# Epic Fantasy Command Battlefield of Doom (with Cats extension)

PROJET DE MCR

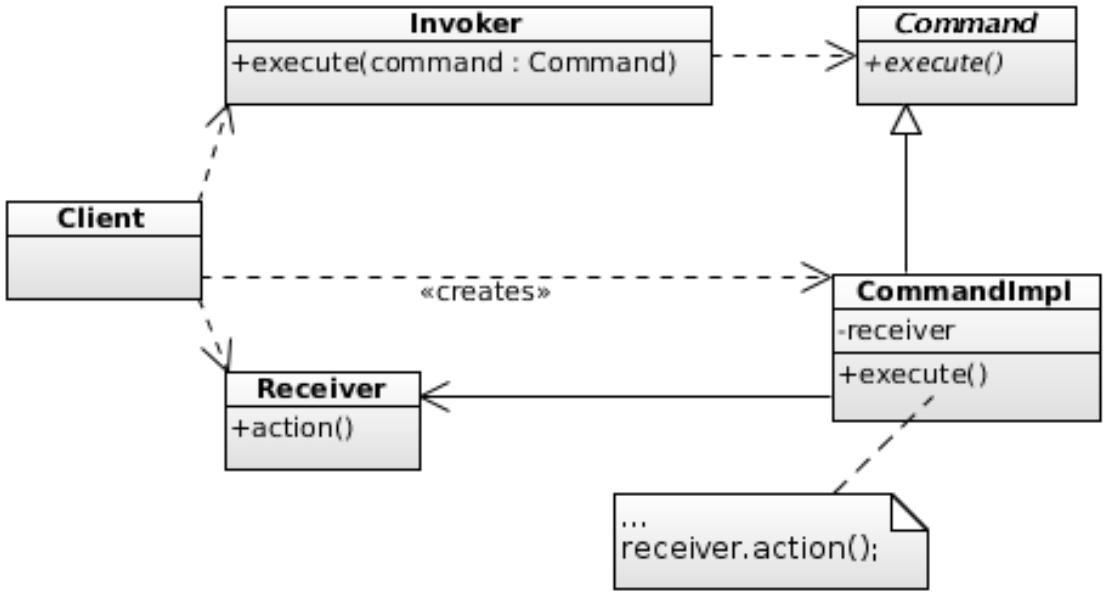
Adrien Allemand | Loyse Krug | Kamil Amrani
Vinceng Guidoux | Guillaume Hochet

# Introduction au modèl commande

Le modèle commande est un modèle de conception réutilisable visant à encapsuler la notion d'invocation d'une méthode sur un objet donné. Il permet ainsi de complètement séparer l'initiateur de cet appel du code de l'action lui-même. De ce principe découlent plusieurs aspects intéressants:

- Aspect d'invocation beaucoup plus générique au moyen d'interfaces
- Découplage facilité, le modèle commande se marie aisément avec le concept de *do* et *undo* d'opérations
- Permet la construction de composants génériques et hautement réutilisables nécessitant la délégation ou l'exécution différée de méthodes

## REPRÉSENTATION




*1 - Model commande*

Un `Receiver`, l'objet dont on veut encapsuler l'appel d'une méthode dans une commande. Sur le schéma, on souhaite ainsi encapsuler la méthode `receiver.action()`.

Une interface `Command` qui définit une méthode `execute()` responsable de l'exécution de l'appel d'une méthode sur un objet donné

Un `Invoker` qui s'occupera d'appeler la méthode `execute()` sur les commandes qui lui sont passées

Une implémentation de Commande qui encapsule ainsi l'appel à la méthode du `Receiver`.

### CONTEXTE

L'appel de la méthode d'un objet dépend d'un contexte donné qui sont les arguments et autres supports dont elle a besoin pour s'exécuter. Ainsi, la création d'une commande doit encapsuler ce contexte afin de fonctionner.

Ceci permet notamment de maintenir en mémoire les différents contextes d'exécutions des commandes (exécutions successives), dans le cas du *do/undo* afin de pouvoir librement naviguer dans l'historique de ces exécutions.

### INCONVÉNIENTS

Le modèle commande introduit un nombre important de classes nécessaires pour fonctionner. L'encapsulation de l'appel d'une méthode d'un objet donné nécessite une nouvelle commande. Ainsi, ce modèle de conception se prête allègrement aux langages supportant les lambda expressions et/ou les classes anonymes, permettant de définir les implémentations des commandes simplement et rapidement.

