Modelando Dragones

Roni Barylko

22 de julio de 2018

1. Introducción

La idea de este pequeño documento es hacer un paso a paso de cómo se modela en un lenguaje como Python. Para esto, propondremos un juego simple, lo imaginaremos en términos ajenos a la computación y, luego, lo pasaremos a Python.

Es importante que tengamos en cuenta varias cuestiones antes de comenzar. En primer lugar, a la hora de modelar no hay verdades absolutas: hay decisiones buenas y malas. Es posible que planteen un modelo propio, sin errores, y luego al comparar con otra persona, tengan dos enfoques completamente distintos. Esta bien, eso puede y va a ocurrir.

En segundo lugar, no sobrepiensen el modelado. Supongamos que estamos modelando el sistema de notas de la Facultad, y para eso tienen que diseñar la clase Alumno, no tendría sentido que le pusieran atributos como peso, altura, color de pelo, porque son variables que no entran dentro de la función que debe cumplir nuestro programa, y lo maś probable es que los lleve a confundirse y sobrepensar todo el diseño. Tengan cuidado, creen las clases necesarias y suficientes, sin diseñar de más ni de menos. **Todos los problemas que vayamos a querer resolver tienen un dominio específico**

En tercer lugar, sean críticos con su código y rediseñen. No esperen tener un problema resuelto de forma instantánea, ni que la primera solución que les venga a la mente sea la indicada. Les va a pasar de tener un diseño hecho en un 75 %, y de la nada darse cuenta que les falta agregar una función importantísima que, por la forma en la que modelaron, no tienen donde meter. Mala suerte, a diseñar de nuevo. **Siempre busquen tener un modelo consistente y escalable.** Menos no.

¿Entendido? Fantástico. Vamos a profundizar entonces en estos dos conceptos.

1.1. Consistencia y escalabilidad

¿De qué hablamos cuando hablamos de un modelo consistente y escalable? Básicamente, de un modelo que se acopla bien al mundo real, y que es capaz de crecer sin necesidad de ser cambiado. Vamos a explicarlo con un poco más de detalle.

Empecemos hablando de la **consistencia**. Supongamos que tenemos que modelar la clase Manzana. Pondríamos como atributo su color, y su tamaño. A su vez, tenemos la clase Persona, que es capaz de comer todo tipo de frutas. **¿Tendría sentido que, a la hora de comer una manzana, tengamos que llamar a la clase Manzana con la función** `comer()`?. No, porque las manzanas no saben comer: las personas comen a las manzanas. Por lo tanto, tendríamos que colocar la función `comer()` en la clase Persona, y hacer algo del tipo `persona.comer(manzana)`

¿Se entiende? Queremos que nuestro modelo se comporte de manera consistente a la realidad. Ojo, no significa que tenemos que tener un modelo realista: simplemente, tenemos que ser fieles a lo que construimos Podríamos hacer que nuestra Manzana se comiera a si misma, pero no sería una Manzana común y corriente, sino una fruta superpoderosa. Podríamos hacer que los Perros hagan ruido de pato, pero no podemos llamar a `perro.hacerRuidoDePato(pato)` y decir que .[el] pato está haciendo ruido", porque habría un problema de consistencia entre lo que decimos(o imaginamos) y lo que realmente está pasando.

Pasemos ahora a considerar la **escalabilidad**. La idea es que, al armar nuestro modelo, no lo pensemos para una cantidad determinada de situaciones. Por ejemplo, si planteamos un juego de mesa, no tiene sentido que diseñemos todo considerando tan solo la posibilidad de realizar una partida con dos jugadores, porque si en algún momento se desea extender el juego, deberemos repensar el diseño y cambiar cada una de las clases. **Evitar los problemas de escalabilidad se reduce, básicamente, a quitar las limitaciones cuantitativas de nuestro modelo.**

Vamos a un ejemplo más complejo. Supongamos que, a la Persona que comía manzanas, le queremos colocar cada una de las comidas diarias como una variable: va a tener desayuno, almuerzo y cena. Perfecto, le agregamos esas tres variables. Pasan dos meses y nos damos cuenta que no estamos considerando la merienda, entonces se la agregamos. Pasan dos semanas más, y nos encontramos con que hay personas que se levantan a la medianoche a comer: le agregamos también esa comida. Pasan dos años y por decreto nacional se elimina la existenca del desayuno.

