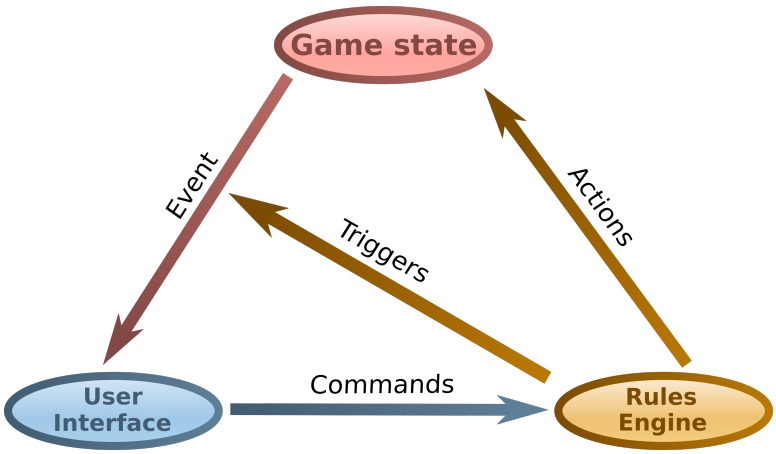
CHAPTER 4

Rules engine

4.1 General approach

4.1.1 Presentation

As quickly presented in Chapter 1, the general approach followed in this book is a modified version of the Model-View-Controller (MVC) approach:



As a model, we find the state of the game, presented in Chapter 2. Unlike the MVC approach, the game state in the proposed approach is only a storage location: the only available modifications are basic setters. In the example of the Pacman game presented so far, the attributes of the `PlayGameMode` class form the game state. For example, the `level` attribute contains the cells of the world. As a view, we find the user interface presented in Chapter 3. Finally, as a controller, we find the rules engine, embodied by the `update()` method in the Pacman game example. Unlike the MVC approach, it is the rule engine that validates changes in the state of the game.

Concerning the interaction between these three actors, the major difference lies in the passive nature of the game state. When the user acts, such as pressing a key or selecting an item in a menu, the user interface produces a command, such as "move the character to the right." This command is sent to the rules engine, which deduces the actions needed to modify the state, for example, to change the character's *x* coordinates. Besides, the rule engine triggers the sending of events indicating state changes - it is not the state that transparently indicates that it has changed. When the user interface receives the event, it modifies its rendering data accordingly, and for example, it modifies the *x* coordinate of the character's sprite.

4.1.2 Motivation

This approach is the result of constraints specific to video games, and to any application whose display requires the same constraints. Indeed, the user interface, on the one hand, and the state of the game and the game engine on the other, evolve in two "parallel" universes:

- The user interface must provide high-speed rendering, typically at 60 frames per second. Aesthetics motivate this constraint, on the contrary to office applications that usually ignore it;
- The state of the game evolves at a slower speed, regularly or irregularly. The first reason for this constraint is simply that not all screens work at the same refresh rate. It would be strange to watch a game whose evolution is faster on a screen at 144Hz than on a screen at 60Hz. It is also motivated by aspects of computational resources, especially when one wishes to include advanced artificial intelligence. Finally, the management of the multiplayer game requires it, and networks can not operate at frequencies as high as the refresh of a screen.

To manage these aspects, one must be able to operate the two entities independently. Communication should take place only briefly at key moments in the game cycle. Between these moments, everyone must evolve independently:

- When the rules engine changes the game state, the user interface should not be interrupted. It must be able to offer a consistent display until the state of the game is available for viewing. For example, in the GUI facade presented

earlier, that's why layers have their list of sprites. They are disconnected from the state of the game and can be drawn without consulting the data of the game state.
- When the user interface produces a command, for example, as a result of a pressed key, it can not be consumed if the rules engine is changing the state. We must find solutions to postpone this consumption.

