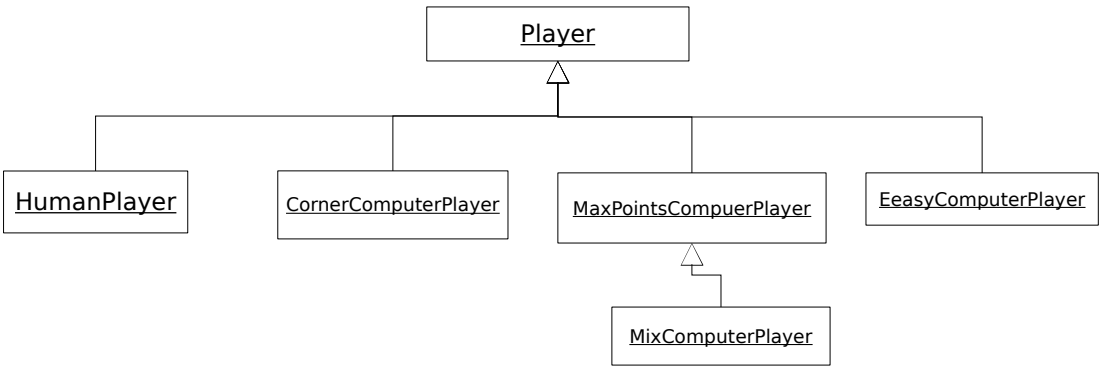
Melloras no deseño das clases

O deseño da aplicación se pode mellorar en varios aspectos. Esta guía orienta e programa algúns de eles. **É de especial interés comprender as razóns dos cambios propostos e comparar coa implementación orixinal.**

- Un dos detalles no código anterior é o método estático de ***Board*** para obter a representación en pantalla dunha ficha. Unha aproximación alternativa, máis flexible, e máis "orientada a obxecto" é considerar as fichas do xogo non como un número, se non coma un obxecto máis do xogo (***Piece***). Estes obxectos terían un nome e un valor como atributos, e serían capaces de retornar a súa representación como *String* (sobrepoñendo *toString()* de *Object*). Tamén é necesario poder determinar cando dúas *Piece* son iguais, polo que deberíamos sobrepoñer *equals*... Esto nos daría gran flexibilidade, xa que as Piece teñen a posibilidade ser moi distintas unhas de outras e servir para varios xogos distintos e os cambios son mínimos.

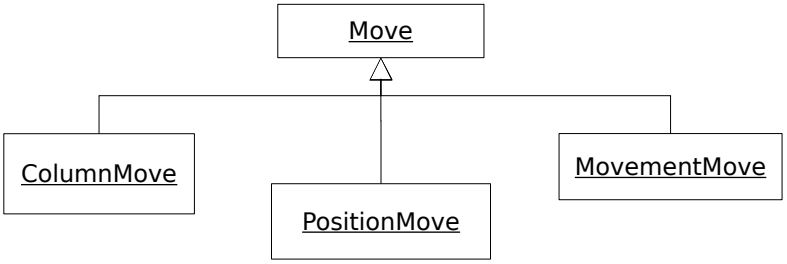
```
class Piece {
        private String name;
        private int value;

        public Piece(String name,int value) {
                this.name=name;
                this.value=value;
        }

        public int getValue() { return value;  }

        public String getName() { return name;  }

        @Override
        public String toString() { return name; }

        // Supoñemos que dúas piezas son iguais si teñen o mesmo nome e o mesmo valor
        @Override
        public boolean equals(Object opiece) {
                if (opiece==null) return false;
                if (this==opiece) return true;
                if (! (opiece instanceof Piece)) return false;
                Piece piece=(Piece) opiece;
                return (name.equals(piece.getName()) && (value==piece.getValue()));
        }
}
```

- Si examinamos a clase *Player*, vemos que en realidade representa únicamente a un *Player* humano. Un mellor deseño, e deseñar un *Player* xeral, herdando logo os Players específicos que se necesiten. A árbore de clases "correcta" sería:



Os xogadores “Player” realizan xogadas que no caso do Othello consisten en indicar a posición (representada cun obxecto ***Position***) en que se desexa xogar. Sen embargo si queremos que o Player poda participar en xogos nos que as xogadas funcionan de outro modo, a solución sería ter un obxecto que represente unha xogada xenérica, que podemos chamar ***Move***. Deste xeito, os xogadores (Player) crean movementos (Move). Nos xogos de taboleiro podemos nun principio distinguir entre tres tipos de moviementos.

- Movementos (Move) nos que a xogada simplemente é unha Posición (Position) consistente en un número de fila e un número de columna.
- Movementos (Move) nos que a xogada consiste nunha Posición de inicio, e unha Posición final, representando o desprazamento da ficha dun punto a outro.
- Movementos (Move) nos que a xogada consiste unicamente nun número de columna, coma o catro en raia



A implementación destas clases Move podería ser así:

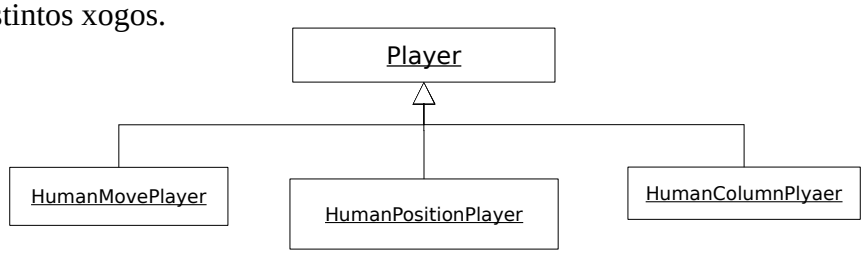
```
class Move {
        /** Constructor: Crea o movemento.
            Esta clase so serve para darlle unha raíz común a todos os tipos de movemento e podelos tratar como
            obxectos do tipo Move e aproveitar o polimorfismo
        */
        public Move()  { }
}
```

```
class MovementMove {
        Position fromPosition;
        Position toPosition;

        /** Constructor: Crea o movemento dunha peza dende a posición from á posición to
        */
        public MovementMove(Position from, Position to)
                fromPosition=from;
                toPosition=to;
        }

        /** Devolve a representación como String deste Move
        */
        @Override
        public String toString() {
                return "From "+fromPosition+" to "+toPosition;
        }
}
```

```
class PositionMove {
        Position position;

        /** Constructor: Crea o movemento (colocación)  dunha peza a posición position.
        */
        public PositionMove(Position pos) throws Exception {
                this.position=position;
        }

        /** Devolve a representación como String deste Move
         */
        @Override
        public String toString() {
                return position.toString();
        }
}
```

```
public class ColumnMove extends Move {
   public int column;  // Columna onde queremos xogar a partir de 0...

   /** CONSTRUCTOR (Sobrecargado): Crea o obxecto a partir da letra de columna A...
    */
   public ColumnMove(char column) throws Exception {
      // https://docs.oracle.com/javase/7/docs/api/java/lang/Character.html#getNumericValue(char)
      // Transformamos a letra en número
      this.column=Character.getNumericValue(column)-Character.getNumericValue('A');
      if ((this.column < 0)||(this.column>25))  throw new Exception("Bad Column");
   }

   /** CONSTRUCTOR (Sobrecargado): Crea o obxecto a partir do número de columna 0..
    */
   public ColumnMove(int column) throws Exception {
      if (column<0) throw new Exception("Bad Column");
      this.column=column;
   }

   /** Devolve a representación como String deste Move
    */
   @Override
   public String toString() {
      return "["+(((char)column)+'A')+"]";
   }
}
```

En canto a clase “Player” define que é un un Player xenérico. Os Player automáticos deben seguir unha estratexia e unhas regras que dependen do xogo, os humanos coñecen as regras e saben planificar as estratexias, polo que nese caso a clase Player so necesita permitir elixir o movemento a realizar (co teclado na consola nesta versión). Podemos deseñar clases Player que permitan a xogadores humanos realizar os distintos tipos de movementos (Movimento, Columna e Posición) e empregalos en distintos xogos.



Examinando o código feito anteriormente podemos deducir que como quedaría a clase Player...

```
class Player {
   private String name;   // Nome do xogador. TODOS os xogadores teñen un nome
   private int pieceId;     // ID da ficha ou do conxunto de fichas asociado ao xogador.
   protected Game game;    // O Player necesita coñecer a situación do xogo para planificar as xogadas

   // CONSTRUCTOR: Todos os Player se crean indicando o seu nome
   public Player(String name) { this.name=name;  }

   // Setter para a ficha. TODOS os Player teñen un tipo de fichas do xogo
   public void setPieceId(int id) {  pieceId=id;  }

   // Setter para o game. TODOS os Player teñen acceso ao Game no que participan
   public void setGame(Game g) {  game=g;  }

   // Getter para o nome. TODOS os Player son capaces de indicar o seu nome
   public String getName() {   return name;   }

   // Getter para a cor. TODOS os Player son capaces de indicar cales son as súas fichas
   public int getPieceId() {  return pieceId;  }

   // Sobreposicion: Sobrepoñemos toString para representar os Player.
   @Override
   public String toString() {
           return name+" (“+color+")";
   }

   // Elección da posición da xogada. TODOS os players poden elexir unha posición para xogar, pero non existe
   // un xeito de elixir común a TODOS os Player. Polo tanto, non é posible codificalo “en xeral”, e se lanza
   // unha “unchecked exception” que indica que non se sabe como xogar
  public Position doMovement()  {
      throw new UnsupportedOperationException("I don’t know how to play");
   }
}
```

A partir de ahí podemos crear os distintos *Player* que permitirán xogar a unha persoa contra os distintos xogos de tableiro, segundo o tipo de movimento que necesiten. Unicamente será necesario herdar de Player e codificar o algoritmo de ***doMovement()***:

```
// Permite a un Xogador Humano indicar un movemento consistente na posición na que se desexa xogar
class HumanPositionPlayer extends Player {

   // Constructor: Simplemente chama ao constructor da clase Player
   public HumanPositionPlayer(String name) {   super(name);    }

   // doMovement - realiza o movemento do xogo indicando a posición onde se desexa xogar
   // O xogador debe indicar a coordenada na forma 1A , 2B, 5C. O número indica a fila, a letra a columna
   @Override
   public Move doMovement() throws Exception {
      java.util.Scanner scn=new java.util.Scanner(System.in);
      Board board=game.getBoard();          // recuperamos o Board do xogo (Game)
      int row;
      char column;
      String line;

      board.show();                         // Visualiza o estado do taboleiro, para que o xogador poda decidir
      System.out.print("Xogada de "+this+"\n [1..][A..]?: ");
      line=scn.nextLine();
      row=(int)(line.charAt(0)-'0');        // Recuperamos a primeira letra, e a convertimos nun número
      column=line.charAt(1);                // Recuperamos a segunda letra
      Position pos=new Position(row,column);
      // Devolvemos o obxecto PositionMove (Obxecto Move de tipo “Posición”)
      return new PositionMove(pos);
   }
}
```

```
// Permite a un Xogador Humano indicar un movemento dende unha posición a outra
public class HumanMovePlayer extends Player {

   // Constructor: Simplemente chama ao constructor da clase Player
   public HumanMovePlayer(String name) {    super(name);    }

    // doMovement - realiza o movemento do xogo indicando un movemento dunha posición a outra as posicións
    // se indican de modo similar a anterior 1A-5C sería mover dende a posición 1A (0,0) á 5C (4,2)
   @Override
   public Move doMovement() throws Exception {
      java.util.Scanner scn=new java.util.Scanner(System.in);
      Board board=game.getBoard();          // recuperamos o Board do xogo (Game)
      int orow,drow;
      char ocolumn,dcolumn;
      String line;

      board.show();                         // Visualiza o estado do taboleiro para que o xogador poda decidir
      System.out.print("Movemento de "+this+"\n Movemento? [1..][A..]-[1..][A..]?: ");
      line=scn.nextLine();
      orow=(int)(line.charAt(0)-'0');       // Recuperamos a primeira letra, e a convertimos nun número
      ocolumn=line.charAt(1);               // Recuperamos a segunda letra
      drow=(int)(line.charAt(3)-'0');       // Recuperamos a primeira letra, e a convertimos nun número
      dcolumn=line.charAt(4);               // Recuperamos a segunda letra
      // Devolvemos o obxecto MovementMove (Move de tipo "Movemento")
      return new MovementMove(new Position(orow,ocolumn),new Position(drow,dcolumn));
   }
}
```

```
// Permite a un Xogador Humano indicar a columna na que desexa xogar
public class HumanColumnPlayer extends Player {

   // Constructor: Simplemente chama ao constructor da clase Player
   public HumanColumnPlayer(String name) { super(name);   }

   // doMovement - realiza o movemento do xogo indicando a letra da columna (A é 0, B é 1 ... etc)
   @Override
   public Move doMovement() throws Exception {
      java.util.Scanner scn=new java.util.Scanner(System.in);
      Board board=game.getBoard();          // recuperamos o Board do xogo (Game)
      char column;
      String line;

      board.show();                         // Visualiza o estado do taboleiro
      System.out.print("Xogada de "+this+"\n Columna [A..]?: ");
      line=scn.nextLine();
      column=line.charAt(0);                // Recuperamos a segunda letra
      // Devolvemos o obxecto ColumnMove (Move de tipo "Columna")
      return new ColumnMove(column);
   }
}
```

