CHAPTER 5

Artificial Intelligence

5.1 Preparation

Before starting the design and implementation of artificial intelligence, some preparations are required. Firstly, we need to design a read-only game state to avoid unintentional errors. Secondly, we need to define an interface for artificial intelligence (AI), to separate AI internal mechanics from its use.

5.1.1 Immutable game state

So far, the game state has always been editable by any component, including those who have no reason to change it. For example, the rendering engine should not change the state of the game. Similarly, artificial intelligence must not change the state of the game, except in the case where it is allowed to cheat.

There are two main approaches to ensure that an object can not be changed: trusting members of your team or making changes impossible. In the first case, even the most rigorous developers can sometimes make mistakes, especially when a method modifies the data in a counter-intuitive or undocumented manner. This kind of scenario is common after several years of development, where the project is made up of thousands of classes.

The safest approach is to make changes impossible. In this section, we propose two approaches, one when the classes to be protected are not modifiable (ex: external library), and the other when the classes can be modified.

5.1.1.1 Approach with the Proxy Pattern

When the classes to protect are not modifiable, it is possible to use the Proxy pattern. As a reminder, this one consists in defining a new class which copies the contents of the classes to be processed, while providing the same interface. Then, we replace the objects with their proxy: the users of the targeted class do not see the difference, except for the added features. In this case, all methods that do not modify the attributes are unchanged, and those that modify them throw an exception.

Classes without containers

Here is a first example with the `Wall` class of the Pacman game state. We define an `ImmutableWall` class that inherits from the `Wall` class:

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

There is only one copy constructor, which duplicates the unique attribute of the class and its parent classes. Then, only the setter `setWallTypeId()` of the attribute is redefined to return an exception stating that the use of this method is forbidden.

Classes with containers

We repeat this principle for all classes without a container, such as `Space`, `Pacman` and `Ghost`. For classes with a container, we have to be careful, especially if some getters return a container, like the `getChars()` method of the `Characters` class:

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

Although this does not modify the `chars` attribute, it does not prohibit modifying its content. To prevent this, for example, you can throw an exception for this method,

and then add accessors for items in the `chars` list. You can also return another proxy from the list. The standard library already provides such a proxy in the `Collections` class with the `unmodifiableList()` static method:

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

It is also possible to build this non-modifiable list directly in the constructors of the class, to avoid repeating their creation if we often use the accessor.

Warning: the container elements must not also have containers, in which case the protection is not complete.

For accessors in a container class, such as the `get()` method of the `Characters` class, we must also make sure that the returned objects are unmodifiable:

```
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("Invalid type");
}
```

Use

To use the proxy, create non-editable versions when necessary. For example, when the state notifies changes, the supplied state is a non-editable version:

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

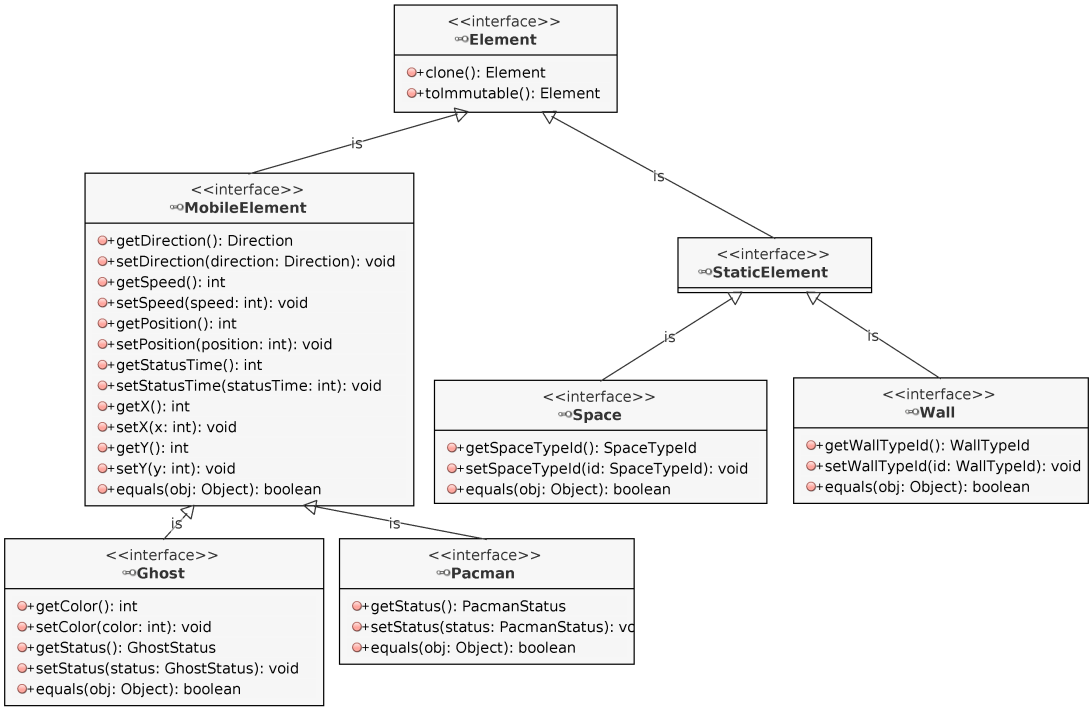
Thus, the rendering engine can not change the state unintentionally.