¿Se ve el problema? Cada vez que alguien quiere agregar una comida nueva, tenemos que entrar a cambiar la clase Persona, para agregar o quitarle variables particulares que, a la larga, son todas partes de un mismo conjunto. Entonces, nos bastaría con crear la clase ComidaDiaria y que cada instancia de la clase Persona tenga una lista con sus comidas. Chau, un problema menos.

2. Modelado

Ya tenemos los principios básicos para un modelado correcto. Ahora, es solo cuestión de imaginar lo que queremos hacer, diseñarlo y llevarlo a Python.

2.1. Definiendo el juego

Vamos por el principio: tenemos que plantear qué es lo que queremos modelar. La idea con la que crearemos el juego es la siguiente. Tendremos dragones y domadores, repartidos por un mundo imaginario. El objetivo de los domadores será cazar a los dragones. Cada uno de estos dragones tendrá una capacidad de ataque, que se medirá en base a la edad del dragon y a la capacidad de entrenamiento de su domador. Para cazar un dragón, el domador tiene que encontrarse en la misma posición que él y, en caso de que no esté domado, lo domará automáticamente. El juego termina cuando todos los dragones están domados, y gana aquel domador cuyos dragones sumen la mayor capacidad de ataque.

2.2. Modelando las clases

Dado el modelo, tendremos que diseñar cuatro clases: las dos obvias, que son **Dragón** y **Domador**; la tercera, que será el **Mundo** y nos permitirá definir posiciones y demás; y la cuarta, la más dificil de notar tal vez, que será el **Juego**, que nos servirá para avanzar los turnos y terminar la partida cuando haga falta. Para evitarnos crear la clase Posición, usaremos un valor entero (en vez de una coordenada (x,y)), por lo que tendremos la posición 0, la posición 10, etc.

2.2.1. Dragón

Por la forma en la que creamos el juego, nuestro Dragón no tendrá un comportamiento muy extenso. Bastará con darle los atributos correspondientes a la posición actual y la edad, y una variable que nos diga si está domado o no, además de la funcionalidad necesaria para que sepa cómo moverse, cuál es su capacidad de ataque y cómo cambia su estado cuando es domado. Entonces:

Atributos

- posicionActual
- edad
- estaDomado

Funciones de observación

- obtenerPosicionActual()
- obtenerEdad()
- estaDomado()

Otras funciones

- capacidadDeAtaque(capacidad)
- mover()
- cambiarEstado()

2.2.2. Domador

Para esta clase, tendremos que contar con una lista de los dragones domados, que nos permitirá saber al final del juego quién es el ganador. Además, contaremos con la posición y la capacidad de entrenamiento. En cuanto a la funcionalidad extra, bastará con que le digamos al domador como moverse y como cazar dragones, además de tener una función que nos diga la capacidad de ataque del Domador (es decir, la suma de capacidades de todos sus dragones)

Atributos

- posicionActual
- listaDragones
- capacidad

Funciones de observación

- obtenerPosicionActual()
- obtenerListaDragones()
- obtenerCapacidad()

Otras funciones

- mover()
- cazarDragon(dragon)
- capacidadTotal()

2.2.3. Mundo

Dado que contamos con la clase Juego, el Mundo servirá simplemente como representación del tamaño que tendrá nuestro juego. De hecho, podríamos no crearlo, pues al tomar la decisión de que las posiciones se representan con un número, nuestro mundo contará tan solo con el atributo tamaño, y con la funcionalidad obtenerTamaño. Por ende, podría ser simplemente un parámetro que le pasamos al juego y ya. Tenemos aquí una mala decisión de diseño: dado que al crear nuestras clases fuimos tomando decisiones que nos guiaron para un camino, esta clase pasó a ser inutil y, por ende, al crearla tan solo estaríamos agregándole complejidad a nuestro problema

Es importante que noten estas cosas, tanto cuando se vuelve necesario quitar una clase como cuando se vuelve necesario agregarla. Si hubieramos decidido usar posiciones con coordenadas, tendríamos que haber creado la clase Posición, aún si ya estabamos en la mitad del desarrollo y eso implicaba retroceder algunos pasos.

De este modo, no crearemos la clase Mundo, y delegaremos la responsabilidad del tamaño del tablero a la clase Juego. Esto lo podemos hacer porque, en este

caso, el Mundo se representa con un atributo. Mundo es 100, es 50, es 30. Alcanza con tener un número guardado en la clase Juego y llamarlo **mundo**. No dejamos de representarlo, solo simplificamos nuestro modelo para evitar tener una clase innecesaria.

2.2.4. Juego

2.3. Tiempo de Python