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 Irrgarten 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.

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. Para que cada vez que se integre 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 (y resto de sistemas operativos más usados) sí. Puedes tener problemas en los exámenes si no prestas atención a cómo nombras los archivos.

Herramientas

Para el desarrollo en Java 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 el desarrollo en Ruby puedes utilizar cualquier editor de textos y ejecutar tu proyecto directamente desde la consola. Dadas las características del proyecto, y no habiendo dependencias no triviales que mantener, el uso de un entorno de desarrollo no es necesario.

Utilización de un sistema de control de versiones: git

Se recomienda que utilices el sistema de control de versiones git para la gestión del código fuente de tu proyecto, aunque este aspecto no es obligatorio ni será evaluado. Los profesores de prácticas te ayudarán en esta tarea.

Para utilizar git no es necesario disponer de una cuenta en servicios tales como github, pero este tipo de plataformas te facilitarán trabajar en distintos ordenadores o en equipo con otros compañeros ya que te facilitan un lugar centralizado y accesible de forma remota para almacenar tu proyecto.

Identificación

Hay distintas formas de identificarse en github. Aquí dispones de una breve explicación sobre el uso de pares de llaves (llave pública y llave privada) para tal fin.

ssh-keygen -f github # Crea los ficheros github.pub (llave pública) y github (llave privada)

Se recomienda que tus llaves se almacenen en el directorio /*home/<tu_usuario>/.ssh*

Sube la llave **pública** a tu cuenta en github.com (la llave privada nunca debe abandonar tu ordenador). Para ello debes dirigirte a Settings → SSH and GPG Keys en tu cuenta de github y añadir esa llave pública como llave SSH. Esto permitirá acceder a tus repositorios identificándote con la llave privada.

Creación de un repositorio

Crea un repositorio **privado** vacío. Se recomienda que sea privado, ya que de otra forma cualquier persona tendría acceso a tu proyecto. Si vas a trabajar con un compañero de prácticas, añádelo como colaborador.

Ten en cuenta que vas a acceder a tu repositorio mediante SSH y que necesitarás referirte a este repositorio usando la URL para este protocolo.

Vídeo

Se proporciona un vídeo donde se explican brevemente las principales operaciones con un repositorio git además de su uso desde Netbeans. Estas operaciones son:

- *commit*: guardar los cambios realizados desde la operación de commit anterior y guardarlos en el repositorio local.
- *push*: subir a un repositorio remoto (en este caso github) todos los commits pendientes.
- *pull*: bajar de un repositorio remoto todos las novedades y *mezclarlas* con el contenido local.
- *clone*: descargar una copia local de un repositorio ya existente para comenzar a trabajar.

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 **irrgarten** para **Java** y de un módulo denominado **Irrgarten** para **Ruby**.

En Java, se utilizará notación lowerCamelCase para los métodos y los consultores y modificadores seguirán la siguiente nomenclatura get<NombreAtributo> y set<NombreAtributo> respectivamente. En Ruby, se utilizara la notación *snake_case* para los métodos y los consultores y modificadores se llamarán exactamente igual que los atributos a los que van asociados. Las clases se nombrarán en ambos lenguajes utilizando exactamente el nombre indicado en la documentación en ambos lenguajes.

Para evitar proporcionar la misma información dos veces, en la documentación y diagramas se hará referencia a los métodos usando la notación *lowerCamelCase*. Recuerda adaptar esta información y usar en Ruby la notación *snake_case*.

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.

Directions: {LEFT, RIGHT, UP, DOWN}
Este enumerado representa las direcciones en que se puede mover el jugador por el laberinto.

Orientation: {VERTICAL, HORIZONTAL}
Representa los dos tipos de desplazamientos existentes en el juego.

