ESEO

Dossier conception MonkeyIsland

COOL-QL

Auteurs
Cailyn Davies
Guillaume Muret
Relecteurs
Clément Pabst
François de Broch d'Hotelans

Sommaire

Introduction

Dans ce rapport, nous traiterons les différents choix d'architecture que nous avons implémentés dans le code et nous expliquerons les différents patterns utilisés pour chaque architecture.

Ainsi, nous commencerons par présenter l'architecture générale de notre projet, puis celle des managers associés pour ensuite terminer par l'architecture de notre communication.

1 – Architecture générale du projet :

Le schéma de l'architecture générale est disponible ci-après (Figure 1).
L'architecture de notre projet se sépare en différentes couches :

- Une couche model où les objets n'ont aucune intelligence et sont de simples objets avec des attributs. Ces objets sont par exemple les « Pirate », les « Monkey », les « Treasure », les « Rum » ou les « Square ».
- Une couche métier représenté par un « Brain » et ses managers comme « MapManager », « CharacterManager » ou « ObjectManager ». Tous ces managers gèrent les objets qui leurs sont rattachés et représentent donc l'intelligence de ces objets. On a ainsi :
    - « MapManager » qui contrôle la « Map »
    - « CharacterManager » qui contrôle les « Pirate » et les « Monkey »
    - « ObjectManager » qui contrôle les « Rum » et le « Treasure ».

La « Communication » s'occupe de gérer les sockets du serveur (connexion et communication), l'encodage des différents messages en accord avec le protocole de communication avec leur envoie et leur lecture.

Au démarrage du serveur, le « Brain » instancie directement les managers, la « Communication » et attend le premier événement du jeu envoyé par la « Communication » qui est « notifyNewConnection() ».
Le « Brain » du système s'occupe de gérer les différents événements du jeu et de notifier tous les managers lors d'un de ces événements. Cette classe est l'unique classe qui s'occupe de demander à la « Communication » d'envoyer les messages aux joueurs par les méthodes « sendXXX() » du « Brain ».
Tous les managers héritent de la classe « AbstractManager ». Cette dernière implémente l'interface « IGameEvent ». Lors des événements de jeux, par exemple lors d'une nouvelle connexion d'un joueur au serveur, la « Communication » annonce au « Brain » qu'une nouvelle connexion vient de s'effectuer. Ainsi, ce « Brain » va notifier les managers par la méthode « notifyNewConnection() ». Cette méthode appelle la méthode « onNewConnection() » de l'interface « IGameEvent » pour spécifier aux managers qu'une nouvelle connexion est apparue (principe de l'observateur).

« Pourquoi avoir choisi ce type d'architecture ? »

Nous avons préféré choisir ce type d'architecture car nous voulions implémenter l'intelligence de nos objets à l'extérieur de ceux-ci. En effet, nos managers sont donc des « dictateurs » pour les objets qu'ils doivent gérer.

Les avantages liés à ce type d'architecture sont qu'elle permet de bien séparer chaque activité du projet et de ne pas mélanger les intelligences sur des objets qui pourraient être compliqués.
A l'inverse, ce type d'architecture ne permet donc pas à nos objets d'avoir une intelligence. Ainsi, nous nous sommes imposé de ne pas utiliser le pattern stratégie qui aurait pu nous être très utile notamment pour les déplacements des singes et des pirates.

« Pourquoi votre « CharacterManager » est-il beaucoup plus fourni par rapport aux autres managers ? »

En effet notre « CharacterManager » se trouve plus fourni car toutes les exigences primaires du jeu se trouvent dans le déplacement des « Pirate » et des « Monkey ». Dans ce cas, le « CharacterManager » gérant les personnages de notre serveur, c'est donc cette instance qui est la plus fournie. Si par exemple, une évolution serait d'avoir des « Square » dynamiques, l'instance du « MapManager » serait beaucoup plus fournie.

Dans l'ensemble de ce projet, nous nous sommes imposé d'utiliser des noms de méthodes, de variables, d'attributs, de classes et de package les plus explicites possibles pour permettre une meilleure compréhension lors de la lecture du code.
Dans l'intégralité de ce projet, nous nous sommes efforcés de ne pas utiliser les singletons comme variable globale mais comme un moyen de ne pouvoir créer qu'une seule instance. Nous avons donc limiter les appels au « SingletonXXX.getInstance() » uniquement dans les constructeurs pour permettre une meilleure lisibilité du code et éviter tout appel intempestif à des variables globales.

Patterns utilisés

Dans cette architecture générale, on peut constater les différents singletons utilisés pour les managers, le « Brain » et la « Map ».
Le pattern observateur est implémenté par le « Brain » qui notifie les managers par les méthodes « notifyXXX() ». Ces méthodes « notifyXXX() » permettent ensuite d'appeler les méthodes « onXXX() » correspondantes de l'interface « IGameEvent » implémentée par tous les managers.

Voici le schéma de l'architecture générale de notre projet. Pour une meilleure lisibilité, toutes les classes, tous les attributs et toutes les méthodes des classes n'ont pas été représentés.

