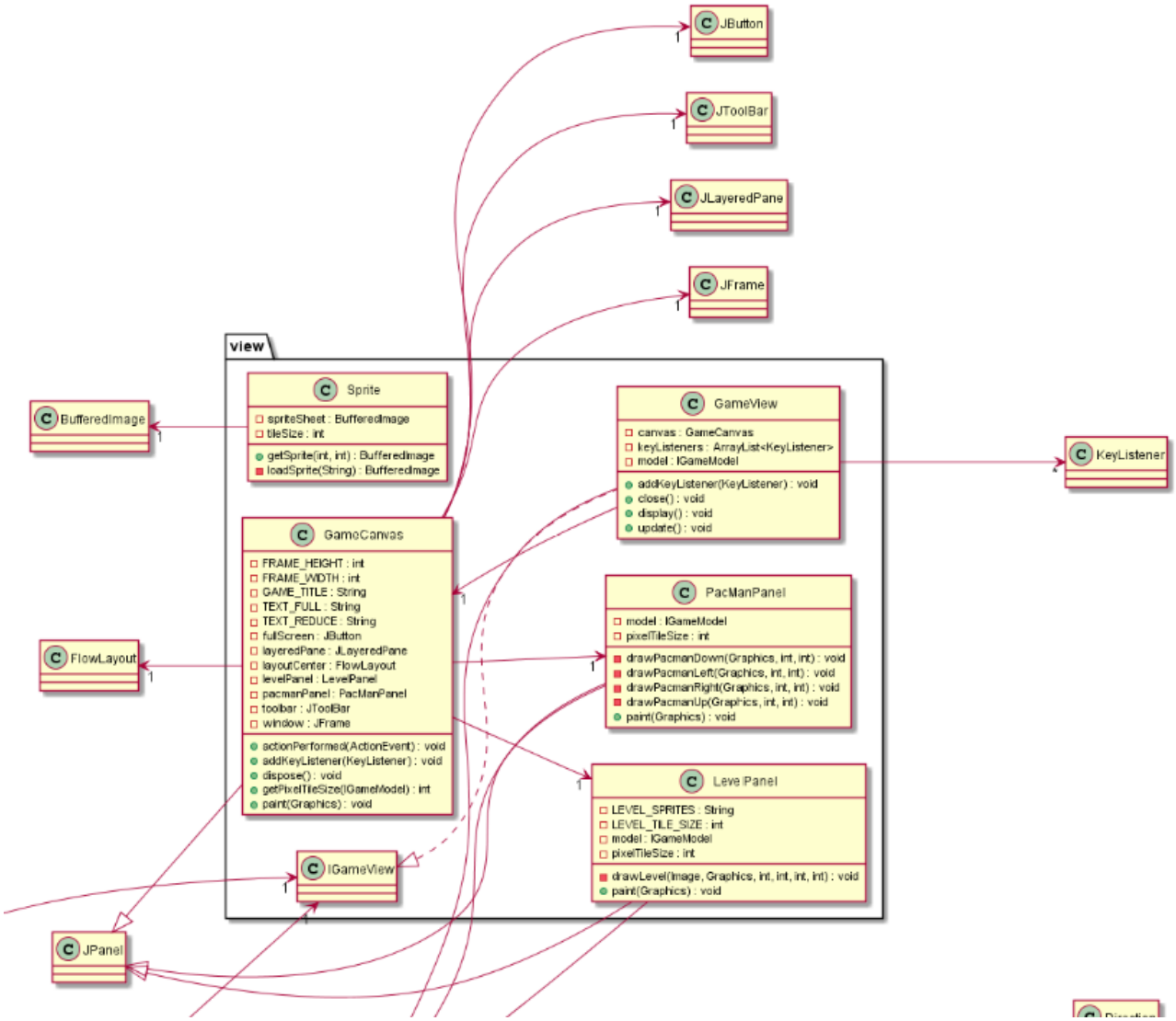

Figure 6: Diagramme UML de la vue

2.4 Diagrammes UML du contrôleur seulement

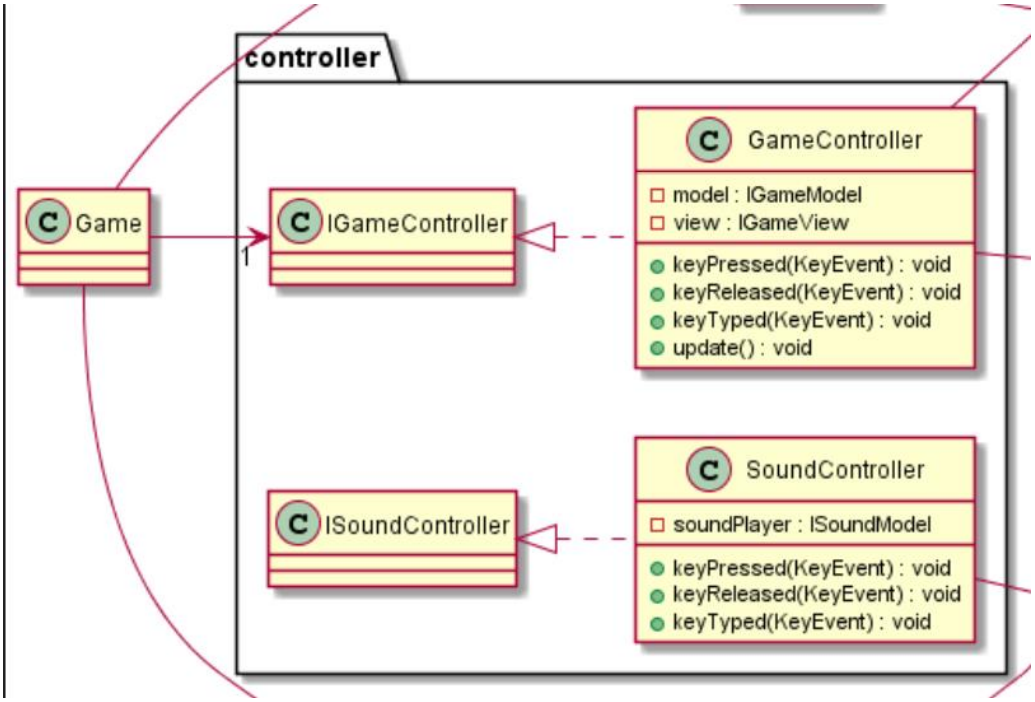

Figure 7: Diagramme UML du contrôleur

2.5 Architecture haut-niveau MVC

Le patron de modèle-vue-contrôleur est utilisé. Cela permettra de garder l'application modulaire et de permettre plus facilement les changements futurs. Par exemple, on pourrait avoir différentes vues comme d'afficher le jeu pour commencer en console avec des caractères ASCII, puis éventuellement avoir une interface graphique plus belle. Pareillement, on pourrait utiliser différents contrôleurs comme le clavier, la souris ou bien une manette de jeu vidéo. Les interactions sont ainsi :

- L'utilisateur utilise le contrôleur (par exemple en appuyant sur la flèche du haut)