Tamén podemos programar os Player que simulan Xogadores para que o ordenador xogue, específicos para o xogo Othello. Como este tipo de xogador necesita elixir de modo automático o movemento, debe seguir a normas específicas de cada xogo (tanto para decidir o movemento como para seguir unha estratexia de xogo) e polo tanto non é trivial nin aconsellable deseñar clases Player que podan xogar a varios xogos, sendo moito mellor deseñar Player específicos segundo o xogo e a estratexia a seguir:

```
// Estratexia : Xogar no primeiro sitio dispoñible recorrendo dende a esquina superior dereita hacia abaixo
class OthelloEasyComputerPlayer extends Player {

  public OthelloEasyComputerPlayer(String name) { super(name);  }  // Constructor

 // doMovement – O Othello xoga con movementos de tipo PositionMove
 @Override
  public Move doMovement()  throws Exception {
     Board board=game.getBoard(); // Recuperamos o taboleiro do xogo
     int total;
     Position pos;

     for(int f=0;f<8;f++) { // Recorremos as filas
          for(int c=0;c<8; c++) { // Para cada fila, recorremos as columnas
                    if (board.get(f,c)==null) { // Si a posición non ten ficha, e posible que se poda xogar...
                              pos=new Position(f,c);
                              total=game.countPlay(this,pos,null);
                              if (total!=0) {
                                        System.out.println(this+": "+pos);  // Indico a xogada en pantalla
                                        return new PositionMove(pos);       // Devolvo o PositionMove asociado á posición
                              }
                    }
          }
     }
     return null;  // Aquí nunca debería chegar a execución. O  xogo non manda xogar si non existe xogada
  }
}
```

```
//  Estratexia : A estratexia é xogar no sitio que atrape máis fichas contrarias
class OthelloMaxPointsComputerPlayer extends Player {

  public OthelloMaxPointsComputerPlayer(String name) { super(name);  }  // Constructor

  // doMovement – O Othello xoga con movementos de tipo PositionMove
  @Override
  public Move doMovement()  throws Exception {
     Board board=game.getBoard();  // Recuperamos o taboleiro do xogo
     Position pos;
     Position maxpos=null;
     int total, maxtotal=0;

     for(int f=0;f<8;f++) { // Recorremos as filas
          for(int c=0;c<8; c++) { // Para cada fila, recorremos as columnas
                    if (board.get(f,c)==null) { // Si a posición non ten ficha, e posible que se poda xogar...
                              pos=new Position(f,c);
                              total=game.countPlay(this,pos,null);
                              if (total > maxtotal) { // O total de fichas conseguidas nesta posición é maior que a gardada
                                        maxpos=pos;
                                        maxtotal=total;
                              }
                    }
          }
     }
     System.out.println(this+": "+maxpos);  // Indico a xogada en pantalla
     return new PositionMove(maxpos);      // Devolvo o PositionMove asociado á posición
  }
}
```

```
// Estratexia: Xogar nas posicións máis cercanas ás esquinas en diagonal
class OthelloCornerComputerPlayer extends Player {

    public OthelloCornerComputerPlayer(String name) { super(name); } // Constructor

   // doMovement – O Othello xoga con movementos de tipo PositionMove
   @Override
   public Move doMovement()  throws Exception {
      Board board=game.getBoard(); // Recuperamos o taboleiro do xogo
      Position pos=null;
      int row=0, column=0;
      int incr=0;   // Dirección do movemento nas filas
      int incc=1;   // Dirección do movemento nas columnas
      int maxrow=7;  // fila límite
      int maxcol=7;   // Columna límite
      int minrow=1;  // Fila de comezo
      int mincol=0;   // Columna de comezo

      for(int i=0;i<64;i++) {  // Recorremos as 64 celas si é necesario
         if (board.get(row,column)==null) { // Si a posición non ten ficha, e posible que se poda xogar...
            pos=new Position(row,column);
            if (game.countPlay(this,pos,null)>0) break;  // Se pode xogar, rompemos o bucle
         }
         column=column+incc;
         row=row+incr;
         if ((column > maxcol)&&(incc!=0)) {  // Chegamos a maxcol, baixamos de fila ..
            column=maxcol;
            maxcol--;
            row++;
            incc=0;
            incr=1;
         }
         if ((row > maxrow)&&(incr!=0)) {  // Chegamos a maxrow, retrocedemos de columna ..
            row=maxrow;
            maxrow--;
            column--;
            incc=-1;
            incr=0;
         }
         if ((column < mincol)&&(incc!=0)) { // Chegamos a mincol, subimos de fila ..
            column=mincol;
            mincol++;
            row--;
            incc=0;
            incr=-1;
         }
         if ((row < minrow)&&(incr!=0)) { // Chegamos a minrow, avanzamos de columna ..
            row=minrow;
            minrow++;
            column++;
            incc=1;
           incr=0;
         }
      }
      System.out.println(this+": "+pos);   // Indico a xogada en pantalla
      return new PositionMove(pos);   // Devolvo o PositionMove asociado á posición
   }
}
```

```
/** Estratexia: Se xoga nunha das catro posicións libres máis prósimas á esquina. Si non é posible, se xoga na cela que atrape máis fichas contrarias
*/
class OthelloMixComputerPlayer extends MaxPointsComputerPlayer {

   public OthelloMixComputerPlayer(String name) { super(name); } // Constructor

   // doMovement – O Othello xoga con movementos de tipo PositionMove
   @Override
   public Move doMovement()  throws Exception {
      Position[] corner=getFreeCell();  // Obtemos as 4 celdas libres máis próximas ás esquinas
      Position pos=null;

      for(int i=0;(i<4)&&(pos==null);i++) {
         if ((corner[i]!=null)&&(game.countPlay(this, corner[i], null)!=0)) {
            pos=corner[i];
         }
      }
      if (pos==null)   return super.doMovement(); // Chamamos ao método de MaxPointsComputerPlayer
      System.out.println(this+" (MIX): "+pos);   // Con MIX distinguimos que estratexia elexiu a Posición
      return new PositionMove(pos);   // Devolvo o PositionMove asociado á posición
   }

   /** Método auxiliar
    * Devolve as catro primeiras celdas libres máis próximas as esquinas
    * @return unha táboa coas 4 celdas máis cercanas as esquinas e bordes
    */
   private Position[] getFreeCell()  {
      int inc=0;
      int rowi=0;
      int rowf=7;
      int coli=rowi;
      int colf=rowf;
      Board board=game.getBoard();
      Position[] pos=new Position[4];

      try {
                  while(rowi<rowf) {
                     while(coli<colf) {
                        if ((pos[0]==null)&&(board.get(rowi,coli)==null)) pos[0]=new Position(rowi,coli);
                        if ((pos[1]==null)&&(board.get(rowi,colf)==null)) pos[1]=new Position(rowi,colf);
                        if ((pos[2]==null)&&(board.get(rowf,colf)==null)) pos[2]=new Position(rowf,colf);
                        if ((pos[3]==null)&&(board.get(rowf,coli)==null)) pos[3]=new Position(rowf,coli);
                        coli++;
                        colf--;
                        if ((pos[0]!=null)&&(pos[1]!=null)&&(pos[2]!=null)&&(pos[3]!=null))
                           return pos;
                     }
                     // Percorre as diagonais hacia o centro
                     rowi++;
                     rowf--;
                     coli=rowi;
                     colf=rowf;
                  }
      } catch (Exception e) {  // board.get lanza unha excepción si a posición é erronea. Nunca debería pasar...
            System.out.println("ERROR: "+e.getMessage());
      }
      return pos;
   }
}
```