## Conception du jeu

**Command Battlefield** est un jeu de stratégie tour par tour où deux joueurs s'affrontent dans une bataille sanglante dans le but d'éradiquer les unités ennemies.

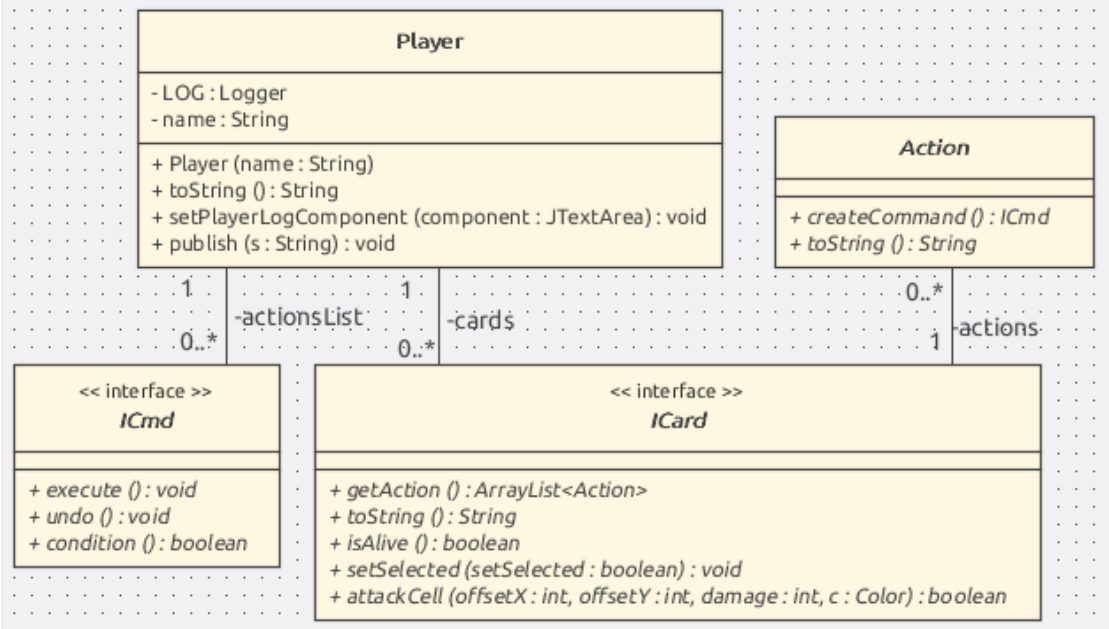
Dans un premier temps, chaque joueur va choisir 5 actions à effectuer pour son tour. Ensuite, d'un commun accord, un des deux joueurs va appuyer sur le bouton *Play* et toutes les actions sélectionnées précédemment vont se dérouler en alternance, une à la fois.

La partie se termine lorsqu'un des deux joueurs n'a plus d'unité en vie sur le terrain, il est déclaré perdant.

Ce jeu est implémenté avec un modèle commande. Lorsqu'un joueur sélectionne une action, celle-ci est encapsulée dans une commande qui est mise dans la liste des actions de ce tour pour le joueur. Lorsque le tour est "joué", les 10 commandes sont exécutées une à la fois.

## Implémentation

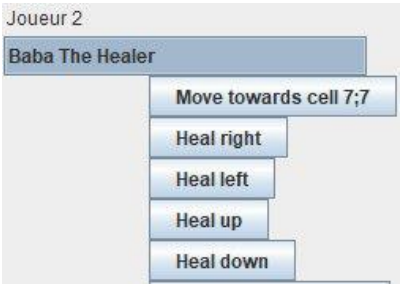
Comme expliqué précédemment, nous utilisons le modèle commande pour encapsuler les actions choisies par le joueur pour son tour afin de les exécuter plus tard.




Si nous nous concentrons sur la partie de l'UML qui concerne la génération des commandes, on peut retrouver l'interface *ICmd* qui nous offre les fonctions utilisées par le contrôleur pour les exécuter.

Le *Player* quant à lui possède une liste de *ICard*, qui représentent des "groupes d'actions propre à une unité ou un sort". Dans un souci d'extensibilité, nous voulons dans une version future implémenter une grande collection de "Cartes" qui implémentent *ICard* et donner la possibilité au joueur de choisir ses "Cartes" donc les sorts et les unités qui composeront son équipe. Dans la version 1.0 les 7 "Cartes" de chaque joueur sont prédéfinies et identiques.

Une *ICard* implémente entre autre la fonction *getAction()* qui retourne une liste d'actions que la carte peut effectuer et est utilisée pour offrir au joueur la possibilité de sélectionner celles-ci pour son tour.



