Se dispone del código fuente de un prototipo de videojuego basado en la franquicia Pokemon. Tal y como se puede ver en la figura 1, en este juego se establecen combates entre las criaturas (clase Pokemon) de diferentes entrenadores (clase EntrenadorPokemon). Cada entrenador dispone de un equipo formado por 2 pokemons. En el prototipo actual los equipos están formados siempre por un Pikachu y un Charmander. En el anexo 1 se pueden ver el código de estas clases.

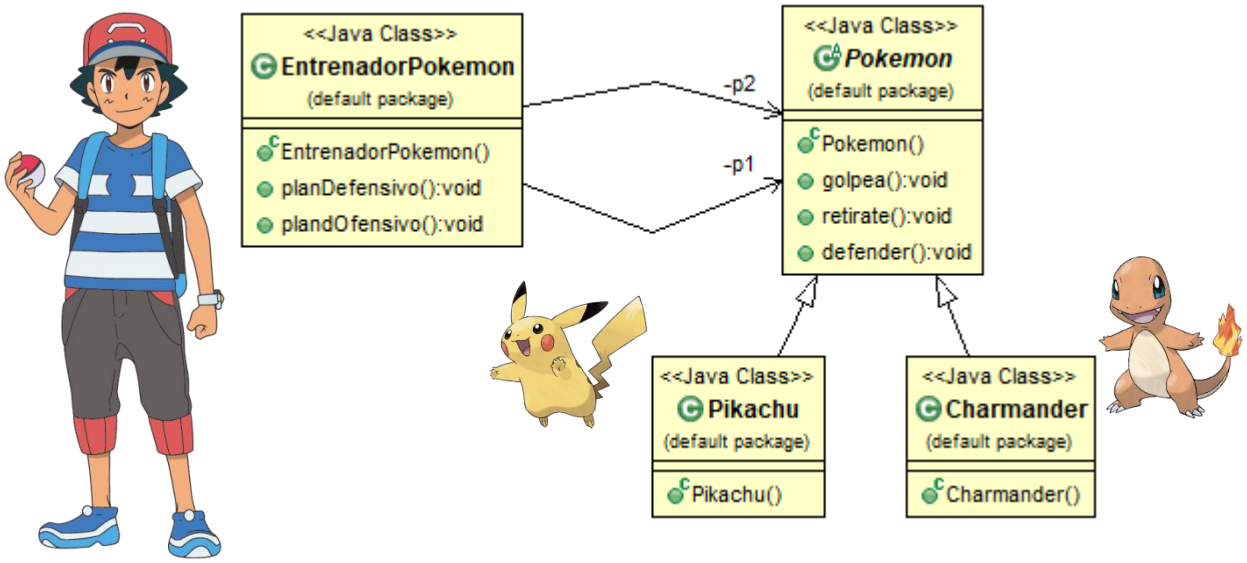


Figura 1: Clases disponibles en el prototipo inicial.