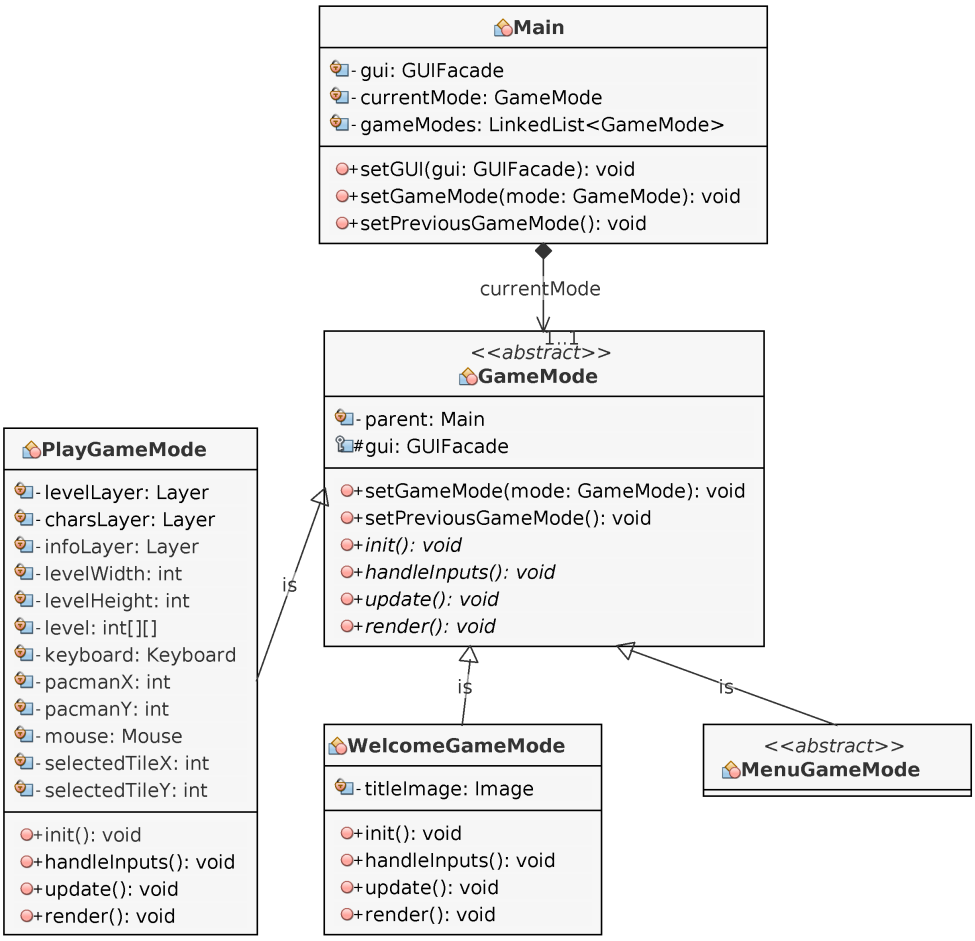
The purpose of this chapter is to present different approaches and associated design patterns to meet these requirements. Section 2 focuses on the separation between game state and rendering in the user interface. Section 3 presents the rules engine and its interaction with the other two actors. Finally, section 4 presents several features that can be easily added thanks to the proposed approach. Indeed, if it meets the constraints presented above, it also offers other very interesting benefits.

⇒ Note: The general approach followed in this book is not the only solution to every conceivable problem. It is perfectly suited to a large number of cases, partially answers the problems raised by others, and is irrelevant in some cases. Once again, the primary goal of this book is to train the mind in software design techniques, primarily using design patterns. The example of this approach has many pedagogical interests and helps you to create new approaches. For the video game developed while reading this book, beginners are invited to reproduce it, and the most experienced should use it as an inspiration to design their solution.

