Université de Genève

Concepts et langages orientés objets
12X003

TP : Projet

Author: Joao Filipe Costa da Quinta

E-mail: Joao.Costa@etu.unige.ch

June 13, 2021

UNIVERSITÉ DE GENÈVE
FACULTÉ DES SCIENCES
Département d'informatique

Intro

Pour ce TP je n'ai pas repris un TP spécifique vu en cours depuis lequel commencer, mais plutôt, je vais travailler sur un côté du jeu RPG qu'on a introduit lors de nos TPs, mais qu'on n'a pas de tout développé.

On a parlé de personnage, qui a une classe, on a vu la classe des mages et des guerriers, on a vu que tout personnage a des points de vie qui changent selon des actions, et qu'ils ont aussi des points de mana. C'est sur les termes que je viens de mentionner que je vais travailler dans ce projet.

Je choisis d'ignorer l'idée de pièce d'armure qu'on peut équiper, d'armes etc.., je le fais car ces termes ne sont pas nécessaires à mon implémentions.

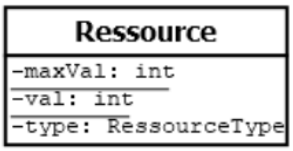
Cahier des Charges

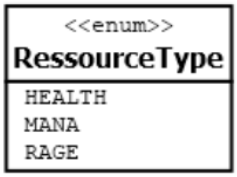
Développer une super classe GameChar qui possède des Ressources qu'on peut modifier à volonté avec une seule fonction. Pour que cette fonction unique marche on doit donc prendre avantage de la flexibilité des génériques, les classes et les fonctions plus importantes seront présentés dans les sections suivantes:

Ressource

Nous avons parlé de ressource dans notre jeu RPG, on a introduit les points de vie, et la mana, souvent la vie sert à prendre des coups sans mourir, et le mana é pouvoir lancer des sorts dans le but de faire perdre des points de vie à quelqu'un d'autre.

J'introduis donc la classe Ressource, qui contient 3 attributs, la valeur maximale, la valeur actuelle, et le type de ressource. Le type de ressource est en fait une enum, RessourceType, voici le UML correspondant: (getters et setters omis)



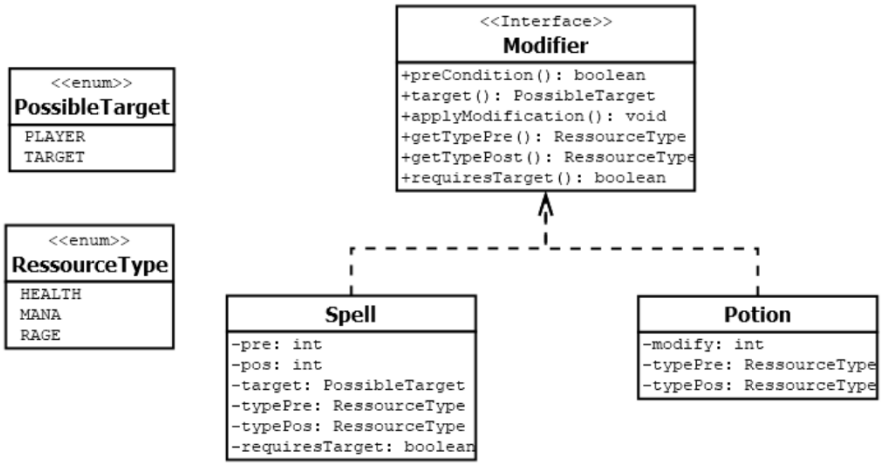


J'introduis aussi une nouvelle ressource, la Rage

Modifier

Modifier est la représentation d'un comportement, un comportement qui a pour but la modification de ressources, que ça soit chez le joueur ou chez l'ennemi, ou les deux en même temps, on peut penser aux potions de vie et mana qu'on a vu, qui modifient les ressources correspondantes, je vais aussi introduire les sorts, qui sont lancé par les joueurs pour pouvoir blesser les ennemis.

