PROTOTYPE

MCR - Projet

HEIG-VD

Crüll Loris, Jaquet David, Rod Julien, Rohrbasser Yoann & Selim Stephan

Table des matières

1 Introduction

Dans le cadre du cours de Modélisation de Conception Réutilisable (MCR), nous sommes amenés à réaliser un projet inventif et original illustrant l'utilisation d'un modèle de conception. Ce modèle a été choisi en 7ème semaine du semestre et nous avons choisi le modèle de conception « `Prototype` ».

Nous avions cinq semaines pour nous documenter et réaliser une présentation théorique sur le modèle. Ensuite, nous avions de nouveau cinq semaines pour réaliser la partie pratique du projet. Ce rapport concerne cette dernière partie.

2 Modèle de Conception Réutilisable

2.1 Prototype

Comme dit dans l'introduction, nous avons choisi le modèle de conception « `Prototype` » pour faire ce projet. L'utilisation de ce modèle se fait lorsque la création d'une instance possède un grand nombre d'instructions ou prend un temps considérable. Pour faire simple, ce modèle est une sorte d'amélioration du modèle « `Fabrique` », mais les objets « prototypés » possèdent une méthode « `clone` ».

2.2 Implémentation

Pour implémenter un « `Prototype` » dans un programme, nous avons besoin des éléments suivants :

- Gestionnaire de prototypes
- Classe parente à chaque élément que nous voulons prototyper
- Une classe par élément à prototyper

2.3 Exemple d'utilisation

2.3.1 Situation

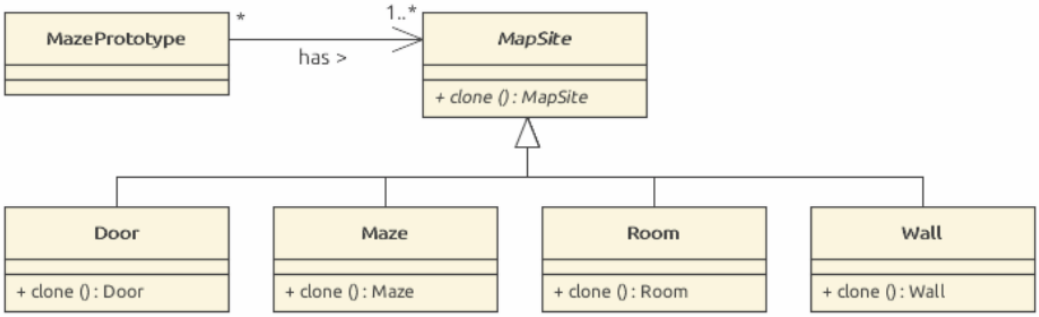
Un exemple d'utilisation de notre modèle de conception serait la création d'un labyrinthe. Pour cet exemple, nous pouvons imaginer que nous voulons une génération dynamique du labyrinthe afin d'implémenter un système de niveau (une fois un labyrinthe terminé, un nouveau labyrinthe est généré).

Dans une telle situation, on peut imaginer que chaque instance des éléments du labyrinthe (porte, pièce, mur et même labyrinthe) a des éléments communs avec les autres instances d'une même classe. En conséquence, à la place de créer plusieurs instances presque identiques, nous pouvons créer un « `prototype` » d'une classe que nous clonons et modifions des valeurs propres.

Pour reprendre l'exemple du labyrinthe, nous pouvons avoir un `prototype` d'une porte contenant des éléments qui ne changeront pas (taille, matériaux, etc.) et définir les éléments pouvant être modifiés (pièces reliées à la porte, etc.) lors du clonage.

2.3.2 Schéma UML

Voici un schéma UML de la situation décrite au point précédent. Il est important de noter que ce schéma n'est pas complet (les classes ne possèdent pas tous les attributs nécessaires, n'ont pas toutes les méthodes nécessaires, etc.). Néanmoins, ce schéma est suffisant pour illustrer l'utilisation du modèle de conception « `Prototype` ».



