Projet de Développement

Conception Orientée Objet : Livraison Finale

Université de Nice Sophia Antipolis

Avril 2020

Unité de l'enseignement : Projet de développement

Enseignants : Philippe RENEVIER / Jérôme DELOBELLE

Licence 3 – MIAGE

SESSION 2019 –2020

Etudiants : BOUCHE Steven, CHAPOULIE Dorian, Longin Remi, KAROUIA Alaedine, GARNIER Corentin

Table des matières

Point de vue générale de l’architecture

Glossaire

ThreadPoolExecutor : Une sous-classe d'exécuteur qui utilise un ensemble de threads pour exécuter les appels de manière asynchrone.

Bassin / Pool : Le contenant des faces d’un même cout dans la forge

Island : Ile dans le temple

Callable : Retourne un objet suite à l'exécution des traitements.

Choice : Objet qui correspond au choix du joueur

Mapper : Objet permettant de coder/décoder afin de transmettre/recevoir via réseau

Pattern : Un modèle.

Netty : Librairie socket.io server.

Socket : Un lien de communication bidirectionnel entre deux programmes fonctionnant sur le réseau.

Event : Evènement

Emit : Envoie de donnée à une socket

Module : Correspond à une partie du projet.

Trigger : Provoque un traitement particulier en fonction d'événements.

Un run : démarrage d’un thread

Un bot : IA qui interagie avec le système.

Launcher : Permet de lancer le serveur

Thread : Unité d’exécution faisant parti d’un programme

Printer : Permet d’afficher une commande

Enum : Une énumération

Stats : Les statistiques

Setup : Mettre en place

Package : Contient des classes et des interfaces

Network : Tout ce qui concerne le réseau

Datalisteners : objet permettant de recevoir un certain type de ressource en particulier.

Représentation générale du projet Maven DiceForge

Le projet Diceforge est composé au total de 4 modules. Le module Launcher a pour rôle d'instancier un serveur de jeu ainsi que de l'instanciation des clients qui devront se connecter au serveur sans l'aide du launcher en réseau. Le module serveur crée la partie réseau via socketio netty et héberge toutes les données de la partie dans les classes dédiées dans le module share. Le module share est composé de toutes les classes utilisables par le client et le serveur permettant d'éviter des dépendances cycliques. Le module client aura simplement pour rôle de se connecter et d'attendre des demandes serveur, faire un choix en fonction des données passer par le serveur dans la demande puis d'envoyer sa réponse au serveur.

Fonctionnalités

Notre programme est composé de deux modes. S'il n'y a pas d'arguments alors il lance un launcher qui exécutera une partie de DiceForge. Le launcher lance un serveur de partie ainsi que tous les clients nécessaires afin de lancer une partie qui pour l'instant est uniquement composée de 4 joueurs La partie s'exécute alors en distribuant de l'argent en début de partie à tous les joueurs en fonction de leur ordre d'arrivée. Puis déclenche une faveur majeure pour tous les joueurs. Si le joueur tombe sur une face simplement avec un gain de ressource alors l'ajoute dans son inventaire sinon lui demande de choisir la ressource voulue pour les faces à choix multiple. Une fois que tous les joueurs ont répondu il demande au joueur actif qu'il a au préalable sélectionné s'il veut forger une face sur un de ses dés ou acheter une carte. Une fois son choix fait, il répond au serveur avec sa réponse et le serveur exécute le choix du client. Une manche est composée de tour actif étant égal au nombre de joueur. Une partie est composée de 9 manches. Dans le cas où l'argument "-p" suivie d'un nombre de partie est spécifiée le programme divisera le nombre de partie à exécuter en fonction d'une taille de pool de launcher dans une classe de configuration. Il lancera alors X launchers avec chacune des Y parties à exécuter. Le programme s'arrête quand tous les launchers ont fini leur travaille.

Fonctionnalités qui sont implémentées :

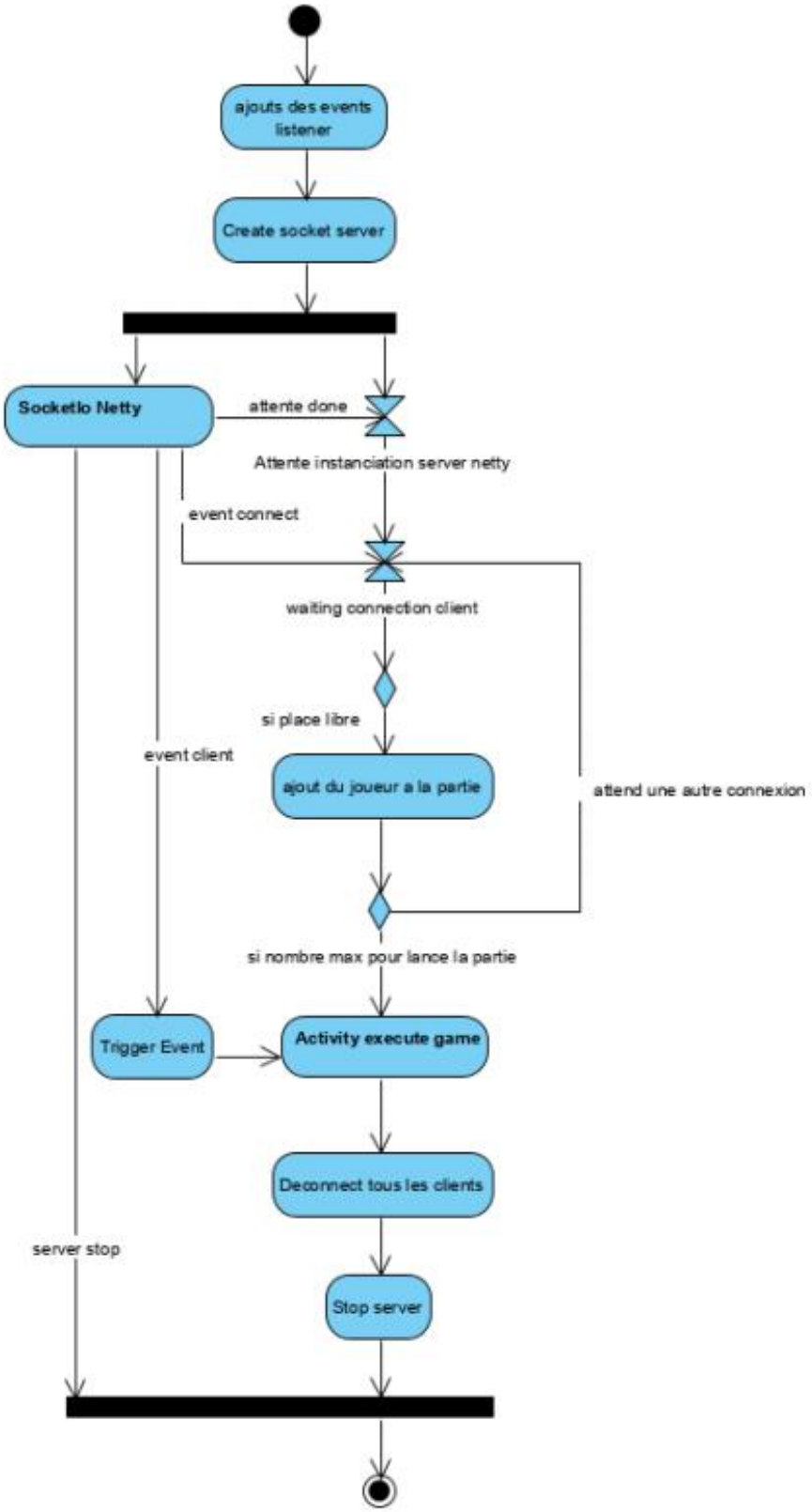
- Communication client server par réseau
- Boucle de jeux
- Système de commande
- Forge et Temple
- Toutes les actions des cartes basiques
- Toutes les actions des faces simple hybride et spéciale
- Multi partie parallèles
- Partie qui se joue uniquement à 4 joueurs
- Implémentation des statistiques
- Commencer à faire une visualisation des stats dans un browser mais pas terminer
- Plusieurs versions de bot

Diagrammes d'activités

Notre projet est composé de plusieurs activités. L'activité Main qui lance l'activité du launcher qui lui-même lance le serveur de partie ainsi que ses clients. Une fois tous les clients connectés le serveur exécute la partie. Dans cette partie Nous ne détaillerons pas l'activité concernant l'exécution de la partie car celle-ci sera plus détaillée dans la partie du serveur.

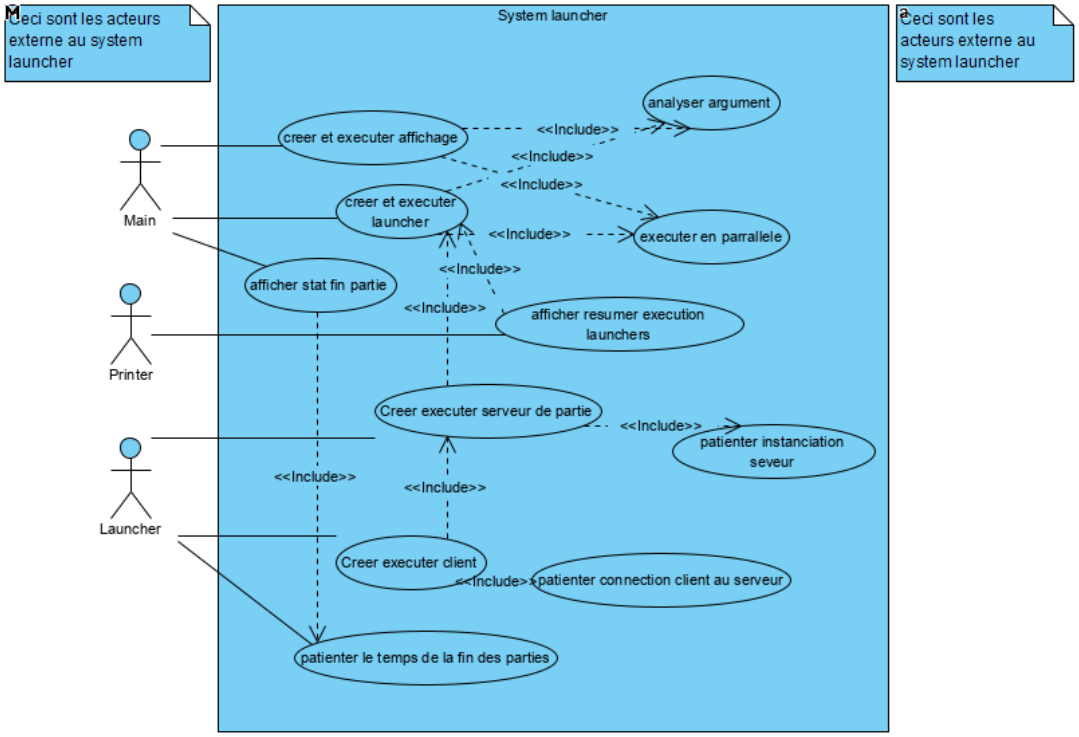
Diagramme d'Activité du Serveur

Nous avons détaillé la connexion dans la partie network. Certains détails dans le diagramme d'activité ont été volontairement omis comme la connexion au serveur et l'exécution de la Game. Cela est expliqué dans la partie réseau (voir Gameloop).


ajouts des events listener
Create socket server
SocketIo Netty
attente done
Attente instanciation server netty
event connect
waiting connection client
si place libre
event client
ajout du joueur a la partie
attend une autre connexion
si nombre max pour lance la partie
Trigger Event
Activity execute game
Deconnect tous les clients
server stop
Stop server

Module principal (Launcher)

Diagrammes Use Case



User Story

En tant que Main je souhaite pouvoir exécuter un Launcher ou plusieurs Launchers

En tant que Main je souhaite pouvoir faire varier l'affichage entre une ou plusieurs parties

En tant que Main je souhaite pouvoir afficher les statistiques de fin de parties

En tant que Launcher je souhaite pouvoir exécuter le serveur de parties créer.

En tant que Launcher je souhaite pouvoir exécuter les clients pour rejoindre les parties.

En tant que Launcher je souhaite pouvoir attendre l'exécution des parties.

En tant que Launcher je souhaite pouvoir récupérer les statistiques des parties et les merges.

En tant que Launcher je souhaite pouvoir renvoyer au Main les résultats des parties.

En tant que PrinterLauncher je souhaite récupérer les informations du launcher pour les affichées en multi parties.

Scénarios

1. Créer et exécuter les launchers Main
    a. Analyser les arguments
    b. Instancier et exécuter le printer en fonction des arguments dans un thread
    c. Instancier le bon nombre de launchers en fonction des arguments avec sa configuration
    d. Exécuter le launcher dans un thread
    e. Attendre la fin d'exécution des launchers
    f. Afficher les résultats des launchers

2. Créer et exécuter le serveur Launcher
    a. Instancier un objet serveur avec sa configuration
    b. Exécuter le serveur dans un thread
    c. Attendre l'initialisation du serveur
    d. Créer et exécuter les clients

3. Créer et exécuter les clients Launcher
    a. Instancier 4 clients avec leurs configurations
    b. Exécuter les clients dans un thread
    c. Attendre la connexion des clients au serveur
    d. Attendre la fin des threads serveur et clients

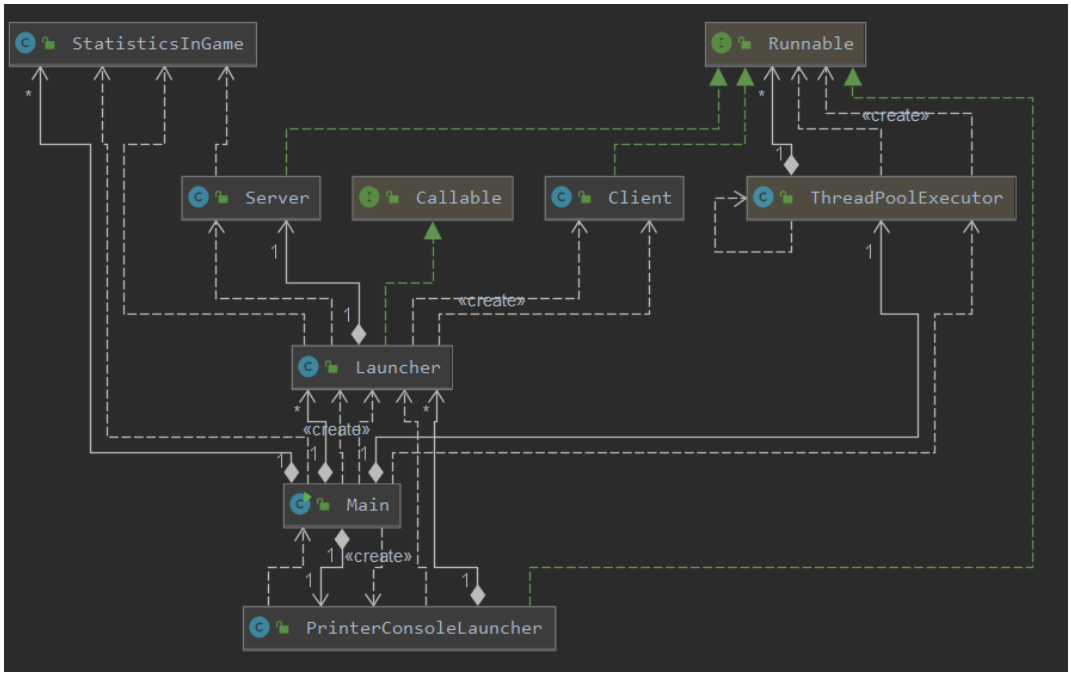
4. Fin exécution serveur et clients
    a. Récupération des stats
    b. Finir l'exécution Launcher et renvoyer les stats

5. Fin exécution des parties Main
    a. Récupérer les résultats des Launchers
    b. Fusionner les stats des launchers
    c. Afficher les stats
    d. Mettre fin au programme

6. Afficher les données du launcher
    a. Récupérer les données launchers
    b. Afficher un récapitulatif

Conception Logicielle

Point de vue statique



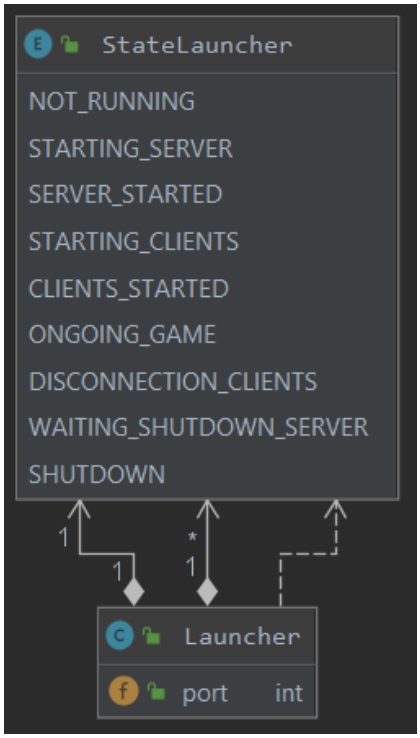
L'entrée de notre programme est la classe Main ci-dessus. La classe Main est composée de plusieurs Launcher, cela dépend en fonction des variables de configurations dans le package share. De plus le Main est composé d'un printer qui permet d'afficher l'état en cours des launchers quand le mode multi parties est activé, sinon nous gardons un affichage classique.

Le but d'un Launcher est d'exécuter X parties calculées par le Main. Par exemple, si nous avons configuré l'utilisation de 5 Launcher pour le Main et que l'on souhaite exécuter 500 parties, un Launcher va alors exécuter 100 parties. Il lancera alors un serveur de partie dans un thread à part et créera les threads clients associés aux serveurs jusqu'à attendre la fin des parties du serveur.

Le Main lancera alors X Launcher devant exécuter X parties. Le Main lance alors tous les launchers en parallèle c'est à dire qu'ils vont chacun exécuter leurs parties en même temps. Les Launchers renverrons en résultat leurs statistique des parties qu'ils ont exécuté.

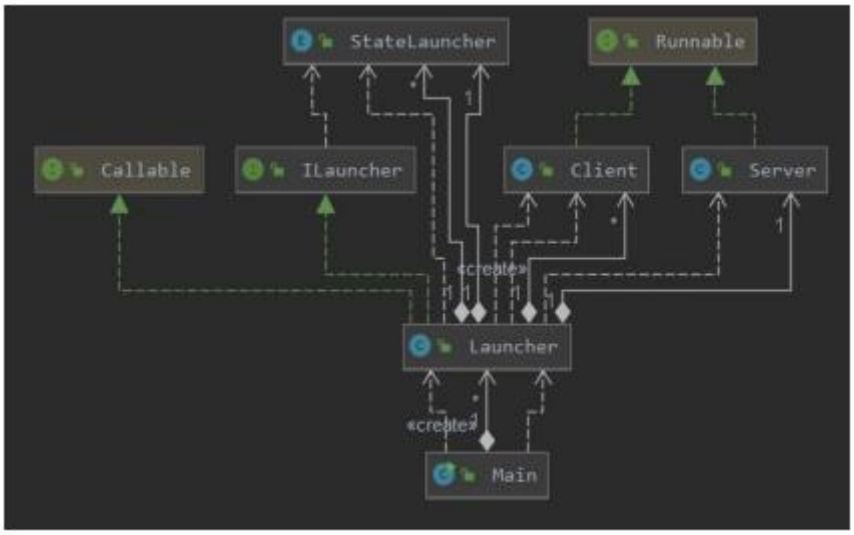
Pour lancer les threads le Main utilise un ThreadPoolExecutor lui permettant de gérer les threads et leurs résultats sous forme de Futur pour les threads Callable Launcher qui renvoi un objet StatisticsInGame.

Un launcher est composé d'un état disponible dans l'enum State Launcher.