Vu que Modifier représente un comportement, ce sera une interface.



preConition() vérifie si le joueur respecte les conditions nécessaires à activer le Modifier.

getTypePre() représente le type de ressource pour la preCondition

target() indique à qui la post condition du Modifier sera appliqué, ça peut être le jouer lui même, ou son target (la joueur qu'on vise).

plyModification() applique les post conditions du Modifier

getTypePost() représente le type de ressource pour la posCondition

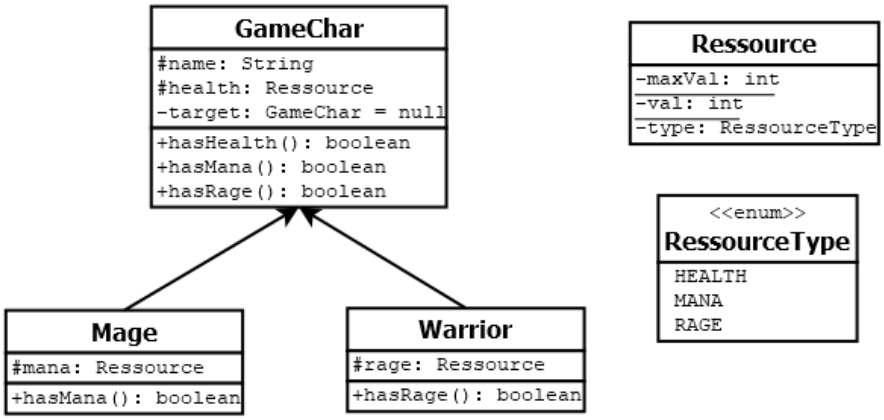
requiresTarget() indique si le jouer a besoin d'avoir une personne cible pour activer le modifier.

(1) La première classe qui implémente Modifier sera donc la classe Potion.
Le constructeur de la classe Potion prend comme argument la valeur qu'on modifie, et le type de ressource à modifier.
La pré condition pour consommer une potion, est que le jouer soit en vie, puis si c'est une potion de mana, on calcule la nouvelle quantité de mana, pareil pour vie.

(2) La deuxième classe qui implémente Modifier sera donc la classe Spell, elle est un peu plus complexe.
Un exemple d'un Spell, est le Frostbolt, le Frostbolt peut être lancé que si le joueur possède assez de mana pour, et si on a une cible, ensuite il fait du dégâts à l'ennemi.
Spell frostbolt = new Spell(-10, -20, PossibleTarget.TARGET, RessourceType.MANA, RessourceType.HEALTH, true);
-10, RessourceType.MANA -> indiquent que pour lancer Frostbold le joueur doit avoir 10 ou plus de mana.
-20, PossibleTarget.TARGET, RessourceType.HEALTH -> indiquent que en lançant le Spell, le PossibleTarget va recevoir 20 de dégâts de vie.
true -> indique qu'il faut avoir un target ennemi pour lancer le sort.
On voit bien les pre post Conditions du lancement d'un sort.

GameChar

Maintenant le but est de créer une classe pour le jouer, qui contient une fonction qui vérifie le respect des pré conditions et exécute les post conditions sans problèmes. De plus, cette fonction va se comporter de façon voulu, tant qu'on l'utilise avec un objet qui implémente Modifier.
Je définis donc GameChar qui a un nom, des points de vie représentés par la classe Ressource et le champ target, qui est du type GameChar, et initialisé à null.
Je crée deux sous-classes, le mage et le warrior, le mage aura un attribut en plus, le mana, représenté par une Ressource, et le warrior aura la Rage.



GameChar aura une fonction hasRessourceType() : boolean, hasHealth() retourne true, car tout personnage aura de la vie, puis toutes les autres retournent false, puis dans les sous classes correspondantes on surcharge les fonctions correspondantes à la ressource qu'on a, donc un mage qui a du mana, surcharge la hasMana() et retourne true. Il y a bien sur les getters de ces mêmes ressources qui fonctionneront avec la même logique.

```java
public boolean hasHealth() {
    return true;
}
public boolean hasMana() {
    return false;
}
public boolean hasRage() {
    return false;
}
```

Figure 1: GameChar.java

```java
@Override
public boolean hasMana() {
    return true;
}
```

Figure 2: Mage.java

Vu qu'on introduit la possibilité d'avoir un target, faut bien sur introduire les fonctions pour, on a donc setTarget(GameChar), puis loseTarget().