GameCharacter: {PLAYER, MONSTER}
Representanta a los dos tipos de personajes del juego

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
    PLAYER  = :player
    MONSTER = :monster
end
```

Ejemplos de acceso a uno de estos valores:

- GameCharacter:: PLAYER
- Irrgarten::GameCharacter:: PLAYER si lo hacemos desde fuera el módulo Irrgarten.

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

Clases

Weapon

Esta clase representa las armas que utiliza el jugador en los ataques durante los combates.

Crea una clase denominada *Weapon*. Añade los siguiente atributos de instancia privados:

- *power:* de tipo float
- *uses:* de tipo int

Añade un constructor y como parámetros un valor para cada uno de esos atributos (en el mismo orden que en la lista anterior).

Añade un método de instancia público sin parámetros llamado attack que devuelva un número en coma flotante representando la intensidad del ataque del jugador. Si el arma aún tiene usos disponibles (*uses > 0*), se decrementa ese valor y se devuelve el valor de *power*. En otro caso el método devuelve 0.

Añade el método

- *public String toString()* en Java
- *to_s* en Ruby

que devuelva un representación en forma del cadena de caracteres del estado interno del objeto. Así devolverá “W[2.0, 5]” para un arma con una potencia de disparo de 2 unidades y que aún puede ser usada 5 veces.

Shield

Esta clase representa los escudos que utiliza el jugador cuando se defiende de un ataque de un monstruo.

Crea una clase denominada *Shield*. Añade los siguiente atributos de instancia privados:

- *protection:* de tipo float
- *uses:* de tipo int

Añade un constructor y como parámetros un valor para cada uno de esos atributos (en el mismo orden que en la lista anterior).

Añade un método de instancia público sin parámetros llamado *protect* que devuelva un número en coma flotante representando la intensidad de la defensa del jugador. Si el escudo aún tiene usos disponibles (*uses > 0*), se decrementa ese valor y se devuelve el valor de *protection*. En otro caso el método devuelve 0.

Añade el método
- *public String toString()* en Java
- *to_s* en Ruby

que devuelva un representación en forma del cadena de caracteres del estado interno del objeto. Así devolverá “S[3.0, 4]” para un escudo que proporciona un nivel de protección de 3 unidades y que aún puede ser usado 4 veces.

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.

Dado que se considera que no tiene sentido que existan distintas instancias de esta clase, con estados distintos, toda la funcionalidad requerida la proporciona la propia clase.

Crea una clase denominada *Dice*. Añade los siguientes atributos **de clase** privados:

MAX_USES = 5 (número máximo de usos de armas y escudos)
MAX_INTELLIGENCE = 10.0 (valor máximo para la inteligencia de jugadores y monstruos)
MAX_STRENGTH = 10.0 (valor máximo para la fuerza de jugadores y monstruos)
RESURRECT_PROB = 0.3 (probabilidad de que un jugador sea resucitado en cada turno)
WEAPONS_REWARD = 2 (numero máximo de armas recibidas al ganar un combate)
SHIELDS_REWARD = 3 (numero máximo de escudos recibidos al ganar un combate)
HEALTH_REWARD = 5 (numero máximo de unidades de salud recibidas al ganar un combate)
MAX_ATTACK = 3 (máxima potencia de las armas)
MAX_SHIELD = 2 (máxima potencia de los escudos)

Añade otro atributo de clase privado llamado *generator* inicializado a una instancia de la clase *Random* (existe en ambos lenguajes). Todos los números aleatorios se generarán usando métodos de instancia de esta clase. Investiga el funcionamiento del método *rand* (y de en qué medida cambia su funcionamiento en función de los parámetros suministrados) de esa clase en Ruby y de los métodos

nextFloat y *nextInt* de esa clase en Java.

A continuación se proporciona, a modo de ejemplo, un método basado en el azar en Ruby que en media devuelve *true* el 30% de las veces y *false* el 70% de las ocasiones.