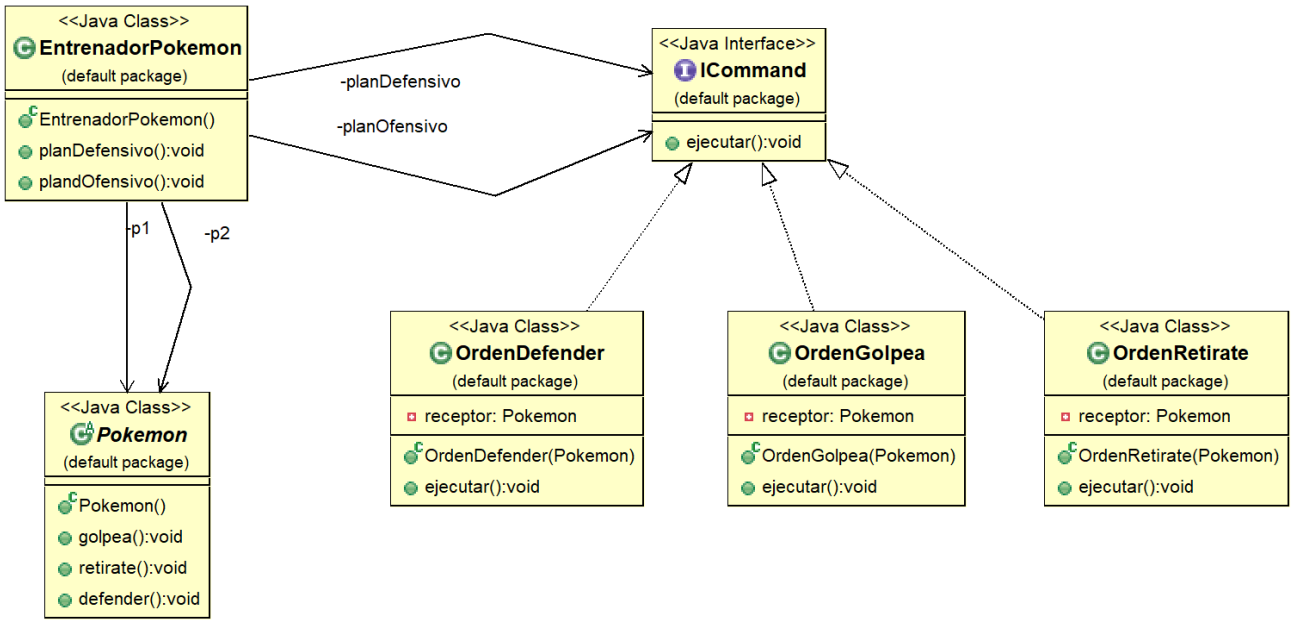
A la vista de los métodos planDefensivo() y planOfensivo() de la clase EntrenadorPokemon() los autores del juego se dan cuenta que para modificar los comportamientos de los pokemons es necesario reescribir una y otra vez estos métodos y volver a compilar el programa. Nos plantean al equipo de programación si no habría algún patrón de diseño que nos facilitara modificar y/o reprogramar los comportamientos de los Pokemons de un equipo sin tener que editar el código fuente y recompilar cada vez. Es decir que en tiempo de ejecución se pudiera incluso (por parte del usuario) programar los comportamientos de los pokemons: indicando que secuencia de acciones realizar y que pokemon del equipo debe realizarlas.

En este caso vamos a necesitar un patrón de comportamiento de tipo Command. Ya que en los planes de actuación de los entrenadores pokemon pretendemos que cada operación, el orden en que se ejecutan, y el responsable de ejecutarla, puedan ser establecidos en tiempo de ejecución. Es decir, que las acciones y los ejecutores han de poderse manipular como si fueran otro tipo más de dato.

1.- Aplica el patrón Command con el objetivo planteado. Describe qué función tiene cada nueva Interfaz/Clase nueva. Indicando que papel realiza cada una de ellas (ej: Cliente, Receptor, ICommand, Orden,…). Dibuja el nuevo diagrama de clases UML e implementa las operaciones de cada una de ellas.

Actualiza la definición de la clase **EntrenadorPokemon del Anexo2** reimplementando planDefensivo() y planOfensivo() para reproducir exactamente las mismas acciones y por los mismos pokemons que en la versión original: **[3 puntos]**

*En este caso los roles que se asumen en el patrón Command, son **EntrenadorPokemon** actuando como Cliente, y **Pokemon** actuando como Receptor. Cada uno de los métodos de **Pokemon** dan lugar a una nueva clase de Orden que implementa la interface **ICommand**: **OrdenGolpea**, **OrdenRetirate**, **OrdenDefender***



*El código de las clases que implementan la interface **ICommand** es el siguiente:*

```
public class OrdenDefender implements ICommand {

	// receptor que ejecuta la acción
	private Pokemon receptor;

	public OrdenDefender(Pokemon p) {
		this.receptor = p;
	}

	@Override
	public void ejecutar() {
		receptor.defender();
	}

}
```

```java
public class OrdenGolpea implements ICommand {

	// receptor que ejecuta la acción
	private Pokemon receptor;

	public OrdenGolpea(Pokemon p) {
		this.receptor = p;
	}

	@Override
	public void ejecutar() {
		receptor.golpea();
	}

}
```

```java
public class OrdenDefender implements ICommand {

	// receptor que ejecuta la acción
	private Pokemon receptor;

	public OrdenDefender(Pokemon p) {
		this.receptor = p;
	}

	@Override
	public void ejecutar() {
		receptor.defender();
	}

}
```