- Le contrôleur manipule le modèle (par exemple, il lui indique l'intention de bouger le Pac-Man vers le haut)
- Le modèle se met à jour en respectant les règles du jeu (par exemple, le Pac-Man frappe un fantôme et meurt)
- La vue se met à jour et s'affiche à l'utilisateur en utilisant les données du modèle (par exemple en affichant l'animation de mort)

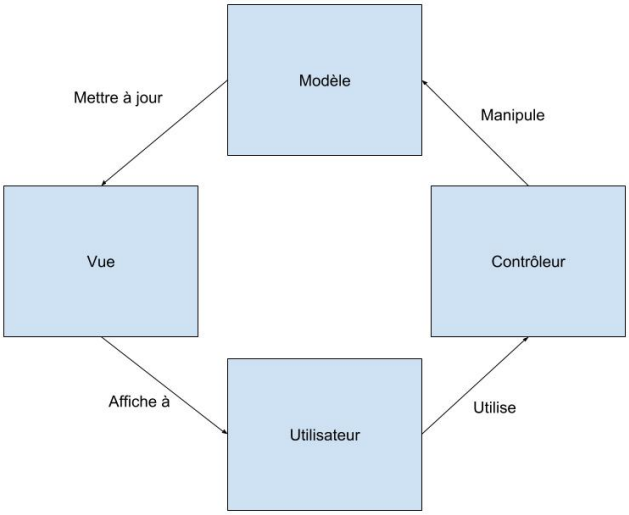

Figure 8: Architecture MVN

2.6 Boucle de jeu en temps réel

Lors de la dernière session, le programme de PaintUS mettait à jour son modèle uniquement lors des actions de l'utilisateur. Au contraire, un jeu vidéo en temps réel comme Pacman doit continuellement se mettre à jour. En effet, les fantômes continuent de pourchasser le joueur même si celui-ci va se faire un café.

