Incorporando código suministrado por los profesores a tu proyecto

Sigue los siguientes pasos para incorporar el código que os proporcionamos para ser incorporado a vuestro proyecto:

- Averigua la ruta de tu proyecto: haz clic con el botón derecho sobre el proyecto en NetBeans y selecciona la opción "*Properties*". Debes hacer clic sobre el nodo del que cuelgan todos los elementos del proyecto en el panel izquierdo.

- En Java: si la ruta de tu proyecto es R, en R/src verás una carpeta por cada paquete de tu proyecto. Ahora mismo solo debes tener la carpeta deepspace correspondiente al paquete con el mismo nombre. Copia los ficheros fuente suministrados en R/src/deepspace/

- En Ruby: copia los ficheros fuente suministrados en R/lib/

No debes modificar ninguna de las clases suministradas por los profesores. Si hay un problema de concordancia repasa en primer lugar las clases que ya has creado para ver si siguen totalmente las especificaciones dadas. Si después de ese repaso persiste el problema, consulta con tu profesor.

Desarrollo de esta práctica

En esta práctica se realizará la implementación del diagrama de clases completo de Deepspace. Este diagrama se suministra adjunto a este guión. En concreto se crearán todas las clases (algunas serán suministradas por los profesores), para cada clase se definirán sus atributos, incluyendo aquellos que surgen de las relaciones entre clases, se definirán las cabeceras de todos los métodos, y se implementarán algunos de ellos. Los métodos que queden sin implementar en esta práctica, y que se indica en este guión cuáles son, se implementarán en la práctica siguiente.

Al inspeccionar el citado diagrama de clases te darás cuenta que algunas de clases ya las implementaste en la práctica anterior. Efectivamente es así y esa parte del trabajo ya la tienes completada salvo por algunos detalles que se detallen posteriormente.

Un objetivo importante de esta práctica es continuar con tu aprendizaje de diseño de software, por ello es fundamental que no te quedes solamente en los detalles de implementación o en las particularidades de los lenguajes; debes analizar el diagrama de clases y observar cómo este diseño representa el juego que se está desarrollando y cómo las distintas clases tienen unas responsabilidades muy concretas siguiendo los principios de alta cohesión y bajo acoplamiento.

Elementos que debes añadir a lo elaborado en la P1

Aunque en el diagrama de clases no aparece, añade un método toString en Java y to_s en Ruby a cada clase, este método debe devolver una representación en forma de cadena de caracteres de una instancia. Añade un método de este tipo también a las clases que implementaste en la práctica anterior. Utilizaremos estos métodos como ayuda a la depuración y prueba del software.

Por otro lado, debes añadir el siguiente método a algunas de las clases que ya tienes creadas

(comprueba a cuales utilizando el diagrama). Para una clase ***Foo***:

Foo*ToUI getUIVersion()*

Estos métodos simplemente construyen un nuevo objeto ***Foo****ToUI* a partir de la propia instancia que recibe el mensaje y lo devuelve (return new ***Foo****ToUI(this);*). Estos objetos constituyen una capa que permite conectar el modelo con la interfaz de usuario manteniendo cierto nivel de aislamiento entre ambos niveles.

Las clases *******ToUI* son suministradas junto con el resto material de esta práctica y solo necesitas incorporarlas a tu proyecto en el propio paquete *deepspace* o módulo *DeepSpace*.

Añadiendo nuevas clases

Además de las clases ya suministradas, deberás implementar las siguientes:

- Hangar
- Damage
- EnemyStarShip
- SpaceStation
- GameUniverse

Debes añadir todos los atributos y la cabecera de todos los métodos del diagrama (deja el cuerpo de los métodos nuevos vacío). Presta atención a la visibilidad de cada elemento. En Java, para evitar problemas de compilación, puedes utilizar la siguiente sentencia como única línea del cuerpo de los métodos de los que por ahora solo tienes la cabecera:

```
throw new UnsupportedOperationException();
```

Debes además añadir todos los atributos generados por las relaciones entre las distintas clases. Todos esos atributos producidos por asociaciones serán privados.

