METODOLOGÍA DE LA PROGRAMACIÓN

Guion de prácticas de laboratorio.

Profesores: Jorge Puente, Alfredo Alguero, Eugenia Díaz, Juan Luis Mateo y Ramiro Varela

TEMA 1. Programación Orientada a Objetos. HERENCIA

Material para dejar en el Campus Virtual: 1 fichero zip con 2 proyectos que contienen las siguientes clases con sus correspondientes ejemplos de prueba

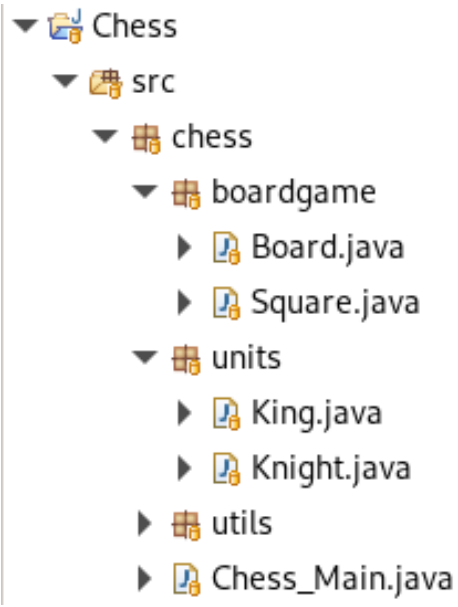
- **`Chess`:** Componentes de un juego de tablero: clases **`Board`**, **Square** y **King** que se utilizarán para representar los elementos básicos de un juego de tablero, en este caso un prototipo básico del juego del ajedrez. El proyecto se divide en diferentes packages, y se incluyen pruebas unitarias de las clases.
- **`Jeroquest_1`**: Jerarquía de clases **`Character`**, **`Barbarian`** y **`Mummy`**

PRÁCTICA 1.5 Usando clases predefinidas

Objetivo: Saber cómo trabajar con un proyecto con una estructura más compleja, usando la funcionalidad de clases ya definidas y añadir nueva funcionalidad.

Ejercicios 1.5.1. Estudia el proyecto **Chess** (Ajedrez). Utilizando para ello los comentarios Javadoc del código, y los diagramas de clases UML.

a) Abre el proyecto **Chess**. En este proyecto hay una serie de clases para representar el juego del ajedrez de una forma simplificada, y están organizadas en paquetes como se puede ver en la figura siguiente.



b) Analiza las clases del paquete **chess.boardgame**, principalmente las clases **Board** y **Square**. Como es de esperar, la clase **Board** modela el tablero como una matriz con filas y columnas. Cada elemento del tablero se modela mediante la clase Square, una casilla. Las piezas del juego se colocan en las casillas, por el momento solo la pieza del rey, representada por la clase **King**. Utiliza el diagrama de clases que está en el fichero **Chess1.gaphor** para tener una perspectiva global de las clases.Nota: Necesitarás la aplicación Gaphor para poder visualizar y editar sus diagramas. Puedes descargarla de su página Web: https://gaphor.org/download.

c) Analiza la clase **King** del paquete **chess.units**. Una pieza tiene como estado su posición actual y uno de los métodos más importantes que define su comportamiento es **validPositions(Board)** que debe devolver todas las posiciones del tablero que la pieza puede alcanzar desde la posición actual.

d) Estudia la clase **Chess_Main** donde en el método **main()** se muestra un ejemplo de cómo usar las clases anteriores. En este ejemplo, las piezas se mueven de forma aleatoria por el tablero hasta completar una serie de rondas. Ejecuta esta clase y observa el resultado.

