Trabajo Práctico 2
Programación Orientada a Objetos

Paradigmas de Lenguajes de Programación — 1er cuat. 2023

Fecha de entrega: 15 de Junio de 2023

1. Introducción

En este taller modelaremos el comportamiento de una colonia de *slimes*. Los *slimes* son unas criaturas gelatinosas que se dedican principalmente a andar por ahí comiendo todo lo que encuentran (existen reportes de *slimes* que hacen muchas otras cosas, pero no son miembros de la colonia que estamos estudiando). Los *slimes* no sienten dolor pero pueden recibir daño, lo cual les quita energía. Si bien suelen recuperar energía comiendo, ocasionalmente algunos slimes atacan a otros para robarles energía, o los retan a duelo para ver quién es más fuerte.

2. Ejercicios

Ejercicio 1

Definir la función constructora `Slime(ataque, defensa)` que permita crear *slimes* con capacidades de ataque y de defensa determinadas. Los *slimes* nacen con la cantidad máxima de energía (definida como 100 según la escala de medición que utiliza nuestro equipo de investigación) y con nivel 1 (volveremos sobre esto más adelante).

Tanto el ataque como la defensa deben ser mayores o iguales que 1. Si se recibe un valor inferior, se deberá asignar el valor 1 al atributo correspondiente.

Ejercicio 2

Los *slimes* deben poder responder al mensaje `actualizarEnergía(incremento)`. Este mensaje se utilizará para registrar la energía ganada o perdida por un *slime*. Por ejemplo, si come obtendrá un incremento positivo, y si sufre daño obtendrá un incremento negativo. Es

importante notar que la energía de un *slime* nunca puede ser menor que 0 ni mayor que el valor máximo.

Escribir el código necesario para reflejar este comportamiento.

Ejercicio 3

Escribir el código necesario para que los *slimes* puedan atacarse entre sí. Un slime solo puede atacar a otro si dispone de algo de energía. Al atacar causa un daño determinado por su nivel y su capacidad de ataque, así como también el nivel y la defensa de su presa. La fórmula para calcular el daño ya se encuentra definida. El atacante recibe una cantidad de energía igual al daño realizado. Desde ya, no se puede robar más energía de la que tiene la presa (esto ya está contemplado en la fórmula del daño).

Ejercicio 4

Llegó la hora de modelar los duelos entre *slimes*. Un *slime* debe poder retar a otro a duelo, atacándolo y permitiendo que el otro contraataque.

Al recibir el mensaje `duelo(oponente)`, el retador deberá primero atacar, luego recibir el contraaataque y finalmente devolver el ganador del duelo, el cual será el retador si este logra que su oponente termine con menos energía de la que tenía al empezar, y su contrincante en caso contrario. Como premio, el ganador del duelo aumenta en 1 su nivel.

Ejercicio 5

Nuestro equipo de investigación ha descubierto que existen *slimes* con la capacidad de curar a otros o incluso a sí mismos. Para incluirlos en nuestro modelo, se deberá definir la función constructora `Slime(ataque, defensa, poder)`, la cual además del ataque y la defensa registra también el poder de curación del *slime* en cuestion. No se debe repetir el código de la función constructora para *slimes* genéricos. Se recomienda utilizar la función `call(receptor, argumentos)` que se encuentra definida en `Function.prototype` para inicializar los atributos comunes.

Similitudes y diferencias con otros slimes: los *slimes* sanadores son, antes que nada, *slimes*, y deben poder hacer las mismas cosas que hacen los otros. Además, deben poder responder al mensaje `curar(paciente)`, el cual restaura hasta una cantidad de puntos de energía equivalente al producto del nivel del sanador por su poder de curación. Naturalmente, la energía del paciente no podrá superar el valor máximo.

Dado que la forma de curar es común a todos los slimes sanadores, se espera que este comportamiento se defina de manera acorde y no se incluya explícitamente como atributo de cada *slime* sanador.

Ejercicio 6

Los *slimes* se reproducen de manera asexual. Esto no significa que puedan hacerlo en cualquier momento, ya que procrear un nuevo *slime* consume una gran cantidad de energía. De hecho, consume la parte entera de la razón entre la energía máxima y el nivel del progenitor (básicamente $\lfloor 100/nivel \rfloor$).

Si la energía del *slime* no es mayor que la requerida, no podrá reproducirse.

Por otro lado, si tiene éxito, producirá una cría de nivel 1 con la máxima energía, y con el resto de sus atributos y capacidades iguales a los del *slime* original.

1. Permitir que los *slimes* respondan al mensaje `reproducirse()`, que refleje el comportamiento indicado anteriormente. En caso de éxito, se deberá devolver la cría. En caso contrario, no se devuelve nada.

2. Permitir que los *slimes* respondan al mensaje `esDescendienteDe(otro)`, que indica si el receptor es o no descendiente del *slime* indicado.

   Puede suponerse que `otro` será siempre un *slime*.

3. Pautas de Entrega

Todo el código producido por ustedes debe estar en el archivo `taller.js` y es el **único** archivo que deben entregar. Para probar el código desarrollado abrir el archivo `TallerJS.html` en un navegador web. Cada función o método (incluso los auxiliares) asociado a los ejercicios debe contar con un conjunto de casos de test que muestren que exhibe la funcionalidad esperada. Para esto se deben modificar las funciones *testEjercicio* en el archivo fuente *taller.js*, agregando todos los tests que consideren necesarios (se incluyen algunos tests de ejemplo). Pueden utilizar la función auxiliar *res.write* para escribir en la salida. Si se le pasa un booleano como segundo argumento, el color de lo que escriban será verde o rojo en base al valor de dicho booleano. Se debe enviar un e-mail a la dirección plp-docentes@dc.uba.ar. Dicho mail debe cumplir con el siguiente formato:

- El título debe ser `[PLP;TP-POO]` seguido inmediatamente del nombre del grupo.
- El código JavaScript debe acompañar el e-mail y lo debe hacer en forma de archivo adjunto con nombre `taller.txt` (pueden cambiarle la extensión para que no sea detectado como posible software malicioso).
- El código entregado **debe** incluir tests que permitan probar las funciones definidas.
- El código de cada ejercicio debe escribirse dentro de la sección correspondiente. No se permite modificar el código de los ejercicios anteriores.

No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté **adecuadamente** comentado (son comentarios adecuados los que ayudan a entender lo que no es evidente o explican decisiones tomadas; no son adecuadas las traducciones al castellano del código). Los objetivos a evaluar son:

- Corrección.
- Declaratividad.
- Prolijidad: evitar repetir código innecesariamente y usar adecuadamente los métodos previamente definidos.

Importante: se admitirá un único envío, sin excepción alguna. Por favor planifiquen el trabajo para llegar a tiempo con la entrega.

Referencias del lenguaje JavaScript

Como principales referencias del lenguaje de programación JavaScript, mencionaremos:

- **W3Schools JavaScript Reference:** disponible online en: https://www.w3schools.com/jsref/default.asp.
- **MDN Web Docs: Mozilla - Referencia de JavaScript**: disponible online en: https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia.