CONCEPTION D'ARCHITECTURES DISTRIBUÉES

DOSSIER D'ANALYSE-CONCEPTION POUR UN JEU DE BATAILLE NAVALE

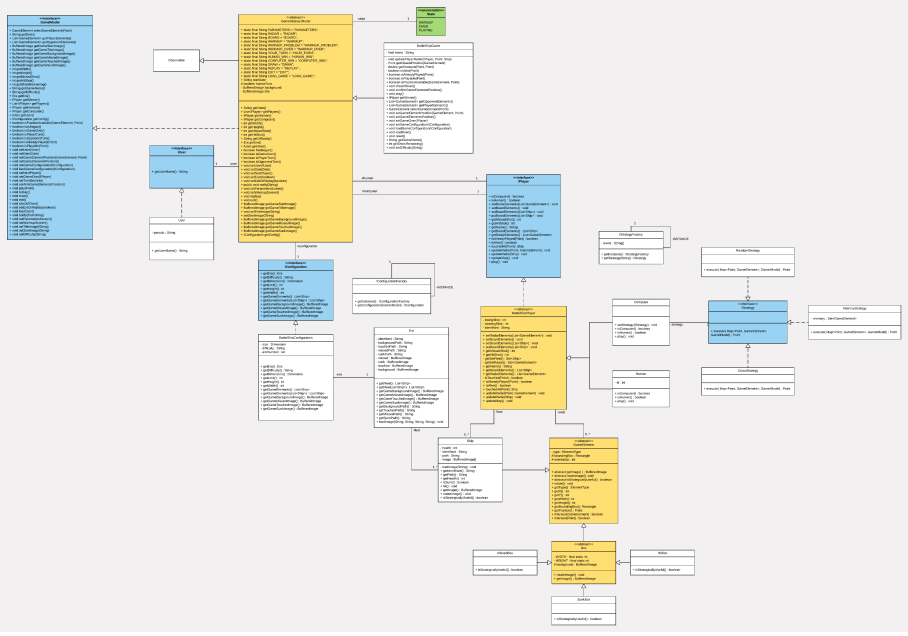
BELLEC Léopold
DAUZVARDIS Juozas
JUNGES Pierre-Marie

MASTER 1 INFORMATIQUE 2017

Table des matières

1 Notre choix de conception

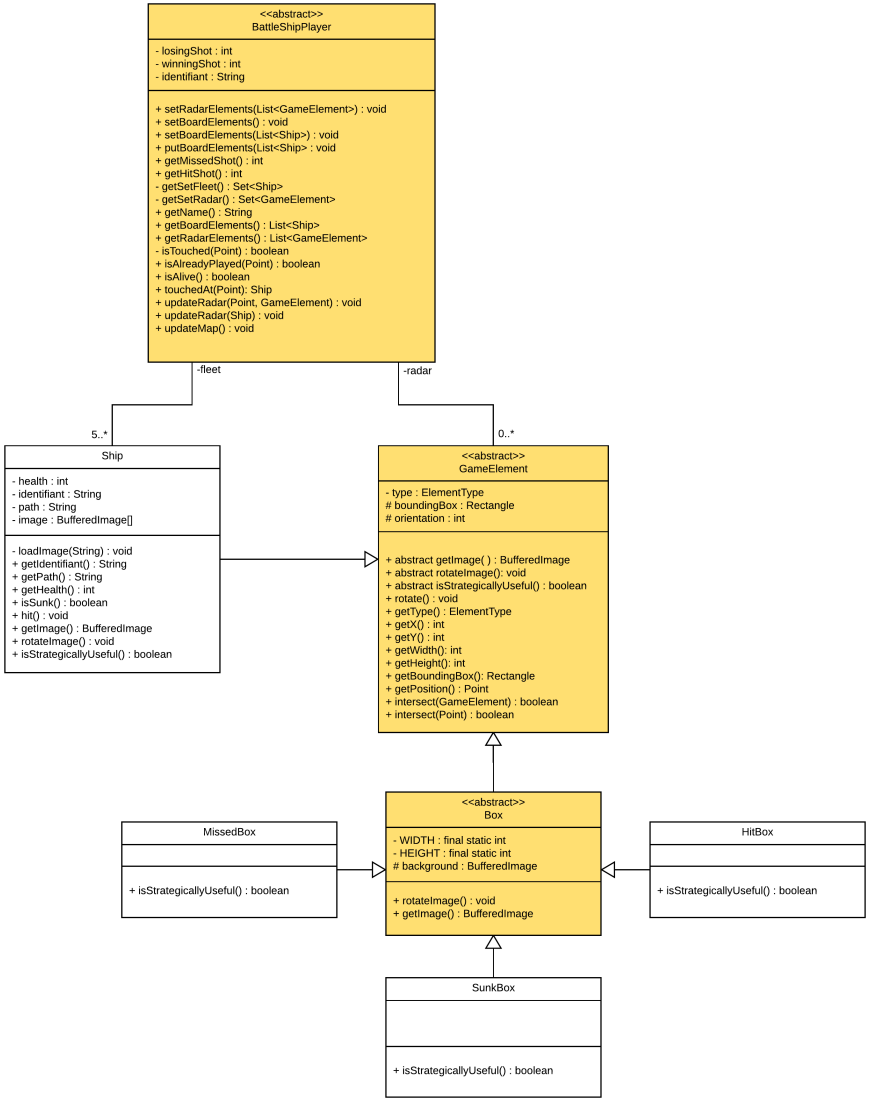
Figure 1: Vue d'ensemble des liens entre les éléments de notre jeu.



