Projet de Développement
Conception Orientée Objet

UNIVERSITÉ DE NICE SOPHIA ANTIPOLIS

Mars 2020

Enseignants : Philippe RENEVIER / Jerome DELOBELLE

Licence 3 – MIAGE

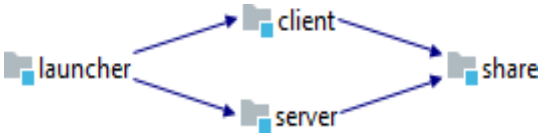
SESSION 2019 / 2020

Etudiants : BOUCHE Steven, CHAPOULIE Dorian, KAROUIA Alaedine, GARNIER Corentin, LONGIN Rémi

Table des matières

Architecture générale

Composition du projet Maven Diceforge

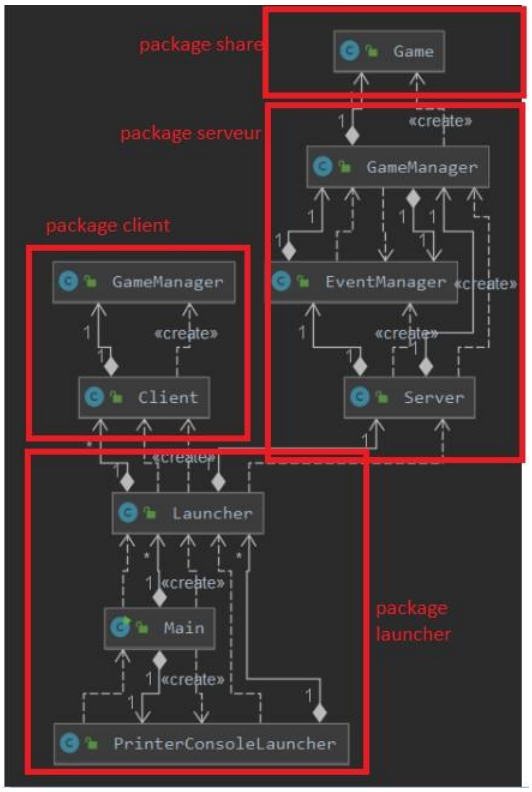


Le projet Diceforge est composé au total de 4 modules. Le module Launcher a pour rôle de l'instanciation d'un serveur de jeu ainsi que de l'instanciation des clients qui devront se connecter au serveur sans l'aide du launcher en réseau.

Le module serveur lui-même 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. Plus tard il serait intéressant pour chaque action d'un client de broadcast le résultat de l'action à tous les clients afin que l'IA puisse prendre en compte toutes les données de la partie. Pour le moment le client choisit de façon aléatoire parmi les différents choix, il n'a donc pas besoin de toutes les données du plateau pour l'instant.

Exemple de classe en fonction des packages :



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 Activité du Main

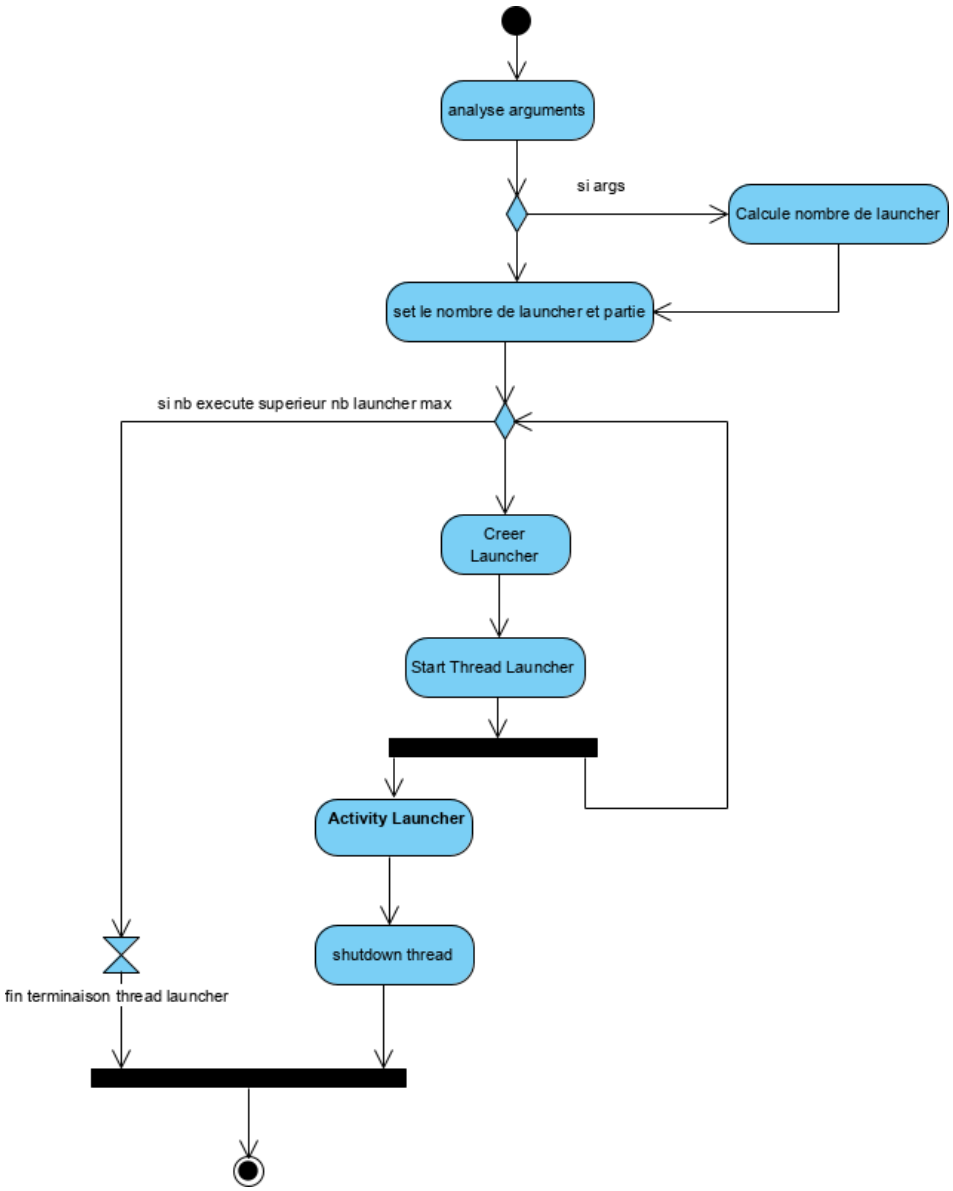


Diagramme Activité du Launcher

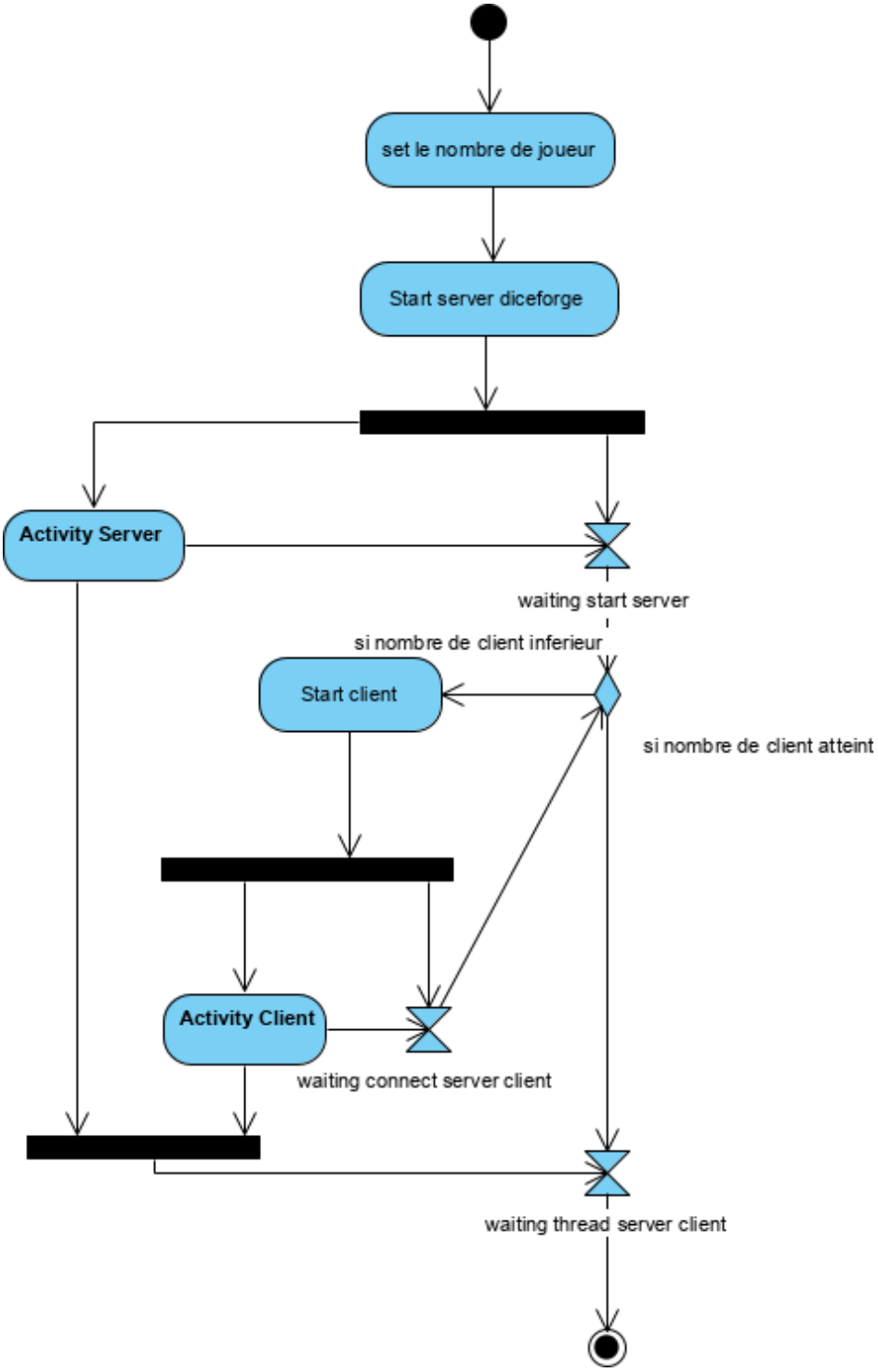
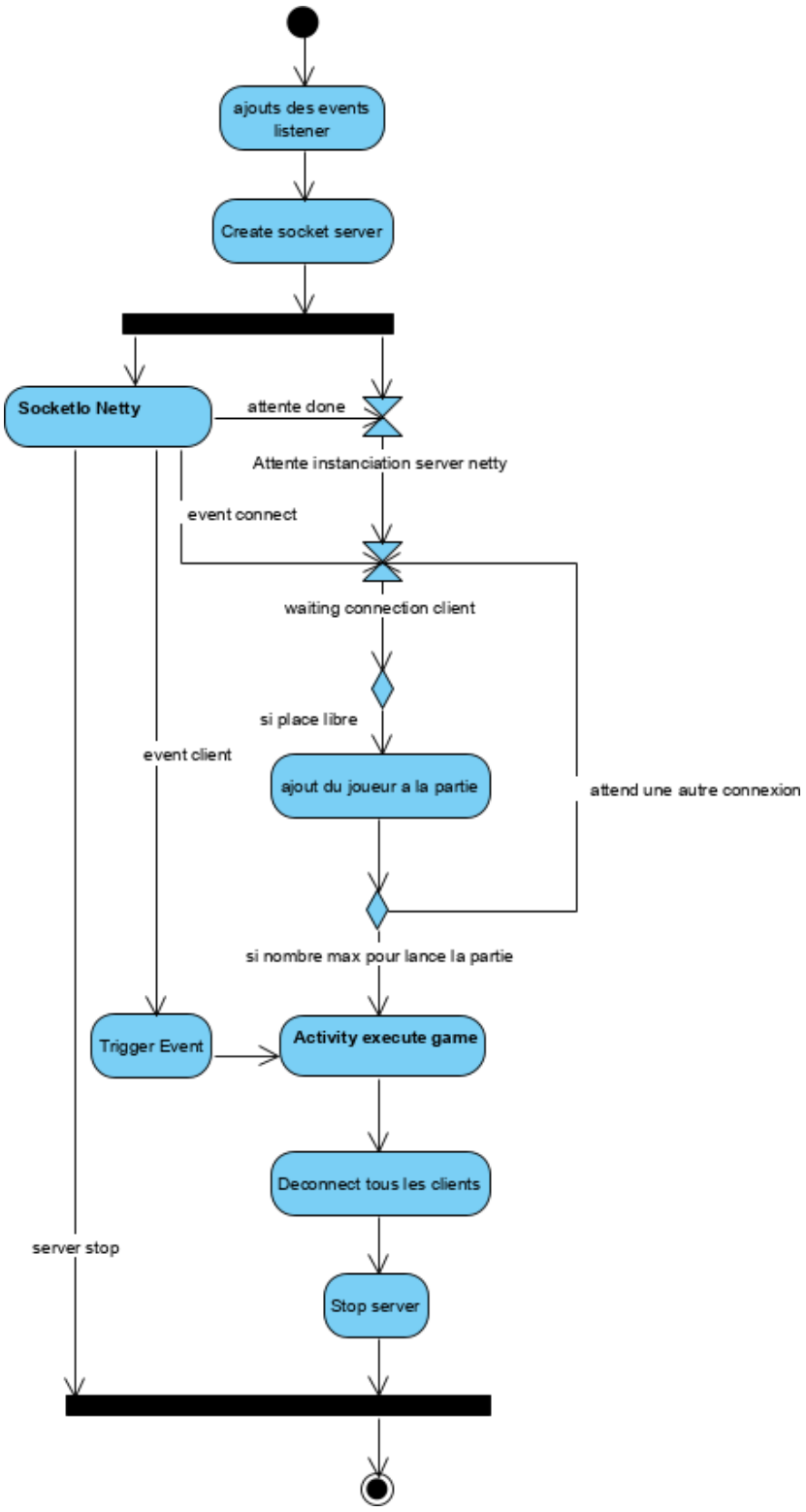


Diagramme Activité du Serveur



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.

Enfin il met fin au tour actif. Une manche est composée de tour actif étant égal au nombre de joueur. Une partie est composée de 9 manches.

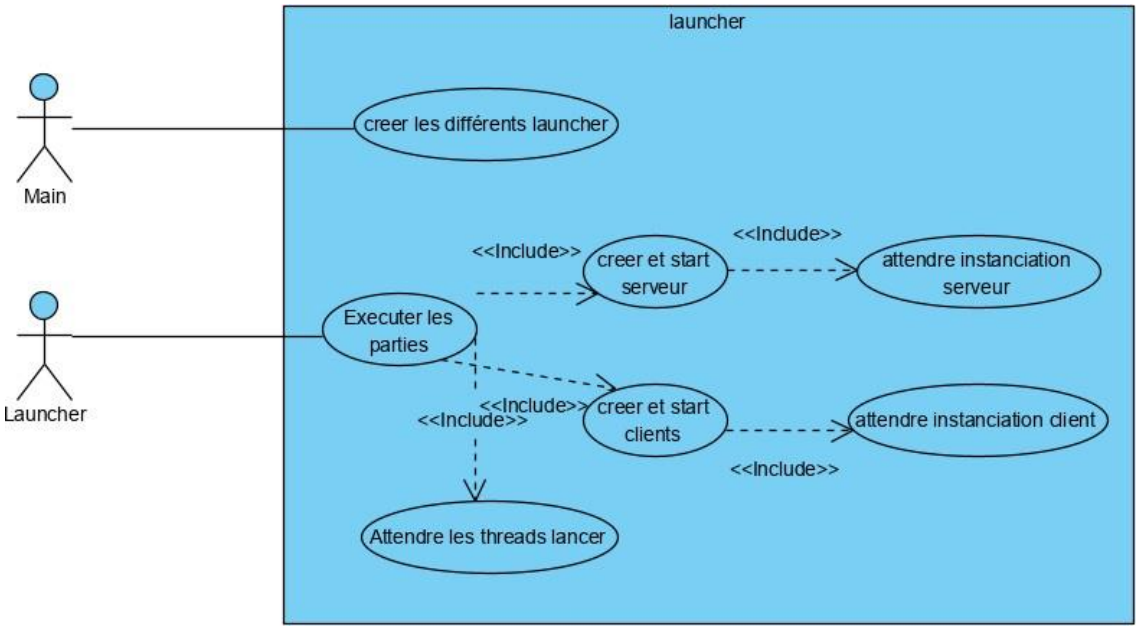
Dans le cas où l'argument "-p" suivie d'un nombre de partie est spécifiée le programme divisera le nombre de partie à exécuter en fonction d'une taille de pool de launcher dans une classe de configuration. Il lancera alors X launchers avec chacune des Y parties à exécuter.
Le programme s'arrête quand tous les launchers ont fini leur travaille.