UML d'un `Prototype` exemple

Voici la correspondance des éléments selon la liste fournie lors de la description de l'implémentation du modèle :

- `MazePrototype` : Gestionnaire de prototypes
- `MapSite` : Classe parente à chaque élément que nous voulons prototyper
  - `Sous-classes` : Élément à prototyper

3 Notre implémentation

3.1 Description du projet

Pour implémenter le « `Prototype` », nous avons décidé de créer un mini-jeu. Notre application reprend l'exemple du labyrinthe détaillé précédemment. Nous l'avons toutefois rendu plus amusant.

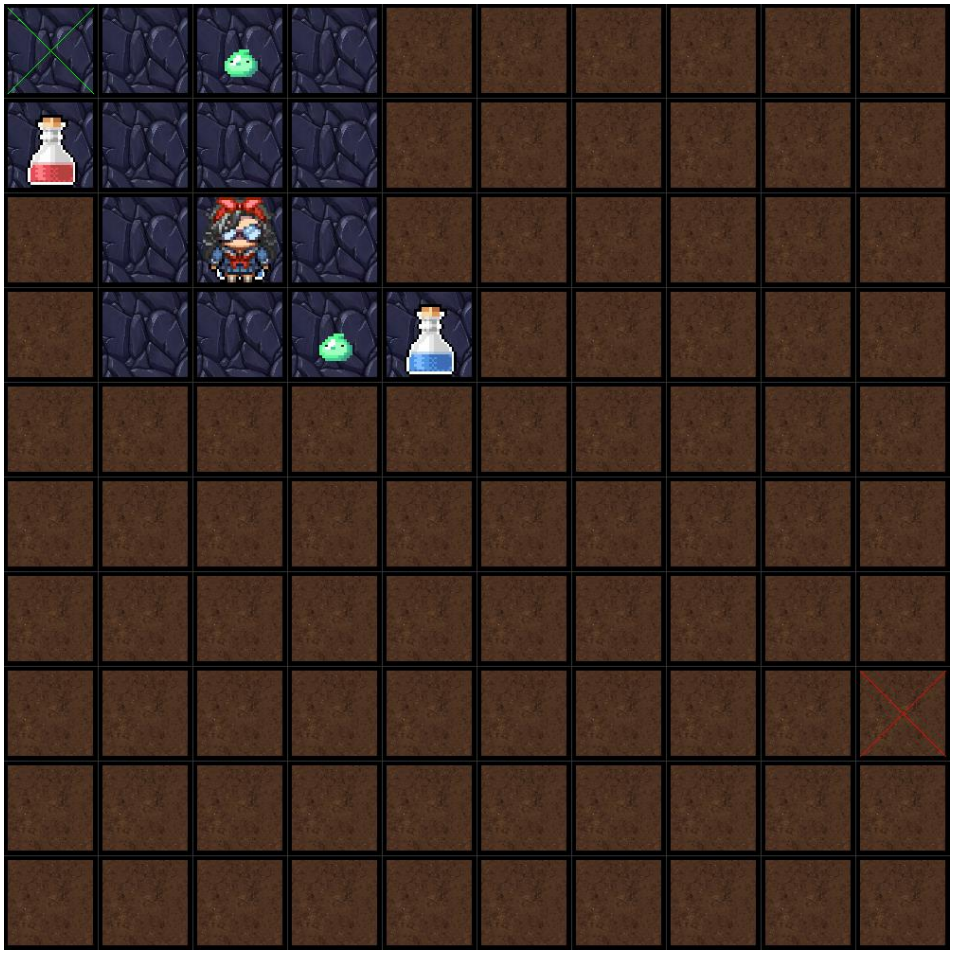
Nous avons donc développé une application où nous pouvons tout d'abord choisir un personnage jouable. Il est important de noter que la sélection des personnages à une influence sur le déroulement du jeu. En effet, chaque personnage a des statistiques et des sorts différents. Une fois cette longue étape franchie, nous arrivons sur tableau de `10x10` cases.