Par rapport à notre compte rendu initial, nous avons donc rajouté une "couche d'abstraction" à un peu près toutes les classes importantes du jeu. Notre diagramme de classe étant trés volumineux nous avons jugé préférable d'expliquer sa structure en expliquant un à un les packages et leurs rôles. Néanmoins voici une rapide introduction de chacun des packages (pour plus de détails, voir la section correspondante dans le rapport) :

- Le package **element** contient les élements liés au jeu, c'est à dire les bateaux, les box etc.
- Le package **player** contient les classes censées représenter l'utilisateur (et aussi son adversaire) dans le jeu.
- Le package **game** regroupe toutes les classes propres aux mécanismes du jeu, on y retrouve notamment la classe **GameAbstractModel** qui nous sert de modèle.
- Le package **storage** rassemble les classes de type DAO.
- Le package **graphics** stocke les classes liées à l'interface graphique ainsi que les listeners associés.

1.1 Explication du package element



Comme le diagramme l'indique, nous avons décidé de stocker dans la classe **BattleShipPlayer** une Map pour gérer les **Ship** et une autre Map de **GameElement** afin de gérer les coups ratés (**MissedBox**), touchés (**HitBox**), ou bien de savoir si le bateau est coulé (**SunkBox**).

Ensuite, nous avons créé une classe abstraite **GameElement** afin de factoriser de nombreuses lignes de code communes car les classes **Ship** et **Box** ont quelques propriétés communes. De plus cette classe nous permettra au moment de dessiner les **Ship** et **Box** de regrouper tout cela dans une même liste (ou toute autre structure de données). Notre fonction de dessin sera donc une simple itération où l'on demandera à dessiner chacun des éléments de cette liste.
C'est pourquoi notre choix de 2 Maps rend les choses plus aisées. En effet si un **IPlayer** souhaite tirer à une position *p* alors il suffira de regarder si dans la Map fleet du **IPlayer** adversaire existe une clé *p*, et si c'est le cas nous pouvons donc directement travailler avec un objet de type **Ship**. Et lui faire subir des dommages etc. Parlons maintenant de la deuxième Map que possède le **IPlayer**, la Map de **GameElement**. Elle nous permet de savoir rapidement si le **IPlayer** a déjà fait une tentative (donc un clic de souris) pour une position *p* donnée.

Mais, n'aurait-il pas été plus simple de travailler avec des cases "bateau" par exemple ?

C'était effectivement une des possibilités, sauf que, étant donné que nous avions déjà effectué un jeu 2D en Java (Space Invader), et que, comme tout bon programmeur les efforts inutiles sont à proscrire. Nous avons donc jeté un coup d'oeil à notre implémentation graphique, et plus particulièrement sur la fonction *paint(Graphics g)*, que nous avons jugé préférable de réutiliser ainsi que d'autres fonctions bien sûr.
Maintenant supposons que nous avions fait des classes type **BoxY**, avec **Y** n'importe quels éléments possibles (bateau, avion, vide etc.). Alors si nous avions un élément qui tenait sur plusieurs cases, un porte-avion par exemple (dans "Bataille navale", il fait 5 cases), nous aurions dû faire en sorte de stocker sur chacune des 5 **BoxShip** une des textures du porte-avion correspondante, et faire attention aux erreurs dans le cas où le **IPlayer** fait faire une rotation à cet élément.