Point de vue statique delta itération 5

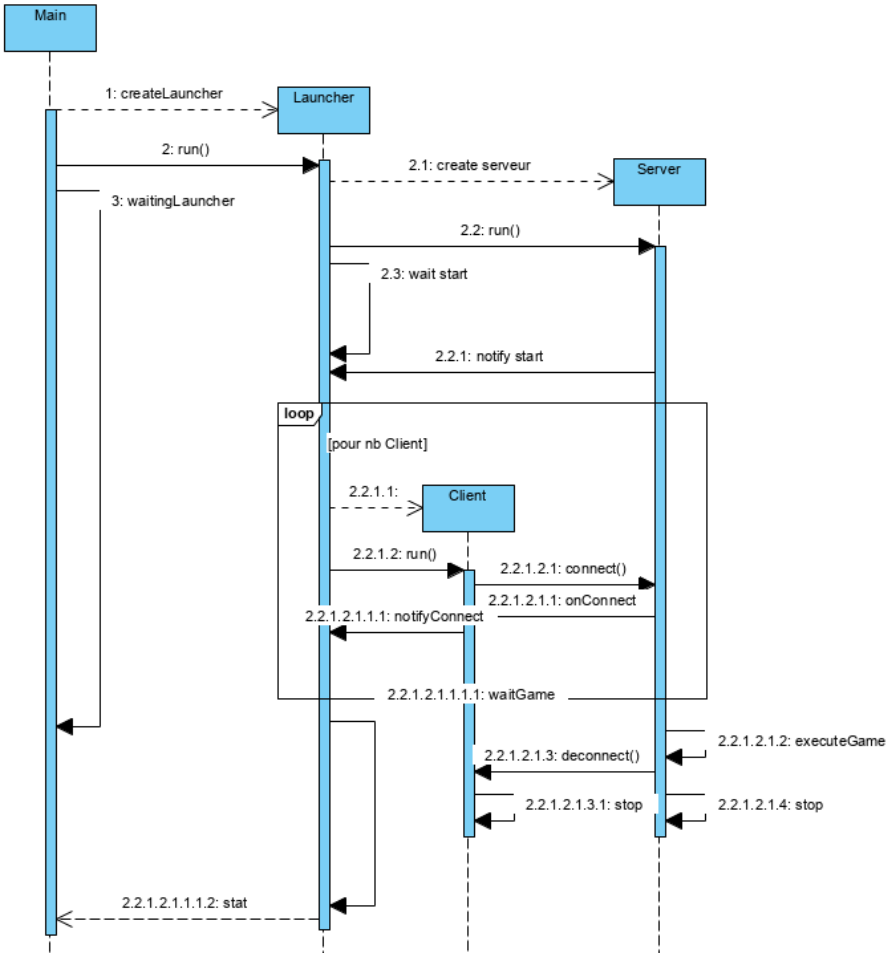
Rappel itération 5 :



Par rapport à l'itération 5 concernant le module Launcher, il n'a pas subi de très grosses refontes. Il prend désormais en compte les statistiques rendu par l'exécution des parties.

Point de vue dynamique

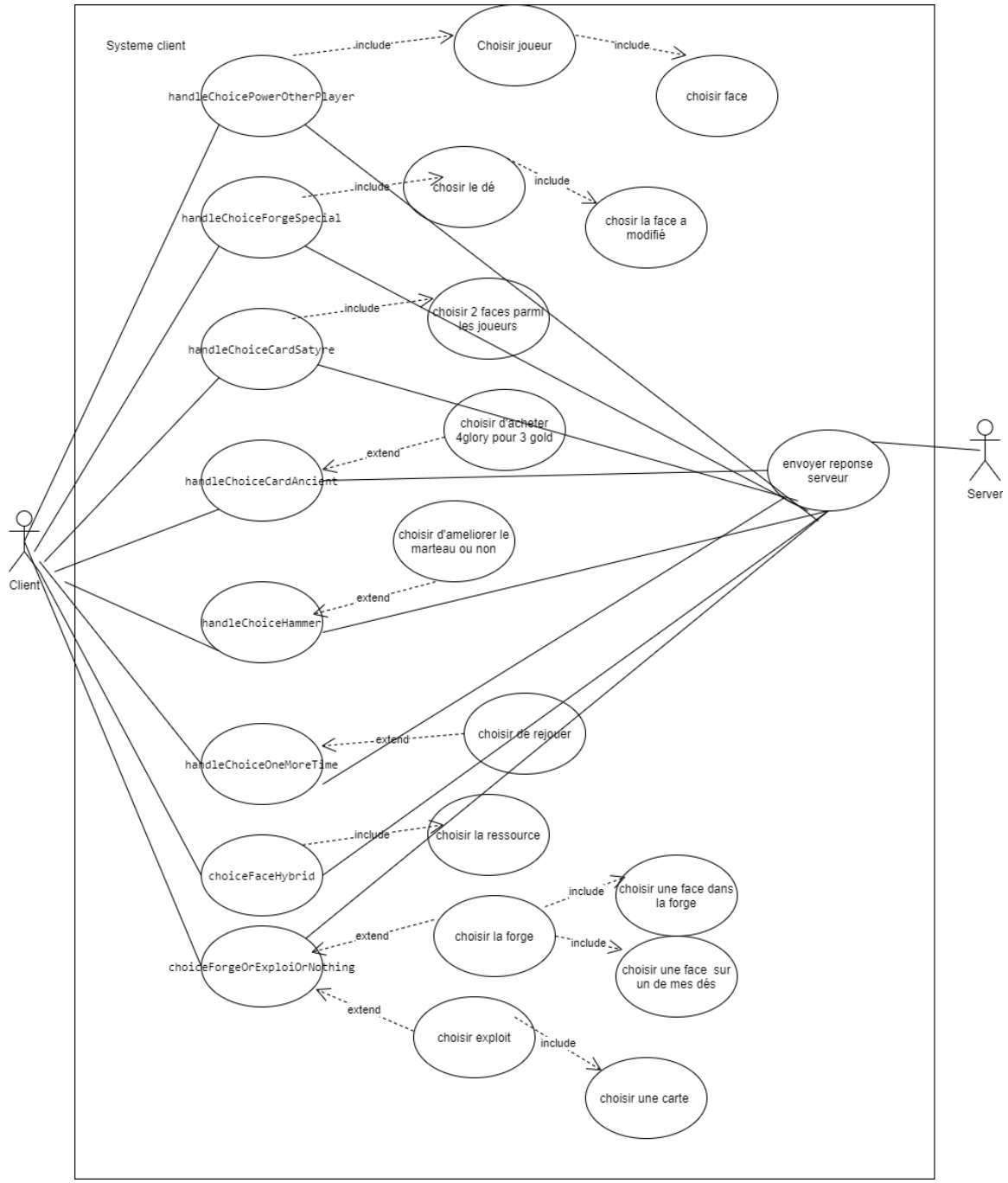
Nous prenons l'exemple de l'exécution d'une seule partie :



Ce diagramme détail l'exécution général de la partie Launcher. Le rôle du Main est d'instancier le Launcher, puis attend son résultat. Le launcher une fois run instancie le serveur et le run, il attend ensuite la notification du serveur comme quoi le serveur est bien lancer. Ensuite pour le nombre de client nécessaire le Launcher instancie X clients puis les runs. Il attend entre chaque instanciation client qu'il notifie le launcher lorsqu'il s'est bien connecté au server. Une fois tous les clients connecter le serveur lance la partie et le launcher attend la fin d'exécution des parties. Une fois fini le serveur déconnecte les clients et stop le thread. Cela réveille le launcher qui récupère les résultats pour finir son exécution et renvoyer les résultats au Main qui les affiche et arrête son traitement.

Module Client / Joueur

Diagrammes Use case



User Story

En tant que joueur je souhaite pouvoir choisir entre forger et exploit.

En tant que joueur je souhaite pouvoir choisir la face de dé à modifier.

En tant que joueur je souhaite pourvoir choisir la face ajouter sur un de mes dés.

En tant que joueur je souhaite pouvoir choisir une carte dans une des iles.

En tant que joueur je souhaite pouvoir choisir entre plusieurs ressources lorsque que je lance les dés et tombe sur une face à choix.

En tant que joueur je souhaite pouvoir rejouer si je le peux.

En tant que joueur je souhaite être capable d'améliorer la carte marteau si je l'ai.

En tant que joueur je souhaite pouvoir choisir d'acheter 4 points de gloire contre 3 gold si j'ai la carte 'Ancient'.

En tant que joueur je souhaite pourvoir choisir entre 2 faces parmi les joueurs s'il a la carte "Satyre".

Scénario

Précondition :

Le serveur a précédemment envoyé une demande de choix.

Postcondition :

Le client a renvoyé l'objet choice.

Tout d'abord le client décode le choix que le serveur lui a envoyé puis en fonction du choix il va agir :

- HandleChoicePowerOtherPlayer :

  Le joueur choisit un joueur parmi les autres joueurs et choisi une des deux faces des dés du joueur choisi a utilisé pour soi.

- HandleChoiceForgeSpecial :

  Le joueur choisit un de ces 2 dés puis choisit la face à remplacer par la face spéciale.

- HandleChoiceCardSatyre :

  On choisit 2 faces parmi le lancer de dés des autres joueurs.

- HandleChoiceCardAncient :

Le joueur choisi oui ou non d'acheter les 4gloires contre 3 gold.

- HandleChoiceHammer :

Le joueur choisit d'améliorer ou non son marteau s'il a assez de gold et si oui de combien il souhaite l'améliorer.

- HandleChoiceOneMoreTime

  Le joueur choisit de rejouer s'il a les 2 Solar nécessaires.

- ChoiceFaceHybrid

  Le joueur choisit une ressource entre les ressources qui lui sont proposées sur la face hybride.

- HandleChoiceForgeExploitAndDoAction

  - ChoiceTemple

    S'il n'y a plus de cartes dans les iles, le joueur va chercher à forger si cela n'a pas déjà été testé. Sinon le joueur choisit une ile selon le cout et choisit une carte à acheter.

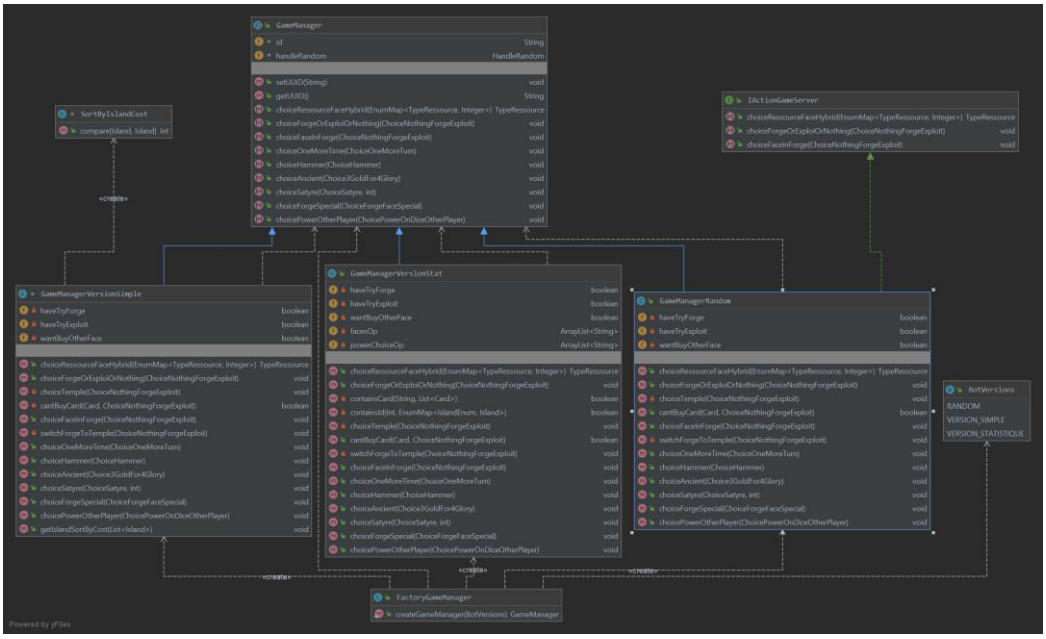
  - ChoiceFaceInforge

    S'il n'y a plus de face à acheter dans la forge le joueur va chercher à acheter des cartes si cela n'a pas déjà été testé. Le joueur choisit un bassin et la face qu'il souhaite acheter s'il peut. Puis choisi une face sur l'un de ces dés à modifié.

Conception Logicielle

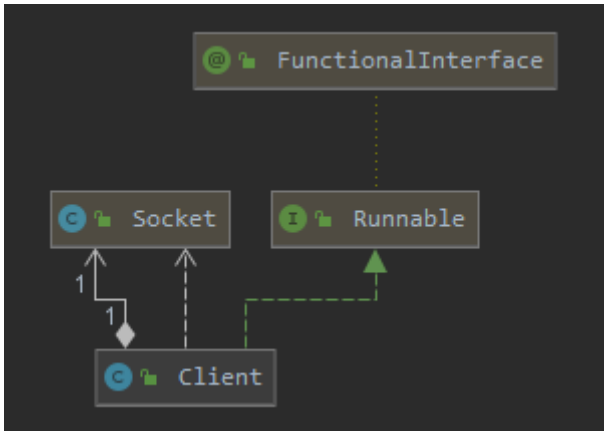
Point de vue statique itération Finale

https://zupimages.net/up/20/16/u2vk.png

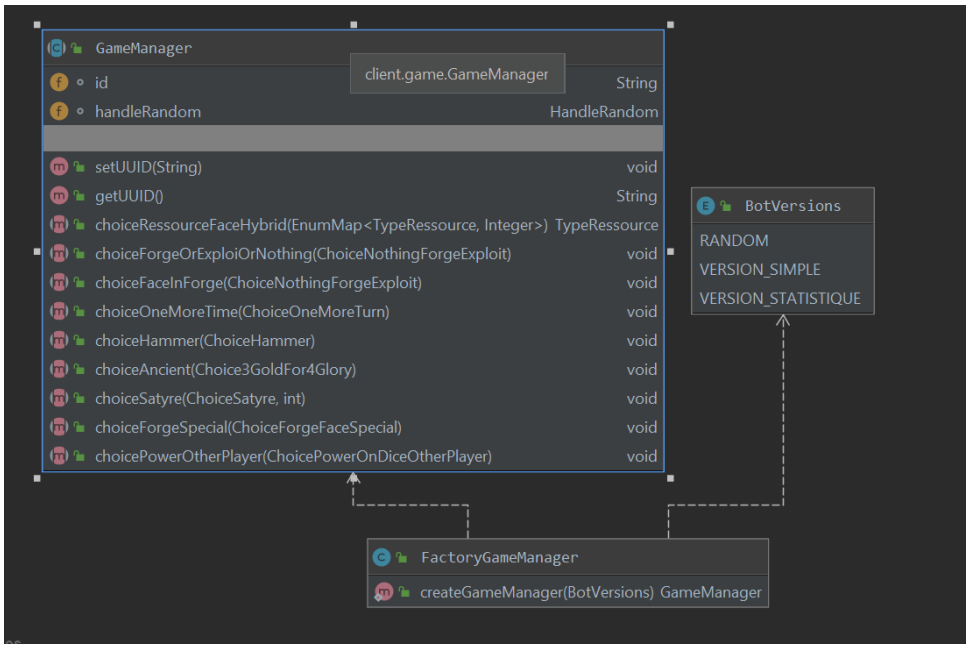


Point de vue Réseau :

La partie Réseau est intégrée au client. Le client est composé d'un socket qui permet de s'inscrire aux évents et d'y répondre via un émit. Cela permet au client de se connecter au serveur.

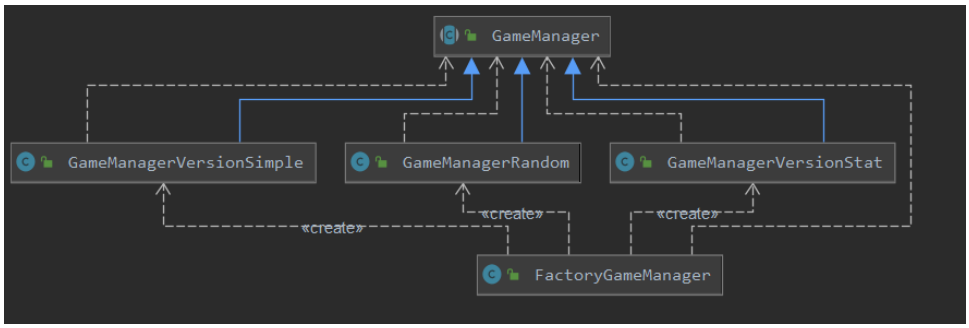


Point de vue statique delta itération 5



Nous avons réalisé la partie client sur un Design pattern fabrique simple qui utilise un pattern Stratégie qui en fonction de l'Enumeration BotVersion, la classe FactoryGameManager va instancier le bon gameManager .

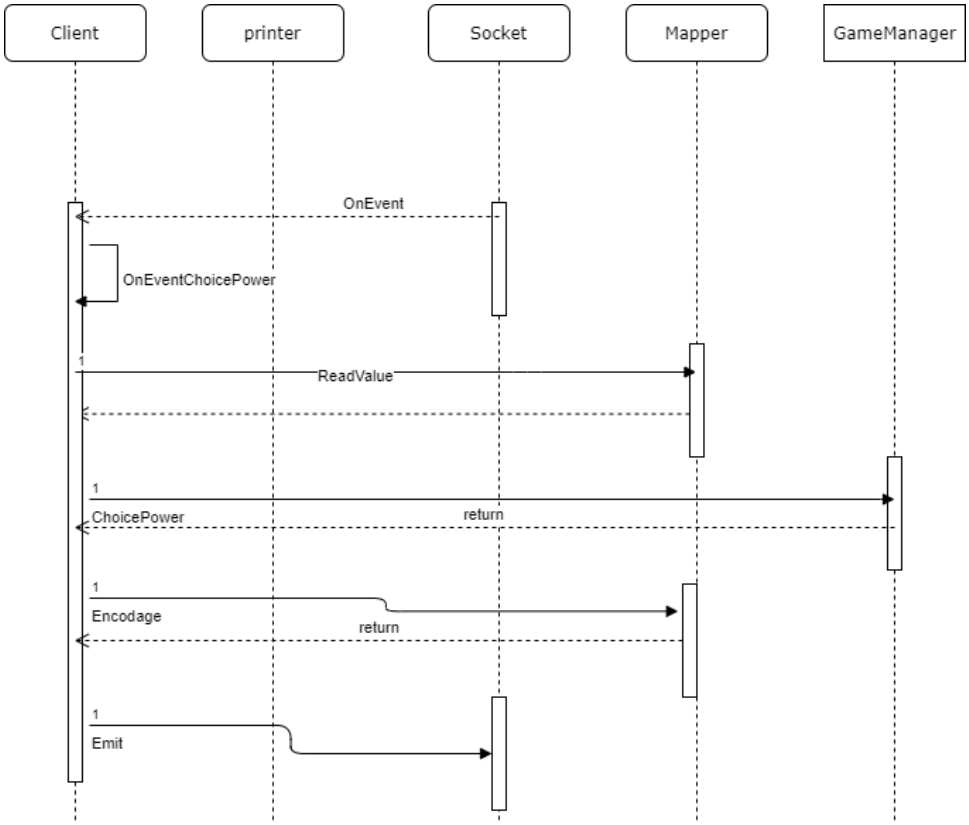
Nous détaillerons dans la partie dynamique les différences entre les 3 bots.



Point de vue dynamique itération Finale

Toutes les actions menées par le joueur suivent un même schéma. Les diagrammes de séquences dans la partie ont donc le même format. Nous avons choisi de prendre comme exemple le choix où il faut choisir la face de l'un des joueurs.

Tout d'abord le serveur envoie l'Event lié à ce choix, puis coté client, il va utiliser une méthode sur lui-même pour pouvoir gérer l'évent. Il décoder ce qu'il a reçu via le serveur grâce à l'objet Mapper puis envoyer le contenu au GameManager en question qui lui va renvoyer le choix du joueur. Le choix du joueur va ensuite être encoder par avec le Mapper puis va être émis par le client au serveur.



Bot version Random :

Le bot Random aura toujours un choix aléatoire dans la limite des données du jeu.

Bot version Simple :

Le bot version Simple va toujours chercher à acheter le premier élément.

En effet, par exemple dans la forge lorsqu'il doit choisir, le bot va essayer d'acheter une face dans le bassin le moins cher. Et de même pour les cartes, il cherchera à acheter sur l'ile la moins cher. Le bot cherchera toujours a amélioré le marteau, Nous avons implémenté une classe avec une méthode qui permet de trier les iles selon leur cout. Nous avons gardé quelques éléments aléatoires afin d'éviter par exemple que le bot ne modifie pas toujours sa même face ou encore qu'il n'essaye que de forger à chaque fois, dans ces cas les choix restent aléatoires. Seuls les achats, les choix de ressources lors des choix de face hybride.