Magus

Kagami

Warrior

L'utilisateur de notre application peut ensuite de déplacer de case en case à l'aide des touches du clavier correspondant au mouvement de la plupart des bons jeux sur ordinateur. Il s'agit donc des touches « `A` », « `S` », « `D` » et « `W` » correspondant respectivement au déplacement vers la gauche, le bas, la droite et le haut.



Au fur et à mesure que l'utilisateur déplace son personnage, de plus en plus de cases deviennent visibles. Ces cases peuvent soit être des cases de type « murs » nous bloquant l'accès, soit des cases de types « sols ». Le personnage se déplace de case en case, mais uniquement sur des cases « sols ».

Les cases « sols » peuvent avoir plusieurs éléments au-dessus d'elles. En effet, ces dernières peuvent avoir des objets (potion de vie, potion de mana), un monstre ainsi que bien évidemment le personnage contrôlé par l'utilisateur.

Potion de vie

Potion de mana

Il existe cinq types de monstres différents :

Poulpe

Pieuvre

Kraken

Slime

Invocateur

Chaque type monstre a des statistiques et sorts différents. Il est toutefois important de préciser que le type de monstre « `summoner` » va invoquer un nouveau monstre tous les `6 tours`. Le monstre invoqué sera soit un monstre « `octopus` » (pieuvre) soit un monstre « `squid` » (poulpe). De plus, les monstres de types « `slime` » se clonent tous les `12 tours`.

À chaque fois que l'utilisateur effectue un mouvement, l'application vérifie si le personnage jouable se trouve sur la même case qu'un objet ou qu'un monstre. Dans le premier cas, l'objet est ajouté à l'inventaire du personnage et dans le second cas, un combat se lance.

Une fenêtre s'affiche contenant les informations d'un combat. Cette fenêtre se compose de la manière suivante :

Cette fenêtre se compose de la manière suivante :

Numéro	Correspondance
1	Image du monstre que l'utilisateur est entrain de combattre
2	Information sur le déroulement du combat
3	Informations sur l'utilisateur (point de vie et point de mana)
4	Informations sur le monstre (point de vie et point de mana)
5	Liste de boutons permettant respectivement d'attaquer, lancer un sort et utiliser un objet
6	Listes de sorts à lancer et d'objets à utiliser. Le sort ou l'objet se lance en changeant l'élément sélectionné ou en appuyant sur le bouton correspondant

Il est important de préciser que si deux monstres se trouvent sur la même case que l'utilisateur, les combats s'enchaineront l'un après l'autre. Le combat s'achève lorsque les points de vie de l'utilisateur ou du monstre tombent à zéro. Si l'utilisateur perd son combat, un écran de défaite s'affiche.

Le niveau se termine lorsque l'utilisateur a réussi à se déplacer jusqu'à la case de fin de niveau.

3.2 Éléments prototypés

Pour illustrer notre modèle si souvent évoqué, nous avons décidé de le mettre en évidence lors de deux phases décrites dans les points suivants.

3.2.1 Génération du niveau

La première de ces phases est la génération du niveau. Pour des raisons de simplicité de développement, la génération du niveau ne se fait pas de manière aléatoire, mais prend un tableau de chiffres entiers. Voici le tableau de notre niveau en guise d'exemple :

9	7	7	1	0	0	0	1	7	8
1	2	1	1	0	0	1	1	7	8
7	1	8	1	0	0	1	0	0	0
1	1	1	2	0	0	3	0	0	0
0	0	0	8	0	8	1	1	4	0
0	1	3	1	0	1	0	0	1	0
0	1	0	0	0	2	0	1	1	1
0	4	1	1	1	1	0	1	1	1
0	1	1	7	1	1	0	1	1	6
0	7	1	5	8	1	0	10	6	0

