Técnicas algorítmicas en ingeniería del software

Grado en Ingeniería del Software (UCM)

PRÁCTICA DE ÁRBOLES DE JUEGO — Curso 2013/2014

FECHA LÍMITE DE ENTREGA: 31 DE ENERO

Juegos Reunidos

En esta práctica vamos a implementar unos "Juegos Reunidos". La aplicación nos dejará jugar contra el ordenador a varios juegos que podremos ir incorporando fácilmente. Todos los juegos tienen en común que son juegos de tablero de dos jugadores en los que solo hay dos clases de fichas (como Go, Otelo, Conecta 4, Tres en raya). Cada juego tendrá una interfaz gráfica que mostrará el estado del tablero y en la que el jugador podrá indicar su jugada con un click del ratón. Además, podremos elegir el tipo de jugador automático, que podrá ser aleatorio o inteligente de nivel bajo, medio o alto.

Tenemos por tanto tres clases abstractas: `JuegoLogT2` (parte lógica de los juegos), `InterfazGrafT2` (interacción gráfica con el usuario) y `JugadorAutT2` (jugadores automáticos).

Clase abstracta `JuegoLogT2`

La clase abstracta `JuegoLogT2` sirve como base para las clases de los juegos de tablero de dos jugadores de suma nula cuyas jugadas consisten en elegir una casilla del tablero. Cada casilla del tablero puede estar vacía o bien contener una ficha, que puede ser del humano o de la máquina. Para representarlas, conjuntamente con los jugadores y el tablero, utilizaremos los siguientes tipos

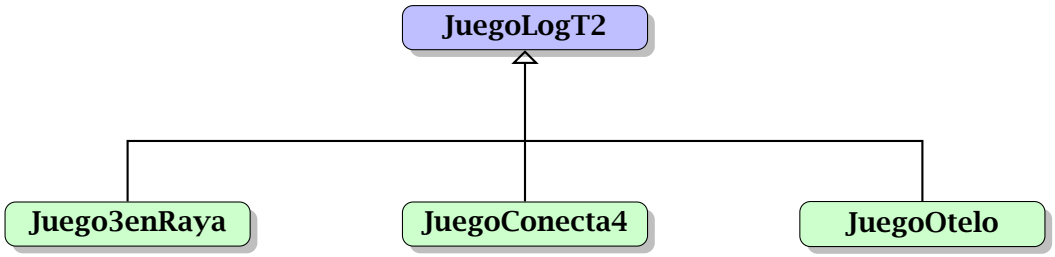
```
typedef enum {Jn, Jhum, Jmaq} Turno;
inline Turno cambia(Turno t){return (t==Jhum?Jmaq:Jhum);}

typedef Turno Ficha;

class Casilla {
public:
    unsigned int col;
    unsigned int fil;
    Casilla(unsigned int c=0, unsigned int f=0):col(c),fil(f){};
};
```

La clase abstracta `JuegoLogT2` es la siguiente

```
class JuegoLogT2 {
protected:
    Turno turno;  // indica cuál es el turno actual (a quién le toca jugar)
    bool ganador; // indica si ya hay un ganador
    Matriz<Ficha>* tablero; // puntero a una matriz cuyos elementos pueden ser
                            // Jn (casilla vacía), Jhum (casilla ocupada por el humano)
                            // o Jmaq (casilla ocupada por el ordenador).

    // copia en el atributo el tablero del juego recibido como argumento
    virtual void copiaTablero(const JuegoLogT2& EJ);
public:
    JuegoLogT2();
    JuegoLogT2(const JuegoLogT2& EJ);
    virtual ~JuegoLogT2();

    unsigned int numCol() const throw();
    unsigned int numFil() const throw();
    Turno dameTurno() const throw();
    bool enRango(unsigned int c,unsigned int f) const throw();
    bool enRango(Casilla cf) const throw();
    Ficha dameCasilla(unsigned int c,unsigned int f) const throw(EJuego);
```

```
    Ficha dameCasilla(Casilla cf) const throw(EJuego);

    // deja el tablero en su configuración inicial y el turno pasa a ser de JI
    virtual void reinicia(Turno JI);

    // método abstracto que determina si se puede jugar en la posición (c,f)
    virtual bool sePuede(unsigned int c,unsigned int f) const throw()=0;

    // análogo al anterior para la jugada cf
    bool sePuede(Casilla cf) const throw();

    // método abstracto que aplica la jugada caracterizada por la posición (c,f) del tablero.
    // Si dicha jugada no es legal se lanza una excepción
    virtual void aplicaJugada(unsigned int c,unsigned int f) throw(EJuego)=0;

    // análogo al anterior para la jugada cf
    void aplicaJugada(Casilla cf) throw(EJuego);

    // indica si no se pueden realizar más jugadas
    virtual bool fin() const=0;

    // indica si se ha alcanzado el final del juego, bien porque no se pueda seguir jugando
    // o porque ya haya ganador
    bool final() const throw();

    // devuelve si hay ganador
    bool hayGanador() const throw();

    // devuelve el ganador, que por defecto es el turno actual cuando hay un ganador
    virtual Turno dameGanador() const throw();

    // método abstracto que devuelve un puntero a una copia del juego actual
    virtual JuegoLogT2* clona() const=0;
};
```

