Apellidos:	**Nombre:**
UO:	**Firma:**

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.

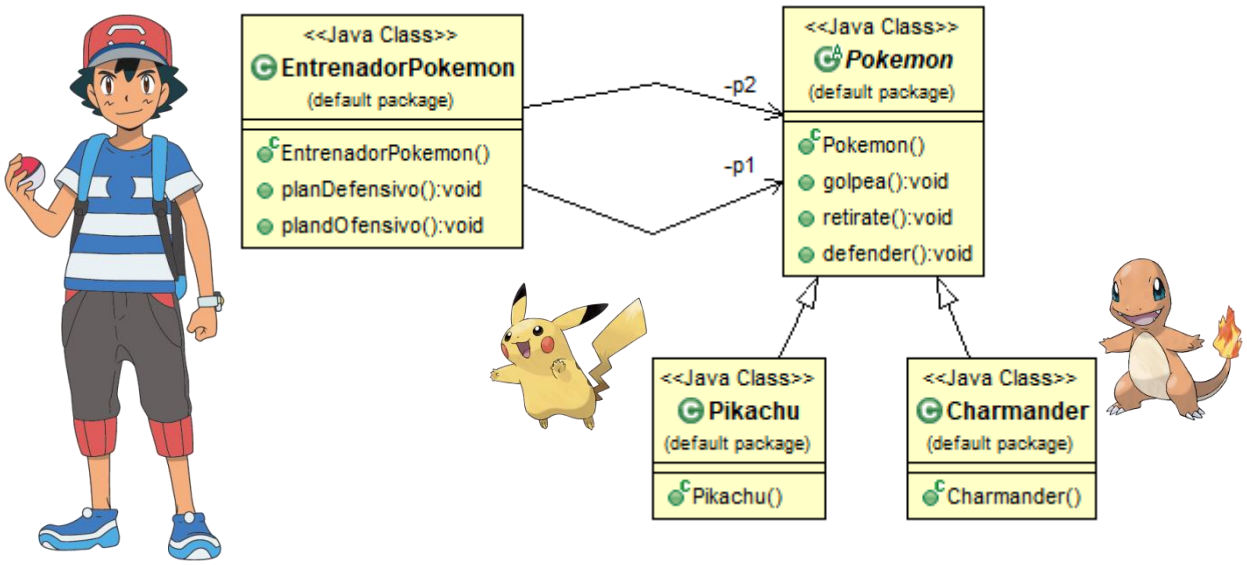


Figura 1: Clases disponibles en el prototipo inicial.

En el juego final pueden participar en realidad cientos de Pokemons distintos, cada uno con su propia subclase. Además se prevé que las ordenes que reciban los pokemons sean independientes de su tipo concreto.

Así que se nos solicita que como diseñadores, que cualquier método del juego que necesite crear Pokemons debería ser diseñado para que su implementación sea independiente de los tipos de Pokemons disponibles en cada versión del juego. Y así por ejemplo, que cada entrenador pudiera recibir en su constructor dos Strings indicando el tipo de los pokemons que desea crear y formar con ellos su equipo. Así para crear un entrenador con un equipo como el del anexo 1 bastaría con:

EntrenadorPokemon ash = new EntrenadorPokemon(“Pikachu”, “Charmander”);

O por ejemplo si creamos las subclases de Pokemon Goldeen y Staryu podríamos definir otro equipo:

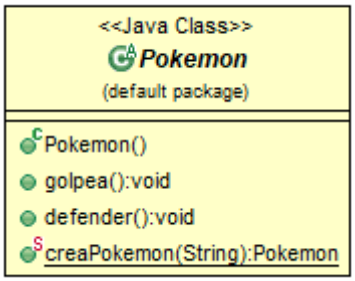
EntrenadorPokemon misty = new EntrenadorPokemon(“Goldeen”, “Staryu”);

1.- 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 la creación de los pokemons de cómo se utilizan a continuación. De forma que el código sea robusto ante cambios en la forma de crear los pokemons o los tipos que se añadan o retiren de los disponibles en el programa. En este caso el patrón más adecuado es el FactoryMethod. Ya que solo hay un tipo de producto (Pokemon) a generar. En concreto cada cliente (entrenador) va a utilizar distintos tipos de este producto, así que la versión más adecuada es la parametrizada.

