Curso 2018 / 2019

Práctica 1:

Ejercicios básicos de programación orientados a objetos

Prácticas de la asignatura PDOO

Durante el desarrollo de las prácticas de esta asignatura se implementará en dos lenguajes de programación orientada a objetos (**Java y Ruby**) y por etapas, el juego DeepSpace del cual ya dispones de las reglas.

En esta práctica se tomará un primer contacto práctico con el paradigma de la orientación a objetos y los lenguajes de programación Java y Ruby. Para ello se realizará la implementación de una serie de clases y tipos de datos enumerados simples que formarán parte del diseño completo del citado juego DeepSpace.

El desarrollo de cada práctica (salvo la última) se hará en ambos lenguajes de programación: Java y Ruby, siendo ambos lenguajes igual de importantes en el desarrollo de la asignatura. Debido a las características que los diferencian, el buen conocimiento de ambos lenguajes será muy positivo en tu formación sobre el paradigma de la orientación a objetos.

Durante el desarrollo de las prácticas hay que ser muy estricto con el nombre que se le da a cada elemento, siguiendo fielmente los guiones y diagramas que se entreguen, haciendo distinción igualmente entre mayúsculas y minúsculas. Ten en cuenta que cada alumno forma parte de un equipo de desarrollo junto con los profesores. Parte del código será proporcionado por los profesores. Para que cada vez que se junte código de distintos desarrolladores no haya conflictos de nombres ni otros errores, todos debemos ceñirnos a la documentación común con la que trabaja el equipo.

Ten especial cuidado si en tu ordenador personal trabajas en Windows, ya que en este sistema operativo no se distingue entre mayúsculas y minúsculas y en Linux sí. Puedes tener problemas en los exámenes si no prestas atención a cómo nombras los archivos.

Herramientas

Para el desarrollo de estas prácticas se utilizará el entorno de desarrollo **NetBeans**. Puedes descargar la última versión estable en: https://netbeans.org. Asegúrate de que la versión descargada incluya al menos soporte para Java SE.

Para que este entorno de desarrollo incluya soporte para el lenguaje **Ruby** será necesario instalar un **plugin** que puede descargarse de Prado. Se encuentra en la sección de Prácticas.

En el aula de prácticas tienes disponible NetBeans en la imagen de arranque de **Ubuntu 16.04**. Sin embargo, **deberás realizar la instalación del plugin para Ruby en cada sesión**. En las aulas de prácticas tienes disponible el citado plugin en:

~/Escritorio/Departamentos/lsi/pdoo/pluginNetbeansRuby/

La **instalación de este plugin** se hace desde el menú *Tools/Plugins* de NetBeans. En la ventana asociada a esta opción se debe seccionar la pestaña *Downloaded* y hacer clic sobre el botón *Add Plugins*. Después se debe navegar hasta al carpeta que contiene el plugin descomprimido, activar la visualización de **todos los ficheros (*.*) y seleccionarlos todos**. Finalmente, haciendo clic sobre el botón *Install* y reiniciando NetBeans el plugin estará listo para ser usado. **Es imprescindible seleccionar todos los ficheros (*.*) al realizar la instalación**.

Los **exámenes de prácticas** se realizarán en los ordenadores del aula y usando la imagen de Ubuntu 16.04 con el código **examenubu16.** No se dispondrá de acceso a Internet durante estos exámenes, solo a la plataforma Prado. Se recomienda probar las prácticas en los ordenadores del aula y con dicha imagen en concreto, para evitar situaciones incómodas durante el desarrollo de los exámenes.

Desarrollo de esta práctica

En esta práctica se realizará la implementación de una serie de clases y de tipos de datos enumerados. Estas tareas servirán como aproximación a la programación y diseño orientado a objetos.

Todas las tareas de esta práctica se realizaran dentro de un paquete denominado **deepspace** para **Java** y de un módulo denominado **Deepspace** para **Ruby**.