Las clases no abstractas que hereden de `JuegoLogT2` han de implementar sus métodos abstractos (virtuales puros) y posiblemente redefinir algunos de sus métodos virtuales.



Clase `JuegoOtelo`

Esta clase hereda de `JuegoLogT2` e implementa el juego del Otelo. Tiene constructora por defecto, con parámetro (el turno que empieza a jugar) y por copia. Además, puede interesar un atributo `int numFichas[3]` para saber cuántas fichas hay de cada tipo.

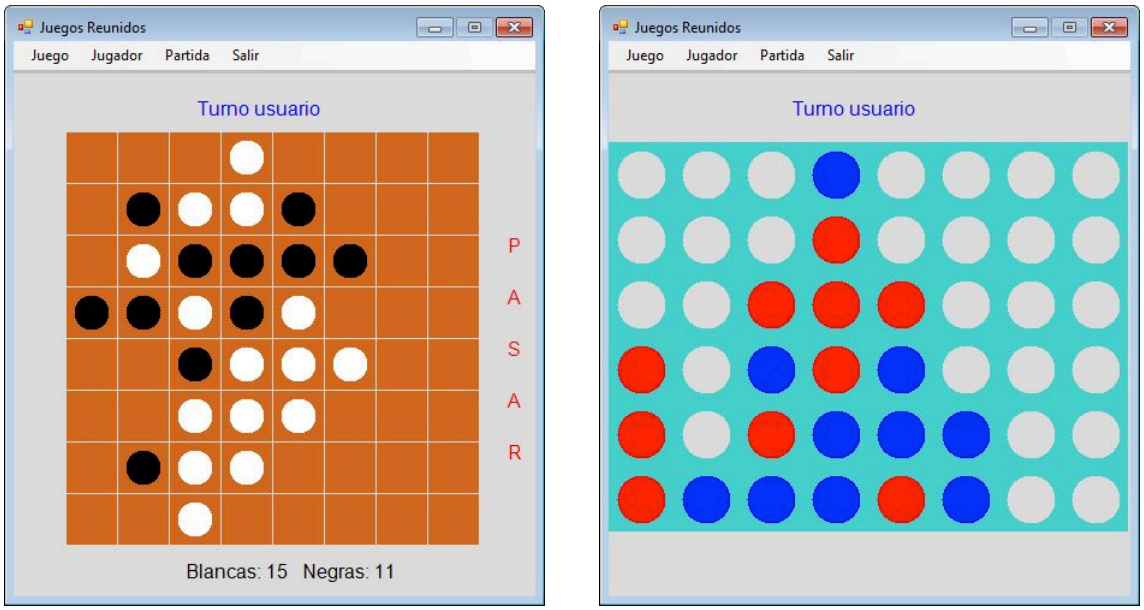
En este juego hay que pasar cuando no se puede colocar una ficha en ninguna casilla. Dado que en los juegos que heredan de `JuegoLogT2` jugar significa elegir una casilla, y en el caso del Otelo existe la posibilidad de que jugar consista en pasar, vamos a añadir una columna extra, de manera que elegir una casilla en esa columna se corresponda con la jugada pasar.