The diagrams of this example are available in the “Class Diagrams/chap05/immutable01” folder of the sample UML project. The code is present in the “examples/chap05/immutable01” folder of the Java sample project.

5.1.1.2 Approach with the Decorator Pattern

The proxy pattern is relevant when the classes to be protected can not be changed (like external libraries). However, it has some drawbacks. The first is that we must not forget to redefine the mutators in the proxy class. When we add new features, it is usual to forget these redefinitions or to postpone them. One way to ensure that protections are still in place is to define an interface and two implementations: one with all access and modification features, and the other with only access features. The other problem is the overhead generated by the proxy pattern. The proposed solution makes it possible to overcome these two problems.

The decorator pattern is very interesting to define a read-only class. As a reminder, it consists in defining a class with an attribute of the type of the class to decorate (or several if necessary), then to implement the same interface by calling the methods of the attribute. These calls can be without modifications, or with modifications. It is also possible to add new methods. For example, for element classes, the class hierarchy is reproduced with interfaces:



We can notice the `toImmutable()` method of the `Element` class, which did not exist in the initial interface. This method returns a non-editable version of the object and is very useful in several cases presented below.

Editable classes

Editable classes implement these interfaces and are named `MutableXXX`, for example `MutablePacman` for the `Pacman` class. The implementation is identical to the initial version, except for the `toImmutable()` methods. For example, for non-abstract classes like `MutableWall`, a decorated version is returned:

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

Non editable classes without containers

Non-modifiable classes also implement these interfaces and are named `ImmutableXXX`. Classes without a parent, such as the `ImmutableElement` class, have a unique attribute that points to a mutable object:

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

A single constructor is used to initialize the attribute:

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

For the `clone()` method, we can safely return an editable copy since the copies are deep (see the previous Chapter):

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

The `toImmutable()` method returns a non-modifiable version of the object, which is itself:

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

For non-abstract classes, such as `ImmutableWall`, we use the attribute to implement getters, for example:

```
public WallTypeId getWallTypeId() {
    return ((MutableWall)element).getWallTypeId();
}
```

And for the setters, we throw an exception:

```
public void setWallTypeId(WallTypeId wallTypeId) {
    throw new IllegalAccessError();
}
```

In addition, we declare non-modifiable classes as final. It means that all their methods are final, and can not be redefined:

```
public final class ImmutableWall ... {
    ...
}
```

This information allows the compiler to optimize calls to the decorated methods, in many cases removing the intermediate call.

Non-editable classes with containers

For classes with containers, we must care as in the previous case, and ensure that the elements contained are also non-modifiable. For example, for the `get()` method of the `Characters` class:

```
public MobileElement get(int index) {
    MobileElement me = chars.get(index);
    return (ImmutableMobileElement)me.toImmutable();
}
```

The use of non-modifiable classes with the decorator pattern is the same as with the proxy pattern.

The diagrams in this sample are available in the “Class Diagrams/chap05/immutable02” folder of the sample UML project. The code is present in the “examples/chap05/immutable02” folder of the Java sample project.

5.1.2 Artificial Intelligence Interface

Before starting the design of artificial intelligence, an environment is created to ease their management.

