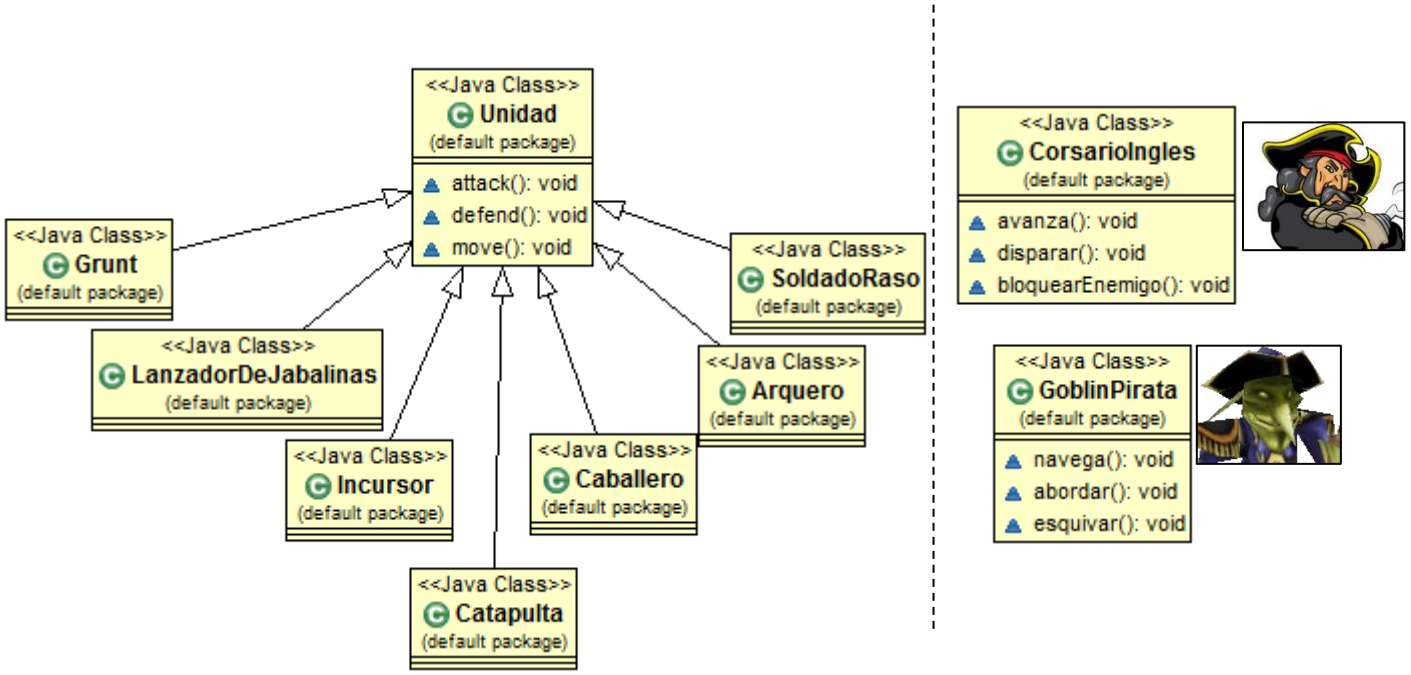
El juego Warcraft: Humanos vs Orcos© es un juego de estrategia donde dos especies combaten entre sí: los humanos y los orcos. Cada Especie dispone de los mismos tipos de unidades militares, aunque particularizada para cada especie. Para representar en memoria estos personajes se ha decidido utilizar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



