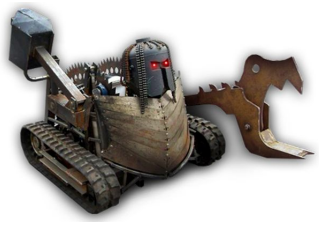
Trabajo Práctico 3
Programación Orientada a Objetos

Paradigmas de Lenguajes de Programación – 2° cuat. 2016

Fecha de entrega: 10 de noviembre



Objetivo

El objetivo de este TP es implementar un simulador sencillo de luchas de robots[1] .

La lucha de robots consiste en un enfrentamiento entre dos robots armados controlados remotamente, los cuales se atacan mutuamente hasta que uno de los dos es destruído.

Todo robot tiene un armazón de un material y dureza determinados, y un arma cuya efectividad varía de acuerdo al armazón del oponente. Cada arma tiene una potencia, que define la intensidad de sus ataques, y una cierta velocidad, que le permite realizar una determinada cantidad de ataques sucesivos en un segundo.

Tipos de ataques

Un arma puede realizar simultáneamente ataques de distintos tipos (por ejemplo fuego, impacto y/o abrasión).

A su vez, un armazón puede ser más o menos vulnerable a ciertos tipos de ataques. Por ejemplo, la madera es muy vulnerable al fuego, mientras que el metal es muy resistente a él pero no es tan resistente a la abrasión.

En este TP los tipos de ataques serán distinguibles por sus nombres.

Ejercicio 1. Definir la clase `TipoDeAtaque`, cuyas instancias respondan correctamente los mensajes `=` y `hash`. La clase `TipoDeAtaque` debe poder responder los mensaje `fuego`, `impacto` y `abrasion` para devolver una instancia del tipo de ataque correspondiente. Al finalizar este ejercicio deberán pasar el test de la clase `TestsEj1`.

[1] Para más detalles, ver https://en.wikipedia.org/wiki/Robot_Wars_(TV_series) .

Armazón

Ejercicio 2. Implementar la clase `Armazón`, que permita modelar los diferentes armazones que existen. Cada uno posee un nombre (String).

Deberán incluir los siguientes métodos (de clase o de instancia, según corresponda):

- `nombre: vulnerabilidades: dureza:`, que crea un nuevo armazón con los parámetros especificados.
- `nombre`, `vulnerabilidades` y `dureza`, que devuelven los datos correspondientes del armazón receptor.
- `vulnerabilidadA:`, que indica la vulnerabilidad del armazón a un tipo de ataque determinado (si la vulnerabilidad a este de ataque no fue definida, devuelve 1, que es el valor por defecto).
- `madera:`, `metal:` y `acrilico:`, que crean armazones con los nombres correspondientes, con la dureza pasada como parámetro, y con las siguientes vulnerabilidades.

Armazón \ Ataque	**Fuego**	**Impacto**	**Abrasión**
Acrílico	1	1/2	2
Madera	2	1	1/2
Metal	1/4	1/2	1

Al finalizar este ejercicio deberán pasar el test de la clase `TestsEj2`.

Arma

Ejercicio 3. Implementar la clase `Arma`, que permita modelar los tipos de armas que existen. Cada una posee un nombre (String). Además, cada arma puede realizar ataques de uno o más tipos a la vez. Por ejemplo, el **lanzallamas** realiza solamente ataques de **fuego**, mientras los ataques de la **sierra** son **1/3 impacto y 2/3 abrasión**, y los del **martillo** son **2/3 impacto y 1/3 abrasión**.

Definir los siguientes métodos (de clase o de instancia, según corresponda):

- `nombre: ataques: potencia: velocidad:`, que permite crear un arma con los parámetros indicados.
- `nombre`, que devuelve un `String` con el nombre del arma receptora, por ejemplo 'martillo'.
- `potencia`, que devuelve la potencia de un arma (un valor numérico).
- `velocidad`, que devuelve la cantidad de ataques por seguno que el arma puede realizar.
- `lanzallamas: velocidad:`, `martillo: velocidad:` y `sierra: velocidad:`, que crean las 3 armas básicas con las potencias y velocidades pasadas como parámetro, y los tipos de ataque mencionados arriba.

Importante: el modelo debe ser extensible, permitiendo definir no solo nuevas armas y armazones, sino también nuevos tipos de ataque, sin que esto implique modificar lo ya definido.

