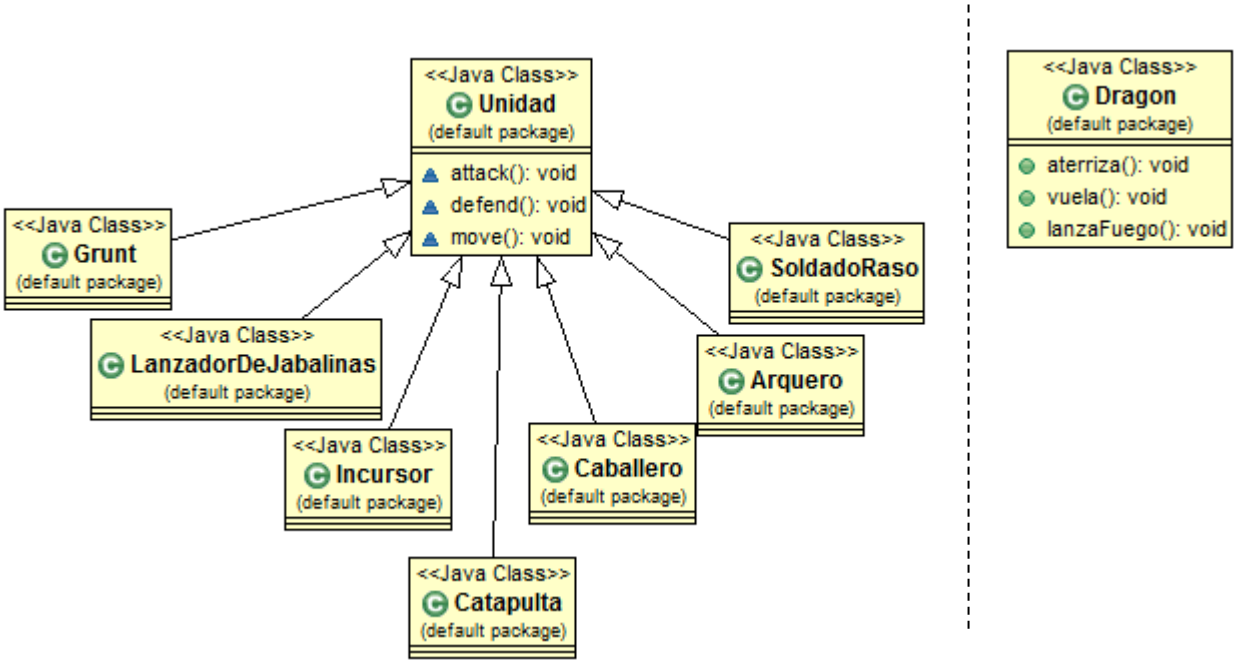
El juego Warcraft: Humanos vs Orcos© es un juego de estrategia donde dos especies combaten entre sí: los humanos y los orcos. Cada Especie dispone de los mismos tipos de unidades militares, aunque particularizada para cada especie. Para representar en memoria estos personajes se ha decidido utilizar la siguiente jerarquía:



Un programador colega nuestro nos proporciona una clase Dragon que nos gustaría incluir al conjunto de unidades de nuestro Juego. Sin embargo esta clase nueva no respeta la interfaz de Unidad así que no podemos hacer que herede directamente de ella. Para ello decidimos que un dragón actuando como Unidad debe:

- Atacar: echando a volar y a continuación lanzando fuego.
- Defenderse: atacando (para eso es un dragón)
- Moverse: volando y luego aterrizando.

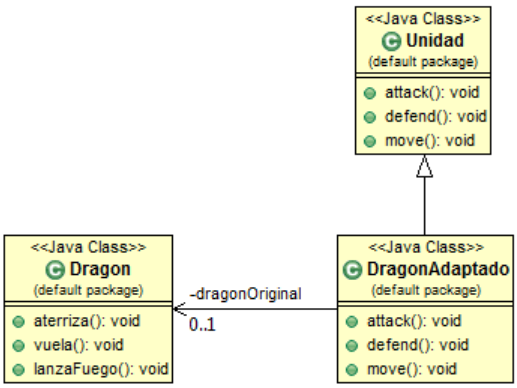
Se pide:

1º/ Indica cual sería el patrón adecuado para poder crear y manipular dragones como un tipo más de unidades. **Justifica tu respuesta**. **(1 punto)**

Aquí debemos utilizar un patrón estructural Adapter, ya que la interfaz de Dragon no se corresponde con la de Unidad. En nuestro caso ya que tanto Dragon como Unidad son clases, (y Java no tiene herencia múltiple) lo más adecuado sería utilizar el patrón por composición.

