



TRON TRON

Modèles de Conception Réutilisables

Amel Dussier, Ioannis Noukakis & Fabien Salathe

Table des matières

Introduction.....	3
Contexte	3
Objectifs	3
Analyse	4
Fonctionnalités	4
Architecture globale	5
Choix des technologies	5
Conception	6
Protocole de communication	6
Fonctionnement	6
Messages	7
Modèles.....	8
Serveur	9
Fonctionnement	9
Diagramme de classes	10
Client.....	11
Fonctionnement	11
Diagramme de classes	12
Le patron Mediator	13
Présentation	13
Fonctionnement	14
Implémentation dans notre application.....	15
Captures d'écran	17
Gestion du projet.....	18
Rôles des participants	18
Outils communs.....	19
IntelliJ	19
Git	19
Stratégie de tests.....	20
Planification.....	21
Planning prévisionnel	21
Planning effectif.....	21
Conclusion	22
Etat du projet à l'échéance	22

Projet de MCR

Problèmes rencontrés	22
Améliorations possibles	22
Annexes	23
Table des illustrations.....	23
Journal de travail	24
Semaine 1	24
Semaine 2	24
Semaine 3	24
Semaine 4	24
Semaine 5	24
Semaine 6	25
Semaine 7	25
Semaine 8	25
Semaine 9	26
Semaine 10	26

Introduction

Contexte

Ce projet se fait dans le cadre de notre cours « Modèles de conception réutilisables ». Le but de ce cours est de découvrir et de nous familiariser avec des *Design Patterns*. Ce sont des solutions connues répondant à des problèmes récurrents de conception logicielle, et qui permettent de concevoir des applications robustes et modulaires.

Après avoir étudié les différents patrons de conception, de manière théorique et dans des exercices, nous allons maintenant les appliquer dans un projet plus conséquent. Il s'agira d'un travail de groupe, et chaque groupe au sein de la classe aura comme objectif d'utiliser un *design pattern* principal, choisi à l'avance. Evidemment, la taille du projet permettra d'en utiliser d'autres, afin de mettre en œuvre les bonnes pratiques étudiées lors du cours.

En plus du projet de développement et du rapport, chaque groupe fera une présentation orale afin de partager son projet et son expérience avec les autres groupes de la classe.

Objectifs

Notre groupe a choisi de travailler sur le patron de conception « Mediator ». Il s'agit d'un *design pattern* qui permet de simplifier la communication entre objets. Le médiateur est un intermédiaire entre les objets, et ceux-ci n'ont pas besoin de connaître la nature ou le nombre d'objets avec lesquels ils communiquent. Ce patron sera décrit de façon détaillée dans le chapitre « Conception ».

L'exemple classique pour ce patron est une application de messagerie ou un forum de discussion. En effet, un utilisateur n'a pas besoin de connaître chaque personne connectée pour envoyer un message à tout le monde. C'est le serveur (le médiateur) qui s'en charge pour lui, en traitant le message envoyé.

Dans le cadre de notre projet, nous avons décidé de réaliser un jeu de course multijoueur. Il s'agira d'une version simplifiée et revisitée des courses de motos *LightCycle* du film *Tron*. Nous allons mettre en application le modèle de conception « Mediator » pour communiquer par exemple les actions d'un joueur aux autres joueurs de la partie. Cela nous permettra de rendre la communication entre les joueurs plus modulaire et plus simple à gérer.

Analyse

Fonctionnalités

Nous allons commencer par un petit rappel concernant les courses de *LightCycles*. Dans le film *Tron* original, les personnages doivent participer à une course de motos virtuelles, les *LightCycles*. Ces motos évoluent sur un plan en deux dimensions, et ne peuvent faire que des virages à 90 degrés (angle droit). Durant la partie ils ne disposent pas de freins pour ralentir ou s'arrêter, ils ne peuvent qu'avancer et changer de direction. Chaque moto laisse dans son sillage un mur de lumière (« Jetwall » en anglais). Si un concurrent entre en collision avec un mur de lumière, il a perdu la partie. Le dernier joueur dans la partie gagne. Notre jeu va reprendre en grande partie ces principes, et en ajouter quelques nouveaux.