El juego acaba cuando no quedan casillas libres o bien el jugador que tiene el turno no puede jugar, habiendo pasado el turno anterior el otro jugador. Para controlar si el otro jugador ha pasado en su turno, se puede utilizar un atributo `bool haPasado` que indica si el otro jugador pasó en la anterior jugada.

Clase `JuegoConecta4`

Esta clase hereda de `JuegoLogT2` e implementa el juego Conecta 4. Tiene constructora por defecto, con parámetro (el turno que empieza a jugar) y por copia. Además, puede interesar añadir un atributo `int libres` que indique cuántas casillas quedan libres.

Hay que tener en cuenta que en el caso del Conecta 4, una jugada queda determinada por la columna.



Clase abstracta `JugadorAutoT2`

La clase abstracta `JugadorAutoT2` de los jugadores automáticos de juegos `JuegoLogT2` sirve como base para los jugadores de juegos no abstractos. Tiene constructora por defecto, destructora y un método abstracto para seleccionar la siguiente jugada.

```
class JugadorAutoT2 {
public:
    JugadorAutoT2(){};
    virtual ~JugadorAutoT2(){};

    // método abstracto que elige la posición del tablero donde jugar en el estado actual del juego EJ
    virtual Casilla juega(const JuegoLogT2& EJ) const throw(EJugador)=0;
};
```

Cualquier subclase no abstracta de `JugadorAutoT2` ha de implementar su método `juega`.

Clase `JugadorAleaOtelo`

Hereda de `JugadorAutT2`, implementando el método abstracto `juega` de manera que si el juego que recibe es del tipo correcto juega eligiendo al azar una casilla válida, y si no lanza una excepción. Si no se puede jugar, también lanza una excepción.

```
class JugadorAleaOtelo : public JugadorAutoT2 {
public:
    JugadorAleaOtelo(){};
    virtual ~JugadorAleaOtelo(){};
    virtual Casilla juega(const JuegoLogT2& EC) const throw(EJugador);
};
```

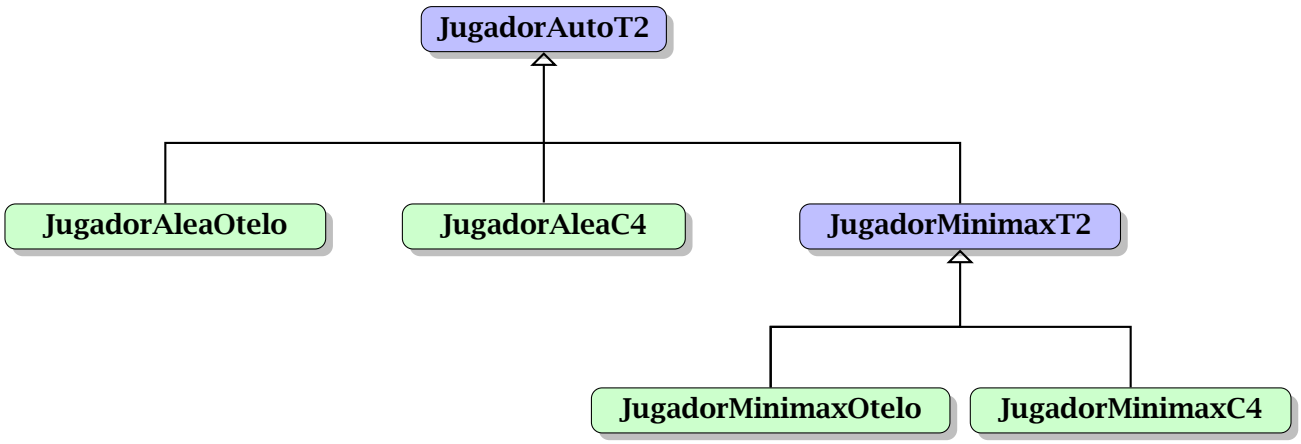
Clase `JugadorAleaC4`

Análoga a la anterior para el juego Conecta 4.