Alors que dans notre cas, nous utilisons directement les fonctions *drawRect*, *drawImage* que nous offre Java. Dans notre implémentation un **Ship** à une position ainsi qu'une largeur et une longueur, donc un seul appel à *drawImage* suffit à dessiner le bateau sans se soucier d'autre chose. De même, l'opération de rotation est également simplifiée car dans notre implémentation nous avons juste à faire une rotation sur une seule image et le tour est joué.

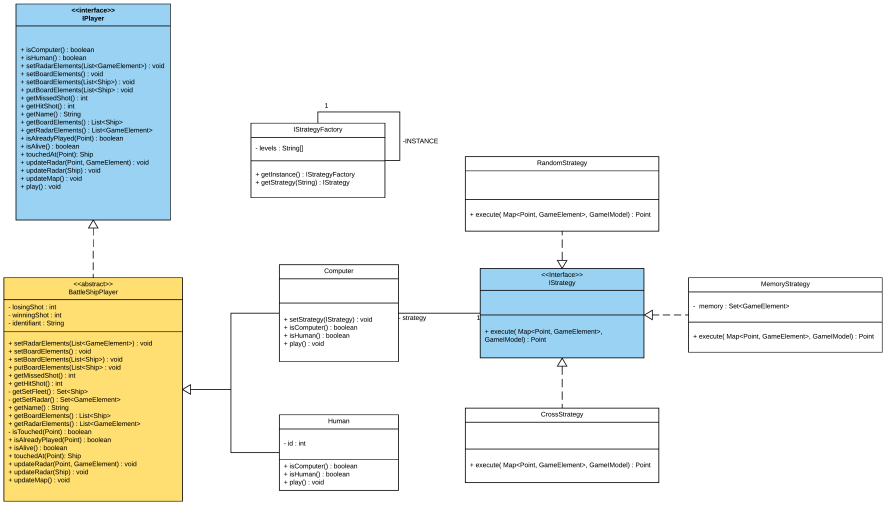
Dessiner une **Box** dans les deux implémentations ne pose pas de réel problème.
Expliquons maintenant la différence entre les 3 types de **Box** disponibles :

- **MissedBox** permet de modéliser un tir raté, donc un clic souris sur une case ne contenant pas de **Ship**.

- **HitBox** permet de modéliser un tir réussi, le clic souris a touché un **Ship**, cependant le **Ship** n'est pas encore coulé.

- **SunkBox** permet de modéliser un **Ship** coulé. Elle permet également d'indiquer au joueur si par exemple nous avons 5 **HitBox** d'affilés si elles representent 1 **Ship** de 5 cases ou bien un **Ship** de 3 cases et un de 2 cases.

1.2 Explication du package player



De facon à obtenir un code flexible et permettant une certaine évolutivité, nous avons donc créé une interface **IPlayer** contenant des méthodes que nous avons jugé utiles pour un jeu(humain versus ordinateur, où il y aurait 2 fenetres de jeu). Et, comme notre objectif est de faire une bataille navale, nous avons donc créé une classe abstraite **BattleShipPlayer** implémentant notre précèdente interface. Donc grace à cette structure nous pouvons dans la classe **BattleShipPlayer** modifier les méthodes disponibles afin de les faire fonctionner sur un jeu type bataille navale.

De plus comme notre jeu comporte 1 **Human** et 1 **Computer** ayant des comportements différents (notamment à cause de l'IA) nous avons donc créé ses 2 classes supplémentaires. La différence entre un **Human** et un **Computer** se trouve dans la façon de jouer.
En effet, un joueur **Human** aura choisi sa **Box** via un **MouseListener** par exemple, alors que le joueur **Computer** va avoir une **IStrategy** qui va choisir une coordonnée à frapper. En implémentant de cette manière cela nous permet de rendre flexible le "raisonnement" de l'ordinateur, par exemple il est aisé de changer de stratégie en cours de partie.
De plus, comme demandé dans le sujet, la classe **BattleShipPlayer** permet de stocker les coups ratés *losingShoot* et les coups gagnants *winningShot*.