```
def aleatorio
  if (generador.rand < 0.3) # generador es una instancia de Random
    return true
  else
    return false
  end
end
```

Con los atributos definidos y la información adicional proporcionada, añade los siguientes métodos de clase públicos:

int randomPos(int max): devuelve un número de fila o columna aleatoria siendo el valor del parámetro el número de filas o columnas del tablero. La fila y la columna de menor valor tienen como índice el número cero.

int whoStarts(int nplayers): devuelve el índice del jugador que comenzará la partida. El parámetro representa el número de jugadores en la partida. Los jugadores se numeran comenzando con el número 0.

float randomIntelligence(): devuelve un valor aleatorio de inteligencia del intervalo [0, MAX_INTELLIGENCE[

float randomStrength(): devuelve una valor aleatorio de fuerza del intervalo [0, MAX_STRENGTH[

boolean resurrectPlayer(): indica si un jugador muerto debe ser resucitado o no.

int weaponsReward(): indica la cantidad de armas que recibirá el jugador por ganar el combate. Será un número aleatorio desde 0 (inclusive) que nunca debe superar el máximo indicado en la definición de los atributos de clase. Es decir, el número aleatorio debe estar en el intervalo cerrado [0,WEAPONS_REWARD].

int shieldsReward(): indica la cantidad de escudos que recibirá el jugador por ganar el combate. Será un número aleatorio desde 0 (inclusive) que nunca debe superar el máximo indicado en la definición de los atributos de clase.

int healthReward(): indica la cantidad de unidades de salud que recibirá el jugador por ganar el combate. Será un número aleatorio desde 0 (inclusive) que nunca debe superar el máximo indicado en la definición de los atributos de clase.

float weaponPower(): devuelve un valor aleatorio en el intervalo [0, MAX_ATTACK[

float shieldPower(): devuelve un valor aleatorio en el intervalo [0, MAX_SHIELD[

int usesLeft(): devuelve el número de usos que se asignará a un arma o escudo. Será un número aleatorio desde 0 (inclusive) que nunca debe superar el máximo indicado en la definición de los atributos de clase.

float intensity(float competence): devuelve la cantidad de competencia aplicada. Será un valor aleatorio del intervalo [0, competence[

boolean discardElement(int usesLeft): este método devuelve *true* con una probabilidad inversamente proporcional a lo cercano que esté el parámetro del número máximo de usos que puede tener un arma o escudo. Como casos extremos, si el número de usos es el máximo devolverá *false* y si es 0 devolverá *true*. Es decir, las armas o escudos con más usos posibles es menos probable que sean descartados.

Aunque algunos de los métodos indicados tienen una implementación idéntica, se ha decido tener métodos distintos para que su nombre refleje claramente su función y para permitir en un futuro que decisiones aleatorias distintas dejen de tener la misma implementación aunque inicialmente sí se implementen igual.

Añadido a las clases Weapon y Shield

Ahora que ya se dispone de dado, añade un método público a las clases *Weapon* y *Shield*:

boolean discard(): este método produce el resultado delegando en el método discardElement del dado pasando al mismo la cantidad de usos restantes como parámetro. Este método implementa la decisión de si un arma o escudo debe ser descartado.

GameState

Crea una clase llamada *GameState* con los siguientes atributos de instancia privados:

- *labyrinthv* de tipo String
- *players* de tipo String
- *monsters* de tipo String
- *currentPlayer* de tipo int (representa el índice del jugador actual)
- *winner* de tipo lógico
- *log* de tipo String

Esta clase permitirá, de forma muy sencilla, almacenar una representación del estado completo del juego: el estado del laberinto, el estado de los jugadores, el estado de los monstruos, el índice del jugador que tiene el turno, un indicador sobre si ya hay un ganador y un atributo adicional para guardar en una cadena de caracteres eventos interesantes que hayan ocurrido desde el turno anterior.

Crea un constructor para esta clase con un parámetro cara inicializar cada atributo.

Añade un consultor para cada atributo. Fíjate que para las clases anteriores no se ha indicado que añadas consultores/modificadores. Esto es intencionado ya que no son necesarios ni deseados en este contexto.

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 utiliza todos sus métodos.
- Prueba a utilizar los valores de los tipos enumerados creados.
- Prueba la clase *Dice:* llama a cada método 100 veces y 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.