Bot version Statistique :

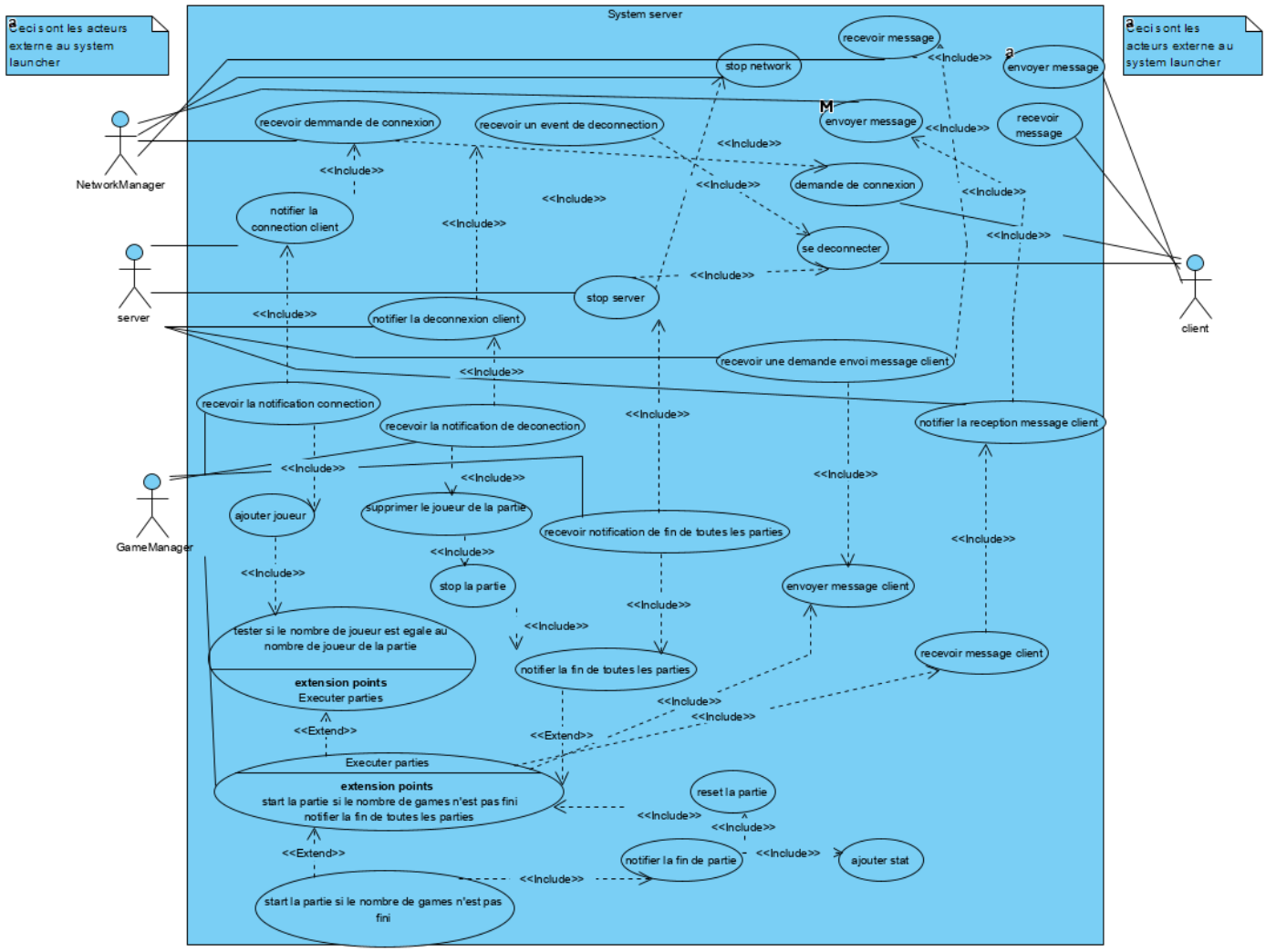
Nous avons récupéré les statistiques de centaines de parties en utilisant les 2 premiers bots. Et avec ces données que nous avons récupérées, les choix du bot statistique seront en priorité les choix avec le plus de chance de gagner.

Point de vue dynamique delta itération 5

Par rapport à la dernière, il n'y a pas de changement dans le système mais juste des choix supplémentaires.

Module Serveur / Moteur de jeu

Diagrammes Use Case



User Story

En tant que serveur je souhaite pouvoir exécuter une partie ou plusieurs parties via un gestionnaire de parties

En tant que serveur je souhaite pouvoir connecter des clients via un gestionnaire réseau

En tant que serveur je souhaite pouvoir envoyer des demandes aux clients via un gestionnaire réseau

En tant que serveur je souhaite pouvoir recevoir les différents évènements client via un gestionnaire réseau

En tant que gestionnaire de partie je souhaite pouvoir exécuter une boucle de jeux

En tant que gestionnaire de partie je souhaite pouvoir exécuter des commandes via un gestionnaire de commande.

En tant que gestionnaire de partie je souhaite pouvoir rediriger les réponse client vers la commande en attente via le command manager.

En tant que gestionnaire de partie je souhaite pouvoir envoyer un message aux clients via le serveur.

En tant que gestionnaire de commande je souhaite pouvoir créer et exécuter des commandes

En tant que gestionnaire de commande je souhaite pouvoir rediriger là les réponses du gestionnaire de partie vers la commande en attente de choix.

En tant que gestionnaire réseau je souhaite pouvoir connecter des clients ou les déconnecter

En tant que gestionnaire réseau je souhaite pouvoir rediriger les réponses client vers le serveur

En tant que gestionnaire réseau je souhaite pouvoir envoyer un message aux clients

En tant que commande je souhaite pouvoir accéder aux objets de la partie

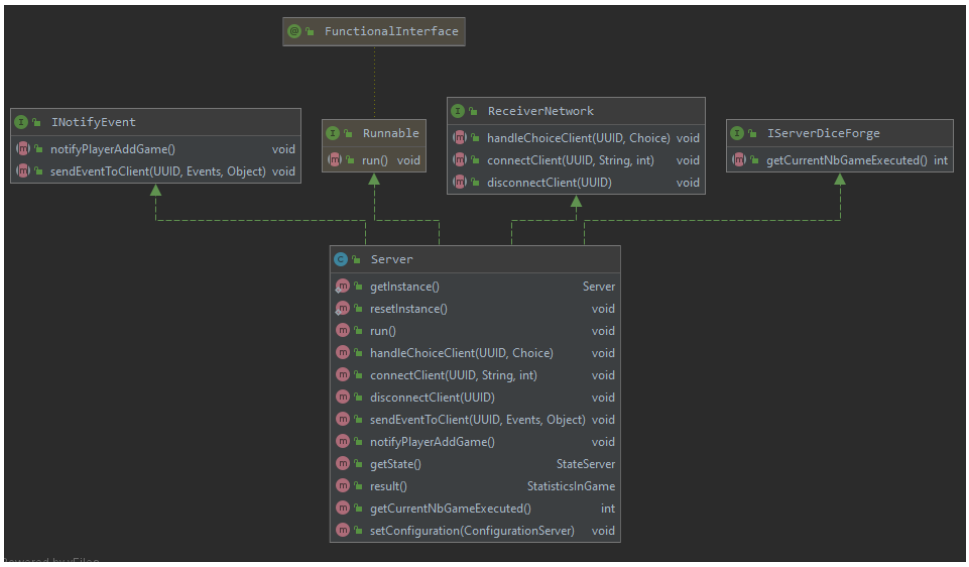
En tant que commande je souhaite pouvoir déclencher d'autres commandes via le gestionnaire de commande

En tant que commande je souhaite pouvoir attendre une réponse d'un client via le gestionnaire de commande

Conception Logicielle

Point de vue statique itération Finale

Server

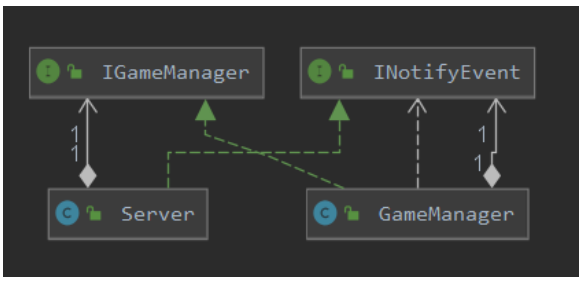


La classe principale du package serveur est la classe Server. La classe Server a pour but de faire le médiateur entre notre réseau et notre gestion de la partie. Il implémente alors l'interface Runnable qui permettra au Serveur d'exécuter les parties en parallèle des clients.

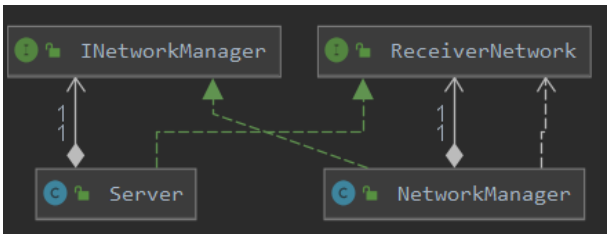
De plus il implémente d'autres Interfaces. INotifyEvent pour faire le lien avec la classe de gestion de la partie. Mais aussi ReceiverNetwork pour pouvoir recevoir des données de la partie réseau.

De plus notre serveur est composé d'un IGameManager qui s'occupe de la gestion de la partie, un INetworkManager qui aura pour rôle d'interagir avec les clients c'est à dire les joueurs. Il fait donc la transition entre nos clients et notre partie.

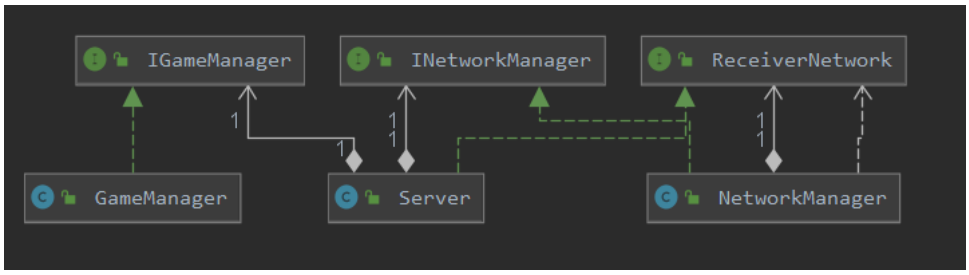
Le GameManager est alors composé d'un INotifyEvent lui permettant d'envoyer une requête à un joueur via le Serveur mais aussi de l'avertir quand un joueur à bien était ajouté à la partie au Serveur.



Le NetworkManager est alors composé d'un ReceiverNetwork lui permettant de l'avertir quand un client a répondu, s'est connecté ou s'est déconnecté.



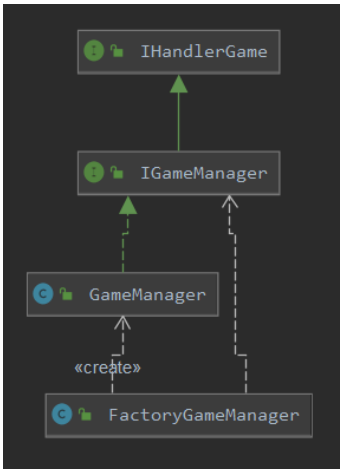
Pour conclure de manière générale sur l'aspect de la classe server, elle permet l'interaction entre les clients et la partie en cour comme nous pouvons le voir sur le diagramme ci-dessous, server sert de passerelle entre le réseau et la partie.



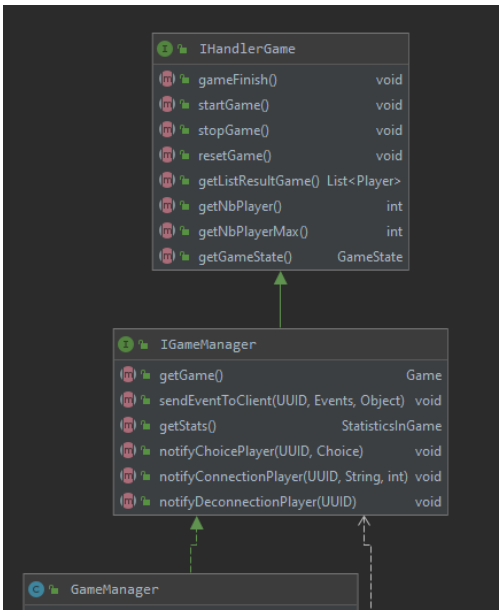
Nous allons maintenant détailler le gestionnaire de partie et son fonctionnement.

GameManager

Tout d'abord le GameManager est créé par une factory ce qui permet de casser la dépendance de la création du GameManager avec le Server. De plus le GameManager implémente une interface qui implémente une autre interface.

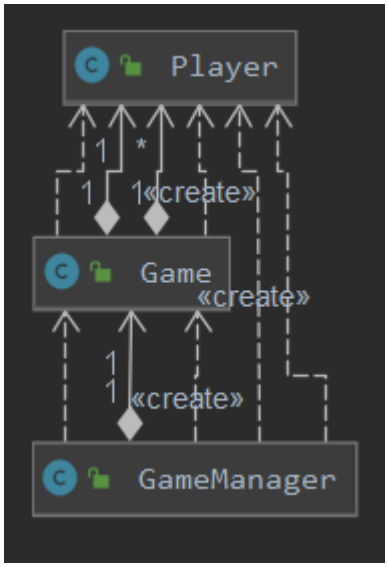


L'interface IHandlerGame a un objet de pouvoir avoir un comportement de gestion de parties. Il peut start, stop et reset par exemple la partie. De plus nous avons l'interface IGameManager qui est une spécialision de IHandlerGame permettant au GameManager de pouvoir transmettre l'objet de la partie en cours, les statistiques, mais aussi de pouvoir réagir dans le cas d'une connexion/déconnexion client mais aussi de réagir à un message client, et de pouvoir envoyer un message à un client. Nous pouvons voir cela ci-dessous.

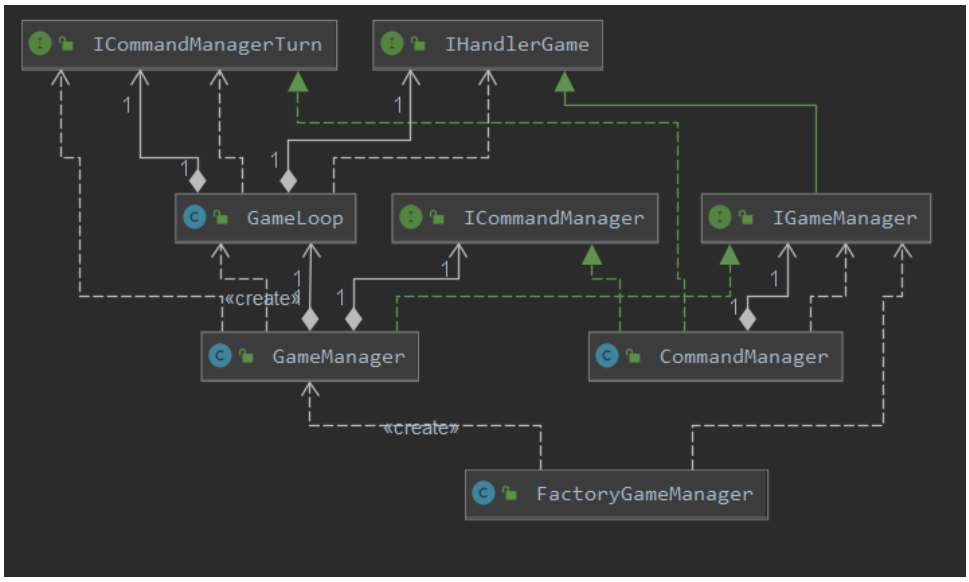


Nous allons maintenant regarder ce qui se passe derrière le GameManager, comment il fonctionne.

Pour commencer le GameManager possède une classe Game. La classe Game est la classe principale du package share que nous verrons plus tard. La classe Game est en fait le modèle ou le domaine de l'application. C'est un objet qui contient toutes les données de la partie. Nous considérons alors que la classe Game est le contexte du GameManager.

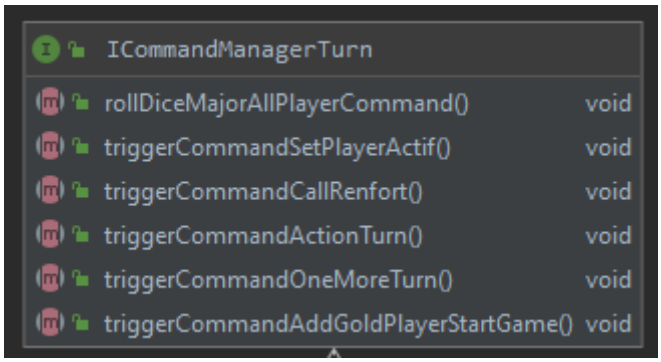


D’un point de vue général :



Nous pouvons voir qu’il y a principalement deux classes en relation avec le GameManager, le GameLoop et le CommandManager. Nous verrons le comportement du CommandManager et son interconnexion avec le GameManager plus tard et nous allons nous concentrer sur le GameLoop.

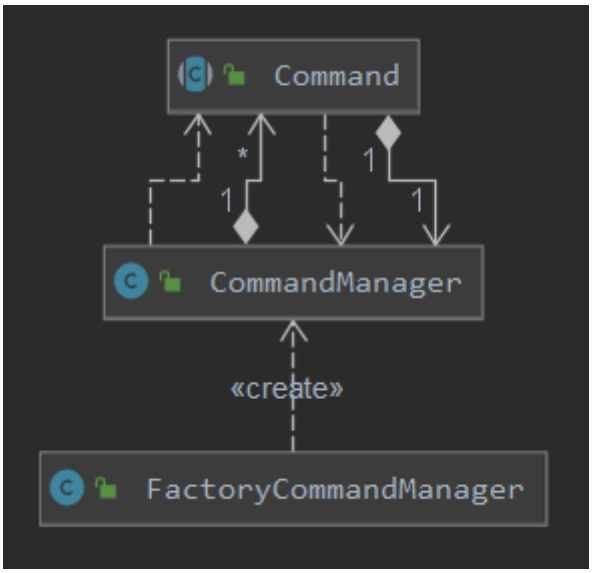
Le GameLoop est une classe permettant au GameManager de dédier la responsabilité de l’exécution de la partie. C’est lui qui exécutera la boucle de jeu. Il est alors composé de deux Interfaces ICommandManagerTurn pour exécuter des taches concernant les tours et IHandlerGame pour manipuler le contexte de la partie c’est à dire l’objet Game mais aussi manipuler l’état de la partie.



L'interface ICommandManagerTurn permet au game loop d'executer des commandes, et permet au game loop de dédié la responsabilité d'une tache lors du tour à une commande comme par exemple de changer le joueur actif de la partie.

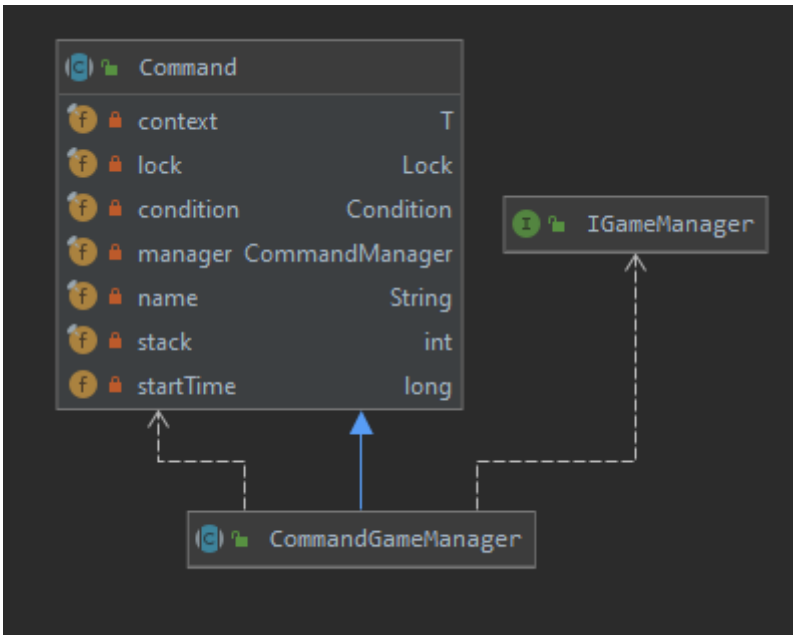
Nous allons maintenant détailler le CommandManager.

CommandManager

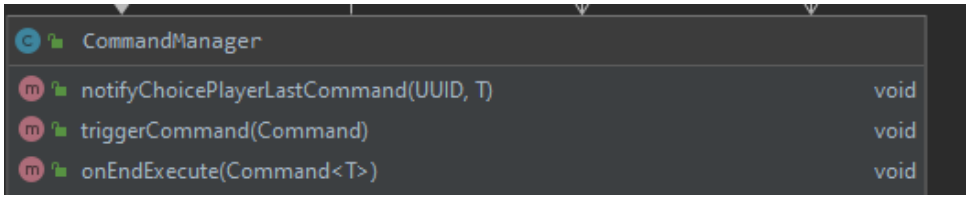


Pour commencer la responsabilité de la création du CommandManager est dédiée à une factory permettant de casser ce lien de dépendance avec le GameManager. Nous constatons aussi que le CommandManager est composé de plusieurs commandes. Il est responsable de la création et exécution des commandes.

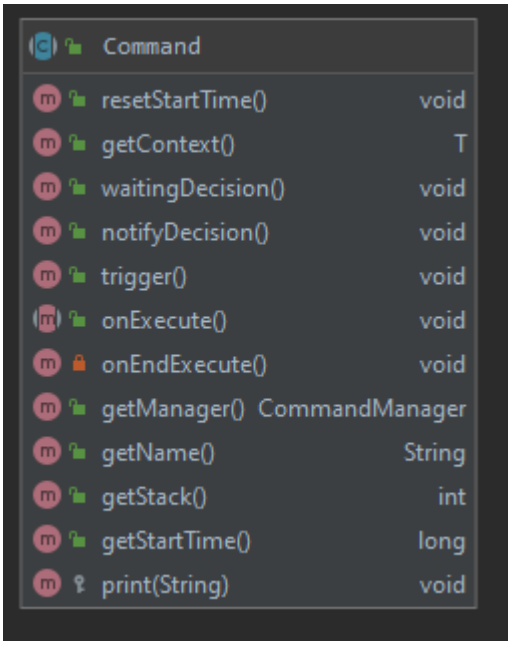
Plus en détails la classe Command est une classe abstraite, une super classe. Elle a pour rôle d'accéder à un contexte qui est une classe générique comme on peut le voir avec T. Elle permet aussi d'effectuer de la synchronisation via le Lock et une Condition pour l'attente.



