1. Introducción

1.1 Qué hacer ante un problema real

Aun siendo conscientes de que no existe una definición universalmente aceptada, podemos considerar a la Informática como un cuerpo de conocimiento cuyo objetivo principal es la resolución de problemas por medio de un ordenador. Para llegar a ese objetivo se tendría que dar respuesta a las cuestiones siguientes:

a) ¿Puede un problema dado ser resuelto por un ordenador?,
b) ¿Cómo construimos el programa que lo resuelva?,
c) ¿ Cuánto tiempo y espacio necesitará la solución?,
d) ¿Resuelve realmente nuestro programa el problema dado?.

De forma gráfica, el proceso puede verse en la figura 1.1

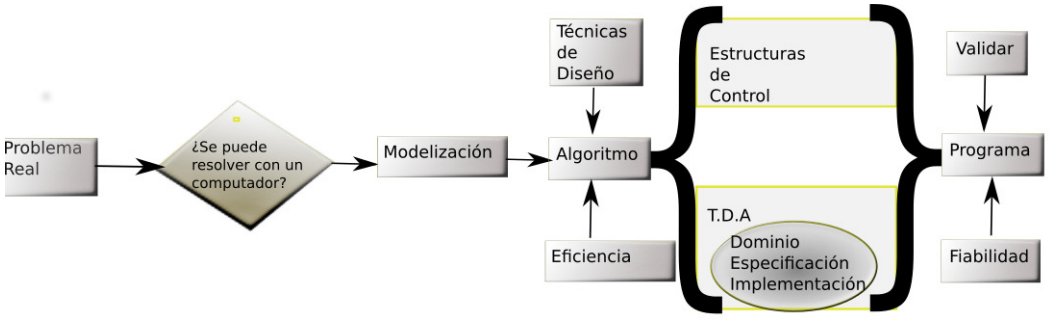


Figura 1.1: Esquema: Pasos a seguir para resolver un problema en un ordenador

La respuesta a la primera pregunta queda fuera del ámbito de este libro y la obviaremos considerando que todos los problemas que presentaremos serán resolubles mediante un ordenador. No obstante se puede indicar que para saber qué problemas pueden ser resueltos por un ordenador, es necesario disponer de un modelo matemático que sea capaz de ejecutar una descripción finita y bien especificada de una computación que pueda ser realizada en tiempo finito. Dicho de otra forma, es necesario un modelo matemático que se corresponda con la noción intuitiva de algoritmo. El modelo generalmente considerado como capaz de ejecutar cualquier algoritmo está basado en los trabajos realizados en la década de los 30 por el matemático inglés A. Turing. La conjetura de que cualquier computación para la que exista un algoritmo puede ser realizada por una máquina de Turing está basada en consideraciones muy sólidas y asimila el concepto de máquina de Turing con la noción de algoritmo. Esta asimilación permite abordar desde un punto de vista teórico la resolución de problemas usando el ordenador. Si no puede encontrarse una máquina de Turing que resuelva un problema y si aceptamos la conjetura anterior (Tesis de Church-Turing), el problema no puede ser resuelto por un ordenador. Así pues, la resolubilidad de un problema usando un ordenador está basada en la existencia de una máquina de Turing que lo resuelva.

Supuesto que un problema puede ser resuelto usando un ordenador, **¿ Cómo construímos el programa que lo resuelve ?.**

Haciendo un recorrido histórico habría que señalar que en la primera época del desarrollo de la informática los programas se expresaban en lenguaje máquina, que pueden ser ejecutados directamente por el ordenador, con los consiguientes problemas que esto acarreaba, debido a la poca flexibilidad cara al usuario de tales lenguajes, la dificultad para depurar de forma adecuada los programas y sobre todo el problema de que cada lenguaje solía estar ligado a un ordenador concreto de forma que los mismos programas no podían usarse en ordenadores diferentes. Surgió entonces la necesidad de idear otros mecanismos para construir y expresar los programas. El hilo conductor de tales mecanismos fue la **abstracción:** separar el programa del ordenador y acercarlo cada vez más al problema.