Al finalizar este ejercicio deberán pasar el test de la clase `TestsEj3`.

Robot

Ejercicio 4. Implementar la clase Robot, con los siguientes métodos (de clase o de instancia, según corresponda):

- `nombre`, `armazon`, `arma` y `daño` (o `danio` para evitar problemas de codificación), que permitirán observar el estado actual de un robot (el nombre es un String que varía de un robot a otro, aunque nada impide que haya dos robots con el mismo nombre).
- `cortados`, `dragon` y `tronquito`, que permitan crear los robots por defecto, con las siguientes características.

Nombre	Armazón	Dureza	Arma	Potencia	Velocidad
#Tronquito	Madera	10	Martillo	40	2
#Dragon	Acrílico	20	Lanzallamas	60	15
#Cortados	Metal	50	Sierra	45	10

Importante: todo nuevo robot que se cree (tanto los básicos como cualquier otro) debe tener daño 0.

Al finalizar este ejercicio deberán pasar el test de la clase `TestsEj4`.

Batallas

Ejercicio 5.

- Implementar el método `atacar: oponente` en la clase Robot teniendo en cuenta los siguientes aspectos:
  - El efecto de un ataque es causar daño al oponente. Pensar de qué manera puede implementarse para que el oponente refleje el daño recibido, y a quién le corresponde la responsabilidad de calcular este daño (pista: uno nunca conoce realmente el daño que causa a otros).
  - La fórmula para calcular el daño producido por un ataque es la siguiente:

$$\sum_{\text{tipos de ataque del agresor}} \text{efectividad} * \text{potencia} * \text{velocidad} * \text{valor del ataque/dureza del armazón}$$

    donde la efectividad es la vulnerabilidad del armazón del oponente al tipo de ataque correspondiente.

- Definir el mensaje `batallaContra: oponente` tal que, al recibirlo, el robot y su oponente se ataquen mutuamente (por turnos) hasta que el daño de uno de ellos llegue a 100 o más. Un robot cuyo daño alcance el valor de 100 no puede continuar la batalla, y se considera derrotado. El robot receptor es el que ataca primero (siempre y cuando esté en condiciones de atacar, ya que en caso contrario la batalla está perdida antes de empezar). El resultado de este mensaje debe ser el robot ganador.

Una vez concluido este ejercicio deberán poder pasar todos los tests.

Pautas de entrega

El entregable debe contener:

- un archivo `.st` con todas las clases implementadas
- versión impresa del código, comentado adecuadamente (puede ser el propio .st sin los tests)
- **NO** hace falta entregar un informe sobre el trabajo.

Se espera que el diseño presentado tenga en cuenta los siguientes factores:

- definición adecuada de clases y subclases (si corresponde), con responsabilidades bien distribuidas
- uso de polimorfismo y/o delegación para evitar exceso de condicionales (no usar más de un **ifTrue:**, **ifFalse:** o **ifTrue:ifFalse:**)
- intento de evitar código repetido utilizando las abstracciones que correspondan.

Consulten todo lo que sea necesario.

Consejos y sugerencias generales

- Lean al menos el primer capítulo de *Pharo by example*, en donde se hace una presentación del entorno de desarrollo.
- Explorar la imagen de Pharo suele ser la mejor forma de encontrar lo que uno quiere hacer. En particular tengan en cuenta el buscador (`shift+enter`) para ubicar tanto métodos como clases.
- No se pueden modificar los test entregados, si los hubiere, aunque los instamos a definir todos los tests propios que crean convenientes.

Importación y exportación de paquetes

En Pharo se puede importar un paquete arrastrando el archivo del paquete hacia el intérprete y seleccionando la opción "`FileIn entire file`". Otra forma de hacerlo es desde el "`File Browser`" (botón derecho en el intérprete > `Tools` > `File Browser`, buscar el directorio, botón derecho en el nombre del archivo y elegir "`FileIn entire file`").

Para exportar un paquete, abrir el "`System Browser`", seleccionar el paquete deseado en el primer panel, hacer click con el botón derecho y elegir la opción "`FileOut`". El paquete exportado se guardará en el directorio `Contents/Resources` de la instalación de Pharo (o en donde esté la imágen actualmente en uso).