On peut finalement parler de la fonction qui fait toute la magie:

```java
public <T extends Modifier> boolean cast(T m) {
    if(canCast(m.requiresTarget()) && m.preCondition(this)) {
        m.applyModification(this, this.target);
        return true;
    }
    return false;
}
```

La première chose qu'on fait c'est vérifier que notre jouer peut cast/utiliser ce modifier, si le joueur n'a pas de target, et veut lancer une Frostbolt, on l'empêchera. C'est en quelque sorte la pré condition basique, ensuite on vérifie les autres pré conditions, cette fonction reçoit this comme argument, car les pré conditions sont toujours en lien avec le jouer, pour Frostbolt, qu'il ait assez de mana disponible pour. Dans la fonction elle même on fait appel à this.hasMana(), donc un warrior ne pourra jamais cast un Frostbolt, seulement si hasMana() retourne true, on pourra alors vérifier que le GameChar a aussi assez de mana pour. S'il en a assez, alors les pré conditions sont respectées, on calcule donc les nouvelles valeurs.

Finalement on exécute applyModifications() pour les post conditions, on donne comme argument le jouer et son target car on ne vérifie pas ici sur qui les modifications s'appliquent.

Ensuite la vrai logique pour chaque classe qui implémente Modifier, est dans la classe elle même, on va voir l'exemple pour la classe Spell:

```
// je vais expliquer le cas de figure pour HEALTH, les 2 autres cas marchent de la même façon
case HEALTH:
    if(player.hasHealth()) {// si le personnage dispose de la ressource HEALTH
        Ressource health = player.health;
        if(health.getVal() >= pre) { // alors on vérifie qu'il ait assez de HEALTH pour 'payer' le sort
            // s'il a assez de ressource, alors on calcule la nouvelle valeur pour la ressource HEALTH
            health.setVal(health.getVal() + pre);
            return true;
        }
        return false;
    }
    return false;
```

Logique très simple, on vérifie la pré condition et on l'applique, puis on retourne true ou false.

```
GameChar toChange;
// on défini toChange = GameChar à qui on applique les modifications
if(this.target == PossibleTarget.PLAYER) {
    toChange = player;
}else {
    toChange = target;
}
//selon le typePost, on sait quelle ressource du joueur il faut modifier
switch(this.typePost) {
case HEALTH:
    if(toChange.hasHealth()) {
        // cas de health -> pareil pour les autres
        Ressource health = toChange.health;
        int newVal = health.getVal() + pos;// on défini la nouvelle valeur
        if(newVal <= health.getMaxVal()) {
            if(newVal > 0) {
                health.setVal(newVal);
            }else {
                health.setVal(0);
            }
        }else {
            health.setVal(health.getMaxVal());
        }
        //on véfirie bien que la valeur est entre 0, le min, et le maxVal,
        //on pourrait aussi mettre -> health.setVal(Math.max(0, Math.min(health.getMaxVal(), newVal)));
    }
```

Ensuite la fonction qui applique les post conditions, on vérifie sur qui on dont faire les modifications, sur qu'elle ressource, puis on les fait. Rien de très compliqué.

Avec cette logique j'ai pu coder des des Spell très intéressants, et complètement différents.

Frostbolt	Coûte 10 mana, fait 20 de dégâts au target.
Charge	Coûte rien, donne 20 de Rage au joueur.
heal	Coûte 10 mana, donne 20 de vie au joueur.
DeathPact	Coûte 20 de vie, donne 20 de mana au joueur.

En regardant un jeu comme World of Warcraft, on peut coder beaucoup de Spells dans le jeu avec cette logique.

Conclusion

On peut ajouter beaucoup de fonctionnalités, on peut donner les pré conditions et post conditions sous forme de array, pour en avoir plusieurs en même temps pour le même Spell.

Si on veut ajouter une nouvelle classe pour personnage, comme les Rogue, on peut aussi très facilement ajouter une unique ressource qu'ils peuvent utiliser, comme l'ENERGY, et ca ne serait pas très compliqué d'implémenter la logique. En effet mon code n'est pas de tout compact, mais je pourrais faire le switch pour savoir quel ressource on doit modifier, puis avoir le changement de valeur en commun, car c'est indépendant de la ressource.

On peut aussi définir beaucoup plus de choses comme étant une Ressource, comme par exemple l'armure, et du coup on implémente la classe ArmorPiece qui implémente Modifier, et quand on l'équipe la pièce au joueur, on augmente la ressource armor. Après faudrait aussi coder le coté pour déséquiper un item, qui donnerait les valeurs opposés en armor, mais ça marcherait bien.

J'ai utilisé beaucoup de choses vu en cours, mais l'idée principale était autour des génériques. Avec une fonction qui est simple, on est capable de modifier des différentes ressources, avec des objets différents, de façon voulu, et ça rend la suite du travail / ajout de mécaniques, très simple.