Tout d'abord, les nouveaux arrivants dans la partie vont démarrer dans une zone d'entrée neutre (« lobby »). Dans cette zone, les joueurs sont invincibles et peuvent se familiariser avec les commandes. Lorsqu'ils rencontrent le bord de la carte, ils seront automatiquement repositionnés au centre.

Dans la zone d'entrée se trouve également un portail de téléportation, qui donne accès à la carte principale où a lieu la partie. Sur cette carte, plus grande et d'une couleur différente, les joueurs meurent s'ils rencontrent un bord ou s'ils touchent le mur de lumière laissé par la moto d'un autre joueur. Ils peuvent ensuite recommencer depuis la zone d'entrée.

Nous avons également ajouté des « bonus » sur la carte, permettant par exemple de modifier la vitesse ou la taille des motos, et ainsi donner un avantage temporaire à un joueur. Pour accéder à ces bonus il suffit de rouler dessus et leur effet est immédiat.

L'aspect graphique est important pour que les joueurs prennent plaisir à utiliser notre jeu. Nous allons faire au mieux pour rendre notre application visuellement attrayante, mais nous allons mettre l'effort principal sur l'implémentation du patron de conception « Mediator » et la gestion de la communication entre les joueurs. Il s'agit après tout de la raison d'être de ce projet.

Architecture globale

Un schéma qui montre simplement le server et un ou deux clients

Choix des technologies

Java

Librairie Slick

Justifier pourquoi, ou dire si c'était imposé

Conception

Protocole de communication

Fonctionnement

Le protocole de communication découle directement de l'architecture de notre application, nous allons donc décrire son fonctionnement en premier.

Le protocole permet de faire le lien et gérer la communication entre la partie cliente et la partie serveur

Utilisation des sockets et de la sérialisation

Messages

Nous avons au total quatre types de messages qui sont échangés entre le client et le serveur, qu'on va décrire ci-dessous.

Les messages envoyés du serveur aux clients :

- **PlayerIdentity**
Il s'agit du message qui est envoyé au client suite à sa demande de connexion. Ce message contient simplement un identifiant unique, que le client utilisera pour ses demandes de mise à jour ultérieures.
- **UpdateWorld**
Ce message contient le contenu de la carte dans laquelle le joueur est en train d'évoluer, avec la nouvelle position des objets et des autres joueurs.

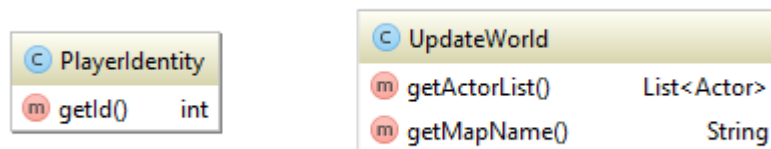


Figure 1 : Messages envoyés du serveur au client

Les messages envoyés du client au serveur :

- **JoinGame**
Message de demande de connexion d'un joueur, contenant le nom choisi par le joueur et qui sera communiqué aux autres joueurs dans la partie
- **ChangeDirection**
Il s'agit du message envoyé au serveur par le client quand le joueur décide de changer de direction. Ce changement est ensuite transmis aux autres joueurs dans un message « UpdateWorld ».

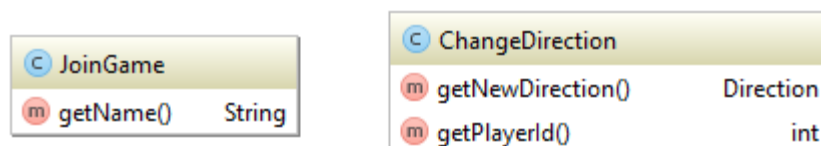


Figure 2 : Messages envoyées du client au serveur

Modèles

Ci-dessous le diagramme de classes des modèles, ainsi que des messages utilisés par le protocole de communication :

