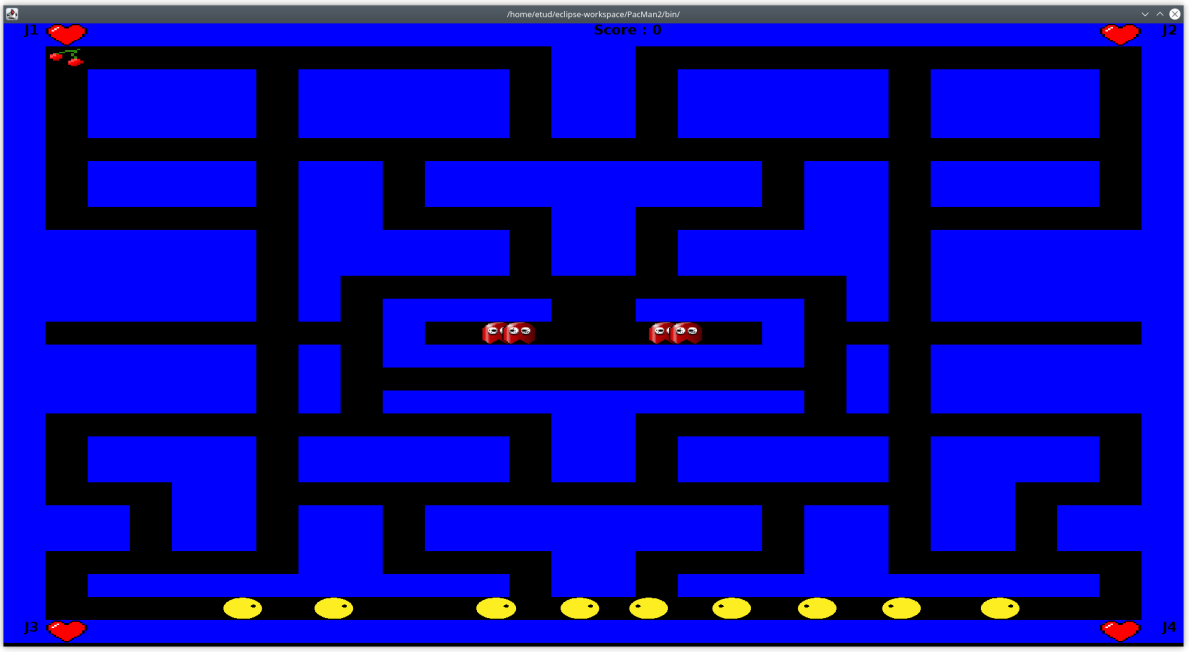
Rapport de projet Pac-Man - Design patterns

Objectif : *Réaliser un jeu Pac-Man en Java en utilisant des designs patterns appropriés.*



I - Liste des améliorations / ajouts

A) Le menu

Un menu a été ajouté, avec des paramètres comme l'activation de l'anti-aliasing, les touches ou le nombre de FPS… Également le choix de la carte est plus intuitif et le rôle des joueurs et des agents peut-être choisi directement sans le coder en dur. Dans le jeu avec la touche de Pause (par défaut "Echap"), permet de mettre en pause et de revenir au menu.

B) La refonte de l'interface graphique

Les graphismes ont été améliorés avec des positions flottantes pour permettre plus de fluidité, un imageManager a été introduit pour permettre de ne pas surcharger la mémoire, et de pouvoir libérer quand le nombre d'images chargées est trop grand.

C) Une Musique

Une musique de fond et quelques bruitages ont été ajoutés aux jeux.

D) Les vies et score des Pac-Man

Des constantes permettent de donner un score total aux PacMan et des cœurs sont disponibles pour les différents PacMan.

E) Comportements

Nous avons un comportement Aléatoire, un mode multijoueur jusqu'à 4 sur la même machine, un comportement A* au clic, sur les fruits ou les PacMan et la fuite des fantômes.

II - Designs Patterns utilisés

A) Patterns Patron de méthode

Les différents games héritent de Moteur.Game comme Moteur.SimpleGame ou Moteur.PacManGame, aussi le Menu fonctionne comme cela en possédant un Panel qui changent en fonction des actions.