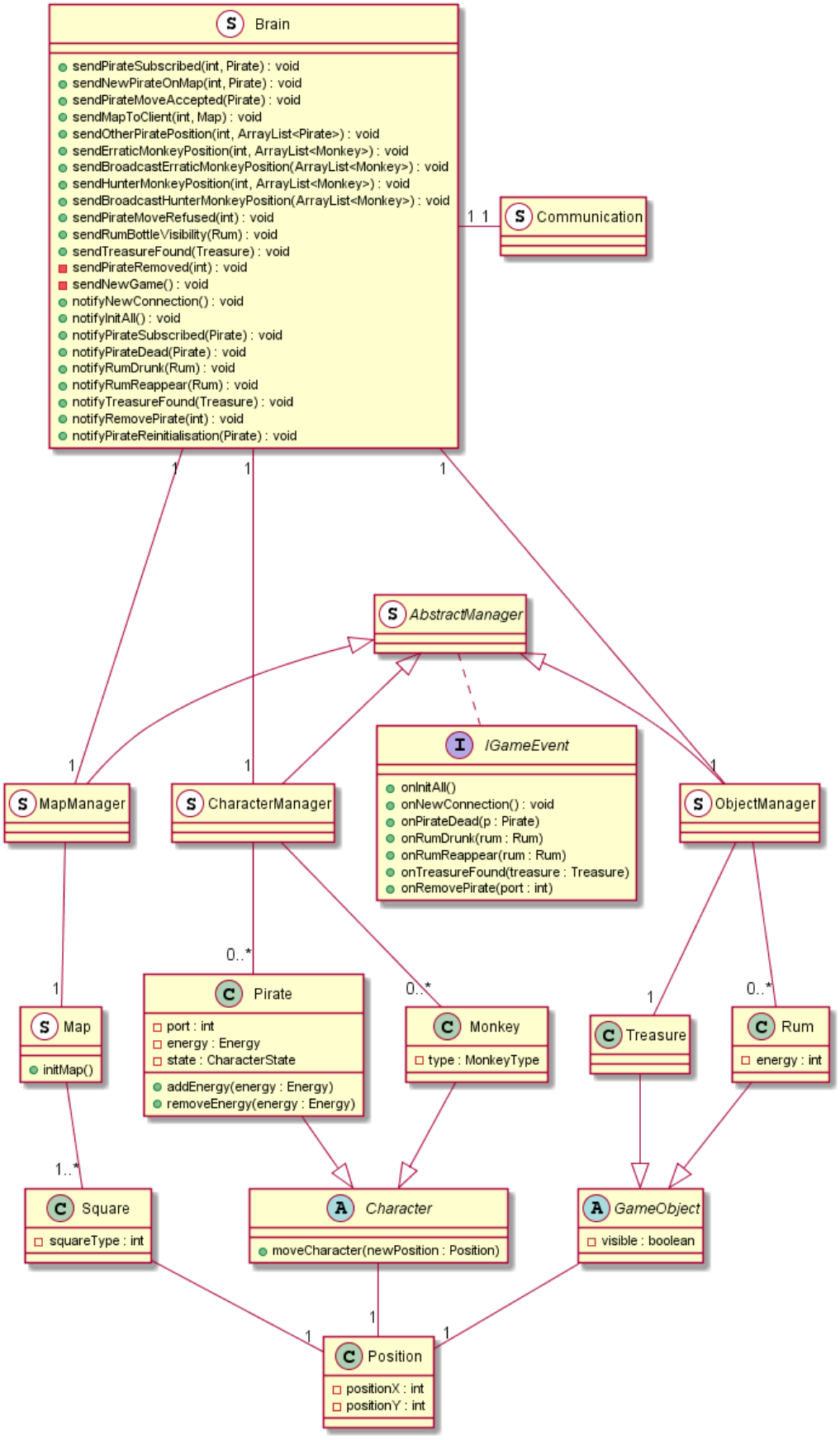


Figure 1: architecture générale du projet

2 - Architecture du « CharacterManager »

Le « CharacterManager » a pour objectif général de gérer les « Character » présents sur le plateau, soit les « Pirate » et nos deux types de « Monkey » actuellement. Une première partie concernera l'architecture mise en place autour des « Character », la deuxième partie traitera rapidement des threads de déplacement des « Monkey ». La dernière partie concernera les déplacements pour l'ensemble des « Characters », puis enfin les événements qui peuvent en survenir.

2.1 – Architecture autour de « Character »

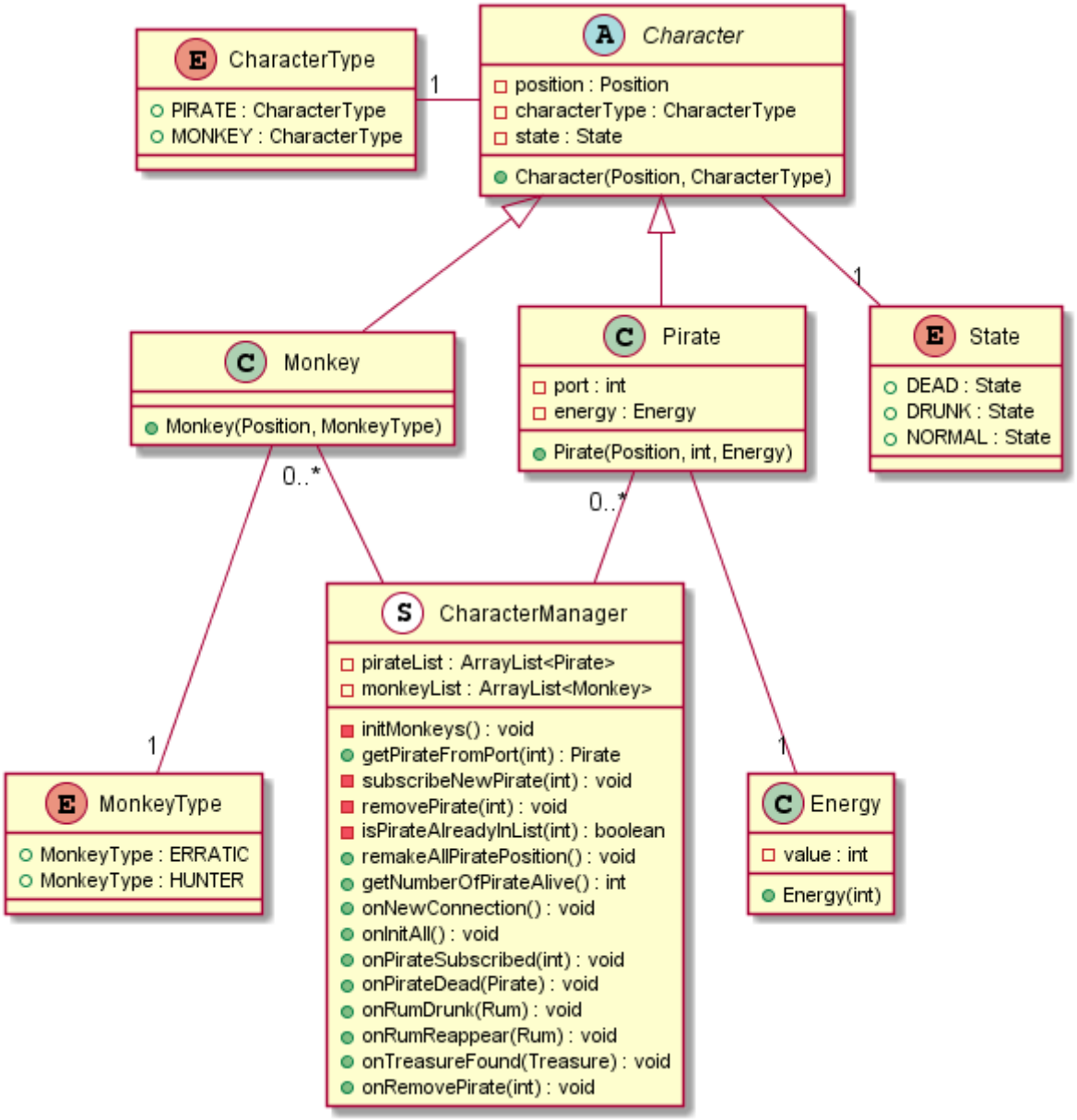


Figure 2: architecture autour de "Character"

Dans ce schéma on retrouve un certain nombre d'énumérations qui permettent de mettre en place une évolutivité de notre code pour la suite. L'énumération « State » concerne l'état du pirate s'il est saoul, mort ou normal. L'évolutivité pourrait être une attribution de différents états à nos « Monkey », ce qui permettrait d'influencer leur déplacement en fonction de leur état.