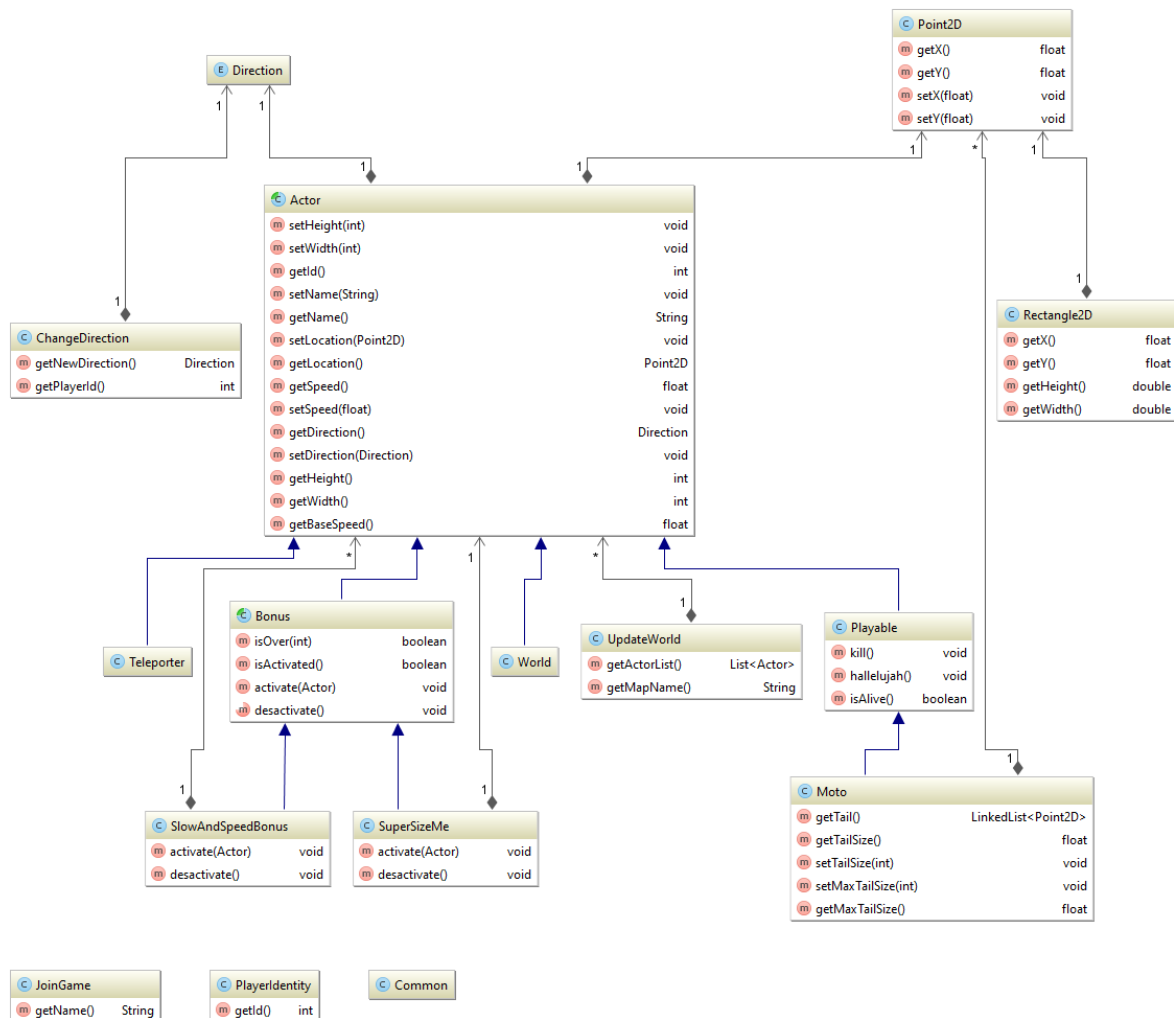


Figure 3 : Diagramme de classes de la partie protocole et modèles

On peut voir que la majorité des modèles héritent de la classe « Actor », qui est une classe abstraite implémentant le comportement de base de chaque acteur du jeu. Les acteurs principaux sont les motos, les bonus et les téléporteurs.

