# Trabajo Práctico 1 Programación III

## Introduccion

En el presente trabajo se llevó a cabo la construcción de un juego tipo “Gimnasio Pokémon” en el cual se realiza un torneo de 16 participantes (“entrenadores”) donde cada uno presenta su lista de Pokemones a competir. El torneo consta de eliminaciones por rondas hasta llegar a la Final, donde se elige al campeón del mismo. Cada entrenador por su parte puede presentar una carta (con un máximo de dos) que afectará al Pokémon del entrenador contrario. Existen 6 tipos de Pokemones en este juego, los cuales son: Tierra, Fuego, Electricidad, Agua, Psíquico y Hielo. Cada tipo de Pokémon tiene distintas estadísticas y distinta forma de realizar sus ataques. Por cada ronda finalizada los Pokemones reinician sus estadísticas y los ganadores reciben un “premio”, por lo que ven aumentadas sus estadísticas un 10%. Se van informando los acontecimientos de cada ronda por pantalla, así como el ganador de la misma.

## Conceptos Teóricos utilizados

Para la realización del trabajo se fueron utilizando distintos conceptos teóricos de la Programación Orientada a Objetos que iremos detallando paso por paso a continuación.

### Patron Singleton

La clase Torneo , dentro del paquete modelo, se decidió crearla como un Singleton ya que se necesitaba instanciarla una sola vez. Durante toda la ejecución del programa no puede haber más de una instancia de Torneo , ya que se lo considero como una clase donde sucederán las invocaciones de las rondas y los enfrentamientos. Sería incorrecto que hubiera más de un torneo simultáneamente.

### Herencia de clases

El programa cuenta con una clase abstracta principal llamada Pokemon para el diseño general de los Pokemones, y luego cada tipo específico de Pokemon hereda de ella sus métodos y atributos generales comunes a todos los Pokemones. Dependiendo el tipo de Pokémon, cada uno utilizara el método de la clase madre o lo sobreescribirá.

## Interfaces

Se crearon varias interfaces, Atacable, Clasificable, Hechizable e ICarta. Las últimas dos se crearon para la funcionalidad de “Hechizar” a un Pokemon, la cual está explicada en el área de Double Dispatch. La interface Atacable le otorga la posibilidad a la clase Pokemon, que es la que la implementa, de atacar (a otro atacable) , recibir daño y obtener su numero de ataque. Gracias a esta interface y haciendo uso del Polimorfismo se puede llamar al método atacar(Atacable) y llamar al metodo recibirDano() de ese Atacable, sin preguntar tipos de pokémon ni otras variables que interfieren en el proceso de ataque, ya que el propio Pokemon que reciba el daño es el que se ocupara de esto.

La interface Clasificable otorga el método calculaClasificacion, que es utilizado tanto por Pokemones como Entrenadores para calcular su puntaje al final del torneo.

## Double Dispatch

Se decidió utilizar doble envío para la mecánica de Cartas y Hechizos, ya que esta depende tanto de la clase que esté enviando el hechizo ( Niebla, Viento o Tormenta) como el tipo de pokémon que la reciba, ya que en base a cada tipo se realizara un efecto diferente.

Los hechizables tienen el método serHechizado pasando por parámetro un objeto de tipo ICarta. Este método llamará al método hechizar de la carta correspondiente pasándole como parámetro el mismo como Hechizable.

Las clases Viento, Niebla y Tormenta, que se encuentran dentro del paquete “hechizo” utilizan la interfaz ICarta, la cual les proporciona para sobreescribir el método hechizar(). Este método requiere como parámetro un objeto de tipo Hechizable.

Luego, el hechizable se hechiza  a sí mismo, invocando dentro de su clase correspondiente, el método hechizar invocado por la carta acorde. Dentro de la clase abstracta Pokemon se decidió crear los 3 métodos de hechizo correspondientes a cada carta, pero algunos pokemons hijos sobreescriben este método, cambiando los efectos.

**Excepciones:** Se utilizaron exceptions para validar los datos de cada entrenador:

**DatoInvalidoException:** Esta clase se creó pensando en un posible uso a futuro donde se tengan que validar distintos datos ingresados utilizando una interfaz gráfica.

**PokemonInvalidoException:** Esta exception se encarga de validar el tipo del pokemon/es ingresado/s para cada entrenador.

**CantidadHechizosExcedidosException:** Esta última se lanzará cuando un entrenador intente lanzar un hechizo y el atributo cantidadHechizos este en 0, devolviendo una carta “null” que indica que no se usará una carta de hechizo por parte de ese entrenador en el combate.

## Patrón Factory:

Se utilizó el patrón factory para hacer la instanciación de los pokemones (no la clase en sí, sino alguna de sus 6 subclases ). Se utilizó el polimorfismo de sobrecarga de métodos para esta clase, ya que el Pokémon de tipo Hielo tenía la posibilidad de tener “gran recarga” y había que pasarla por parámetro, asique al invocar al método con este parámetro extra, se sabría que el Pokémon sería de tipo Hielo, o en su defecto, invalido.

## Metodo Clone

Tanto la clase Pokemon como la clase Entrenador tiene la capacidad de ser clonados, aunque no se utilizan en este proyecto, los métodos están sobreescritos.

## Patron Template

La clase abstracta Pokemon posee un método atacar() , en el cual se efectúan una serie de “pasos” de ataque de los Pokemones. GolpeInicial, Recarga, y GolpeFinal. Esta secuencia siempre se ejecuta por todos los pokemones cuando atacan. Inicialmente, se pensó en aplicar un método Hook , ya que se había diseñado un modelo donde hubiera clases hijas de pokemon que no poseyeran Recarga, luego se cambio a que todas las clases hijas de Pokemon tuvieran la facultad de hacer Recarga, pero dependiendo de cada instancia y del parámetro pasado en el constructor. Por lo tanto el método Recarga, ahora chequea si posee el atributo Recarga = True para ejecutarse, y sino no hace nada.

GolpeFinal es sobreescrito por cada clase Hija de pokemon, ya que cada tipo de pokemon hace un ataque distinto.

## Sobre el TP en general

| Tipo | Golpe Inicial | Recarga | Golpe final | Recibe daño |
|---|---|---|---|---|
| Fuego | Todos los tipos realizan el mismo ataque inicial, infligiendo al adversario un daño igual a su fuerza de ataque, luego su fuerza de ataque se reduce a la mitad (cansancio) | Incrementa un 10% la fuerza y la vitalidad. | Provoca al adversario un daño igual a su fuerza más un 25% y luego la fuerza se agota por completo (queda en cero) | El escudo y la vitalidad absorben la mitad del daño cada uno (decrementándose) |
| Agua | “” | Incrementa un 10% la fuerza y la vitalidad. | Provoca al adversario un daño igual a su fuerza y luego su fuerza se reduce a la Mitad | El escudo absorbe todo el daño, solo cuando éste se agota comienza a decrementarse la vitalidad |
| Hielo | “” | Si el Pokemon posee capacidad de “gran recarga” su fuerza tomará un valor de 400 pero no recargará su vitalidad, si no posee dicha capacidad, tendrá el comportamiento habitual. | Provoca al adversario un daño igual a su fuerza menos un 10% pero conserva la misma fuerza | El escudo absorbe el 75% del daño y la vitalidad el otro 25% |
| Electricidad | “” | Aumenta su ataque en un | Provoca al adversario un | El escudo se desgasta en un 120% del daño |

| | | | | |
|---|---|---|---|---|
| | | 20% | ataque equivalente al 150% de su daño | recibido. Y cuando se acaba la vitalidad recibe el 100% |
| Psiquico | “” | Recupera su vitalidad en un 50% si esta herido, sino, gana un 20% de vitalidad. | Realiza el ataque del adversario al adversario + su ataque. Luego se reduce a un 20% | El escudo absorbe el daño hasta acabarse, luego baja la vida. |
| Tierra | “” | Aumenta su escudo en un 50% | Provoca al adversario un daño equivalente al 100% de su escudo actual + 20% de su daño. Luego el escudo se agota. | El escudo absorbe el 100% del daño recibido y se desgasta en un 60% |

| Tipo | Vitalidad | Escudo | Fuerza |
|---|---|---|---|
| Agua | 500 | 100 | 120 |
| Hielo | 500 | 120 | 100 |
| Fuego | 530 | 200 | 80 |
| Electricidad | 500 | 150 | 110 |
| Psiquico | 700 | 100 | 40 |
| Tierra | 650 | 250 | 40 |

**Descripción de los hechizos:**

Hechizar viento: si el pokemon es de fuego, reduce la efectividad de sus ataques en un 30%. El resto de los pokemones afectados por este hechizo, tienen una chance del 20% de ver reducida su vitalidad en 15%

Hechizar niebla: Todos los pokemones ven reducida su visibilidad, 20% de chance de fallar su ataque menos el psíquico.