2.- Aplica el patrón seleccionado del punto 1. Describe qué función tiene cada nueva Interfaz/Clase nueva, e implementa las operaciones de cada una de ellas. Actualiza la definición de la clase **EntrenadorPokemon** para generar ese mismo equipo inicial por defecto: {Pikachu, Charmander} **[2,5 puntos]**

Se crea un método estático Pokemon creaPokemon(String) que actúa como factoría de los pokemons conocidos en cada versión del código, simplemente indicando que subclase de Pokemon crear a partir de su nombre habitual. El único cambio en el cliente (el EntrenadorPokemon) es que en lugar de crear directamente objetos Pokemon, delega la creación en este método estático de la clase Abstracta Pokemon.




```java
public abstract class Pokemon {
//...
	public static Pokemon creaPokemon(String tipo){
		switch(tipo){
		case "pikachu": return new Pikachu();
		case "charmander": return new Charmander();
		default: return null;
		}
	}
}
public class EntrenadorPokemon {
// ...
	public EntrenadorPokemon(){
		p1 = Pokemon.creaPokemon("pikachu");
		p2 = Pokemon.creaPokemon("charmander");
	}
// ...
}
```

Se nos ofrece la posibilidad de incorporar las criaturas de otra franquicia a nuestro videojuego: Digimon. La jerarquía de criaturas de Digimon sigue el diagrama de la figura 2. Como podemos ver por cada Digimon existe una subclase de Digimon. En el diagrama solo aparecen 2 de las decenas de digimons que existen. Además nos indican que nos permiten utilizar las clases Digimon, pero **no modificarlas**.

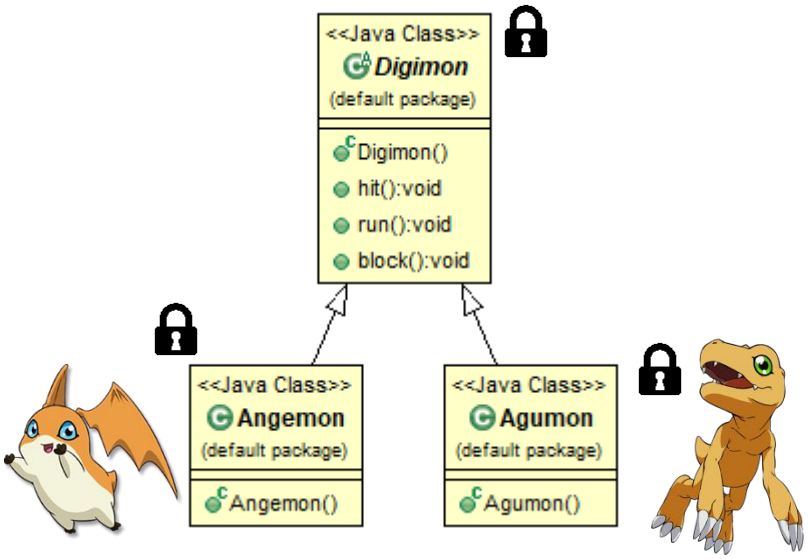


Figura2: Jerarquía de clases de las criaturas Digimon. Angemon y Augumon son solo 2 de las decenas de Digimons existentes en la franquicia.