Serveur

Fonctionnement

Explications

Projet de MCR

Diagramme de classes

Ci-dessous le diagramme de classes de la partie serveur :

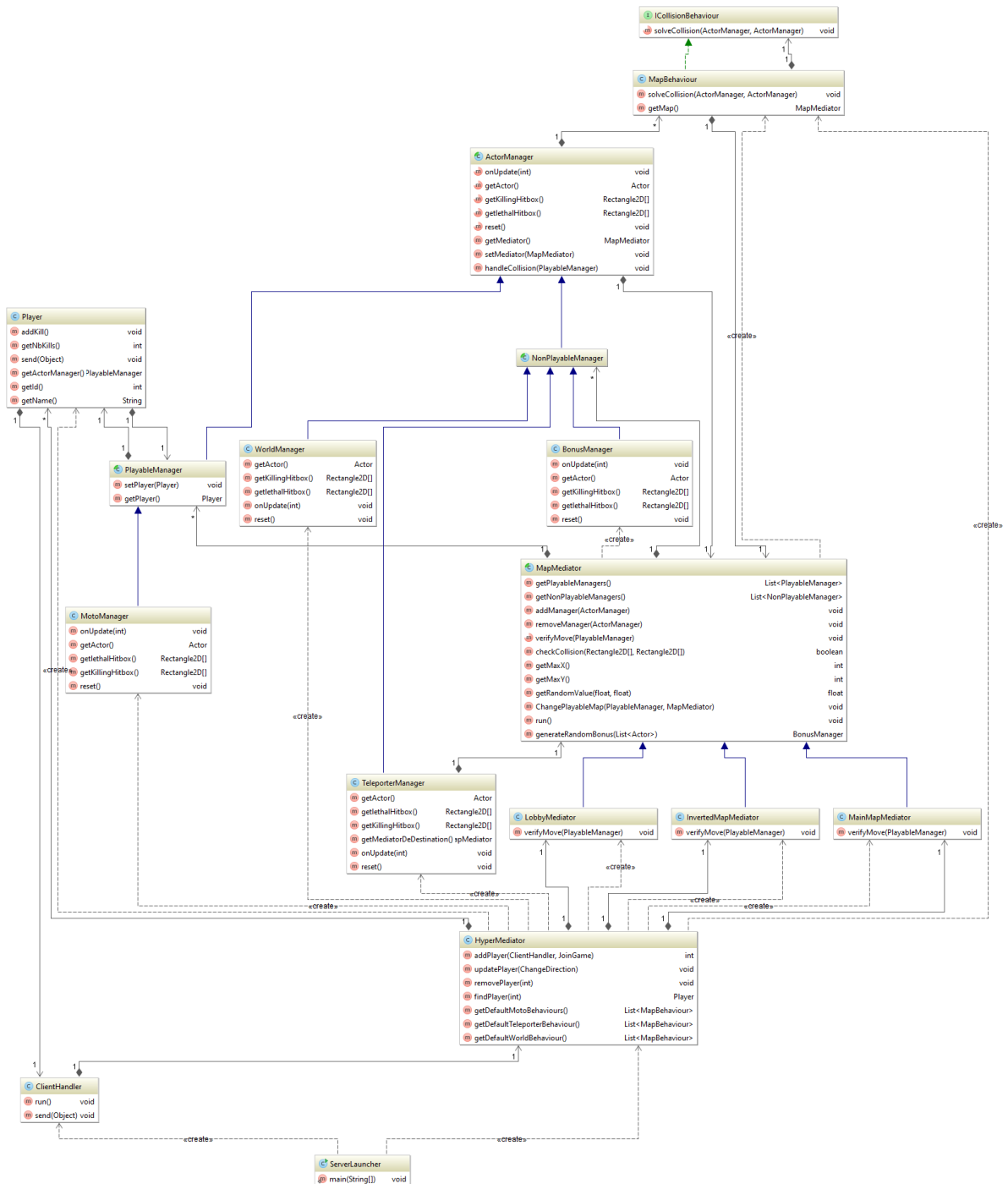


Figure 4 : Diagramme de classes de la partie serveur

Client

Fonctionnement

Explications

Diagramme de classes

Ci-dessous le diagramme de classes de la partie cliente :