Fonctionnalités qui sont implémentées :

- Client / Server
- Boucle de jeux
- Forge
- Temple
- Face Simple de dés
- Face hybride de dés
- Iles
- Cartes sans actions
- Action des faces hybrides
- Action des faces simples
- Multipartis

Module principal (launcher)

Diagramme Use Case



User Story

En tant que Launcher je souhaite pouvoir exécuter X parties

En tant que Launcher je souhaite pouvoir créer le thread serveur et le start

En tant que Launcher je souhaite pouvoir créer les threads clients et les starts

En tant que Launcher je souhaite pouvoir attendre la fin de la partie

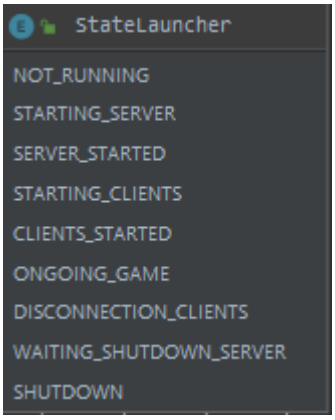
Scénarios

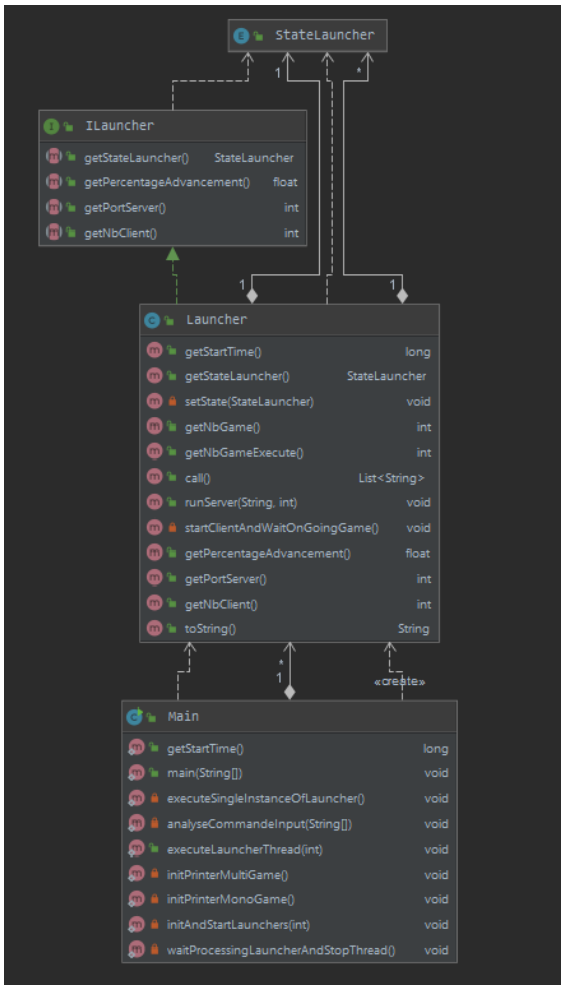
1. Créer et lancer le serveur
    a. Instancier un objet serveur avec un port particulier
    b. Lancer le server dans un thread
    c. Attendre l'instanciation du server
    d. Instancier les clients avec le port serveur
    e. Start les clients dans plusieurs threads
    f. Attendre la fin d'exécution du serveur et des clients

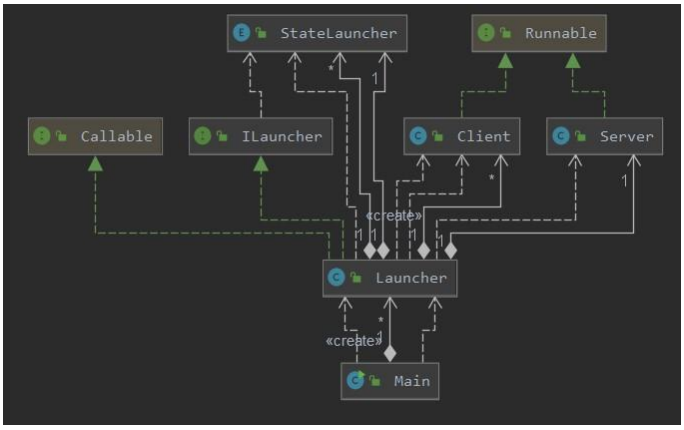
2. Créer et lancer le Client
    a. Instancier un objet client avec tous les paramètres
    b. Lancer le client dans un thread
    c. Attendre la notification du client qu'il s'est bien connecter
    d. Si le nombre de joueur actuel est inferieur au nombre de joueur max alors retourner à l'étape a. sinon attendre la fin des threads clients

Conception Logiciel

*P*oint de vue statique







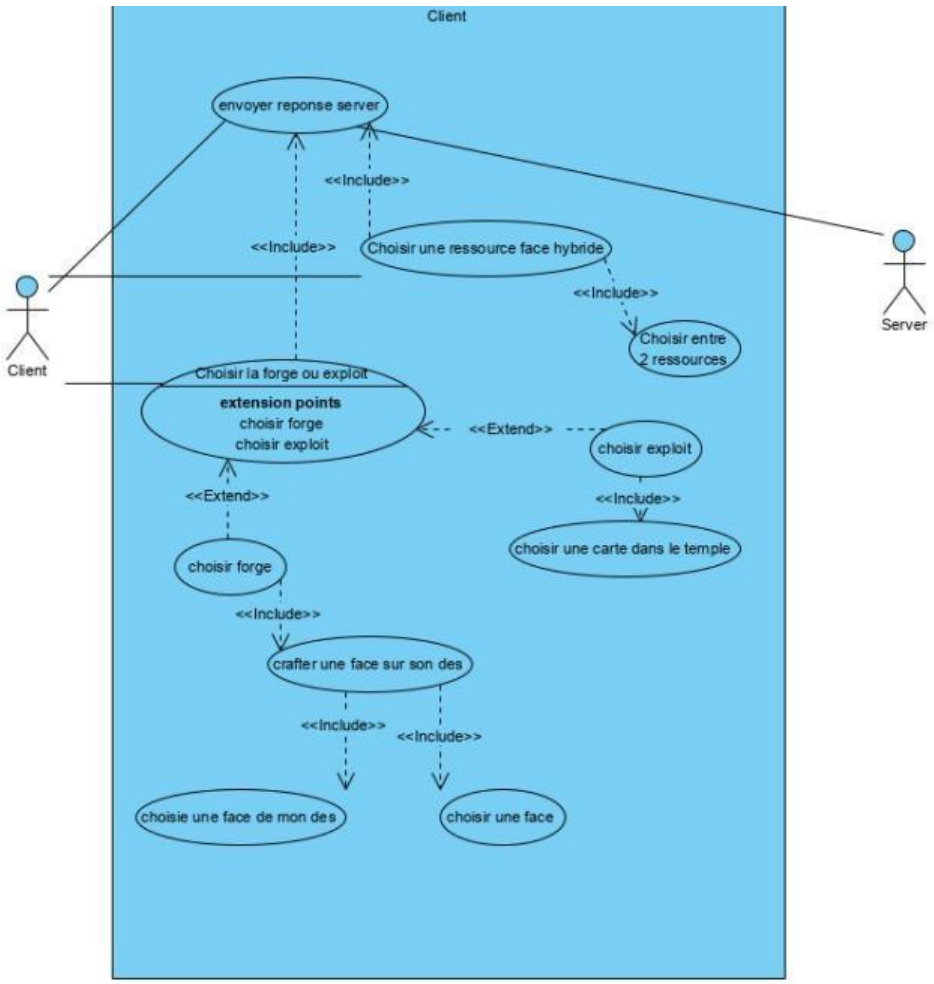
Le launcher est instancié par le programme principal. Le launcher doit exécuter un certain nombre de partie. On peut alors voir sur le schéma au-dessus que le launcher lance X thread clients ainsi qu'un thread serveur des autres modules.

Module Client

Analyse des besoins

Les fonctionnalités réseau seront exposées dans la partie client/serveur plus en détail.

Diagramme Use Case



User Story

En tant que client je souhaite pouvoir choisir si je veux forger une face sur mon dé ou acheter une carte.

En tant que client je souhaite pouvoir choisir entre 2 ressources si je tombe sur une face hybride

Scénario

1. choixForgeorExploitorNothing

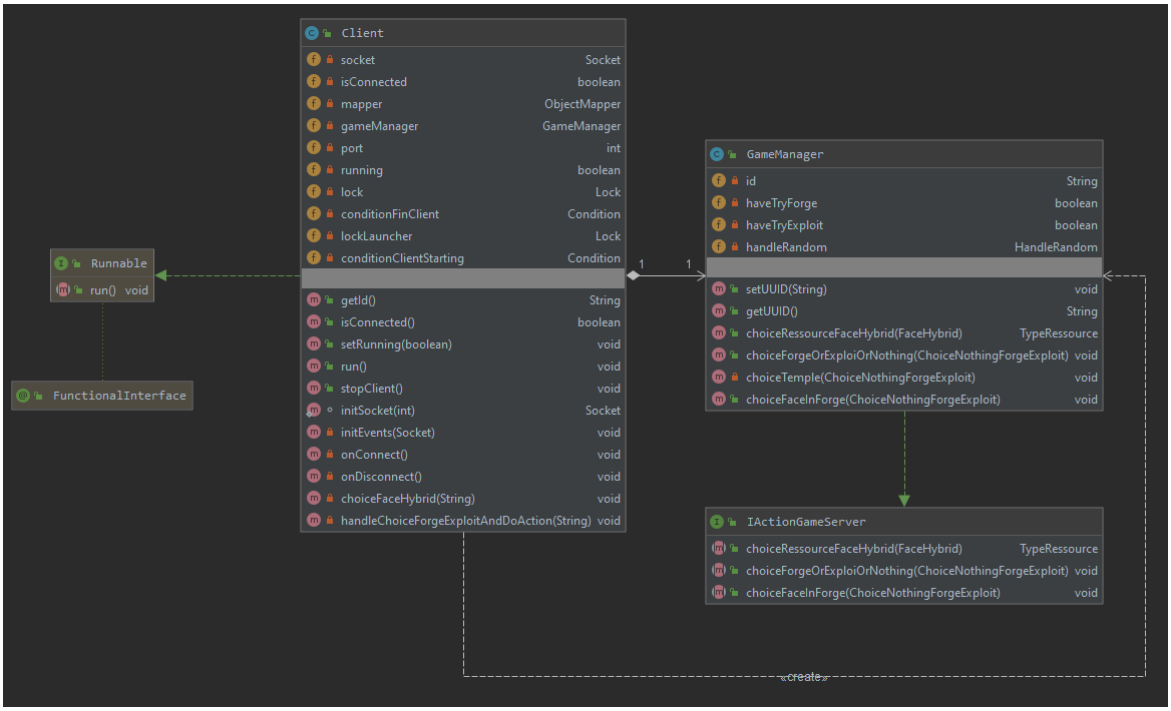
    a. choisirForge si je peux
    b. Forger une face sur son dé
    c. Choisir le dé
    d. Choisir la face à modifier
    e. ChoisirExploit si possible
    f. Choisir une carte dans le temple
    g. Ne rien faire

2. choixFaceHybrid

    a. Choisir carte dans le temple

Conception Logiciel

Point de vue statique



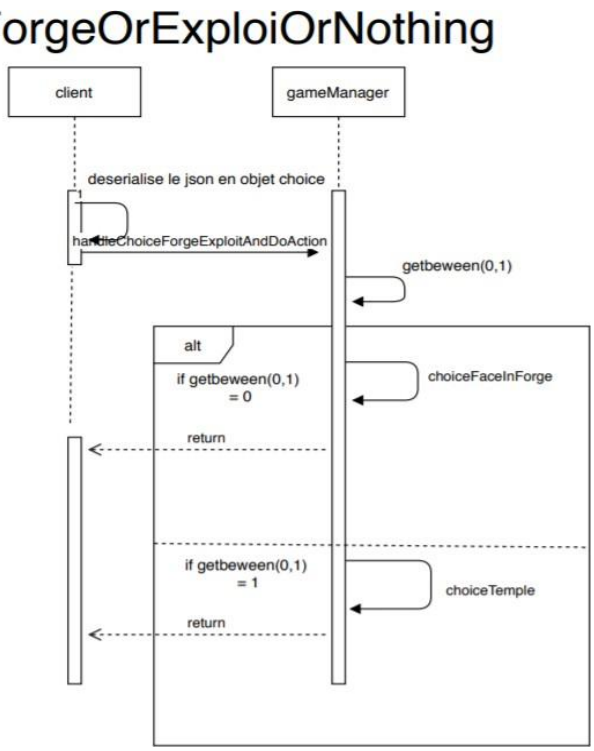
Le client est composé d'un gameManager. Le gameManager a pour rôle d'effectuer les décisions du choix en fonction de l'objet envoyé par le serveur.
Il contient alors les méthodes pour choisir de forger une face sur le dé ou alors d'acheter une carte en fonction de l'événement que le serveur lui envoi.
Tout l'aspect réseau est traité dans la partie réseau.

Point de vue dynamique

ChoiceForgeOrExploiOrNothing

Précondition : Le serveur a précédem envoyé une demande de choix.

Postcondition : Le client a renvoyé l choice rempli avec ses décisions.



Le client désérialise le JSON en objet choice envoyé précédemment par le serveur puis ensuite l'envoie au gameManager qui possède une méthode public pour gérer le choix et des méthodes privé pour compléter l'objet choice en fonction du getbetween(0,1).

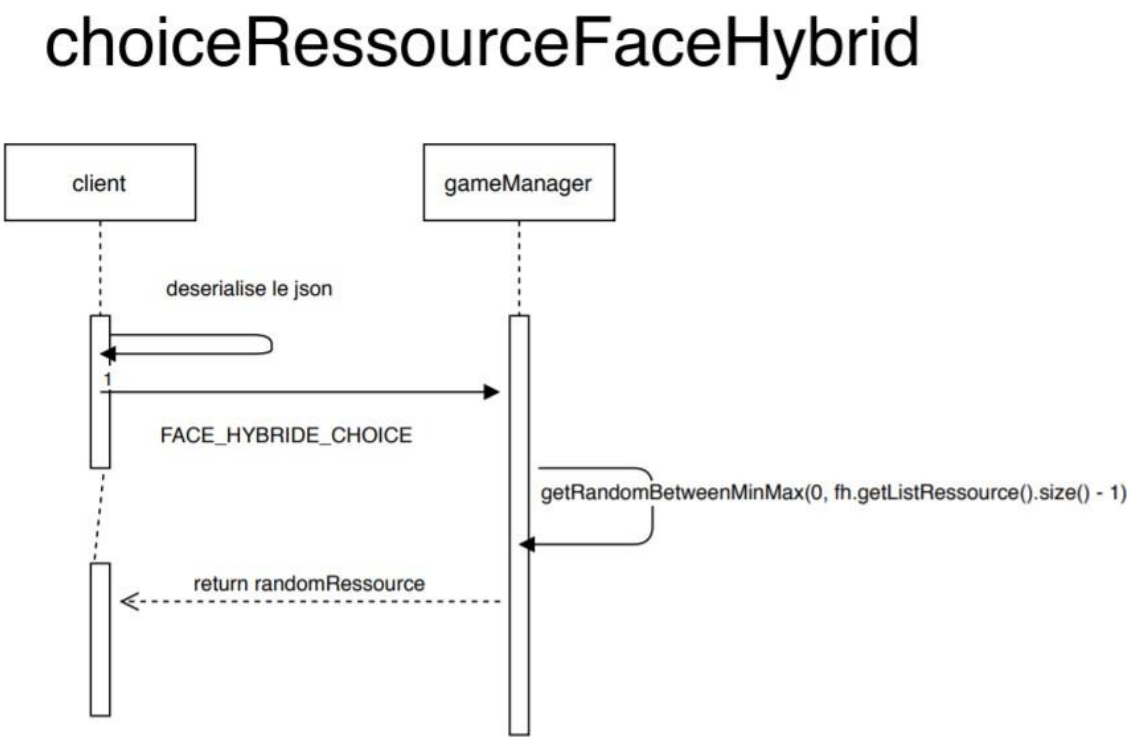
S'il obtient un 0, il utilise la méthode privé choiceFaceforge et complète l'objet puis return.