Los subprogramas empezaron ya a usarse a principios de los 50, dando lugar posteriormente al primer tipo de abstracción, la basada en procedimientos y funciones. Sin embargo en esos años solo eran considerados como formas de ahorrar trabajo y no propiamente como abstracciones. Un poco más tarde, a principios de los 60, se empezaron a entender los conceptos abstractos asociados a estructuras de datos básicas pero aún no se separaban los conceptos de las implementaciones. Con el nacimiento en esta época de los primeros lenguajes de alto nivel, Fortran p.ej., se llegó a la abstracción sintáctica (al abstraerse la semántica de las expresiones matemáticas y encapsular el acceso a ellas a través de la sintaxis propia del lenguaje). En cualquier caso con el desarrollo de estos lenguajes de alto nivel se solventaron los problemas de flexibilidad en la comunicación con el ordenador que hasta entonces se tenían y se empezaron a estudiar los programas de forma independiente del ordenador concreto en que se probaran y del lenguaje concreto en que se expresaran.

Otro mecanismo de abstracción importante que comenzó a desarrollarse a finales de los 60 fue el de los datos (tipos abstractos de datos). La idea es identificar los tipos de objetos que son necesarios para resolver un problema, y a continuación especificar sus propiedades y las operaciones básicas necesarias para trabajar con ellos. Se pasa posteriormente a su implementación utilizando las facilidades que dé el lenguaje con que trabajemos, y el resultado es la ampliación del lenguaje inicial con los nuevos tipos de datos construidos, a los que ya nos referiremos no por su implementación, sino por su especificación.

La pregunta natural que surge a continuación es cómo incorporar estas abstracciones en la cons-

trucción de un programa. Existen diversas metodologías para hacerlo, pero la mayoría comparten que para empezar y una vez que hemos descrito el problema y se ha desarrollado para el mismo un modelo matemático, se formulan algoritmos en términos de ese modelo. Esta primera versión se compone de ideas generales a través de las cuales expresamos la solución. En pasos sucesivos, se va refinando la solución, hasta que en un momento dado quedan claras las operaciones a realizar en los diferentes módulos en que dividimos la solución. De esta forma, y a partir de un pequeño número de partes con relaciones simples entre ellas, se construye un programa abstracto y solo después de finalizar este proceso de construcción, se entra en la fase de la implementación. Si cada concepto abstracto definido tiene su análogo en el lenguaje de programación base, el proceso finaliza, y si no, se establecerán los mecanismos para implementar los conceptos abstractos que nos queden. El resultado es un programa.

Para terminar este apartado sobre la construcción de programas, tres comentarios: primero, que ante el problema de encontrar técnicas adecuadas para diseñar algoritmos que cumplan ciertas especificaciones, podríamos dar una respuesta pesimista indicando que no hay recetas para inventar algoritmos de forma que cada problema algorítmico es un desafío para el diseñador. Lo que ocurre es que, por suerte, algunos algoritmos siguen algunos modelos generales, y apoyándose en esto se han desarrollado algunas técnicas de diseño de algoritmos, que vienen a ser modelos abstractos de los mismos aplicables a gran variedad de problemas reales. Así ante un nuevo problema, buscaremos algún modelo de algoritmo que haya dado buenos resultados para problemas similares y lo adaptaremos a nuestra situación concreta. Si no podemos hacerlo, todo quedará en manos de nuestra capacidad de diseño. El segundo comentario es que existen diversas metodologías en la construcción de programas, y este libro se enmarca dentro de la metodología del diseño orientado a objetos de gran importancia en el actual desarrollo software. El último comentario, es que no debe verse a la Informática solo como el arte de construir programas, sino como el estudio científico de las abstracciones, conceptos, modelos y métodos, que permitan analizar un problema, obtener una solución (algoritmo) y expresarla de una forma (programa) que en última instancia un ordenador sea capaz de entender y ejecutar.

En este punto de nuestro recorrido, ya tenemos un programa que en principio es solución de un problema. Se plantea entonces la duda de qué hacer en caso de que para el mismo problema seamos capaces de construir otro programa que también lo resuelva. ¿Cómo decidirnos por una u otra solución? o más aún ¿qué ocurre si el programa aún siendo correcto consume demasiados recursos y es inaceptable?. La respuesta viene dada a través del tercer punto en nuestro recorrido: el **análisis del tiempo y espacio que necesita una solución concreta**, en definitiva el estudio de la **eficiencia** de los programas.

