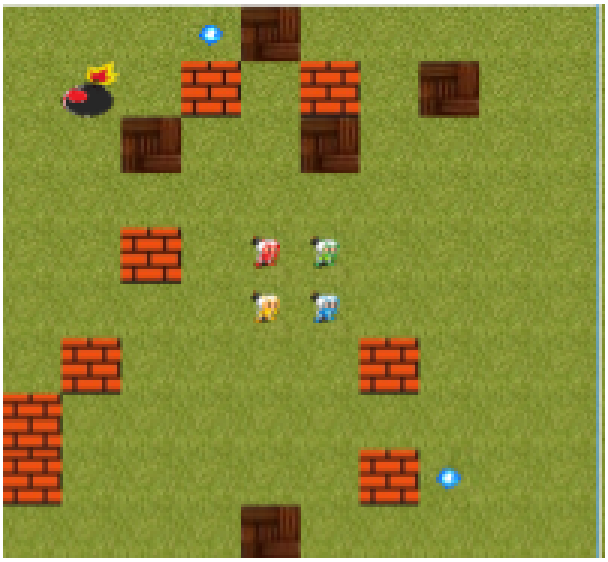
MÉTHODES DE CONCEPTION

DEVOIR DE CONTRÔLE CONTINU

Rapport de projet



Gonzalez Antoine 21504712
Erwann Leroux 21500894
Mamadou Aliou Bah
Mamadou Lamara Bah

L3 Informatique Td B B2
Année : 2018-2019

UNIVERSITÉ CAEN NORMANDIE

Table des matières

1 Introduction

Dans le cadre du module "méthodes de conceptions",il nous a été demandé de réaliser un projet java. Le sujet du projet nous demande de réaliser un jeu de plateau 2D dans lequel nous confrontons 4 combattants dans un format tour par tour. Chaque combattant a pour but d'être le dernier survivant, pour cela il dispose de divers objets qui lui permettent d'attaquer un autre combattant ou de se défendre (Bombe,bouclier,mine...). Le but de se projet est de nous faire réutiliser les divers patrons de conception étudiés en cour. Par mis ces "patterns" on retrouve le pattern MVC (modèle, vue et contrôleur) que nous avons étudié l'année dernière.Celui-ci nous posant comme contrainte d'avoir un modèle indépendant de la vue. Dans un premiers temps nous allons vous présenter notre conception logiciel avec notamment une présentation des différents pattern utilisé. Puis, nous présenterons le fonctionnement de certain algorithmes et terminerons avec une présentation du fonctionnement de l'application.

Remarque : Dans la partie conception nous ne parlons pas de la partie compréhension du sujet et mise en place des tâches à réaliser car elles sont explicites dans le sujet du devoir.

2 Conception de l'application

2.1 Architecture de l'application

Ce projet java présente une base de fichiers et de ressources assez volumineuse. A fin de pouvoir utiliser et comprendre au mieux le contenue de notre application, il est important de concevoir une architecture logique et segmentée. Nous allons dans cette partie vous présenter l'architecture mise en place.

A la racine de notre application on trouve 4 répertoires :

- un répertoire **src** contenant le code commenté
- un répertoire **doc** contenant la Javadoc
- un répertoire **dist** contenant l'exécutable (un fichier .jar, et d'éventuelles ressources nécessaires à l'exécution)
- un répertoire **libs** contenant les différentes librairies utilisées
- un fichier **build.xml** permettant de re-générer le contenu de dist via ANT.

Dans le répertoire src on trouve différents "packages" contenant notre code. Voici comment nous les avons organisés :

- le package **models** contient les fichiers permettant de générer et manipuler les données de l'application. Il se décompose des "sous-packages" suivant :
  - **models.element** : contient les classes permettant de représenter les éléments de notre plateau de jeu (joueur , mur , bombe ...)
  - **models.strategies** : contient les classes permettant d'utiliser le patron de conception "Strategy"
  - **models.factories** : contient les classes permettant d'utiliser le patron de conception "Factory"
  - **models.gamestate** : contient les classes permettant de représenter un état du jeu, autrement dit il contient la base du modèle de l'application