S'il obtient un 1, il utilise la méthode privé choiceTemple et complète l'objet puis return.

ChoiceRessourceFaceHybrid

Précondition : Le serveur a précédemment envoyé une demande de choix.

Postcondition : Le client a renvoyé la ressource aléatoire.



Le client désérialise le json renvoyé par le serveur puis l'envoie au gameManager FaceHybrideCHOICE.

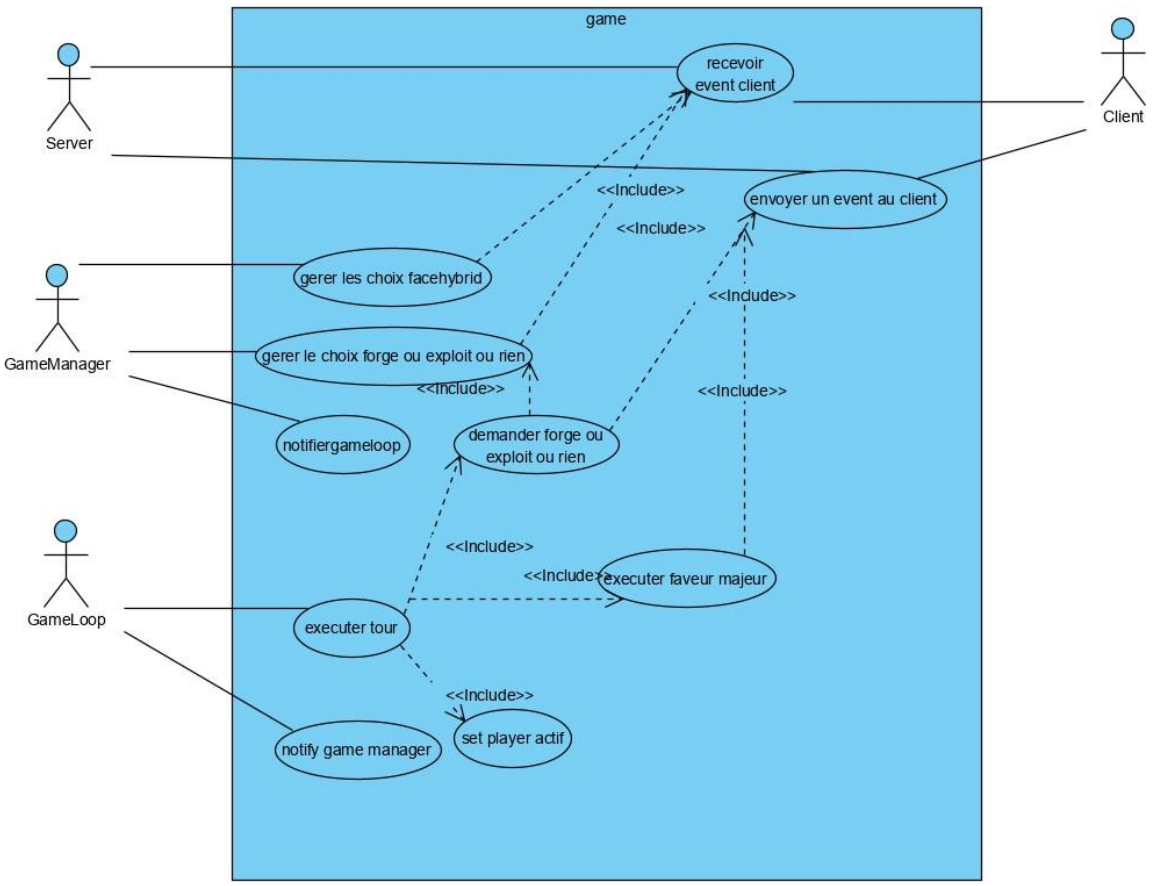
En fonction du random, une ressource va être choisie puis renvoyée au client.

Module Serveur

Analyse des besoins

Les fonctionnalités réseau seront exposées dans la partie client/serveur plus en détail.

Diagramme Use Case



User story

En tant que serveur je souhaite pouvoir exécuter une partie

En tant que serveur je souhaite pouvoir connecter des clients

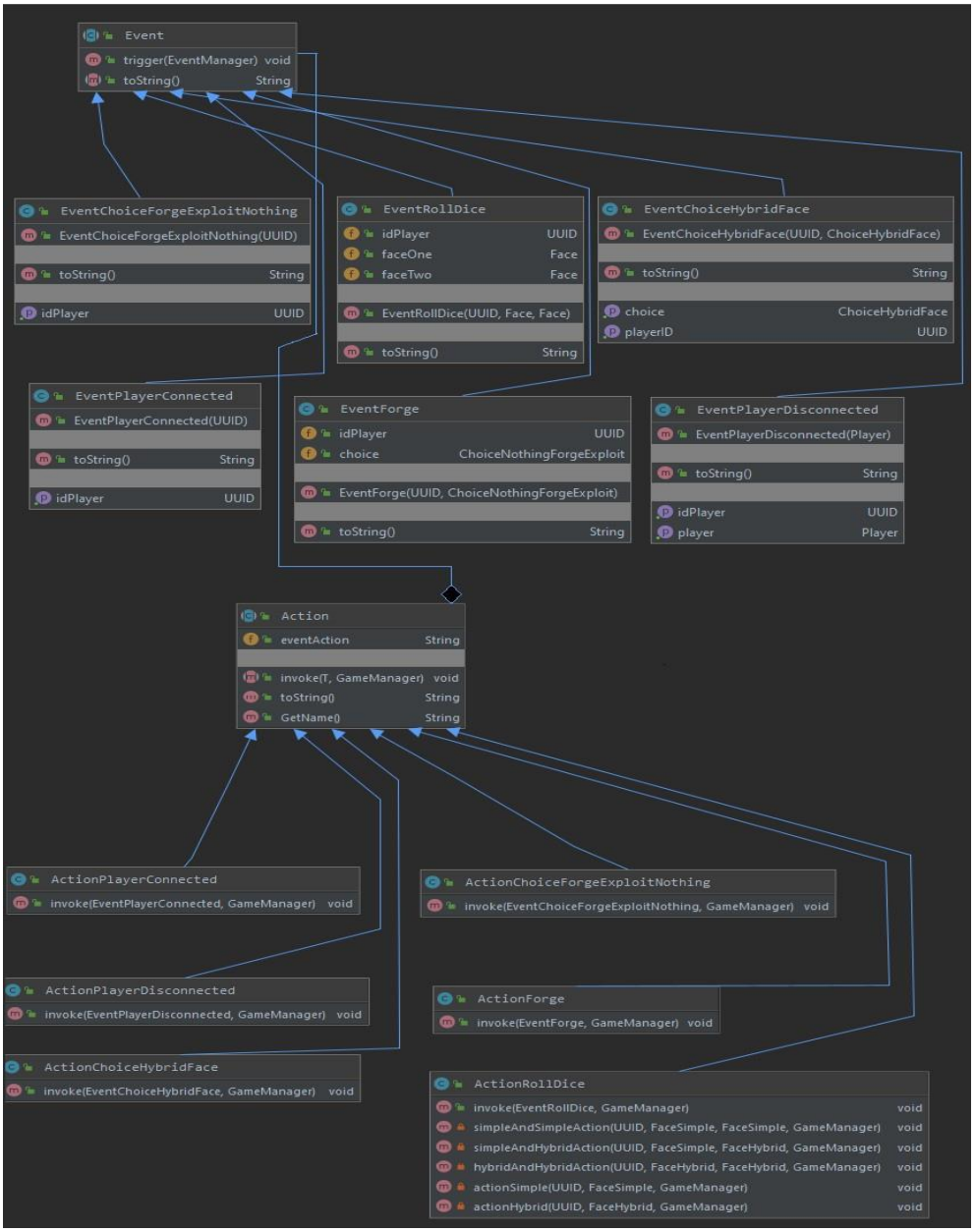
En tant que serveur je souhaite pouvoir gérer les différents évènements client :

- Choix face hybrid
- Choix Forge ou Temple

Conception Logiciel

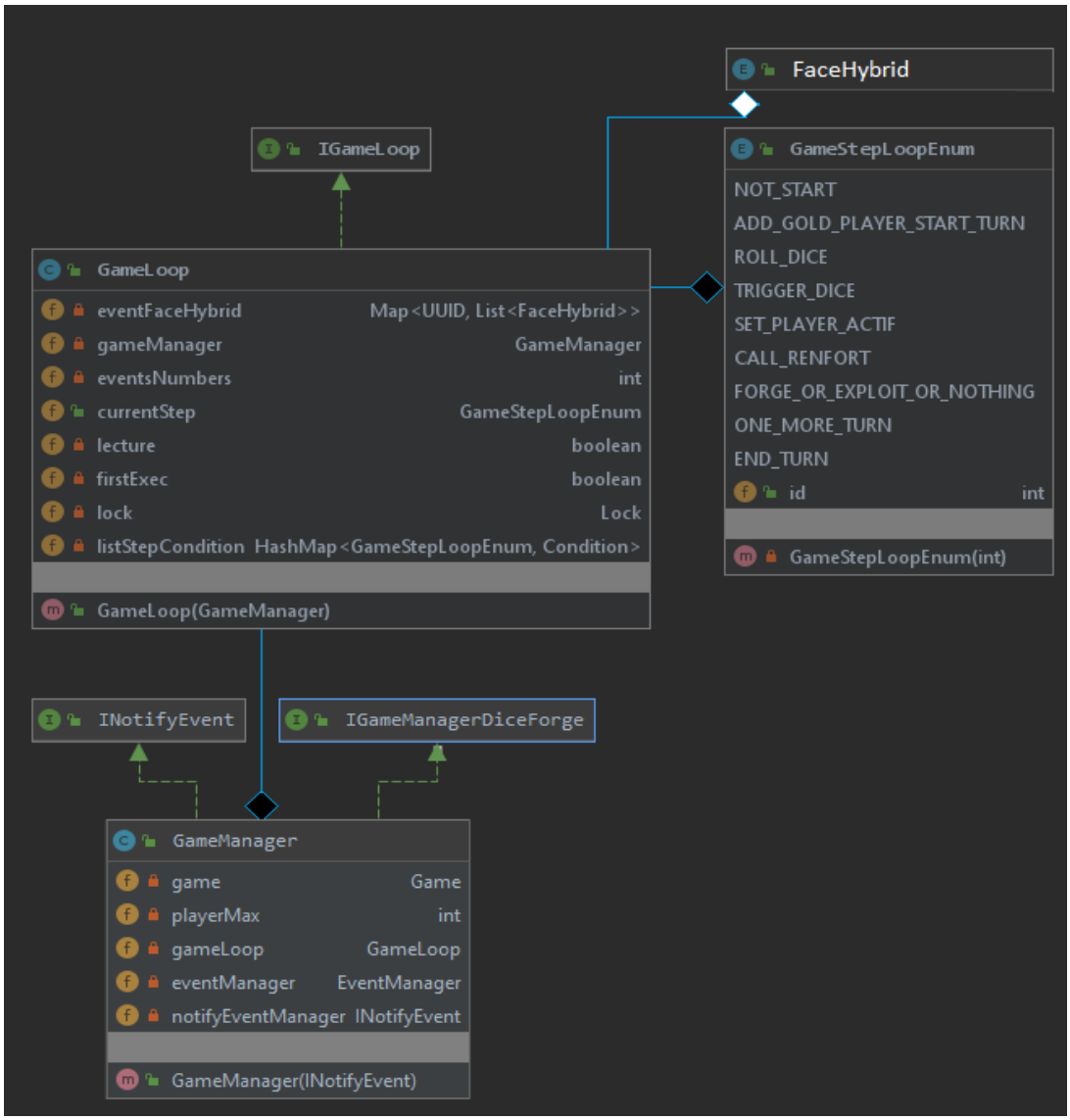
Point de vue statique itération 3

1. Les événements et les actions



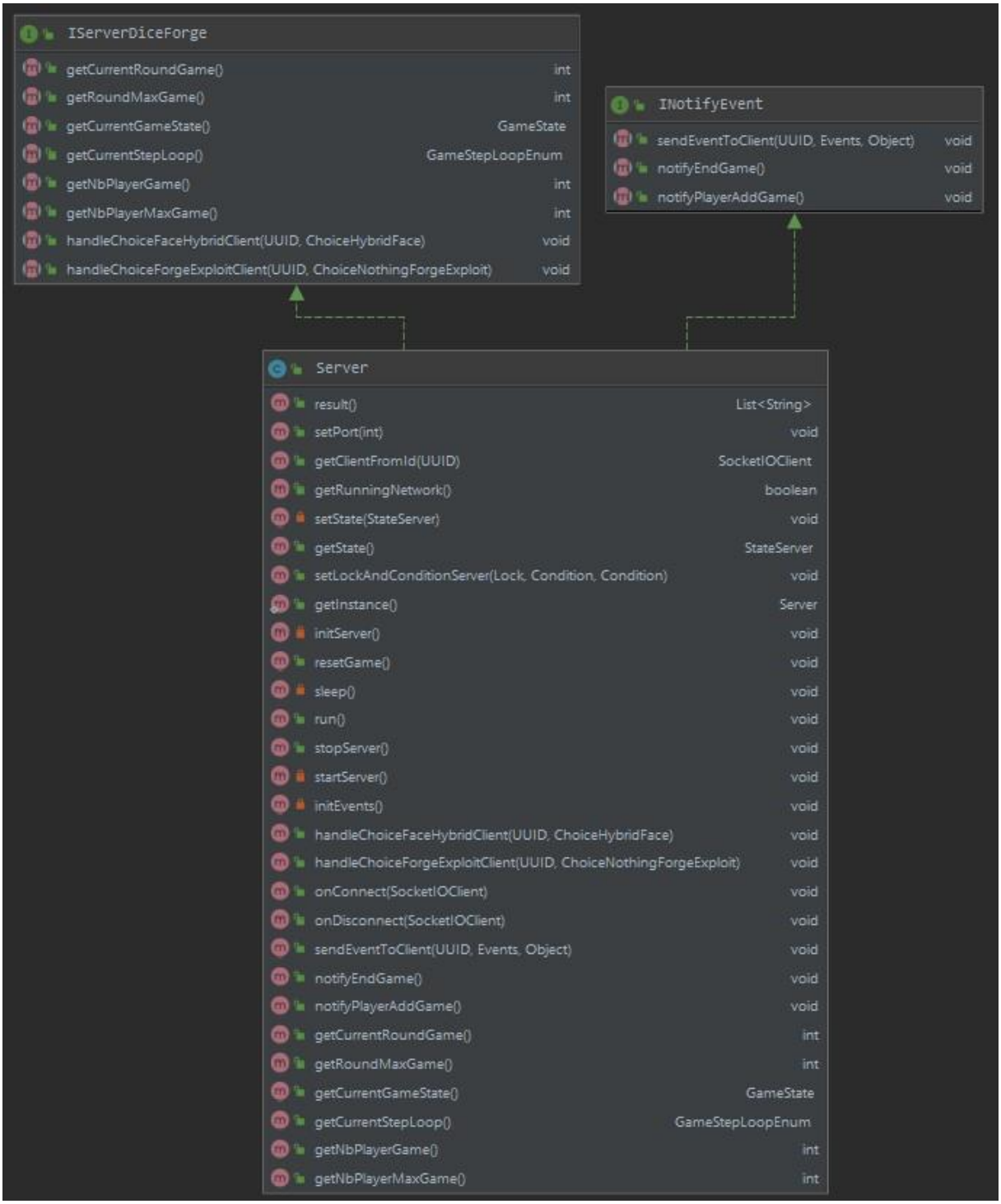
Le serveur utilise un design pattern observeur afin de gérer tous les événements liés au jeu. Chaque événement est lié à une action qui sera invoquée. Nous avons créé une classe par Evenements et donc par Action.
Par exemple nous avons une classe “EventRollDice” qui hérite de la classe “Event” et une classe action complémentaire : “ActionRollDice” qui hérite de la classe Action et qui sera invoquée par l'événement.