Le CommandManager centralise tout le système de trigger des Commandes. C'est le CommandManager qui a la responsabilité de trigger des objets Command.

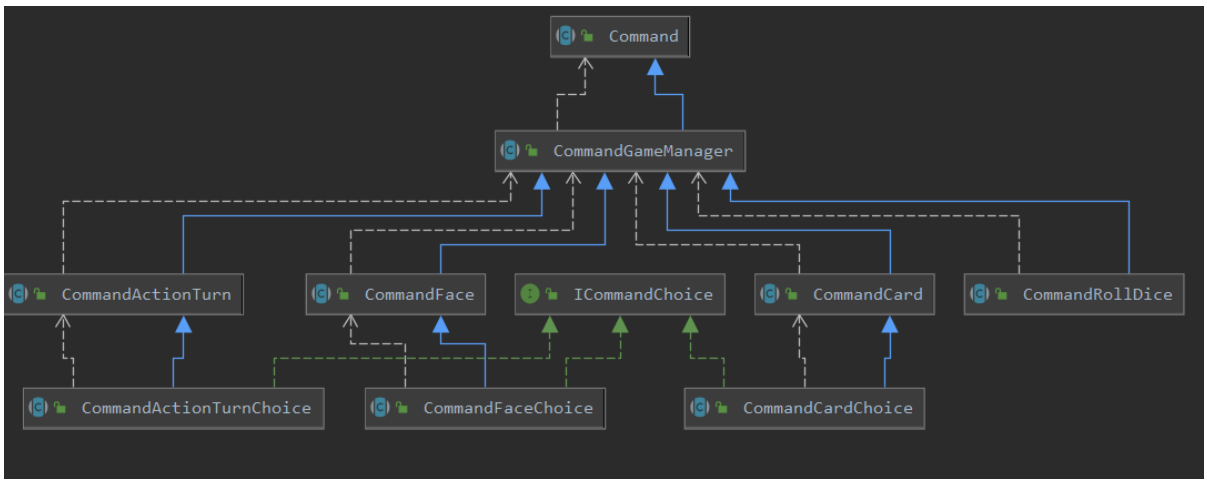


Quand le CommandManager trigger une commande il l'empile sur un stack et à sa fin la commande notifie le CommandManager de la fin de son exécution et dépile la commande. Cela nous permet de garder une trace de l'enchainement des commandes.



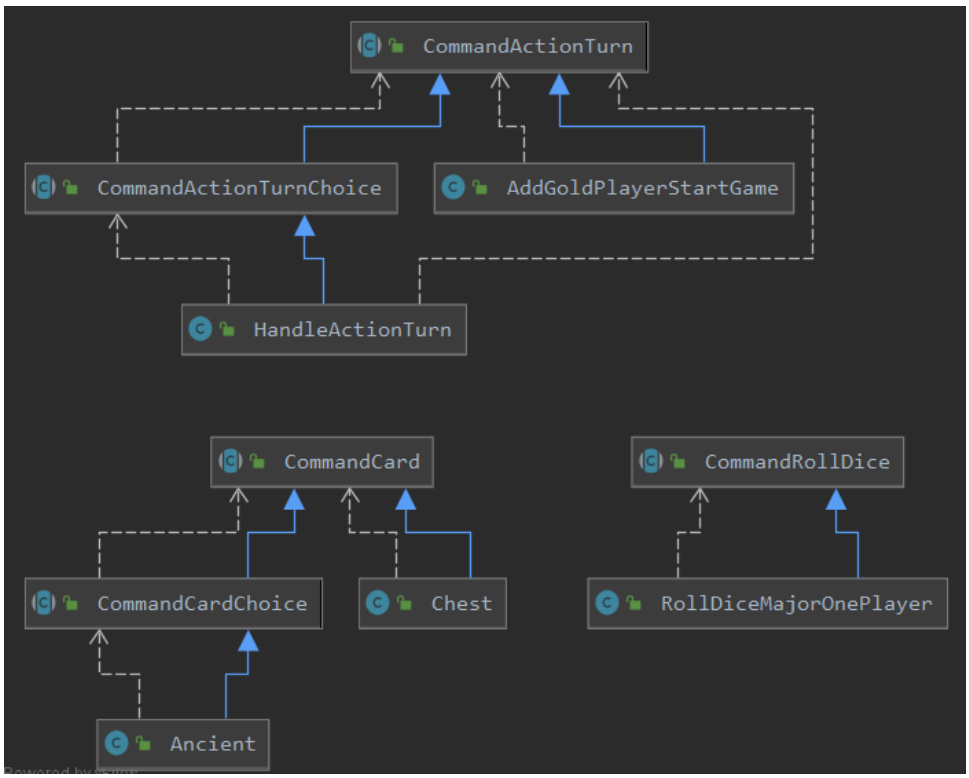
La classe CommandGameManager est elle aussi une super classe héritant de la classe Command. Via l'héritage la classe CommandGameManager spécifie comme type générique du contexte de la classe Command la classe IGameManager. Cela permet à une commande héritant de CommandGameManager d'accéder aux données de la partie. De plus une commande référence son manager (CommandManager) permettant d'interagir avec d'autres commandes.

Nous obtenons alors l'architecture suivante pour notre système de commande :



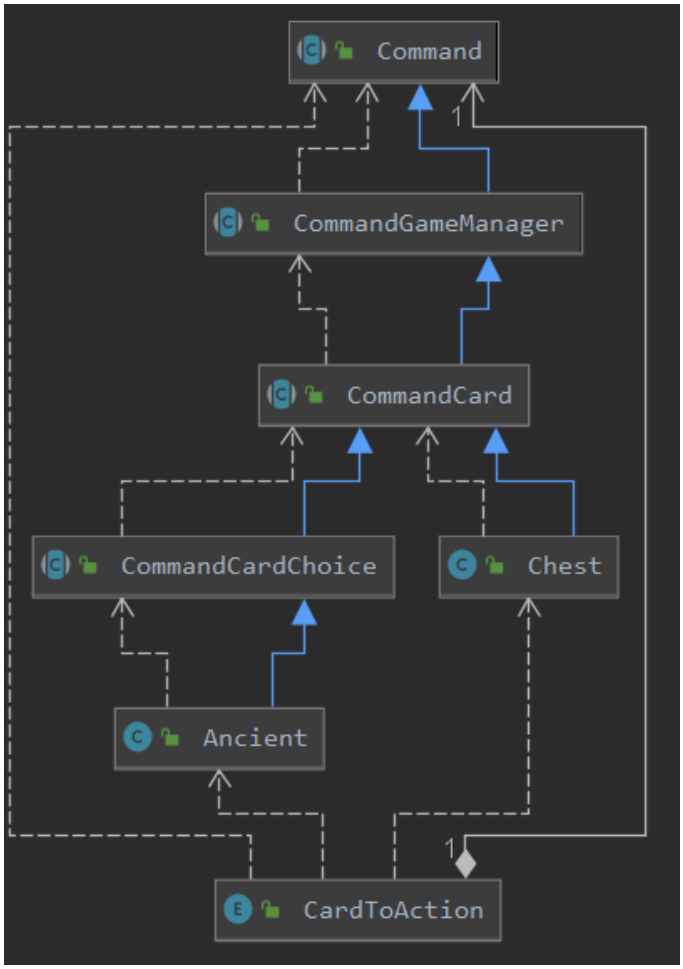
Nous n'avons pas mis toutes les commandes existantes, simplement quelques classes héritant de CommandGameManager. Nous pouvons voir que de nouveaux types de commandes existent. Des commandes permettant de faire des actions concernant le tour de la partie en cour, mais aussi des commandes de tour nécessitant un choix client. C'est pourquoi l'interface ICommandChoice existe. Il permet à une commande de pouvoir bloquer

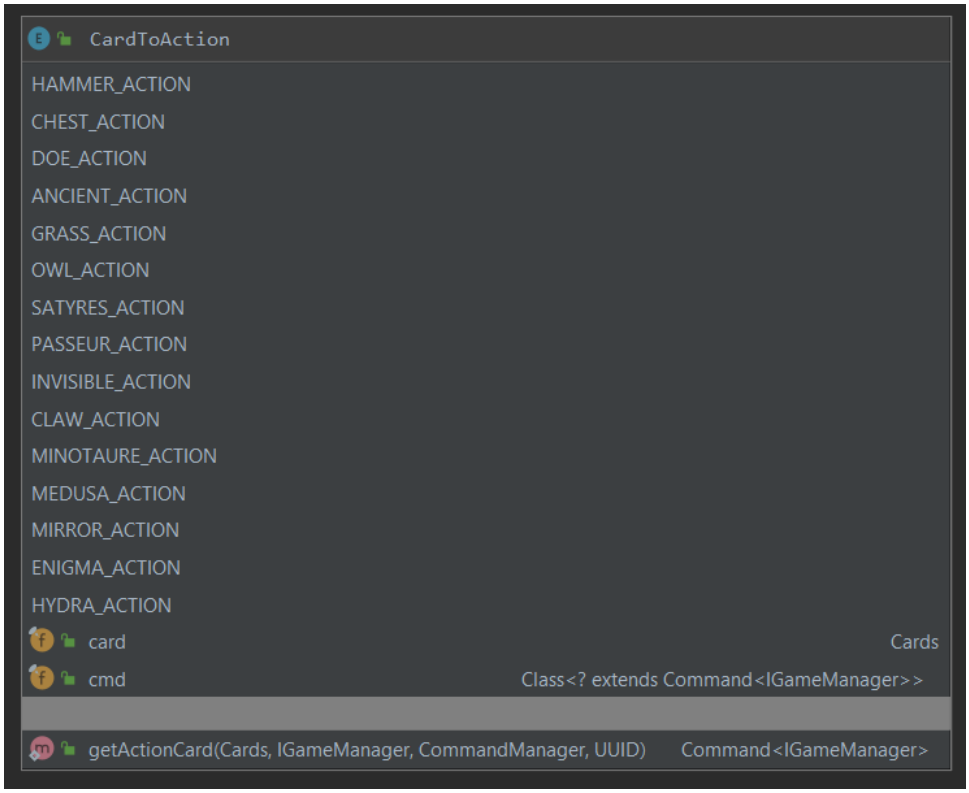
l'exécution du thread de la partie tant qu'elle n'est pas notifiée du choix client. Il existe aussi des commandes liées aux cartes, aux faces, au mécanisme de lancer de dés.



Le système d'héritage de commande est en réalité un design pattern Décorateur. Il attache dynamiquement des responsabilités supplémentaires à un objet. Les décorateurs fournissent alors une alternative souple à la dérivation, pour pouvoir étendre les fonctionnalités d'une façon transparente, c'est-à-dire sans affecter les autres objets.

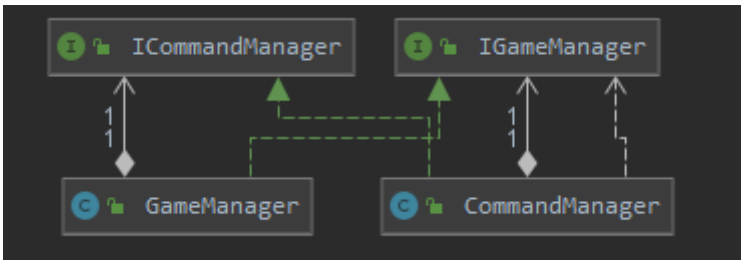
Quant aux commandes des cartes, elles sont créées dynamiquement via l'énumération CardToAction qui permet de lier une commande spécifique à une carte d'énumération de share. Il permettra d'instancier la commande à partir d'un objet Class.


Command
1
CommandGameManager
CommandCard
CommandCardChoice
Chest
Ancient
1
CardToAction

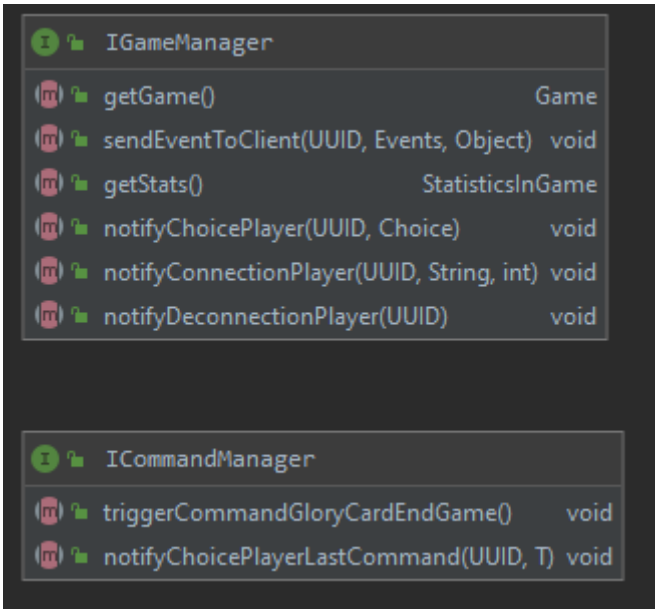

CardToAction
HAMMER_ACTION
CHEST_ACTION
DOE_ACTION
ANCIENT_ACTION
GRASS_ACTION
OWL_ACTION
SATYRES_ACTION
PASSEUR_ACTION
INVISIBLE_ACTION
CLAW_ACTION
MINOTAURE_ACTION
MEDUSA_ACTION
MIRROR_ACTION
ENIGMA_ACTION
HYDRA_ACTION
card
Cards
cmd
Class<? extends Command<IGameManager>>
getActionCard(Cards, IGameManager, CommandManager, UUID)
Command<IGameManager>

Interconnexion entre GameManager et Command Manager

Le GameManager et le CommandManager sont interconnectés via des interfaces pour casser la dépendance entre eux.

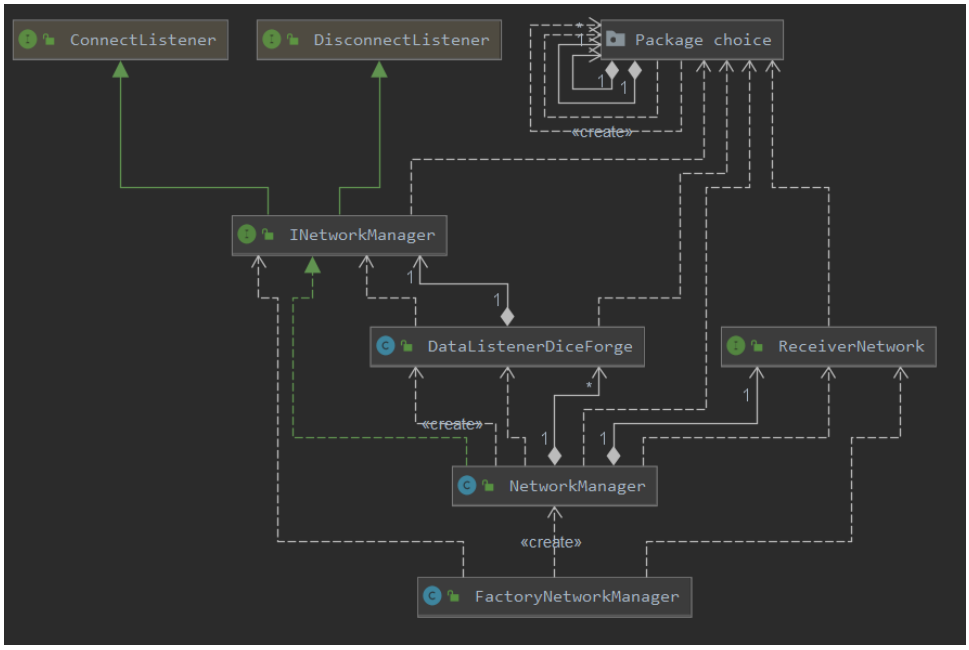


IGameManager permet au système d'envoyer des évents aux clients. L'interface ICommandManager permet au GameManager de notifier le CommandManager quand il reçoit une réponse du client.



NetworkManager

Le NetworkManager à pour responsabilité de notifier le serveur lorsqu'il y a une connexion, une déconnexion ou réception d'un message du client. Il permet aussi d'envoyer des messages aux client.

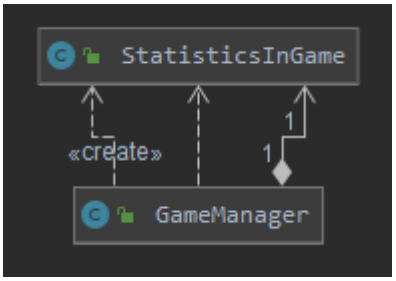


Pour cela le serveur implémente l'interface ReceiverNetwork pour pouvoir être notifié d'un event par le NetWorkManager. Quant à lui le NetWorkManager implémente de INetworkManager pour pouvoir envoyer des messages aux clients depuis le serveur.

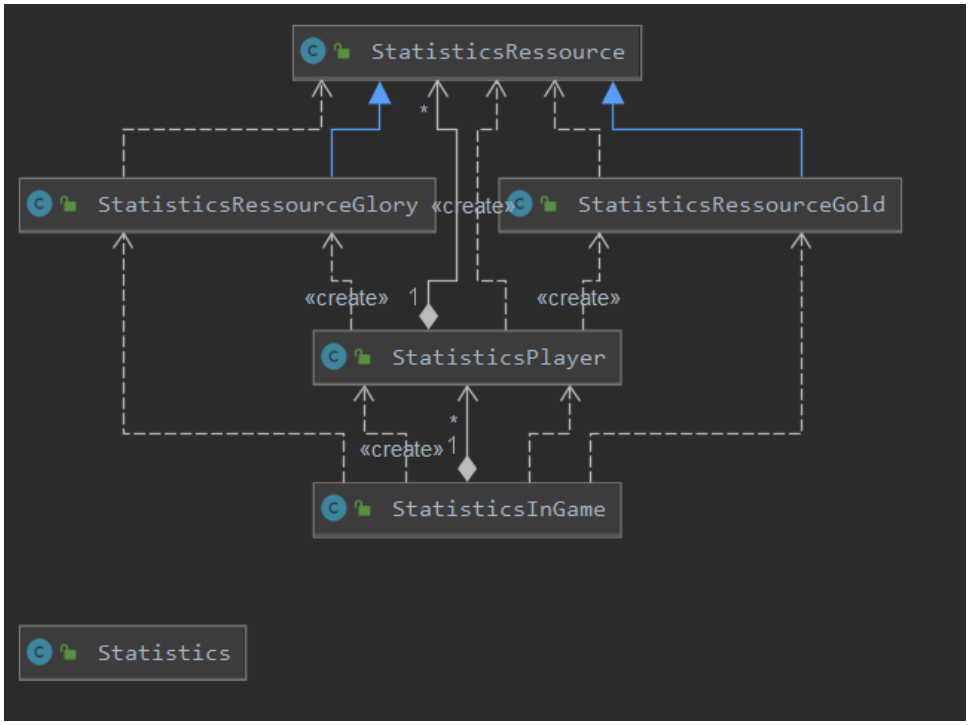
De plus le NetworkManager gère la réception des messages via plusieurs DataListeners qui ont pour rôle de sérialiser le bon objet reçu. Il y a alors un lien entre DataListeners et les choix.

Statistiques

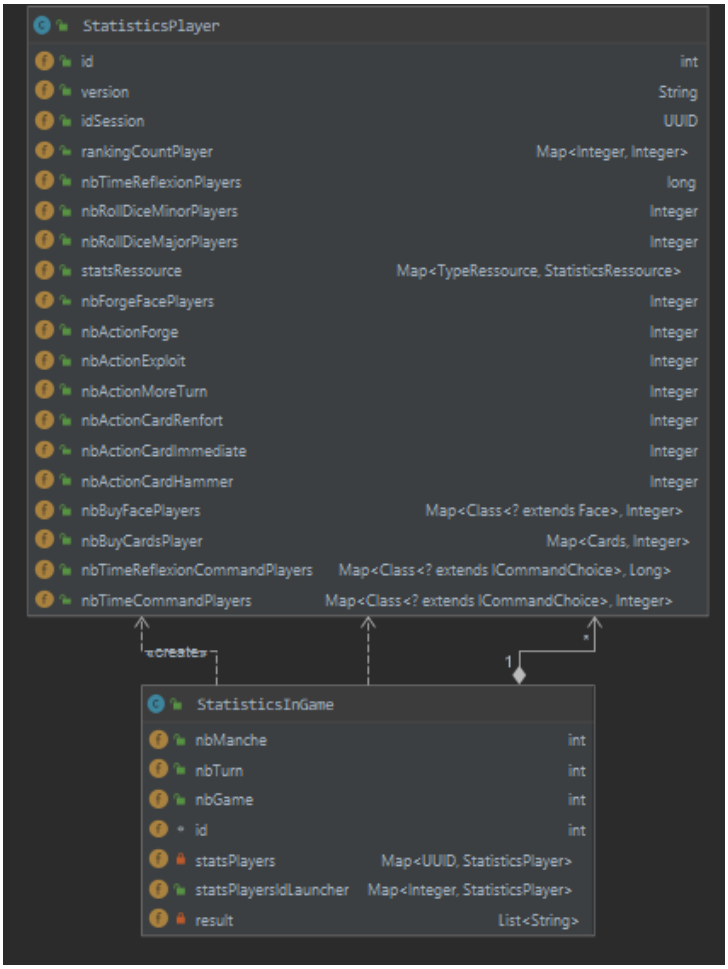
L'objet contenant les statistiques est contenu dans le GameManager.