Sont également présents un certain nombre de méthodes « onXXXX » dans « CharacterManager » qui correspondent aux notifications du « Brain » liées au pattern observateur qui est implémenté (Cf : Partie 1).

2.2 – Architecture des threads pour le déplacement des singes

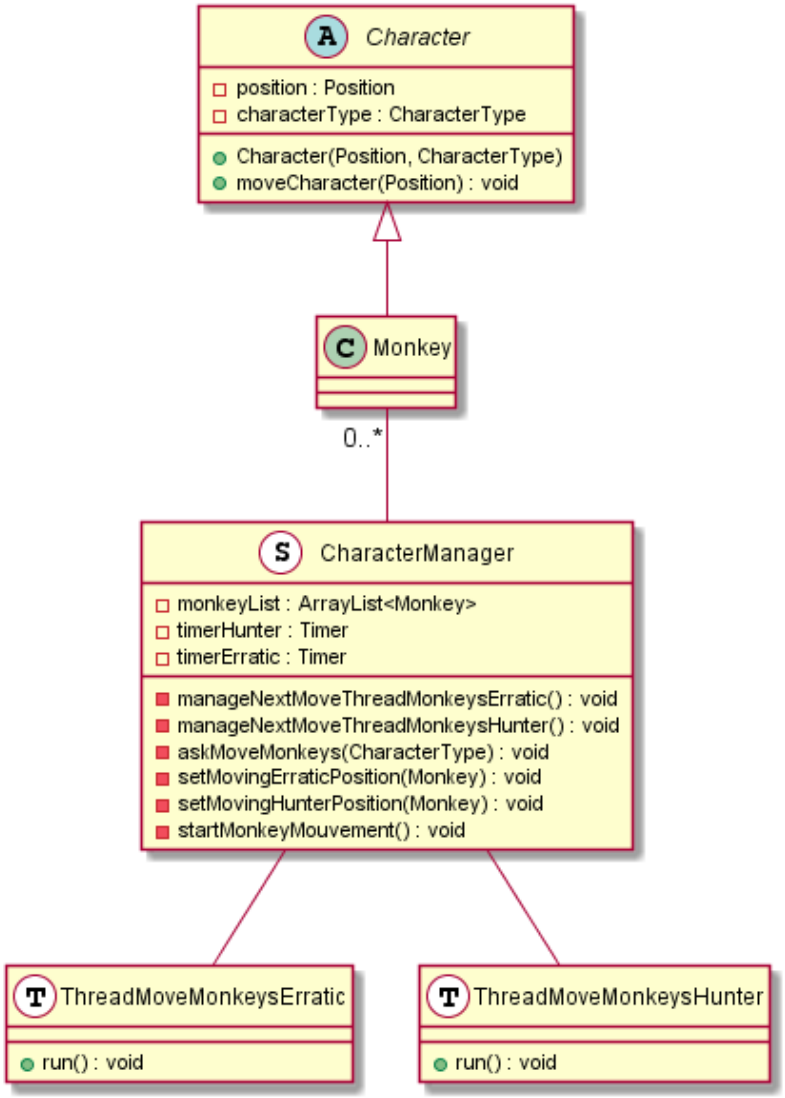


Figure 3: architecture des threads pour le déplacement des singes

Les deux threads ici présents permettent de gérer en parallèle de l'exécution du serveur les déplacements des singes. Les « Timer » en attribut du « CharacterManager » permettent de donner une période d'exécution voulue.

2.3 – Architecture des déplacements

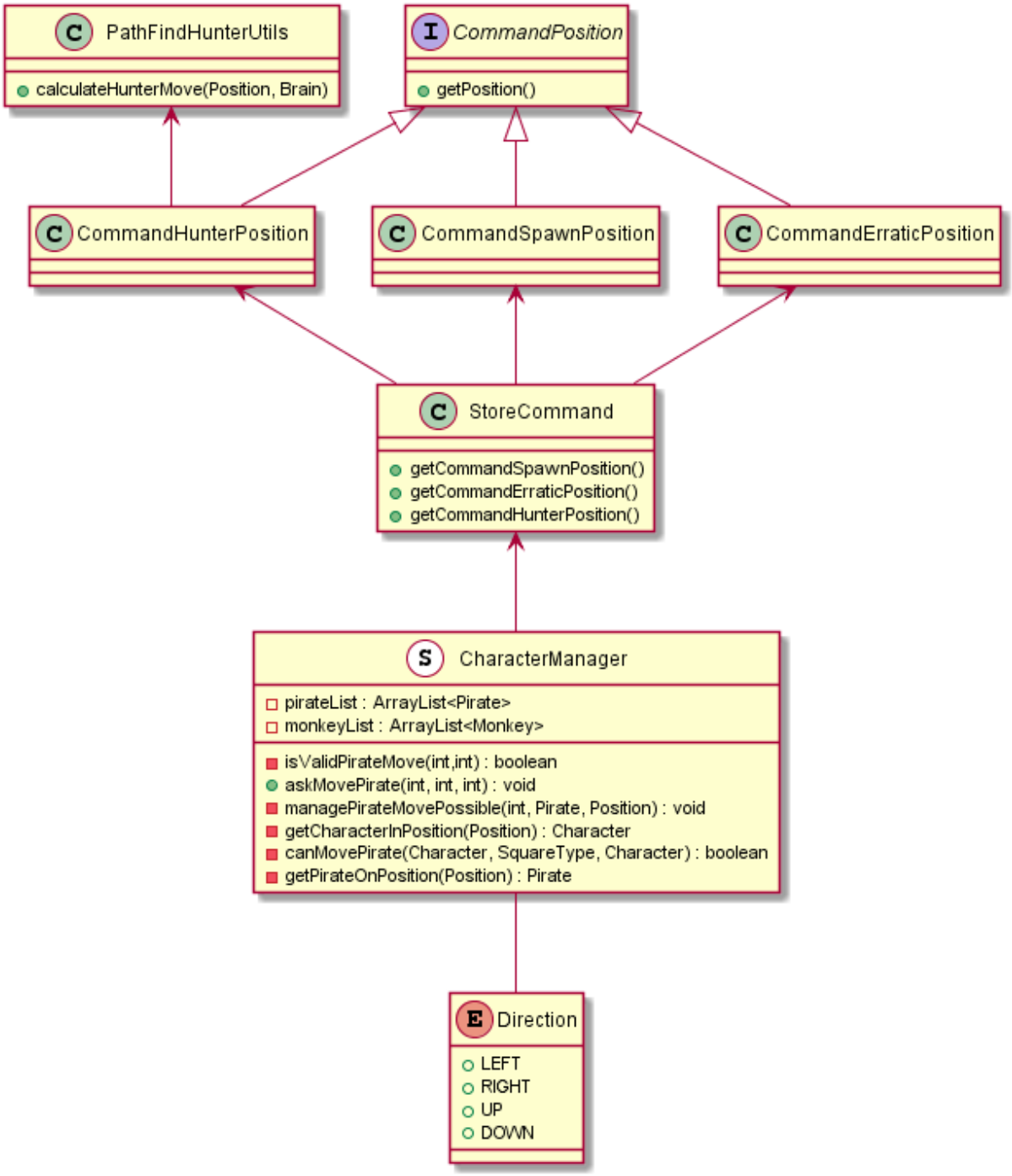


Figure 4: architecture des déplacements

Pour le déplacement des différents « Character » nous utilisons un pattern commande à l'aide de l'interface « ICommandPosition », elle est utilisée de trois manières différentes :