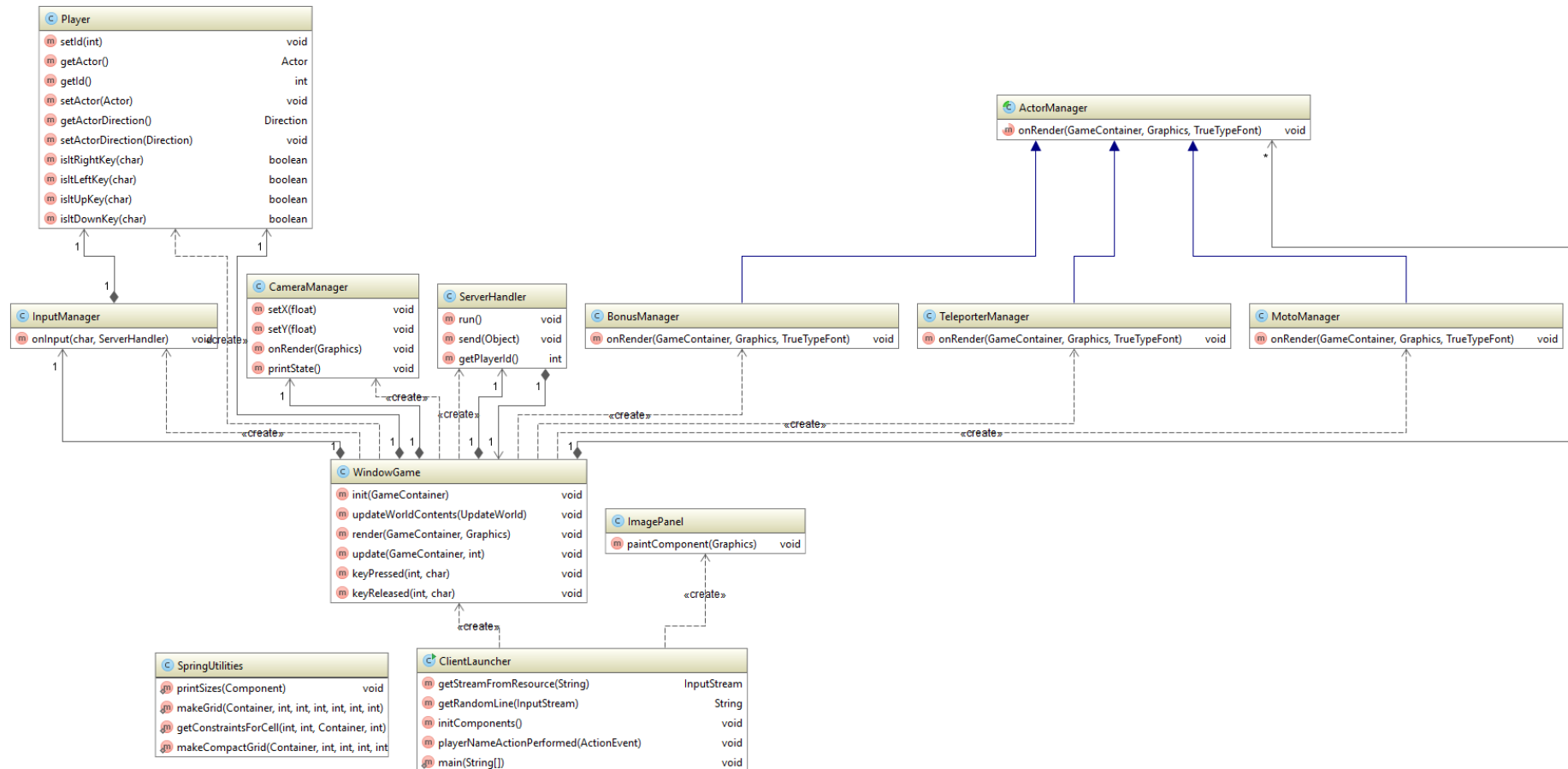


Figure 5 : Diagramme de classes de la partie cliente

Le patron Mediator

Présentation

Selon le GoF (Gang of four), le Mediator est un *design pattern* de type comportemental qui s'applique au domaine objet. Son objectif est de définir un objet, le médiateur, qui encapsule la manière dont communiquent les autres objets.