Numéro	Correspondance
0	Case « mur »
1	Case « sol »
2	Case avec un `slime`
3	Case avec un poulpe
4	Case avec une pieuvre
5	Case avec un `summoner`
6	Case avec un `Kraken`
7	Case avec une potion de vie
8	Case avec une potion de mana
9	Case de départ du personnage
10	Case de fin de niveau

Nous avons donc un gestionnaire de prototype (appelé « `GamePrototypeManager` ») qui contient un prototype pour chaque élément décrit dans le tableau précédent. Lors de la génération de niveau, l'application va vérifier quel est le type de la case et cloner le prototype correspondant grâce à l'instruction « `switch` ». Ce clonage se fait dans la classe « `Dungeon` ».

3.2.2 Invocation des monstres

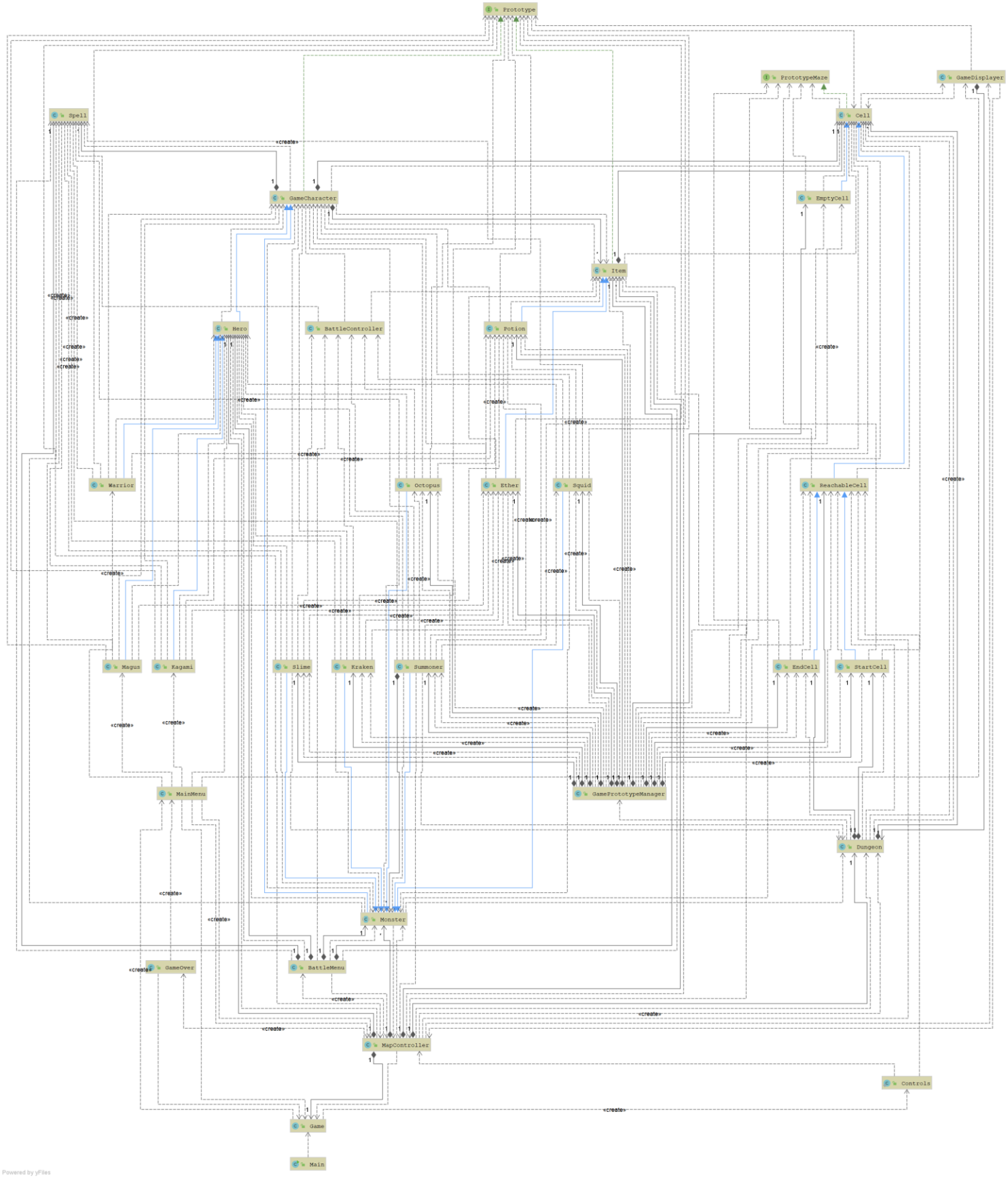
La seconde étape est l'invocation des monstres. Comme expliqué précédemment, il y a deux manières distinctes d'invoquer un monstre :