*Ahora modificamos la clase EntrenadorPokemon para reconstruir el **planDefensivo** y el **planOfensivo**, en este caso utilizamos un array estático de Java, pero podría ser cualquier otra estructura lineal:*

```java
public class EntrenadorPokemon {
	// pokemons del entrenador
	private Pokemon p1;
	private Pokemon p2;

	// atributo que contendrá el plan defensivo
	private ICommand planDefensivo[];

	// atributo que contendrá el plan ofensivo
	private ICommand planOfensivo[];

	// constructores y resto de métodos...
}
```

Definimos ahora en el constructor los elementos de cada plan:

```
	public EntrenadorPokemon(){

		// creamos los pokemons
		p1 = new Pikachu();
		p2 = new Charmander();

		// creamos el plan defensivo
		planDefensivo = new ICommand[2]; // son dos acciones...
		planDefensivo[0] = new OrdenDefender(p1); // 1- p1 defiende
		planDefensivo[1] = new OrdenDefender(p2); // 2- p2 defiende

		// creamos el plan ofensivo
		planOfensivo = new ICommand[4]; // son 4 acciones...
		planOfensivo[0] = new OrdenGolpea(p1); // 1- p1 golpea
		planOfensivo[1] = new OrdenGolpea(p2); // 2- p2 golpea
		planOfensivo[2] = new OrdenGolpea(p1); // 3- p1 golpea
		planOfensivo[3] = new OrdenGolpea(p2); // 4- p2 golpea
	}
```

Por último, reimplementamos los métodos que ejecutan los planes de acción:

```
	public void planDefensivo(){
		// ejecuta el plan defensivo
		for(ICommand o: planDefensivo)
			o.ejecutar();
	}

	public void plandOfensivo(){
		// ejecuta el plan ofensivo
		for(ICommand o: planOfensivo)
			o.ejecutar();
	}
```

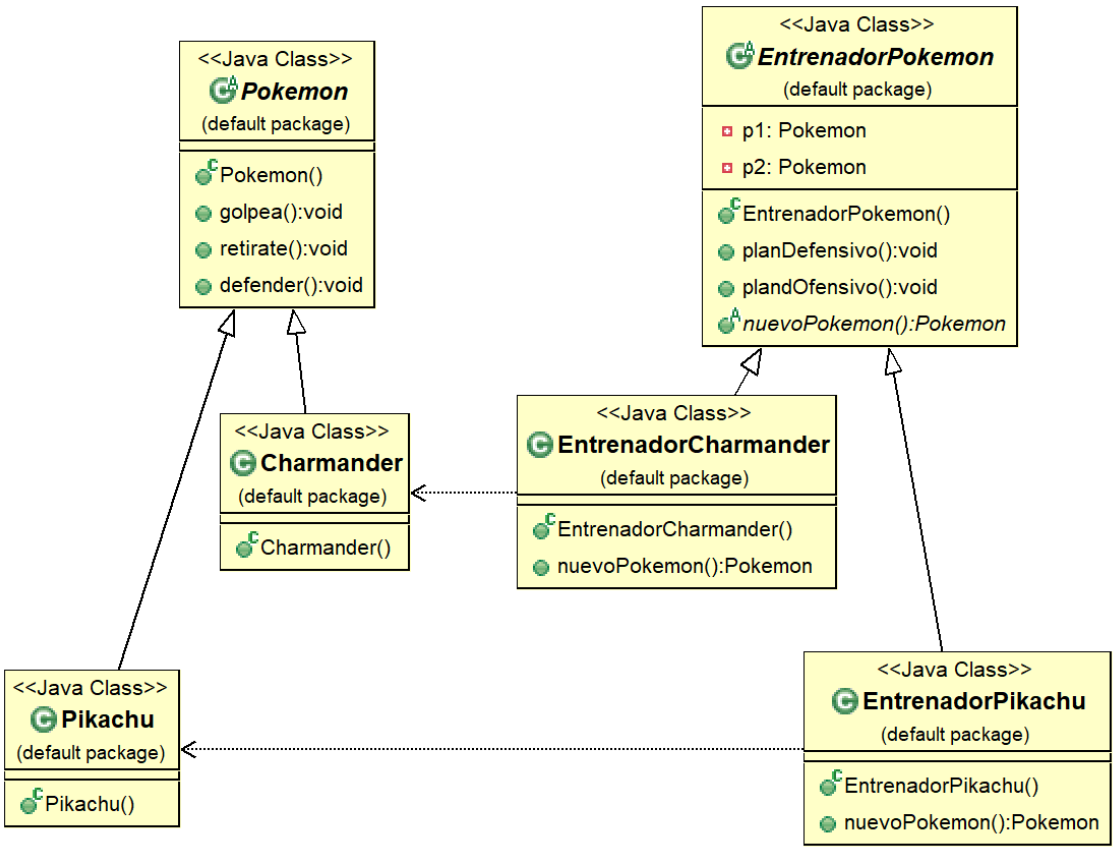
Cada entrenador Pokemon puede gestionar solo un tipo de pokemon concreto. En este caso aparecen las clases **EntrenadorPikachu** (sus pokemons son todos de de tipo **Pikachu**) y **EntrenadorCharmander** (sus pokemons son todos de tipo **Charmander**). Sin embargo, los planes tanto defensivos como ofensivos siguen siendo los mismos para todos los tipos de entrenadores. Se busca por tanto un patrón que permita un diseño que facilite la creación de nuevos tipos de entrenadores de forma que compartan los comportamientos pero que cada uno sólo puedan tener un tipo concreto de Pokemon.