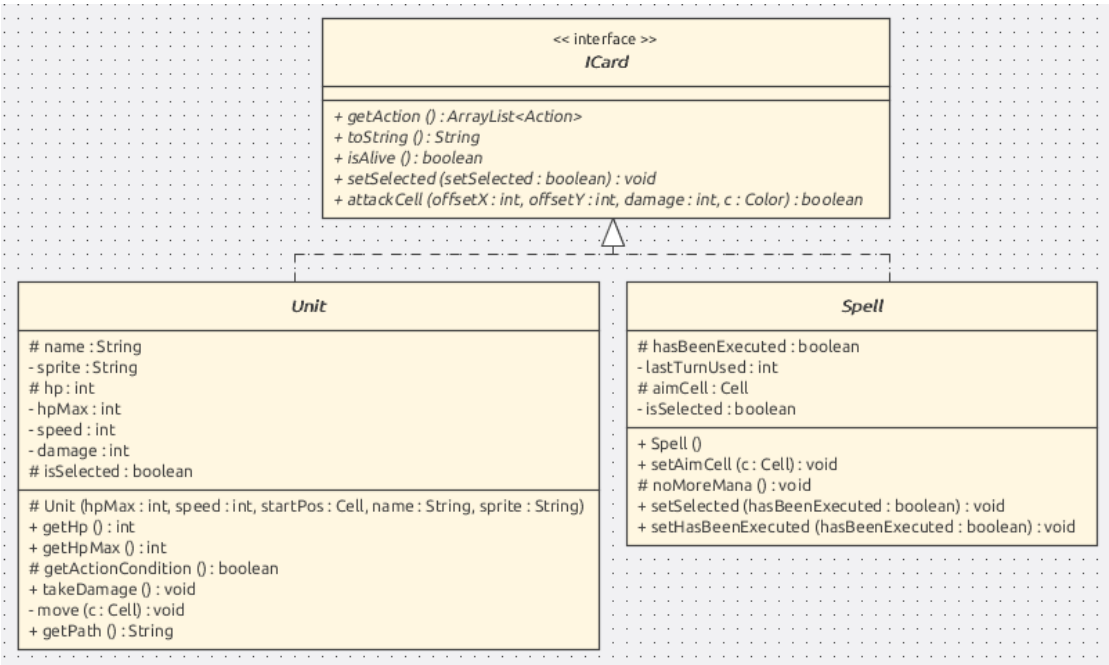

*2 - Les boutons liés aux actions de la carte "Baba The Healer" du joueur 2*

La classe *Action* est une factory abstraite de *ICmd*. C'est elle qui va être utilisée pour générer une *ICmd* au moment où le joueur sélectionne l'action via le bouton qui lui est lié. La commande générée encapsulera les informations nécessaires pour que l'action soit exécutée à la fin du tour. Par exemple, si un joueur clique sur le bouton "Heal right"

de l'image 1 ci-dessus, une *ICmd* encapsulant cette action sera généré par sa factory *Action* et mise a la suite des commandes du joueur pour ce tour.

## *ICard, Unit* et *Spell*

Il y a deux types de *ICard*: les *Unit* et les *Spell*. Les *Unit* on un lien direct avec le terrain et sont contenus a tout moment dans une *Cell*. Les *Spell* ne sont pas présents physiquement sur le terrain mais possèdent d'autres propriétées tel que le nombr de fois qu'ils peuvent être lancés par tour.




*3 - UML de la relation entre ICard, Unit et Spell*

Les *Unit* ont de la vie, la possibilité de subir des dégats ou d'être soignés et l'action de se déplacer en direction d'une cellule d'un certain nombre de cases au maximum.

Une *Unit* ou un *Spell* implémentent l'interface *ICard* et donc doivent être capables de retourner une liste d'action. Ils construisent leur liste d'action en y ajoutant des *Action* anonymes. Chacune de ces *Action* est une factory d'*ICmd* et définissent la méthode *createCommand()* qui crée et retourne une *ICmd* implémentée de manière anonyme encapsulant l'action a exécuter et son contexte d'exécution au moment de son instanciation.

