8 | Ajedrez

Meta

Que el alumno se familiarice con el uso de herencia en diseño orientado a objetos.

Objetivos

Al finalizar la práctica el alumno será capaz de:

1. Definir clases y métodos abstractas.
2. Programar clases derivadas de clases abstractas.
3. Dar ejemplos de jerarquías de herencia.

Antecedentes

Diagramas de clase UML

Para expresar mejor el trabajo que se realizará en esta práctica se haremos uso de un notación especial: *los diagramas de clase de UML*.

UML son las siglas en inglés de *Unified Markup Language*.

Clases

Las clases se representan con cajas donde se pueden escribir:

- El nombre de la clase.
- Los atributos de la clase.
- Los métodos de la clase.

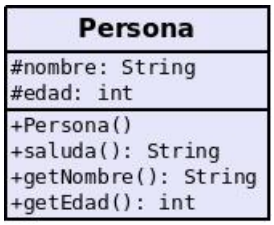

Figura 8.1 Ejemplo de diagrama de clase UML

Para representar los diferentes tipos de acceso se utilizan los símbolos siguientes:

- privado. Solamente los métodos de la clase pueden acceder a estos atributos.

\# protegido. La clase y clases que la extiendan, directa o indirectamente, pueden acceder a estos atributos.

\+ público. Todos pueden leer y escribir directamente estos atributos. Se recomienda sólo utilizarlos cuando los atributos son `final`, es decir, su valor nunca cambia.

Las variables estáticas (o de clase) se destacan por ir subrayadas. Una imagen dice más que mil palabras, por lo que la Figura 8.1 muestra un ejemplo.

Como verás, la notación para indicar los tipos de las variables es distinta a lo que se utiliza para programar en `Java`. En general, estos diagramas fueron diseñados para ser utilizados con cualquier lenguaje de programación, por lo que su notación es algo independiente de dichos lenguajes y hay ligeras variaciones dependiendo del software que se use para dibujarlos. Aquí seguiremos el estándar implementado por el software para dibujo de diagramas *Dia*. Es software libre y lo puedes descargar gratuitamente. Descubrirás que puede dibujar mucho más que diagramas de clase.

Generalización

La relación de herencia entre dos clases se conoce como una relación de *generalización*, pues se dice que la clase ancestra *generaliza* a sus clases descendientes. En otras palabras, la clase ancestra contiene datos que todas sus descendientes tienen (aún si no los pueden acceder directamente) y puede realizar operaciones que todas sus descendientes pueden realizar.

En UML la generalización se representa con una flecha con un triángulo vacío como punta, desde la clase que extiende hacia la clase más general [Figura 8.2].

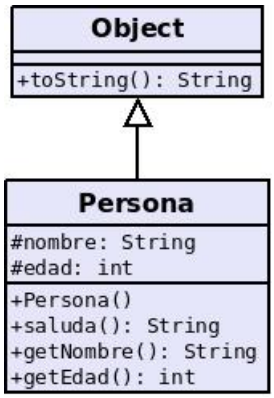

Figura 8.2 Ejemplo de generalización UML

Figura 8.3 Ejemplo de una interfaz en UML

Interfaces

En su versión más simple, las interfaces únicamente contienen declaraciones de métodos y no atributos. No todos los lenguajes de programación tienen interfaces como `Java`, por lo que es necesario construir un símbolo para ellas a partir de las opciones que ofrece UML. Una opción común se ilustra en la Figura 8.3.

Realización

Curiosamente, UML sí tiene un símbolo para indicar que una clase cumple con un contrato. Cuando una clase implementa una interfaz utilizaremos una flecha semejante a la de herencia, pero con la línea punteada. La Figura 8.4 muestra cómo dos clases implementan una interfaz.

Contención

Las relaciones de *contención* indican que un objeto contiene a otro (como un atributo). Se distinguen dos tipos especiales:

Agregación El objeto contenido puede existir, aunque el contenedor deje de existir. Se representa con un diamante vacío del lado de la clase contendora. Por ejemplo: un `Chofer` existe aún afuera de su coche, el `Coche` sigue existiendo aunque