En Java, para nombrar los consultores se utilizará notación lowerCamelCase y los consultores seguirán la siguiente nomenclatura get<NombreAtributo>. En Ruby los consultores se llamarán exactamente igual que los atributos a los que van asociados. De forma equivalente, en Java los modificadores se llamarán set<NombreAtributo> y en Ruby se llamarán exactamente igual que los atributos a los que van asociados.

Dado que Ruby no dispone de la visibilidad a nivel de paquete, se utilizará la visibilidad pública como sustituto.

En todos los archivos Ruby que se usen caracteres especiales, es decir, eñes, vocales con tilde, etc. debe aparecer, **como primera línea del fichero**, la siguiente:
#encoding:utf-8

De no hacerlo, esos caracteres no serán reconocidos y el intérprete Ruby se detendrá dando un error.

Enumerados

Crea los siguientes tipos enumerados. En cada caso se proporciona el nombre del tipo y sus posibles valores además de una breve descripción del mismo.

CombatResult: {ENEMYWINS, NOCOMBAT, STATIONESCAPES, STATIONWINS}
Este enumerado representa todos los resultados posibles de un combate entre una estación espacial y una nave enemiga.

GameCharacter: {ENEMYSTARSHIP, SPACESTATION}
Representanta a los dos tipos de personajes del juego

ShotResult: {DONOTRESIST, RESIST}
Representa el resultado de un disparo recibido por una nave enemiga o una estación espacial.

Investiga como crear estos tipos de datos enumerados en Java.

En Ruby no existen los tipos enumerados como tales y utilizaremos módulos que contengan y encapsulen los posibles valores de cada enumerado para simular una funcionalidad parecida a la que se obtiene en Java. Cada valor será una variable con el nombre del valor inicializada a un símbolo. Como ejemplo se proporciona la implementación de *GameCharacter*:

```
module GameCharacter
   SPACESTATION=:spacestation
   ENEMYSTARSHIP=:enemystarship
end
```

Ejemplos de acceso a uno de estos valores:

- GameCharacter::SPACESTATION
- Deepspace::GameCharacter::SPACESTATION si lo hacemos desde fuera el módulo Deepspace.

Investiga lo que son los símbolos en Ruby y entiende bien la técnica que se está usando.

WeaponType

Este enumerado representa a los tipos de armas del juego y no será simplemente una lista de valores, sino que tendrá cierta funcionalidad adicional. En este caso, cada valor posible del tipo enumerado tendrá asociado un valor numérico concreto igual a la potencia de disparo de cada tipo de arma.

En Java, gracias a que los tipos enumerados apenas se diferencian de las clases, esto puede hacerse fácilmente. El tipo enumerado tendrá:

- Un atributo de instancia privado de tipo *float* denominado *power*.
- Un constructor con visibilidad de paquete con un único parámetro que inicialice el atributo *power*.
- Un consultor con visibilidad de paquete getPower() que devuelve el valor del atributo correspondiente.
- Estos tres valores posibles y el valor para el atributo power asociado a cada uno de ellos {LASER(2.0), MISSILE(3.0), PLASMA(4.0) }

El ejemplo de los planetas que puedes encontrar aquí:
https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html
te servirá como referencia para crear este enumerado.

En Ruby, para en este caso el tipo enumerado, se seguirán los siguientes pasos:

- Crear un módulo WeaponType
- Dentro del módulo define una clase denominada *Type* con un atributo de instancia *power* y un consultor para ese atributo. Se deberá además crear un constructor para esta clase *Type* que inicialice el atributo *power*.
- También dentro del módulo WeaponType, pero fuera de la clase *Type*, crea tres instancias de *Type* denominadas LASER, MISSILE, PLASMA utilizando en cada caso el valor apropiado en el único parámetro del constructor para que el atributo *power* quede correctamente inicializado.