2.- Selecciona el patrón más adecuado para esta tarea. **Justifica tu respuesta**. Indica su tipo, y si el patrón utilizado tiene varias versiones justifica cual utilizarías. **[1,5 puntos]**

*En este caso necesitamos un patrón **Creacional** que independice el código que utiliza los pokemons de la creación de los mismos. En concreto necesitamos un patrón **FactoryMethod**, ya que solo hay un tipo de producto: los **Pokemon**. Como se nos indica todos los entrenadores tienen los mismos métodos, aunque cada uno de ellos trabaja con un solo tipo específico de Pokemon (entrenadorPikachu → Pikachu, entrenadorCharmander →Charmander, etc.). Por tanto, necesitamos **la versión básica** de FactoryMethod.*

3.- Aplica el patrón seleccionado del punto 2. Describe qué función tiene cada nueva Interfaz/Clase nueva, e implementa las operaciones de cada una de ellas. Dibuja el nuevo diagrama de clases UML e implementa las operaciones de cada una de ellas. Define con este nuevo diseño las clases **EntrenadorPikachu** y **EntrenadorCharmander**. **[3 puntos]**

*En este caso entrenadorPokemon actua como **Creador** y Pokemon como **Producto**. La jerarquía quedaría por tanto como:*



*La clase **EntrenadorPokemon** en lugar de crear Pokemons con el operador new utiliza un método (abstracto) **nuevoPokemon()**, por tanto la clase será también abstracta:*

```
public abstract class EntrenadorPokemon {
	private Pokemon p1; // primer pokemon de su equipo
	private Pokemon p2; // segundo pokemon de su equipo


	public EntrenadorPokemon(){
	 // Los pokemons se generan con el método factoria
		p1 = nuevoPokemon();
		p2 = nuevoPokemon();
	}

	 // Cada tipo de entrenador definirá el tipo de pokemon que genera y entrena
	public abstract Pokemon nuevoPokemon();

	public void planDefensivo(){
		p1.defender();
		p2.defender();
	}

	public void plandOfensivo(){
		p1.golpea();
		p2.golpea();
		p1.golpea();
		p2.golpea();
	}

}
```

Este método se implementará en cada clase concreta de Entrenador:

```
/**
 * Este tipo de entrenador solo trabaja con Pikachus
 */
public class EntrenadorPikachu extends EntrenadorPokemon {
	/**
	 *  El método factoria solo retorna pokemons del tipo Pikachu
	 *
	 *  @return el nuevo objeto Pikachu creado
	 */
	@Override
	public Pokemon nuevoPokemon() {
		return new Pikachu();
	}
}
```

```
/**
 * Este tipo de entrenador solo trabaja con Charmanders
 */
public class EntrenadorCharmander extends EntrenadorPokemon {

	@Override
	public Pokemon nuevoPokemon() {
		return new Charmander();
	}
}
```

Pikachu es un personaje emblemático y único en la franquicia, así que no se desea que pueda aparecer más de un Pikachu en el juego. Como consecuencia:

i. A pesar de ser un personaje único, su creación debe producirse cuando un entrenador lo utilice por primera vez y no antes.
ii. Este objeto Pikachu debe poder recuperarse desde cualquier punto del código, sin necesidad de pasarlo como argumento en los diferentes métodos donde se utilice.
iii. Siempre que se necesite a Pikachu, se obtendrá el mismo objeto siempre (lo que permitirá por ejemplo que vaya acumulando puntos de experiencia, o que el daño sufrido se mantenga entre las diferentes apariciones en el juego).
iv. Se debe impedir que se puedan crear otros Pokemons de esta clase.