Clase abstracta `JugadorMinimaxT2`

La clase abstracta `JugadorMinimaxT2` hereda de `JugadorAutoT2` e implementa el método `juega` seleccionando la mejor jugada a partir de la valoración de un árbol de juego utilizando el algoritmo de poda α-β. Tiene un atributo que fija la profundidad del árbol de juego y métodos privados para valorar los nodos internos, según sean nodos MAX o nodos MIN. Para la valoración de las hojas se utiliza el método abstracto `Heuristica` que las clases concretas que hereden de esta deben implementar.

```
class JugadorMinimaxT2 : public JugadorAutoT2 {
protected:
    int nivel;
    virtual float Heuristica(const JuegoLogT2& EJ, Turno T) const=0;
    float valoraMin(const JuegoLogT2& EJ, Turno T, int n, float A, float B) const throw(EJugador);
    float valoraMax(const JuegoLogT2& EJ, Turno T, int n, float A, float B) const throw(EJugador);

public:
    JugadorMinimaxT2(int n=4) : nivel(n) {};
    virtual ~JugadorMinimaxT2(){};

    // implementa el método juega de JugadorAutoT2 explorando un árbol de juegos con raíz EJ
    virtual Casilla juega(const JuegoLogT2& EJ) const throw(EJugador);
};
```



Clase `JugadorMinimaxOtelo`

La clase `JugadorMinimaxOtelo` hereda de `JugadorMinimaxT2` redefiniendo el método `juega` para averiguar si el juego recibido es de la clase `JuegoOtelo`. Si es así juega invocando al método de la superclase. En caso contrario, se lanza una excepción. También se implementa el método abstracto `Heuristica`. La heurística por defecto cuenta el número de fichas del jugador (asignándole un peso según la posición en que se encuentra, dado por la matriz de `pesos`) y el número de fichas del rival y devuelve su diferencia.

```
const int pesos[8][8] = {
        {50, -1,5,2,2,5, -1,50},
        {-1,-10,1,1,1,1,-10,-1},
        { 5,  1,1,1,1,1,  1, 5},
        { 2,  1,1,0,0,1,  1, 2},
        { 2,  1,1,0,0,1,  1, 2},
        { 5,  1,1,1,1,1,  1, 5},
        {-1,-10,1,1,1,1,-10,-1},
        {50, -1,5,2,2,5, -1,50}
};
```

Clase `JugadorMinimaxC4`

Esta clase es análoga a la anterior pero para el juego Conecta 4. La heurística por defecto suma el valor de todas las posibles líneas de 4 posiciones del jugador y le resta la suma de los valores de

todas las posibles líneas de 4 posiciones del contrincante:

$$\sum_{linea4} \text{valor}_{jug}(linea4) - \sum_{linea4} \text{valor}_{contr}(linea4)$$

donde $\text{valor}_A(linea) = \begin{cases} 0 & \text{si hay fichas de } B \\ 10^k & \text{si no hay fichas de } B \text{ y hay } k \text{ fichas de } A \end{cases}$

