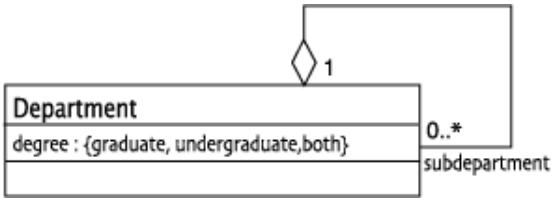
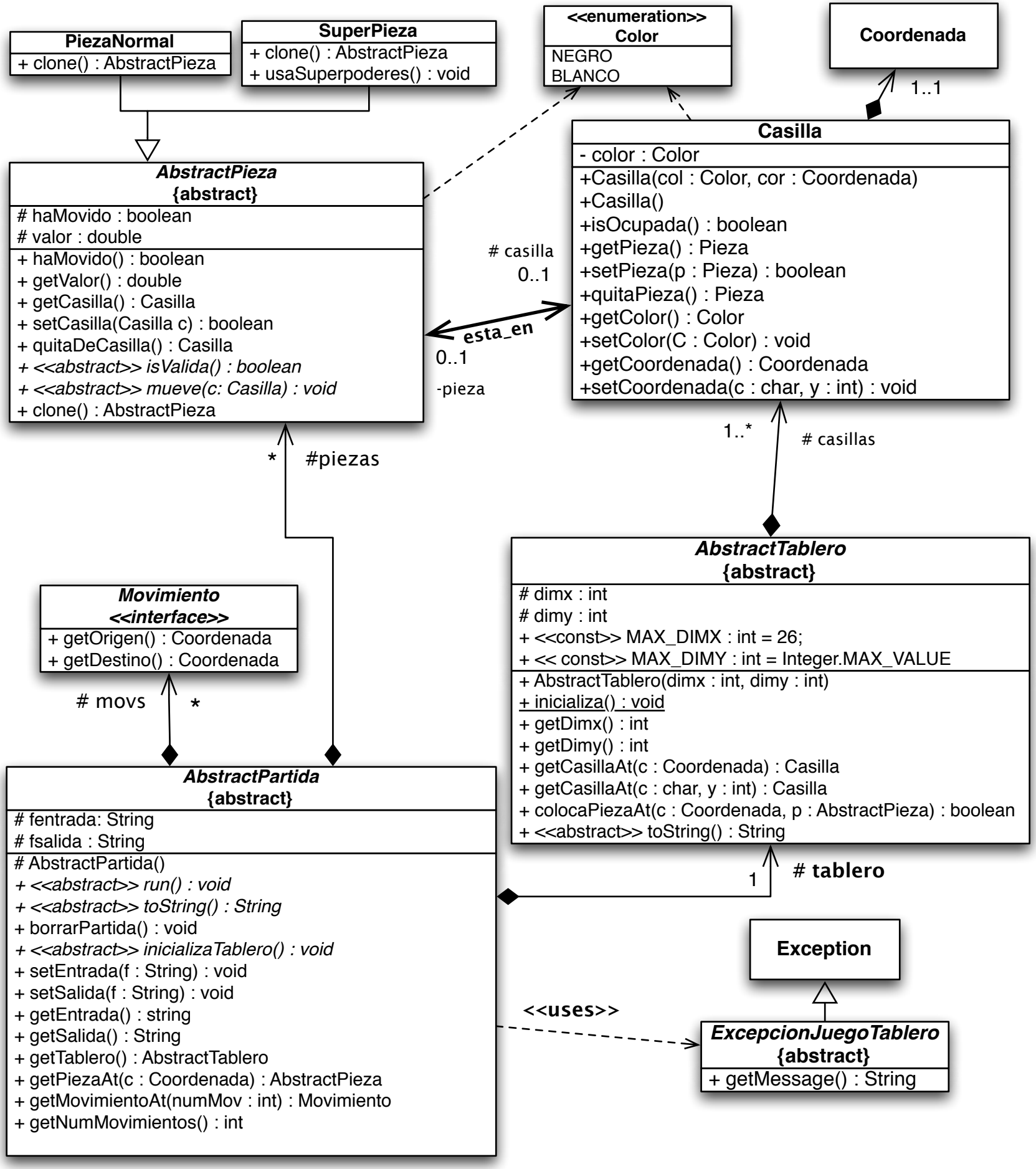
Programación 3
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Alicante

1. La forma canónica de una clase varía dependiendo del lenguaje de programación que se utilice. V
2. La siguiente relación