2. La boucle principale et le manager



La boucle de jeu principale aide le game manager a gérer tous les traitements du jeu, ainsi qu’un “current state” qui correspond à l’état actuel du jeu (lancé de dé, utilisation de la forge, exploits …).

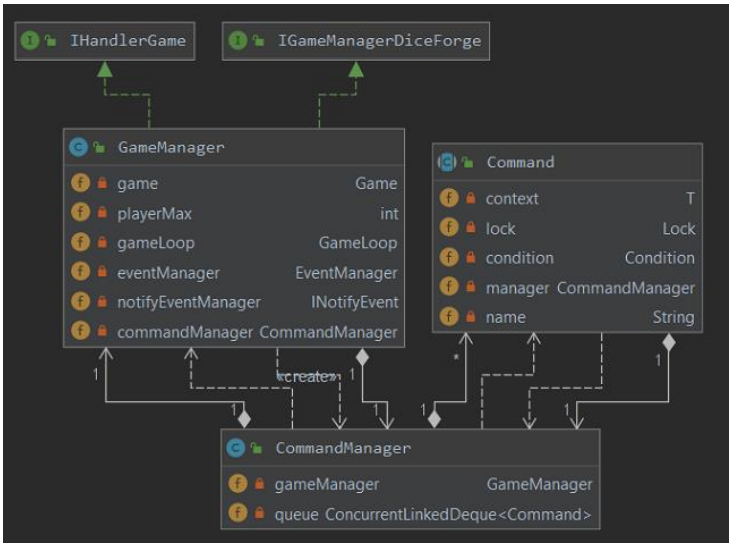
3. Le serveur



Le serveur s'occupe de gérer les événements (envoi des événements au clients, et handle des événements reçus), en fonction de l'état du jeu ("current state") et réalise les redirections en fonction.

Point de vue statique itération 5

Le système d'événement à complètement était revu pour prendre en compte le mécanisme des cartes. Nous nous basons maintenant sur un système de commande.

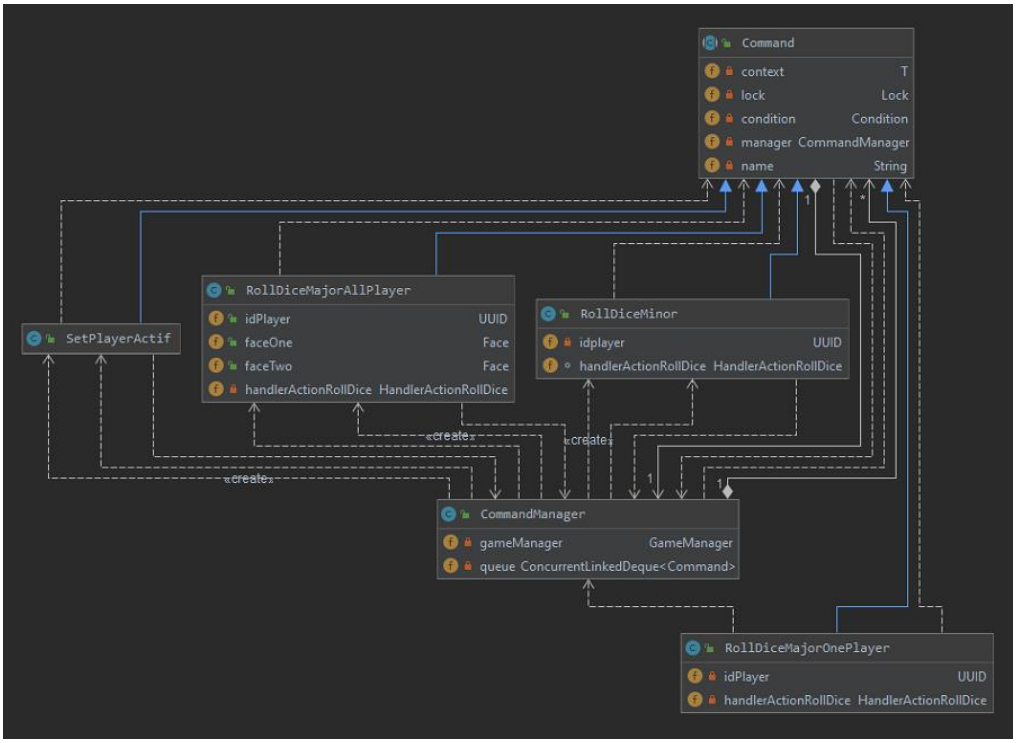


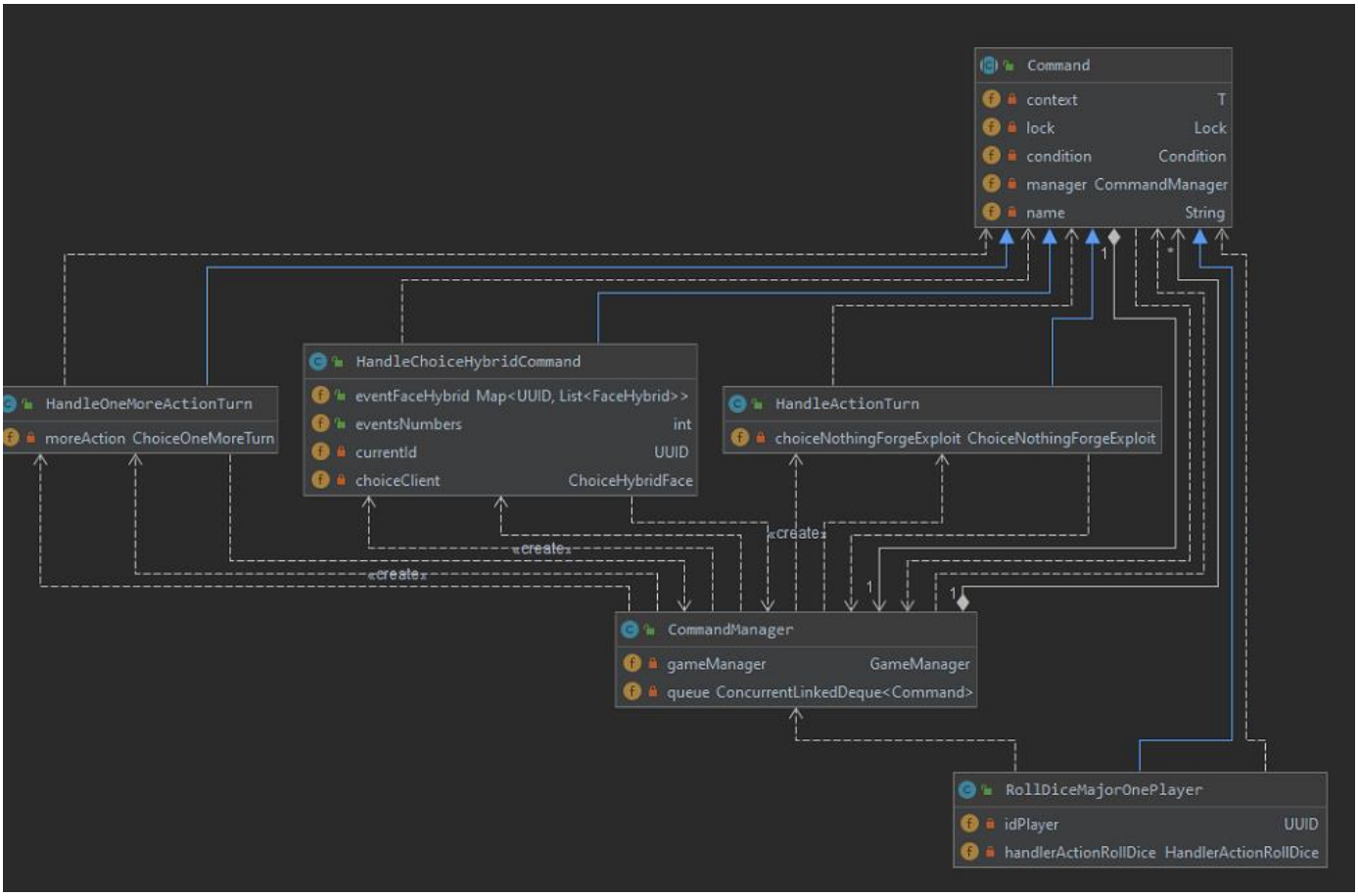
Le game manager maintenant est composé d'un commandManager. Son rôle est de schedule et créer les commandes entre elles ainsi que centraliser certaines actions comme l'ajout de gold pour un joueur. Le gameManager fait le lien entre les évenements réseaux du serveur et le commandManager pour exécuter le bon traitement.

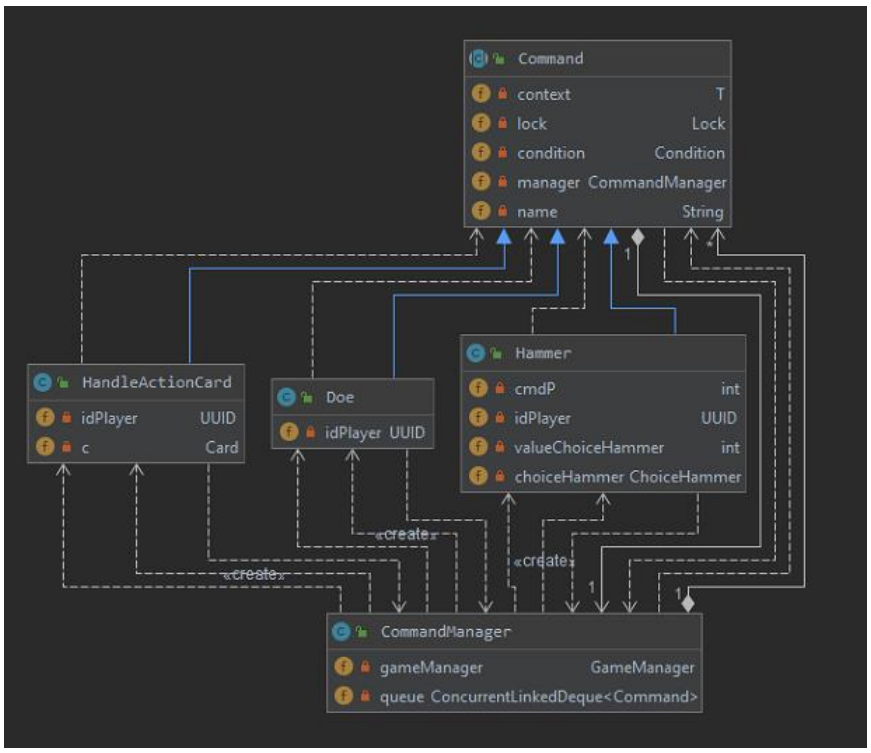
Le commandManager est composé de plusieurs commandes que nous verrons plus tard en détails. Chaque commande hérite de la classe abstraite Command. Une commande possède un context généralement le gameManager permettant à la commande de récupérer toutes informations utiles de la partie lors de son exécution. De plus une commande possède un lien vers le commandManager lui permettant d'exécuter une commande dans une commande et ainsi de suite. Pour gérer l'enchainement des commandes, le command Manager possède une Queue et empile et dépile les commandes quand elles doivent être exécuter ou quand elles sont finies.

Chaque commande possède un verrou et une condition. Cela permet à la commande d'attendre le choix client s'il est nécessaire à la commande d'obtenir une réponse du joueur.

Pour l'instant nous avons plusieurs commandes fonctionnelles comme par exemples les commandes changent les données de la partie :




Command
context T
lock Lock
condition Condition
manager CommandManager
name String
HandleChoiceHybridCommand
eventFaceHybrid Map<UUID, List<FaceHybrid>>
eventsNumbers int
currentId UUID
choiceClient ChoiceHybridFace
HandleOneMoreActionTurn
moreAction ChoiceOneMoreTurn
HandleActionTurn
choiceNothingForgeExploit ChoiceNothingForgeExploit
«create»
«create»
«create»
CommandManager
gameManager GameManager
queue ConcurrentLinkedDeque<Command>
RollDiceMajorOnePlayer
idPlayer UUID
handlerActionRollDice HandlerActionRollDice

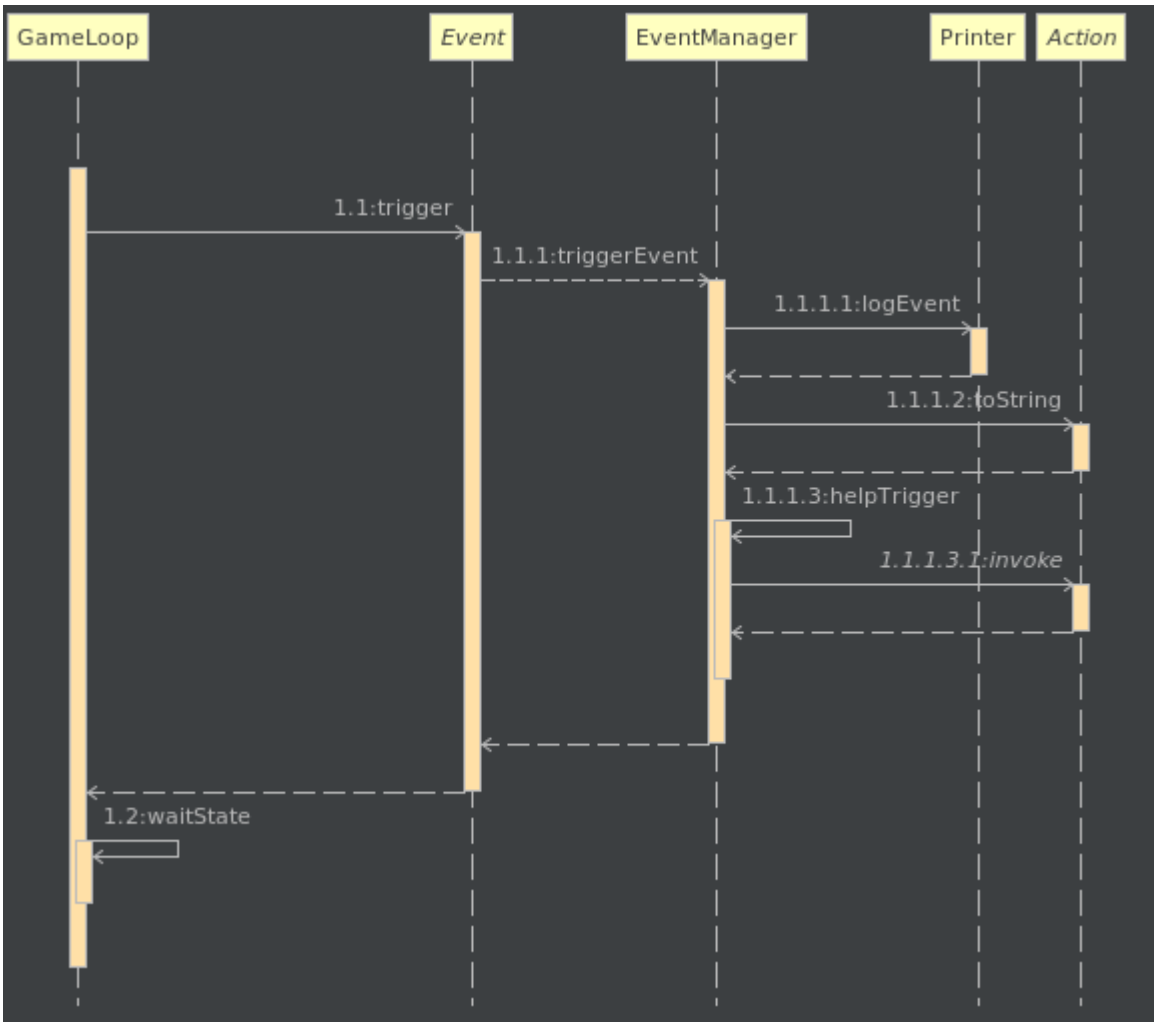

Command
context T
lock Lock
condition Condition
manager CommandManager
name String
HandleActionCard
idPlayer UUID
c Card
Doe
idPlayer UUID
Hammer
cmdP int
idPlayer UUID
valueChoiceHammer int
choiceHammer ChoiceHammer
«create»
«create»
«create»
CommandManager
gameManager GameManager
queue ConcurrentLinkedDeque<Command>

Point de vue dynamique itération 3

ChoiceForgeOrExploiOrNothing

Pré-condition :
Le client est connecté
Le client doit choisir une action

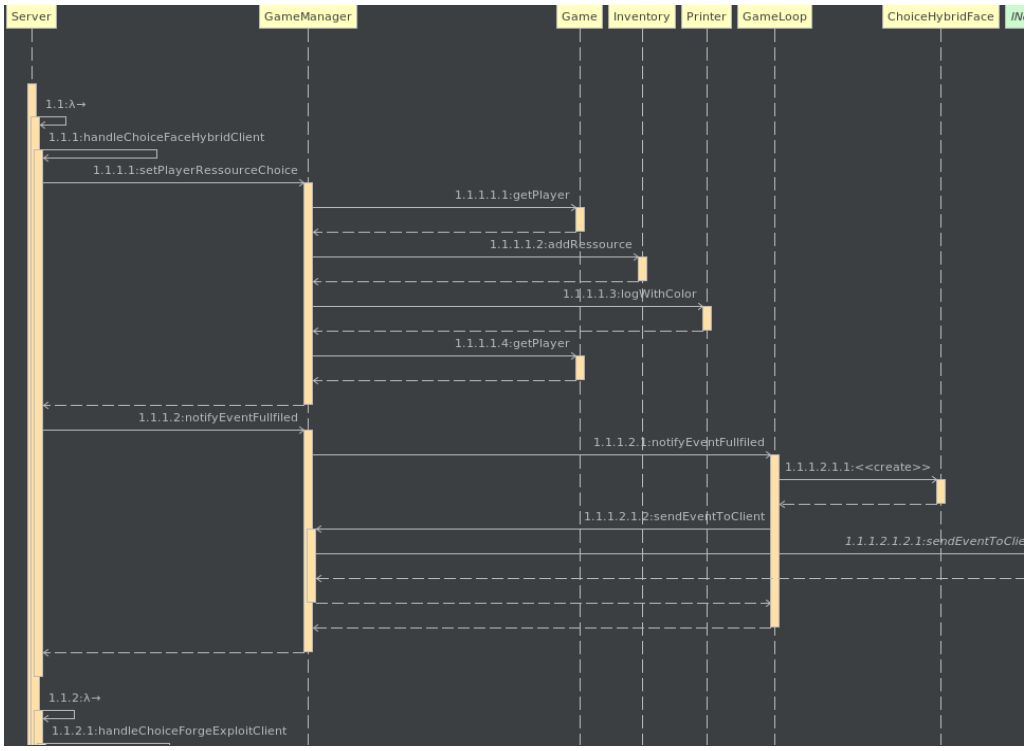
Post-condition :
Le serveur exécute l'action choisie
Le serveur invoke l'action choisie par le client et gère son choix.