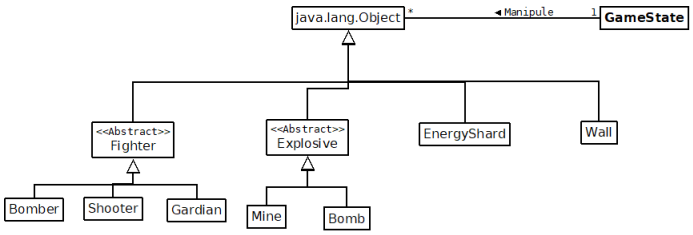
- **models.utils** : contient les classes nous permettant d'avoir des fonctionnalités additionnelles ou utilitaires, on y trouve par exemple la classe **SoundPlayer** permettant de jouer un son,ou bien encore la classe **Coord** qui nous permet de représentant simplement un couple d'entier x,y.

— le package **views** contient les fichiers permettant de générer l'affichage de notre application, c'est à dire la partie graphique

— le package **controllers** contient les fichiers en rapport avec le contrôleur de notre architecture MVC, il est donc chargé de la partie pris de décision de notre application

2.2 Représentation de notre plateau de jeu

Le modèle de notre jeu repose essentiellement sur la représentation du plateau de jeu où vont s'affronter les différents combattant. Cette grille doit pouvoir être remplit de différentes entités comme par exemple des murs , des combattants , des bombes , des pastilles d'énergies ... C'est pourquoi nous avons choisis de représenter notre plateau de jeu par une **grille d'Objets : Object[taille][taille]**. Cette grille recevra des objets de différents types, qui sont défini dans le package **models.element**. Le diagramme ci-dessous représente bien de quoi on parle :



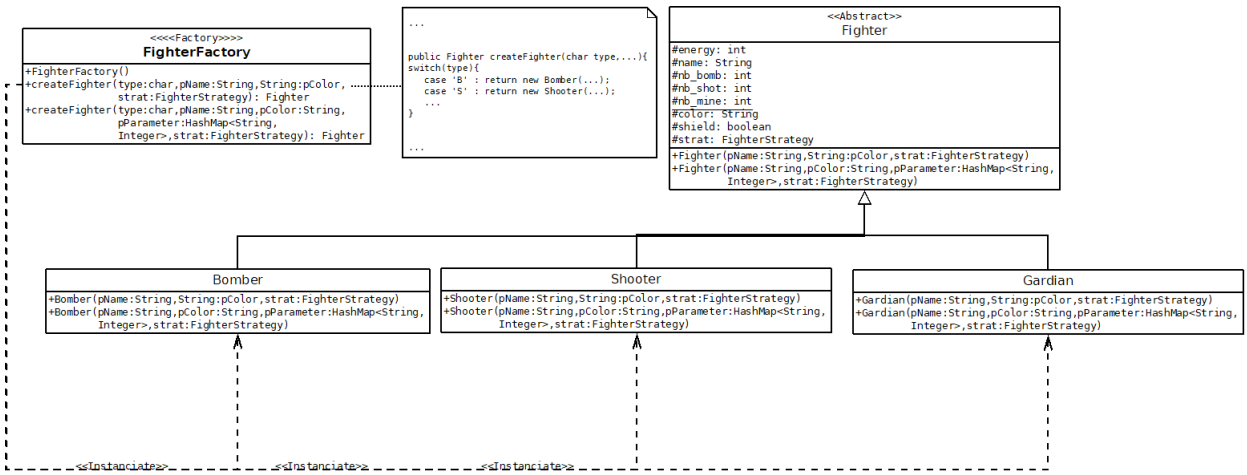
2.3 Les patterns de conceptions

Comme nous l'avons déjà souligné le principal but du devoir est de nous faire travailler les patrons de conceptions. La partie qui va suivre présente notre mise en place de ces patrons au sein de notre application. Par souci de simplicité vous remarquerez que nous ne présentons pas toutes les classes et leurs rôle, si cela bloque la compréhension d'un diagramme nous vous invitons à consulter la javadoc associée.