- « CommandSpawnPosition » : utilisée pour donner une position initiale aux Pirates

- « CommandErraticPosition » : utilisée pour faire le déplacement d'un singe erratique et pour un pirate qui est ivre.
- « CommandHunterPosition » : utilisée pour faire le déplacement du singe chasseur en faisant appel à l'algorithme de pathfind.

L'ensemble de ces commandes sont appelées par le « CharacterManager » via le « StoreCommand ». Elles exécutent chacune le déplacement correspondant.

Avec du recul, il aurait été certainement préférable d'un point de vue évolutivité d'avoir un pattern stratégie et non commande. L'attribution d'une stratégie aurait été intéressante avec un simple appel de méthode par exemple « nextMove » qui se chargera d'effectuer le déplacement des « Monkey » en fonction de la stratégie attribuée à ceux-ci. La stratégie pourra ensuite être mise à jour en fonction des différents événements du jeu. Notre erreur a été de partir directement dans l'idée qu'aucune intelligence ne devait être intégrée au modèle, ce qui a rendu le « CharacterManager » très complexe également. Dans cette configuration, les managers sont « dictateurs » sur l'intelligence du modèle.

2.4 – Architecture des événements

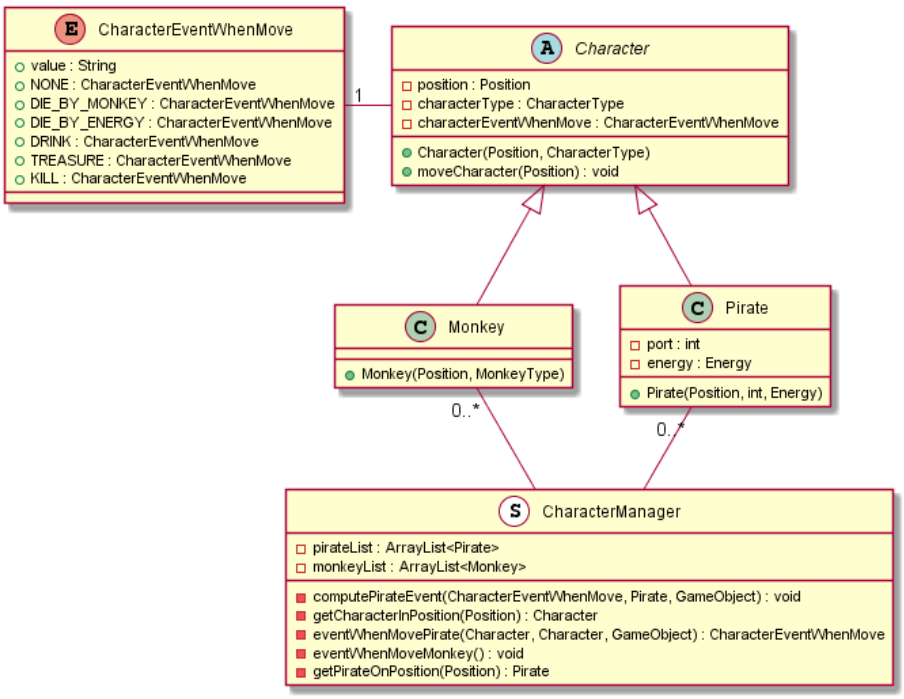


Figure 5: architecture des événements

Une fois les déplacements effectués, des événements doivent avoir lieu notamment si le « Character » en question tombe sur un autre « Character » ou un « GameObject ». « CharacterManager » saura en regardant l'attribut « characterEventWhenMove » du personnage quel événement appliquer, et quels « Character » et « GameObject » ont besoin d'être manipulés. La notification des événements est réalisée par le « Brain » (cf : partie 1).

3 – Architecture de l' « ObjectManager

L'« ObjectManager » s'occupe de gérer tous les objets de notre système. L' « ObjectManager » est ici très simple car les types d'objets gérés par ce manager sont les « Rum » et le « Treasure ». Si de nouveaux objets apparaissaient par exemple un « BateauPirate » qui pourrait attaquer les « Pirate » sur l'île, l' « ObjectManager » serait bien plus fourni que cela.