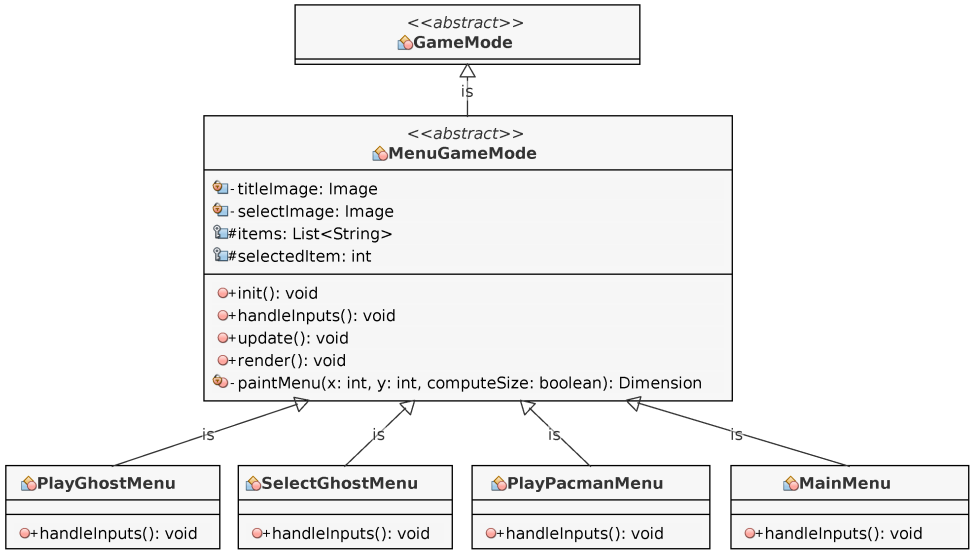
4.2 Synchronization between state and user interface

The purpose of this section is to synchronize the game state data, as designed in Chapter 2, with the user interface presented in Chapter 3.

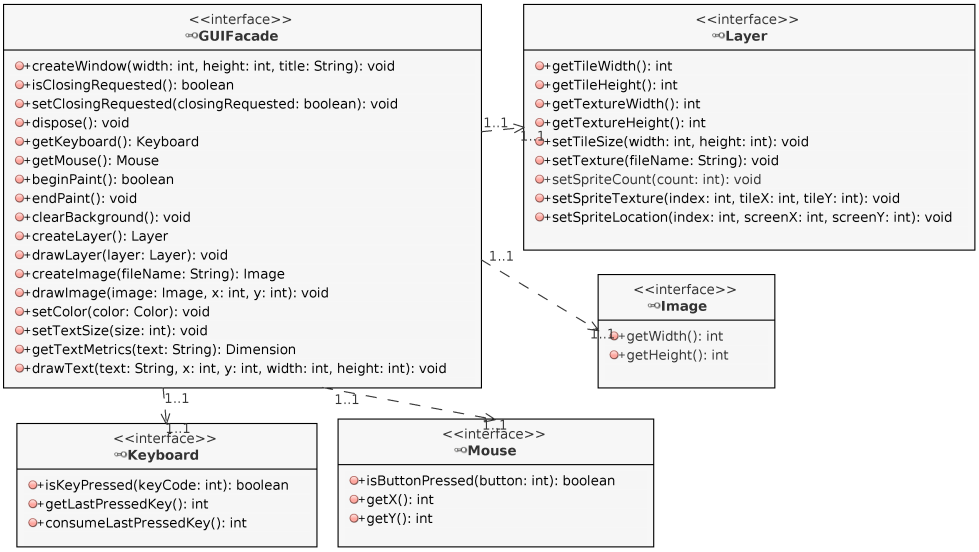
Before starting, here is a summary of the elements designed so far, with a little reorganization. First, the main classes are kept in the main package, and separated from the menu classes:



All menu classes are placed in the `menu` package:

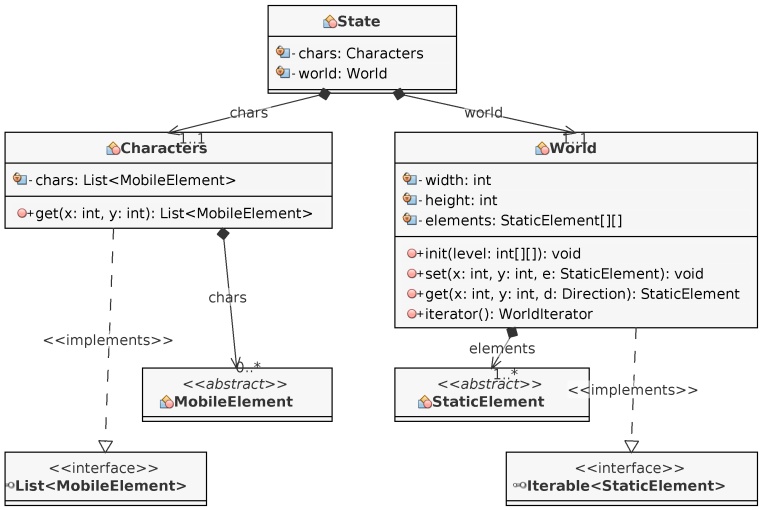


The facade for the user interface is kept, and placed in the `gui` package:

