Curso 2021 / 2022

Práctica 4:

Ampliación del sistema incluyendo herencia y polimorfismo

Modificando el diseño de nuestro juego

En esta práctica se modificará el diseño del juego para añadir una serie de elementos, tales como la herencia entre clases, que te permitirá practicar sobre una base de conceptos de programación orientada a objetos mayor. Observarás además cómo el diseño del juego mejora en algunos aspectos con las modificaciones que se introducen en esta práctica.

Se proporciona un diagrama de clases que indica los cambios que hay que realizar sobre el que ya disponías. Las clases resaltadas son las que contienen estos cambios y es importante indicar que estas solo muestran los cambios (principalmente elementos añadidos), no los elementos que permanecen inalterados.

Gestión de cartas

Los profesores te proporcionan un diseño alternativo en Java al sistema de mazos de cartas utilizando herencia y clases paramétricas. Aparecen una serie de clases nuevas (*Safe**) que consiguen que los mazos de cartas siempre devuelvan una copia de la carta que deba ser devuelta, aunque utilizando técnicas distintas a las usadas en las versiones que se proporcionaron en prácticas anteriores.

Debes añadir a tu proyecto, sustituyendo los ficheros de los que ya dispones por sus nuevas versiones, todos los ficheros proporcionados. Deberás además entender el funcionamiento de estos nuevos mazos de cartas estudiando su código.

Daños sufridos en combate

Como ya has podido comprobar, el actual diseño de la clase *Damage* no es el más adecuado. Requiere numerosas comprobaciones relacionadas con el uso o no de la lista de tipos específicos de armas a perder. Ahora que ya conoces la herencia, vamos a rediseñar esta parte del juego.

En el diagrama de clases adjunto verás que la clase *Damage* pasa a ser abstracta y que aparecen dos subtipos concretos de esta. Ahora se indicará específicamente el tipo de daño que llevará asociado una nave enemiga usando los dos subtipos concretos de *Damage*.

Piensa como separar la actual clase *Damage* en las dos clases indicadas, qué debe moverse a cada una de ellas y qué debe quedarse en la clase abstracta por ser común a sus descendientes.

Los cambios relacionados con este apartado harán que el antiguo constructor copia de *Damage* deje de ser útil, ya que ahora nunca existirán instancias de esa clase. Sustituiremos ese constructor copia por un método de instancia *copy* que devolverá una copia del objeto que reciba el mensaje. Cuando sea necesario crear una copia de un objeto que represente daños, simplemente se llamará a este método *copy* de la instancia que se desea copiar.

Nuevas interfaces

Fíjate en que aparecen dos interfaces nuevas en el diagrama. En Java, deberás implementarlas al igual que los cambios en las clases correspondientes relacionadas con ellas.

Ciudades espaciales

Las estaciones espaciales ahora pueden transformarse en ciudades espaciales. La transformación viene indicada por el botín. Fíjate en lo que devuelve el método *setLoot* y en los nuevos elementos en la clase Loot. En función de que el botín incluya o no algún tipo de transformación se producirá un tipo de valor de salida de *setLoot* u otro. Además, en los posibles resultados del combate se ha añadido un valor para indicar que se ha producido tal transformación.

Las ciudades espaciales están formadas por la estación que tenía el jugador y las estaciones espaciales del resto de jugadores. Son a su vez un tipo de estación espacial.

Cuando disparan, suman su potencia de disparo a las del resto de jugadores. Así, para que una ciudad espacial dispare, deben disparar todos sus componentes. Igual ocurre con el nivel de protección del escudo.

En lo relativo al combustible y los desplazamientos, utilizan solo los recursos propios provenientes de la estación espacial base que tenía el jugador propietario de la estación que generó la ciudad.

Las ciudades espaciales ya no pueden sufrir ninguna transformación, aunque siempre conservan una referencia de la estación espacial base en torno a la que se generó la ciudad. Para evitar las transformaciones se redefine el método *setLoot()*.