O resto da aplicación so se ve afectada pola existencia da nova clase **Piece**, que produciría cambios mínimos nos métodos que traballan coas fichas (varios en Game, e algún en Board, por exemplo).

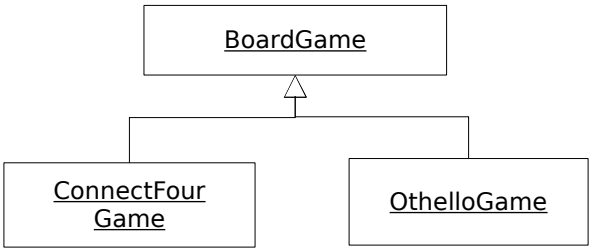
Outra reorganización de clases a podemos facer no propio xogo. Si nos fixamos no esquema de xogo (método *runGame* de **Game**), vemos que o esquema de xogo é o mesmo para prácticamente todos os *xogos de taboleiro con dous xogadores*:

Facer
O xogador elixe xogada
Se realiza a xogada (efectúa a xogada elixida polo xogador):
- Si é correcta, se cambia de quenda
Mentres non remate a partida
Se determina o gañador e se indica o resultado

En este esquema varias operacións son subxectivas, cada xogo as pode facer de modo distinto:

- **O concepto de *"partida rematada"*** depende do xogo.
- **O concepto de *"elixir xogada"*** depende do Xogador. Un humano simplemente examinará o estado do xogo e decidirá qué xogada debe facer, e é capaz de xogar a calqueira xogo do que coñeza as normas. En cambio, si queremos poder xogar contra o ordenador, necesitaremos un obxecto Xogador que elixa a xogada segundo as normas do xogo, polo que precisamos un "tipo de xogador" específico para cada xogo. (Como xa vimos anteriormente na clase ***Player***, xa o temos deseñado...)
- **O concepto de *"realizar xogada"*** depende do xogo, xa que debe realizarse seguindo as normas específicas do mesmo.
- **O concepto de *"cambio de quenda"***, normalmente sempre é igual: consiste en alternar entre dous valores que representan ao Player que ten o turno. O máis simple é facelo entre 1 e 0. Sen embargo, existen xogos con comportamentos "especiais", como o Othello en que non sempre se cambia de turno.

Si deseñamos deste xeito o obxecto Game, poderemos realizar facilmente xogos de distintos tipos (Othello, 4enRaia, Reversi, incluso damas ou xadrez) simplemente herdando da clase ***Game*** e sobrepoñendo os métodos particulares indicados anteriormente.