Vous pouvez voir ci-dessous l'ajout de l'action de création d'un obstacle pour le *Spell* "Create Fake Unit".

```java
//Add fakeUnit spell command factory to action list
actions.add(new Action(){

    public ICmd createCommand() {

        return new ICmd() {
            String hasBeenThrown = "";
            boolean hasBeenSpawned = false;
            Cell c = game.getSelected();

            //there's no need for a unit to be alive in order to launch the spell
            @Override
            public boolean condition() throws InterruptedException {
                return true;
            }

            //A fake unit is created on the chosen cell if the content is still null
            @Override
            public void execute() throws InterruptedException {
                //The spell can only be called once per turn
                if(hasBeenExecuted){
                    hasBeenThrown = noMoreMana();
                }else{
                    //We note that the spell has been executed
                    hasBeenExecuted = true;
                    FakeUnit fu = new FakeUnit(c);
                    if(c.getCellContents() != null){
                        c.setContent(fu);
                        hasBeenSpawned = true;
                    }
                }
            }

            //The fake unit is destroyed
            @Override
            public void undo() {
                if(hasBeenSpawned){
                    c.setContent(null);
                }
            }

            //Name of the command
            @Override
            public String toString() {
                return "Create an obstacle in" + c.x + ":" + c.y + "! " + hasBeenThrown ;
            }
        };
    }
    //Name of the action
    @Override
    public String toString() { return "Create an obstacle"; }
});
```

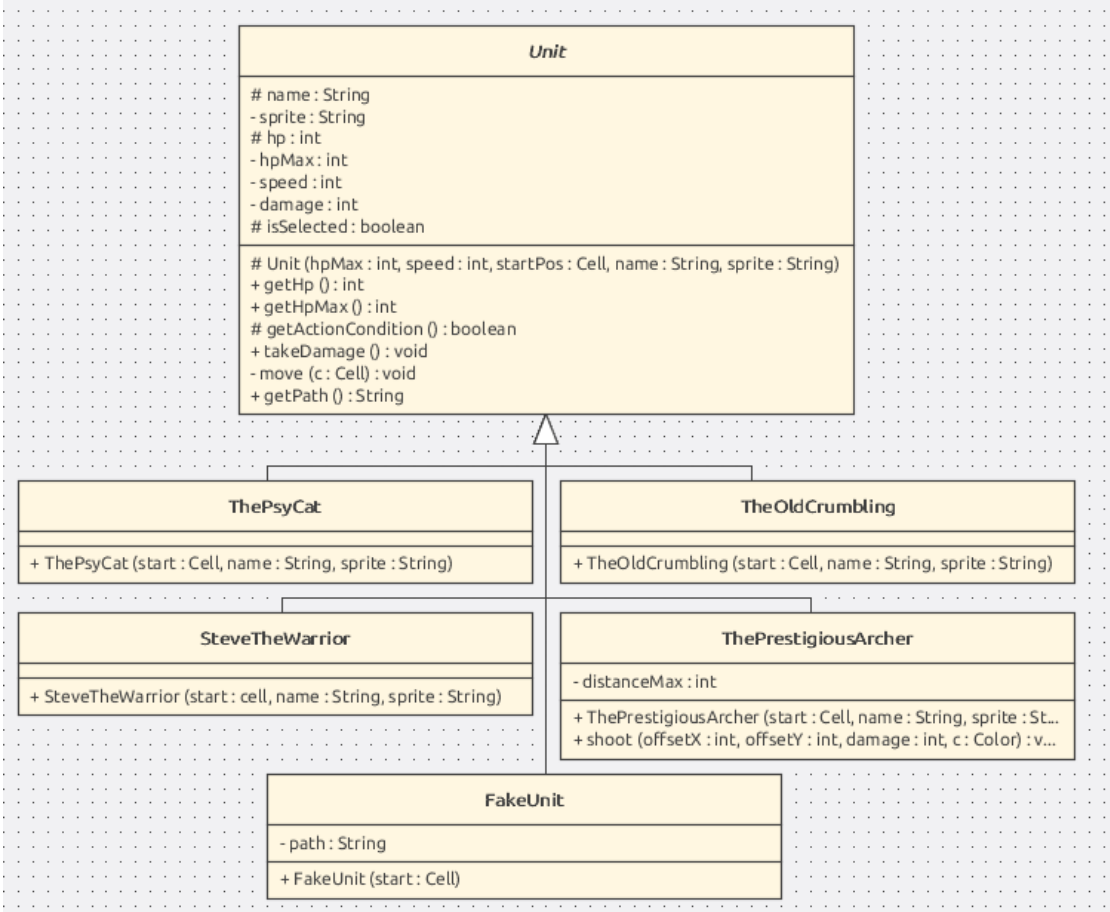
*4 - Ajout d'une nouvelle action*

L'*ICmd* anonyme commence par enregistrer son contexte d'exécution, on voit ici qu'elle sauve la cellule actuellement sélectionnée et déclare des variables locales nécessaire à son exécution ou son annulation.

```
String hasBeenThrown = "";
boolean hasBeenSpawned = false;
Cell c = game.getSelected();
```

Puis elle implémente les fonctions *condition()*, *execute()*, et *undo()* afin de réaliser l'action désirée.