Ejercicio 1.5.2 Añade nueva funcionalidad al proyecto.

a) A pesar de que la clase **Knight** ya existe en el proyecto, no está acabada ni está integrada con el resto de las clases. En primer lugar, completa esta clase para que proporcione el comportamiento esperado. La letra que debe representar esta pieza es '**N**', y los movimientos de caballo son en forma de L, como se muestra a continuación. Usa como guía los comentarios TODO para saber dónde son necesarios los cambios.

b) La clase **Square** está definida para poder albergar piezas del tipo rey, pero no caballos. Para poder hacerlo también es necesario hacer algunos cambios. De nuevo, déjate guiar por los comentarios para saber dónde realizar cambios, y ten en cuenta como se gestionan las piezas del rey para hacer lo propio con las del caballo.

c) Por último, también la clase **Board** requiere ser actualizada para la nueva pieza. En este caso es para poder mover o quitar del tablero al caballo.

d) Usa las clases de test, en la carpeta test para comprobar que los cambios realizados están de acuerdo a lo esperado y que no alteran la funcionalidad previa.

PRÁCTICA 1.6 Superclases, herencia y clases abstractas

Ejercicio 1.6.1. En el proyecto anterior, te proponemos crear una nueva clase, denominada **ChessPiece** (ficha de ajedrez), que agrupe todos los atributos y métodos comunes de estos dos tipos de fichas, y que actúe como superclase de **King** y de **Knight**.

No existe un consenso en cuanto a las posiciones válidas para una ficha genérica, así que el método **generatePossiblePositions()** no tendrá una implementación en **ChessPiece** aunque sí la declaración de su prototipo. Revisa la visibilidad de este método para que sus subclases puedan acceder a él y dar su propia implementación y que no sea accesible para el resto de las clases del programa.

Resuelve de forma similar la definición del método **toChar()** al no existir un símbolo por defecto para una ficha genérica.

Asegúrate de utilizar la anotación @Override en todos aquellos métodos que sobrescriban un método heredado.

PRACTICA 1.7. Ejercicios con la jerarquía básica del Jeroquest: `Character, Barbarian y Mummy`

Objetivo: Entender y saber utilizar una jerarquía de clases no simple como la de personajes del Jeroquest.

Ejercicio 7.1. Añadir operaciones al programa de prueba de la jerarquía de clases **`Character, Barbarian y Mummy`**

a) Accede al proyecto Jeroquest_1 y abre el modelo UML de este proyecto contenido en el fichero **`Jeroquest1.gaphor`** y examina la jerarquía formada por las tres clases.
b) Define un bárbaro y una momia y haz que uno ataque al otro un número máximo de veces o hasta que se muera (el otro).
c) Define un vector de bárbaros y otro de momias, los dos del mismo tamaño, y haz que alternativamente un bárbaro ataque a una momia y viceversa, siguiendo el protocolo que se indica a continuación: se realiza un número entero N de **rondas**, en cada **ronda** todos los bárbaros y todas las momias atacan en orden desde el primero al último; cuando ataca un bárbaro *i* elige a una momia *j* aleatoriamente; si el bárbaro *i* o la momia *j* no están vivos, entonces no hay ataque, en otro caso el bárbaro *i* ataca a la momia *j*. Análogamente cuando ataca la momia *i*. Es decir, primero ataca el bárbaro 0 a una momia aleatoria, luego ataca la momia 0 a un bárbaro aleatorio, luego ataca el bárbaro 1, y así sucesivamente. El juego termina cuando se consumen los N **rondas**, o bien cuando todos los personajes de una clase se mueran.
d) Lo mismo que en el ejercicio del apartado d), pero haciendo que el personaje atacado sea siempre el que más puntos de vida tiene.
e) Muestra los personajes de cada clase al final del juego. Aparecerán ordenados según el valor del campo **body**, de mayor a menor.