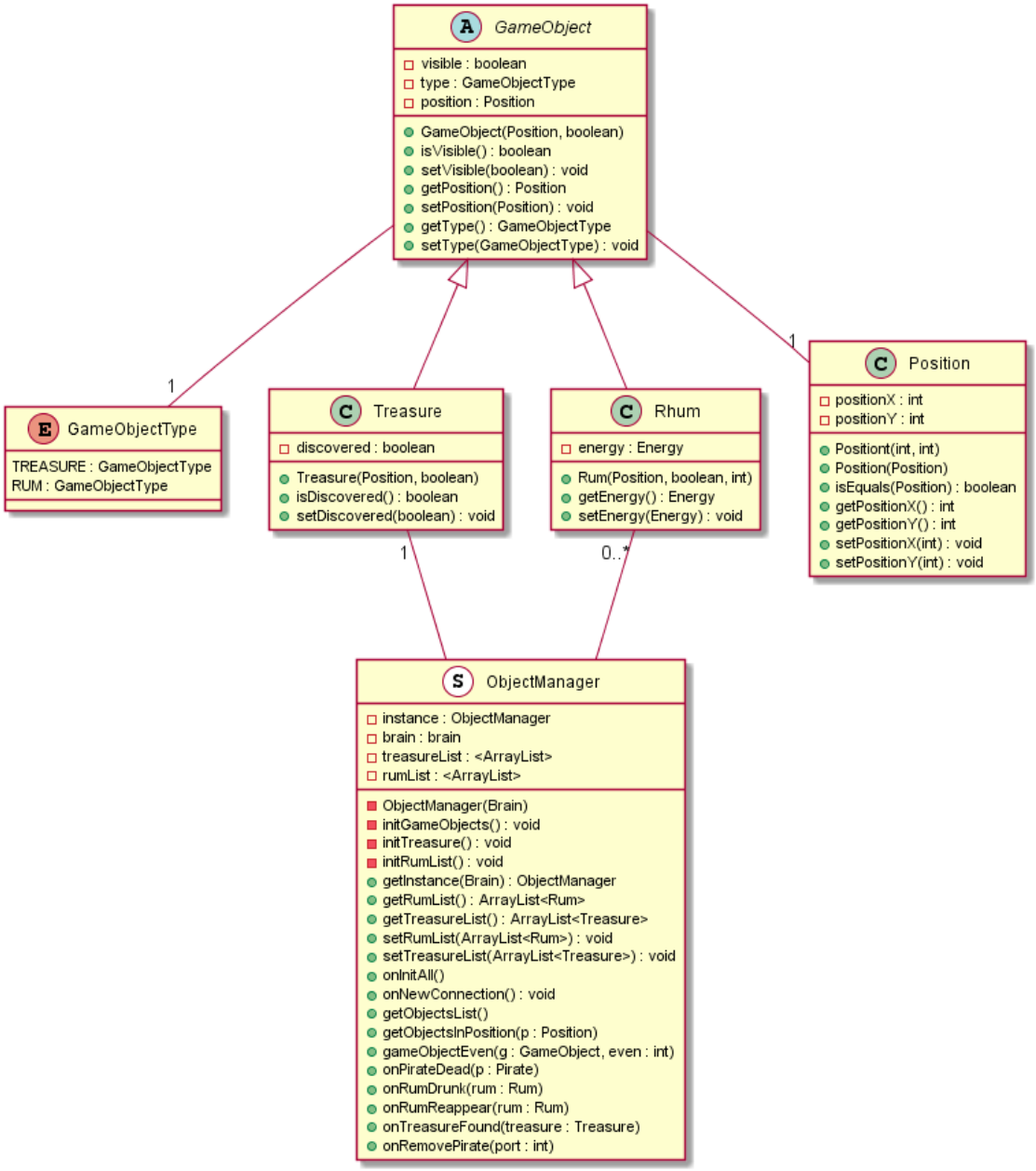


Figure 6: architecture de "ObjectManager"

Patterns utilisés

Les patterns utilisés dans cette architecture sont l'observateur (énoncé en partie 1) et le singleton pour l' « ObjectManager ».

4 – Architecture du « MapManager »

Le « MapManager » s'occupe de gérer la « Map » et les « Square ». On constate aussi pour ce manager que son architecture est simple. Si nous avions voulu par exemple avoir des « Square » dynamiques, avec par exemple des raz de marrés qui se forment aléatoirement ou bien la formation de lacs, la classe « MapManager » serait bien plus fourni.

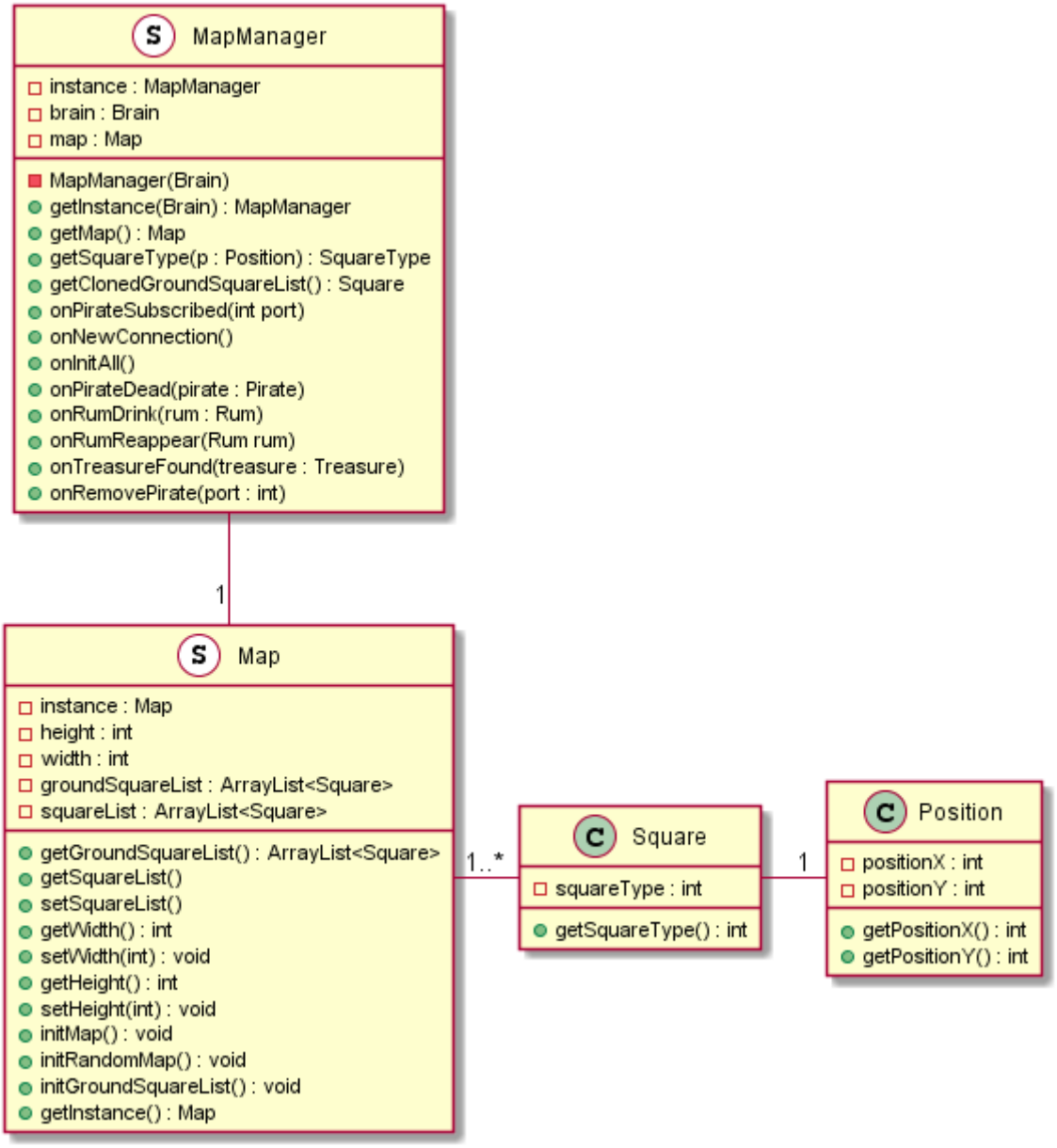


Figure 7: architecture du "MapManager"

Patterns utilisés

Les patterns utilisés dans cette architecture sont l'observateur (énoncé en partie 1) et les singletons pour le « MapManager » et la « Map ».