Se nos solicita aplicar el patrón más adecuado para poder utilizar en nuestro videojuego tanto criaturas Pokemon como Digimon. Así, todos los Digimons deben poder ser manipulados como si fueran nuevos tipos de Pokemons. Además se requiere que, la incorporación de cada nuevo tipo de Digimon **no suponga la creación de nuevas clases**, sino que la solución aportada permita utilizar como Pokemons a cualquier futura criatura que herede directa o indirectamente de la clase Digimon.

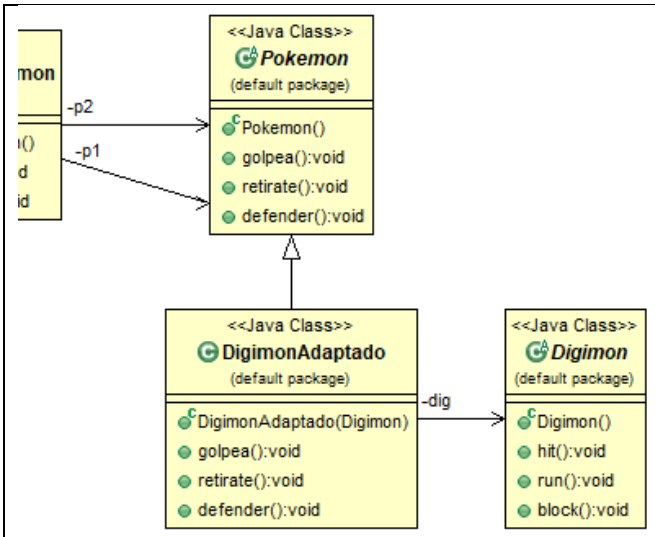
3.- Selecciona el patrón más adecuado para esta tarea. **Justifica tu respuesta**. Indica su tipo, y si el patrón utilizado tiene varias versiones justifica cual utilizarías. **[1,5 puntos]**

En este caso se requiere de un patrón estructural que nos permita manejar los Digimon como si fueran pokemon. Lo que necesitamos es un patrón Adapter, concretamente en su versión de Objetos para poder con una única clase adaptar todas las subclases de Digimon. Reinterpretando las operaciones de Pokemon como ordenes equivalentes de la clase Digimon.

4.- Aplica el patrón seleccionado del punto 5. Describe qué función tiene cada nueva Interfaz/Clase nueva, e implementa las operaciones de cada una de ellas. Actualiza la definición de la clase **EntrenadorPokemon** para generar el siguiente equipo inicial de "pokemons": un Angemon y un Agumon **[3 puntos]**

Crearemos una nueva clase que, apoyándose en la composición, permita adaptar cualquier subtipo de Digimon. Para ello creamos la clase DigimonAdaptado que hereda de Pokemon y utiliza por composición un objeto Digimon. Para permitir adaptar cualquier subtipo de Digimon será el constructor el que reciba como argumento el objeto Digimon a ser adaptado. Gracias al polimorfismo el argumento de tipo Digimon acepta cualquier objeto de sus subclases directas o no. Simplemente resta reescribir las operaciones de Pokemon en función de los métodos del objeto Digimon usado por composición:

- Pokemon.golpea() →Digimon.hit()
- Pokemon.retirate()→Digimon.run()
- Pokemon.defender()→Digimon.block()




```java
public class DigimonAdaptado extends Pokemon {
	private Digimon dig;

	public DigimonAdaptado(Digimon _dig){
		dig = _dig;
	}

	@Override
	public void golpea() {
		dig.hit();
	}

	@Override
	public void retirate() {
		dig.run();
	}

	@Override
	public void defender() {
		dig.block();
	}
}
```

Se reescribe además el constructor de EntrenadorPokemon para que cree el equipo con dos Digimons adaptados:

```java
public class EntrenadorPokemon {
	// ...
public EntrenadorPokemon(){
	p1 = new DigimonAdaptado(new Agumon());
	p2 = new DigimonAdaptado(new Angemon());
}
// ...

}
```

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 debe realizarlas.

5.- 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. **[2 puntos]**

En este caso necesitaríamos 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.