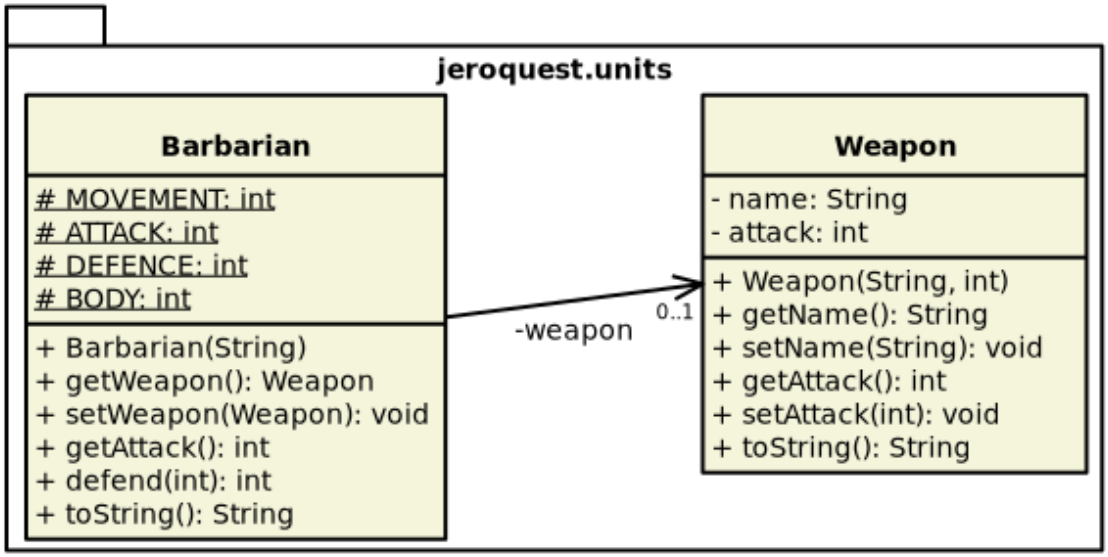
PRACTICA 1.8. Ejercicios con la jerarquía básica del Jeroquest: modificación de algunas clases `Character, Barbarian and Mummy`

Objetivo: Saber modificar las clases de una jerarquía para cambiar la funcionalidad de un programa.

Ejercicio 1.8.1. Equipando a los bárbaros por *composición*.

Todos los héroes del juego pueden tener un arma (weapon) para atacar. En este caso un héroe lanzará tantos dados como indique el arma (en lugar de los que indica su atributo **`attack`**). Si en algún momento pierde el arma volverá a atacar con los dados indicados por su atributo **`attack`**. El arma inicial de los bárbaros es una broadsword (espada ancha) como la de la figura de la derecha.

La siguiente figura muestra los cambios a introducir en la jerarquía de clases.



Lo que hay que hacer en este ejercicio es lo siguiente:

a) Para representar armas crea una nueva clase **`Weapon`**. Esta clase contendrá la descripción del arma y el número de dados a utilizar cuando se ataque con ella.

b) Modifica la clase **`Barbarian`** para que contenga un nuevo atributo **`weapon`** para representar su arma actual (si es que tiene alguna).

c) Modifica el constructor para que cree una espada ancha para el bárbaro.

d) Modifica el método **`getAttack`** para que tenga en cuenta el arma utilizada. Observa que en este caso el método observador no devuelve simplemente el valor del campo **`attack`**, sino que tiene que calcular los dados de ataque teniendo en cuenta si el barbaro tiene o no tiene arma.

e) En ambas clases crea los observadores y modificadores necesarios y sobrescribe el método **`toString`** para mostrar la nueva información.

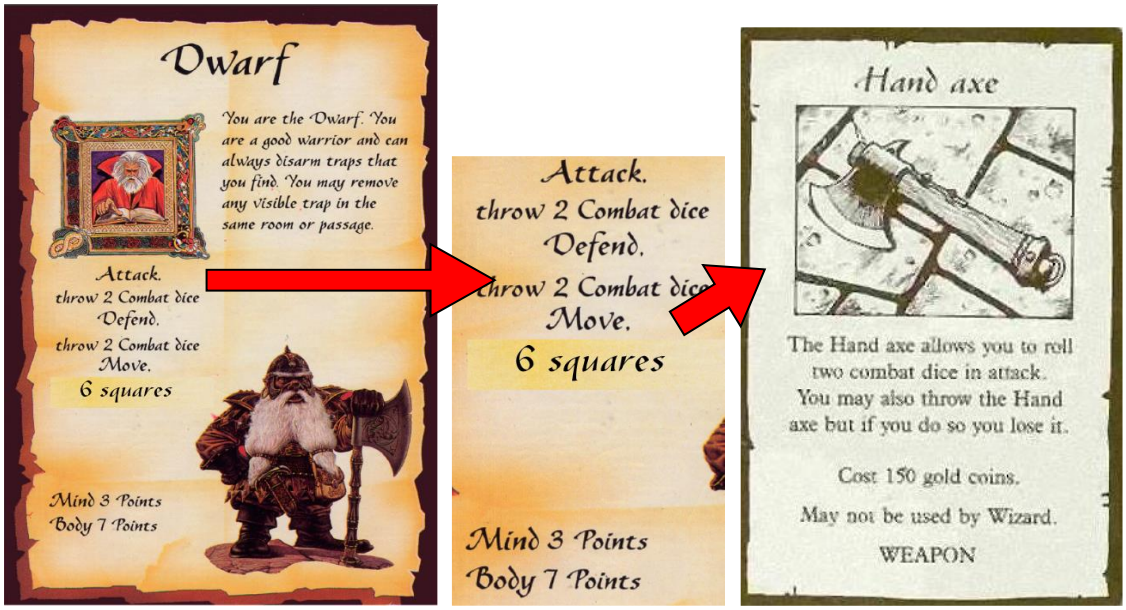
f) ¿Realmente los bárbaros utilizan su arma para atacar en los combates? Verifícalo y corrígelo si fuese necesario.