Pattern Factory

Le **pattern Factory** est un patron de conception dit de construction, il permet d'instancier des objets dont le type est dérivé d'un type abstrait. Pour ce faire on vient déléguer l'instanciation de ces types à une nouvelle classe : la "factory classe".

Dans notre application nous avons créer une classe abstraite **Fighter** pour représenter génériquement un combattant. De cette classe abstraite, héritent les classes **Bomber**,**Shooter** et **Gardian**.Elles nous permettent d'avoir trois types d'instance possibles pour un combattant, chaque type faisant varier les caractéristiques de l'entité tel que "l'énergie". A fin de faciliter l'instanciation de ces différents type nous utilisons la classe **FighterFactory** et ses différentes méthodes. Pour créer une instance concrète d'un Fighter, il faudra utiliser la méthode **createFighter** de notre "factory classe" en spécifiant le type à l'aide d'un caractère passé en paramètre (voir le morceau de code sur le diagramme ci-dessous). Ainsi pour une instance de **Bomber** on passera le caractère **'B'** en paramètre de la fonction.



Pattern Strategy

Le **pattern Strategy** est un patron de conception dit de comportement, il permet d'externaliser le contenu de certaines méthodes pour pouvoir modifier leurs comportements dynamiquement. Pour cela on crée une méthode abstraite ou une interface déclarant de manière abstraite les méthodes que l 'on souhaite externaliser. On crée ensuite les différentes classes qui vont implémenter leur propre version de ces fonctions. Enfin il faut déclarer un attribut du type abstrait dans la classe où l'on souhaite utiliser une de ces méthodes, cette attribut recevra une instance d'un type concret en fonction du comportement que l'on souhaite avoir.

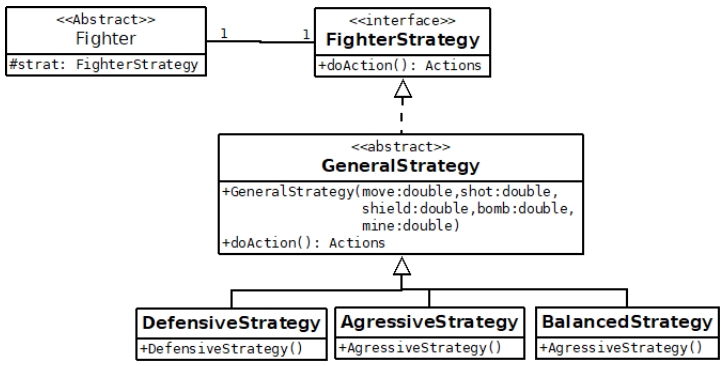
Dans notre application on utilise le **pattern Strategy** deux fois :

— La première utilisation nous sert à définir différents comportements pour le choix des actions des combattants.

Pour cela on crée une interface **FighterStrategy** qui définit la méthode **doActions** qui renvoie une action de jeu pour un combattant. Nous définissons ensuite différents comportements dans le choix de cette action par implémentation de l'interface. La classe abstraite **GeneralStrategy** permet de choisir l'action à réaliser en définissant des probabilités pour chaque action. Ainsi :

- **AgressiveStrategy** déclare des probabilités élevées pour les actions visant à faire des dégâts aux autres combattants.
- **DefensiveStrategy** déclare des probabilités élevées pour les actions visant à se protéger.
- **BalancedStrategy** déclare des probabilités équilibrées.

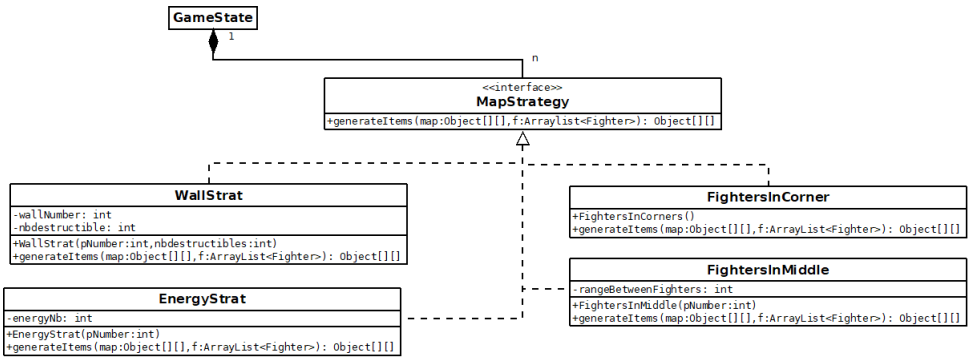
Le pattern nous permet donc de modifier le comportement d'un **Fighter** en lui changeant l'instance de son attribut **FighterStrategy strat**.