ChoiceRessourceFaceHybrid

Pré-condition :
Le client est connecté
Le client doit choisir une ressource dans sa face

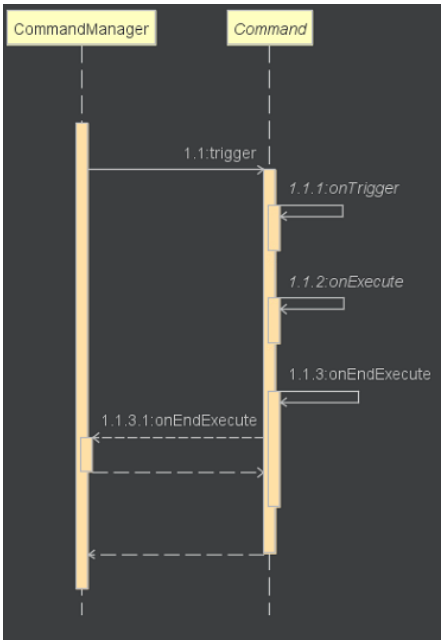
Post-condition :
Le serveur ajoute la ressource choisie à l'inventaire du client
Le serveur gère le choix du client, modifie son inventaire et affiche son choix dans la console puis notifie le client



Point de vue dynamique itération 5

Execution d'une commande :

Lorsqu'on demmande de trigger une commande le commande manager créer la commande et la trigger. Quand il trigger la commande cela va exécuter dans l'ordre onTrigger, OnExecute et OnEndExecute. Avant le trigger le commande manager add la queue la commande et lors de onEndExecute depile la commande.

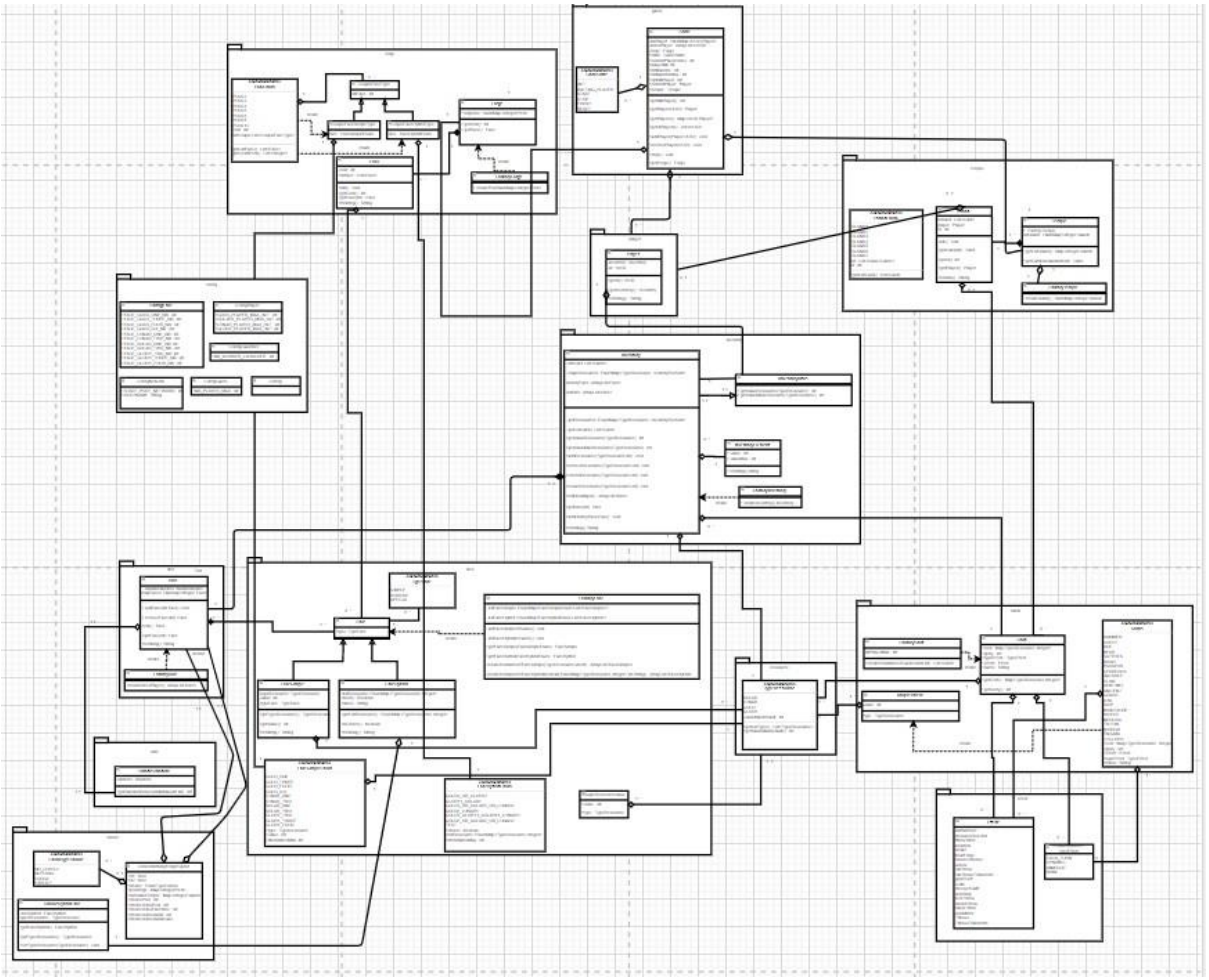


Module Share

Conception Logiciel

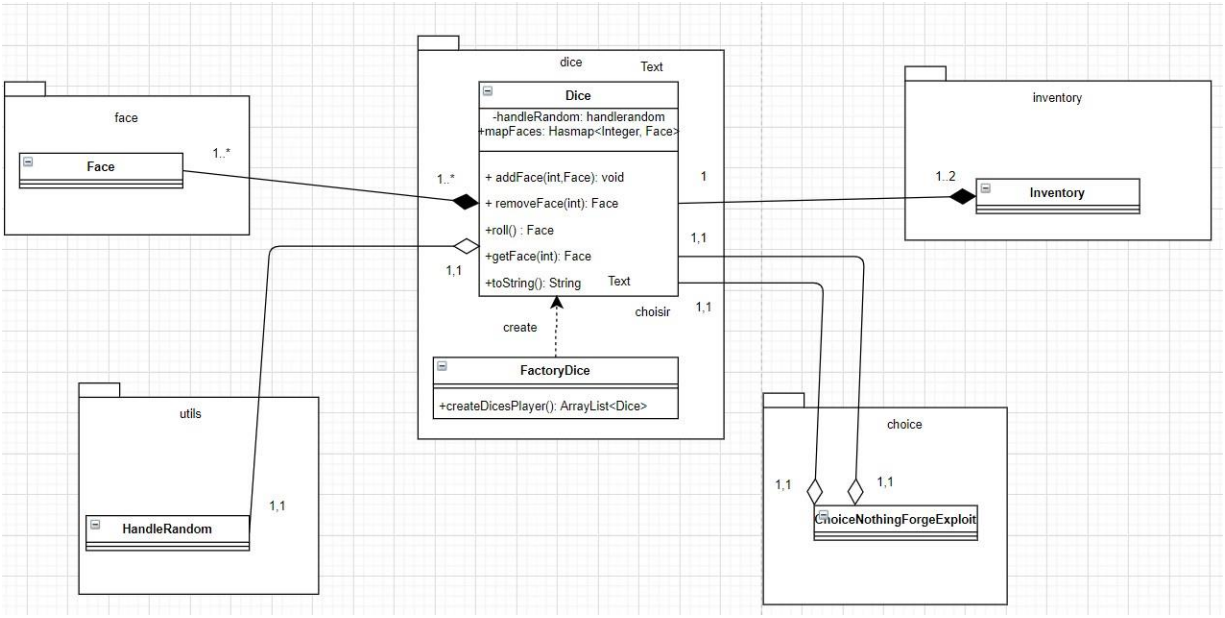
Point de vue statique

1. Représentation générale du package share



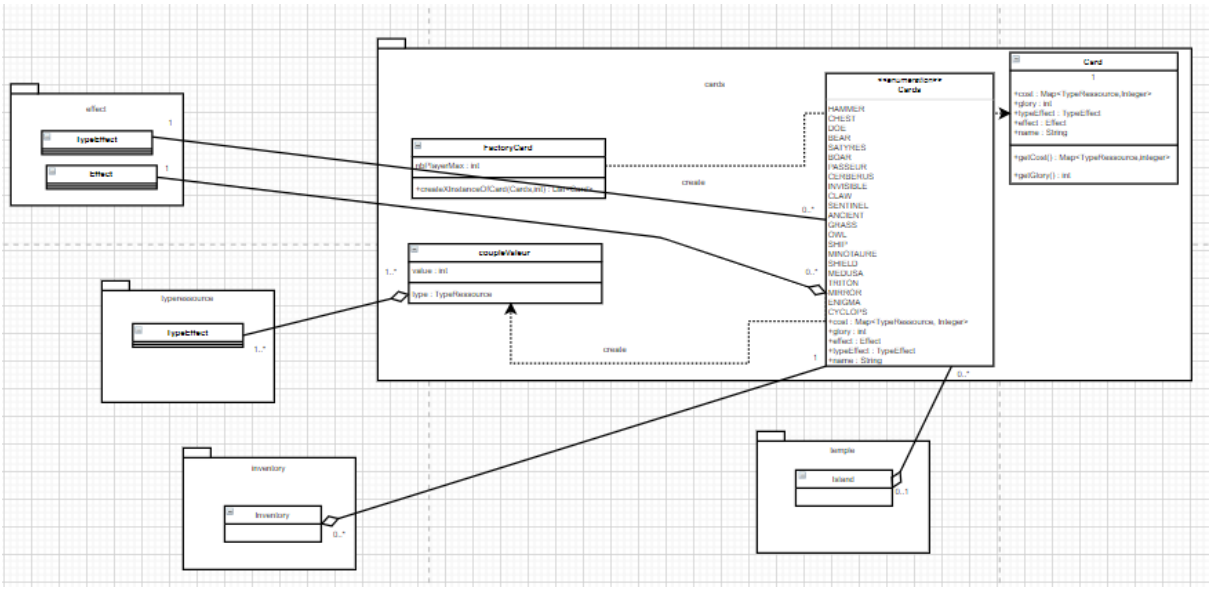
Lien permettant de voir un meilleur rendu :

2. Gestion des dès



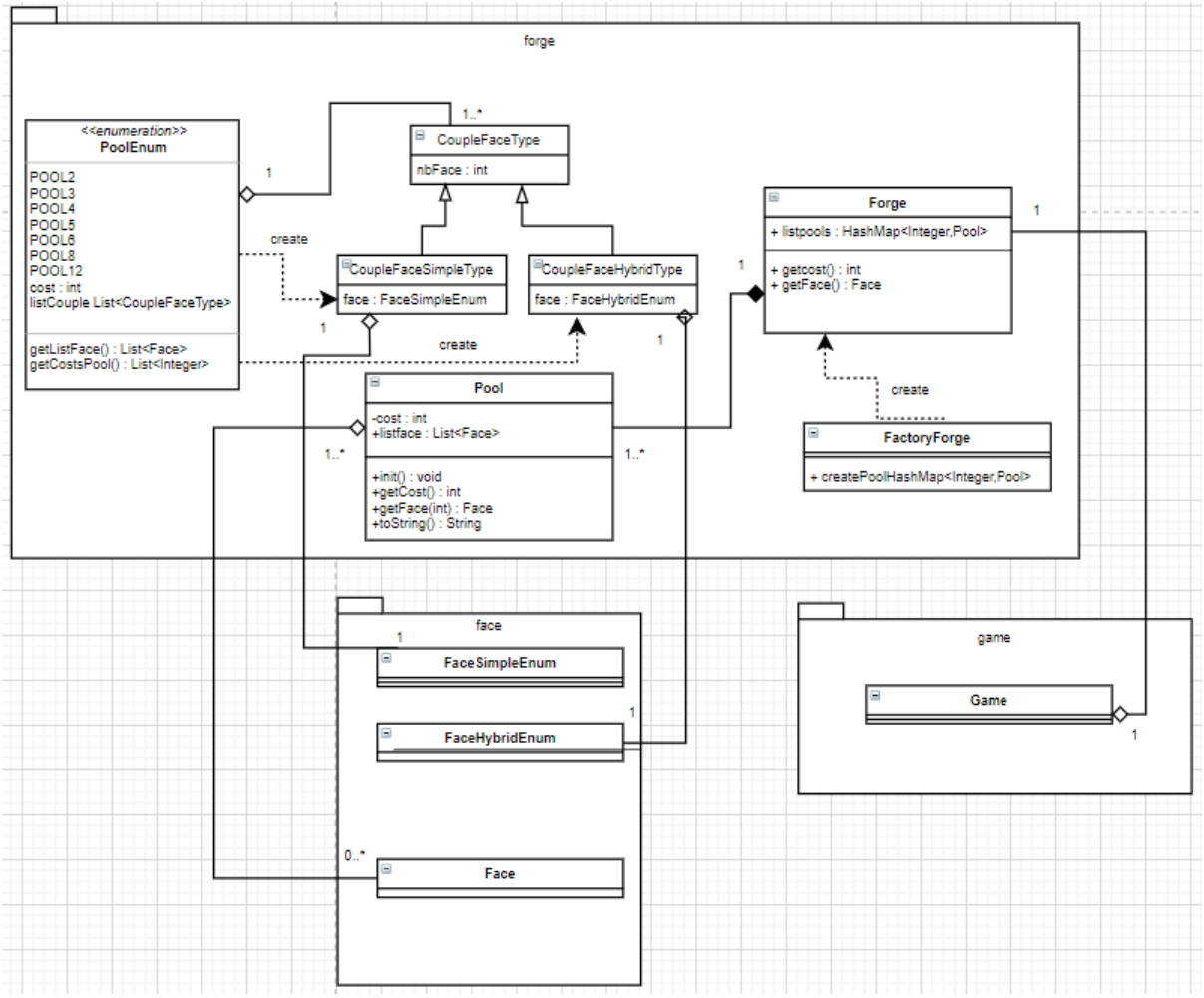
Un dé est obligatoirement rangé dans un inventaire, un inventaire peut accueillir 2 dés. Un dé est composé de plusieurs faces et une face peut composer plusieurs dés.
Lorsqu'un dé est lancé (roll) il fait appel à la fonction HandleRandom.
FactoryDice permet de créer les 2 dés d'un joueur.