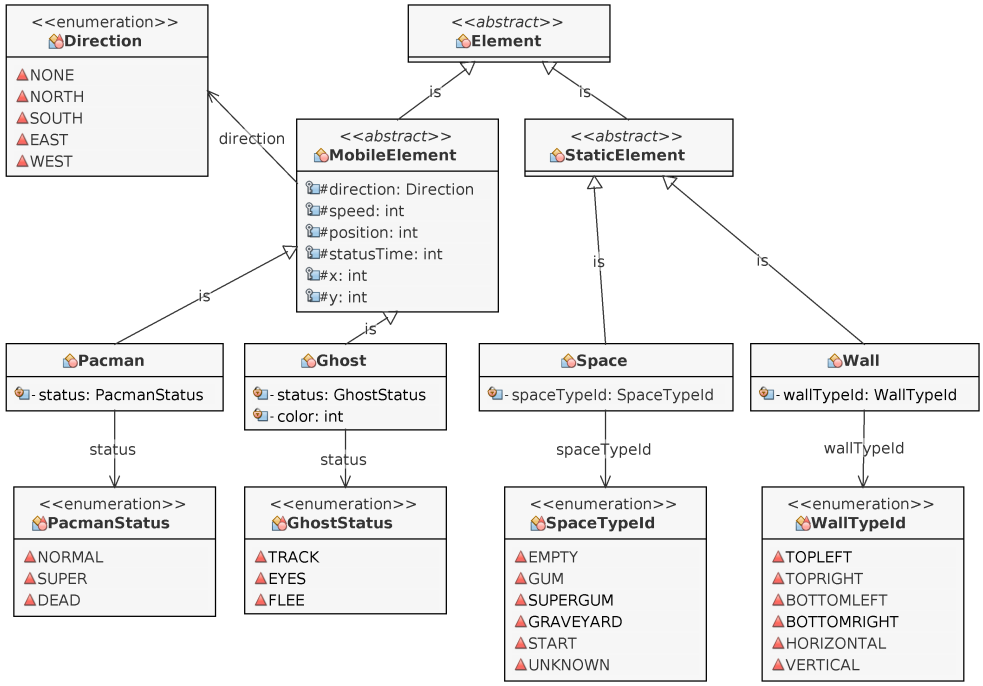


The implementation with the AWT library is also preserved, and placed in the `gui.awt` package. It is possible to use another implementation - all the following does not depend on a particular implementation, as long as the new features of the facade are added.

The state of the Pacman game, presented in Chapter 2, is reorganized into two parts: a first part with the containers, placed in the `state` package (here without the `WorldIterator` class for clarity):



The second part with the elements is placed in the `state.element` package:



4.2.1 Render a view of the game state

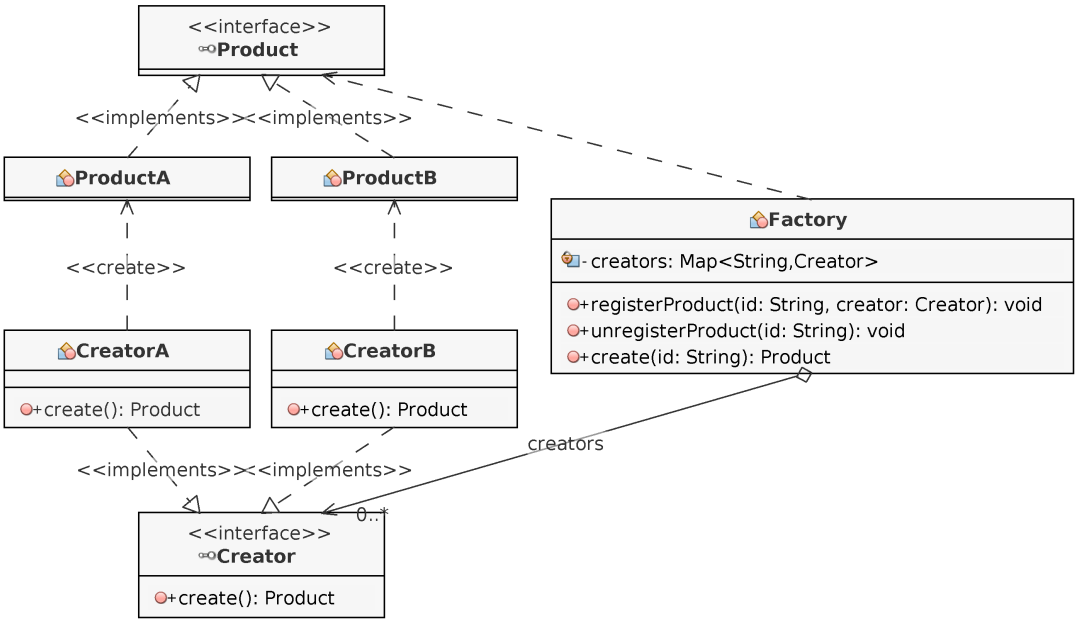
4.2.1.1 Initialize the world (Abstract Factory Pattern)

To render a state of the game, we must fill it with data that represents a level of Pacman. In the examples in Chapter 3, a simple native array was used:

```
static final int[][] level = new int[][] {
    { 15,11,11,11,11,11,11,11,16 },{ 12,5,3,3,3,3,3,3,12 },
    { 12,3,15,11,11,11,16,3,12 },{ 14,3,13,11,11,11,14,3,13 },
    { 3,3,3,3,3,3,3,3,3 },{ 11,11,11,11,11,11,11,11,11 }
};
```

This array is very good for presenting the display, but does not have all the advantages of a state like the one designed in Chapter 2. The goal now is to use it to initialize the world's data. We have to build instances of the `Element` class according to the different codes. For example, for code 15, we want to make an instance of the `Wall` class with the `WallTypeId.TOPLEFT` type. A naive solution is to program a large `switch ... case`. In this section, a richer approach is proposed, based on the *Abstract Factory Pattern*.

Abstract Factory Pattern



- The `Product` interface defines the objects that you want to make. In the illustration above, there are two implementations: `ProductA` and `ProductB`.
- The `Creator` interface is used to define classes that can build `Product` implementations. For example, the `CreatorA` class makes an instance of `ProductA` with the `create()` method. We present the simplest case here: there are no arguments to the `create()` methods. It is possible to propose methods of creation with arguments.
- The `Factory` class is the `Product` factory. There are several ways to define and implement it - here, the most common case is presented. The goal of this factory is to be able to build a product based on an identifier. In this example, the identifier is a string (`String`). Other types are possible, as well as any form of setting or context that influence the nature of the created object. All associations between an identifier and a product creator are stored in the `creators` attribute. The class has the following methods:
  - `registerProduct()` method: allows to define an association between an identifier and a product to be manufactured;
  - `unregisterProduct()` method: deletes an existing association;
  - `create()` method: returns a new product based on an identifier. This method must be consistent with that of the `Creator` interface.

Here is an example that illustrates the use of this factory:

```
Factory factory = new Factory();
factory.registerProduct("A", new CreatorA());
factory.registerProduct("B", new CreatorB());
factory.create("A").show();
factory.create("B").show();
```

Line 2 adds the association between the string “A” and the creation of `ProductA`. Similarly, line 3 adds the association between the string “B” and the creation of `ProductB`.

Line 4 makes a product using the string “A”, then invokes the `show()` method of the newly created product. Line 5 does the same with string “B”.

Knowing that the `show()` method of the `ProductA` class displays `Product A`, and that `ProductB` displays `Product B`, the following is displayed:

```
Product A
Product B
```

The first advantage of this pattern lies in the separation of the different features:

- The way a product is made is implemented in a specific class. For example, the `CreatorA` class creates `ProductA` objects;
- The associations between the identifiers and the creators are chosen by the user of the factory;
- A factory is interchangeable: a user can instantiate different factories, and switch from one to another to modify the behavior of the application.