La complejidad en espacio se mide por el número de variables y el número y tamaño de las estructuras de datos que se usan, y la complejidad en tiempo por el número de acciones elementales llevadas a cabo en la ejecución del programa. Tal tiempo y espacio ocupados por un programa son función del número de datos de entrada y los estudios de eficiencia se basan en el análisis del caso más desfavorable y el caso medio a la hora de procesar tales datos. Dichos estudios nos pueden dar una idea de cómo crecen los recursos necesarios conforme el tamaño de las entradas va en aumento y nos permiten comparar soluciones . Estos resultados también nos ayudarán a decidir la aceptabilidad de una solución recurriendo a las técnicas de análisis y diseño de algoritmos en caso de que rechazáramos tal algoritmo.

El último punto de referencia en nuestro recorrido viene inducido por el problema de comprobar si el programa que hemos construido resuelve realmente nuestro problema. Normalmente los programado-

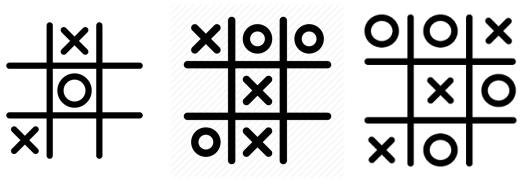
res prueban sus programas sobre una gran cantidad de datos de entrada para descubrir la mayoría de los errores lógicos presentes, aunque con este método (al que suele denominarse de **prueba y depuración**) no se puede estar completamente seguro de que el programa no contiene errores. Necesitaríamos para realizar la **verificación** , reglas que describan de forma precisa el efecto que cada instrucción tiene en el estado actual del programa para, aplicando dichas reglas, demostrar rigurosamente que lo que hace el programa coincide con sus especificaciones. En cualquier caso y cuando la prueba formal resulte muy complicada, podemos aumentar la confianza en nuestro programa realizando en el mismo los tests cuidadosos de que hablábamos al principio. Para ello es fundamental determinar el estado en que se encuentra el programa en cada punto, y usar estructuras de control ideales en las que cada trozo del programa tiene un solo punto de entrada y de salida. De esta forma, la legibilidad del programa aumenta y la verificación del mismo se hace más fácil. Quedaría finalmente preguntarnos si se puede construir un programa para realizar la validación. Desgraciadamente la respuesta es negativa si aceptamos como modelo de computación la máquina de Turing.

De todo lo anterior se deduce que las Estructuras de Datos y el Análisis y Diseño de Algoritmos, son áreas que están muy relacionadas por cuanto responden a la idea de que los programas no son más que formulaciones concretas de algoritmos basados en representaciones y estructuras concretas de datos, de forma que no se pueden tomar decisiones sobre los tipos de datos de un programa sin conocer los algoritmos que van a aplicarse a los mismos, y recíprocamente, la estructura y la elección de algoritmos a menudo dependen de los tipos de datos en que se apoyan. En definitiva los tipos de datos y los algoritmos están inseparablemente ligados, aunque no obstante se suele comenzar estudiando los primeros, fundamentalmente porque se tiene la idea intuitiva de que los datos son previos a los algoritmos, puesto que es preciso disponer primero de los objetos antes de operar con ellos. En cualquier caso, la esencia de la teoría es que los datos, en primera instancia, representan abstracción de fenómenos reales formulados como estructuras abstractas que no tienen por qué ser necesariamente las existentes en los lenguajes de programación usuales, y que, utilizadas adecuadamente dentro de los algoritmos junto con técnicas apropiadas de diseño de éstos, conformarán cualquier programa solución a un problema dado. Este libro tratará fundamentalmente con las estructuras de datos y el análisis básico de la eficiencia de algoritmos. Todos los tipos que estudiaremos comparten la idea de que las operaciones básicas que subyacen en su creación son las de inserción, borrado, búsqueda y ordenación y se evolucionará de unas a otras en base a buscar una mejora en eficiencia en alguna(s) de ellas.

1.2 Un primer ejemplo

Este ejemplo inicial solo pretende incidir en la importancia que tienen la abstracción y las estructuras de datos en la resolución de problemas. Para ilustrarlo, se va a programar la solución de un juego: el 3 en raya. Es un juego de “lapiz y papel” . Tiene numerosas variantes. La más simple es un juego entre dos jugadores (uno de los cuales puede ser un ordenador): O y X, que marcan los espacios de un tablero de 3×3 alternadamente. Cada jugador tiene como objetivo colocar sus fichas en una misma línea recta (horizontal, vertical o diagonal).

