Chapitre 5
Intelligence Artificielle

1. Préparation

Avant de démarrer la conception et l'implantation d'intelligences artificielles, quelques préparatifs s'imposent. Le premier concerne la possibilité de rendre non-modifiable l'état du jeu, ce qui permet de prévenir les erreurs. Le second concerne la définition d'une interface pour toutes les intelligences artificielles.

1.1 État du jeu non modifiable

Jusqu'à présent, l'état du jeu a toujours été modifiable par n'importe quel acteur, y compris ceux qui n'ont aucune raison de le modifier. Par exemple, le moteur de rendu ne doit pas modifier l'état du jeu, au risque de perturber l'équilibre de l'ensemble. De même, une intelligence artificielle ne doit pas modifier l'état du jeu, sauf dans le cas où elle est autorisée à tricher.

Pour garantir qu'un objet et ceux qu'il contient ne seront pas modifiés, il y a deux principales voies : se faire confiance et/ou faire confiance aux membres de son équipe, ou tout simplement rendre impossibles les modifications. Dans le premier cas, même les développeurs les plus rigoureux peuvent parfois faire des erreurs, en particulier lorsqu'une méthode modifie les données d'une manière contre-intuitive ou non documentée. Ce type de scénario est courant au bout de plusieurs années de développement, où le projet est constitué de milliers de classes.

La méthode la plus sûre consiste à rendre les modifications impossibles. Il existe plusieurs approches. Dans cette section, deux approches sont proposées, l'une lorsque les classes à protéger ne sont pas modifiables (ex. : bibliothèque externe), et l'autre lorsque les classes peuvent être modifiées.

1.1.1 Approche avec le Patron Proxy

Lorsque les classes à protéger ne sont pas modifiables, il est possible d'utiliser le patron Proxy. Pour rappel, celui-ci consiste à définir une nouvelle classe qui copie le contenu des classes à traiter, tout en répondant à la même interface. Puis, les objets sont remplacés par leur proxy : les utilisateurs de la classe ciblée ne voient pas la différence, sinon les fonctionnalités ajoutées. Dans le cas présent, toutes les méthodes qui ne modifient pas les attributs sont inchangées, et celles qui les modifient jettent une exception.

Classes sans conteneurs

Voici un premier exemple avec la classe `Wall` de l'état du jeu exemple Pacman. On définit une classe `ImmutableWall` qui hérite de la classe `Wall` :

```
public class ImmutableWall extends Wall {
    public ImmutableWall(Wall wall) {
        super(wall.getWallTypeId());
    }
    public void setWallTypeId(WallTypeId wallTypeId) {
        throw new IllegalAccessError();
    }
}
```

Il n'y a qu'un constructeur de copie, qui duplique l'unique attribut de la classe et de ses classes mères. Puis, seul le mutateur (*setter*) `setWallTypeId()` de l'attribut est redéfini pour renvoyer une exception stipulant que l'utilisation de cette méthode est interdite.

Classes avec conteneurs

Le même principe est répété pour toutes les classes sans conteneurs, comme `Space`, `Pacman` et `Ghost`. Pour les classes avec conteneur, il faut faire attention, en particulier s'il y a des accesseurs (*getters*) qui renvoient un conteneur, comme la méthode `getChars()` de la classe `Characters` :

```
public List<MobileElement> getChars() {
    return chars;
}
```

Bien que cela ne permet pas de modifier l'attribut `chars`, cela n'interdit pas d'en modifier le contenu. Pour l'empêcher, on peut par exemple jeter une exception pour cette méthode, puis ajouter des accesseurs pour les éléments de la liste `chars`. On peut également renvoyer un autre proxy de la liste, ce qui est déjà fourni dans la bibliothèque standard dans la classe `Collections` avec la méthode statique `unmodifiableList()` :

```
public List<MobileElement> getChars() {
    return Collections.unmodifiableList(chars);
}
```

Il est aussi possible de construire cette liste non modifiable directement dans les constructeurs de la classe, afin d'éviter de répéter sa création si l'accesseur est souvent utilisé.

Attention : les éléments du conteneur ne doivent pas également posséder des conteneurs, auquel cas la protection ne sera pas complète.

Pour les accesseurs dans une classe conteneur, comme la méthode `get()` de la classe `Characters`, il faut également s'assurer que les objets renvoyés sont non modifiables :

```
public MobileElement get(int index) {
    MobileElement me = chars.get(index);
    if (me instanceof Pacman) {
        return new ImmutablePacman((Pacman)me);
    }
    if (me instanceof Ghost) {
        return new ImmutableGhost((Ghost)me);
    }
    throw new RuntimeException("Type d'élément invalide");
}
```

Utilisation

Pour utiliser le proxy, il suffit de créer les versions non modifiables lorsque cela est nécessaire. Par exemple, lorsque l'état notifie des modifications, l'état fourni est une version non modifiable :

```
public void notityStateChanged() {
    ImmutableState roState = new ImmutableState(this);
    for (StateObserver observer : observers) {
        observer.stateChanged(roState);
    }
}
```