L'application ne sera donc pas seulement dirigée par les actions de l'utilisateur sur le contrôleur. Il est nécessaire d'introduire une boucle de jeu qui gère le temps et met à jour périodiquement l'état du modèle et de la vue. Ces deux mises à jour seront indépendantes, de sorte qu'il sera possible d'avoir une fréquence d'affichage différente de celle du jeu. On aura donc plus de contrôle sur l'utilisation des ressources et on pourrait augmenter la fréquence d'affichage et réduire la vitesse du jeu indépendamment si cela s'avère utile éventuellement.

2.7 Description de comment les tests ont été réalisés

D'abord, nous avons utilisé la librairie JUnit pour tester notre logiciel. L'arborescence des fichiers de tests est identique à l'arborescence des fichiers du code source. Ainsi, chaque classe est testée dans son propre fichier. Les méthodes des classes du modèle ont été testées unitairement, à l'exception des accesseurs et des mutateurs. Certains tests d'intégration permettent également de valider l'interaction entre plusieurs méthodes. Bref, la figure 9 montre que les classes du

modèle sont testées à 60%. L'équipe devra améliorer ce pourcentage au cours des prochains sprints.

model	57,6 %	415	305	720
Level.java	37,6 %	35	58	93
Position.java	23,6 %	17	55	72
MovementManager.java	40,7 %	35	51	86
GameModel.java	77,8 %	140	40	180
MoveValidator.java	70,0 %	63	27	90
SoundModel.java	0,0 %	0	23	23
Ghost.java	12,5 %	3	21	24
PacMan.java	73,0 %	27	10	37
Levels.java	60,0 %	12	8	20
SuperPacGum.java	30,0 %	3	7	10
Direction.java	92,9 %	65	5	70
MoveRequest.java	100,0 %	15	0	15

Figure 9: Couverture des tests unitaires et d'intégration

Ensuite, notons que certains membres de l'équipe ont utilisé l'approche de développement piloté par les tests, c'est-à-dire de d'abord coder un test unitaire qui échoue. Ensuite, le code implémenté s'assure que le test passe. L'approche permet de mieux préciser le comportement de l'application et elle offre une cohérence entre les tests et le code. Aussi, la méthode assure que les tests unitaires sont véritablement écrits.

Finalement, à défaut d'utiliser Jenkins comme outil d'intégration continue, l'équipe pense utiliser Travis, qui s'intègre bien avec GitHub. Ainsi, à chaque fois qu'un membre de l'équipe ouvre une « Pull Request » sur GitHub, les tests doivent passer afin que cette dernière puisse être intégrée. Cela permet d'avoir une version stable en développement et, surtout, en production.

2.8 Utilisation des principes SOLID

Le logiciel développé utilise les principes SOLID afin que le code développé soit lisible, extensible, testable et maintenable. D'abord, le principe de responsabilité unique est respecté puisque chaque classe à une seule responsabilité. Par exemple, la classe PacManView est uniquement responsable de dessiner le Pac-Man, alors que classe LevelView est responsable de dessiner le labyrinthe.

Ensuite, le principe d'ouverture/fermeture est également utilisé. Les changements au code source demandent peu de changement, puisque celui-ci est « ouvert à l'extension, mais fermé à la modification ». L'abstraction est utilisée afin de respecter le principe. Ainsi, lorsqu'une classe est intégrée au code source, elle peut seulement être étendue par la suite.

Le principe de l’inversion des dépendances est utilisé. Par exemple, la classe MoveValidator implémente la classe IMoveValidator, ce qui permet aux classes de haut-niveau et de bas-niveau de dépendre de l’abstraction IMoveValidator seulement. La validation des mouvements peut donc être modifiée, sans affecter la classe de haut-niveau.

Finalement, la ségrégation des interfaces permet de séparer les interfaces en plusieurs petites interfaces spécifiques afin de ne pas polluer les interfaces. Les classes qui implémentent une interface doivent en effet implémenter toutes les méthodes de cette dernière.