3. Gestion des cartes



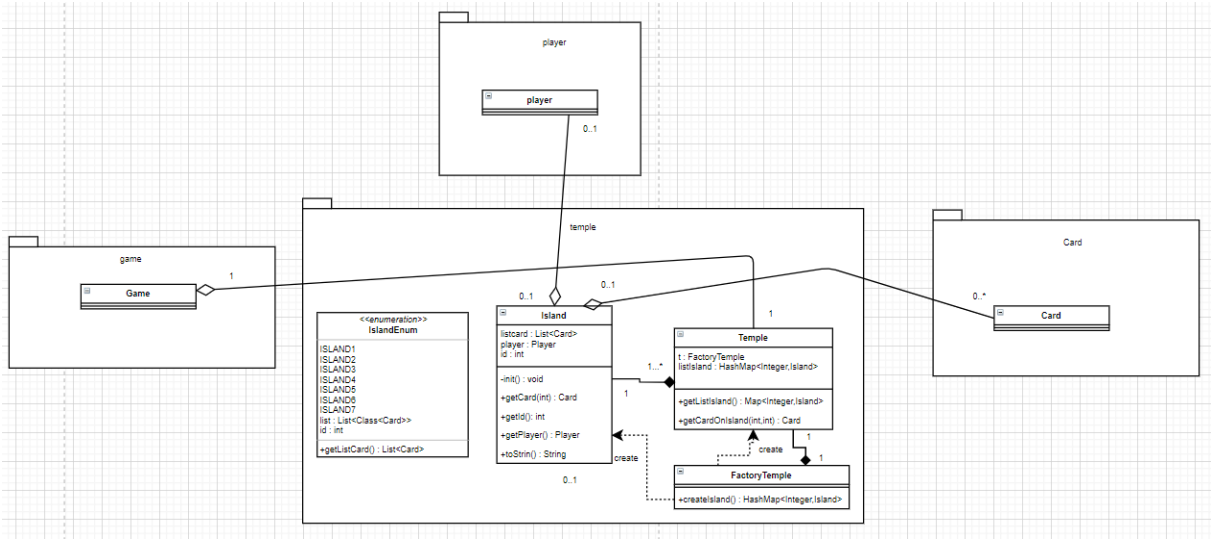
Une carte peut se trouver dans une island ou dans un inventaire. Chaque carte a un effet et un type d'effet (recurrent ou dynamique) une carte à un coût, l'énumération Cards permet de répertorier toutes les cartes d'une partie. FactoryCard permet de créer les cartes dans les islands au début de la partie.

4. Gestion de la forge



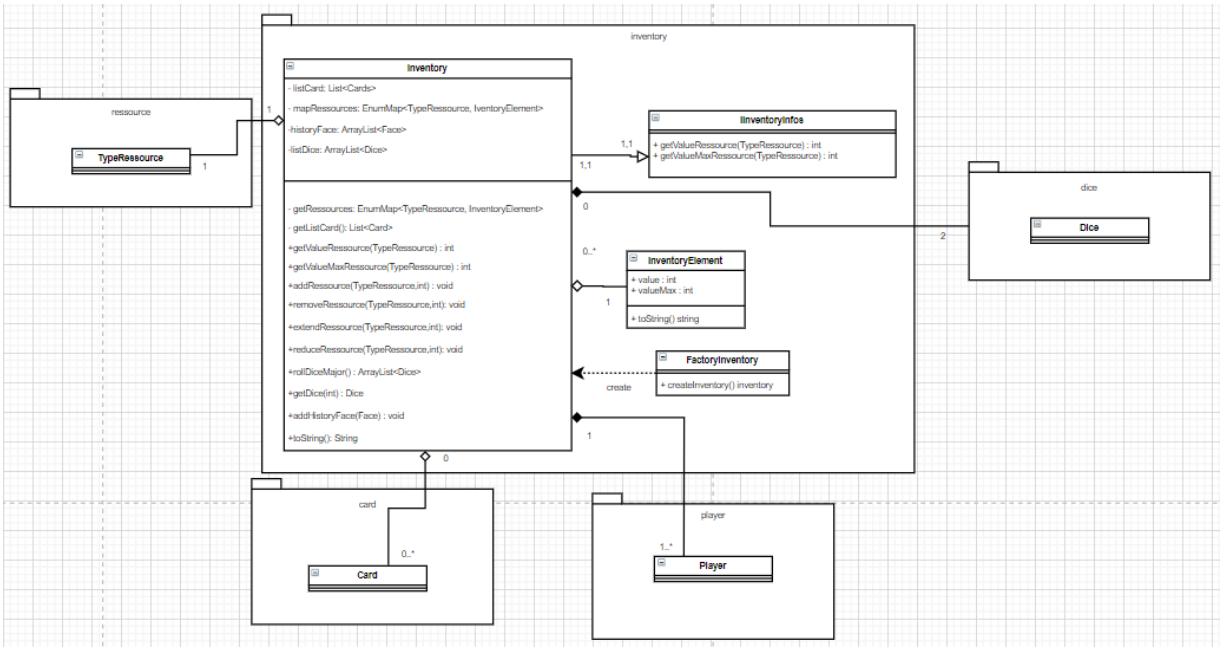
La forge est composée de 7 pools, chaque pool possède un coût et est constituée de différentes faces simple ou Hybrid qui sont dans une liste. Une forge est dans une seule partie et une partie est composée d'une seule forge.
FactoryForge permet de créer la forge de la partie.

5. Gestion du temple



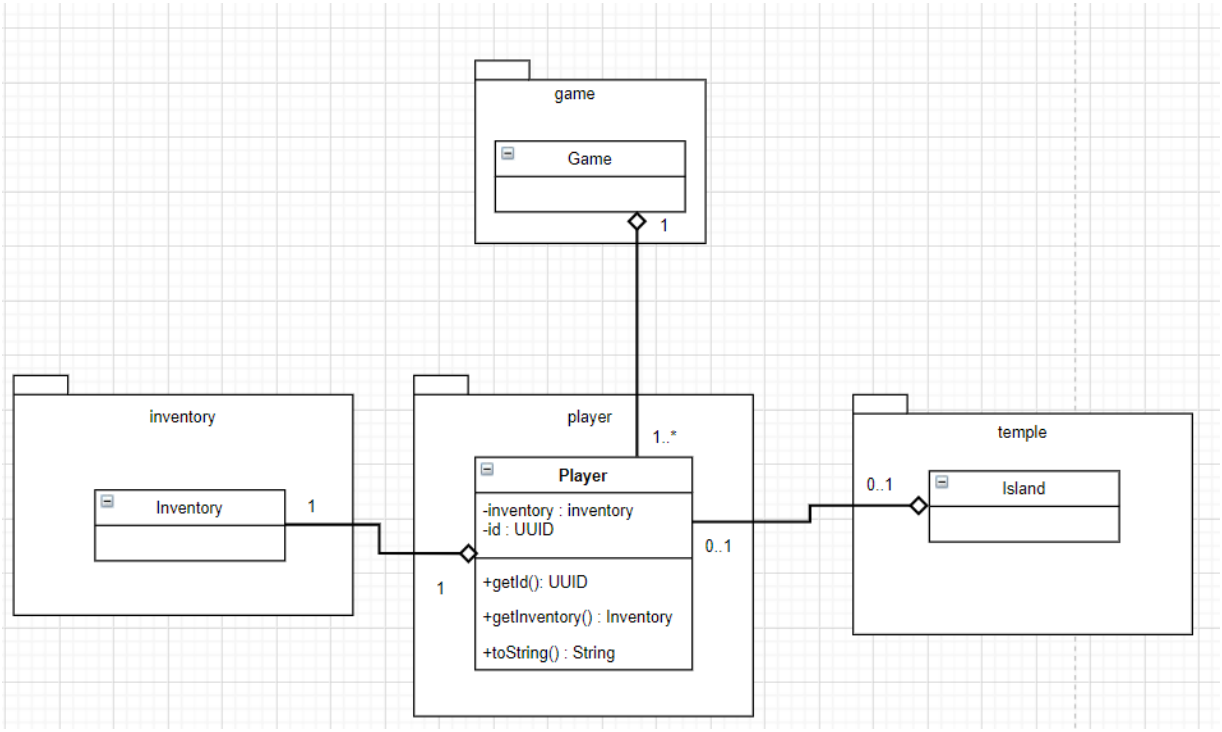
Un temple fait partie d'une seule partie et une partie possède un seul temple. Un temple est composé de plusieurs islands qui peuvent accueillir un seul joueur et sont composées de plusieurs exemplaires d'une carte.
FactoryTemple permet de créer le temple de la partie.

6. Gestion de l'inventaire



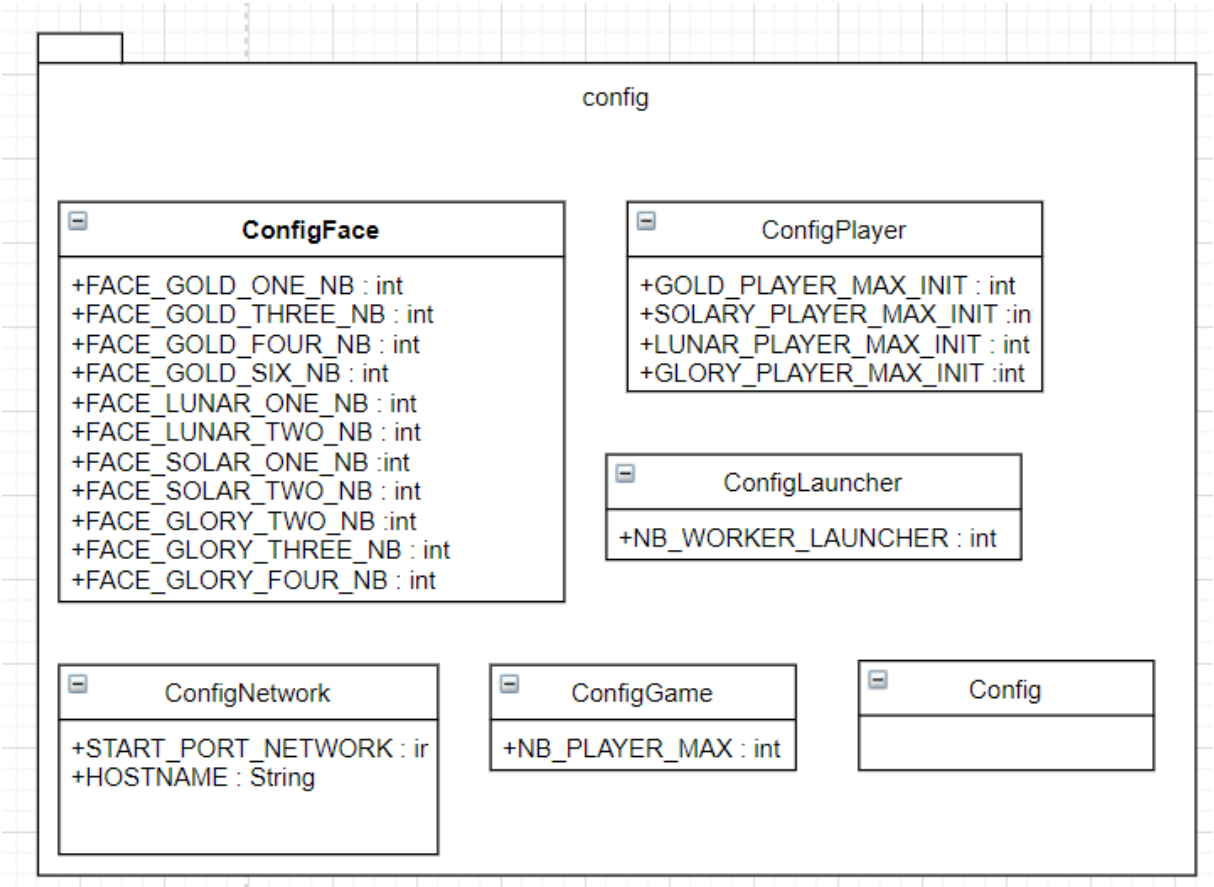
Un inventaire est obligatoirement relié à un joueur, un inventaire possède plusieurs types de ressources qui ont une valeur courante et une valeur maximale, l'inventaire est aussi composé de 2 dés, il peut contenir les cartes des exploits accomplies par le joueur.
FactoryInventory permet de créer l'inventaire d'un joueur.

7. Gestion du joueur



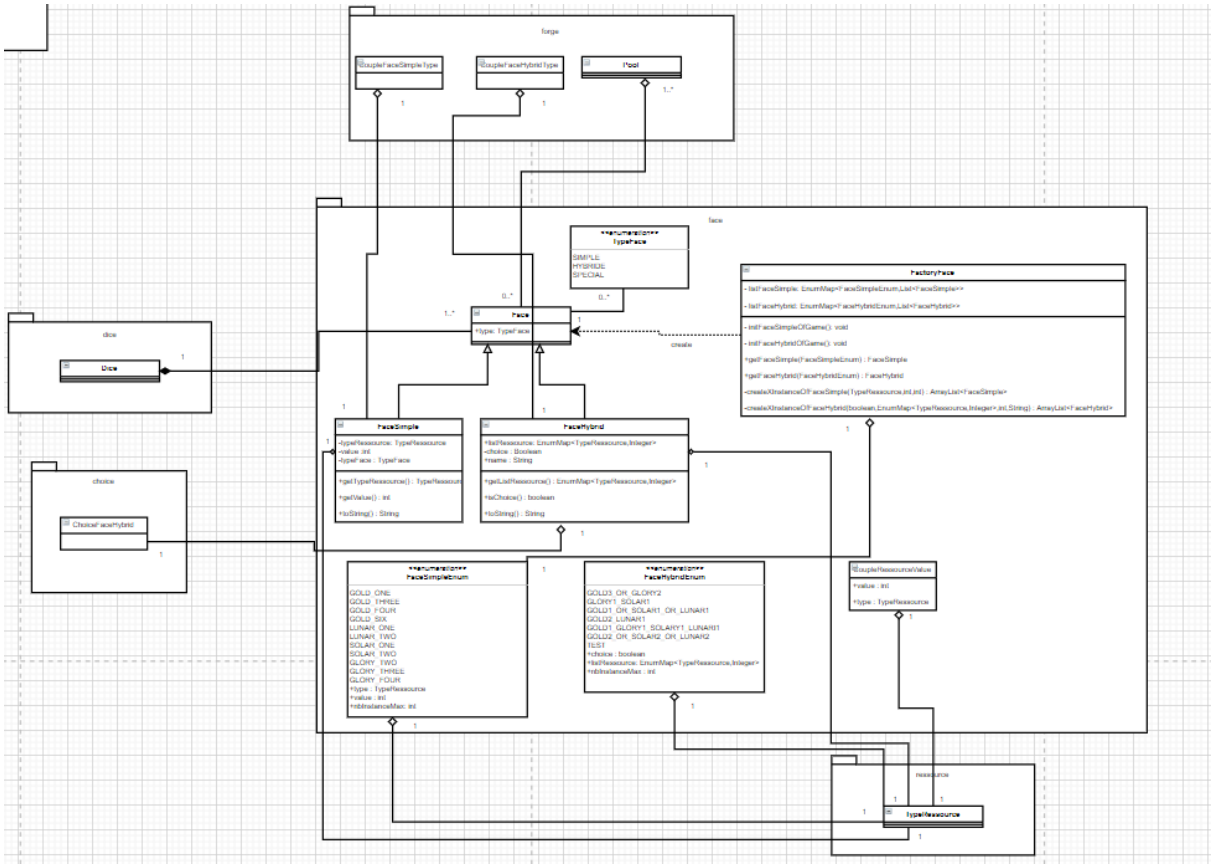
Un joueur possède un seul inventaire, s'il a effectué un exploit, celui-ci sera placé sur une île. Un joueur est connecté à une seule partie et une partie peut accueillir plusieurs joueurs.