4.- Selecciona el patrón más adecuado para esta tarea. **Justifica tu respuesta** indicando de qué manera se verificarían los puntos **i** al **iv**. Indica su tipo, y si el patrón utilizado tiene varias versiones justifica cual utilizarías. **[2,5 puntos]**

*Necesitamos un patrón de tipo **Creacional**. En concreto el patrón **Singleton** sobre la clase **Pikachu** que nos permite cubrir todos los requisitos planteados:*

*i. La inicialización perezosa permite retrasar la creación del único objeto **Pikachu** hasta que se requiera su primer uso.*
*ii. El método **public static Pikachu getInstancia(),** al ser un método de clase, está accesible desde cualquier punto del código simplemente invocando **Pikachu.getInstancia();***
*iii. Efectivamente nunca se generan objetos **Pikachu** con el operador new, en su lugar se recupera siempre la misma instancia estática privada por medio del método estático **getInstancia()**.*
*iv. Para ello el patrón fuerza a que todos los constructores de la clase **Pikachu** sean privados, con lo que solo están visibles desde los métodos de la propia clase, en nuestro caso el método **getInstancia()**.*

Anexo I: Código Fuente

```java
public abstract class Pokemon {
	public void golpea(){
		// golpea a algún pokemon enemigo a tiro
	}
	public void retirate(){
		// vuelve a su pokeball
	}
	public void defender(){
		// el pokemon se concentra para recibir menos daño en el siguiente ataque
	}
}
```

```java
public class Pikachu extends Pokemon {
	// ...
}
```

```java
public class Charmander extends Pokemon {
	// ...
}
```

```java
public class EntrenadorPokemon {
	private Pokemon p1; // primer pokemon de su equipo
	private Pokemon p2; // segundo pokemon de su equipo


	public EntrenadorPokemon(){
		p1 = new Pikachu();
		p2 = new Charmander();
	}

	public void planDefensivo(){
		p1.defender();
		p2.defender();
	}

	public void plandOfensivo(){
		p1.golpea();
		p2.golpea();
		p1.golpea();
		p2.golpea();
	}
}
```

```java
public abstract class Digimon {
	public void hit(){
		// ataca a su enemigo (equivalente a Pokemon.golpea() )
	}
	public void run(){
		// huye del combate (equivalente a Pokemon.retirarse() )
	}
	public void block(){
		// bloquea el siguiente ataque de su enemigo (equivalente a Pokemon.defender() )
	}
}
```

Anexo II: Código Fuente, tras aplicar el patrón Command

```
public class EntrenadorPokemon {
	// pokemons del entrenador
	private Pokemon p1;
	private Pokemon p2;

	// atributo que contendrá el plan defensivo: COMPLETAR

     private ICommand planDefensivo[];



	// atributo que contendrá el plan ofensivo: COMPLETAR

      private ICommand planOfensivo[]


	public Cliente_EntrenadorPokemon(){

		// creamos los pokemons
		p1 = new Pikachu();
		p2 = new Charmander();

		 // creamos el plan defensivo

		 planDefensivo =  new ICommand[2]; // son dos acciones...
		 planDefensivo[0] = new OrdenDefender(p1); // 1- p1 defiende
		 planDefensivo[1] = new OrdenDefender(p2); // 2- p2 defiende


		// creamos el plan ofensivo

		 planOfensivo =  new ICommand[4]; // son 4 acciones...
		 planOfensivo[0] = new OrdenGolpea(p1); // 1- p1 golpea
		 planOfensivo[1] = new OrdenGolpea(p2); // 2- p2 golpea
		 planOfensivo[2] = new OrdenGolpea(p1); // 3- p1 golpea
		 planOfensivo[3] = new OrdenGolpea(p2); // 4- p2 golpea
	}

	public void planDefensivo(){
		// ejecuta el plan defensivo: COMPLETAR

		 for(ICommand o: planDefensivo)
			o.ejecutar();
	}

	public void plandOfensivo(){
		// ejecuta el plan ofensivo: COMPLETAR

	  for(ICommand o: planOfensivo)
			o.ejecutar();
	}
}
```