— La seconde utilisation nous sert à définir différents comportements pour la génération de notre plateau de jeu.

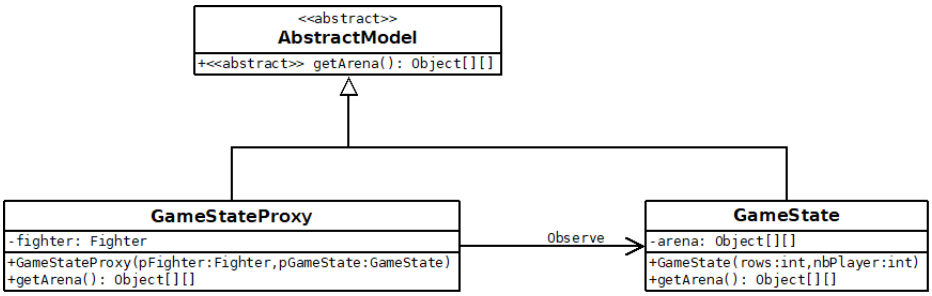
Pour cela on crée une interface **MapStrategy** qui définit la méthode **generateItems**. Nous définissons ensuite différents comportements dans le choix de générations par implémentation de l'interface. Ainsi :

- **WallStrat.generateIems** génère un plateau avec des murs.
- **EnergyStart.generateIems** génère un plateau avec des pastilles d'énergie.
- **FighterInMidlle.generateIems** génère un plateau avec des combattants au centre du plateau.
- **FighterInCorner.generateIems** génère un plateau avec un combattant dans chaque coin du plateau.