Ejemplos de partidas podrían ser:



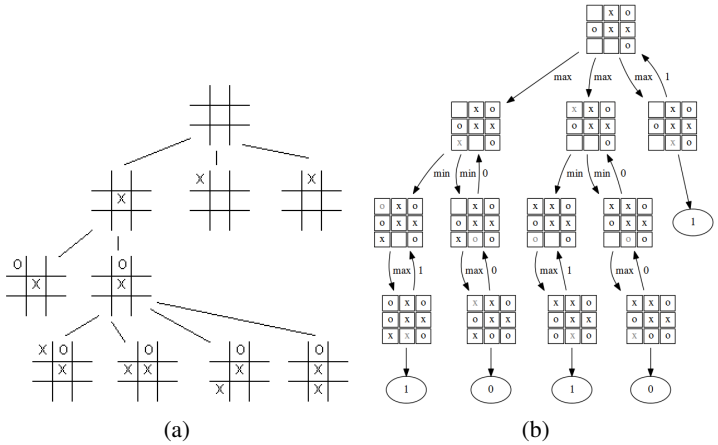
(a) (b) (c)

Figura 1.2: Diferentes posiciones del juego

Figura 1.3: Partida ganada por el primer jugador, X

Figura 1.4: Partida terminada en empate

Los jugadores no tardan en descubrir que el juego perfecto termina en empate sin importar qué haga el primer jugador. La simplicidad del juego de tres en raya lo hacen ideal como herramienta pedagógica para enseñar los conceptos de teoría de juegos y como ejemplo básico en la rama de IA que se encarga de la búsqueda de árboles de juegos.



(a) (b)

Figura 1.5: Posibles árboles asociados al juego

Si nos planteamos hacer un programa para jugar, hemos de pensar de forma coordinada en algorítmos y en estructuras de datos que los soporten, de forma que podrían derivarse múltiples programas solución al problema en función de la complejidad del algorítmo en que pensemos. Así, podría pensarse en el uso para la solución de estructuras de datos como vectores, matrices o árboles, solas o combinadas.

Aquí solo nos plantearemos una solución muy simple que no implique el uso de estructuras de datos complejas a las que nos enfrentaremos más adelante. Por tanto solo nos preocuparemos de la modularización del problema y de enfocar la solución.

1.2.1 Módulos para la solución

Al iniciar el juego se solicitarán los nombres de cada jugador y su nick. En el juego se debe especificar cuál de los jugadores es el primero (por elección o al azar), pudiendo decidirse que uno de los jugadores sea el ordenador.

Las clases que se implementen en el juego tendrán como base una matriz bidimensional de enteros con un tamaño de 3 por 3 que simulará el tablero del juego. El constructor debe inicializar la matriz: Los jugadores tienen un orden de juego especifico, el primer jugador realiza un movimiento y mostrará una O en el cuadro que desee escoger. Cuando toque al segundo jugador, este realizara un movimiento en el cuadro que desea escoger y mostrará una X. Al realizar cada movimiento el juego debe determinar si se ha ganado o existe un empate. Las clases deben tener atributos, métodos,constructores y las sobrecargas correspondientes. No usaremos una interfaz gráfica que podría añadirse más tarde.

Un esquema de los módulos y su relación entre sí, podría ser el de la figura 1.6

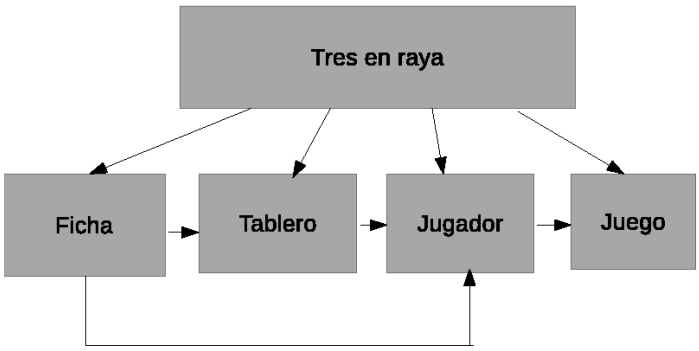


Figura 1.6: Esquema de relación entre los módulos en que se ha dividido el juego

Definición de ficha:

```
1  //fichero ficha.h
2
3  #ifndef __FICHA__H__
4  #define __FICHA__H__
5
```

```
#include <iostream>

// Tipos de fichas que se pueden poner en el tablero

enum Ficha {BLANCO, CIRCULO, CRUZ};

// Sobrecarga de << para mostrar en ostream el símbolo de la ficha

 ostream & operator<< (ostream & salida, const Ficha & fic);

#endif
```