Estaciones espaciales eficientes

Las estaciones espaciales, pueden también transformarse en estaciones espaciales eficientes (*PowerEfficientSpaceStation*). En este caso también viene indicado por el botín la posible transformación. El esquema de la transformación es equivalente al utilizado para las ciudades espaciales.

Las estaciones espaciales eficientes consiguen que se incremente la potencia de disparo y de protección en un determinado factor constante respecto a las estaciones estándar. Estas no pueden transformase en ciudades espaciales . Para evitar las transformaciones se redefine el método *setLoot()*.

Existe una subclase de la estación espacial eficiente que utiliza una tecnología en pruebas (beta) y que es potencialmente más eficiente al disparar (*BetaPowerEfficientSpaceStation*). Solo al disparar, se determina aleatoriamente (usando un dado) si para ese disparo la estación va a funcionar simplemente como una estación eficiente normal o como una estación eficiente mejorada. Esto se determina cada vez que se produce un disparo con el método *extraEfficiency()* del dado.

Tal y como está diseñado el juego, una *PowerEfficientSpaceStation* podría transformase en una *BetaPowerEfficientSpaceStation y viceversa.*

Piensa que ocurriría si una estación forma parte de una ciudad espacial (como colaboradora) y se

convierte en algún tipo de estación eficiente. No será necesario que tengas en cuenta esta posibilidad en la implementación que estás realizando del juego, pero reflexionar sobre esta cuestión es un ejercicio que los profesores te recomendamos que realices.

GameUniverse

En esta clase aparecen una serie de métodos nuevos:

makeStationEfficient(). Pregunta al dado si debe convertir la estación espacial actual en una estación eficiente o en una estación eficiente beta. El dado es el que determina la conversión en un tipo o en otro con el método *extraEfficiency().* Actualiza la referencia a currentStation.

*createSpaceCity(). S*i el juego no dispone ya de una ciudad espacial (haveSpaceCity==false) convierte la estación espacial actual en una ciudad espacial usando como estación espacial base la actual y como colaboradoras el resto las estaciones espaciales. Actualiza la referencia a currentStation y también el atributo haveSpaceCity.

Finalmente, es necesario realizar un añadido al método:
combat(SpaceStation station,EnemyStarShip enemy).

Cuando la estación espacial vence al enemigo, al recibir el botín hay que añadir el código necesario para tratar la conversión a estación eficiente o a ciudad espacial si ese botín así lo indica. En este código se utilizará los dos nuevos métodos explicados anteriormente.

Loot

Habrás observado que en el diagrama de clases aparecen ahora parámetros adicionales en el constructor de esta clase. En esta ocasión, en Ruby, no se añadirá un nuevo constructor para permitir construir objetos suministrando esos nuevos parámetros, sino que se añadirán dos nuevos parámetros al constructor ya creado. El valor por defecto para será *false* tanto para el parámetro que indica si se producirá una conversión a una estación eficiente como el que se refiere a la conversión en ciudad espacial. Se pretende que no deje de funcionar código de construcción de objetos de esta clase ya existente. En Java sí se añadirá un constructor que acepte también los nuevos parámetros.

Cambios en la interfaz de texto del juego

Modifica el código de la interfaz de texto para contemplar el caso de que una estación además de ganar, se haya convertido.
Pista: El añadido de código hay que hacerlo en el lugar donde se hace un procesamiento u otro en función del resultado del combate.

Comprobando lo aprendido hasta ahora

Una vez finalizada esta práctica y las anteriores deberías saber y entender los siguientes conceptos:

- Los indicados en el guion anterior.
- Herencia.
- Visibilidad.
- Implementar una clase que “deriva” de otra.

- Saber qué métodos y atributos se pueden usar en la clase “derivada” de entre los que tiene la “superclase” y cómo usarlos.
- Polimorfismo y Ligadura dinámica.
- Saber redefinir un método heredado y cuándo es necesaria esta técnica.
- Tipo estático vs. Tipo dinámico.
- Dada una variable que referencia a un objeto concreto, saber qué mensajes se le puede enviar, y qué métodos se ejecutarían en cada caso.
- Saber qué significa hacer un casting que involucra clases derivadas y superclases.