Añadiendo funcionalidad

En este apartado crearás el cuerpo de una serie de métodos sencillos para ir completando la funcionalidad del juego. Los métodos más complejos se abordarán en la práctica siguiente.

Además de los métodos indicados a continuación debes añadir los constructores de las nuevas clases indicados en el diagrama.

Hangar

boolean spaceAvailable(). Devuelve true si aún hay espacio para añadir elementos y que por lo tanto no se ha llegado a la capacidad máxima.

boolean addWeapon(Weapon w). Añade el arma al hangar si queda espacio en el Hangar, devolviendo true en ese caso. Devuelve false en cualquier otro caso.

boolean addShieldBooster(ShieldBooster w). Añade el potenciador de escudo al hangar si queda espacio. Devuelve true si ha sido posible añadir el potenciador, false en otro caso.

Weapon removeWeapon(int w). Elimina el arma número w del hangar y la devuelve, siempre que esta exista. Si el índice suministrado es incorrecto devuelve null.

ShieldBooster removeShieldBooster(int s). Elimina el potenciador de escudo número s del hangar y lo devuelve, siempre que este exista. Si el índice suministrado es incorrecto devuelve null.

El resto de consultores se autoexplican por su nombre

Damage

Los objetos de esta clase representan el daño producido a una estación espacial por una nave enemiga cuando se pierde un combate. Cada instancia indicará la perdida de una cantidad de potenciadores de escudo y por otro lado, o bien una cantidad de tipos indeterminados de armas o un conjunto de tipos de armas concretas que se deben eliminar.
Todas las instancias deben ser unas independientes de otras y por lo tanto no deben compartir instancias de objetos mutables (modificables)

void discardWeapon(Weapon w). Si la instancia dispone de una lista de tipos concretos de armas, intenta eliminar el tipo del arma pasada como parámetro de esa lista. En otro caso simplemente decrementa en una unidad el contador de armas que deben ser eliminadas. Ese contador no puede ser inferior a cero en ningún caso.

void discardShieldBooster(). Decrementa en una unidad el número de potenciadores de escudo que deben ser eliminados. Ese contador no puede ser inferior a cero en ningún caso.

boolean hasNoEffect(). Devuelve true si el daño representado no tiene ningún efecto. Esto quiere decir que no implica la pérdida de ningún tipo de accesorio (armas o potenciadores de escudo).

int arrayContainsType(ArrayList<Weapon> w,WeaponType t). Devuelve el índice de la posición de la primera arma de la colección de armas (primer parámetro) cuyo tipo coincida con el tipo indicado por el segundo parámetro. Devuelve -1 si no hay ninguna arma en la colección del tipo indicado por el segundo parámetro.

adjust(ArrayList<Weapon> w, ArrayList<ShieldBooster> s). Devuelve una versión ajustada del objeto a las colecciones de armas y potenciadores de escudos suministradas como parámetro. Partiendo del daño representado por el objeto que recibe este mensaje, se devuelve una copia del mismo pero **reducida** si es necesario para que no implique perder armas o potenciadores de escudos que no están en las colecciones de los parámetros.

El resto de consultores se autoexplican por su nombre.

EnemyStarShip

float protection(). Devuelve el nivel de energía del escudo de la nave enemiga (shieldPower).

float fire(). Devuelve el nivel de energía de disparo de la nave enemiga (ammoPower).

ShotResult receiveShot(float shot). Devuelve el resultado que se produce al recibir un disparo de una determinada potencia (pasada como parámetro). Si el nivel de la protección de los escudos es menor que la intensidad del disparo, la nave enemiga no resiste (DONOTRESIST). En caso contrario resiste el disparo (RESIST). Se devuelve el resultado producido por el disparo recibido.