Clases

Loot

Esta clase representa el botín que se obtiene al vencer a una nave enemiga. Puede incluir cantidades que representen un número de paquetes de suministros, armas, potenciadores de escudo, hangares y/o medallas.

Crea una clase con visibilidad de paquete denominada *Loot*. Añade los siguiente atributos de instancia privados:

- *nSupplies*
- *nWeapons*
- *nShields*
- *nHangars*
- *nMedals*

todos de tipo entero.

Añade un constructor con visibilidad de paquete que reciba como parámetro un valor para cada uno de esos atributos (en el mismo orden que en la lista anterior).

Añade un consultor público por cada atributo siguiendo la siguiente nomenclatura *getNSupplies()*, *getNWeapons()* , etc. En Ruby los consultores se denominarán *nSupplies*, *nWeapons*, etc.

SuppliesPackage

Esta clase representa a un paquete de suministros para una estación espacial. Puede contener armamento, combustible y/o energía para los escudos.

Crea una clase con visibilidad de paquete denominada *SuppliesPackage*. Añade los siguientes atributos de instancia privados, todos de tipo float:

ammoPower
fuelUnits
shieldPower

Añade un constructor con visibilidad de paquete que reciba como parámetro un valor para cada uno de esos atributos (en el mismo orden que en la lista anterior). Añade también un constructor copia (con la misma visibilidad) que construya una instancia de *SuppliesPackage* a partir de otra instancia de la misma clase suministrada como parámetro: *SuppliesPackage(SuppliesPackage s).* En Ruby, el constructor copia se implementará como un método de clase llamado *newCopy* que devolverá una copia construida a partir de la instancia recibida como parámetro.

Añade un consultor público por cada atributo siguiendo la misma nomenclatura que en la clase anterior.

ShieldBooster

Esta clase representa a los potenciadores de escudo que pueden tener las estaciones espaciales.

Crea una clase con visibilidad de paquete denominada ShieldBooster. Añade los siguientes atributos de instancia privados:

name de tipo *String*
boost de tipo *float*
uses de tipo *int*

Añade dos constructores siguiendo exactamente la misma pauta que en la clase anterior.

Añade un consultor público para los atributos *boost* y *uses* siguiendo la misma nomenclatura que en la clase anterior.

Añade un método de instancia público llamado useIt (sin parámetros). Este método, si el valor del atributo *uses* es mayor que 0, lo decrementa en una unidad y devuelve el valor del atributo boost; devuelve el valor 1.0 en otro caso.

Weapon

Esta clase representa a las armas de las que puede disponer una estación espacial para potenciar su energía al disparar.

Crea una clase con visibilidad de paquete denominada Weapon. Añade los siguientes atributos de instancia privados:

name de tipo *String*
type de tipo *WeaponType*
uses de tipo *int*

Añade dos constructores siguiendo exactamente la misma pauta que en la clase anterior.

Añade un consultor público para los atributos *type* y *uses* siguiendo la misma nomenclatura que en la clase anterior.

Añade un método de instancia público denominado *power()* que devuelva la potencia de disparo indicada por el tipo de arma.

Añade un método de instancia público llamado useIt (sin parámetros). Este método, si el valor del atributo *uses* es mayor que 0, lo decremente en una unidad y devuelve el valor del método *power()*; devuelve el valor 1.0 en otro caso.

Dice

Esta clase tiene la responsabilidad de tomar todas las decisiones que dependen del azar en el juego. Es como una especie de dado, pero algo más sofisticado, ya que no proporciona simplemente un número del 1 al 6, sino decisiones concretas en base a una serie de probabilidades establecidas.

Crea una clase con visibilidad de paquete denominada *Dice*. Añade los siguientes atributos de instancia privados:

NHANGARSPROB
NSHIELDSPROB
NWEAPONSPROB
FIRSTSHOTPROB

todos de tipo float y constantes.

Añade otro atributo de instancia privado denominado *generator* del tipo *Random*.