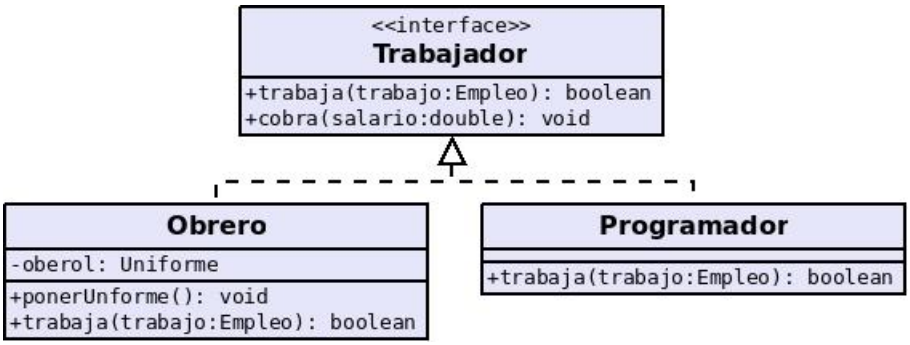

Figura 8.4 Ejemplo de realización UML

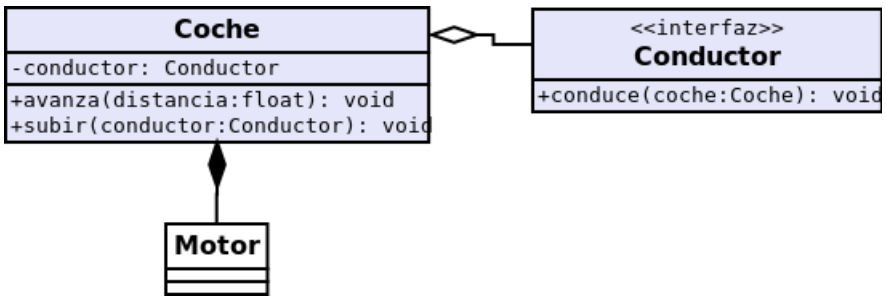

Figura 8.5 Ejemplos de contención UML. Agregación entre `Coche` y `Conductor`. Composición entre `Coche` y `Motor`.

se haya salido su chofer. Pero cuando esté manejando el coche, el chofer estará dentro de él Figura 8.5.

Composición El objeto contenedor está hecho de los objetos que contiene. Se representa con un diamante lleno del lado de la clase contendora. El ejemplo más tradicional son el cuerpo y las partes del cuerpo (brazos, piernas, torso, cabeza). También el coche está hecho de motor, carrocería, puertas, etc. y si falta alguno de estos componentes ya no tenemos un coche completo.

Dependencia

La relación de *dependencia* indica que una clase depende del trabajo realizado por otra (por ejemplo, de la respuesta que calcule cuando mande llamar a un método suyo) o de la estructura que define. En su forma más general, esta relación se representa con una flecha cuya punta está dibujada con dos líneas, desde la clase que ocupa el servicio hacia la que lo proveé. De hecho, la generalización y la realización son tipos particulares de relaciones de dependencia, que tienen asignada su propia variante en la notación.

Asociación

Cuando dos clases están relacionadas de cualquier forma, se dice que están asociadas. La forma más genérica de representar una asociación es con una línea uniendo ambas clases.

Paquetes

Por último, si queremos ilustrar las relaciones entre las clases de un sistema complejo, querremos indicar la agrupación de sus componentes en paquetes con funcionalidades distintas. El símbolo para los paquetes es un folder con el nombre del paquete en su pestaña.

El estándar de UML contiene muchos más símbolos y formas de utilizar la notación, que resultan útiles durante el proceso de ingeniería del software; sin embargo estos serán suficientes para que trabajemos ahora. Observa también que la cantidad de detalles que se incluyen en el diagrama puede facilitar o dificultar su lectura. Cuando se utilizan para documentar un proyecto es importante buscar un buen balance entre la expresividad[1] del diagrama y la claridad de lo que ilustra. Algunos paquetes de software también permiten generar cógido a partir de estos diagramas y, para que el código sea correcto, es necesario seguir las restricciones impuestas por ese paquete.

Diseño orientado a objetos

Todo sistema programado con el paradigma orientado a objetos busca promover las siguientes cualidades Viso y Peláez 2007:

Modularidad Se deben trazar fronteras claras entre partes del sistema para que la tarea se pueda repartir entre varios programadores.

Bajo nivel de acoplamiento Cada módulo debe usar lo menos posible de otros módulos, de forma que puedan ser reutilizados en otras aplicaciones.

Alta cohesión Todos los elementos ue se encuentren altamente relacionados entre sí, deben encontrarse dentro del mismo módulo. Esto hará que sean más fáciles de localizar, entender y modificar.

[1] Es decir, cuánta información proveé.

