Capítulo 5
Vista rápida de algunos design patterns

1. Introducción

Las nociones de programación orientada a objetos que se han visto hasta ahora, son piezas elementales. Se trata del material básico para construir de manera robusta y eficiente, soluciones para resolver problemas comunes de programación: cómo realizar el procesamiento en una jerarquía de clases sin tener que modificar su implementación, cómo asegurar que solo hay una instancia única de tal clase en todo el programa, cómo cambiar la forma en que se representan dichos datos utilizando la menor cantidad de código posible, etc.

Al combinar las piezas básicas de la POO según ciertos esquemas ya probados, las clases y las relaciones entre ellas pueden formar herramientas fantásticas que responden con robustez y elegancia a estos problemas comunes de programación. Estas combinaciones se denominan "design patterns" o "patrones de diseño" en español. Como el término inglés es el más extendido, será el que se utilizará en el resto de este capítulo.

Un design pattern no es un fragmento de código que podamos copiar y pegar en un programa, para que funcione como se desea. Tampoco es solo una librería para importar. Algunas librerías pueden facilitar la implementación del design pattern deseado, pero de hecho, depende del desarrollador codificarlo adaptándolo a las estructuras y entornos ya existentes. Se puede comparar con una estructura de algoritmo, sin los detalles de implementación. Se trata de ensamblar unas clases genéricas de una determinada manera, para ver aparecer una entidad que resuelve un problema. Depende del desarrollador comprender perfectamente el funcionamiento del design pattern, para transcribirlo a su lenguaje de programación, integrarlo en el contexto empresarial en el que trabaja y respetar las limitaciones técnicas y arquitectónicas de su software.

Los design patterns aparecieron bastante temprano en la programación, pero no fue hasta la década de 1990 cuando se les aplicó un formalismo real. Hoy en día, existen docenas de design patterns aplicables a la programación orientada a objetos. Soluciones casi llave en mano para problemas informáticos comunes, mejoran enormemente la velocidad de programación y la solidez de los programas, porque han sido cuidadosamente pensados, probados y aprobados. Hay muchos libros que se dedican a ellos. Es importante comprender los principales design patterns para tener un buen dominio de la orientación a objetos.

Las siguientes secciones presentan algunos. Fueron elegidos para aplicar los principios POO vistos anteriormente en este libro.

2. Singleton

El uso de variables globales es una práctica que se debe evitar en la mayoría de los casos: contaminan el espacio de nombres global, ocupan recursos durante toda la vida del programa y son difíciles de controlar, ya que todo el mundo puede acceder a ellas. Sin embargo, en algunos casos específicos, es mucho más conveniente poder acceder a la información de forma global, en lugar de transferirla de un objeto a otro.

```
class Madrid:

    def __init__(self, sol):
        self.sol = sol

    def clima(self):
        if (self.sol):
           return "Soleado"
        return "Nublado"

class Espana:

    def __init__(self, sol):
        self.sol = sol
        self.ciudades = [Madrid(sol)]

class Europa:

    def __init__(self, sol):
        self.sol = sol
        self.pais = [Espana(sol)]

class Tierra:

    def __init__(self, sol):
        self.sol = sol
        self.continentes = [Europa(sol)]

class SistemaSolar:

    def __init__(self, sol):
        self.sol = Sol()
        self.tierra = Tierra(sol)
```

En este ejemplo, la instancia de Sol pertenece lógicamente al sistema solar. También vemos que cada lugar representado por las clases declaradas en el ejemplo, puede tener acceso a esta instancia -única- de Sol, para realizar eventuales procesamiento (visualización del clima en una ciudad, por ejemplo). Sin embargo, si queremos evitar tener una instancia global, es imperativo que cada clase que deba acceder al Sol reciba esta instancia en su constructor, para almacenarla y usarla. De ahí la cadena de constructores que toman el Sol como segundo argumento. Usar un objeto global simplificaría enormemente el código:

```
class Madrid:

    def clima(self):
        if (sol):
            return "Soleado"
        return "Nublado"

class Espana:

    def __init__(self):
        self.ciudades = [Madrid()]

class Europa:

    def __init__(self):
        self.pais = [Espana()]

class Tierra:

    def __init__(self):
        self.continentes = [Europa()]

class SistemaSolar:

    def __init__(self):
        self.tierra = Tierra()

sol = Sol()
t – Madrid()
print(t.clima())
>>> Soleado
```

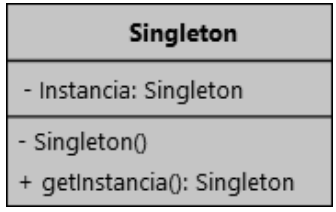
No obstante, un objeto global, por tanto accesible en todas partes, puede proporcionar un cierto control a través de métodos de acceso para evitar una manipulación incorrecta. Pero globalidad debe ir de la mano de unicidad. De hecho, sin la seguridad de que el objeto es el mismo durante toda la vida del programa, su usuario no puede estar seguro de su estado, de los valores que contiene, etc.