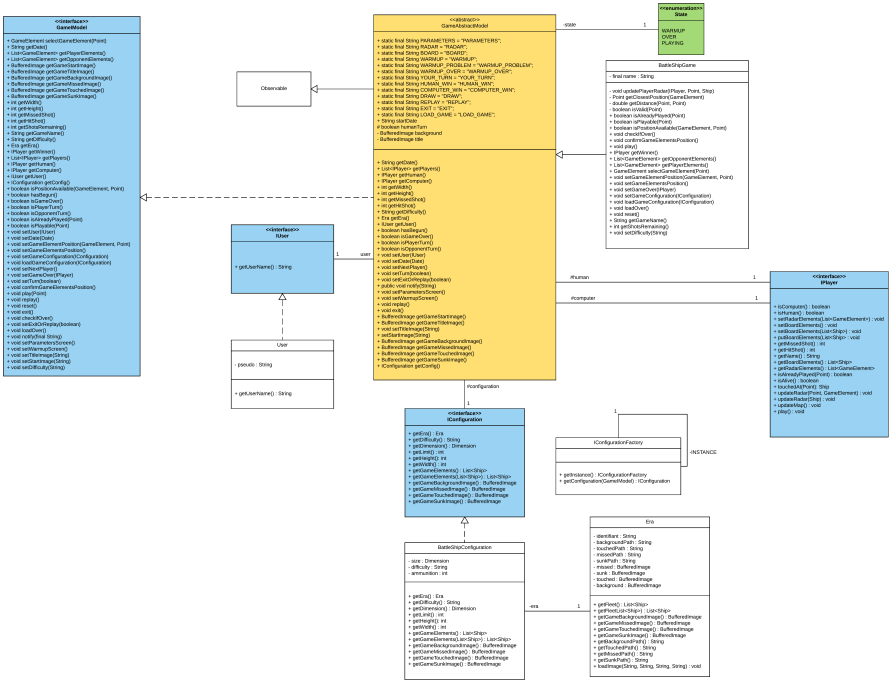
Parlons maintenant des **IStrategy**, nous en avons implémenté 3 correspondant ainsi aux fameux 3 niveaux de difficultés : facile, normal, difficile.

- **RandomStrategy** correspond à une IA facile, tirant aléatoirement.
- **CrossStrategy** correspond à une IA normal, tire en croix.
- **MemoryStrategy** correspond à une IA difficile, tire aléatoirement puis si une case est touchée alors cette stratégie va tirer à proximité jusqu'à faire couler le

Ship.

Si l'on souhaite changer cette correspondance : niveau de difficulté $\longleftrightarrow$ **IStrategy**, il suffit de modifier la classe **StrategyFactory**. En effet, cette factory nous permet lorsque le joueur sélectionne une niveau de difficulté (sous forme de texte donc) de lui affecter une **IStrategy** correspondante.

1.3 Explication du package game



Pour ce package, nous avons décidé d'utiliser la même logique que pour le package *player*.Nous avons donc créé une interface **GameIModel** contenant (pour nous) les méthodes nécessaires au bon fonctionnement d'un jeu, puis nous avons, pour des soucis de factorisation de code créé une classe abstraite **GameAbstractModel**.

Cette dernière a également pour rôle la gestion des vues, car elle hérite de la classe **Observable** ; d'où le fait qu'elle possède de nombreux attributs static final String permettant de mettre à jour de manière plus aisée la(les) vue(s).

Puis, nous avons créé la classe **BattleShipGame** héritant de **GameAbstractModel**.