B) MVC - Modèle View Controller

Ce design pattern est utilisé dans les différents Game, en fait on a un interface Observer :

```
void update Game game );
```

Et une classe ObserverManager qui gèrent les différents Observer:

```
private ArrayList<Observer> observers = new ArrayList<>();
	private Game game;
	public void addObserver(Observer observer) {
		observers.add(observer);
	}
	public void removeObserver(Observer observer) {
		observers.remove(observer);
	}
	public void notifyObservers() {
		for (Observer observer : observers) {
			observer.update(game);
		}
	}
```

Les jeux possèdent un ObserverManager. Par exemple dans SimpleGame on a des views comme ViewSimpleGame :

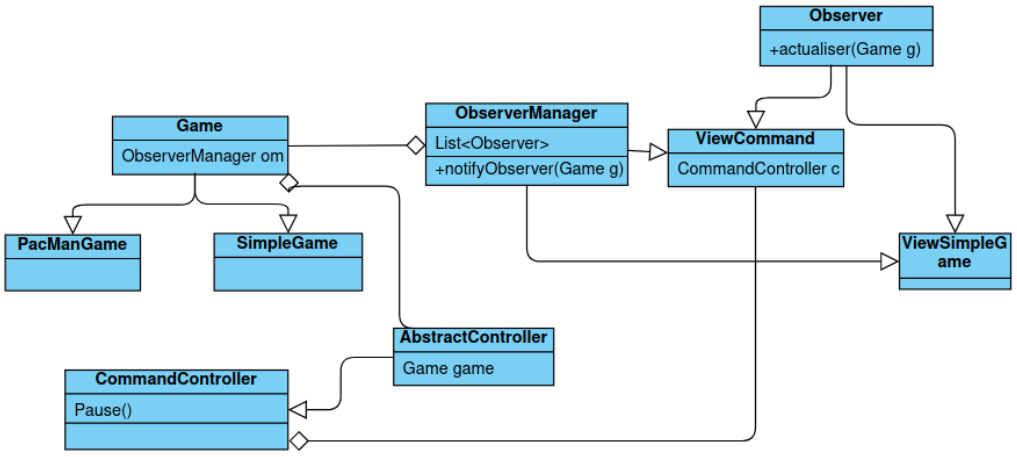
```
public class ViewSimpleGame implements Observer{

	private JFrame frame;

	public ViewSimpleGame() {
		//…
}
	public void update(Game game) {
		//…
	}
}
```

ou ViewCommand qui permettent de contrôler le jeu. Ce design pattern permet de mettre à jour la vue. Si la vue veut changer le jeu, elle utilise son attribut un controller qui permet de faire la liaison entre l'observable et le Jeu.

Voici l'UML simplifié:



C) Pattern Etat

Utilisé dans le ViewCommand pour gérer l'activation et la désactivation des différents boutons (par exemple lors de la pause, on désactive le bouton pause et on active le bouton play).

D) Patterns Stratégie

Ce design patterns permet de changer dynamiquement des attributs, ici il est utilisé pour changer le comportement des Agents du jeu(Pac-Man et Fantôme). Par exemple, on peut changer pour faire jouer un humain et une IA. En cours de jeu les fantômes essaient d'attraper les Pac-Man mais si un mangent un fruit, alors les Pac-Man essaient de fuir, on a donc un changement dynamique grâce au patterns stratégie.

E) Singleton

Il permet d'avoir une **seule instance** de classe de la façon suivante:

```
public class Clavier implements KeyListener {
	private static Clavier instance;

	public static Clavier getInstance() {
		if (instance == null)
			instance = new Clavier();
		return instance;
	}
	private Clavier() {
		//…
	}
}
```

Il est utilisé dans Listener.Clavier, Le controllerPacGame, PacMacGame, Musique, Mouse, Clavier, moveToPacMan, FleeComportement, MenuManager, PanelPacMan.

F) Fabrique

La fabrique est utilisée dans le GamePacMan, il permet de créer les agents plus simplement et si d'autres agents ou comportements se rajoutent dans le futur ils seront **simples** à implémenter.