Clase tablero:

Para definir el tablero, simplemente usamos una matriz de fichas 3x3 estática, añadiendo un campo que nos indique el número de fichas que se han puesto. Añadimos por comodidad 2 funciones privadas que nos permiten copiar un tablero y saber si una ficha de un tipo ha conseguido 3 en raya y completamos la clase en su parte pública con las funciones que se indican:

```
// fichero tablero.h

#ifndef __TABLERO__H__
#define __TABLERO__H__

#include <iostream>
#include "ficha.h"

class Tablero {

  private:
    Ficha tab[3][3];       // Tablero de fichas 3x3
    int numfichas;         // Número de fichas que han sido puestas en total

    // Copia un tablero desde orig
    void copia_tablero(const Tablero &orig);

    // Devuelve true/false si la ficha de tipo fic tiene 3 en raya
    bool hay3raya(const Ficha &fic) const;

  public:
    Tablero();                                  // Constructor por defecto
    ~Tablero() { };                             // Destructor (vacío)

    Tablero(const Tablero &orig);               // Constructor de copia
    Tablero& operator=(const Tablero &orig);    // Sobrecarga de asignación

    // Inicializa el tablero poniendo en blanco todas sus casillas
    void PonerEnBlanco();

    // Pone una ficha de color fic en la fila f y la columna c
    // Devuelve true si la operación ha tenido éxito y false en caso contrario.
```

```
33      // Sólo se pueden poner fichas en las casillas que estén en blanco
34      bool PonFicha(int f, int c, const Ficha &fic);
35
36      // Devuelve el tipo de ficha que hay en la posición (f,c)
37      Ficha QueFichaHay(int f, int c) const;
38
39      // Devuelve si hay algún tipo de ficha que tenga tres en raya. Si no hay 3 en raya
40      // devuelve el valor blanco.
41      Ficha Busca3Raya() const;
42
43      // Devuelve el número de fichas que hay puestas en el tablero
44      int CuantasFichas() const { return numfichas; };
45
46  };
47  // Para mostrar el tablero en pantalla sobrecargamos <<
48   ostream& operator<<( ostream &salida, const Tablero &tab);
49
50  #endif
```

Clase jugador:

Para definir un jugador lo hacemos con su nombre, ficha con la que juega y el tipo de jugador (0, humano, 1, máquina). Añadimos métodos privados que implementan distintas estrategias de juego y la parte pública la configuramos con las funciones que se indican:

```
1   // fichero jugador.h
2
3   #ifndef __JUGADOR__H__
4   #define __JUGADOR__H__
5
6   #include <iostream>
7   #include <string>
8   #include "ficha.h"
9   #include "tablero.h"
10
11  class Jugador {
12    private:
13       string nombre;      // Nombre del jugador
14      Ficha fic;              // Color de la ficha (cruz o circulo)
15      int nivel;              // Nivel del jugador
16
17      // Métodos privados que implementan distintas estrategias de juego
18      //   Nivel 0 : Juega una persona
19      //   Nivel 1 : Juega la CPU de forma muy básica
20      //   ... podríamos implementar nuevos niveles más "inteligentes"
21      void piensa_nivel_0(const Tablero &tab, int &fil, int &col) const;
22      void piensa_nivel_1(const Tablero &tab, int &fil, int &col) const;
23
24    public:
25      // No existe constructor por defecto. Cuando construimos un objeto de tipo
26      // jugador debemos asignarle un nombre y un color obligatoriamente.
```

```
    Jugador(const  string &n, const Ficha &f, int ni);

    //    ~Jugador() { };     // El destructor está vacio

    // Obtener el nombre del jugador
     string Nombre() const         { return nombre; };

    // Obtener el "color" de la ficha
    Ficha Color() const               { return fic; };

    // Le damos el tablero y nos devuelve dónde quiere poner ficha el jugador
    void PiensaJugada(const Tablero &tab, int &fil, int &col) const;
};

// Para mostrar los datos del jugador en consola
 ostream& operator<<( ostream &salida, const Jugador &jug);


#endif
```