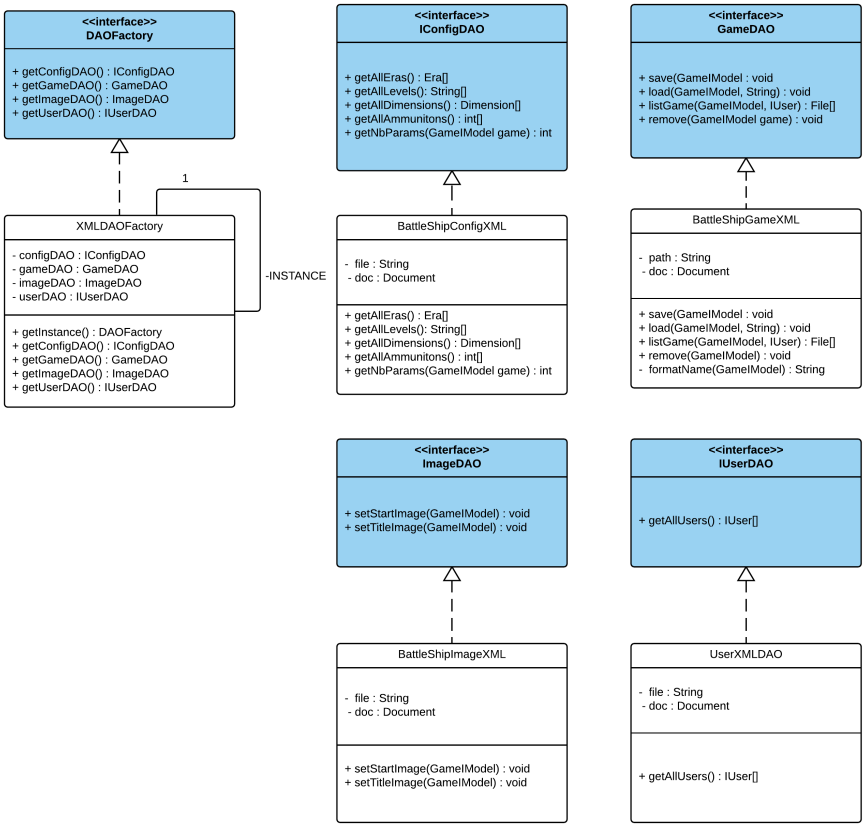
Donc dans **BattleShipGame** nous avons surcharger les méthodes disponibles de la classe héritée afin de respecter les règles du jeu bataille navale.
Donc si nous avons dans le futur, besoin d'implémenter un autre jeu semblable à la bataille navale, il suffira de créer une classe et de la faire hériter de **GameAbstractModel**. De plus, la classe **GameAbstractModel** connait également 1 **IConfiguration**. Cette interface nous permet d'avoir des méthodes pour récupérer ou affecter des paramètres tels que la difficulté ou bien de gérer des images. De même, afin de

garantir une certaine flexibilité, nous avons introduit la classe **BattleShipConfiguration** héritant de **IConfiguration**, qui, comme son nom l'indique, gère les paramètres d'une partie de bataille navale.

Dans notre cas, les paramètres nécessaires à une partie de bataille navale sont : la difficulté, la dimension du plateau de jeu, les munitions disponibles et l'**Era** choisi. Maintenant, parlons de l'**Era**, cette classe nous permet d'obtenir une liste de **Ship** ainsi que les images de fond, de cases touchées, ratées, ou bien les images des cases où se trouve un **Ship** coulé.

Notre package contient également la classe **IConfigurationFactory**, nous permettant d'obtenir de manière fiable une seule **IConfiguration** pour un seul **GameIModel** donné.

1.4 Explication du package storage



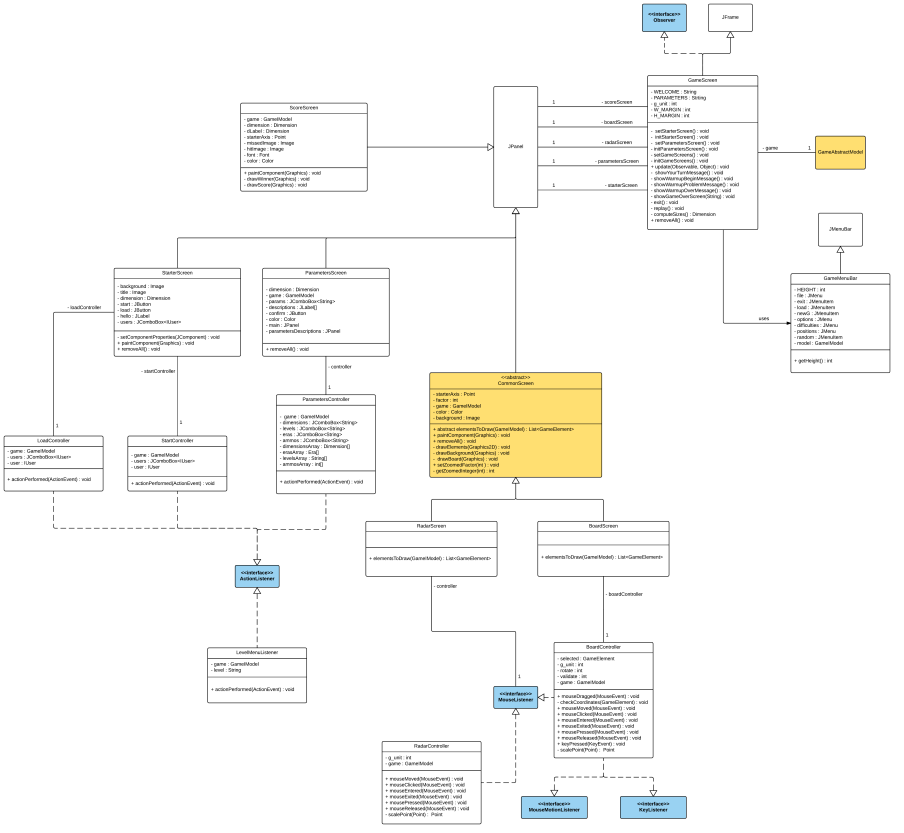
Nous avons décidé de sauvegarder ou charger sous forme XML, donc tout d'abord nous avons introduit une interface **DAOFactory** afin d'obtenir tous les autres DAOs nécessaires.