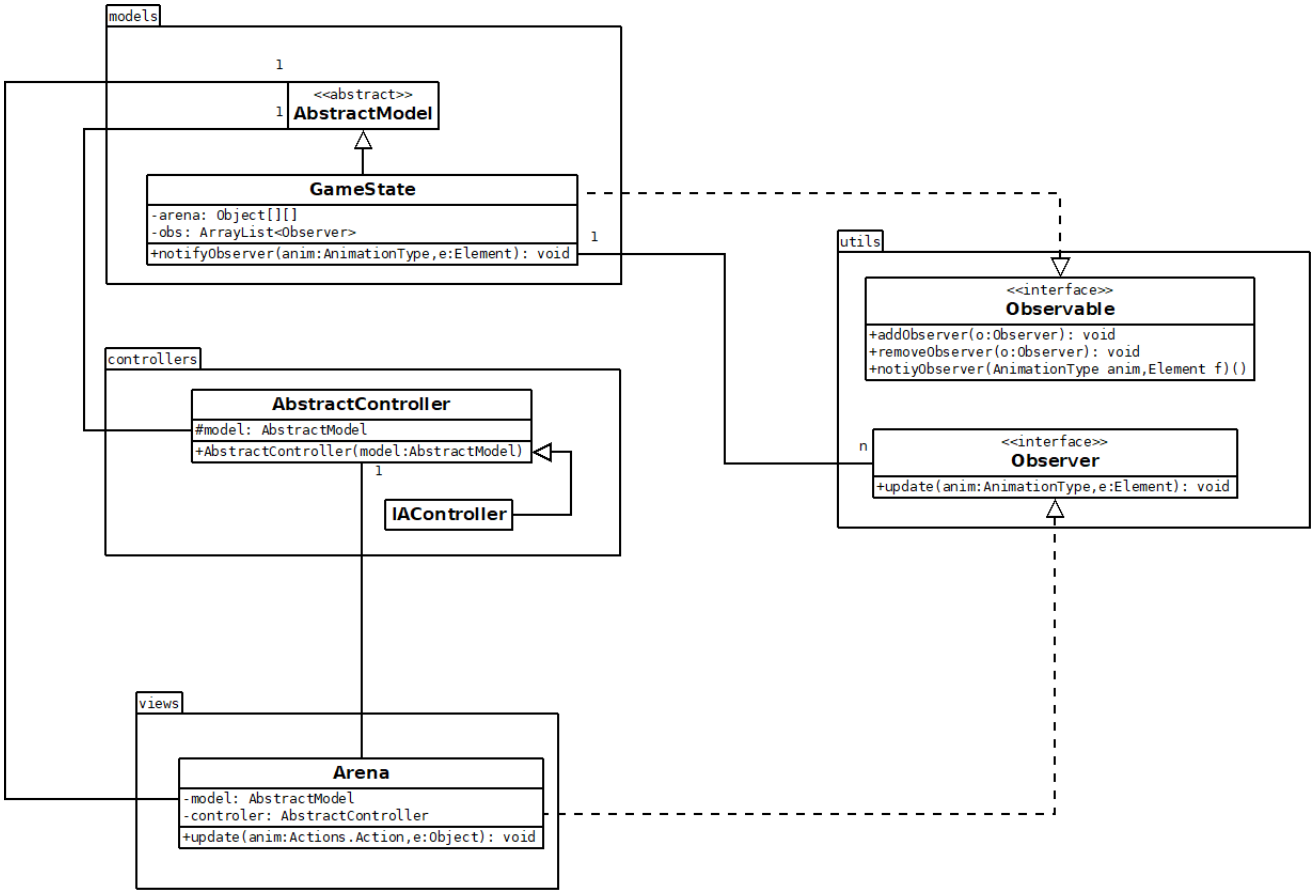
Pattern Proxy

L'une des difficultés du sujet réside sur le fait que les bombes et les mines possèdent deux types de visibilité : visible de tous les combattants, ou visible seulement du combattant qui l'a déposée. De plus il nous est imposé de créer une vue pour chaque combattant affichant seulement les objets qui lui sont visible. Pour répondre à ce problème nous avons dut utiliser le **pattern Proxy**. Il consiste ici à avoir une **classe GameStateProxy** qui va venir lire une instance de notre modèle à fin de pouvoir retourner une version du plateau propre à un joueur. Dans notre application, nous avons crée la classe abstraite **AbstractModel** qui définit la méthode **getArena()**.De cette méthode nous avons fait hériter notre modèle de jeu **GameState** et notre classe proxy. La méthode getArena() de notre modèle va renvoyer la grille d'objet représentant notre plateau tel quelle, tandis que la méthode de notre proxy viendra lire le modèle et retournera la grille avec les éléments uniquement visible par son instance de **Fighter**.



Pattern MVC

Le sujet nous impose une **conception MVC**, celle-ci nous impose de fragmenter notre application de la manière suivante : Une partie **Model**, une partie **Controller** et une partie **Vue**. Le but de se pattern est **de rentre indépendant le model de la vue** et ainsi pouvoir faciliter l'ajout d'un nouvelle affichage, par exemple. Comme nous l'avons expliquer dans la partie architecture nous avons un package réserver à chaque composante de notre conception MVC.