8. Config



La config contient les différentes faces possibles, le nombre de joueurs max d'une partie, le nombre maximale des réserves de gold,lunar et solar et de glorypoint (point de victoire). La config contient aussi le port du réseau et le nom de l'hôte du réseau.

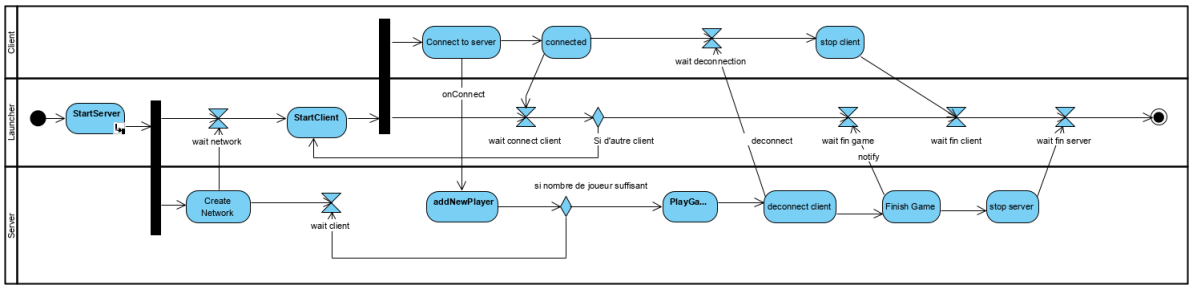
9. Gestion des faces



Toutes les faces sont composées d'un typeface d'un couplevaleur ou d'une liste de couplevaleur. Les faces hybrides obligent à effectuer un choix entre plusieurs typeressources. Une face est soit sur un seul dé soit dans une pool de la forge. FactoryFace permet de créer les faces d'un dé ou d'une pool.

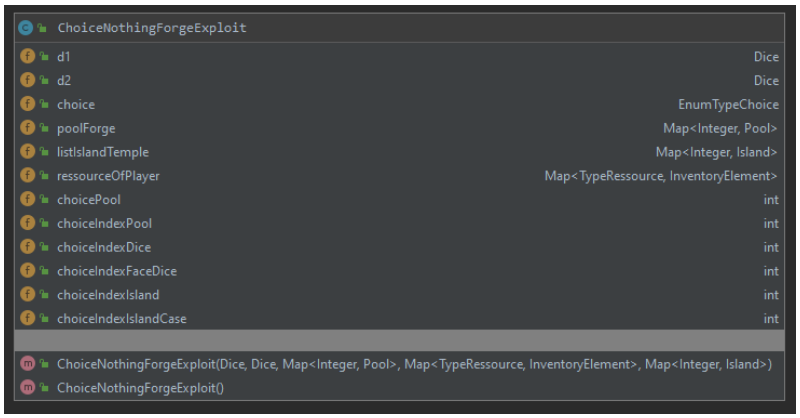
Interaction Client Serveur

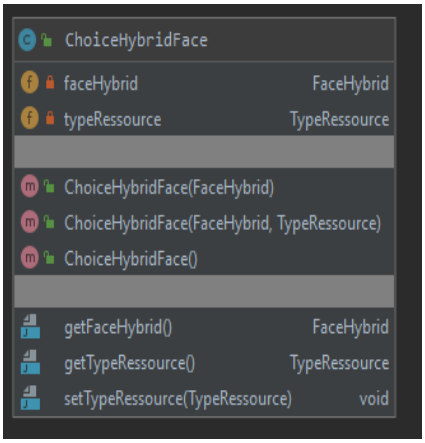
Interaction entre client, server, et launcher

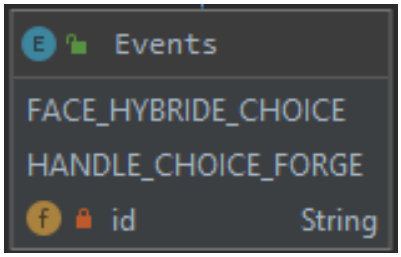


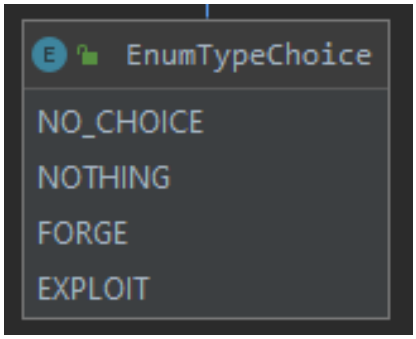
Ce diagramme représente les interactions entre les différents acteurs du système de la connexion à la fin de partie de façon global.

Objet reseau / protocole itération 3







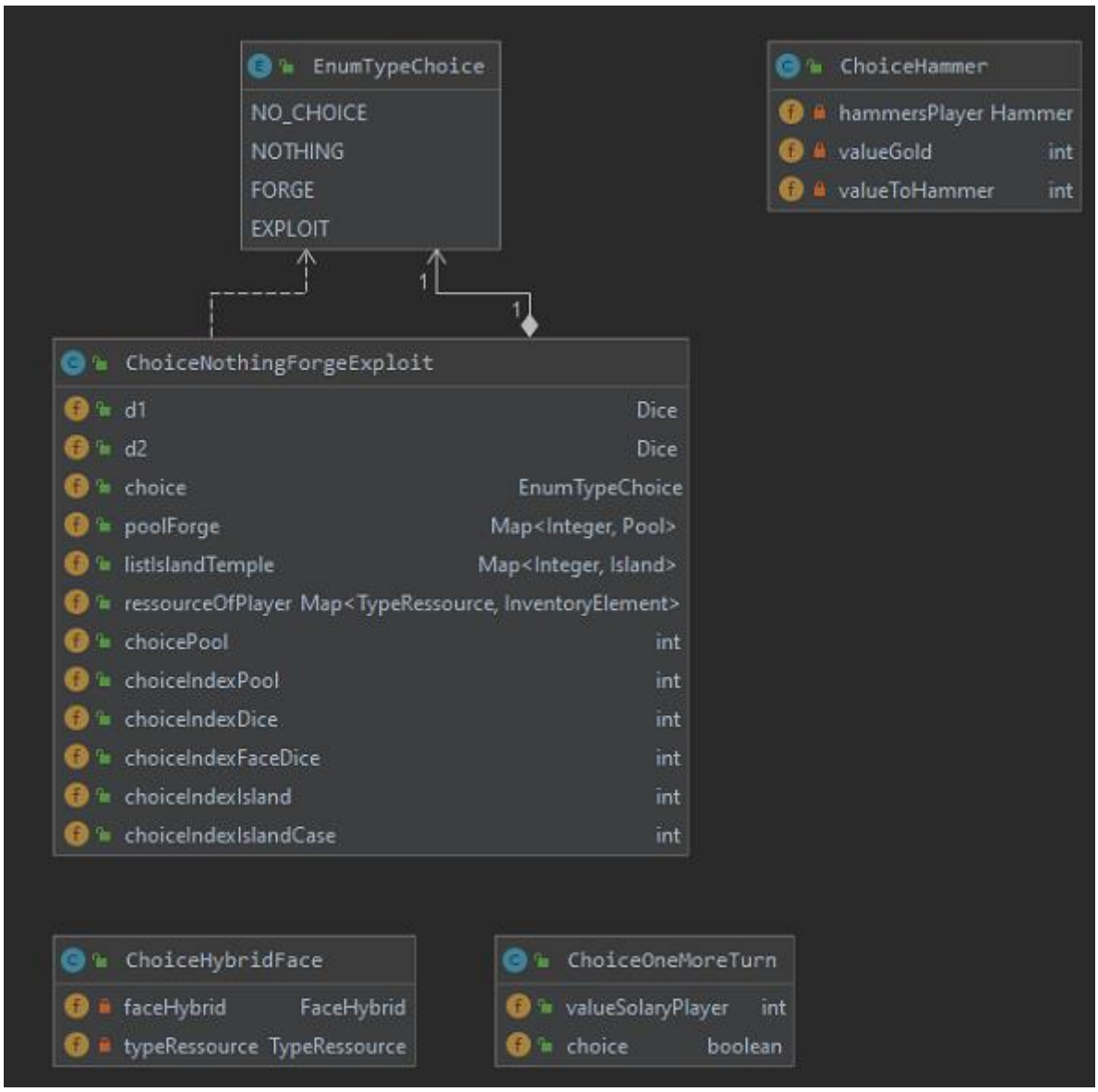


Enumération Events permet de renseigner tous les évènements réseaux entre le client et le serveur.

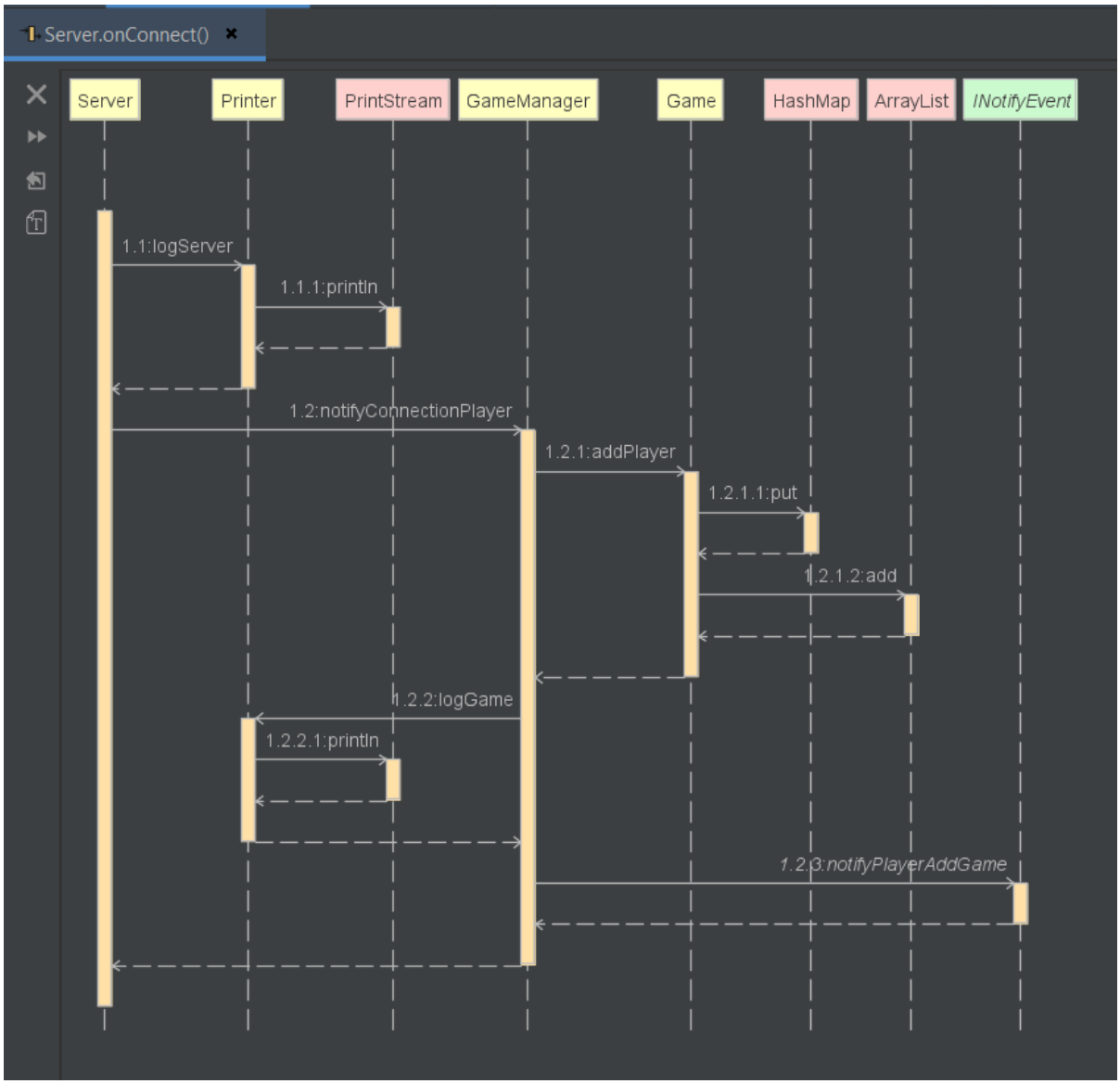
La communication entre le client et le serveur repose sur l'envoi d'object choice au format json. Un objet choice est composé d'une partie data et d'une partie result. Le serveur rempli la partie data et le client rempli la partie result et le renvoi au serveur. Le serveur lui extrait le résultat et effectue le traitement.

Objet reseau / protocole itération 5

Les nouveaux ajouts :