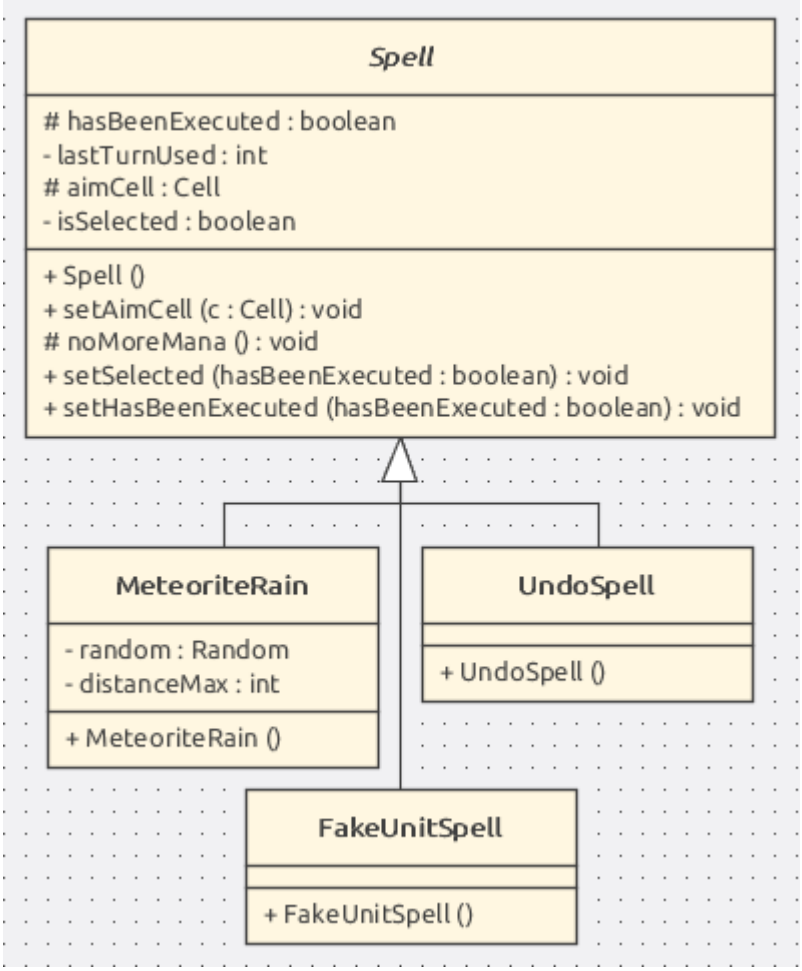
Nous avons actuellement implémenté 5 *Unit* :




*5 - UML des Unit conctètes*

Chacune retourne une liste d'*Action* qui encapsulent des actions hétérogènes en exploitant la force du modèle commande.

Nous avons également implémenté 3 *Spell* :




*6 - UML des Spell concrets*

De nature tout aussi hétérogène, ils retournent également une liste de leur *Action*.

# Conclusion

La réalisation de ce projet a été très intéressante notamment pour approfondir la découverte et la puissance du modèle. Nous avons néanmoins observé beaucoup de difficultés dans l'implémentation de notre projet, pas forcément lié au modèle mais plutôt à nos objectifs.

## DEVELOPPEMENT

Lors de l'élaboration du schéma UML et de la reflexion sur la logique et la structure du projet, le modèle commande semblait être aisé à implémenter tout en permettant une grande simplification du code par le découplage des composants.

Néanmoins étant la première fois que nous l'implémentions ainsi nous avons sous-estimé l'accès à l'information. Notre application a été développée suivant le modèle MVC en plus du modèle commande, et l'accès au modèle s'est avéré plus difficile que prévu engendrant un développement archaïque accompagné d'un peu de code spaghetti qui a été long à factoriser.

Malgré ces difficultés, ce projet a été très intéressant à développer. Nous avons pris beaucoup de plaisir à imaginer un univers pour notre jeu et sommes globalement satisfaits du résultat, même si on retient qu'il est littéralement impossible de faire une interface graphique jolie et visuellement acceptable avec Swing.