Puis nous avons créé une classe **XMLDAOFactory** permettant de récupérer les DAOs liés au format XML.

Ensuite, afin de sauvegarder l'ensemble des éléments du jeu, nous avons décidé de découper toute cette masse de donnée en 4 DAOs, un s'occupant de récupérer les images liées au jeu **ImageDAO**, un s'occupant de charger les **IUser** intitulé **IUserDAO**, un autre **IConfigDAO** s'occupant de charger les éléments nécessaires à instancier une **IConfiguration**.

Puis le dernier et le plus important DAO, **GameDAO**, s'occupant de charger une partie d'un **GameIModel** ou bien de sauvegarder une partie de **GameIModel**.

1.5 Explication du package graphics



Ce package va contenir les classes liées à notre interface graphique.

GameScreen va nous servir de fenêtre principale, car il va s'agir d'une **JFrame**, de plus elle sera mise à jour par le **GameAbstractModel** du fait qu'elle hérite de **Observer** et qu'elle ajoute le **GameAbstractModel** comme étant son modèle. Afin de dessiner le jeu de manière simple et efficace, nous avons jugé bon de diviser en 2 classes distinctes la fenêtre principale.

1. **RadarScreen** va afficher les **Box** touchées ou ratées du joueur **Human**.
2. **BoardScreen** va afficher les **Ship** du joueur **Human** et les **Box** touchées ou ratées du joueur **Computer**.

Ces deux classes héritent de **CommonScreen**, cela nous permet factoriser plusieurs méthodes qui étaient communes et de dessiner les éléments du jeu facilement.