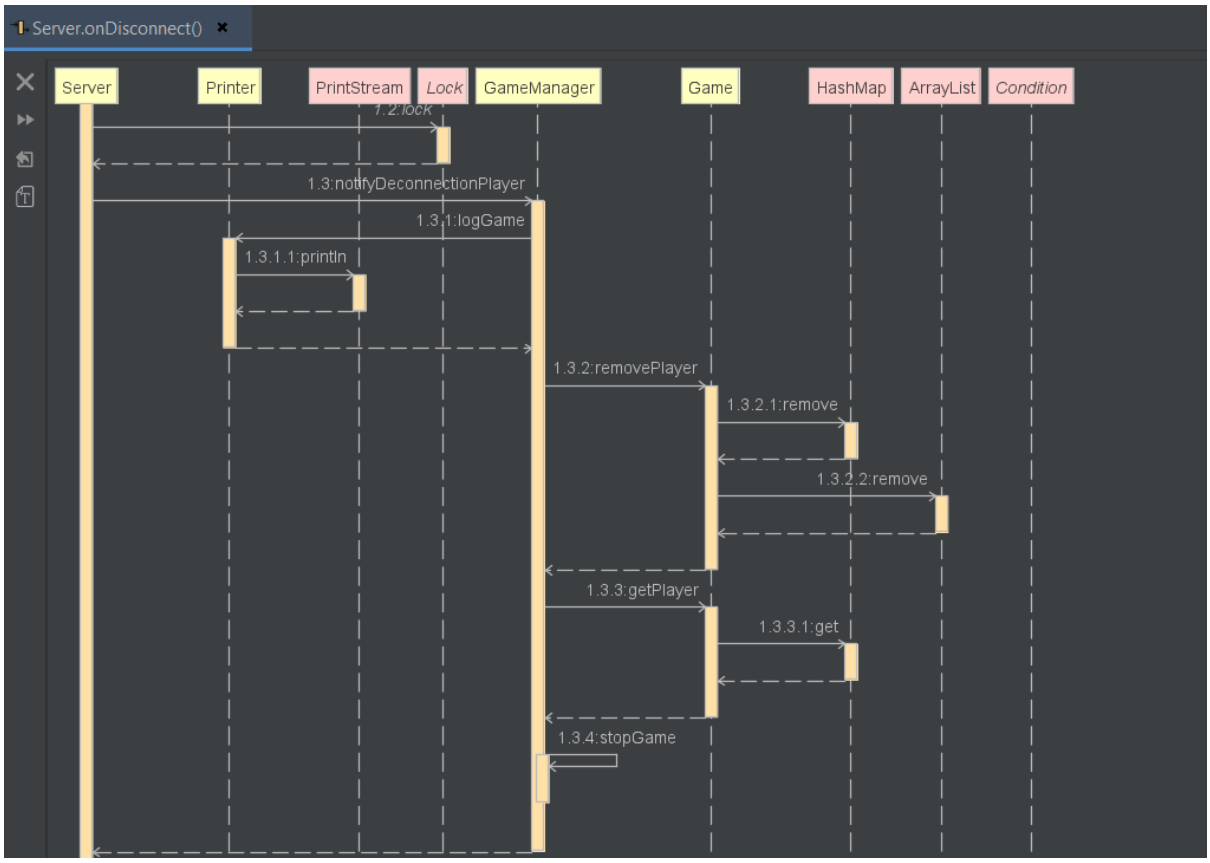


On Connect serveur



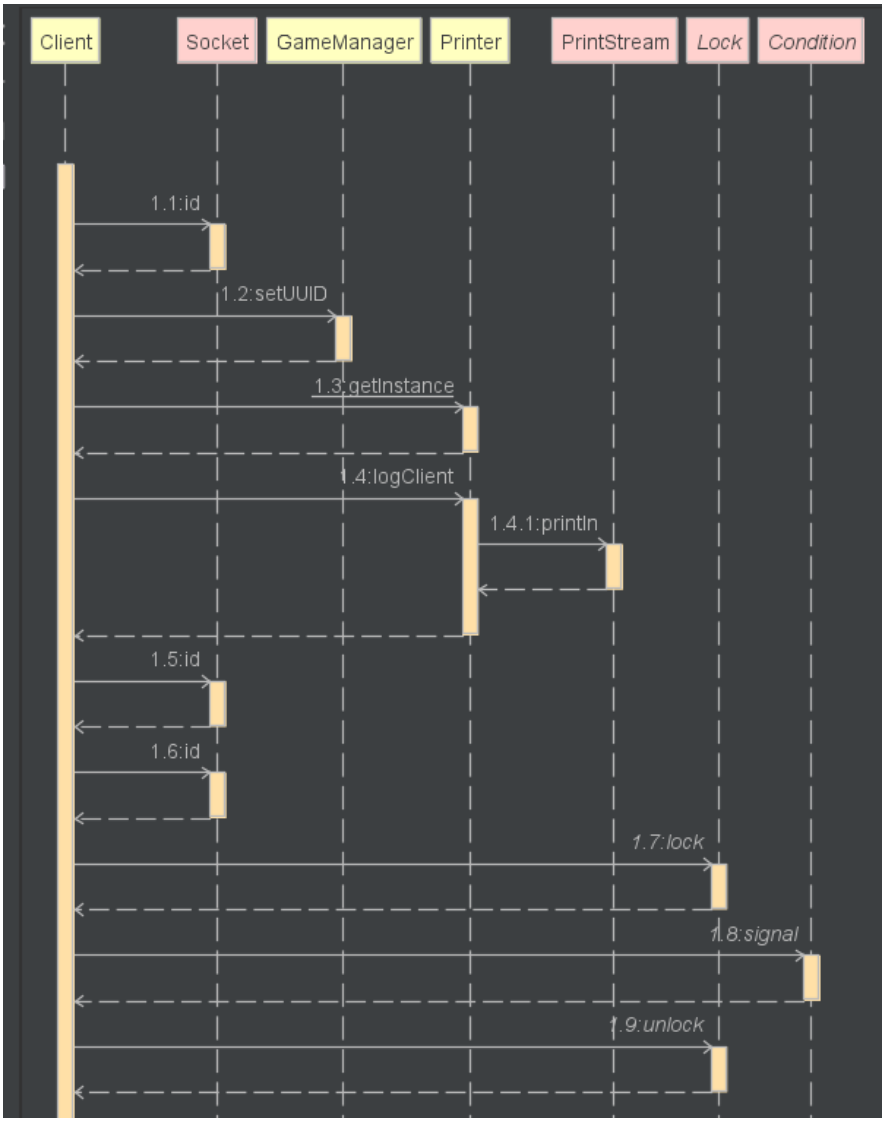
OnConnect est trigger par socket io lors d'une connexion client. Le serveur redirige la demande vers le gameManager qui s'occupe de créer le joueur et l'attacher a la partie. Lorsque le joueur à bien était ajouter il notifie le server du bon ajout.

On Disconnect Serveur

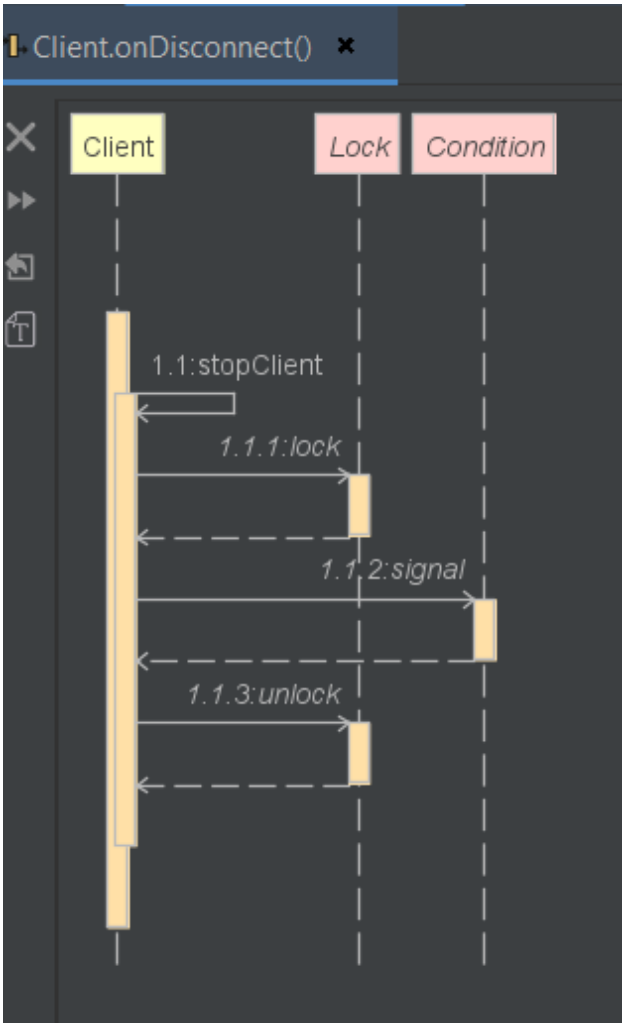


Lors de la déconnexion d'un joueur, nous ne gérons pas la reconnexion, le gameManager remove alors le joueur et stop la partie.

On Connect client

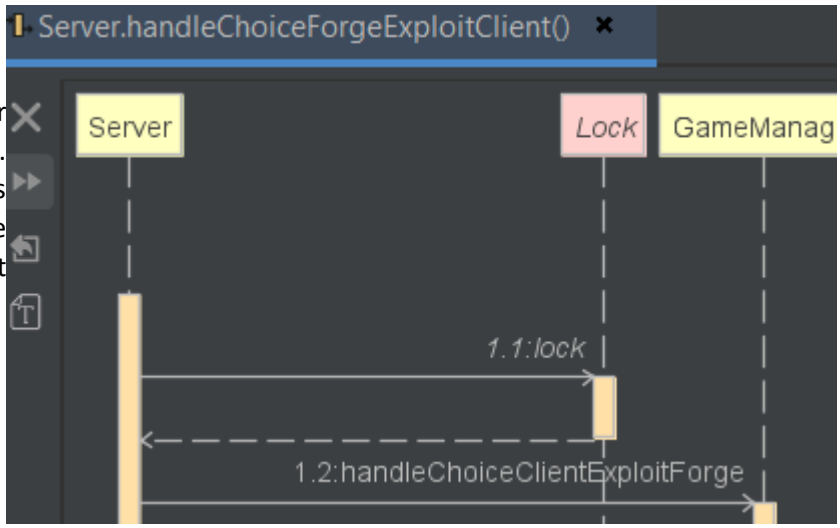


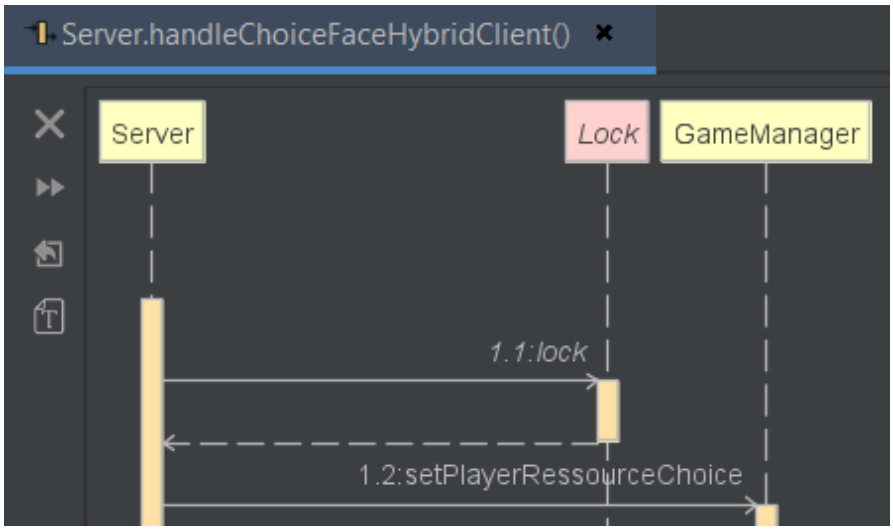
On Disconnect Client



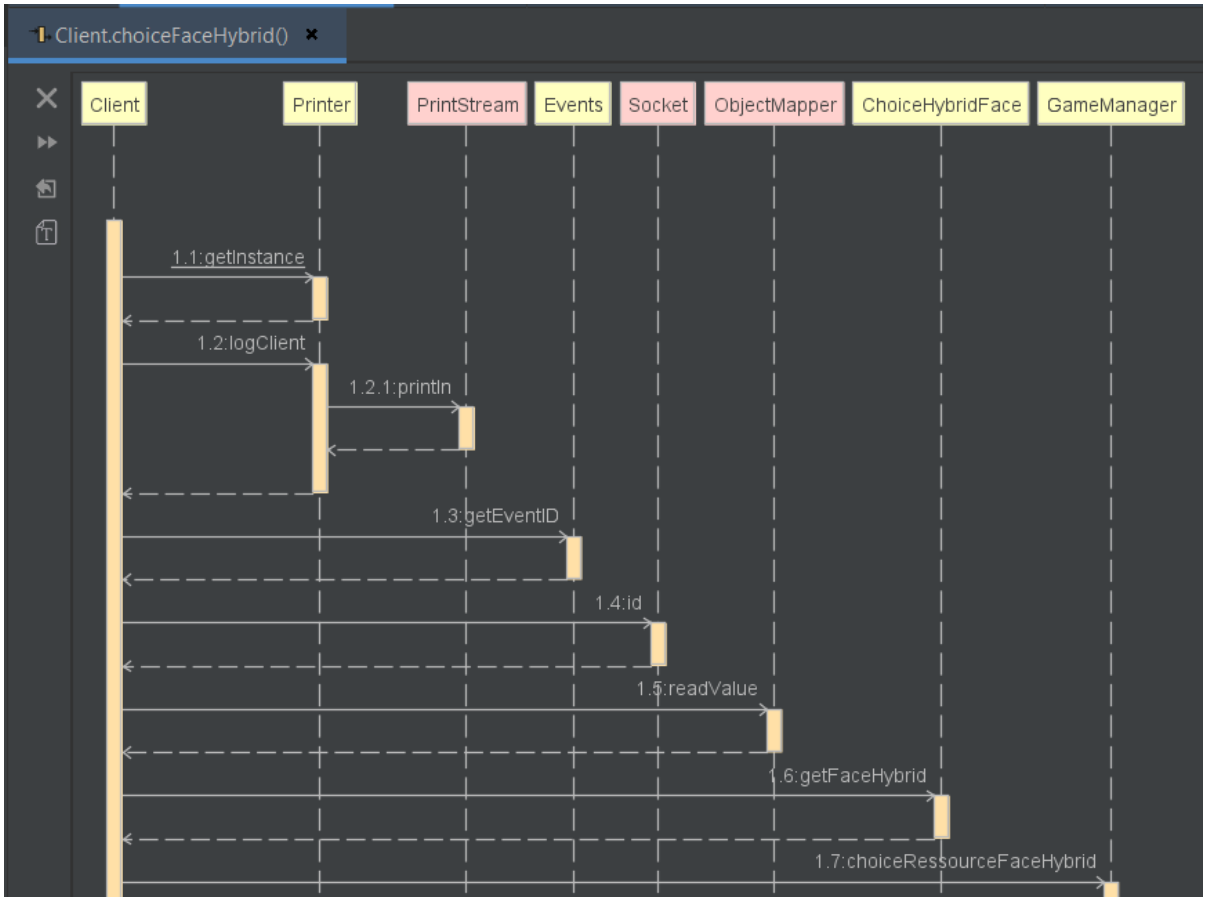
Handle Event Serveur

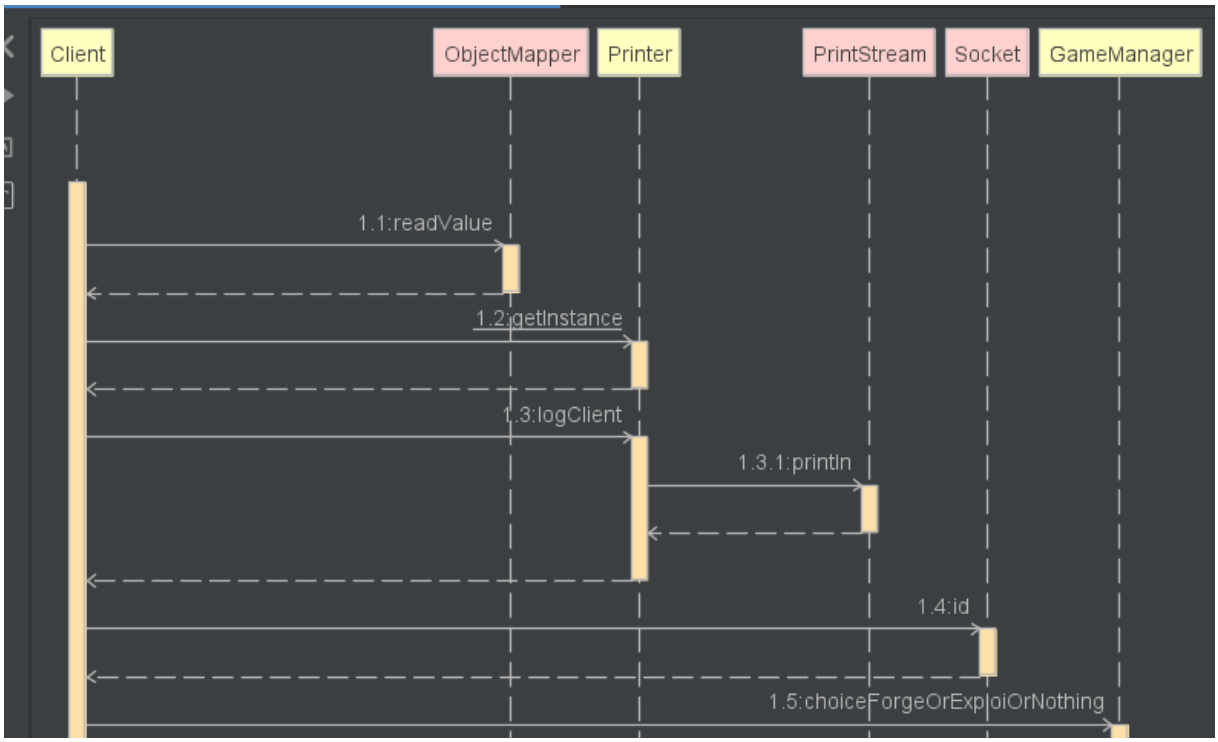
Les handlers d'event permette d'éxecuter le bon traitement lié a un event du réseau. Le Serveur alors route vers les bonnes mèthodes du gameManager. Chaque mèthodes trigger par le réseau est synroniser afin de garantir le Thread safe.





Handle Event Client




Client
ObjectMapper
Printer
PrintStream
Socket
GameManager
1.1:readValue
1.2:getInstance
1.3:logClient
1.3.1:println
1.4:id
1.5:choiceForgeOrExploiOrNothing

Conclusion itération 3

Analyse de votre solution : points forts et points faibles

Nos points forts d'après nous est l'architecture assez modulaire et flexible, ainsi que le nombre de fonctionnalités, réseau client server et multi partie.

Notre point faible est justement à cause d'un de nos points fort étant le nombre de fonctionnalités à concevoir au niveau de la conception qui requiert beaucoup de travail à réaliser.

Évolution prévue

Finir l'implémentation total des cartes ainsi que leurs effets et aussi les statistiques.

Conclusion itération 5

Lors de cette itération toutes notre conception n'a pas subi énormément d'évolution car nous étions déjà bien avancées dans le projet. Nous avons simplement amélioré tout le système de gestion d'vent de la partie vers un système de commandes.

Nous avons commencé à ajouter les actions des cartes comme le Hammer.

Analyse de votre solution : points forts et points faibles

Notre point fort maintenant est le système de commandes qui très efficace et très modulaire avec un système d'attente de choix client et l'implémentation d'une stack de commandes permettant une meilleur visualisation de l'enchainement des commandes. Le multi-partie à était amèliorer et optimiser pour gagner en performance.

Notre point faible est le nombre très importants de classes dans le projet mais qui je pense et nécessaire. Il n'est pas facile pour une personne extérieur de se reperer dans le projet.

Évolution prévue

Finir l'implémentation total des cartes basiques ainsi que les statistiques mais aussi l'implémentation d'un autre type de bot.