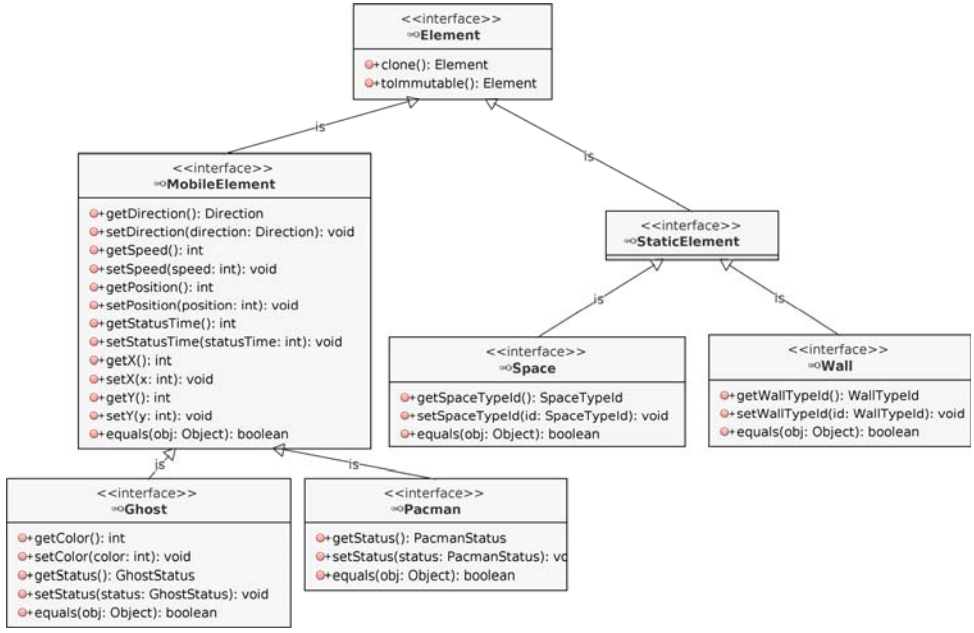
Ainsi, le moteur de rendu ne peut pas modifier l'état par mégarde.

Les diagrammes de cet exemple sont disponibles dans le dossier « Class Diagrams/chap05/immutable01» du projet UML exemple. Le code est présent dans le dossier « examples/chap05/immutable01 » du projet Java exemple.

1.1.2 Approche avec le Patron Décorateur

Le patron Proxy est pertinent lorsque les classes à protéger ne peuvent pas être modifiées. Il souffre cependant de défauts. Le premier vient du fait qu'il ne faut pas oublier de redéfinir les mutateurs dans la classe proxy. Or, lorsque des fonctionnalités sont ajoutées un peu vite, il est courant de l'oublier ou de le remettre à plus tard. Une manière de s'assurer que les protections sont toujours en place est de définir une interface et deux implantations : l'une avec toutes les fonctionnalités d'accès et de modification, et l'autre avec seulement les fonctionnalités d'accès. L'autre problème vient de la surcharge calculatoire (*overhead*) engendrée par le patron Proxy. La solution proposée permet de pallier ces deux problèmes.

Pour définir une classe en lecture seule, le patron décorateur est très intéressant. Pour rappel, il consiste à définir une classe avec un attribut du type de la classe à décorer (ou plusieurs le cas échéant), puis à implanter la même interface en appelant les méthodes de l'attribut. Ces appels peuvent empêcher les modifications ou en comporter. Il est également possible d'ajouter de nouvelles méthodes. Par exemple, pour les classes d'éléments, la hiérarchie des classes est reproduite avec des interfaces :



On peut noter la méthode `toImmutable()` de la classe `Element`, qui n'existait pas dans l'interface initiale. Cette méthode permet de renvoyer de façon systématique une version modifiable de l'objet, et est très pratique dans plusieurs cas présentés ci-après.

Classes modifiables

Les classes modifiables implantent ces interfaces et sont nommées `MutableXXX`, par exemple `MutablePacman` pour la classe `Pacman`. L'implantation est identique à la version initiale, sauf pour les méthodes `toImmutable()`. Par exemple, pour les classes non abstraites comme `MutableWall`, une version décorée est renvoyée :

```
public Element toImmutable() {
    return new ImmutableWall(this);
}
```

Classes non modifiables sans conteneurs

Les classes non modifiables implantent également ces interfaces et sont nommées `ImmutableXXX`. Les classes sans parent comme la classe `ImmutableElement` ont un unique attribut vers la classe équivalente modifiable :

```
public class ImmutableElement implements Element
{
    protected final MutableElement element;
```

Un unique constructeur permet d'initialiser l'attribut :

```
    public ImmutableElement(MutableElement element) {
        this.element = element;
    }
```

Il est toujours possible de proposer une copie de l'élément décoré. Dans ce cas, rien n'interdit de renvoyer une copie modifiable. Cela est possible puisque les copies sont intégrales :

```
    public Element clone() {
        return element.clone();
    }
```

La méthode `toImmutable()` renvoie une version non modifiable de l'objet, soit lui-même :

```
    public Element toImmutable() {
        return this;
    }
}
```