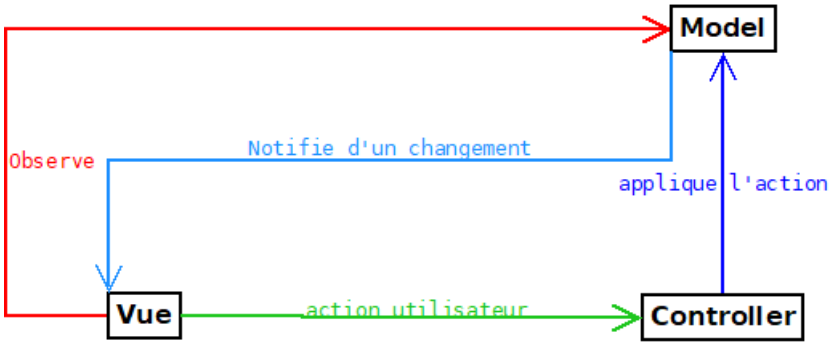


Le diagramme ci-dessus nous permet de comprendre le fonctionnement du pattern. Notre modèle **GameState** va implémenter le super-type **Observable**, ainsi on va pouvoir lui associer une liste d'**Observer** qu'il pourra alors notifier en cas de changement à l'aide de la méthode **notifyObserver**. **Observer** est le super-type que l'on va associé à la vue de notre modèle : **Arena**. Elle lui confère la méthode **update** qui lui permettra de se mettre à jour.

On observe que :

- Notre modèle à un accès à la liste de ses **Observers**.
- La vue **Arena** possède une instance du modèle et du contrôleur.
- Le contrôleur **IAcontroller** possède une instance du modèle.

Ainsi lors d'une interaction utilisateur la Vue va utiliser son instance du contrôleur pour modifier le modèle. Ce dernier ayant subit des modifications il va notifier ses observer d'un changement avec la méthode **notifyObserver**, cette dernière méthode appelle pour chaque observer la méthode **update** et met ainsi à jour toutes les vues observant le modèle.



3 Fonctions secondaires intéressantes :

3.1 Lecture du fichier de paramètres :

Notre application utilise un certain nombre de paramètres qu'il peux être utile de pouvoir changer. Par mis ceux-ci on trouve tous les paramètres relatifs au combattant (nombre d'énergies , nombre de bombes ,etc), les paramètres liés au différentes stratégies. Pour ce faire nous avons mis en place un fichier de paramètre au format JSON. Voici à quoi il ressemble :

```
{
    "Fighter":{
        "Energy":{
            "Gardian" : 100,
            "Shooter" : 80,
            "Bomber" : 60
        },
        "Nb_mine":{
            "Gardian" : 2,
            "Shooter" : 2,
            "Bomber" : 4
        },
        "Nb_bomb":{
            "Gardian" : 2,
            "Shooter" : 2,
            "Bomber" : 4
        },
        "Nb_tir":{
            "Gardian" : 2,
            "Shooter" : 4,
            "Bomber" : 2
        }
    },

    "MapStrategy":{
        "EnergyStrat": false,
        "FightersInCorner": true,
        "FightersInMidlle": true,
        "WallStrat":true
    },

    "FighterStrategy":{
        "DefensiveStrategy": true,
        "CustomStrategy": true,
        "BalancedStrategy": true,
        "AgressiveStrategy": true
    }
}
```