   es unaria y bidireccional. F
3. En java, al implementar una composición, la copia defensiva nos ayuda a prevenir que existan referencias a los objetos 'parte' que sean externas al objeto 'todo'. V
4. La clase `ExcepcionJuegoTablero` en el diagrama UML tiene una relación de dependencia con `AbstractPartida` porque usa información proporcionada por esta última. F
5. Dado el diagrama UML, el siguiente constructor de copia de `AbstractTablero` realiza una copia profunda de un objeto `AbstractTablero`. (suponemos que `casillas` está implementado como un `List<Casilla>`): F
```
public AbstractTablero(AbstractTablero otro) {
super(otro);
dimx = otro.dimx; dimy = otro.dimy;
casillas = new ArrayList<Casilla>();
for (Casilla casilla : otro.casillas)
casillas.add(casilla);
}
```
6. Esta implementación del método `AbstractTablero.inicializa()` del diagrama UML no compilará: `public static void inicializa()` V
```
{
dimx = 0;
dimy = 0;
casillas = null;
}
```
7. A partir del diagrama UML, podemos deducir que diferentes objetos `AbstractTablero` pueden compartir objetos de tipo `Casilla` a través de la relación `casillas`. F
8. Todas las clases que representan excepciones en Java tienen a la clase `Object` como una de sus superclases. V
9. Los lenguajes de programación soportan el reemplazo o refinamiento como una forma de sobrecarga o sobrescritura, pero no hay ningún lenguaje que proporcione ambas técnicas (por ejemplo, Java sólo soporta reemplazo y C++ sólo soporta refinamiento). F
10. Un lenguaje puede combinar tipado estático en algunas construcciones del lenguaje y tipado dinámico en otras. V
11. En el diagrama UML, el método `mueve()` de la clase `AbstractPieza` tiene enlace dinámico. V

12. En el diagrama UML, el comportamiento por defecto del método `AbstractPieza.isValida()` es devolver falso. Por tanto, esta implementación del método en la clase `AbstractPieza` es correcta: F
    ```
    public boolean isValida() { return false; }
    ```
13. En Java, si un método no abstracto `f()` definido en una superclase se sobrescribe en una de sus subclases, se puede invocar desde la subclase el método de la superclase mediante la instrucción `super.f()`. V
14. Dado el diagrama de clases de la figura **??** el siguiente código de Java contiene un error de compilación (suponiendo que ninguno de los métodos invocados declara lanzar excepciones): V
    ```
    AbstractPieza a = new AbstractPieza();
    a.setCasilla(null);
    System.out.println(a.getValor());
    ```
15. Una interfaz puede implementar otra interfaz. F
16. Una clase abstracta no puede tener constructores. F
17. En Java sólo las clases pueden ser genéricas, no así los interfaces. F
18. Para usar reflexión en Java debemos conocer en tiempo de compilación el nombre de las clases que queremos manipular. F
19. La inversión de control en los frameworks es posible gracias al enlace dinámico de métodos. V
20. La refactorización nunca produce cambios en las interfaces de las clases. F
21. Cuando se aplica correctamente, el principio de responsabilidad única (*Single Responsability Principle*) conduce a diseños con mayor acoplamiento. F
22. La refactorización *sustituir condicional con polimorfismo* contribuye positivamente a cumplir el *principio abierto/cerrado* (*Open-Closed Principle*). V


PiezaNormal
+ clone() : AbstractPieza
SuperPieza
+ clone() : AbstractPieza
+ usaSuperpoderes() : void
<<enumeration>>
Color
NEGRO
BLANCO
Coordenada
1..1
Casilla
- color : Color
+Casilla(col : Color, cor : Coordenada)
+Casilla()
+isOcupada() : boolean
+getPieza() : Pieza
+setPieza(p : Pieza) : boolean
+quitaPieza() : Pieza
+getColor() : Color
+setColor(C : Color) : void
+getCoordenada() : Coordenada
+setCoordenada(c : char, y : int) : void
AbstractPieza
{abstract}
haMovido : boolean
valor : double
+ haMovido() : boolean
+ getValor() : double
+ getCasilla() : Casilla
+ setCasilla(Casilla c) : boolean
+ quitaDeCasilla() : Casilla
+ <<abstract>> isValida() : boolean
+ <<abstract>> mueve(c: Casilla) : void
+ clone() : AbstractPieza
# casilla
0..1
esta_en
0..1
-pieza
1..*
casillas
*
#piezas
Movimiento
<<interface>>
+ getOrigen() : Coordenada
+ getDestino() : Coordenada
# movs
*
AbstractTablero
{abstract}
dimx : int
dimy : int
+ <<const>> MAX_DIMX : int = 26;
+ << const>> MAX_DIMY : int = Integer.MAX_VALUE
+ AbstractTablero(dimx : int, dimy : int)
+ inicializa() : void
+ getDimx() : int
+ getDimy() : int
+ getCasillaAt(c : Coordenada) : Casilla
+ getCasillaAt(c : char, y : int) : Casilla
+ colocaPiezaAt(c : Coordenada, p : AbstractPieza) : boolean
+ <<abstract>> toString() : String
1
tablero
AbstractPartida
{abstract}
fentrada: String
fsalida : String
AbstractPartida()
+ <<abstract>> run() : void
+ <<abstract>> toString() : String
+ borrarPartida() : void
+ <<abstract>> inicializaTablero() : void
+ setEntrada(f : String) : void
+ setSalida(f : String) : void
+ getEntrada() : string
+ getSalida() : String
+ getTablero() : AbstractTablero
+ getPiezaAt(c : Coordenada) : AbstractPieza
+ getMovimientoAt(numMov : int) : Movimiento
+ getNumMovimientos() : int
Exception
<<uses>>
ExcepcionJuegoTablero
{abstract}
+ getMessage() : String