- Un `summoner` invoque une pieuvre ou un poulpe (choisi aléatoirement)
  - Cette action se passe tous les `6 tours`
- Un `slime` se duplique
  - Cette action se passe tous les `12 tours`

Ces invocations se trouvent dans la méthode « `DungeonInteraction` » de la classe « `Summoner` » et « `Slime` ».

3.3 Schéma UML

Voici le schéma UML de notre application. Ce schéma étant relativement conséquent, il est également disponible en annexe numérique au format PNG.



3.4 Bugs connus

Voici ci-dessous, une liste de bugs dont nous avons connaissances. Il est important de noter que ces bugs n'ont pas été résolu dû à un manque de temps. Pour la même raison, cette liste peut être incomplète.

3.4.1 Plusieurs monstres peuvent être sur la même case

Plusieurs peuvent se retrouver sur la même case. Avec nos choix d'implémentation, ce n'est pas un problème en soit. Néanmoins, lorsque l'utilisateur se déplace sur une case avec plusieurs monstre, un combat va se lancer mais uniquement contre un seul monstre.

Une fois arrivée sur une case de ce genre, le programme va parcourir la liste des monstres présents sur la case. Un combat ne se lance que contre le dernier monstre de cette liste. Cela a comme conséquence pour l'utilisateur qu'il n'effectuera qu'un seul combat au lieu de « `N` » (où `N` est le nombre de monstre présent sur la case).

3.4.2 Un monstre peut être sur la case de fin

Un ou des monstres peuvent être présent sur la case de fin de niveau. Encore une fois, ce n'est pas un problème en soit. Cependant, en plus du problème cité dans le point précédent, le programme va en premier afficher l'écran de victoire et ensuite lancé le combat.

Il est évident que les étapes doivent être inversée : le combat doit d'abord se lancer, puis, afficher l'écran de fin de niveau en cas de victoire.

3.4.3 Fin de partie

La partie se termine lorsque nous arrivons sur la case de fin. Une fois cette étape franchie, un écran de victoire s'affiche et, lorsque l'utilisateur clique, nous retournons sur l'écran de sélection de personnage.

Le problème est qu'à ce moment on ne peut pas relancer une partie. Une série d'exception se lance lorsqu'on choisit notre personnage.

3.4.4 Affichage des monstres

Pour d'obscurs raisons que nous n'avons pas eu le temps d'approfondir, nous avons remarqué que certains monstres ne s'affichent pas. Ce bug a été découvert très peu de temps avant le rendu et nous avons par conséquent pas eu le temps de résoudre ce problème.

4 Conclusion

Ce projet nous a permis de découvrir et de bien approfondir le modèle de conception « Prototype ». Nous avons profité de ce projet pour de développer un jeu afin de rendre le projet plus attractif et original par rapport à une application simple et potentiellement ennuyeuse. Nous sommes satisfaits du résultat du projet malgré les quelques bugs que nous n'avons pas réussi à résoudre à temps. Cette satisfaction vient évidemment du produit, mais également du travail de groupe dans lequel la dynamique était optimale.