Clase juego:

Para definir el juego simplemente se usan las definiciones que hemos hecho de tablero y jugador y se añade el turno. La parte pública la configuramos con las funciones básicas que se indican:

```
// fichero juego.h

#ifndef __JUEGO__H__
#define __JUEGO__H__

#include "tablero.h"
#include "jugador.h"

class Juego3Raya {
  private:

    Jugador jug1, jug2;    // Jugadores
    Tablero tab;           // Tablero
    int turno;             // A quien le toca jugar

  public:
    // No existe constructor por defecto
    // Contructor. Para crear un nuevo juego hemos de dar un tablero
    // y dos jugadores obligatoriamente
    Juego3Raya(const Tablero &t, const Jugador &j1, const Jugador &j2);
    ~Juego3Raya()  { };     // Destructor vacío

    void NuevoJuego();      // Prepara el juego para comenzar una nueva partida
    void JugarTurno();      // Avanza un turno

    // Devuelve una referencia (const) al tablero de juego (consultor)
```

```
    const Tablero &ElTablero() const { return tab; };

    // Devuelve una referencia al jugador n-ésimo (n=0 ó 1)
    const Jugador &ElJugador(int n) const;

    // Devuelve true si el juego ha terminado (porque haya 3 en raya
    //  o porque haya empate)
    bool HemosAcabado() const;

    // Devuelve el número de jugador a quien le toca poner ficha
    int AQuienLeToca() const  { return turno; };

    // Devuelve el número del jugador que ha ganado. Si aún no ha ganado
    // ninguno o hay empate devuelve -1
    int QuienGana() const;
};

#endif
```

Cómo se desarrollaría el juego:

Para programar la dinámica del juego, preguntamos por teclado los datos de un jugador, inicializamos el generador de números aleatorios, creamos un juego usando un tablero y los dos jugadores y se comienza el juego, añadiendo en cada paso una jugada alternada de cada jugados comprobando tras jugar si la partida ha terminado o no, sea porque un jugador ha ganado o porque termina en empate..

```
// programa principal

#include <iostream>
#include <ctime>    // Para función time()
#include <cstdlib>  // Para números aleatorios
#include "ficha.h"
#include "tablero.h"
#include "jugador.h"
#include "juego.h"

using namespace std;

// Preguntamos por teclado los datos de un jugador y lo devolvemos
Jugador LeeJugador(const Ficha f)
{
  string nom;
  int n;
  cout << "Dime el nombre del jugador "<< f <<" : ";
  cin >> nom;
  cout << "  Dime de que nivel es (0=humano, 1=aleatorio)";
  cin >> n;
  return Jugador(nom,f,n);
}


```

```
26 int main(int argc, char *argv[])
27 {
28   char p;
29
30   srand(time(0));  // Inicializamos el generador de números aleatorios
31   // Creamos un juego usando un tablero y dos jugadores leidos por teclado
32   Juego3Raya juego(Tablero(),LeeJugador(CRUZ),LeeJugador(CIRCULO));
33
34   // También se podría hacer de esta otra forma:
35   //   Jugador j1=LeeJugador(cruz);       // Creamos los jugadores
36   //   Jugador j2=LeeJugador(circulo);
37   //   Tablero tab;                       // Creamos un tablero
38   //   Juego3Raya juego(tab,j1,j2);       // Creamos el juego
39
40   do {
41     cout << "Los jugadores son: " << endl;
42     cout << "   " << juego.ElJugador(0) << endl;
43     cout << "   " << juego.ElJugador(1) << endl;
44     cout << "Comenzamos!!!" << endl << endl;
45
46     juego.NuevoJuego();    // Comenzamos el juego
47     do {
48       cout << "Le toca jugar a : " << juego.AQuienLeToca() << endl;
49       juego.JugarTurno();    // Avanzamos turno
50       cout << "Tras poner la ficha, el tablero queda así: " << endl
51            << juego.ElTablero() << endl;
52     } while (!juego.HemosAcabado());    // Comprobamos si hemos acabado
53
54     cout << "Se acabó la partida !!!" << endl;
55
56     int ganador=juego.QuienGana();   // Consultamos quien ganó
57     if (ganador==-1)
58       cout << "Hubo empate" << endl;
59     else
60       cout << "El ganador ha sido: " << juego.ElJugador(ganador) << endl;
61
62     cout << "¿Otra partida (S/N)?";
63     cin >> p;
64   } while ((p=='s') || (p=='S'));
65 }
```