Une application contient généralement plusieurs classes, qui se partagent la logique et les traitements prévus par le programme. Avec l'augmentation de la taille de l'application et du nombre de classes, la communication entre ces différentes classes devient de plus en plus complexe. Cela crée des dépendances entre les classes, et ce couplage fort rend le code difficile à maintenir ou à faire évoluer.

Le patron de conception Mediator propose de découpler les classes, en déléguant la partie communication à un objet particulier : le médiateur. Ainsi, les classes n'ont plus besoin de savoir avec qui, ou avec combien d'objets, elles communiquent. Le médiateur fait en sorte de traiter les messages et de les faire parvenir aux objets concernés, ce qui rend la communication plus simple.

Ci-dessous une illustration présente dans notre cours de MCR, qui illustre bien la différence entre un système sans et avec un médiateur :

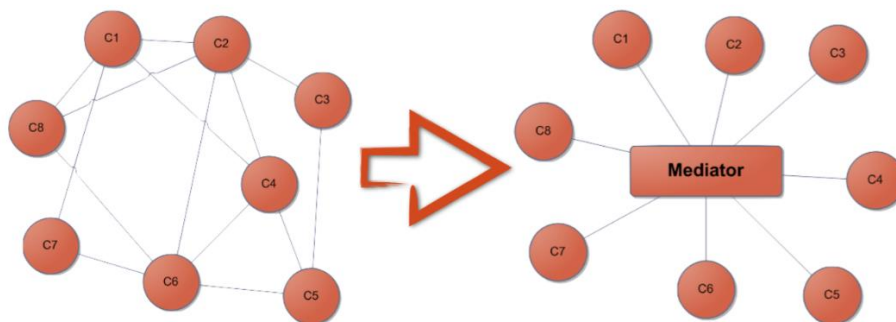


Figure 6 : Fonctionnement simplifié du Mediator

On constate que sans médiateur, les liens entre les classes peuvent être nombreux et complexes. Un changement dans l'application peut avoir des impacts conséquents. En revanche avec le médiateur, la logique de communication est limitée à une classe centralisée, facile à maintenir ou à étendre.

Fonctionnement

Maintenant que nous avons vu l'intérêt d'utiliser le patron de conception Mediator, nous allons nous intéresser à son fonctionnement d'un point de vue plus technique.

Ci-dessous le diagramme de classes du *design pattern* mediator, tel qu'il est généralement représenté :

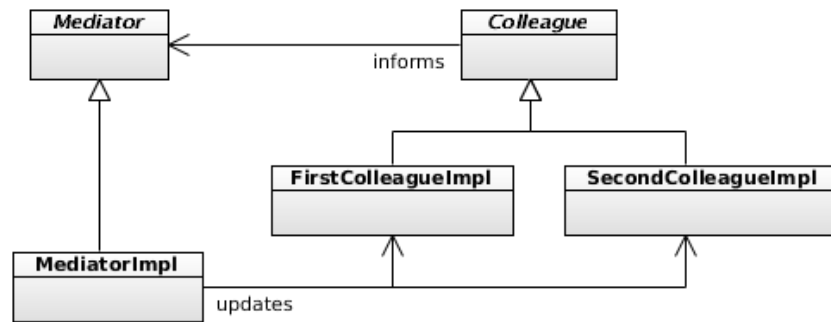


Figure 7 : Diagramme de classes du Mediator

Les participants sont :

- **Colleague** : classe abstraite ou interface qui représente une classe qui communique
- **FirstColleagueImpl & SecondColleagueImpl** : implémentations concrètes de classes communicantes
- **Mediator** : interface ou d'une classe abstraite qui représente le médiateur. Cette interface expose les méthodes nécessaires aux clients pour communiquer
- **MediatorImpl** : implémentation concrète d'un Mediator. Cette classe communiquera avec les collègues concrets

Les classes communicantes sont appelées « collègues », ce qui souligne bien l'esprit de travail en équipe et de partage des responsabilités que ce patron de conception met en avant. Chaque collègue possède une référence vers un médiateur, et peut ainsi le contacter chaque fois qu'il doit communiquer (flèche « informs » du diagramme de classes). Le médiateur concret va ensuite faire suivre l'information ou l'action aux collègues concernés, selon une logique qui reste interne au médiateur (flèche « updates » du diagramme de classes). Un collègue est donc complètement indépendant des autres, et n'a pas conscience de leur existence.

