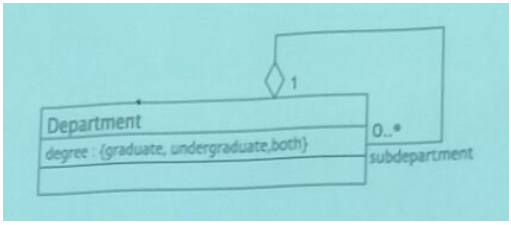
1. La forma canónica de una clase varía dependiendo del lenguaje de programación que se utilice.
2. La siguiente relación es unaria y bidireccional.



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'.
4. La clase 'ExcepcionJuegoTablero' en el diagrama UML tiene una relación de depedencia con AbstractPartida porque usa información proporcionada por esta última.
5. Dado el diagrama UML, el siguiente constructor de copia de AbstractTablero realiza una copia profunda de un objeto AbstractTablero. (Suponemos que casillas esta implementado como un List<Casilla>):

```
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(){
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.
8. Todas las clases que representan excepciones en Java tienen a la clase object como una de sus superclases.
9. Los lenguajes de programación soportan el reemplazo o refinamiento como una forma de sobrecarga o sobreescritura, pero no hay ningún lenguaje que porporcinoe ambas técnicas (por ejemplo Java solo soporta reemplazo y C++ solo soporta refinamiento)
10. Un lenguaje puede combinar tipado estático en algunas construcciones del lenguaje y tipado dinámico en otras;
11. En el diagrama UML, el método mueve() de la clase AbstactPieza tiene enlace dinámico.

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:

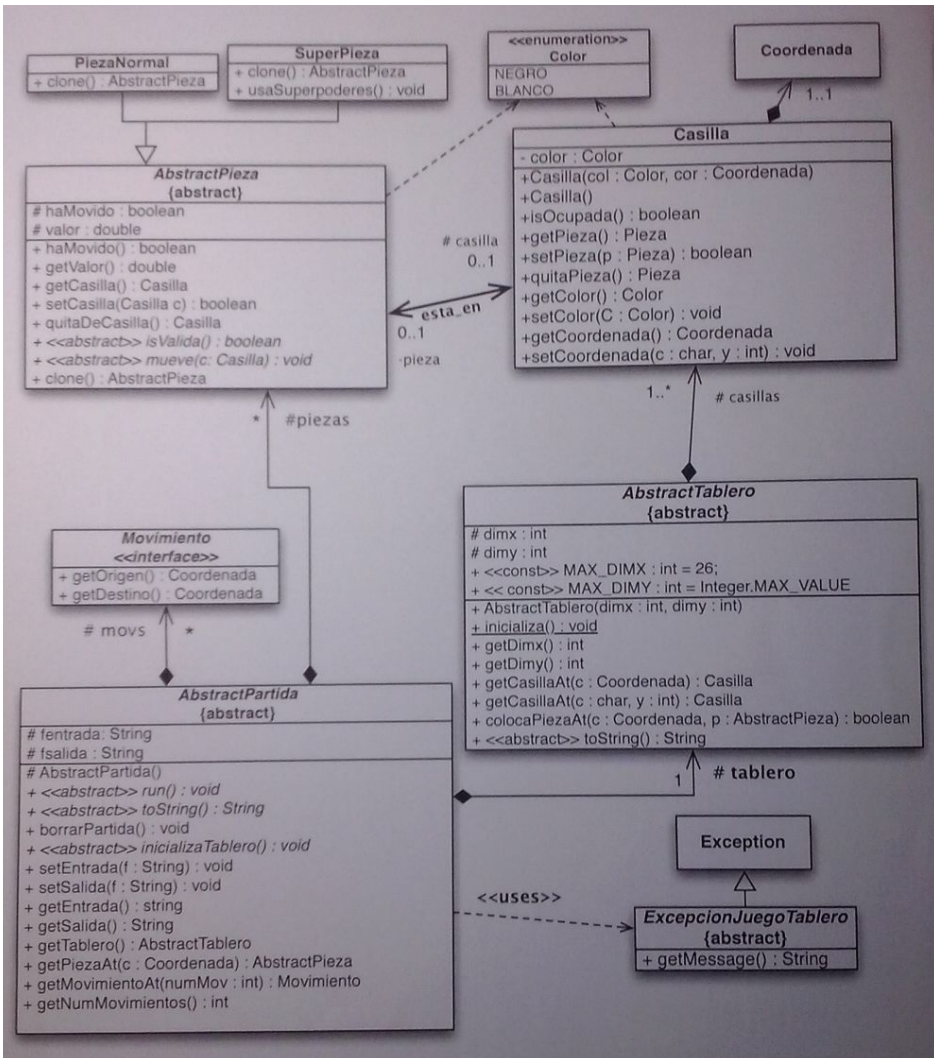
```
Public boolean isValida(){ return false;}
```

13. En java, si un método no abstracto f() definido en una superclase se sobreescribe en una de las subclases se puede invocar desde la subclase el método de la superclase mediante la instrucción super.f()
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):

```
AbstractPieza a = new AbstractPieza();
a.setCasilla(null);
System.out.ptinln(a.getValor());
```

15. Una interfaz puede implementar otra interfaz.
16. Una clase abstracta no puede tener constructores.
17. En java solo las clases pueden ser genéricas, no así los interfaces.
18. Para usar reflexión en java debemos conocer en tiempo de compilación el nombre de las clases que queremos manipular.

19. La inversión de control en los frameworks es posible gracias al enlace dinámico de métodos.
20. La refactorización nunca produce cambios en los interfaces de las clases.
21. Cuando se aplica correctamente, el principio de responsabilidad única (Single Responsability Principle) conduce a diseños con un mayor acoplamiento.
22. La refactorización *sustituir condicional con polimorfismo* contribuye positivamente a cumplir el principio abierto/cerrado (Open-Closed principle)



1	V	6	V	11	V	16	F	21	F
2	F	7	F	12	F	17	F	22	V
3	V	8	V	13	V	18	F		
4	F	9	F	14	V	19	V		
5	F	10	V	15	F	20	F		