A clase ***BoardGame*** podería ser algo así:

```
public class BoardGame {
   protected Board board;      // Taboleiro
   prortected Player[] player;  // Xogadores
   protected int  turn;        // Quenda

   /** Constructor. Crea o xogo para os xogadores indicados.   */
   public BoardGame(Player p1, Player p2) {
      player=new Player[2];          // Creamos un Array de 2 players
      player[0]=p1;                  // player[0] e p1 referencian ao mesmo Player
      player[1]=p2;                  // player[1] e p2 referencian ao mesmo Player
   }

  /** Getter para o taboleiro   */
  Board getBoard() {      return board;     }

   /** Xogo: Este método implementa o algoritmo principal de todos os xogos de tableiro    */
   public void runGame() {
      Move move;
      Player winner;

      // Os xogadores necesitan os datos do xogo para planificar a súa estratexia.
      p1.setGame(this);              // player[0] participa en este Game
      p2.setGame(this);              // player[1] participa en este Game
      turn=initGame();               // Crea o taboleiro (Board), sortea as cores e pon as pezas na posición inicial

      // Facer
      do {
            try {
                  move=player[turn].doMovement();          // O xogador que lle toca, elixe xogada
                  doPlay(move);                            // Facemos a xogada
                  changeTurn();                            // Cambio de turno
            } catch(Exception e) {
                  System.out.println("Movemento “+move+” erróneo: "+e.getMessage());   // Xogada errónea
            }
      } while(!endGame());    // Mentras non remate o xogo
      winner=getWinner();     // Comprobamos o gañador
      if (winner==null)       System.out.println("Tie !!! (Empate)");
      else                    System.out.println("And the Winner Is...."+winner+"!!!!");
   }

   /**  Cambia o turno   */
   public void changeTurn() {      turn=1-turn;     }

   // -----> A partir de aquí non podo implementar os métodos, porque esta clase é demasiado xeral.

  /** Sortea as cores, crea o taboleiro (Board) e pon as pezas na posición inicial. NON se pode realizar,
       xa que é específica de cada xogo. DEVOLVE o xogador que comeza o xogo (0 ou 1)
  */
  protected int initGame() {
         throw new UnsupportedOperationException("I don’t know how create and initialize the board");
  }

  /** Realiza o movemento, si e correcto. En caso contrario lanza unha excepción    */
  public void doPlay(Move move) throws Exception {
         throw new UnsupportedOperationException("I don’t know the rule’s game");
  }

  /** Realiza o movemento, si e correcto. En caso contrario lanza unha excepción   */
  public boolean endGame() {
         throw new UnsupportedOperationException("I don’t know the rule’s game");
  }

  /**  Determina quen foi o gañador   */
  public Player getWinner() {
         throw new UnsupportedOperationException("I don’t know the rule’s game");
  }
}
```

O xogo do ***Othello*** quedaría:

```
public class Othello extends BoardGame {
  Piece[] pieces;                    // No Othello xogamos con 2 fichas. Unha negra e outra branca...

  public Othello(Player p1, Player p2) {  super(p1,p2); }  // CONSTRUCTOR: Chama ao constructor da clase pai

  @Override
  public void changeTurn() {  // Cambia o turno – Sobrepoño o método porque non sempre se cambia de turno
     if (canPlay(player[1-turn])) turn=1-turn;   // Cambio de turno so si o xogador ao que lle vai tocar pode xogar
  }

 @Override
 protected int initGame() {  // Sortea as cores, crea o taboleiro (Board) e pon as pezas na posición inicial.
    int color;
    Random r=new Random();

    r.setSeed(System.currentTimeMillis());  // Usamos os milisegundos como semilla para o número ao azar
    color=r.nextInt(2);          // Número ao azar : 0 ou 1
    board=new Board(8,8);     // taboleiro de 8x8
    pieces=new Piece[2];       // Podemos repetir a referencia en varias celas. Una peza pode estar en varias celas
    pieces[0]=new Piece("Brancas","+",1);
    pieces[1]=new Piece("Negras","-",2);
    player[0].setPieceId(color);      // O xogador 0 pode levar "Brancas" (pieces[0] ) ou "Negras" (pieces[1])
    player[1].setPieceId(1-color);   // O xogador 1 leva o contrario que o xogador 0
    board.put(3,3,pieces[0]);  // Branca
    board.put(4,4,pieces[0]);  // Branca
    board.put(3,4,pieces[1]);  // Negra
    board.put(4,3,pieces[1]);  // Negra
    return (1-color);             // Si color é 0, as negras as leva player[1] e viceversa
 }

 @Override
 public void doPlay(Move move) throws Exception {  // Realiza o movemento lanzando unha excepción si é incorr.
     // reverse gardará as fichas que gaña o movemento nas distintas direccións:
     // Arriba, Abaixo, Esquerda, Dereita, ArribaEsquerda, AbaixoEsquerda, ArribaDereita, AbaixoDereita
     int[] reverse={0,0,0,0,0,0,0,0};
     int total;                                          // Total de fichas que se gañan co movemento
     PositionMove pm=(PositionMove) move;        // Neste xogo os movementos son PositionMove
     int pieceId=player[turn].getPieceId();          // Recuperamos a peza que ten o xogador

     // Contamos cantas fichas gaña o movemento en en qué direccións
     total=countPlay(player[turn],pm.position,reverse);
     if (total==0) throw new Exception("Illegal Position"); // Non se gaña ninguha ficha. Movemento ERRÓNEO
     // Board lanza unha Exception si a posición está fora do tableiro ou está ocupada
     board.put(pm.position,pieces[pieceId]);  // Poñemos a Piece do xogador en pm.position (posición do Move)
     reverseAll(pm.position,pieces[pieceId],reverse); // Damos a volta as fichas
 }

 @Override
 public boolean endGame() {   // Devolve True si finaliza o xogo, se non devolve false
      return (!canPlay(player[0]) && !canPlay(player[1])); // true si ningun xogador pode xogar (o xogo finaliza)
 }

 @Override
 public Player getWinner() {   // Determina quen foi o gañador: 0 empate, 1 xogador 1, 2 xogador 2
         Piece piece0=pieces[player[0].getPieceId()];
         Piece piece1=pieces[player[1].getPieceId()];
         int total0=board.count(piece0);
         int total1=board.count(piece1);
         if (total0 == total1) return null;
         if (total0 > total1) return player[0];
         return player[1];
 }
```

```
//--- ---> Métodos auxiliares: Non forman parte xeral dos BoardGame, son funcións necesarias para
// resolver os algoritmos anteriores (estan arriba en vermello).

 //** Conta o número de fichas que revolve a xogada en r,c
private int countPlay(Player player, Position p,int[] rev) throws Exception {
   int total;
   int pieceId=player.getPieceId();
   if (rev==null) rev=new int[8];  // Si rev é null, creo o array...

   total=rev[0]=test(p,pieces[pieceId],-1,0);          // Up
   total+=(rev[1]=test(p,pieces[pieceId],1,0));        // Down
   total+=(rev[2]=test(p,pieces[pieceId],0,-1));       // Left
   total+=(rev[3]=test(p,pieces[pieceId],0,1));        // Right
   total+=(rev[4]=test(p,pieces[pieceId],-1,-1));      // UpLeft
   total+=(rev[5]=test(p,pieces[pieceId],1,-1));       // DownLeft
   total+=(rev[6]=test(p,pieces[pieceId],-1,1));       // UpRight
   total+=(rev[7]=test(p,pieces[pieceId],1,1));        // DownRight
   return total;
}

 // ** Indica si o xogador p ten algunha xogada posible. Devolve true si o xogador p pode xogar..
 // So accesible dende o mesmo paquete (friendly) xa que a necesitan os xogadores para a súa estratexia...
 boolean canPlay(Player player) {
   Position pos;

   try {
      // board.get pode lanzar excepcións, eu sei que non se lanzan porque as coordenadas están entre 0 e 8
      for(int r=0;r<8;r++)
         for(int c=0;c<8;c++) {
            // countPlay conta as fichas volteadas a partir da posición r,c
            pos=new Position(r,c);
            if ((countPlay(player,pos,null)>0)&&(board.get(r,c)==null)) return true;
         }
   } catch(Exception e) {}   // Non fago nada. Nunca se vai a producir a excepción
   return false;
}

// Pon ao valor Piece as celas indicadas pola táboa reverse
private void reverseAll(Position p, Piece piece, int[] reverse) throws Exception {
   reverse(p,piece,-1,0,reverse[0]);          // Up
   reverse(p,piece,1,0,reverse[1]);           // Down
   reverse(p,piece,0,-1,reverse[2]);          // Left
   reverse(p,piece,0,1,reverse[3]);           // Right
   reverse(p,piece,-1,-1,reverse[4]);         // LeftTop
   reverse(p,piece,1,-1,reverse[5]);          // LeftBottom
   reverse(p,piece,-1,1,reverse[6]);          // RightTop
   reverse(p,piece,1,1,reverse[7]);           // RigthBottom
}

private void reverse(Position p, Piece piece, int drow, int dcolumn,int nrev) throws Exception {
   int row,  column;
   Piece currenPiece;

   row=p.getRow()+drow;
   column=p.getColumn()+dcolumn;
   while(nrev>0) {
      currentPiece=board.get(row,column);
      // Si non temos ficha ou é igual que a actual, e un erro
      if ((currentPiece==null)||(piece.equals(currentPiece))) throw new Exception("Error reversing");
      board.set(row,column,piece);
      row+=drow;
      column+=dcolumn;
      nrev--;
   }
}
```

```
  /* comproba que se pode poñer a Piece piece na Position pos. Devolve  o número de fichas
     que se gañan movendose na dirección indicada por drow e dcolumn
  */
  private int test(Position pos,Piece piece,int drow,int dcolumn) {
      int r=p.getRow()+drow;                  // Comezamos na posición seguinte a que miramos
      int c=p.getColumn()+dcolumn;
      Piece boardPiece;
      int cta=0;

      try {
         boardPiece=board.get(r,c);           // Recuperamos a Piece do taboleiro
         while(!piece.equals(boardPiece)) {   // Mentras non sexa igual que a Piece que estamos poñendo....
            if (boardPiece==null) throw new Exception("Void Cell");
            cta++;                            // Contamos
            r+=drow;                          // Avanzamos na dirección indicada
            c+=dcolumn;
            boardPiece=board.get(r,c);        // Recuperamos a Piece do taboleiro
         }
      } catch(Exception e) { // Fora do tableiro ou celda vacía...
         cta=0; // Non se reviran fichas
      }
      return cta;
   }
}
```