La implementación de las distintas funciones en los diferentes módulos es muy simple y se sintetiza a continuación:

```
1 // fichero ficha.cpp
2
3 #include <cassert>
4 #include "ficha.h"
5
6  ostream& operator<<( ostream &salida, const Ficha &fic)
```

```
{
  assert((fic==BLANCO)||(fic==CIRCULO)||(fic==CRUZ));
  if (fic==BLANCO)
    salida << "   ";
  else if (fic==CIRCULO)
    salida << " O ";
  else   // fic==CRUZ
    salida << " X ";
  return salida;
}
```

```
// fichero tablero.cpp

#include <cassert>
#include "tablero.h"

using namespace std;

Tablero::Tablero()
{
  // El constructor pone en blanco el tablero
  PonerEnBlanco();
}

Tablero::Tablero(const Tablero &orig)
{
  copia_tablero(orig); //realmente en este caso no haría falta este constructor
}

void Tablero::PonerEnBlanco()
{
  // Ponemos en blanco el tablero
  for (int i=0; i<3; i++)
    for (int j=0; j<3; j++)
      tab[i][j] = BLANCO;
  numfichas=0;
}

Tablero& Tablero::operator=(const Tablero &orig)
{
  if (this!=&orig)
    copia_tablero(orig);
  return *this;
}

bool Tablero::PonFicha(int f, int c, const Ficha &fic)
{
  assert((f>=0) && (f<3) && (c>=0) && (c<3));     // Estamos dentro del tablero
  if (tab[f][c]==BLANCO) {                        // No hay ficha en esa casilla
    tab[f][c] = fic;
    numfichas++;
```

```
41      return true;
42    }
43    return false;    // Ya hay ficha en esa casilla
44  }
45
46  Ficha Tablero::QueFichaHay(int f, int c) const
47  {
48    assert((f>=0) && (f<3) && (c>=0) && (c<3));    // Estamos dentro del tablero
49    return tab[f][c];
50  }
51
52  Ficha Tablero::Busca3Raya() const
53  {
54    if (hay3raya(CIRCULO)) return CIRCULO;
55    if (hay3raya(CRUZ)) return CRUZ;
56    return BLANCO;    // No hay 3 en raya
57  }
58
59  ostream& operator<<(ostream &salida, const Tablero &tab)
60  {
61    salida << "   -------------"<<endl;
62    for (int i=0; i<3; i++) {
63      salida<<"   |";
64      for (int j=0; j<3; j++)
65        salida << tab.QueFichaHay(i,j) << "|";
66      salida << endl<<"   -------------"<<endl;
67    }
68    return salida;
69  }
70
71  // Método privado
72  void Tablero::copia_tablero(const Tablero &orig)
73  {
74    for (int i=0; i<3; i++)
75      for (int j=0; j<3; j++)
76        tab[i][j] = orig.tab[i][j];
77    numfichas=orig.numfichas;
78  }
79
80  // Método privado
81  bool Tablero::hay3raya(const Ficha &fic) const
82  {
83    // Buscaremos 3 en raya en las direcciones marcadas por estos vectores
84    static const int dir[4][2] = {{1,0},{1,1},{0,1},{1,-1}};
85
86    for (int f=0; f<3; f++) {    //  Recorremos todas las casillas
87      for (int c=0; c<3; c++) {  //
88        if (tab[f][c]==fic) {        // Cuando encontramos una casilla con fic
89          for (int d=0; d<4; d++) {  // Buscamos en las 4 direcciones 3 en raya
90            int fx=f, cx=c;          //  a partir de dicha ficha
91            fx+=dir[d][0];
92            cx+=dir[d][1];
93            int numfic=1;
```

```
          while ((fx>=0) && (fx<3) && (cx>=0) && (cx<3) && (tab[fx][cx]==fic)) {
            numfic++;
            fx+=dir[d][0];
            cx+=dir[d][1];
          }
          if (numfic==3) return true;
        }
      }
    }
  }
  return false;
}
```

```
// fichero jugador.cpp

#include <cassert>
#include <cstdlib>
#include "jugador.h"

using namespace std;

Jugador::Jugador(const  string &n, const Ficha &f, int ni)