Tomemos, por ejemplo, un objeto que gestiona la configuración del programa (nombres de usuario, colores, cualquier cosa que podamos esperar encontrar en un menú de Opciones o Configuración). Este objeto, dadas sus ramificaciones, debe ser accesible en todas partes. Sin embargo, también debe ser único por una razón obvia: queremos que la configuración del programa se lea solo una vez al inicio (gracias a un archivo de configuración, una lectura en un registro, etc.) y que sea coherente durante todo su uso.

Otro ejemplo de un objeto global que debe ser único podría ser un recurso de hardware, como una tarjeta gráfica o un dispositivo conectado a cualquier puerto. En este caso, la unicidad es fundamental porque no queremos tener que cargar un controlador varias veces, arriesgarnos a dirigirnos al dispositivo equivocado o enviar órdenes contradictorias o simultáneas. El desarrollador quiere acceder a este recurso de forma sencilla y sin miedo a equivocarse.

En este tipo de problemas relativamente extendidos, es donde se utiliza el design pattern Singleton.

La definición de Singleton es simple: se trata de una clase que solo se puede instanciar una sola vez a lo largo de todo el programa. Aquí está su representación UML:



Para garantizar que no se pueda crear una instancia de Singleton más de una vez, el constructor de la clase es privado: nadie puede acceder a él, por lo que nadie puede crear instancias excepto la clase misma.

Para recuperar la instancia única, se propone un método de clase (y no de instancia). Es un método de clase porque, dado que es la única forma de obtener una instancia, no es posible llamarlo a través de una instancia.

La única instancia de Singleton se almacena como un atributo privado de la clase (queremos que nadie más que Singleton acceda a ella). El método que lo devuelve primero prueba si esta instancia es nula. Si es así, esta es su primera llamada y la instancia aún no se ha creado. Aquí es cuando se crea una instancia de la clase utilizando el constructor privado. El método solo tiene que devolver esta instancia para que se pueda usar.

Es todo. El constructor privado, la instancia única también privada y el método de clase para acceder a esta instancia, forman un Singleton.

La implementación en Python no es tan trivial, simplemente porque en Python, la noción de visibilidad no existe: todos los miembros de una clase son públicos. Sin embargo, el hecho de tener un constructor público impide la implementación del principio Singleton, porque cualquiera puede llamarlo y, en consecuencia, crear nuevas instancias. Por lo tanto, debemos profundizar un poco más en la clase `Singleton`, que se debe implementar y modificar de la misma manera que se crea la clase, gracias a las metaclases.

En el capítulo Los conceptos de POO con Python, se mostró que en Python, una clase se usa para instanciar objetos, pero también es en sí misma un objeto. La prueba: es muy posible asignar una clase a una variable:

```
class Test :
    pass

>>> mi_prueba = Test
>>> print(mi_prueba)
<class '__main__.Test'>
```

Dado que una clase es un objeto, necesariamente tiene una clase para definirla. Veamos cuál es el tipo real de clase usando la función integrada `type`:

```
>>> type(1)
<class 'int'>
>>> type(Test)
<class 'type'>
```

Por tanto, una clase es de tipo `type`. De hecho, en Python, `type` es la metaclase que permite generar todas las clases. En otras palabras, cada clase es una instancia de `type`.

Se puede hacer que el constructor de una clase sea privado, modificando la metaclase de la clase `Singleton`, es decir, modificando la forma en que se define la clase. Para hacer esto, todo lo que tiene que hacer es definir una clase derivada de `type`, que es la clase que puede instanciar cualquier clase. Por lo tanto, por herencia, esta clase derivada también podrá instanciar clases. Y dado que se trata de una clase que implementamos, es posible anular los métodos de instanciación de clases:

```
class MetaSingleton(type):
    def __call__(cls):
        pass # Nada por el momento.
```

La clase `MetaSingleton` hereda de `type`, lo que le da el poder de instanciar clases. Cuando instanciamos `Singleton` a través de la instrucción `Singleton()`, se llamará al método `__call__()` con la clase `Singleton` como parámetro (la clase como objeto de tipo `class`).

```
# Estas dos líneas son equivalentes
s = Singleton()
s = Singleton.__call__()
```

Ahora que se ha declarado `MetaSingleton`, debemos especificar a la clase `Singleton` que usa `MetaSingleton` como metaclase. Por lo tanto, `MetaSingleton` realizará la instanciación de la clase `Singleton`. Esto se puede hacer especificando el argumento `metaclass` en la línea de declaración de `Singleton`:

```
class Singleton(metaclass=MetaSingleton):
    pass
```