L'interface du médiateur doit évidemment exposer les méthodes utiles aux collègues. Si le nombre de ces méthodes devient trop important, il est possible de créer plusieurs médiateurs, chacun responsable de la communication d'un groupe ou type de collègues spécifique. On peut même créer une arborescence, avec des médiateurs gérant la communication entre médiateurs.

Implémentation dans notre application

Dans le cadre de notre projet, nous avons utilisé plusieurs médiateurs, avec des responsabilités différentes.

Ci-dessous les relations entre les différents médiateurs de notre application :

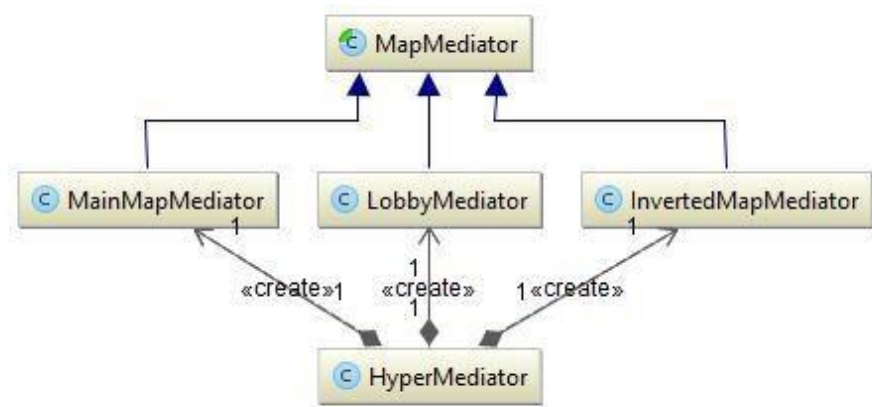


Figure 8 : Mediators utilisés dans le cadre de l'application

Un médiateur principal, qui s'appelle « HyperMediator », et qui a plusieurs responsabilités principales :

- Gestion de la connexion et déconnexion des joueurs
- Traitement des messages émis par les différents joueurs de la partie lorsqu'ils font une action (changement de direction par exemple)
- Il crée et gère la communication entre les médiateurs secondaires
- Traitement des changements de carte par les joueurs (et donc de médiateur secondaire)

On peut voir ci-dessous les détails de la classe « HyperMediator » avec ses méthodes publiques, qui permettent d'inscrire, désinscrire et traiter une mise à jour d'un joueur :



Figure 9 : Méthodes publiques de la classe HyperMediator

Des médiateurs secondaires, un par carte de jeu. Ils héritent tous d'un médiateur abstrait « MapMediator ». Cette classe abstraite définit les comportements identiques à toutes les cartes. Les responsabilités principales de ces médiateurs secondaires sont :

- Gestion de la carte elle-même
- Gestion des joueurs présents sur la carte
- Gestion des objets (portails de téléportation, bonus, etc.)
- Déterminer si une collision a eu lieu entre deux acteurs

Nous avons choisi de mettre en place cette hiérarchie de médiateur afin de ne pas surcharger le médiateur principal, mais également d'avoir une plus grande flexibilité pour ajouter ou enlever une carte au jeu.

Comme chaque carte est gérée par son propre médiateur, il est également plus facile de modifier les règles du jeu d'une carte à une autre. Par exemple, dans la carte d'entrée gérée par la classe « LobbyMediator », les collisions entre les joueurs ne sont pas prises en compte, contrairement à la carte de jeu principale (gérée par la classe « MainMapMediator »).

Captures d'écran

2-3 captures d'écran sympa avec une description

Gestion du projet

Rôles des participants

Analyste

Développement

Tests

Documentation

Outils communs

IntelliJ

Pour coder, décrire vite fait l'éditeur

Tout le monde utilise le même éditeur pour simplifier les choses

Git

Gestion des sources

Décrire GIT en quelques mots

Décrire notre stratégie, nouvelles branches, commits réguliers, etc.

Stratégie de tests

Comment on a testé nos fonctionnalités les unes après les autres

Décrire quelques tests fonctionnels (changement de carte, collision, etc.)