5 - Architecture de la « Communication »

La « Communication » de notre serveur respecte différents patterns appris en I2 lors du ProSE. Ainsi, cette « Communication » utilise plusieurs patterns très puissants qui seront expliqués par la suite. Cette architecture se divise en quatre parties. La première partie concernera l'explication de l'instance « Communication ». La seconde concernera l'explication du « PostmanServer » qui s'occupe de gérer les différentes sockets, la troisième concernera le « ProxyClient » qui s'occupe de gérer les différents « AbstractEncoder » qui eux s'occupent de gérer l'encodage des messages en accord avec le protocole de communication. La quatrième et dernière partie sera pour l'explication du « Distributor » qui s'occupe de gérer la conversion des différentes commandes envoyées par le client.

5.1 – Architecture de l'instance « Communication »

Voici ci-après le schéma de l'instance « Communication » du jeu :

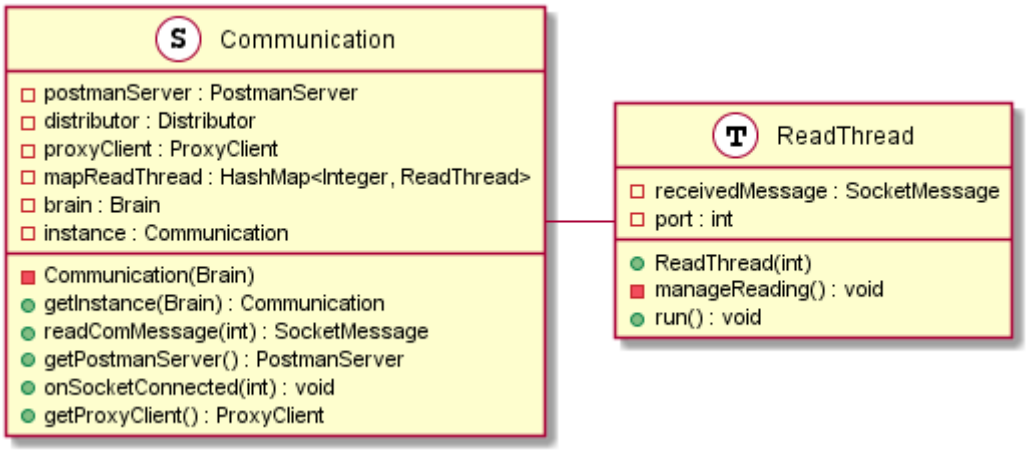


Figure 8: schéma de l'instance "Communication"

On constate sur cette Figure 4 que l'instance « Communication » est un singleton. Cette classe s'occupe de créer les instances des différents composants de notre serveur c'est-à-dire :

- Le « Distributor » qui s'occupe de gérer la conversion des différentes commandes envoyées par le client
- Le « ProxyClient » qui s'occupe de gérer l'encodage des messages
- Le « PostmanServer » qui s'occupe de gérer les sockets

L'ordre de création des objets est très important.

5.2 – Architecture du « PostmanServer »

Voici ci-après l'architecture du « PostmanServer ». Pour une meilleure visibilité de ce schéma, les composants du « Distributor », du « ProxyClient » et du « Brain » n'ont pas été représentés :

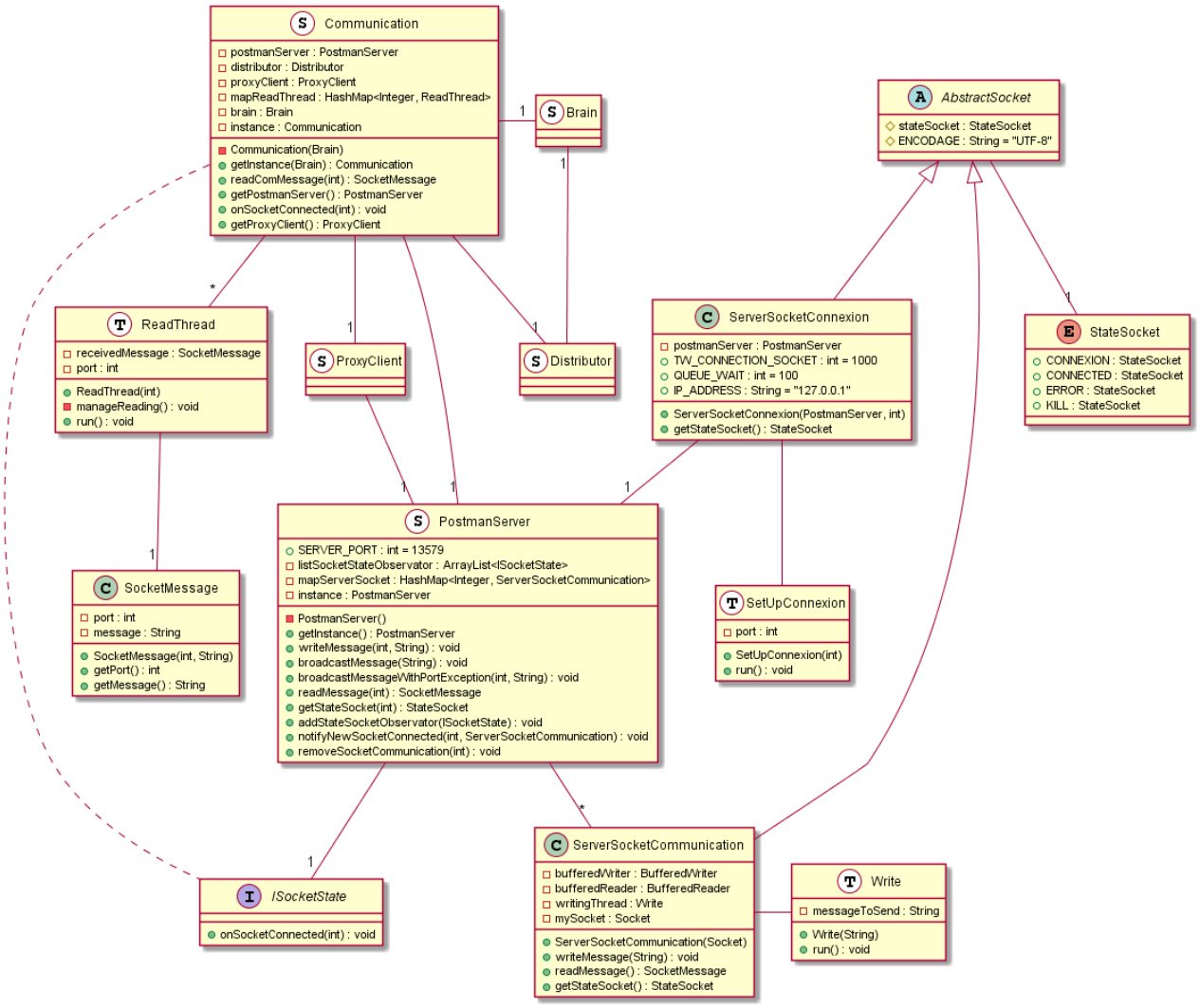


Figure 9: architecture du "PostmanServer"