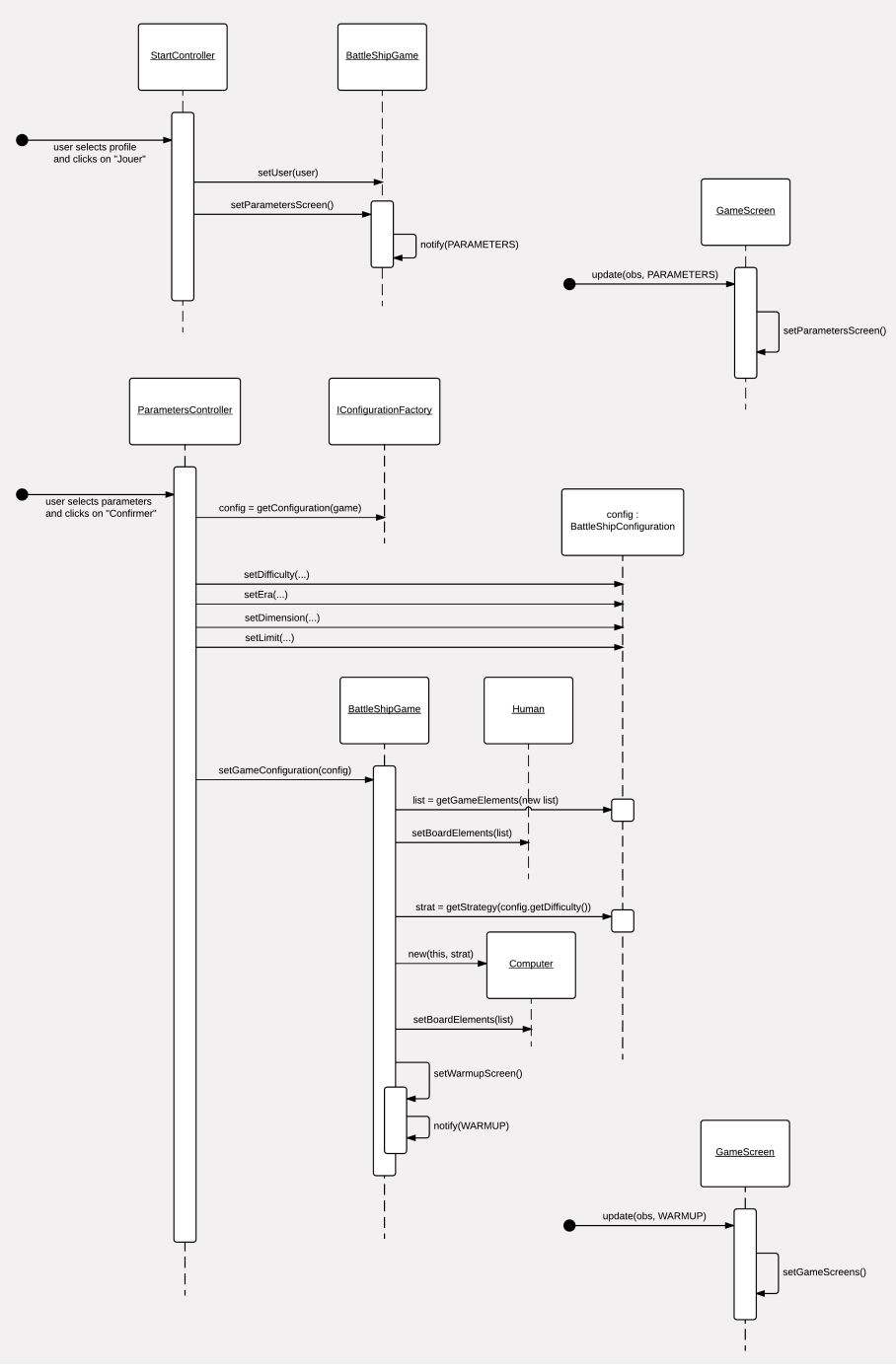
De plus au lancement la classe **GameScreen** contiendra **StarterScreen** qui va permettre à l'utilisateur de décider s'il veut commencer une nouvelle partie ou bien charger une ancienne. Ces deux précèdentes actions sont possibles grâce à l'utilisation des listeners **StartController** pour commencer la nouvelle partie, et **LoadController** pour en charger une ancienne. Une fois que l'utilisateur aura décidé de commencer une nouvelle partie, le **GameAbstractModel** demandera à la classe **GameScreen** de se mettre à jour et donc d'afficher le contenu de la classe **ParametersScreen** qui comme son nom le laisse supposer, permettera à l'utilisateur de sélectionner les paramètres de sa future partie. Puis lorsque l'utilisateur souhaitera confirmer son action, la classe **ParametersController** rentrera en action, afin d'avertir le **GameAbstractModel** qu'une nouvelle partie avec ces paramètres doit s'afficher.

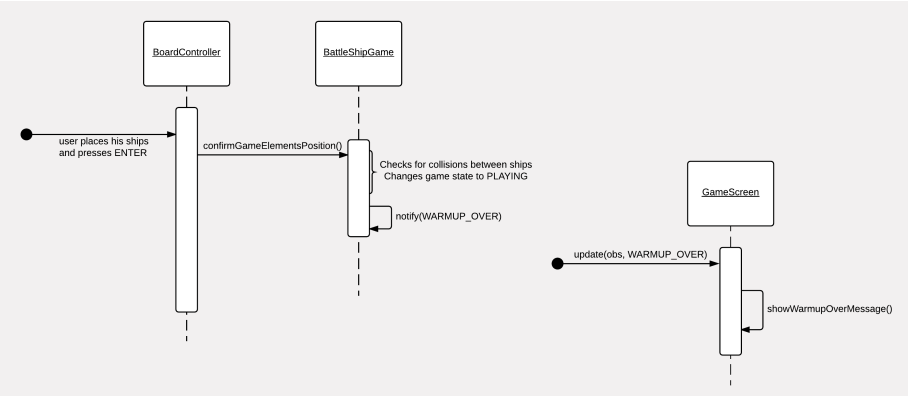
Nous avons également mis à disposition une JMenuBar **GameMenuBar** permettant ainsi à l'utilisateur d'effectuer plusieurs actions telles que :