Hechizar tormenta: Todos los pokemones ven reducida su vitalidad en un 20% debido a descargas eléctricas. El pokémon de electricidad se ve inmune a este hechizo.

**Enfrentamientos:**

Cada batalla entre 2 entrenadores se obtiene sacando cada entrenador de una lista de entrenadores “clasificados”. Tras la batalla, el perdedor es removido de esta lista.

Una vez en el combate se define mediante una variable aleatoria quién será el entrenador que ataque primero así como también que pokemon van a usar en la batalla.

Paso siguiente cada entrenador elige una carta para hechizar al pokemon de su adversario, siendo la carta “null” la opción en donde no se elige una carta de entre las nombradas en la consigna. Esta selección también se da de forma aleatoria. Si el entrenador usó una carta, se resta de la cantidadHechizos del entrenador.

Una vez que ambos pokemones atacaron se calculan sus puntajes en base a sus atributos para definir el pokemon ganador.

El cálculo para el puntaje es :

Vitalidad del Pokemon * 0.5 + Ataque del Pokemon +0.9 + Escudo del Pokemon *0.4

Una vez definido el ganador se reparten los puntos de experiencia y se premia al entrenador victorioso.

## Premios:

Cada vez que un entrenador gana un combate obtendrá una bonificación para todos sus pokémons, esto consta no solo de recargar sus atributos a su valor base sino además recibir una bonificación del 10% en todos sus atributos (esta bonificación es acumulable entre combates).

# Segunda parte

## Introduccion:

En esta segunda parte del trabajo práctico se continua con todo lo generado anteriormente pero con la diferencia de que el proyecto se adapta al patrón mvc, observer,state,persistencia y concurrencia, que forman parte de los temas explicados en la teoría durante la parte final del cuatrimestre. De esta forma se permite importar paquetes de entrenadores y fases del torneo, gracias a la persistencia, se permite ejecutar varios enfrentamientos simultáneamente, gracias a la concurrencia y el patrón observer, y con el MVC se obtuvo una Ventana Gráfica que representa lo sucedido en el torneo pero sin que el propio modelo se vea afectado, usando un controlador.

## Implementacion General:

En esta segunda parte se optó por modificar el esquema del torneo, a diferencia de la parte anterior que se arrancaba directo desde una fase eliminatoria, se implementó un sistema de grupos. Lo cual resulto ser mas dificultoso de lo que se esperaba, porque hubo que agregar mucho cambio tanto en el diseño del torneo como en la interfaz gráfica. Se pueden agregar 8,16 o 32 entrenadores. Se generar grupos acorde de 4 integrantes, los cuales competirán entre sí. Los primeros 2 de cada grupo pasarán a la fase siguiente eliminatoria.

Para implementar este nuevo esquema se creó una clase llamada Grupo, que implementa serializable. El torneo posee un arrayList con la cantidad de grupos. Los grupos tienen la potestad de “entregar” un enfrentamiento adecuado entre sus integrantes, hasta que todos hayan competido.

A su vez, se sobrescribe el método compareTo en la clase Entrenador, para poder comparar los entrenadores en base a su puntaje, la clase Grupo hace uso de esta comparación y ordena los integrantes en base al puntaje. La idea original era usar el metodo Sort de *Collections*, sin embargo no funcionó como era debido.

Está también implementada la opción de importar y exportar fases, sin embargo la fase de grupos no se importa correctamente a diferencia del resto.

## **Persistencia**:

Para la persistencia se optó por usar una serialización binaria debido a la cantidad de clases que posee el proyecto y a la facilidad que otorgaba para importar y/o exportar una fase del torneo, además se pueden importar y/o exportar entrenadores para que la carga de estos sea más fácil y rápida.

Se implementa la interfaz Serializable en todas las clases y subclases del proyecto, particularmente en Torneo, Entrenadores, Pokemones ( y sus hijos), Cartas, Enfrentamientos y Arenas,Grupo.Estas son las clases que se utilizan para registrar los cambios durante el torneo e ir informando ganadores, perdedores, experiencias, batallas, fases de Grupos, etc. Ademas, se utiliza la interfaz IPersistencia.

Se optó por esta opción y no por el camino de los DTO, debido a la cantidad de clases que había que convertir, sumado a los arrayList de Pokemones con todos sus hijos, la cantidad de combinaciones de DTOfromClase y ClaseFromDTO era muy grande.