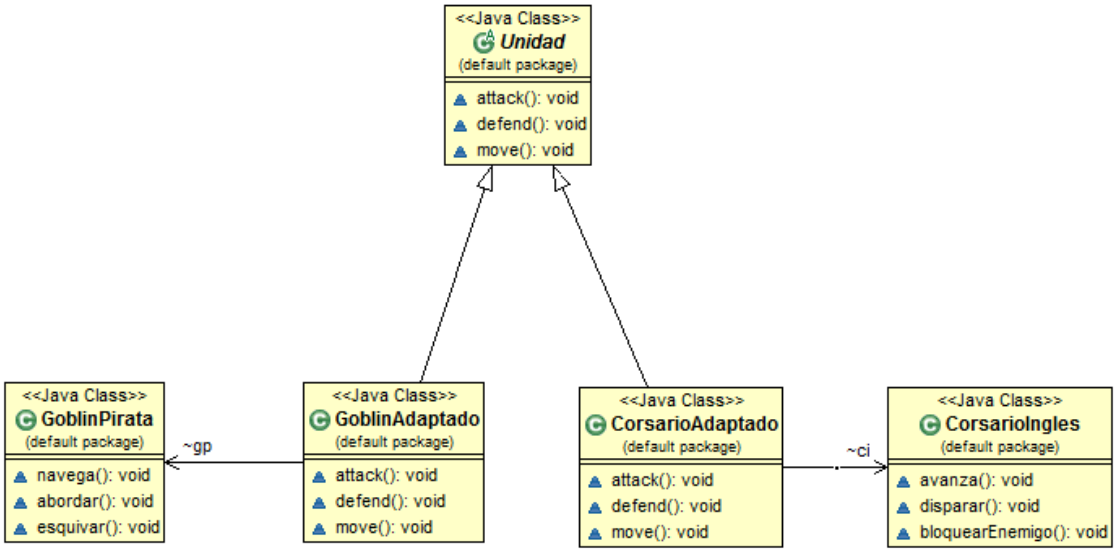
En esta ocasión queremos incorporar un nuevo tipo de unidad, los piratas. Para ello nos ofrecen dos clases provenientes de otro juego: **GoblinPirata** y **CorsarioIngles**.

Se pide:

1º/ **(1 punto) Selecciona y justifica** el patrón más adecuado para poder utilizar ambos tipos de piratas como otras unidades más.

Debemos utilizar el patrón estructural Adapter, ya que nos permite unificar la interfaz de ambos piratas como si fueran otra tipo de unidad (o subclase de Unidad).

2º/ **(2 punto)** Representa y describe como quedarían la jerarquía e implementa sus métodos al utilizar el patrón propuesto en el ejercicio anterior.




```
class GoblinAdaptado extends Unidad {
      GoblinPirata gp = new
GoblinPirata();
      void attack(){gp.abordar();}
      void defend(){gp.esquivar();}
      void move(){gp.navega();}
}
```

```
class CorsarioAdaptado extends Unidad{
      CorsarioIngles ci = new
CorsarioIngles();
      void attack(){ci.disparar();}
      void defend(){ci.bloquearEnemigo();}
      void move(){ci.avanza();}
}
```

3º/ Supongamos que como resultado de los apartados anteriores contamos con dos tipos de unidades más: **CorsarioUnidad** y **GoblinUnidad**; pero que todos los piratas ejecutan la misma secuencia de acciones en el juego:

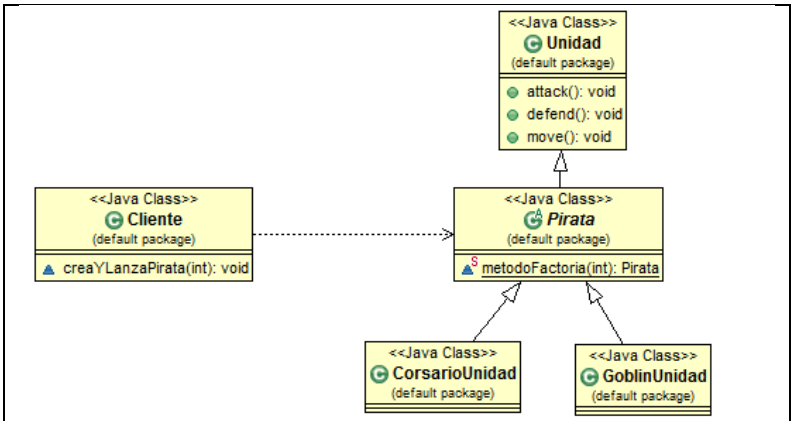
```
Class Cliente{

void creaYLanzaPirata(int tipo){
            if(tipo == 1){
                  CorsarioUnidad p = new CorsarioUnidad();
                  p.move();
                  p.move();
                  p.attack();
                  p.defend();
            }
            if(tipo == 2){
                  GoblinUnidad p = new GoblinUnidad();
                  p.move();
                  p.move();
                  p.attack();
                  p.defend();
            }
      }
}
```