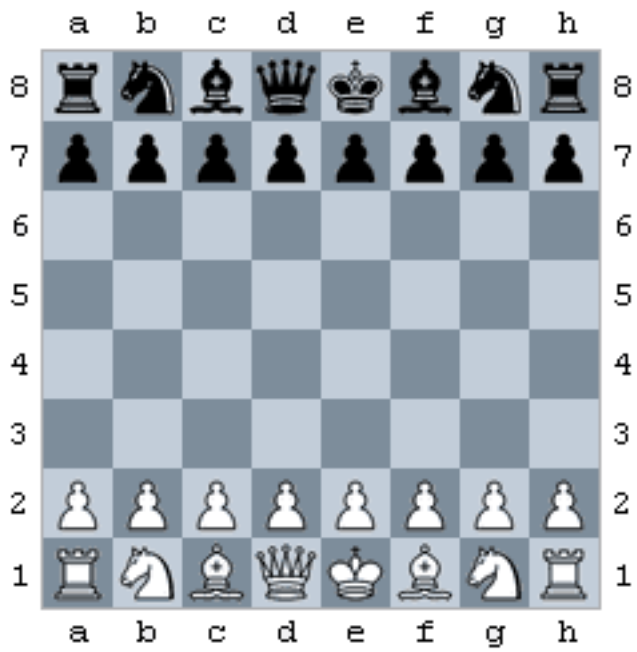

Figura 8.6 Etiquetado de las casillas en un tablero de ajedrez.

Desarrollo

En esta práctica programarás los algoritmos para calcular las casillas a las cuales se pueden mover las piezas de un juego de ajedrez, sin considerar los casos cuando capturarán una pieza enemiga[2]. Para ello, la Figura 8.6 muestra el lenguaje estándar para indicar la posición de una pieza en el tablero de ajedrez. Para los fines de esta práctica, no necesitarás modelar el tablero, sólo las piezas. Por otro lado, la Figura 8.8 muestra los movimientos válidos para cada pieza.

Ejercicios

1. Implementa una pequeña clase `Posicion` que contenga las dos cantidades: renglón y columna. Agregar los *getters* y *setters* que consideres necesarios. Asegúrate de que, cada vez que se asignen valores a `renglon` o `columna`, sólamente se acepten valores válidos. Si los valores de entrada no lo son, lanza una `IllegalArgumentException`. Es decir que, para renglón, los valores deben ser enteros entre 1 y 8, mientras que las columnas son los caracteres $\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}, \mathrm{f}, \mathrm{g}, \mathrm{h}$.

[2]Esta práctica está inspirada en 11.1.Ajedrez (López Gaona 2012, pág. 100)

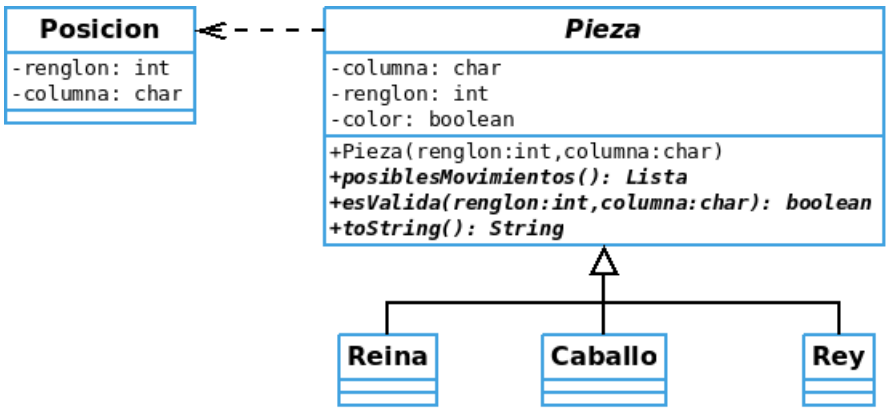

Figura 8.7 Diagrama de clases para las piezas de ajedrez.