Si quixeramos programar outro xogo, por exemplo o 4 en raia, bastaría heredar de ***BoardGame*** e facer os métodos *initGame*, *doPlay*, *endGame* e *getWinner* e implementar unha clases Player que siga unha estratexia para elexir xogadas si queremos xogar contra o ordenador.
Tamén podemos sobrepoñer calqueira dos outros métodos para adaptar o funcionamento da clase ao xogo que estemos a crear.

Se adxunta un proxecto *NetBeans* coa implementación dos dous xogos seguindo este esquema, o Othell e o catro en raia. As clases principais son ***RunOthello*** e ***RunConnectFour*** respectivamente. O Catro en Raia dispon de dous xogadores “automaticos” con estratexias simples. No Otello se implementan os xogadores aquí descritos. As clases “xenéricas” se puxeron no package “boardgames”, o Othello se atopa no package “Othello” eo catro en raia no package “ConnectFour”, deste xeito se aprecia mellor a estructura destas aplicacións.

Minesweeper

O minesweeper (buscaminas) é un **xogo de taboleiro** de un único **xogador** (*solitario*). Inicialmente o taboleiro está composto de **NxN celas** e **M bombas** repartidas aleatoriamente. Cada cela sen bomba indica o número de bombas que teñen ao seu redor.

Inicialmente se amosa o taboleiro con todas as celas ocultas, e se irán descubrindo segundo se van realizando xogadas. O obxectivo é descubrir todo o taboleiro.

O xogador elixe unha cela descubrindo o seu valor. Si o valor é 0 se amosa o valor de todas as celas adxacentes que estean a 0. Si é unha bomba, se perde. O xogo se gaña cando se amosan todas as celas agás as bombas.

Análise do xogo

- Como vemos e un xogo de taboleiro, pero de un único xogador (tipo “solitario”). Podemos consideralo como unha variante do tipo de xogos anterior (***BoardGame***), na que o algoritmo do xogo é lixeiramente diferente:

```
Facer
        O xogador elixe xogada
        Se realiza a xogada
Mentres non remate a partida
Se determina si se gañou
```

  Podemos considerar pol tanto os xogos de tipo ***“Solitario”*** como unha clase que herda (descendente) de ***BoardGame*** sobrepoñendo o método ***runGame()***.
- Os movementos son de tipo **PositionMove**, so indican unha posición de xogada.
- As bombas son un tipo de pezas (obxectos **Piece**) do xogo. Podemos representar as pezas que non son bombas cun valor que vai dende 0 a 8 segundo as bombas que as rodean, e cun valor de 100 (por elixir un...) para as pezas que son bombas. A única diferencia coas pezas “estándar” das que dispoñemos (obxectos Piece) é que en este xogo cando se visualizan teñen un aspecto antes de ser “descubertos” e outro cando xa están “descubertos”. Podemos crear unha nova clase ***MineSweeperPiece*** que incorpore esa característica (un atributo que indique o estado ‘descuberto’ ou ‘cuberto’)
- ***Este xogo permitirá taboleiros de calquera tamaño con calquera número de bombas*** polo que terá un construtor que permita indicalo.
- ***initGame*** consistirá en crear o taboleiro (Board), e colocar o número de bombas indicado (obxectos Piece co valor 100) de xeito que non se toquen. Logo se crean os obxecto Piece para o resto de celas co valor indicado según si ten ao redor bomba ou non.
- ***doPlay*** simplemente verifica que si é unha bomba se remata o xogo. Se non, se descubre esa celda e todas as veciñas que non teñan bombas ao lado.
- ***endGame*** verificará si queda algunha cela que non sexa bomba sen descubrir. Si non queda ningunha o xogo remata e se marcan como descubertas todas as pezas.
- ***getWinner*** non é necesario, e non se usa.