L'objet StatitisticsInGame est composé d'un dictionnaire entre l'identificateur des joueurs et des statistiques player. StatisticsPlayer est composé de plusieurs objets permettant de faire des statistiques sur les ressources.

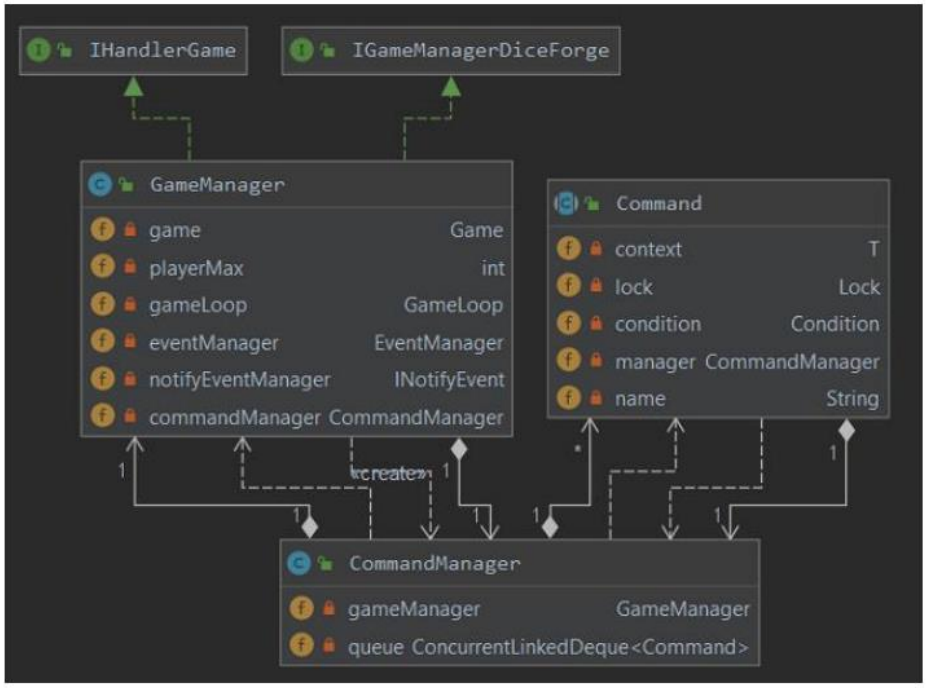


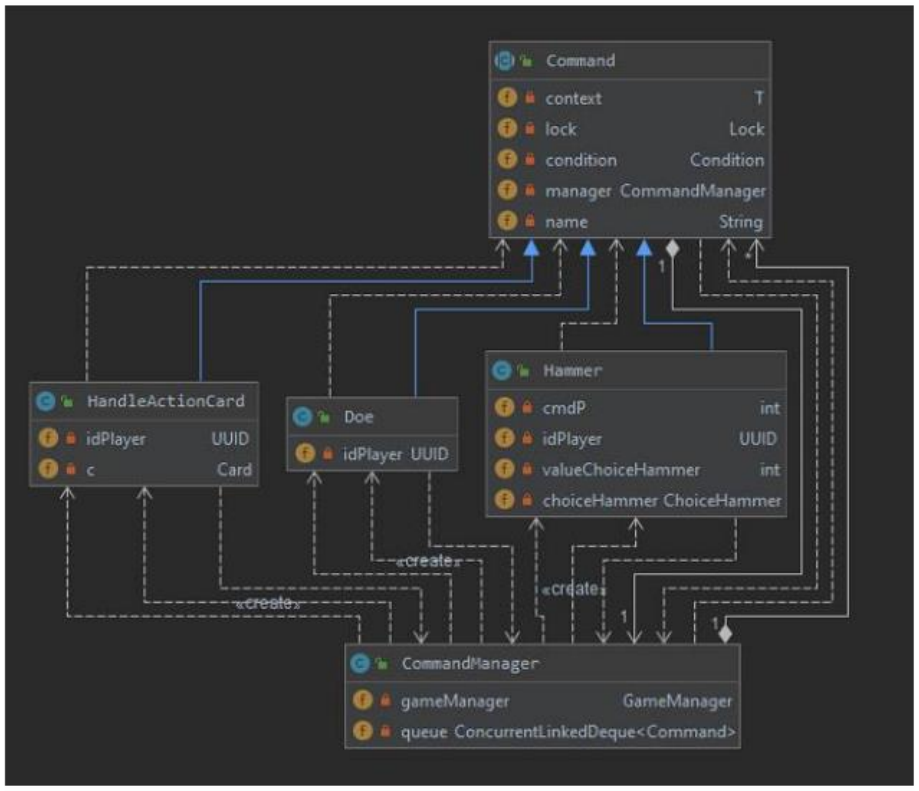
D’autres statistiques :



Point de vue statique delta itération 5

Rappel itération 5 schémas :



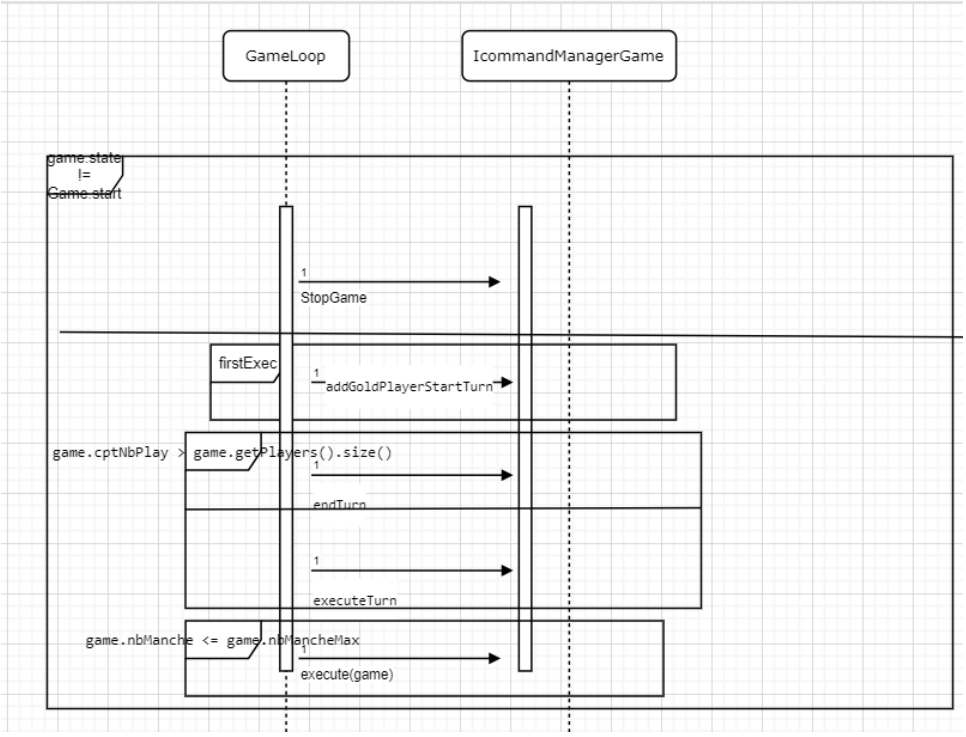


Le système général de commande à était plus approfondis pour gérer plusieurs types de commandes, les distinguées entre elles.

Point de vue dynamique itération Finale

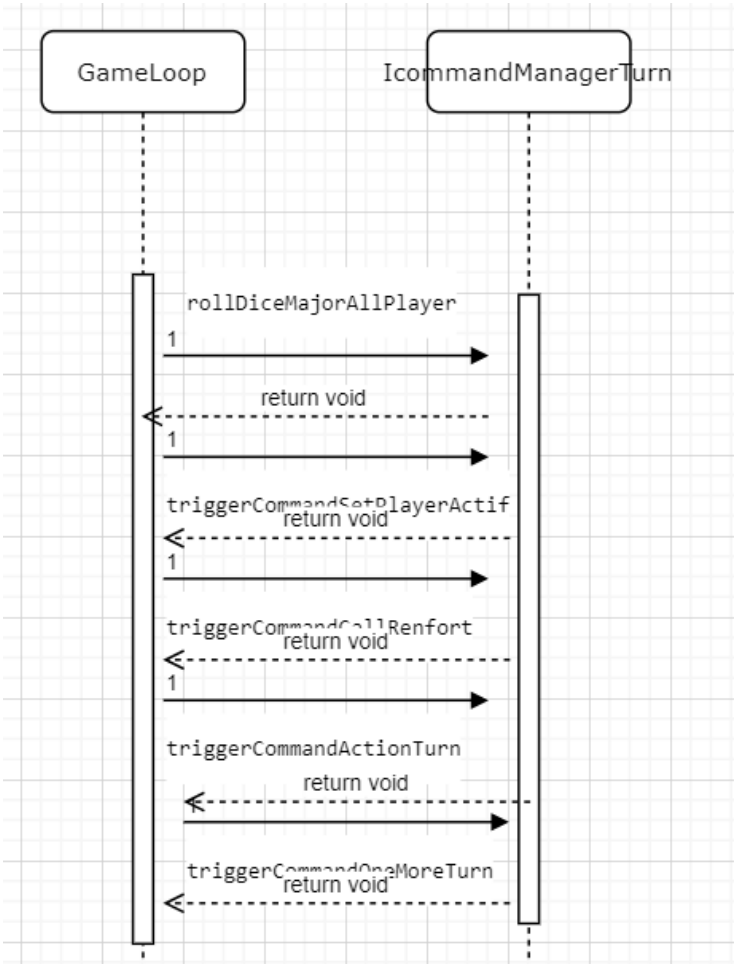
Dans cette partie nous ne détaillerons pas tous le fonctionnement des uses cases, vu le nombre de use cases que doit traiter le serveur. Nous nous concentrons donc sur les parties essentielles du serveur c'est à dire l'exécution de la partie, le système de trigger de command pour effectuer les différents traitements de la partie et enfin la technique pour qu'une commande puisse demander le choix à une client. Les cas réseaux plus détaillés sont spécifiés dans la partie réseau.

Game Loop



Le gameLoop a pour but d'exécuter la logique de la partie. En fonction de certaines conditions, le gameLoop va choisir s'il doit arrêter la partie ou non, s'il doit exécutera la fin du tour mais aussi s'il doit exécuter le tour et enfin réexécuter la manche. Ci-dessus nous voyons alors le fonctionnement d'une manche. À la fin il regarde si le nombre de manche est égale au nombre de manche max, s'il est inférieur et s'il réexécute une manche. Il s'agit d'une méthode récursive.

Ci-dessous nous pouvons voir l'exécution d'un tour.



L'exécution du tour consiste seulement à trigger des commandes en fonction de ce que le jeu doit être capable de faire dans un tour via le ICommandManagerTurn qui est dans notre cas le CommandManager.

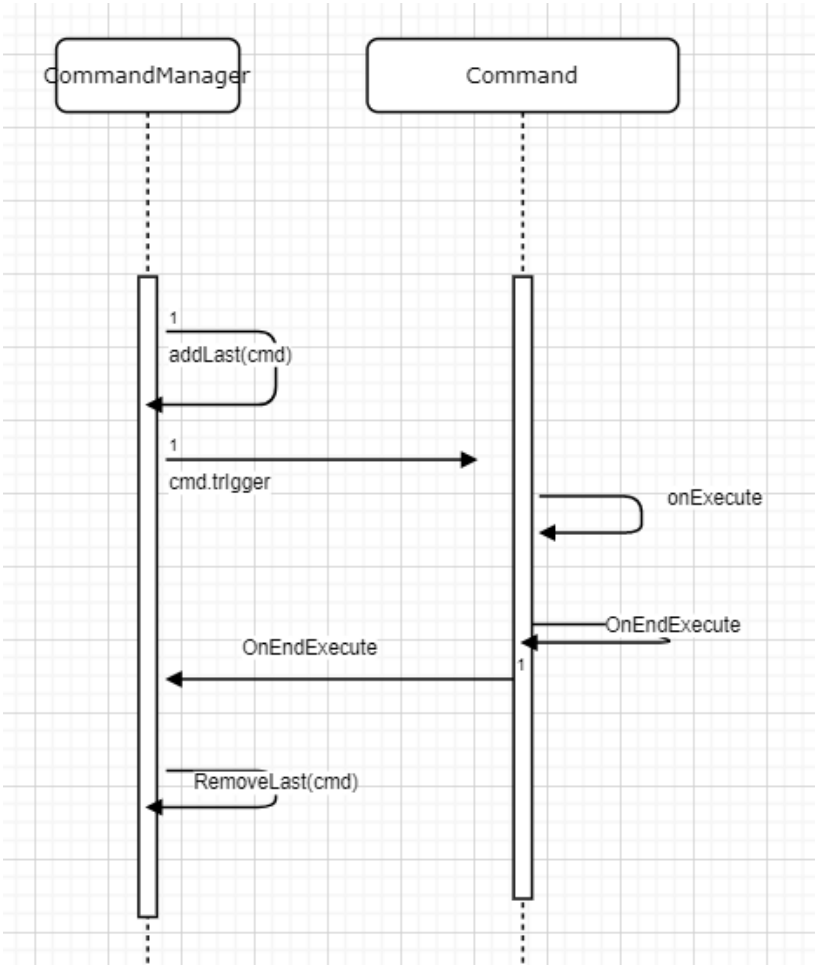
Nous détaillons alors maintenant le système de commande.

Système de trigger de commande

Le CommandManager est composé d'une queue de commande comme nous l'avons vu antérieurement.

Le principe pour exécuter une commande est de l'ajouter à sa queue, et de la trigger. Quand une commande est trigger, elle va faire appel à sa méthode onExecute et à la fin de son exécution la méthode onEndExecute. OnEndExecute notifie au command manager que la

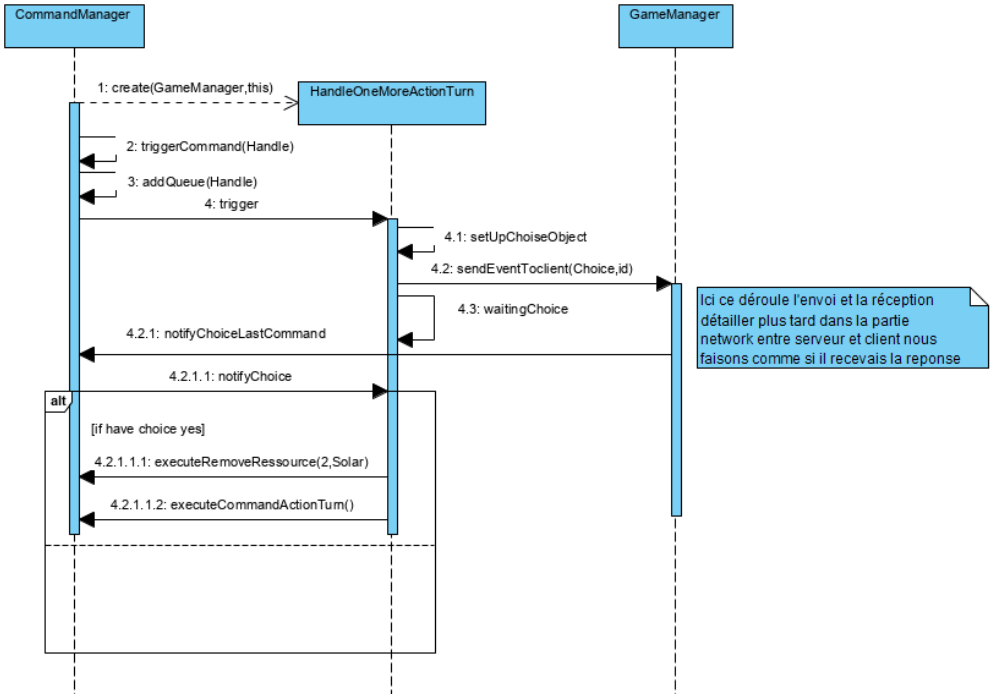
commande est finie pour l'enlever de la queue. Cela nous permet de garder une trace des enchainements des commandes.



Les commandes ont aussi un système de choix pour interagir avec le client.

Système attente de choix client

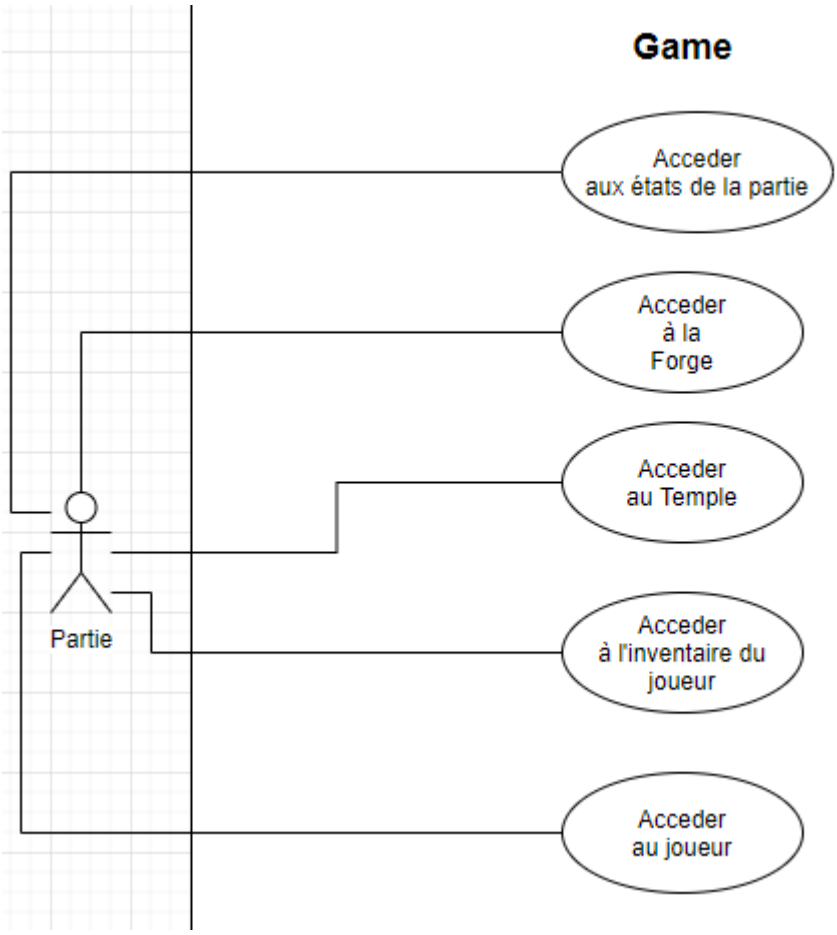
Nous allons prendre un exemple simple, les autres commandes choice reposent sur le même système : CommandChoiceMoreTurn :



Nous reprenons alors le système de trigger dans un cas concret. La précondition demande au CommandManager de trigger cette commande. Il la crée et l'ajoute à la queue puis le trigger. Une fois trigger, la commande exécute son traitement c'est à dire demander au joueur s'il veut effectuer une action supplémentaire dans le tour pour 2 Solar. Elle setup alors un objet choice associé à la commande puis l'envoi via le contexte de la commande qui est en fait le GameManager comme nous l'avons vu plus haut. Ensuite la commande bloque l'exécution du thread en cour pour attendre le choix client. Une fois reçu le GameManager notifie le CommandManager qu'il a reçu une réponse. Le CommandManager notifie la dernière commande exécuter de la réponse du client. Cela réveille via le système de Lock Condition la commande pour qu'elle puisse continuer son exécution en ayant le choix du client.

Module Share

Diagrammes Use Case



User Story

En tant que partie, je souhaite accéder aux informations et aux actions effectuées dans la forge

En tant que partie, je souhaite accéder aux informations et aux actions effectuées dans le temple

En tant que partie, je souhaite accéder aux informations et aux actions effectuées par un joueur

En tant que partie, je souhaite accéder aux informations et aux actions effectuées dans l'inventaire d'un joueur

En tant que partie, je souhaite accéder à mes états

Scénario

La partie est la classe la plus importante du module share, c'est elle qui permet d'accéder à toutes les actions effectuées par les joueurs ou pour les joueurs.