El resto de consultores se autoexplican por su nombre. No te preocupes si existen consultores para los atributos ammoPower y shieldPower además de los métodos fire y protection. Se ha diseñado así intencionadamente.

SpaceStation

Además de la información relativa a los métodos de esta clase se proporciona el significado de alguno de los atributos:

- SHIELDLOSSPERUNITSHOT : Unidades de escudo que se pierden por cada unidad de potencia de disparo recibido.
- MAXFUEL: Máxima cantidad de unidades de combustible que puede tener una estación espacial.

void assignFuelValue(float f). Fija la cantidad de combustible al valor pasado como parámetro sin que nunca se exceda del límite.

void cleanPendingDamage(). Si el daño pendiente (pendingDamage) no tiene efecto fija la referencia al mismo a null.

boolean receiveWeapon(Weapon w). Si se dispone de hangar, devuelve el resultado de intentar añadir el arma al mismo. Si no se dispone de hangar devuelve false.

boolean receiveShieldBooster(ShieldBooster s). Si se dispone de hangar, devuelve el resultado de intentar añadir el potenciador de escudo al mismo. Si no se dispone de hangar devuelve false .

void receiveHangar(Hangar h). Si no se dispone de hangar, el parámetro pasa a ser el hangar de la estación espacial. Si ya se dispone de hangar esta operación no tiene efecto.

void discardHangar(). Fija la referencia del hangar a null para indicar que no se dispone del mismo.

void receiveSupplies(SuppliesPackage s). La potencia de disparo, la del escudo y las unidades de combustible se incrementan con el contenido del paquete de suministro.

void setPendingDamage(Damage d). Se calcula el parámetro **ajustado** (adjust) a la lista de armas y potenciadores de escudo de la estación y se almacena el resultado en el atributo correspondiente.

void mountWeapon(int i). Se intenta montar el arma con el índice i dentro del hangar. Si se dispone de hangar, se le indica que elimine el arma de esa posición y si esta operación tiene éxito (el hangar proporciona el arma), se añade el arma a la colección de armas en uso.

void mountShieldBooster(int i). Se intenta montar el potenciador de escudo con el índice i dentro del hangar. Si se dispone de hangar, se le indica que elimine el potenciador de escudo de esa posición y si esta operación tiene éxito (el hangar proporciona el potenciador), se añade el mismo a la colección de potenciadores en uso.

void discardWeaponInHangar(int i). Si se dispone de hangar, se solicita al mismo descartar el arma con índice i.

void discardShieldBoosterInHangar(int i). Si se dispone de hangar, se solicita al mismo descartar el potenciador de escudo con índice i.

float getSpeed(). Devuelve la velocidad de la estación espacial. Esta se calcula como la fracción entre las unidades de combustible de las que dispone en la actualidad la estación espacial respecto al máximo unidades de combustible que es posible almacenar. La velocidad se representa por tanto como un número del intervalo [0,1].

void move(). Decremento de unidades de combustible disponibles a causa de un desplazamiento. Al número de las unidades almacenadas se les resta una fracción de las mismas que es igual a la velocidad de la estación. Las unidades de combustible no pueden ser inferiores a 0.

boolean validState(). Devuelve true si la estación espacial está en un estado válido. Eso implica que o bien no se tiene ningún daño pendiente o que este no tiene efecto.

void cleanUpMountedItems(). Eliminar todas las armas y los potenciadores de escudo montados a las que no les queden usos.

Los siguientes métodos se abordarán en la siguiente práctica:

float fire()
float protection()
ShotResult receiveShot(float shot)
void setLoot(Loot loot)
void discardWeapon(int i)
void discardShieldBooster(int i)

El resto de consultores se autoexplican por su nombre.

GameUniverse

En el constructor de esta clase hay que realizar las siguientes tareas:

- Se crea el controlador de estados del juego
- Se inicializa el contador de turnos a 0
- Se crea el dado

boolean haveAWinner(). Devuelve true si la estación espacial que tiene el turno ha llegado al número de medallas necesarias para ganar.