Añade un constructor con visibilidad de paquete y sin parámetros que inicialice adecuadamente el atributo *generator* y el resto de atributos a los siguientes valores:

NHANGARSPROB=0.25
NSHIELDSPROB=0.25
NWEAPONSPROB=0.33
FIRSTSHOTPROB=0.5

A continuación se añadirán un conjunto de métodos que se encargan de todas las decisiones del juego en las que interviene el azar. En cada uno de ellos se utilizará el generador de números aleatorios para obtener valores (casi siempre del intervalo [0,1[) que sirvan para tomar tales decisiones.

int initWithNHangars: devuelve 0 con una probabilidad de NHANGARSPROB y 1 en caso contrario. Este método determina el número de hangares que recibirá una estación espacial al ser creada.

int initWithNWeapons: devuelve 1 con una probabilidad de NWEAPONSPROB, 2 con la misma probabilidad y 3 con una probabilidad de (1-2* NWEAPONSPROB). Este método determina el número de armas que recibirá una estación espacial al ser creada.

int initWithNShields: devuelve 0 con una probabilidad de NSHIELDSPROB y 1 en caso contrario. Este método determina el número de potenciadores de escudo que recibirá una estación espacial al ser creada.

int whoStarts(int nPlayers): genera un número aleatorio del intervalo [0,nPlayers-1]. Nota: tanto en Java como en Ruby existe un método de instancia de Random para generar números aleatorios enteros de un determinado intervalo. El método *whoStarts* determina el jugador (su índice) que

iniciará la partida.

GameCharacter firstShot: genera SPACESTATION con una probabilidad de FIRSTSHOTPROB y ENEMYSTARSHIP en otro caso. Este método determina quién (de los dos tipos de personajes del juego) dispara primero en un combate: la estación espacial o la nave enemiga.

boolean spaceStationMoves(float speed): devuelve *true* con una probabilidad de *speed* y *false* en otro caso (se asume que speed será un número entre 0 y 1). Este método determina si la estación espacial se moverá para esquivar un disparo. La probabilidad de moverse será mayor cuanto más cerca está la velocidad potencial actual de la estación espacial de su velocidad máxima potencial.

Programa principal

Crea una clase denominada *TestP1* que tenga asociado un método de clase denominado *main* para hacer las funciones de programa principal.
En Ruby, utilizaremos el mismo esquema aunque no exista el mismo concepto de programa principal asociado a un método con un nombre específico. Así, en el mismo fichero test_p1.rb donde se haya definido la clase *TestP1,* deberás añadir una línea de código al final que produzca la ejecución del método *main* de la misma al ejecutar el fichero test_p1.rb

En el método main crea el código para realizar las siguientes tareas:

- Crea varias instancias de cada clase creada en esta práctica y, utilizando los consultores, muestra en la consola toda la información posible de esos objetos creados.

- Crea una instancia de la clase *Dice*, llama a cada método 100 veces y calcula cuantas veces se obtiene cada uno de los valores posibles. Comprueba si se cumplen a nivel práctico las instrucciones relativas a las probabilidades de cada evento.

Llegado este punto te habrás dado cuenta que ante un error tipográfico habitual como escribir mal un atributo o el nombre de un método, Netbeans en Java avisa mientras se escribe pudiendo subsanar el error al instante. Sin embargo, desarrollando en Ruby no se tiene esa ayuda del IDE, cobrando especial importancia la prueba del software. De todos modos, en Java, el hecho de que un programa compile sin errores **no significa que esté libre de errores**.

Se aconseja realizar pequeños programas principales, en ambos lenguajes, que permitan probar todo el código desarrollado. Cada alumno debe ser su principal crítico y diseñar las pruebas para intentar encontrar errores en su código. Una prueba de código se considera que ha tenido éxito si produce la detección de un error.

En Java, usando el depurador, sigue paso a paso la creación de los objetos del primer punto y observa como se va modificando el valor de los atributos.