Précondition :

Une action a été effectuée

Postcondition :

Nous pouvons accéder à la forge :

- Nous pouvons accéder aux pools puis aux faces qu'elles contiennent.
- Nous pouvons connaitre le prix d'un pool

Nous pouvons accéder au temple :

- Nous pouvons accéder aux iles puis aux cartes qu'elles contiennent.
- Nous pouvons savoir si un joueur est présent sur une ile
- Nous pouvons ajouter un joueur sur une ile

Nous pouvons accéder aux informations du joueur et aux actions effectuées par le joueur et sur le joueur :

- Nous pouvons connaitre le nombre de joueur ainsi que leur identifiant
- Nous pouvons ajouter ou retirer un joueur dans une partie

Nous pouvons accéder à l'inventaire d'un joueur :

- Nous pouvons connaitre les informations, ajouter ou retirer des ressources
- Nous pouvons connaitre les informations et ajouter des cartes
- Nous pouvons connaitre les informations, ajouter ou retirer les faces des dés du joueur
- Nous pouvons sauvegarder les anciennes faces des dés du joueur

Nous pouvons connaitre les états de la partie :

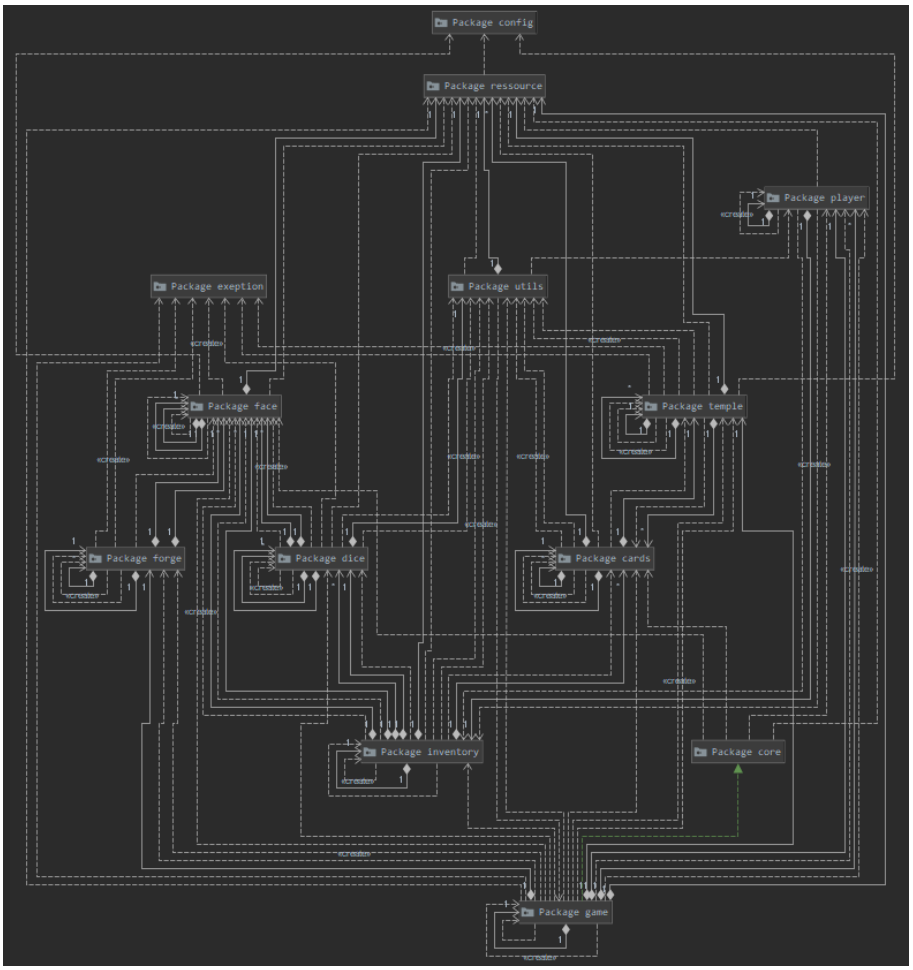
- Initialisation
- Terminaison
- Réinitialisation

Conception Logicielle

Point de vue statique

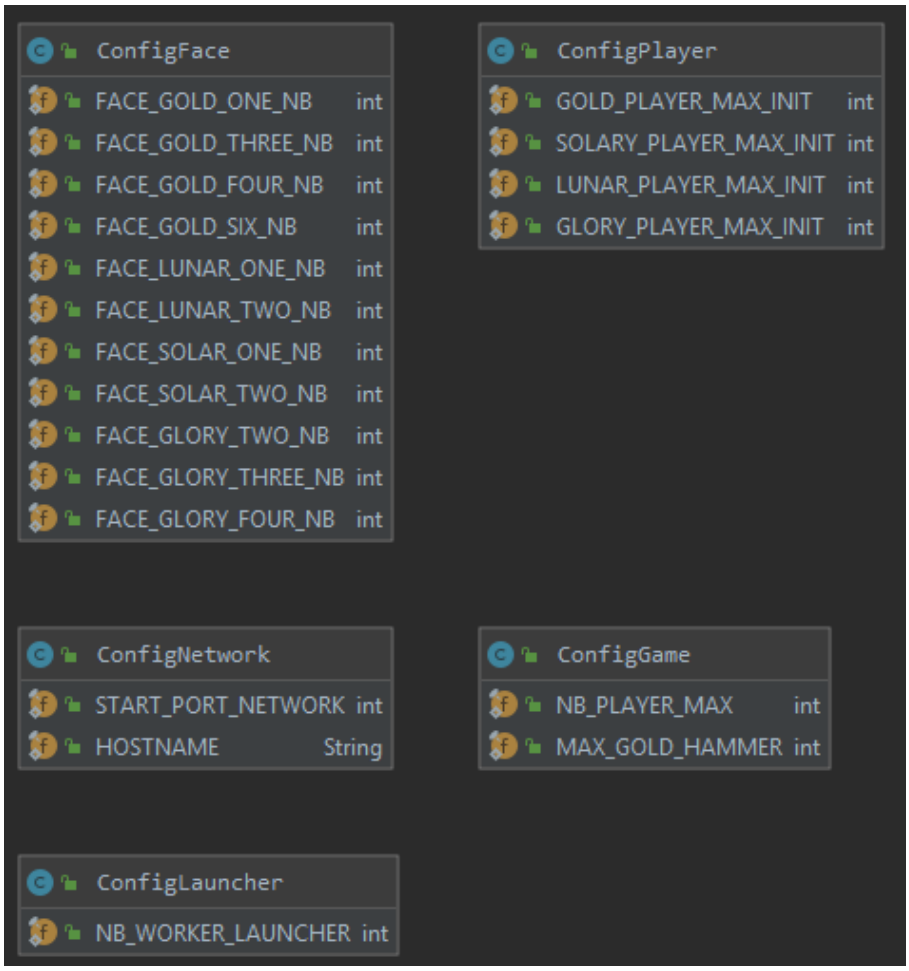
1. Représentation générale du package share

https://zupimages.net/viewer.php?id=20/16/r0p4.png



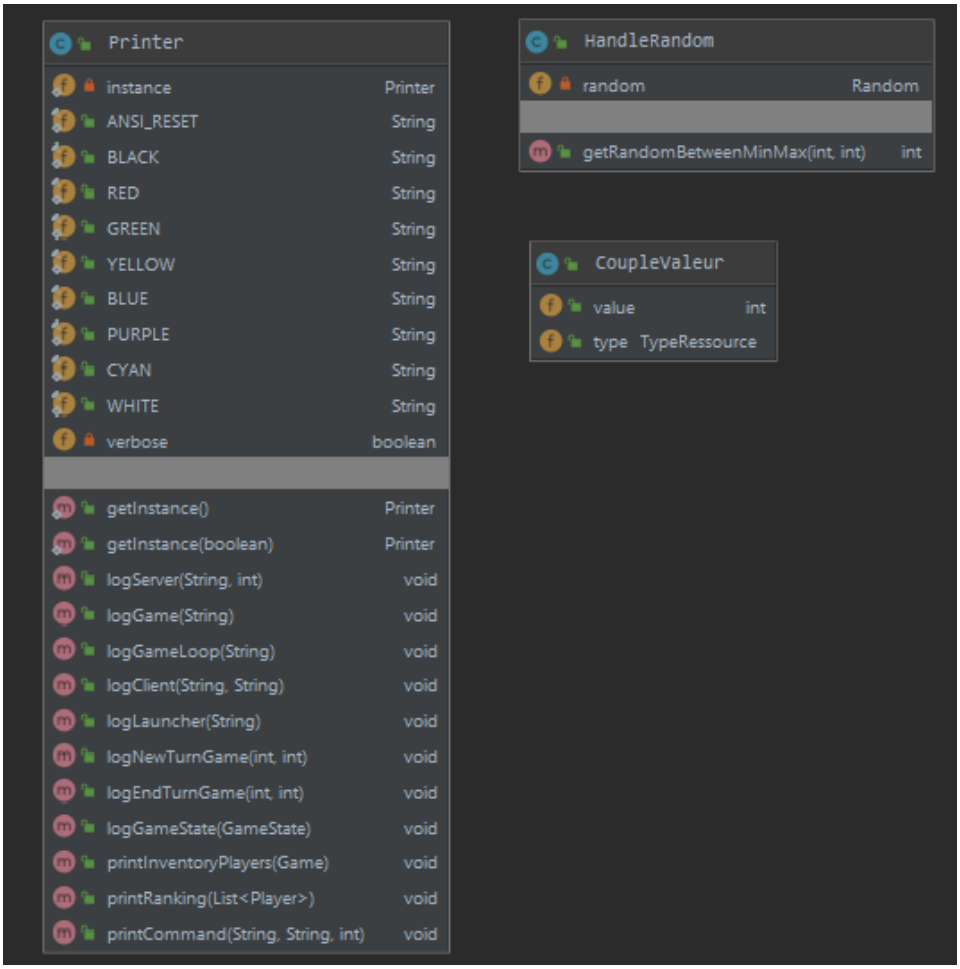
2. Quelques fonctionnalités

Config



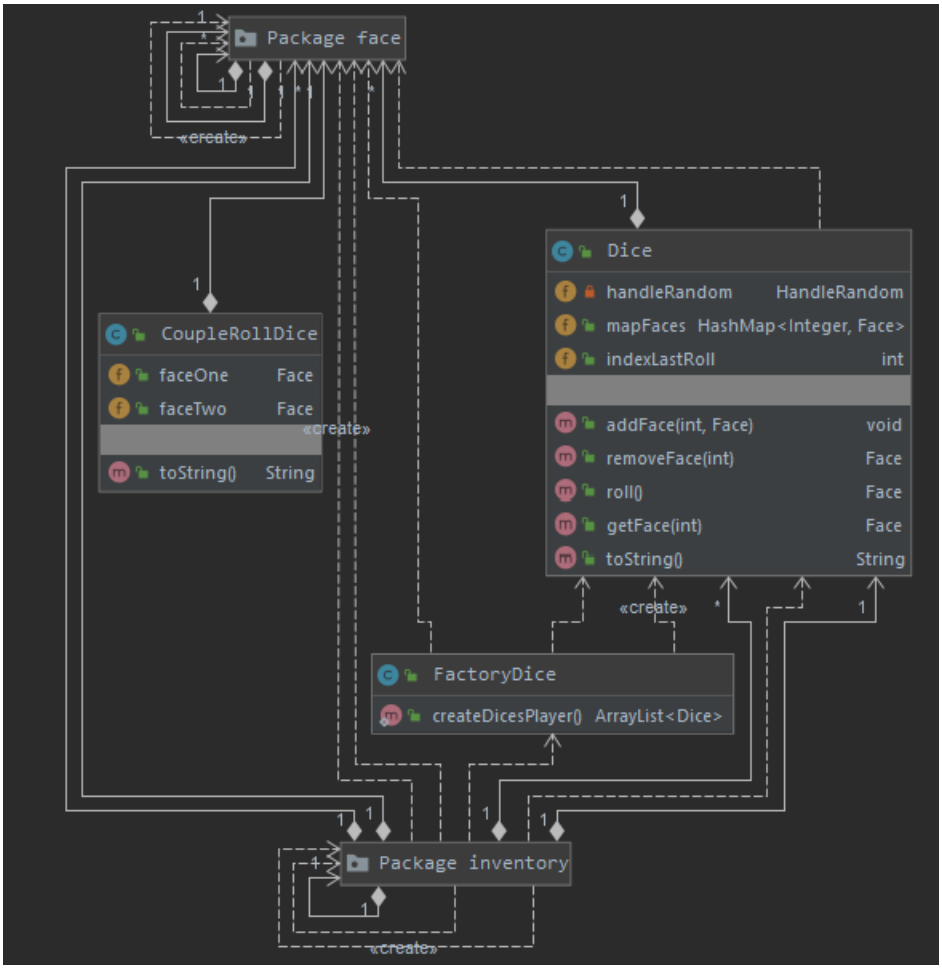
Config contient les différentes faces possibles, le nombre maximal de joueurs d'une partie ainsi que le nombre maximal de ressources (gold, Lunar, Solar et glorypoint). Enfin Config contient le port et le nom de l'hôte du réseau, le nombre de launcher ainsi que le nombre maximal de gold pour la carte du marteau.

Utils



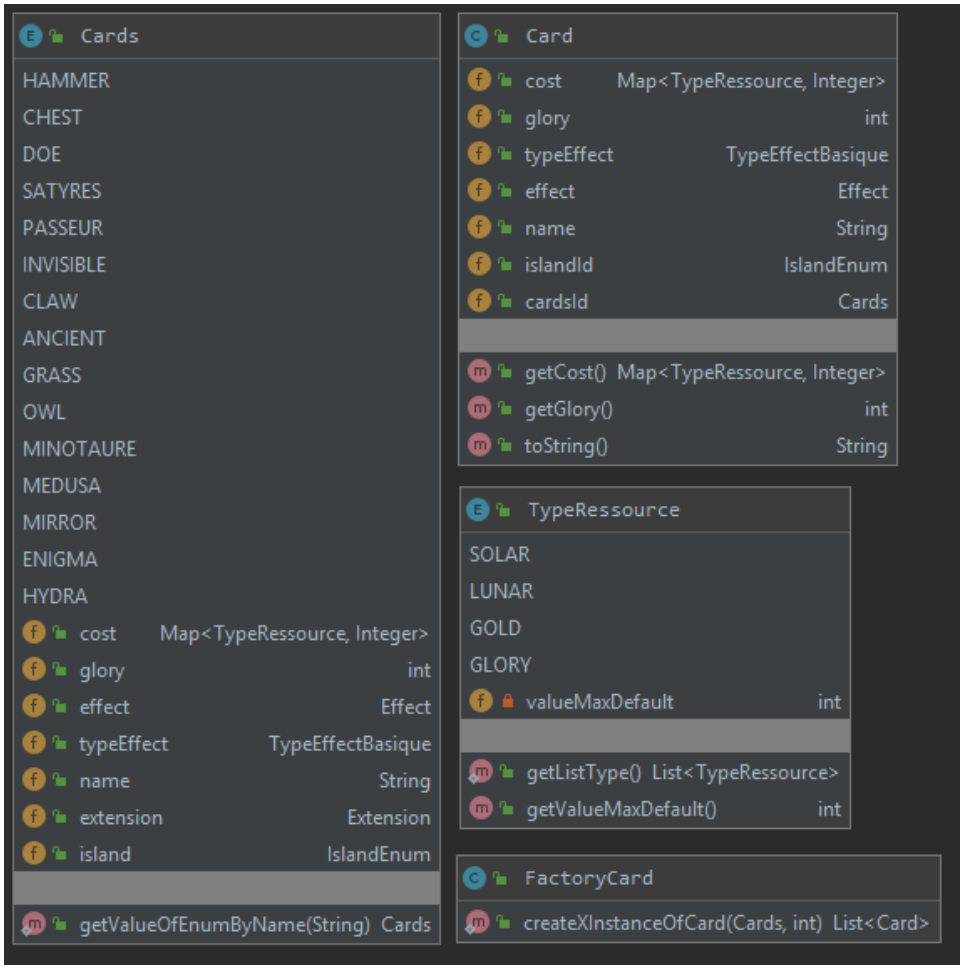
Utils contient plusieurs classes dites “utiles” pour certaines fonctionnalités. Par exemple l’utilisation du random pour le lancement des dès ou autres actions aléatoires. Pour gérer des ressources et principalement pour rendre le déroulement d’une partie plus lisible avec l’utilisation de différentes couleurs pour afficher plus facilement les actions effectuées durant une partie.

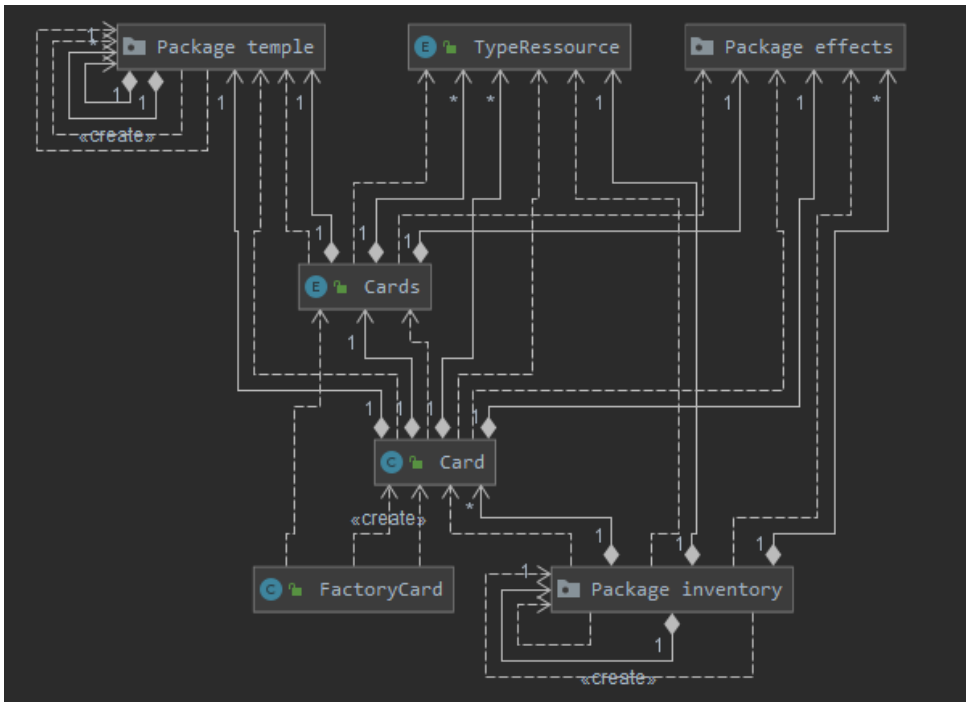
3. Gestion des dès



Notre classe FactoryDice permet de créer les 2 dés d'un joueur. Les dés sont composés de plusieurs faces et une face peut composer plusieurs dés. On utilise CoupleRollDice pour avoir les faces des deux dés. Lorsqu'on lance un dé (on le roll) on fait appel à la fonction handleRandom crée dans le package Utils.
Enfin, ces deux dés sont obligatoirement rangés dans un inventaire qui est prêt à les accueillir.