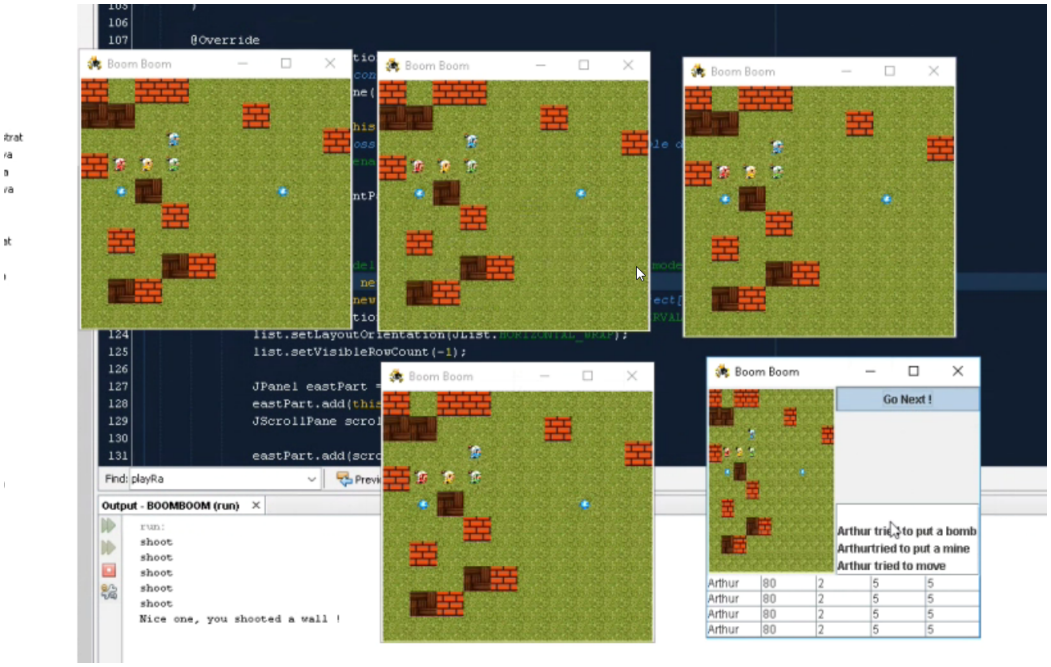
Pour venir lire les données on utilise un "parser" de fichier JSON et notamment la librairie externe **json.simple** dans la classe **ParameterParser** du package utils. Cette classe possède plusieurs méthodes qui retourne des dictionnaires de données associant un paramètre à sa valeur. comme par exemple la méthode **getFighterParameter(char type)** qui retourne les caractéristiques d'un combattant en fonction de son type :

```
public HashMap<String,Integer> getFighterParameters(char  pType){
    HashMap<String,Integer> res =new HashMap<>();
    JSONObject mapSection = (JSONObject) jsonObject.get("Fighter");
    mapSection.keySet().forEach((Object paramKey) -> { //on parcours chaque paramètre.
        JSONObject subMapSection = (JSONObject) mapSection.get(paramKey);
        subMapSection.keySet().forEach((Object type) -> { //on parcours chaque type présent dans le fichier de paramètre(Bombre,Shooter,Gardian).
            if(type.toString().charAt(0)==pType){ //On vérifie que le type est égale à pType passé en paramètre.
                Integer paramValue = Integer.parseInt((String) subMapSection.get(type));
                res.put((String) paramKey,paramValue); // on ajoute le paramètre au dictionnaire.
            }
        });
    });
    return res;
}
```

4 Fonctionnement de l'application et interface utilisateur.

Au lancement de l'application une partie est lancée et cinq fenêtre s'ouvre. Par mis celles-ci, quatre fenêtres correspondent à la vue d'un combattant (ils sont au nombre de quatre). Sur ces fenêtre

aucune interaction n'est possible. La fenêtre restante est la fenêtre principale, elle affiche la vue globale du jeu (tous est visible).



Fenêtres multiples

Dans la fenêtre principale il y a un bouton **"next"** chaque clique sur ce bouton va effectuer un tour de jeu. A chaque tour de jeu un combattant est sélectionné pour réalisé une action :

- Tir infligeant des dégâts à la cible.
- Activation d'un bouclier.
- Pose d'une bombe ou d'une mine sur une case voisine.
- Déplacement sur une case voisine.
- Ne rien faire.

On observe également un tableau affichant les caractéristiques de chaque combattant, son énergie, ses munitions, l'état de son shield ...

5 Conclusion et perspectives d'amélioration

En conclusion, ce devoir nous a permis de nous familiariser avec les différents patrons de conception vu en cour. En effet grâce à un exemple concret, nous avons pu comprendre les problèmes de conception auxquelles répondent chaque pattern, mais aussi la manière dont ils le font. Par ailleurs nous avons aussi put voir l'intérêt de bien modéliser les entités de notre application et leurs relations avant de commencer la rédaction du code. En effet, la mise en place de certain pattern tel qu'une architecture MVC n'est pas toujours évidente et demande une bonne organisation. L'application tel qu'elle est rendu nous laisse plusieurs perspectives d'amélioration. On pourrait imaginer une stratégie plus complexe pour nos combattants ne reposant pas juste sur des probabilités, en effet prendre en compte l'état du combattant et l'état du plateau serait vraiment intéressant. On pourrait aussi avoir une version du jeu faisant jouer non plus des IAs mais des utilisateurs. Enfin notre application présente de la latence entre chaque tour de jeu, il serait intéressant d'optimiser de ce coté pour rendre l'application plus fluide.