Implementación

A implementación pode quedar así:

```
//** Xogo de tipo "Solitario", é como un BoardGame, pero non ten cambio de turno de xogador

public class SolitaireBoardGame extends BoardGame {

   /** CONSTRUCTOR: Crea o xogo a partir dun único xogador
   public SolitaireBoardGame(Player player) {
      super(player, null);  // chama ao constructor da clase base cun único xogador
   }

   //** Leva a cabo o xogo. Este é o método que controla a execución dos xogos de taboleiro
   @Override
   public void runGame() {
      Move move=null;
      Player winner;

      player[0].setGame(this);        // player[0] participa en este Game
      initGame();                     // Crea e inicializa o taboleiro (Board)

      // Facer
      do {
         try {
            move=player[turn].doMovement();          // O xogador que lle toca, elixe xogada
            doPlay(move);                            // Facemos a xogada
         } catch(Exception e) {
            System.out.println("Movemento "+move+" erróneo: "+e.getMessage());        // Xogada errónea
         }
      } while(!endGame());   // Mentras non remate o xogo
      board.show();             // Amosamos o estado final do tableiro
      winner=getWinner();    // null, se perdeu, != null, se gañou
      if (winner==null)      System.out.println("You Lost!!!");
      else                   System.out.println("You Win!!!");
   }
}
```

```java
/**
 * Esta clase representa as pezas do xogo. Son capaces de manter o estado de "ocultas" ou
 * "visibles", amosando imaxes distintas
 */
public class MinesweeperPiece extends Piece {
    private boolean hidden;
    private final String hiddenImage;

    /** Facemos tres constructores sobrecargados entre outras cousas como_exemplo de sobrecarga de constructores
     * NON é necesario facelo así. Este constructor é privado, porque non quero que se instancien obxectos
     * con este constructor, so é para uso dos outros constructores
     */
    private MinesweeperPiece() {
        // Non necesito chamar a super. A clase base ten un constructor sen argumentos que se chama automáticamente
        name="";
        image=" ";
        value=0;
        hidden=true;    // Se crean "Ocultos"
        hiddenImage="X";
    }

    //** CONSTRUCTOR. A partir do nome. Si o nome é "BOMB" crea unha bomba Se non, crea unha peza normal
    public MinesweeperPiece(String type) {
        this();                             // Chamamos ao constructor de arriba
        if (type.equals("BOMB")) {
            name="Bomb";
            image="*";
            value=100;
        }
    }

    //** CONSTRUCTOR: Crea unha peza indicando o seu valor. Si o valor é 100 é unha Bomba
    public MinesweeperPiece(int value) {
        this();
        this.value=value;
        if (value==100) {
            name="Bomb";
            image="*";
        }
    }

    //**  Fai a peza visible
    public void show() {  hidden=false;   }

    //** Oculta a peza
    public void hide() {  hidden=true;  }

    //**  Devolve true si a peza está oculta
    public boolean isHidden() {  return hidden;  }

    //**  Devolve true si a peza é unha bomba
    public boolean isBomb() {  return (value==100);  }

    //** Devolve a presentación da peza tendo en conta o seu estado.
    @Override
    public String toString() {
        if (hidden) return hiddenImage;
        if ((value==0)||(value==100)) return super.toString();
        return Integer.toString(value);
    }
}
```

```
// **  Esta clase implementea o Xogo do buscaminas.

public class MinesweeperGame extends SolitaireBoardGame {
   Random rnd=new Random();    // Xerador de azar do xogo
   private int nbombs;
   private int nrows;
   private int ncols;
   private int discovered;
   private int end; // 0 non rematuo a partida, 1 se gañou, 2 se perdeu

   // ** CONSTRUCTOR: Se indica o xogador, o número de filas e columnas e o número de bombas desexado
   public MinesweeperGame(HumanPositionPlayer player,int nrows,int ncols,int nbombs) throws Exception {
      super(player);
      int maxbombs=(int)Math.ceil(nrows*ncols*80/100);      // Maximo un 80% de bombas
      if (nbombs > maxbombs) throw new Exception("Too much bombs");
      this.nrows=nrows;                                     // Numero de filas
      this.ncols=ncols;                                     // Numero de columnas
      this.nbombs=nbombs;                                   // Numero de bombas
      this.end=0;                                           // Cando valga 1, se gañou a partida. Si vale 2 se perdeu
      this.discovered=0;                                    // Nº de pezas descubertas
      rnd.setSeed(System.currentTimeMillis());        // Usamos os milisegundos como semilla para o número ao azar
   }

   //** Inicialización do xogo: Creamos o tableiro e colocamos as bombas
   @Override
   protected int initGame() {
      try {
         board=new Board(nrows,ncols);
         fillBoard();
      } catch(Exception e) {          //ERRO na inicialización. NUNCA debera pasar
          e.printStackTrace();        // Visualizamos a árbore de erro
         System.exit(1);              // Saimos do programa
      }
      return 0;
   }

   //** Realiza o movimento move conforme as regras do xogo.
   @Override
   public void doPlay(Move move) throws Exception {
      System.out.println("Xogada  en "+move);
      PositionMove m=(PositionMove) move;
      MinesweeperPiece p=(MinesweeperPiece)board.get(m.position.getRow(),m.position.getColumn());
      if (p.isBomb()) {
         end=2;
         discoverAll();
      }
      else {
        if (p.isHidden()) discover(m.position);
      }
   }

   //** Indica si o xogo finaliza. O xogo finaliza cando ningún pode xogar
   @Override
   public boolean endGame() {