Le « PostmanServer » s'occupe de gérer les différents sockets de notre projet. En effet, lors de la création de cette instance, celui-ci va demander la création du « ServerSocketConnexion » correspondant à notre socket de connexion lancé dans un thread appelé « SetUpConnexion ». A chaque nouvelle connexion, le « ServerSocketConnexion » créer un « ServerSocketCommunication » qu'il donne au « PostmanServer ». Le « PostmanServer » l'ajoute à sa liste de « ServerSocketCommunication » appelé « mapServerSocket ». Ensuite, le « PostmanServer » va notifier par l'intermédiaire de l'interface « ISocketState » qu'une nouvelle connexion vient de s'effectuer par la méthode « onSocketConnected() ». Cette méthode n'est implémentée que par la « Communication » et permet de notifier

le « Brain » qu'une nouvelle connexion vient de s'établir et ensuite de créer et d'ajouter dans sa liste (appelé « mapReadThread ») le thread de lecture en scrutation (appelé « ReadThread ») sur le nouveau socket de communication.

Lorsqu'un message sera lu dans un « ReadThread » de la communication, le « Distributor » se chargera de convertir ce message et de l'interpréter pour ensuite l'envoyer au « Brain » afin de gérer cet événement.

Lorsqu'un message devra être écrit, le « ProxyClient » s'occupera de l'encodage des messages puis de demander au « PostmanServer » d'envoyer le message correspondant dans le thread d'écriture appelé « Write ». Cette section sera expliquée plus en détail dans la partie 5.3.

Patterns utilisés

Dans cette architecture, le pattern observateur a été implémenté pour notifier les objets implémentant l'interface « ISocketState » qu'une nouvelle connexion vient de s'effectuer sur le socket de connexion.
Le pattern singleton a aussi été implémenté pour ne pouvoir autoriser la création que d'une seule instance dans le code.

5.3 – Architecture du « ProxyClient »

L'architecture du « ProxyClient » se présente comme suit :

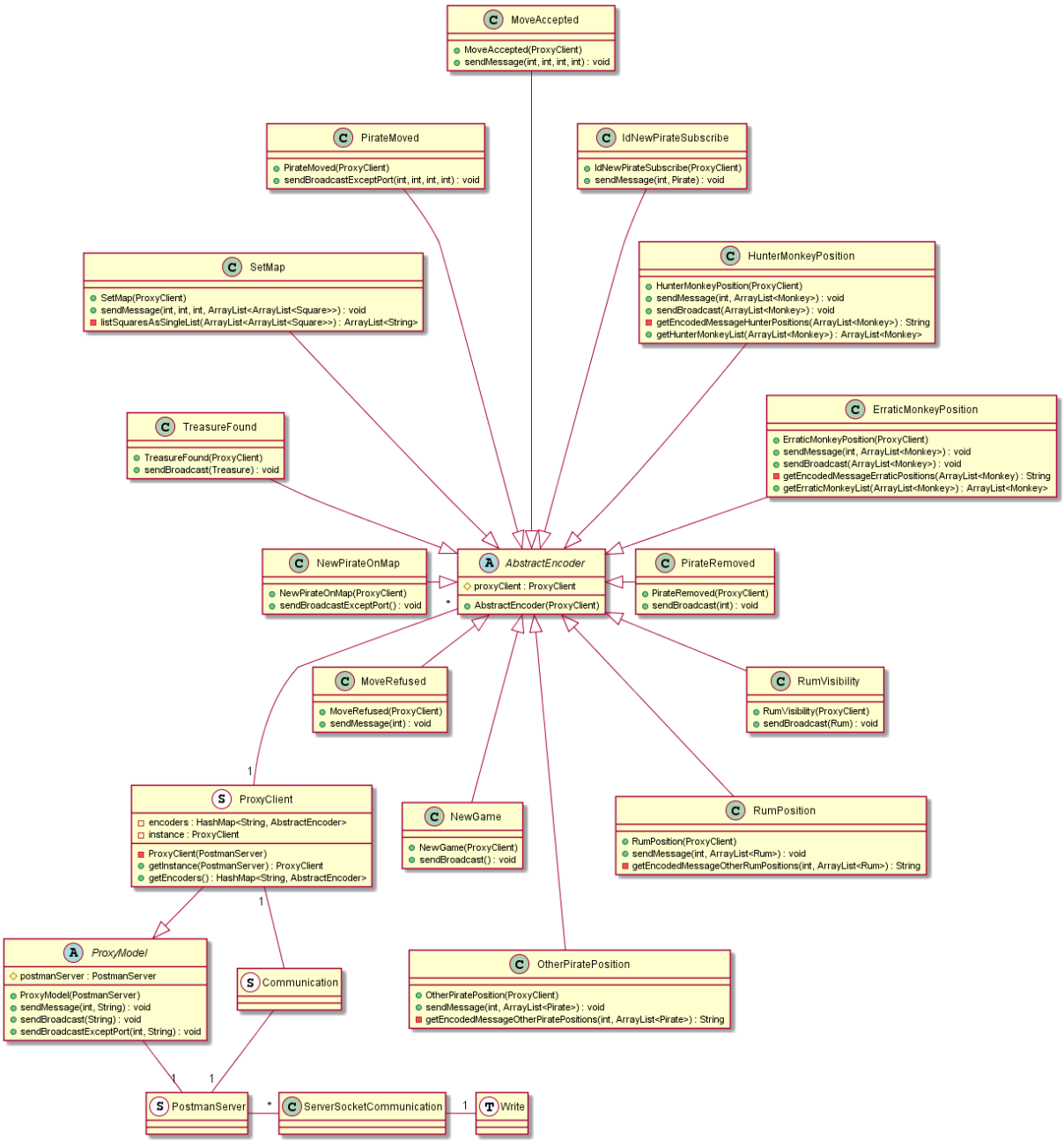


Figure 10: architecture "ProxyClient"

Comme énoncé précédemment, le « ProxyClient » s'occupe de gérer l'encodage des messages pour ensuite demander leur envoie au « PostmanServer » qui s'occupera ensuite de l'envoyer sur le socket de communication dans le thread d'écriture appelé « Write ».

Tous les encodeurs de message héritent directement de la classe « AbstractEncoder ». Ainsi, le « ProxyClient » possède une liste d'encodeurs appelés

« encoders ». A l'initialisation de ce « ProxyClient », celui-ci va créer tous les encodeurs et les ajouter à sa liste d'encodeurs.