4. Gestion des cartes

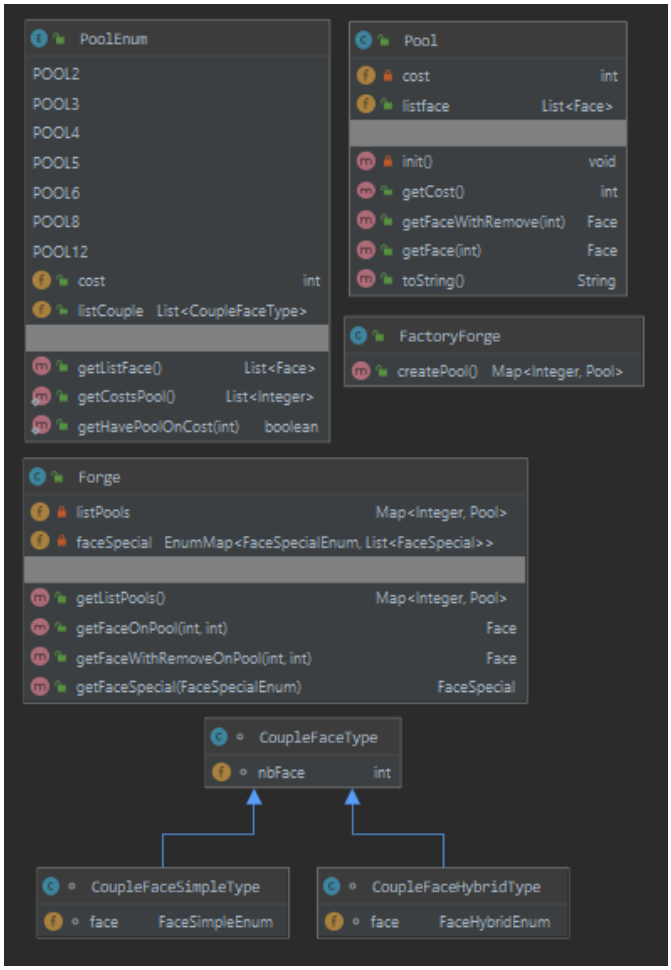


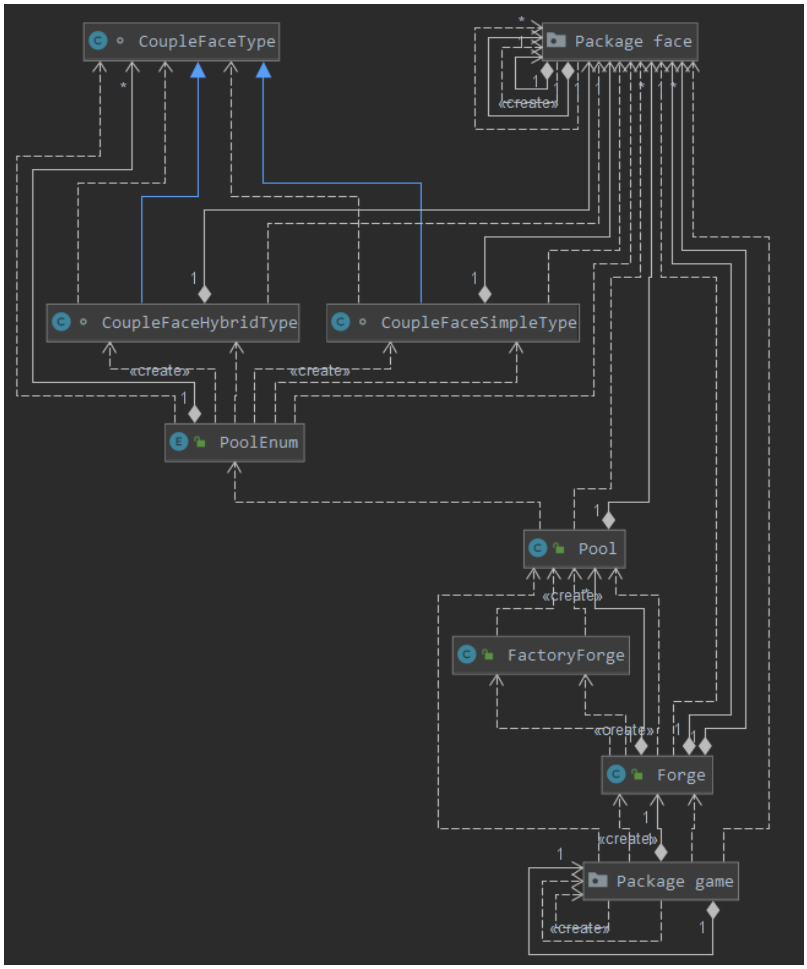


FactoryCard permet de créer les cartes. Les cartes ont chacune leurs propres effets, un type d'effet (immédiat ou chaque tour) et ont un cout qui utilise des ressources. Nous avons une

classe Cards qui énumère toutes les cartes du jeu avec toutes leurs informations.
Enfin, une carte peut se trouver dans une ile (package temple) ou dans un inventaire d'un joueur.

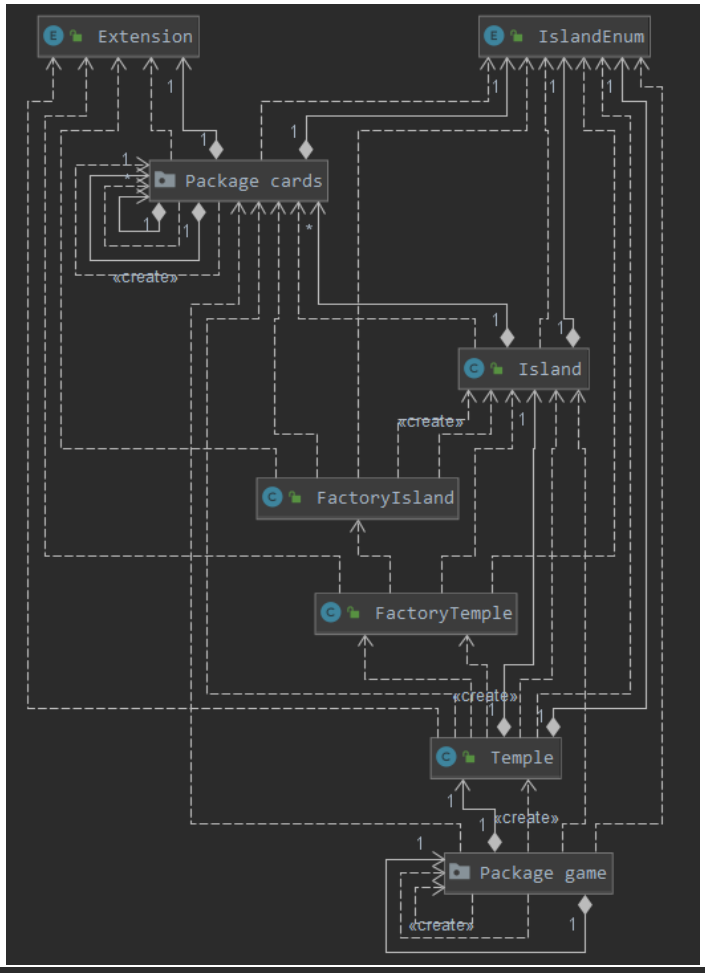
5. Gestion de la forge

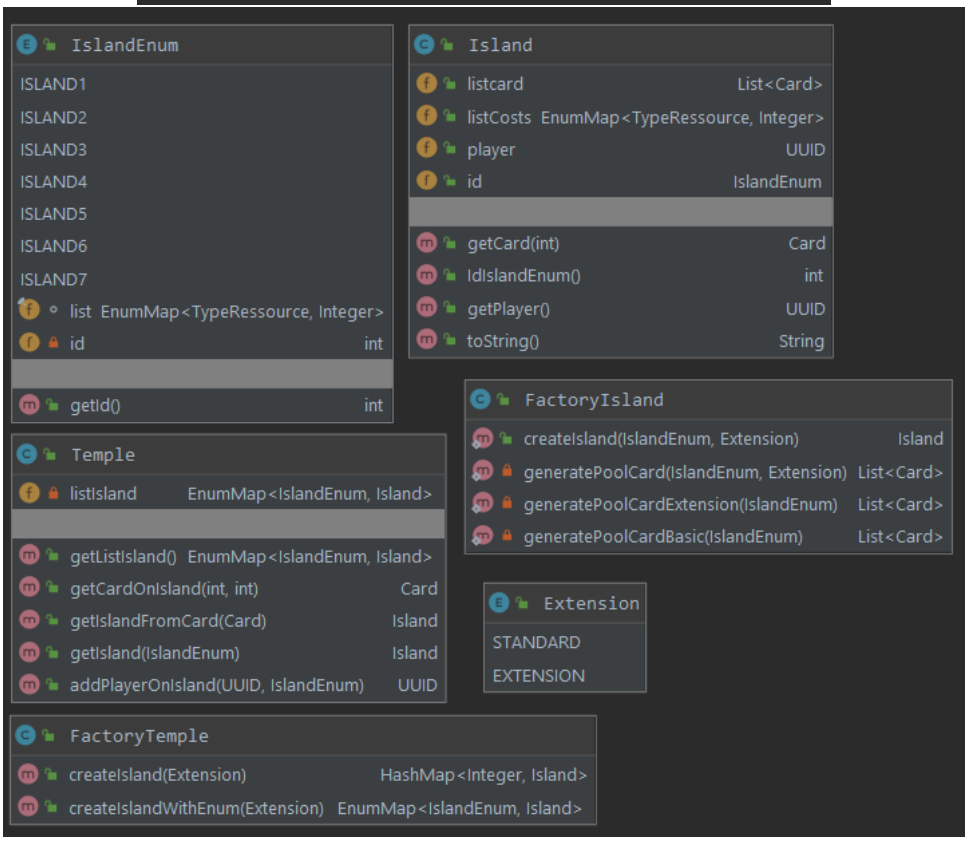




FactoryForge permet de créer notre forge. La forge est composée de 7 pools qui ont chacune un cout et sont constituées de différentes faces qui peuvent être de type simple ou de type hybride. CoupleFace crée les couples de cartes dans les pools.
Il n'y a qu'une forge dans une partie et une partie est composée d'une seule forge.
Enfin, les différentes actions effectuées dans la forge se trouvent dans le package Game.

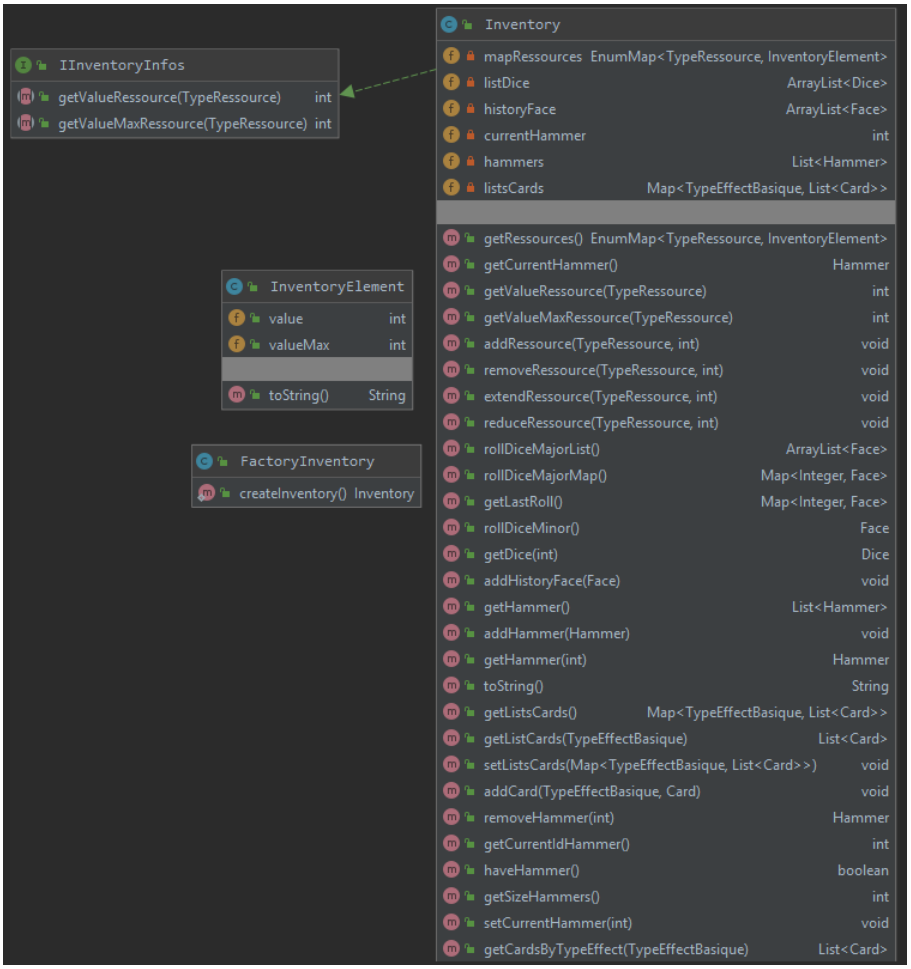
6. Gestion du temple

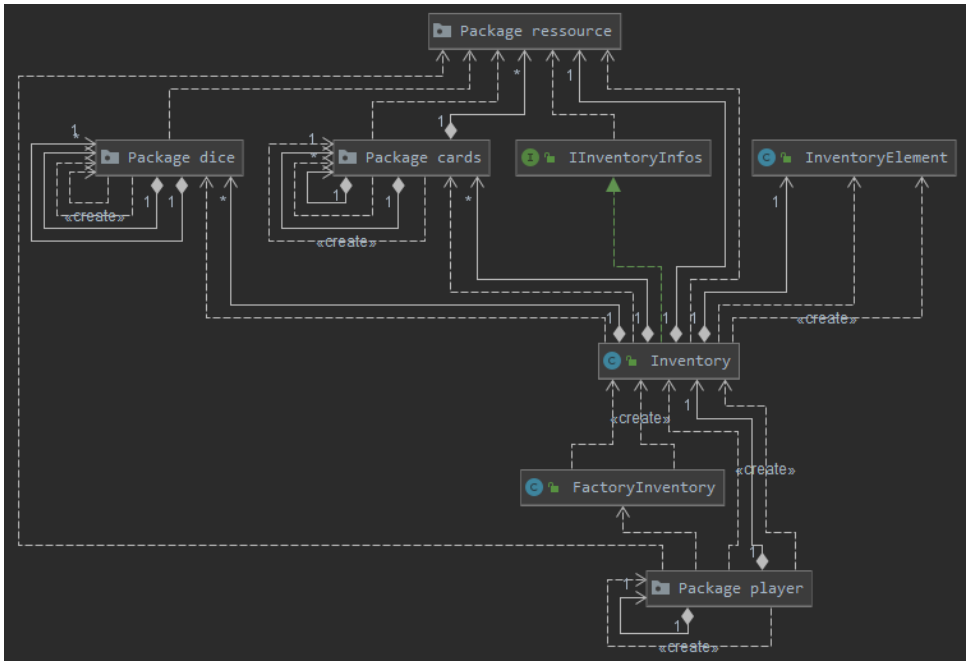




FactoryTemple permet de créer le temple qui va contenir des islands qui sont créés par FactoryIsland. En effet, le temple est composé de plusieurs islands (on en compte 7) qui ont un cout un slot pour les joueurs et qui contiennent différentes cartes qui sont soit Standard soit issues d'une extension. Nous avons islandEnum qui énumère la liste des iles et ceux qu'elles contiennent. Une fois encore, les actions effectuées dans le temple sont gérées dans le package game et enfin le temple fait partie d'une partie et une partie ne peut posséder qu'un seul temple.

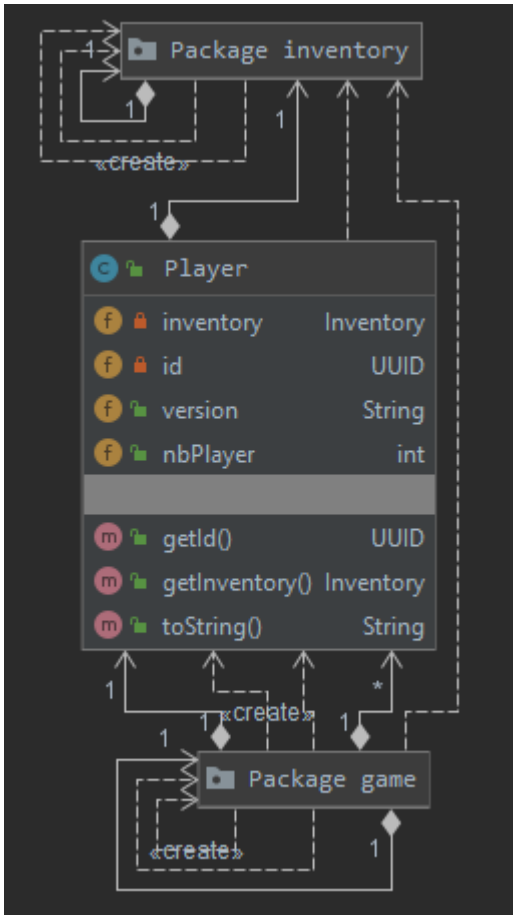
7. Gestion de l'inventaire





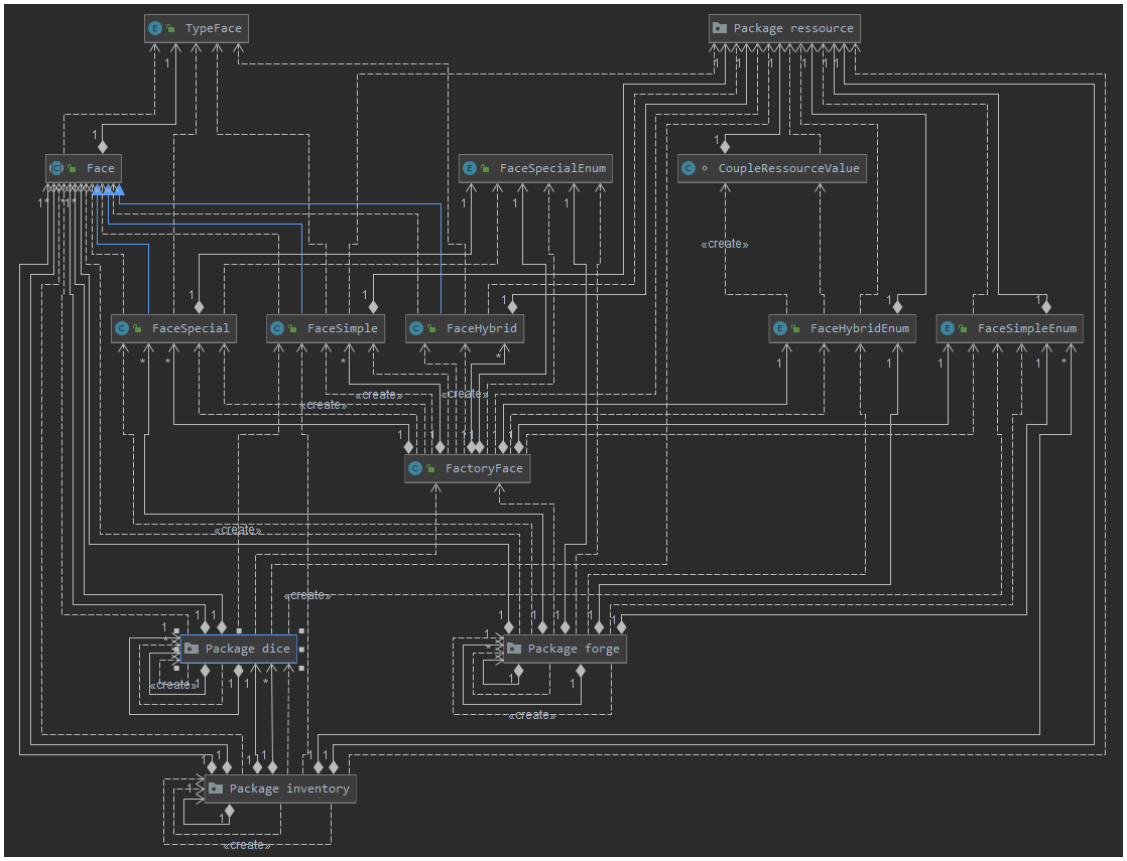
FactoryInventory crée l'inventaire du joueur car l'inventaire est obligatoirement relié à un joueur. L'inventaire possède plusieurs types de ressources qui ont une valeur courante et une valeur maximale, il est ensuite composé de 2 dés et peut contenir les cartes choisies par le joueur dans le temple.

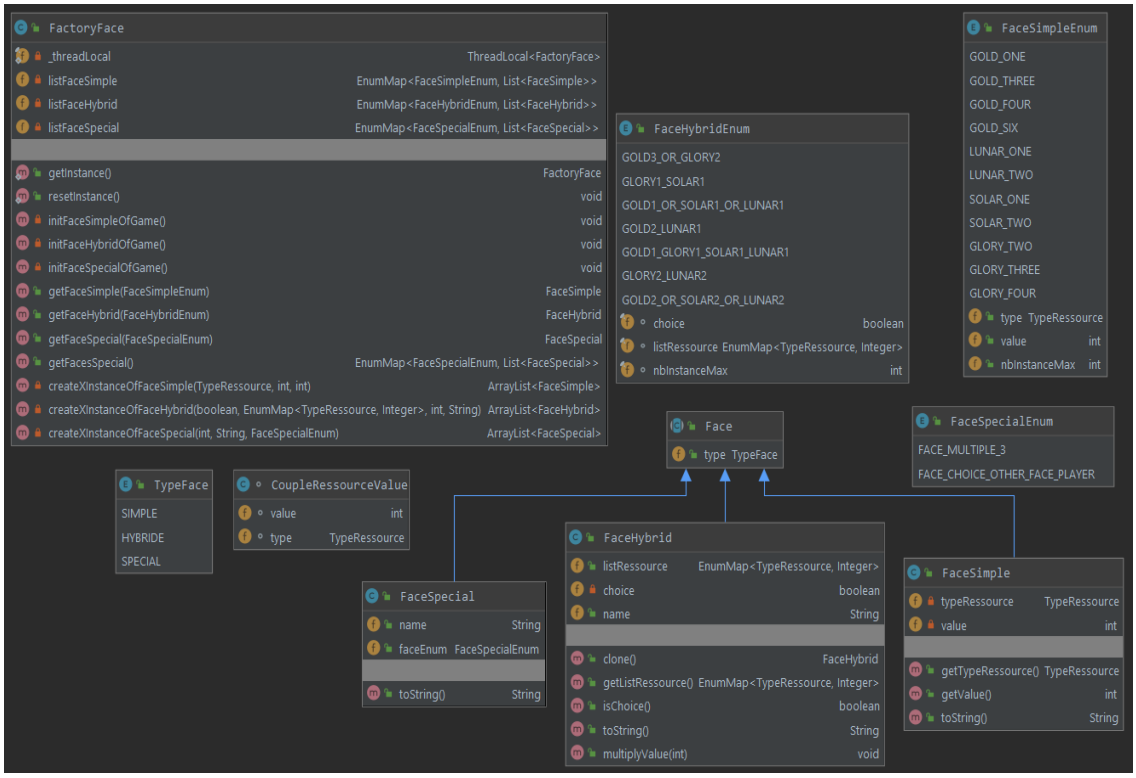
8. Gestion du joueur



Un joueur possède un id pour se différencier des autres et un seul inventaire. De plus, Un joueur est connecté à une seule partie qui peut accueillir plusieurs joueurs. Les actions effectuées par le joueur sont gérées dans le package game.