2º/ ¿Qué clases/interfaces habría que crear y como se combinan con Unidad? **Justifica tu respuesta**. **(2 puntos)**

Para ello crearemos una nueva subclase de Unidad denominada DragonAdaptado, que redefina sus operaciones y que utilice por composición un objeto Dragon. Gráficamente:



3º/ Implementa este nuevo tipo de Unidad dragón con el comportamiento indicado en el enunciado, e indicando las clases de las que hereda y las interfaces que implementa: **(1,5 puntos)**

```
public class DragonAdaptado extends Unidad {

	// Dragon por composición (campo privado)
	private Dragon dragonOriginal = new Dragon();

	//acciones
	/**
	 * attack - un dragón atacará volando y lanzando fuego desde el aire
	 */
	public void attack(){
		this.dragonOriginal.vuela();
		this.dragonOriginal.lanzaFuego();
	}

	/**
	 * defend - un dragón se defenderá "atacando"
	 */
	public void defend(){
		attack();
	}

	/**
	 * move - un dragón se mueve volando y luego aterrizando
	 */
	public void move(){
		this.dragonOriginal.vuela();
		this.dragonOriginal.aterriza();
	}
}
```

Una vez implementada esta clase de unidad, tras varias partidas de prueba los jugadores beta-testers concluyen que estas unidades Dragon son excesivamente poderosas. Así que los diseñadores del juego deciden que a lo largo de una partida sólo puede haber un dragón en juego, y aunque aparezca en distintas partidas será siempre el mismo dragón (así por ejemplo si hubiese resultado herido, al volver a aparecer éste seguirá con el mismo daño). Los diseñadores nos piden una solución que cumpla los tres requisitos siguientes:

- Que el dragón sea siempre el mismo durante toda la ejecución del juego.
- Evitar que el resto de programadores puedan (por error) crear distintos dragones
- Como además el código del juego está dividido en múltiples clases, que se pueda acceder a ese objeto dragón desde cualquier método/clase del código del juego.

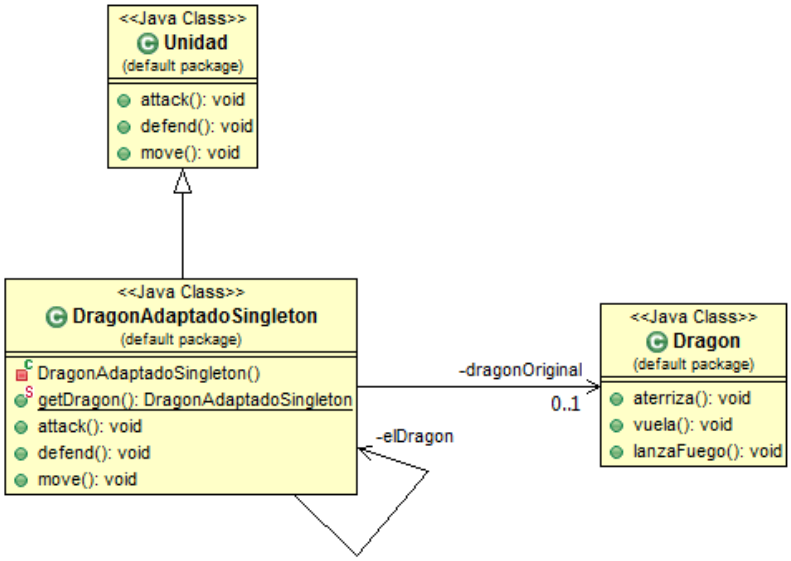
Se pide:

1º/ Indica cual sería el patrón adecuado para manejar de esta manera el dragón y justifica tu elección.

La elección en este caso es claramente el patrón creacional Singleton. Este patrón nos ofrece una accesibilidad global a un objeto que se accederá a través de los métodos estáticos de la clase DragonSingleton. Esta clase sustituye a la clase DragonAdaptado ocultando sus constructores para evitar la creación de otras instancias fuera de la propia clase. Sus métodos de clase nos permitirán devolver siempre la misma instancia DragonSingleton.

5º/ Define la jerarquía de objetos necesaria para implementar esta funcionalidad. Describe e implementa los métodos que tendrá cada nueva Interfaz/Clase. Indicando además quién y cómo se cubren los tres requisitos expuestos. Implementa las operaciones.

En realidad no necesitamos crear nuevas clases o interfaces, en su lugar ampliamos la clase DragonAdaptado con el comportamiento descrito en el apartado anterior y renombramos la clase como DragonAdaptadoSingleton (esto último simplemente para que el nombre describa la clase lo mejor posible)




```
public class DragonAdaptadoSingleton extends Unidad{

	// La instancia privada de dragón
	private static DragonAdaptadoSingleton elDragon = new DragonAdaptadoSingleton();

	// El constructor pasa a ser privado para que no se puedan crear objetos
	// fuera de la clase
	private DragonAdaptadoSingleton(){

	}

	// Único método de clase público que permite acceder a la instancia real
	public static DragonAdaptadoSingleton getDragon(){
		return elDragon;
	}

	/* Aquí irían las propiedades y operaciones de DragonAdaptado */

	private Dragon dragonOriginal = new Dragon();
```

```
	public void attack(){
	...
	}
	public void defend(){
	...
	}
	public void move(){
	...
	}
}
```

6º/ Completa el siguiente ejemplo de código con dos clases distintas C1 y C2, cada una con un método donde se requiere a la unidad dragón y se invoque a uno de sus métodos como unidad. **(1 punto)**

```
class C1{
	void m1(){
		// variable Unidad que contendrá el dragón
		Unidad d;
		// obtenemos el dragón
		d = DragonAdaptadoSingleton.getDragon();
		// lo utilizamos para atacar
		d.attack();
	}
}

class C2 {
	void m2(){
		// variable Unidad que contendrá el dragón
		Unidad d;
		// obtenemos el dragón
		d = DragonAdaptadoSingleton.getDragon();
		// lo utilizamos para moverlo
		d.move();
	}
}
```