                 : nombre(n), fic(f), nivel(ni)
{
  assert((nivel>=0)&&(nivel<2));   // Comprobamos que el nivel es correcto
}

 ostream& operator<<( ostream &salida, const Jugador &jug)
{
  salida <<jug.Nombre()<<"  ("<<jug.Color()<<")";
  return salida;
}

void Jugador::PiensaJugada(const Tablero &tab, int &fil, int &col) const
{
  // En función del nivel del jugador elegimos una estrategia u otra
  switch (nivel) {
    case 0: piensa_nivel_0(tab,fil,col);
            break;
    case 1: piensa_nivel_1(tab,fil,col);
            break;
  }
}

void Jugador::piensa_nivel_0(const Tablero &tab, int &fil, int &col) const
{
  cout << " El tablero es: " << endl << tab;
  do {
    cout << " ¿Donde pones ficha (dime fila y columna)? : ";
    cin >> fil >> col;
  } while ((fil<0)||(fil>2)||(col<0)||(col>2));
```

```
}

void Jugador::piensa_nivel_1(const Tablero &tab, int &fil, int &col) const
{
  cout << " ... estoy pensando ... " << endl;
  do {
    fil = rand()%3;
    col = rand()%3;
  } while (tab.QueFichaHay(fil,col)!=BLANCO);
  cout << "      ... y pongo ficha en (" << fil << "," << col << ")" << endl;
}
```

```
// fichero juego.cpp

#include <cassert>
#include "juego.h"

using namespace std;

Juego3Raya::Juego3Raya(const Tablero &t, const Jugador &j1, const Jugador &j2)
            : jug1(j1), jug2(j2), tab(t), turno(0)
{
}

void Juego3Raya::NuevoJuego()
{
  turno = (turno+1) % 2;  // Al comenzar un nuevo juego hacemos que comience
                          // a jugar el que perdió en la partida anterior
  tab.PonerEnBlanco();
}

void Juego3Raya::JugarTurno()
{
  int f,c;
  Jugador *jug[2] = {&jug1, &jug2};    // Vector de punteros a los jugadores
  // Este vector de punteros se usa para evitar usar un if dentro del bucle

  // Preguntamos al jugador mientras su jugada no sea válida
  do {
    jug[turno]->PiensaJugada(tab,f,c);
  } while (!tab.PonFicha(f,c,jug[turno]->Color()));
  turno = (turno+1) % 2;  // Avanzamos para el siguiente turno
}

bool Juego3Raya::HemosAcabado() const
{
  return ((tab.CuantasFichas()==9) || (tab.Busca3Raya()!=BLANCO));
}

const Jugador & Juego3Raya::ElJugador(int n) const
{
```

```
  assert((n==0)||(n==1));
  return ((n==0) ? jug1 : jug2);
}

int Juego3Raya::QuienGana() const
{
  Ficha g = tab.Busca3Raya();
  if (g==jug1.Color())
    return 0;
  else if (g==jug2.Color())
    return 1;
  return -1;    // No gana nadie
}
```

```
// fichero Makefile

all: raya

juego.o: juego.cpp juego.h tablero.h jugador.h
        g++ -Wall -c juego.cpp -o juego.o

tablero.o: tablero.cpp tablero.h ficha.h
        g++ -Wall -c tablero.cpp -o tablero.o

ficha.o: ficha.cpp ficha.h
        g++ -Wall -c ficha.cpp -o ficha.o

jugador.o: jugador.cpp jugador.h ficha.h
        g++ -Wall -c jugador.cpp -o jugador.o

main.o: main.cpp tablero.h ficha.h jugador.h juego.h
        g++ -Wall -c main.cpp -o main.o

raya: main.o jugador.o ficha.o tablero.o juego.o
        g++ -o raya main.o jugador.o ficha.o tablero.o juego.o

clean :
        -rm main.o ficha.o jugador.o tablero.o juego.o

mrproper : clean
```

Como vemos, programar el juego ha conllevado una etapa de abstracción, otra de representación y finalmente una de implementación. Obviamente, con más recursos (herramientas de documentación, otras estructuras de datos, etc) lo podríamos haber hecho mejor, pero el objetivo era simplemente mostrar la importancia de la modularización a la hora de resolver un problema.