Explication d'un scénario :
Lors d'une demande du « Brain » d'un envoie de message à la « Communication », la « Communication » va demander au « ProxyClient » d'envoyer le message. Pour ce faire, le « ProxyClient » va sélectionner en fonction de la commande à effectuer l'encodeur correspondant au message à envoyer. Une fois que l'encodage du message a été effectué, l'encodeur donne le message au « ProxyClient » qui va directement demander l'envoi du message en broadcast, multicast ou unicast au « PostmanServer ». Le « PostmanServer » va ensuite chercher le(s) socket(s) correspondant et écrire sur ce(s) socket(s) dans le(s) thread(s) d'écriture appelé(s) « Write ».

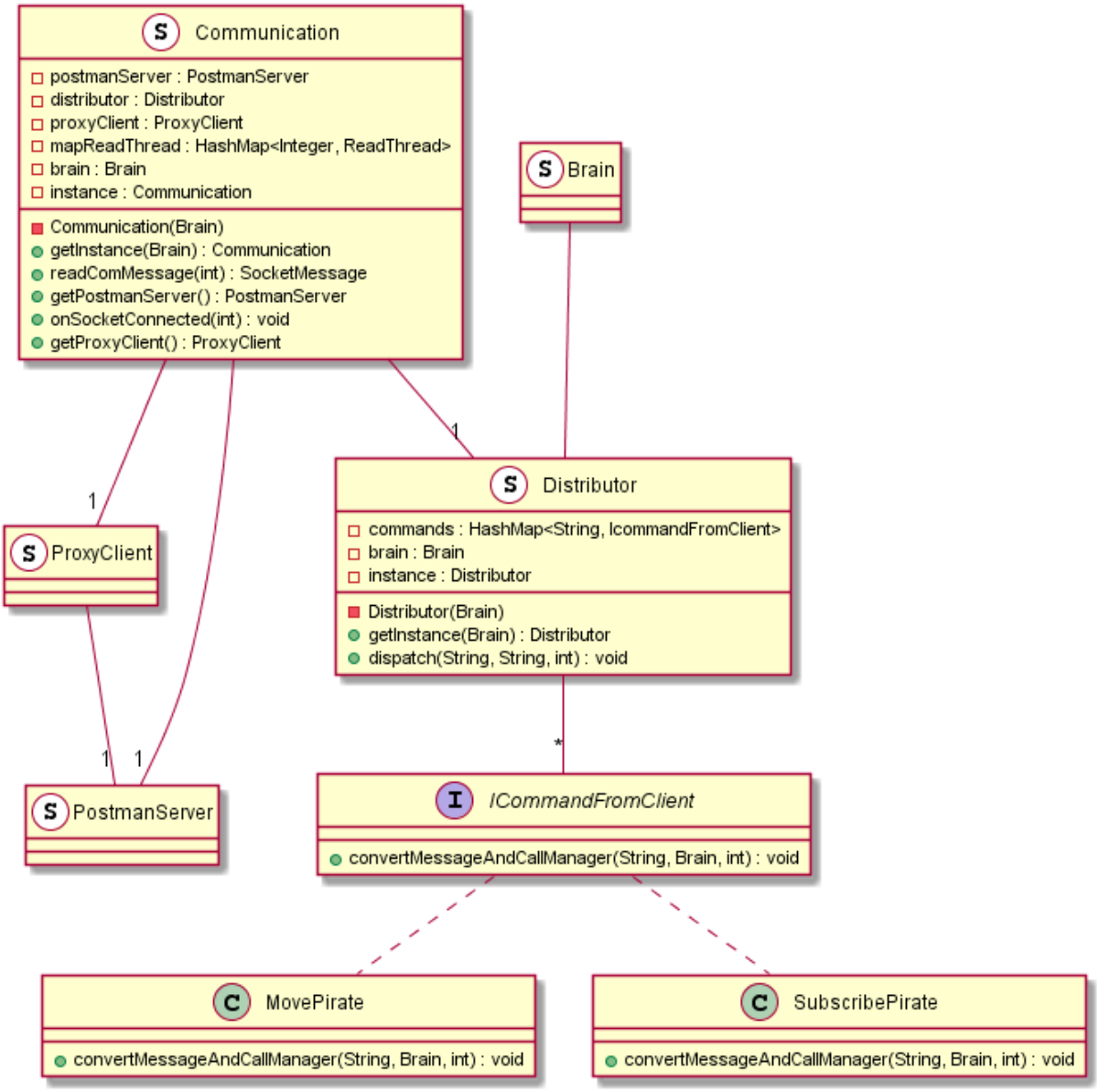
Patterns utilisés

Dans cette architecture, les différents patterns utilisés sont le singleton et le pattern commande. En effet, par l'intermédiaire de ce dernier pattern, notre « ProxyClient » va demander l'encodage des messages et ne récupérer que la chaine de caractère renvoyée par l'encodage des messages des encodeurs.

Dans cette architecture, nous aurions pu implémenter le pattern factory car il nous aurait permis de ne pas nous embarrasser avec tous les constructeurs d'encodeurs. Ce pattern nous aurait permis de ne pas surcharger notre « ProxyClient » et aurait offert une meilleure lisibilité du code.

5.4 – Architecture du « Distributor »

Le « Distributor » de notre système s'occupe de gérer la conversion des différentes commandes envoyées par le client pour ensuite appeler le « Brain » qui gérera cet événement. Voici ci-après l'architecture de ce « Distributor » :



Lors de la création du « Distributor », l'ensemble des commandes pouvant être envoyées par le client sont instanciées et ajouter à la liste de « ICommandFromClient » du « Distributor » appelée « commands ». Ces deux commandes « MovePirate » et « SubscribePirate » permettent de convertir la chaine de caractère reçue en une commande qui sera ensuite interprétée par le « Brain ».

Patterns utilisés

Dans cette architecture, le pattern commande a été utilisé pour convertir les messages : lorsqu'un message arrive par le socket de communication dans le thread de lecture de la « Communication », le messsage est envoyé au « Distributor » qui, par la méthode « dispatch », sélectionnera la commande (en ne récupérant que les deux premiers caractères de la chaine reçu) qui correspondra à la classe de

conversion du message. Après conversion du message en model, la commande est envoyée au « Brain » pour être ensuite interprétée.

Nous aurions pu aussi utiliser le pattern factory pour les mêmes raisons que sur l'architecture du « ProxyClient ». En effet, si nous avions beaucoup de commandes à instancier, le pattern facotry nous aurait permis de ne pas nous embarrasser avec tous les constructeurs des différentes commandes (implémentant ICommandFromClient). Ce pattern nous aurait permis de ne pas surcharger notre « ProxyClient » et aurait offert une meilleure lisibilité du code.

Table des illustrations