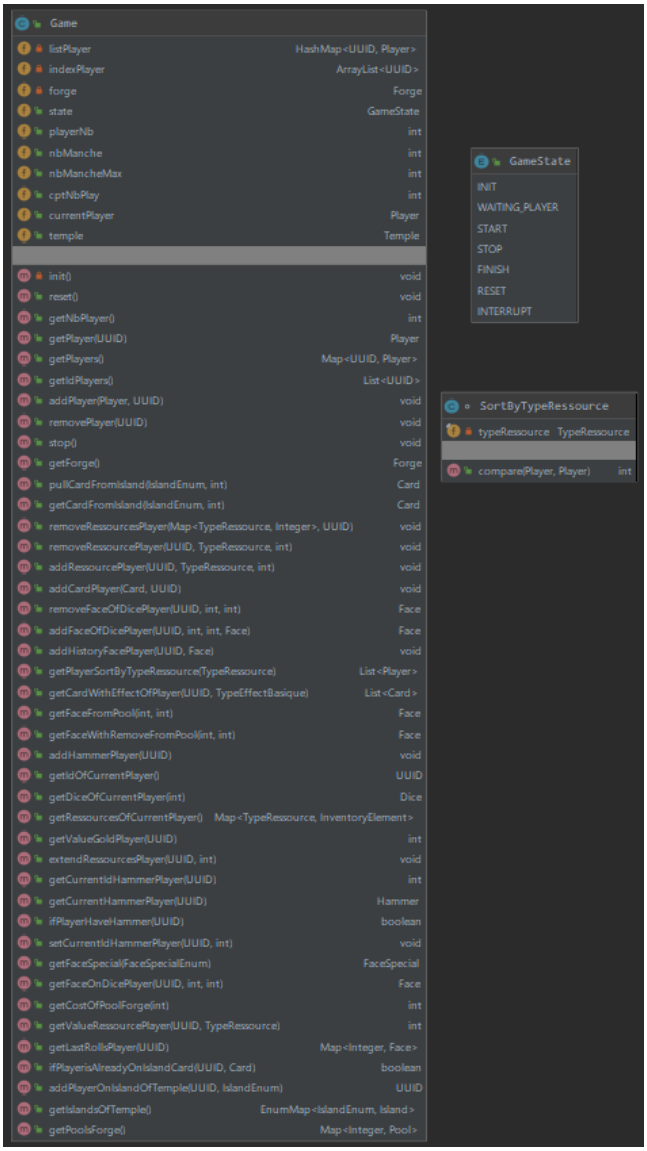
9. Gestion des faces

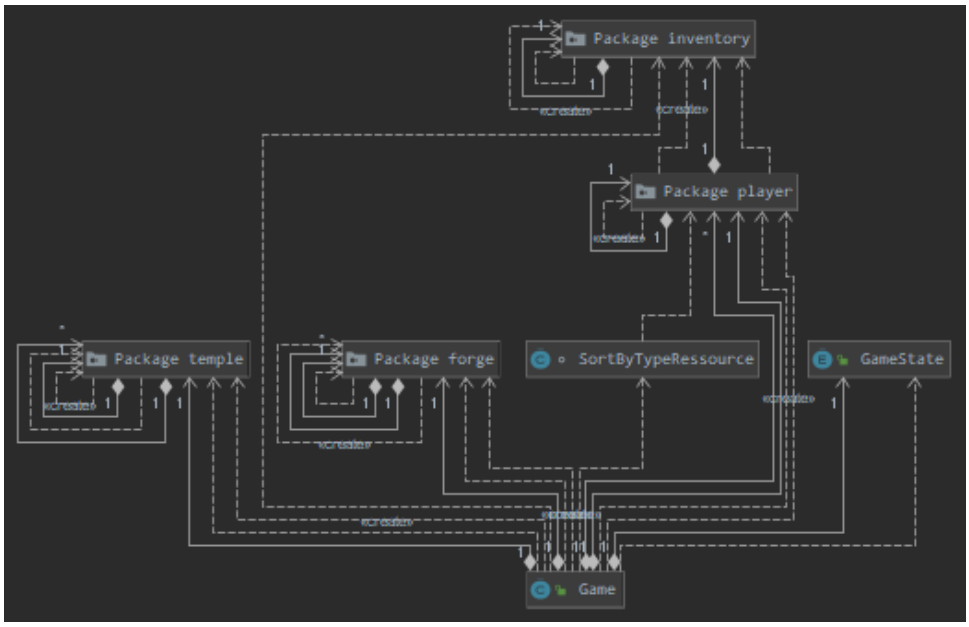




FactoryFace crée les faces qui peuvent être de type simple, hybride et spéciale.
Nous avons alors 3 classes permettant d'énumérer les différentes faces appartenant à ces 3 types. Les faces peuvent être composées d'un couple valeur ou d'une liste de couple valeur. Enfin, les faces se trouvent sur les dés, dans les pools qui se trouvent dans la forge et dans l'inventaire du joueur.

10. Gestion de la game



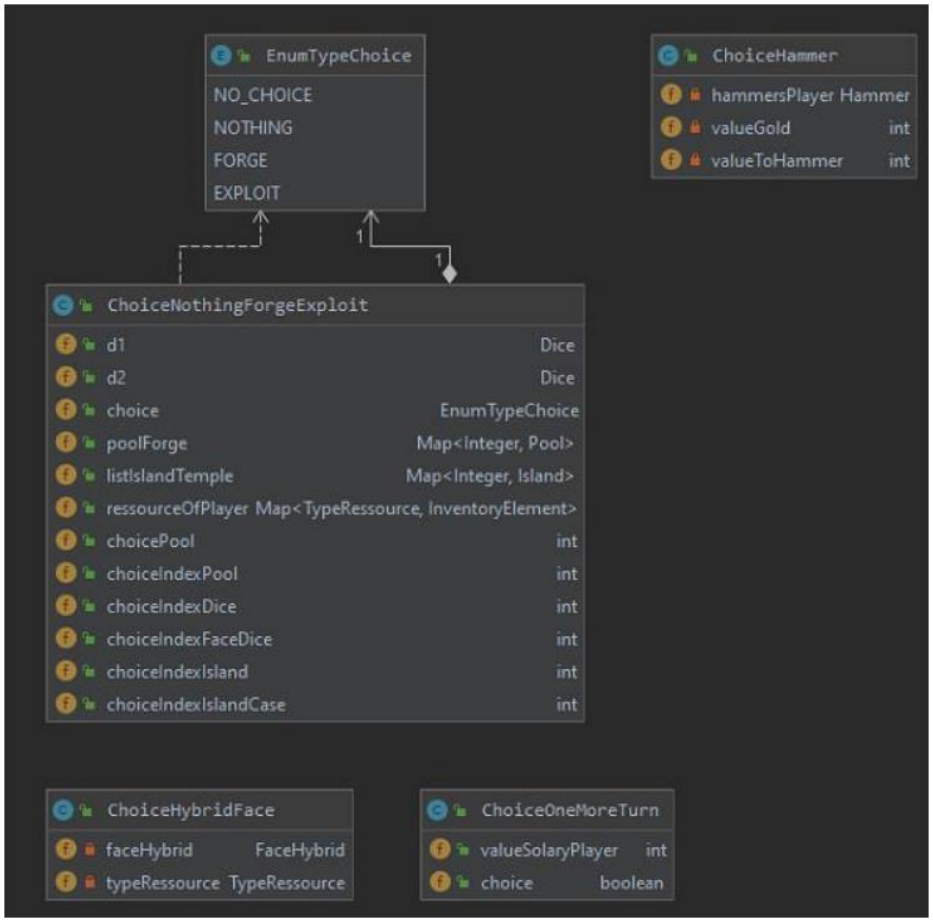


Comme expliqué antérieurement, la classe game est la plus importante car c'est elle qui a un accès à tout. En effet, c'est dans cette classe que nous pouvons connaitre les informations et les actions effectuées dans tout ce qui est nécessaire pour le déroulement du jeu. Ici, ce qui se passe dans l'inventaire d'un joueur, dans la forge, le temple... Comme vu précédemment, la partie ne peut avoir qu'une forge et qu'un temple puis elle peut avoir 1 ou plusieurs joueurs ainsi que leur inventaire et tout ce qui s'y trouve.

Interactions entre le ou les joueurs et le moteur

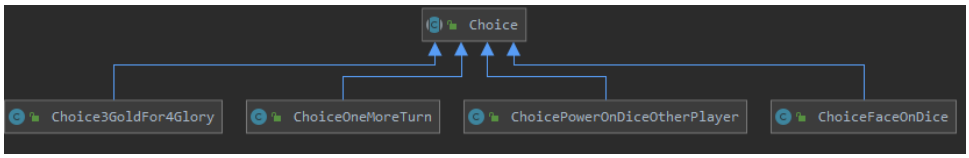
Objet réseau / protocole itération 5

Lors de l'itération 5, il y avait uniquement les choix suivants implémentés dans le jeu.

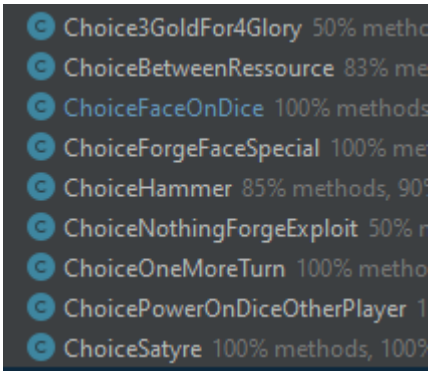


Objet réseau / protocole itération Finale

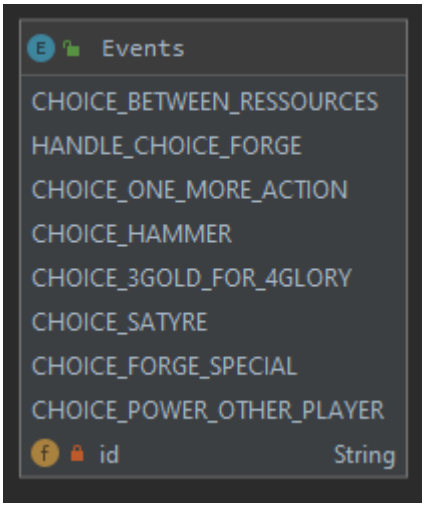
Par rapport à l'itération 5 nous avons ajouté un héritage pour les choix.



Nous utilisons les objets suivant pour la communication avec le client, et lui permettre d'effectuer des choix.

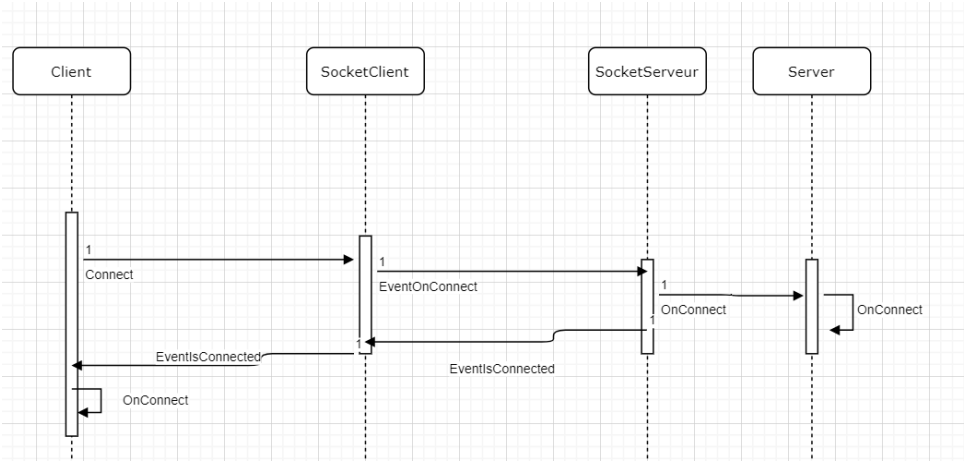


Chaque choix est indirectement associé aux évènements suivants :



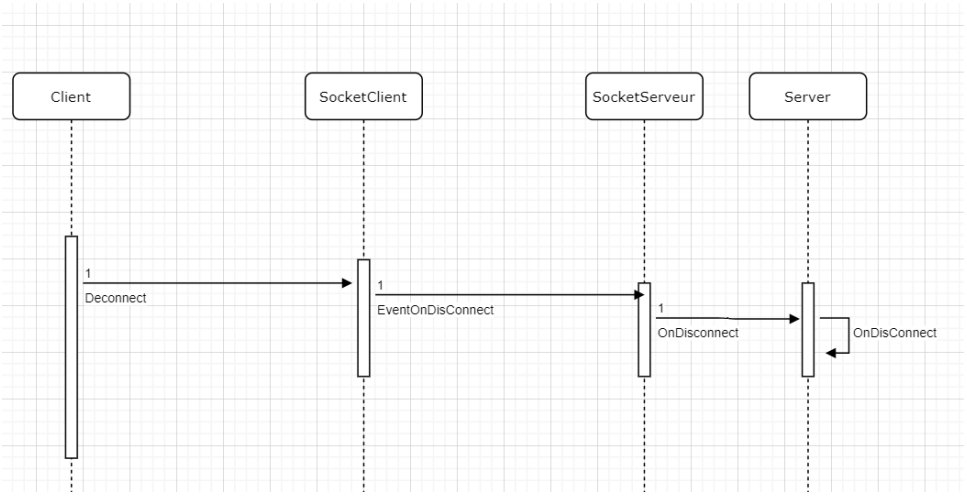
On Connect

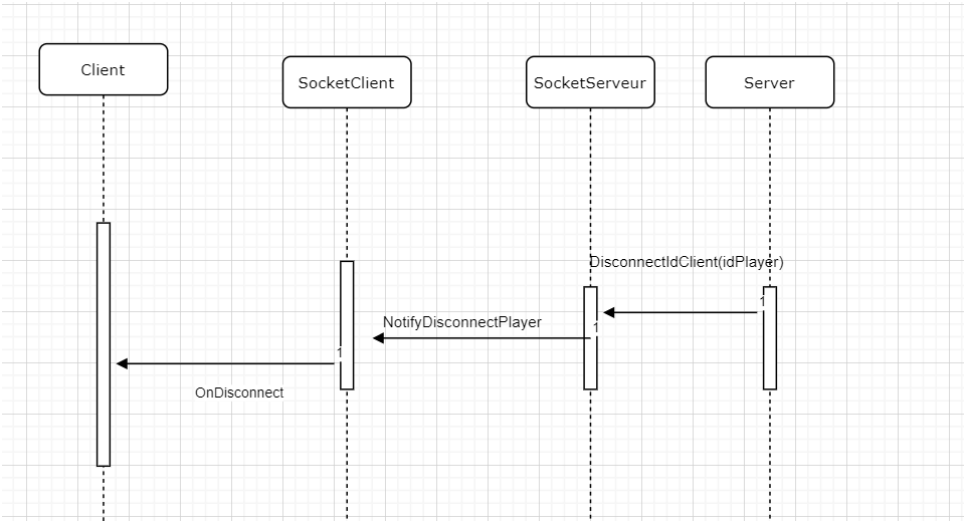
La connexion est toujours initialisée dans la partie du client. Le client déclenche la méthode connect de socket io client. La socket client notifie la socket serveur d'un event connect. La socket serveur notifie le serveur de la réception d'un event connect. En même temps il notifie le client de la bonne connexion avec le serveur.



On Disconnect

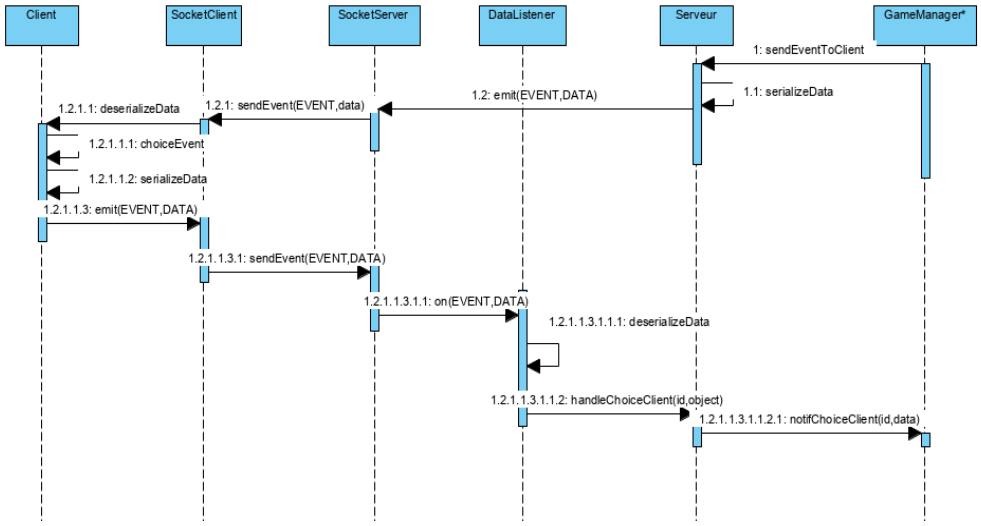
Il y a 2 façons de gérer une déconnexion : une provenant du serveur et l'autre du client :





Echange Message client serveur

Ceci est le système d'échange entre le serveur et le client.



Lorsque la partie a besoin d'un choix client le GameManager réalise une demande de message vers le serveur. Puis le serveur émet le message vers le client associé avec l'event et les données. Une fois le client ayant reçu les données il fais son choix. Puis répond au serveur, le serveur reçoit le message en fonction de l'event sur un DataListener correspondant à l'event qui servira à désérialiser le bon objet. Puis notifie le GameManager de la bonne réception du message pour qu'il puisse continuer son traitement.

Conclusion itération 5

Analyse de notre solution : points forts et points faibles

Etant assez avancé dans le projet, notre conception n'a pas beaucoup évolué depuis la dernière itération. Comme prévu, nous avons amélioré certaines fonctionnalités comme par exemple la gestion d'évènements dans la partie grâce à un système de commande. Nous avons également implémenté quelques cartes dont la carte du marteau qui, nous pensons est la plus difficile à réaliser.
Par ailleurs, notre système de commande s'est montré très efficace avec notamment un système d'attente de choix des clients et une implémentation d'un stack de commande permettant une meilleure visibilité des commandes lors du déroulement d'une partie. De plus, nous avons amélioré et optimisé le multi-partie pour gagner en performance.

Enfin, notre point faible reste toujours le même, plus il y a d'ajout de fonctionnalités et plus le travail devient difficile et également au niveau de la compréhension surtout pour une personne extérieure.

Evolution prévue

Nous devons finir l'implémentation total des cartes, ajouter des statistiques avec aussi une implémentation d'un autre type de bot et finir les tests unitaires. Ensuite, nous améliorerons le code en le simplifiant dans certaines parties et nous ajouterons de la documentation afin de faciliter sa compréhension.

Conclusion itération finale

Analyse de notre solution : points forts et points faibles

Notre point fort est que la réalisation du projet à était terminée avec 4 joueurs et uniquement les cartes basiques et leurs effets, avec un système de commande où l'on peut facilement implémenter de nouvelle mécanique dans le jeu très facilement. Mais aussi notre système de bot qui lui aussi est très souple et nous permet d'ajouter de nouvelles versions de bot sans impacter les anciennes versions.

La réalisation du multi-partie parallèle qui permet une exécution très rapide d'une grosse quantité de partie à réaliser, mais nous a aussi permis de réaliser des statistiques sur un très grand nombre de partie afin de développer un bot spécifique.

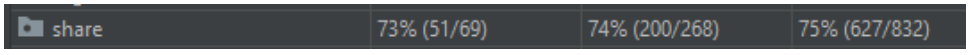
Notre point faible concerne la taille du projet qui augmente de plus en plus à travers les itérations. Nous avons essayé de simplifier au maximum le code en ajoutant par exemple de la documentation afin d'améliorer sa compréhension pour une personne venant de l'extérieur et de mieux détailler notre conception dans le rendu COO. Il est très difficile d'exposer l'entièreté de la conception de notre projet dans ce dossier par rapport au nombre de uses cases nécessaires.

Pour conclure nous avons un jeu entièrement fonctionnel avec 4 joueurs et uniquement les cartes basiques du jeu ainsi que toutes les actions qui doivent etre effectuées dans une partie. Nous avons également réalisé tous les tests du package share et du package server. Nous sommes alors arrivés à un taux de couverture pour :

Module server :

server	94% (68/72)	87% (345/394)	82% (1422/1720)
share	84% (58/69)	85% (228/268)	87% (728/829)

Module share :



share	73% (51/69)	74% (200/268)	75% (627/832)