Planification

Planning prévisionnel

Diagramme de Gantt

On en invente un, le plus simple possible ;)

Planning effectif

Basé ou similaire au journal de travail

Conclusion

Etat du projet à l'échéance

Où on en est

Ce qui n'a pas pu être réalisé

Problèmes rencontrés

Ce qui n'a pas marché dans l'équipe

Ce qu'on aurait pu faire mieux

Améliorations possibles

Fonctionnalités cool à ajouter

Protocole de communication :

- envoyer seulement un delta des changements
- négociation dynamique du port UDP

Annexes

Table des illustrations

Figure 1 : Messages envoyés du serveur au client	7
Figure 2 : Messages envoyées du client au serveur	7
Figure 3 : Diagramme de classes de la partie protocole et modèles	8
Figure 4 : Diagramme de classes de la partie serveur	10
Figure 5 : Diagramme de classes de la partie cliente	12
Figure 6 : Fonctionnement simplifié du Mediator	13
Figure 7 : Diagramme de classes du Mediator	14
Figure 8 : Mediators utilisés dans le cadre de l'application	15
Figure 9 : Méthodes publiques de la classe HyperMediator	15

Journal de travail

Semaine 1

Dates : 11 avril au 15 avril

Introduction au projet et création des groupes. Notre groupe sera composé de : Amel Dussier, Ioannis Noukakis, Fabien Salathe et Raphael Mas Martin

Semaine 2

Dates : 18 avril au 22 avril

Etude des différents patrons de conception. On décide de choisir le patron Mediator pour notre projet de groupe.

Semaine 3

Dates : 25 avril au 29 avril

On décide de fait un « chat » client-serveur, qui est un exemple classique de l'utilisation du Mediator.

Semaine 4

Dates : 2 mai au 6 mai

Définition des fonctionnalités de notre application de chat.

Semaine 5

Dates : 9 mai au 13 mai

Elaboration de la présentation Powerpoint en vue de la présentation intermédiaire du 17 mai.

Semaine 6

Dates : 16 mai au 20 mai

Présentation intermédiaire de notre projet. Le projet est refusé par le professeur, qui nous demande de concevoir une autre application qu'un chat. Nous décidons dans l'urgence de créer un jeu d'arcade.

Semaine 7

Dates : 23 mai au 27 mai

Définition des fonctionnalités, intégration du patron Mediator au niveau de la communication entre les joueurs et entre les cartes.

Première ébauche du jeu, réalisée principalement par Ioannis pour éviter de perdre du temps à se coordonner sur le développement.

Semaine 8

Dates : 30 mai au 3 juin

Amel

- Développement
 - Amélioration du protocole de communication, séparation des threads de communication du reste du code
 - Réorganisation des modèles
 - Nettoyage du code : uniformisation des noms des packages, suppression des fichiers inutiles sur GIT, etc.
- Rapport final
 - Première structure
 - Rédaction de l'introduction

Ioannis

- Développement
 - À compléter, je pense que t'en a fait des trucs entre le serveur et le client 😊

Fabien

- Développement
 - À compléter 😊
- Présentation finale
 - À compléter 😊

Toujours aucune implication dans le projet de la part de Raphael. Discussion avec le professeur, car entre le changement de projet après la présentation intermédiaire et l'absence de participation de Raphael, le projet a pris beaucoup de retard.

Semaine 9

Dates : 6 juin au 10 juin

Amel

- Développement
 - Diverses améliorations sur la partie serveur, suppression de méthodes inutilisées
 - Commentaires sur le code du protocole de communication et des modèles
 - Commentaires sur une partie du code serveur
- Rapport final
 - Rédaction complète ou partielle de différents chapitres
 - Finalisation du rapport

Ioannis

- Développement
 - À compléter ☺
 - Commentaires sur le code client
 - Commentaires sur une partie du code serveur

Fabien

- Développement
 - À compléter ☺
- Présentation finale
 - À compléter ☺
 - Finalisation de la présentation

Raphael

- Rapport final
 - Rédaction d'une partie du chapitre sur le patron Mediator

Semaine 10

Dates : 13 juin au 17 juin

Rendu du projet et présentation finale.