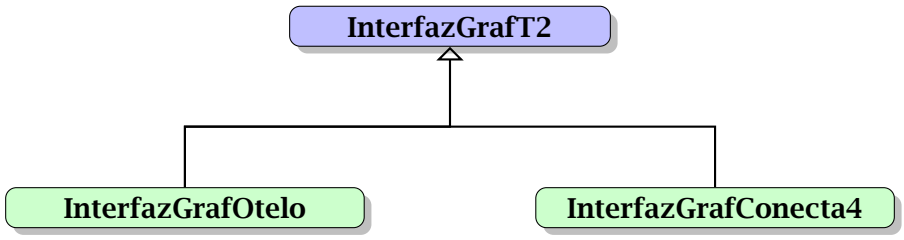
Clase abstracta `InterfazGrafT2`

La clase abstracta `InterfazGrafT2` sirve como base para las clases concretas que muestren el estado de un juego concreto, el tablero posiblemente junto con más información, como a quién le toca jugar o la puntuación actual. Además la interfaz gráfica es la encargada de transformar un click del ratón sobre el dibujo en la posición del tablero donde el usuario quiere jugar.

```
class InterfazGrafT2 {
protected:
    int window_width;   // anchura del lienzo donde se pinta el juego
    int window_height;  // altura del lienzo
public:
    InterfazGrafT2(int w, int h);
    virtual ~InterfazGrafT2();

    // devuelve la columna del tablero correspondiente a la coordenada X
    virtual unsigned int dameColumna(unsigned int X) const throw(EInterfaz)=0;

    // devuelve la fila del tablero correspondiente a la coordenada Y
    virtual unsigned int dameFila(unsigned int Y) const throw(EInterfaz)=0;

    // devuelve la casilla del tablero correspondiente a las coordenadas X e Y
    Casilla dameCasilla(unsigned int X, unsigned int Y) const throw(EInterfaz);

    // dibuja sobre el lienzo canvas el estado del juego EJ
    virtual void muestraEstado(Graphics^ canvas, const JuegoLogT2& EJ) const throw(EInterfaz)=0;
};
```

Clases `InterfazGrafOtelo` e `InterfazGrafC4`

Son clases que heredan de `InterfazGrafT2` implementando todos los métodos abstractos para los juegos concretos de Otelo y Conecta 4. En el método `muestraEstado` se comprueba si el juego recibido es de la clase correcta y si no es así se lanza una excepción.



Programa principal

El programa principal consta de una ventana con un menú para seleccionar el juego y el tipo de jugador al que el usuario quiere enfrentarse (aleatorio o inteligente, y en este caso de qué nivel). También tiene una opción para iniciar una partida indicando quién comienza, y otra opción para salir.

En la ventana se muestra el estado actual del tablero del juego, sobre el que el usuario, en su turno, puede seleccionar con el ratón la casilla con la que quiere jugar. Cada vez que se realice una jugada, habrá que mostrar el nuevo estado del juego. Una vez finalizada una partida, se sigue visualizando el último estado del juego y se indica quién es el ganador (si lo ha habido). Se utilizan

los siguientes atributos privados de tipo puntero (para lograr el polimorfismo):

```
JuegoLogT2* partida;       // apunta a un objeto del juego concreto al que se está jugando
InterfazGrafT2* interfaz;  //        a una interfaz para ese juego
JugadorAutoT2* jugador;    //        a un jugador automático para ese juego
```

Con los menús se seleccionan los objetos concretos a los que estos atributos apuntan, liberando y creando dinámicamente objetos de las clases concretas.

Además también es útil definir los siguientes tipos enumerados

```
typedef enum {conecta4, tresEnRaya, otelo} enuJuego;
typedef enum {otro, alea, minimax} enuJugador;
typedef enum {bajo=2, medio=4, alto=6} enuNivel;
typedef enum {humano, maquina} enuPartida;
```

y tener los siguientes atributos privados:

```
enuJuego juego;          // juego seleccionado
enuJugador tipoJugador;  // tipo de jugador para la máquina
enuNivel nivel;          // nivel del jugador (profundidad del árbol de juego)
Turno jugIni;            // jugador que comienza la partida
```

que guardan el estado de la aplicación.

Entrega

Se ha de implementar la clase abstracta `JugadorMinimaxT2` (sus métodos no abstractos) y las clases concretas para uno de los juegos concretos `JC` mencionados (Otelo o Conecta 4): `JuegoJC`, `InterfazGrafJC`, `JugadorAleaJC` y `JugadorMinimaxJC`.

Opcional: Implementar las clases concretas para el otro juego mencionado o para otros de similares características que se puedan ajustar al marco abstracto definido (por ejemplo, el *Go* o los de la familia de juegos *mancala*). Implementar diferentes heurísticas para un mismo juego y permitir que el usuario seleccione el tipo de jugador al que se quiere enfrentar. Enfrentar a dos jugadores automáticos con la misma o distinta heurística y mostrar al usuario espectador las partidas entre ambos.