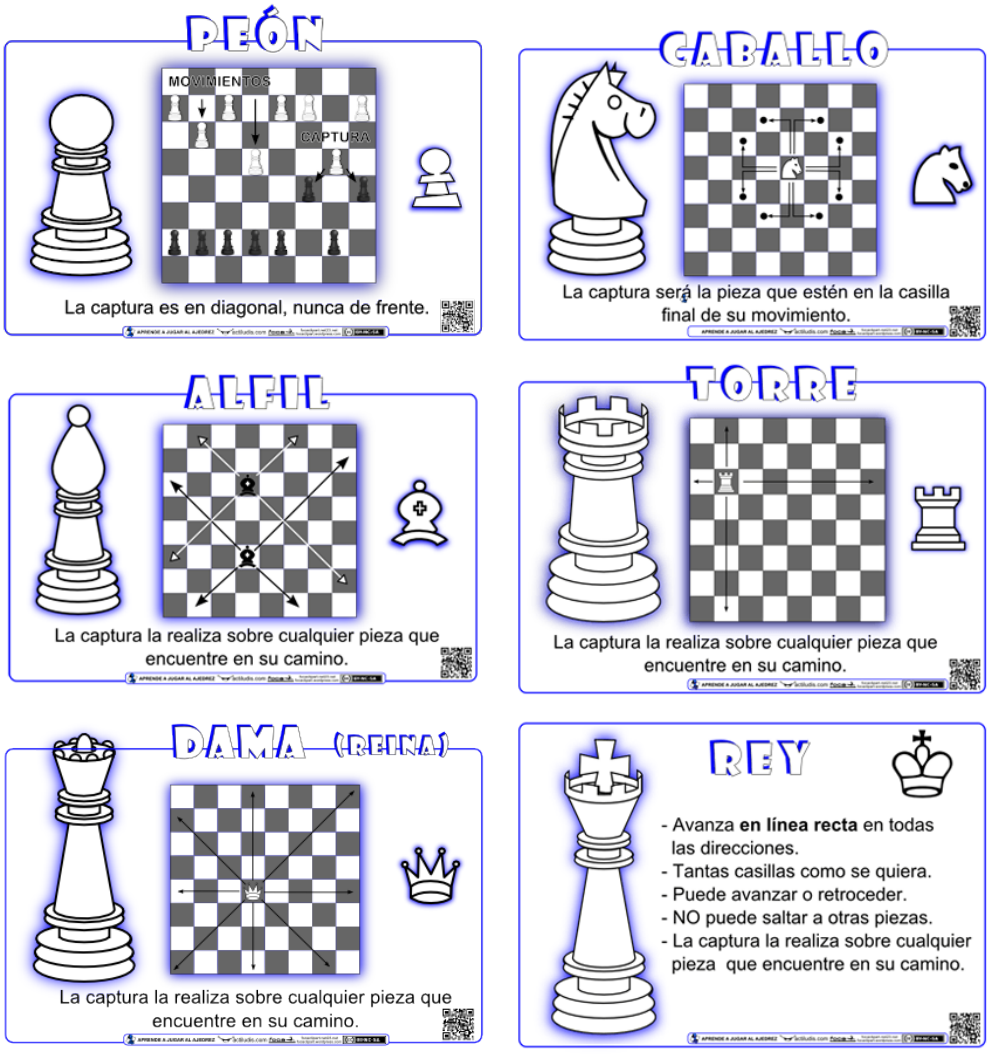

Figura 8.8 Movimientos de las piezas de ajedrez.

2. Implementa la clase `Pieza`, debe ser abstracta pues incluirá tres métodos abstractos:

   - `posiblesMovimientos()` Dada la posición actual de la pieza, devuelve una `Lista` con las posiciones de todas las casillas a las cuales se podría mover esa pieza. Aquí es donde la clase `Posicion` resultará particularmente útil.
   - `esValida(int renglon, char columna)` Indica si sería válido mover a la pieza desde su posición actual hasta la posición indicada en los parámetros. Debe tomar en cuenta, es particular, que la pieza no se salga del tablero.
   - `toString()` Devuelve una representación con cadena de la pieza y su estado actual.

   Agrega los *getters* y *setters* que consideres necesarios.

3. Programa la clase `Reina`. Según las reglas del juego puede desplazarse a cualquiera de las casillas en línea recta, horizontal o diagonal a partir de su posición, tanto hacia adelante como hacia atrás. En el mejor de los casos hay hasta 32 posiciones posibles.

4. Programa la clase `Caballo`. El caballo se puede desplazar en forma de L: dos casillas verticalmente y una horizontal o a la inversa. Puede tener hasta 8 posiciones a las cuales moverse.

5. Programa la clase `Rey`. El rey puede moverse una sola casilla en las ocho direcciones, por lo que hay máximo ocho posiciones.

   Para completar el programa, habría que agregar `Torre`, `Alfil` y `Peon`, pero sólo te serán útiles si de verdad quieres programar el juego y para eso aún falta ver cómo programar el tablero y determinar las capturas. De momento no haremos eso.

6. Agrega una clase de uso, `UsoAjedrez`, donde crees una reina, un caballo y una torre. Pide que genere las posiciones posibles a partir de celdas en el centro y orilla del tablero; así como que indique si la posición es válida a partir de la posición actual tanto para propuestas válidas, propuestas que no estén sobre una casilla válida y posiciones fuera del tablero (ej $[-2, \mathrm{i}]$). Observa que, para cambiar la casilla actual de las piezas, deberás haber programado los *setters* correspondientes.

7. Se darán dos puntos extra si programas una interfaz de usuario que permitan realizar lo anterior mediante un menú para el usuario. Se otorgarán fracciones de estos puntos si la interfaz permite realizar sólo algunas de estas tareas.