Se pide:

i. **(2 puntos)** Justifica porqué los patrones creacionales FactoryMethod o AbstractFactory nos evitarían duplicar el código para cada subtipo de pirata actual o futuro que se cree (siempre y cuando todos los tipos de piratas ejecuten las mismas acciones). Justifica además cuál de estos dos patrones sería el más adecuado.

Estos patrones nos permiten separar la selección o creación de un pirata, de cómo se manipula. El patrón propone la creación de un método que en función de un tipo nos cree un objeto de una clase u otra. La aplicación cliente, en este caso el método creaYLanzaPirata(), se limitará a manipular un "pirata genérico" sin preocuparse del tipo concreto en cada caso.
El patrón a utilizar sería el FactoryMethod, ya que sólo hay un tipo de producto a crear "pirata", así que no hay necesidad de un AbstractFactory que no permitiría crear varios tipos de productos (una familia) de forma coordinada, por ejemplo, cada tipo de pirata y su tipo de barco correspondiente: bucanero y barco_bucanero, corsario y buque_corsario, etc.

ii. **(3,5 puntos)** Utilizando el patrón escogido en el apartado anterior representa como quedaría la nueva jerarquía de clases y sus métodos, considerando los dos tipos actuales de piratas: **CorsarioUnidad** y **GoblinUnidad**. Modifica adecuadamente el método `creaYLanzaPirata()`



La clase abstracta Pirata representa a cualquier pirata presente o futuro. Además incorpora como método estático la operación metodoFactoria(int) que en función del tipo de pirata devuelve un objeto de una clase u otra.

```
// esta clase representa a todos los tipos de piratas
public abstract class Pirata extends Unidad {

	// Metodo factoria que en funcion del tipo devuelve
	// un nuevo objeto pirata "concreto" solicitado
	static Pirata metodoFactoria(int tipo) {
		switch(tipo){
		case 1: return new CorsarioUnidad();
		case 2: return new GoblinUnidad();
		default: return null;
		}
	}
}
```

`void creaYLanzaPirata(int tipo){`
`  // se crea el tipo de pirata escogido`
`  Pirata p = Pirata.metodoFactoria(tipo);`
`  // acciones comunes a todo pirata`
`  p.move();`
`  p.move();`
`  p.attack();`
`  p.defend();`
`}`
	El método ahora define un objeto de tipo pirata, y se apoya en el método factoría para crear el objeto concreto.
	Independientemente de su tipo concreto manipula al objeto por medio de la variable p de tipo ***Pirata****.*

iii. **(1,5 puntos)** Indica que cambios habría que introducir para definir una nueva unidad pirata denominada **Grumete**. y que además pueden aparecer nuevos subtipos de piratas (ej: Bucanero, grumete,…)

Se crearía una clase ***Grumete*** *que hereda de* ***Pirata*** *y se modificaría el método factoría indicando que si el tipo es 3 habría que crear un objeto* ***Grumete****.*

```
// esta clase representa a todos los tipos de piratas
public abstract class Pirata extends Unidad {

	// Metodo factoria que en funcion del tipo devuelve
	// un nuevo objeto pirata "concreto" solicitado
	static Pirata metodoFactoria(int tipo) {
		switch(tipo){
		case 1: return new CorsarioUnidad();
		case 2: return new GoblinUnidad();

		case 3: return new Grumete(); // Grumete

		default: return null;
		}
	}
}
```