      // Si xa temos todas as pezas descubertas, gañamos
      if (discovered==nrows*ncols-nbombs) end=1;
      return end!=0;
   }

   //**  Devolve o gañador. Si end é 2 se perdeu e devolvemos null
   @Override
   public Player getWinner() {
      if (end==2) return null;
      return player[0];
   }
```

```
//------>  Funcións Auxiliares (todas private)

//** Pon  as bombas e o resto de Pezas e calcula o seu valor segundo a proximiade ás bombas
private void fillBoard() throws Exception {
   int totcells=ncols*nrows;      // Total de celas
   Position[] filledArray=new Position[totcells]; // Táboa auxiliar
   Position[] trackArray=new Position[totcells];  // Posicións válidas
   int fcellsIdx,pbomb,track,tbombs;
   Piece bomb;
   int rowaux;
   int colaux;

   // Inicializamos as posicións válidas e a táboa auxiliar
   fcellsIdx=0;
   for(int row=0;row<nrows;row++) {
      for(int col=0;col<ncols;col++) {
         filledArray[fcellsIdx]=new Position(row,col);
         trackArray[fcellsIdx]=filledArray[fcellsIdx];
         fcellsIdx++;
      }
   }

   bomb=new MinesweeperPiece("BOMB");        // Creamos a bomba
   // Poñemos as bombas
   tbombs=0;
   while(tbombs<nbombs) {
      // Sorteo entre as posicións válidas e colocación da bomba.
      // En fcellsIdx teño o número de posicións válidas en trackArray
      pbomb=rnd.nextInt(fcellsIdx);
      board.put(trackArray[pbomb],bomb);

      // Eliminamos a bomba das posicións válidas
      rowaux=trackArray[pbomb].getRow();
      colaux=trackArray[pbomb].getColumn();
      pbomb=rowaux*ncols+colaux;                  // Posición en filledArray
      filledArray[pbomb]=null;

      // Xeneramos de novo a táboa de posicións válidas
      fcellsIdx=0;
      for(int idx=0;idx<totcells;idx++) {
         if (filledArray[idx]!=null) {
            trackArray[fcellsIdx]=filledArray[idx];
            fcellsIdx++;
         }
      }
      tbombs++;
   }

   // Poñemos o resto de pezas. Cada cela debe ter un obxecto Piece distinto, porque poden ter distintos valores
   for(int row=0;row<nrows;row++) {
      for(int col=0;col<ncols;col++) {
         bomb=board.get(row,col);
         if (bomb==null) { // A cela está baleira
            board.put(row,col,new MinesweeperPiece(getCellValue(row,col)));
         }
      }
   }
}
```

```
/** Calcula o valor dunha peza en funcion das bombas que a rodean
 private int getCellValue(int row, int col) {
    int value=0;
    MinesweeperPiece p;

    try {
       if (col>0) {          // Miro a esquerda e dereita
          p=(MinesweeperPiece)board.get(row,col-1);
          if ((p!=null)&&(p.isBomb())) value++;
       }
       if (col<ncols-1) {
          p=(MinesweeperPiece)board.get(row,col+1);
          if ((p!=null)&&(p.isBomb())) value++;
       }

       if (row>0) {          // Miro na fila de arriba (si existe)
          p=(MinesweeperPiece)board.get(row-1,col);
          if ((p!=null)&&(p.isBomb())) value++;
          if (col>0) {
             p=(MinesweeperPiece)board.get(row-1,col-1);
             if ((p!=null)&&(p.isBomb())) value++;
          }
          if (col<ncols-1) {
             p=(MinesweeperPiece)board.get(row-1,col+1);
             if ((p!=null)&&(p.isBomb())) value++;
          }
       }

       if (row<nrows-1) {            // Miro na fila de abaixo (si existe)
          p=(MinesweeperPiece)board.get(row+1,col);
          if ((p!=null)&&(p.isBomb())) value++;
          if (col>0) {
             p=(MinesweeperPiece)board.get(row+1,col-1);
            if ((p!=null)&&(p.isBomb())) value++;
          }
          if (col<ncols-1) {
            p=(MinesweeperPiece)board.get(row+1,col+1);
            if ((p!=null)&&(p.isBomb())) value++;
          }
       }
    } catch(Exception e) {          // ERRO: NUNCA debería pasar
         e.printStackTrace();       // Visualizamos a árbore de erro
         System.exit(1);            // Remato o programa
    }
    return value;
 }
```

```
  //** Cambia o estado da celda da Position indicada e de todas as que a rodean a visible
  private void discover(Position m) throws Exception {
     MinesweeperPiece p;
     int row=m.getRow();
     int column=m.getColumn();

     p=(MinesweeperPiece)board.get(row,column);
     if (p.isHidden()) {
        if (!p.isBomb()) {
           p.show();
           discovered++;
        }
        if (p.getValue()==0) {
           if (column>0) discover(new Position(row,column-1));
           if (column<ncols-1) discover(new Position(row,column+1));

           if (row>0) {
              discover(new Position(row-1,column));
              if (column>0) discover(new Position(row-1,column-1));
              if (column<ncols-1) discover(new Position(row-1,column+1));
           }
           if (row<nrows-1) {
              discover(new Position(row+1,column));
              if (column>0) discover(new Position(row+1,column-1));
              if (column<ncols-1) discover(new Position(row+1,column+1));
           }
        }
     }
  }

  //** Pon visibles todas as fichas (para amosar o taboleiro ao rematar
  private void discoverAll() {
     try {
        for(int row=0;row<nrows;row++) {
           for(int col=0;col<ncols;col++) {
              ((MinesweeperPiece)board.get(row,col)).show();
           }
        }
     } catch(Exception e) {  //ERRO: Non debería pasar nunca
        e.printStackTrace();
        System.exit(1);
     }
  }
}
```

Clase principal, que crea o xogo e o lanza:

```
public class RunMinesweeper {

  public static void main(String[] args) {
     HumanPositionPlayer player=new HumanPositionPlayer("Player");
     try {
        MinesweeperGame g=new MinesweeperGame(player,10,10,11);
        g.runGame();
     } catch(Exception e) {
        System.out.println("ERROR: "+e.getMessage());
     }
  }
}
```