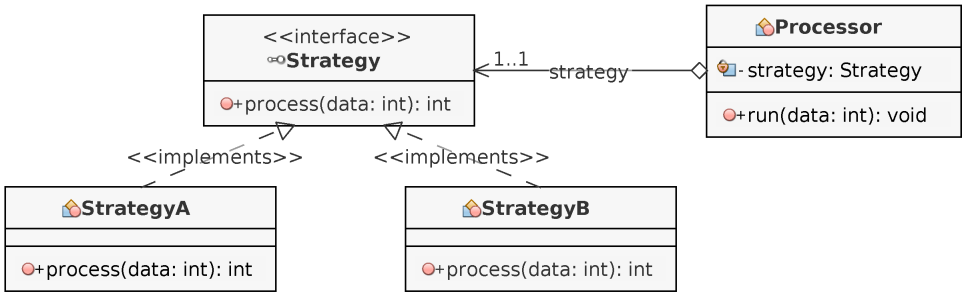
5.1.2.1 List the possible commands (Strategy Pattern)

To be able to propose artificial intelligence with an architecture based on the command pattern, it is enough to propose commands for the artificial players. For example, for the Pacman game, it is enough to propose an orientation command evaluated as the most effective to win.

Before being able to make a choice, we need to know the list of possible commands for a given game state. This task is not artificial intelligence, and it depends on the rules of the game. These rules can vary according to different options, and it is more effective for an AI to delegate this task to a third party. We can use the strategy pattern to get this result.

Strategy Pattern

The *Strategy Pattern* allows you to change the behavior of a process while the program is running. It can be presented as follows:

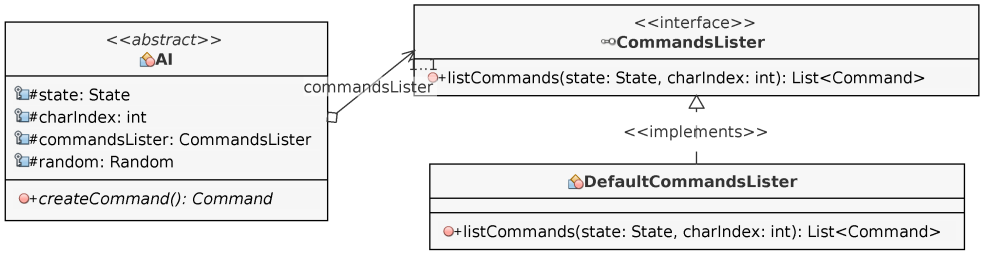


The `Strategy` interface defines the sub-processes that can be modified during program execution. In this illustration, there is only one `process()` method, but we can add other methods with the arguments of our choice. There are two implementations of the interface in this example, `StrategyA` and `StrategyB`. Then, a `Processor` class handles all the processing logic. It uses an implementation of the `Strategy` interface blindly: it knows only the methods of this interface. It can use the implementation that it references with its attribute in a free way.

The typical example of using the Strategy pattern is that of sorting data. Indeed, the algorithm is usually fixed - such as *quick sort* or *binary heap*, but the way you compare data may vary. In the standard library, the `Collections.sort()` static method has two arguments: the list to be sorted and one implementation of the `java.util.Comparator` interface. This interface is the equivalent of the `Strategy` interface in the illustration above.

Use the list of possible commands

The Strategy pattern can be used to separate the logic of artificial intelligence from the process of determining possible commands. We define a `CommandsLister` interface that acts as a strategy and a `DefaultCommandsLister` implementation that provides the commands with the default rules. Then, the implantations of an `AI` interface manages the logic of the AI by making use of a `CommandsLister` implementation:



Implementations of the `createCommand()` method return the best command for the `charIndex` index character based on the data in `state`.

⇒ Note: Words and their meaning: the “strategy” in the sense of the pattern in this example is not the strategy of artificial intelligence, but the nature of the commands that can be considered, regardless of any intelligence algorithm.

⇒ Note: In the proposed example, the `CommandsLister` interface and its implementations are present in the artificial intelligence part of the project. A more complex but more consistent design is naturally to place its elements in the rules engine part, and to enrich the engine so that it is a factory of `CommandsLister`.

5.1.2.2 Random behavior

Implementation of a random AI

A first artificial intelligence is created to test implementations. It consists of making random choices among the possible commands. For the Pacman example game, a `RandomAI` class implements the `AI` interface:

```
public class RandomAI extends AI
{
    public RandomAI(State state, int charIndex,
        CommandsLister lister, Random random) {
        super(state, charIndex, lister, random);
    }
```