- Charger une ancienne partie
- Démarrer une nouvelle partie (l'utisateur sera rediriger vers le **ParametersScreen**
- Changer la difficulté du joueur adversaire

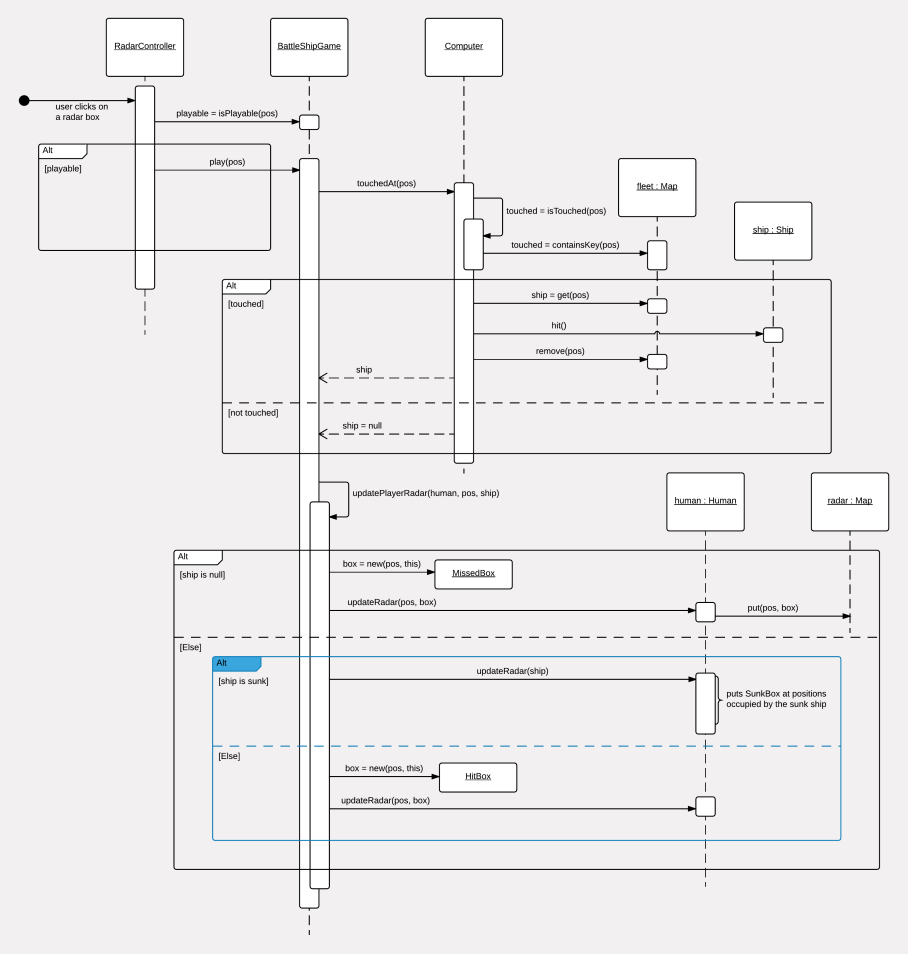
2 Diagrammes de séquence

2.1 Lancer une partie




BoardController
BattleShipGame
user places his ships
and presses ENTER
confirmGameElementsPosition()
Checks for collisions between ships
Changes game state to PLAYING
notify(WARMUP_OVER)
GameScreen
update(obs, WARMUP_OVER)
showWarmupOverMessage()

2.2 Tirer


RadarController
BattleShipGame
Computer
user clicks on
a radar box
playable = isPlayable(pos)
Alt
[playable]
play(pos)
touchedAt(pos)
fleet : Map
touched = isTouched(pos)
ship : Ship
touched = containsKey(pos)
Alt
[touched]
ship = get(pos)
hit()
remove(pos)
ship
[not touched]
ship = null
updatePlayerRadar(human, pos, ship)
human : Human
radar : Map
Alt
[ship is null]
box = new(pos, this)
MissedBox
updateRadar(pos, box)
put(pos, box)
[Else]
Alt
[ship is sunk]
updateRadar(ship)
puts SunkBox at positions
occupied by the sunk ship
[Else]
box = new(pos, this)
HitBox
updateRadar(pos, box)

2.3 Dessiner les élements