Los métodos *mount** y *discard** delegan en el método homónimo de la estación espacial que tenga el turno siempre que se cumpla que el estado del juego sea INIT o AFTERCOMBAT. En caso contrario no tienen ningún efecto.

Los siguientes métodos se abordarán en la siguiente práctica:

void init(ArrayList<String> names)
boolean nextTurn()
CombatResult combat()
CombatResult combat(SpaceStation station,EnemyStarShip enemy)

El resto de consultores se autoexplican por su nombre.

Versión alternativa del mazo de cartas

En los archivos suministrados verás que hay dos versiones del mazo de Cartas (*CardDeck*) y del gestor de los mazos o crupier (*CardDealer*). En clase se explicarán dos alternativas para resolver el problema asociado a que el crupier debe devolver una copia de la carta que se le solicita.

Para que funcione la segunda versión (basada en herencia e interfaces) es necesario que añadas algo a todas las cabeceras de las clases de los objetos que se almacenan en un mazo: *Weapon*, *SuppliesPackage*, etc. En el siguiente ejemplo se muestra como quedaría la cabecera de *Weapon*:

class Weapon ***implements Copyable <Weapon>*** *{*

Todo lo relativo a estas cuestiones será explicado con detalle en clase.

Comprobando lo aprendido en las dos primeras prácticas

Una vez finalizada la práctica deberías saber y entender los siguientes conceptos:

- Saber crear clases a partir de un diagrama de clases. Esto incluye a los atributos y métodos especificados.
- Entender bien la diferencia entre atributos y métodos de instancia y de clases. Diferenciar, igualmente, entre variables locales, parámetros de métodos y atributos.
- Entender y saber implementar las relaciones entre clases en un diagrama de clases.
- Entender la diferencia entre estado e identidad.
- Entender la diferencia entre objeto y referencia a un objeto.
- Entender los distintos tipos de visibilidad de atributos y métodos. Conocer las diferencias entre Java y Ruby al respecto.
- Tener conocimientos sobre el uso mensajes a distintos objetos para obtener una cierta funcionalidad
- Entender lo que representan los paquetes UML y su implementación utilizando paquetes Java y módulos Ruby.
- Comprender la implementación de tipos de datos enumerados utilizando módulos Ruby. Comprender cómo utilizar elementos que están dentro de un módulo desde otro módulo.
- Saber crear un nuevo proyecto, tanto en Java como en Ruby, en la ruta concreta donde querías crearlo.
- Saber crear clases dentro de módulos Ruby y saber cómo acceder a dichas clases desde otros módulos.
- Saber crear un programa principal que se ejecute automáticamente tanto en Java como en Ruby.

Errores más frecuentes en los exámenes

- Con relación al contenido

    - No leer completamente el examen (a veces dedicáis tiempo a hacer tareas que no se piden)
    - No probar el código que se ha desarrollado (entregar código con errores de compilación o ejecución está penalizado)
- Con relación a la entrega
    - No saber exportar un proyecto desde Netbeans.
    - No saber en qué ruta ha guardado Netbeans el proyecto exportado.
    - Trabajar en un proyecto y entregar otro distinto.
    - No guardar los cambios y hacer la entrega sin las últimas modificaciones (o sin ninguna).
    - No calcular bien el tiempo necesario para hacer la entrega.

La entrega del trabajo realizado en tiempo y forma también forma parte de los exámenes. Se recomienda que también se estudie esta tarea y se ensaye el proceso completo de entrega para evitar situaciones incómodas durante los exámenes. Para ello, se encuentra disponible una actividad en Prado para que podáis ensayar la entrega y medir el tiempo que empleáis en ello.

Se recomienda, igualmente, que durante los exámenes realicéis varias entregas (examen-1.zip, examen-2.zip, etc.) según vayáis completando parte del examen. Así, si os quedáis sin tiempo para la entrega final, ya tenéis gran parte del examen subido a la plataforma.