PRACTICA 1.9. Ejercicios con la jerarquía básica del Jeroquest : ampliación de la jerarquía de clases

Objetivo: Saber cómo añadir clases de la forma más adecuada a una jerarquía de clases.

Ejercicio 1.9.1. Ampliar la jerarquía de personajes del juego.

a) Define un nuevo tipo de personaje enano (clase **`Dwarf`**) e incorporarlo a la jerarquía de clases. A la hora de defenderse un enano consigue bloquear un impacto con un 5 o 6 en una tirada de defensa. Usa un dado para atacar, en la carta pone 2 porque se supone que inicialmente tiene el arma. Amplía los ejemplos de prueba anteriores añadiendo un nuevo vector de enanos.



b) Define un nuevo tipo de personaje momia rabiosa (clase **`FuriousMummy`**) e incorporarlo a la jerarquía de clases. Las momias rabiosas atacan **como las normales,** pero hacen el doble de daño; sin embargo, cuando las atacan no se defienden. Amplia los ejemplos de prueba anteriores para incluir personajes de esta clase.

c) Teniendo en cuenta futuros cambios en estos arquetipos de personajes, valora las ventajas de que la clase **`FuriousMummy`** herede de la clase **`Mummy`** o herede directamente de **`Character`**.

Ejercicio 1.9.2. Define la clase **`Hero`** como subclase de **`Character`** y con dos subclases **`Barbarian`** y **`Dwarf`**. Además, considera que todos los héroes son manejados por algún jugador, define para ello el atributo **`player`** (como una cadena de texto) en este tipo de personajes. En cada subclase de **Hero** crea un nuevo constructor que reciba en sus argumentos el nombre del personaje y el nombre del jugador. Actualiza el constructor original para que llame al nuevo constructor pasando **"no name"** como nombre inicial del jugador.

Comprueba que los ejemplos de prueba anteriores siguen funcionando igual. Comprueba que todos los atributos y métodos comunes de los héroes están en la nueva clase **`Hero`** y no en las de los héroes **`Barbarian`** o **`Dwarf`**.

Modifica el método **`attack()`** en la clase **`Character`** para que no sea necesario reescribirlo en sus subclases. (Ayuda: *¿qué método utilizas para preguntar a cualquier personaje con cuantos dados ataca?*)

Si consideras que no tiene sentido en este juego que se puedan crear directamente objetos de la clase **`Hero`** modifica su diseño para que no sea posible hacerlo.

Actualiza utilizando la aplicación gaphor el diagrama de clases UML del proyecto para que refleje todos los cambios que has hecho en el proyecto Jeroquest.

DISTRIBUCIÓN TEMPORAL

Sesión 5. PRÁCTICA 1.5.

Sesión 6. PRÁCTICA 1.6.

Sesión 7. PRÁCTICA 1.7.

Sesión 8: PRÁCTICA 1.8.

Sesión 9. PRÁCTICA 1.9.