### **Concurrencia:**

Para la concurrencia se hizo que la clase Enfrentamiento extienda de thread. Cada enfrentamiento posee un recurso compartido que va a ser la clase Arena. En el torneo se decidió que existirían solo 4 arenas para librar los enfrentamientos.

Para la generación de enfrentamientos existen dos tipos de posibilidades dependiendo en la fase que se encuentre el torneo.

En la fase de grupos, la clase Grupo se encarga de crear enfrentamientos para hacer una combinacion justa entre sus 4 integrantes. Por lo tanto, desde torneo, una vez que están cargados los grupos con sus respectivos integrantes, se decidió que sólo se podrá luchar 1 enfrentamiento a la vez por “fecha”, es decir, si hay 4 grupos, habrá 4 enfrentamientos que se estarán ejecutando concurrentemente y peleando por el recurso compartido. El recurso compartido (la arena) se elige aleatoriamente y es muy probable que suceda que dos enfrentamientos se les haya asignado la misma arena y uno de ellos tenga que esperar al método synchronized para comenzar a pelear.

## **Patron observer/observable:**

El patrón observer fue muy útil para esta parte del trabajo. Por un lado, el torneo observa a las arenas, a medida que cada arena pasa por su respectivo estado, esta va notificando su cambio al torneo, de Preliminar, Batalla, Definición y Limpieza. Esto resultó útil al momento de conectar el modelo con el controlador, ya que el modelo

Observa al Torneo y este recibe todos los cambios de fase de las arenas ( que se ejecutan concurrentemente) para mostrar en la vista. Se presentaron algunas dificultades al momento de ejecutar los update sin embargo, ya que, a pesar de ejecutarse concurrentemente, la vista parecía esperar a que terminara la concurrencia e iterar por todos sus estados a toda velocidad. Se optó por poner un tiempo de espera entre cada informe de cambio de estado para subsanar este defecto.

Además, se está haciendo uso del patrón para informar el ganador del torneo.

## Patron State:

La clase Arena implementa el Patron State. El enfrentamiento llamara a la arena y se pondrá en espera si esta está ocupada, ya que posee un método sincronizado. Luego, siempre se ejecutará el método comenzar, que dependiendo el estado en el que este se ejecutara una acción u otra.

Se creó una interfaz de tipo IStateArena la cual implementan PreliminarState,BatallaState,DefinicionState y Limpieza State.

El estado preliminar se ocupa de presentar a los entrenadores y sus pokemones, a su vez, este configura el estado de la arena como BatallaState y llama al método comenzar. Lo mismo hacen el resto de los estados hasta llegar al LimpiezaState. A medida que se avanza de fase, las arenas se reinician al estado Preliminar.

## Patron MVC:

*Vista*: Se utilizo el Windows Builder como asistente para desarrollar las clases del paquete vista. La clase *Ventana* maneja la ventana principal de la aplicación e implementa la interface *IVista.* Esta interfaz posee una cantidad mínima de métodos que serán sobreescritos y posteriormente utilizados desde el controlador para lograr en enlace entre la vista y el modelo. Al iniciar la aplicación, la ventana se muestra con una cantidad de componentes minimos para permitir la alta, baja y modificación de entrenadores y pokemones, una vez pasado esta etapa la ventana se irá actualizando y se eliminaran/agregan otros componentes de manera dinámica.

*Modelo:* En esta parte se adaptó gran parte de la base del código hecho en la primera parte del trabajo práctico para las nuevas funcionalidades. Sin embargo, el modelo es independiente de la vista y del controlador, es decir que al momento de la programación las funciones se consideraron independientes y se eliminaron por lo tanto

la mayoría de los “SystemOut” que eran utilizados previamente.Finalmente, al lograr que el modelo tenga una independencia de la vista,se posibilitó que en un futuro se pudiera cambiar totalmente el contenido del paquete vista sin que el modelo tenga que ser editado.

*Controlador*: El controlador sirve como puente entre la vista y el modelo. Este posee un atributo de tipo Torneo donde se tiene guardada una referencia a este mismo y también posee un atributo de tipo IVista que servirán para lograr una comunicación entre el modelo y la ventana. Ademas la clase Controlador implementa las interfaces ActionListener, KeyListener, MouseListener, estas agregan un conjunto de métodos para el controlador sea capaz de “escuchar” las distintas acciones que ocurren en la ventana.