Trabajo Final (parte 2)
Gimnasio Pokémon

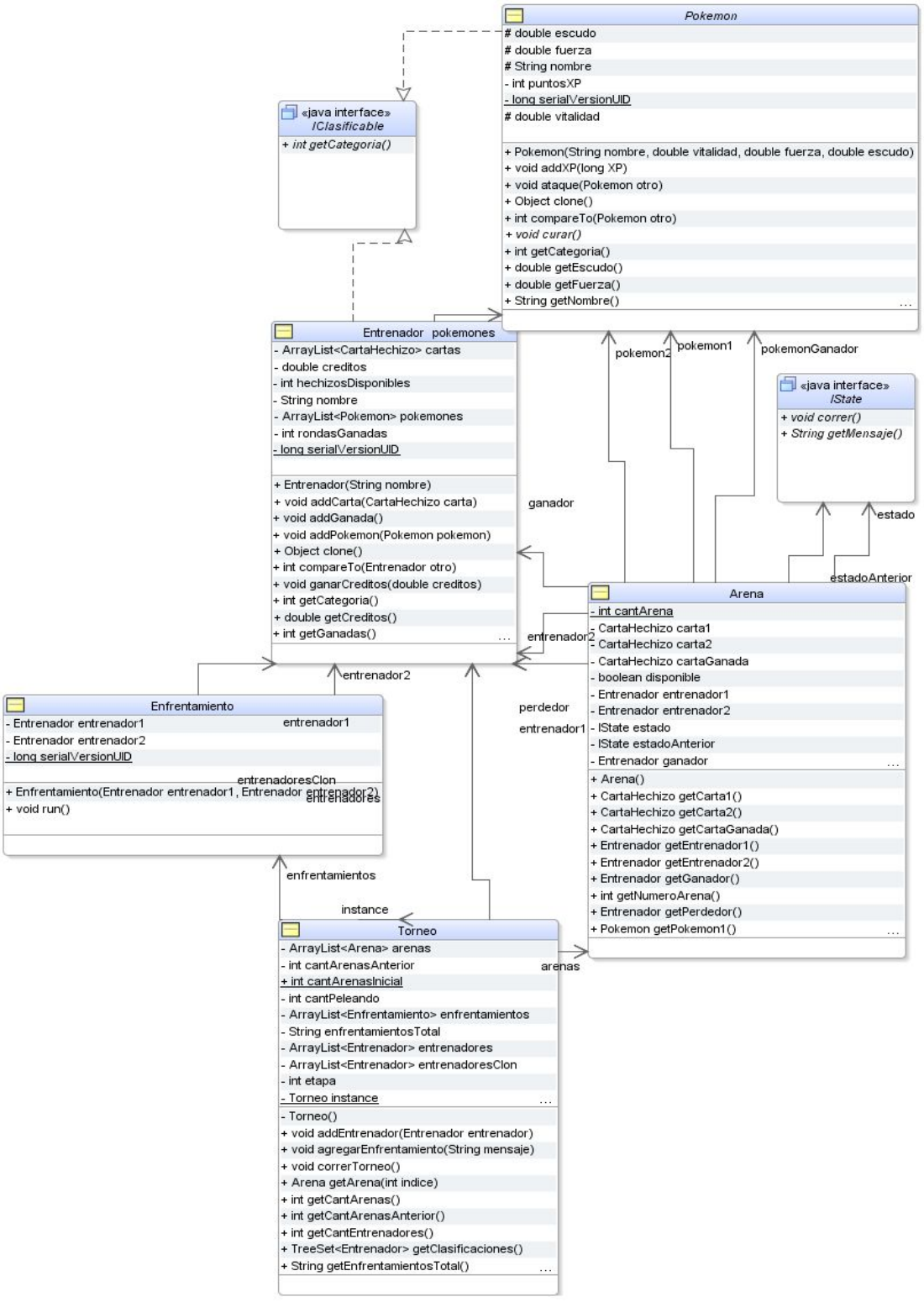
Grupo: 3

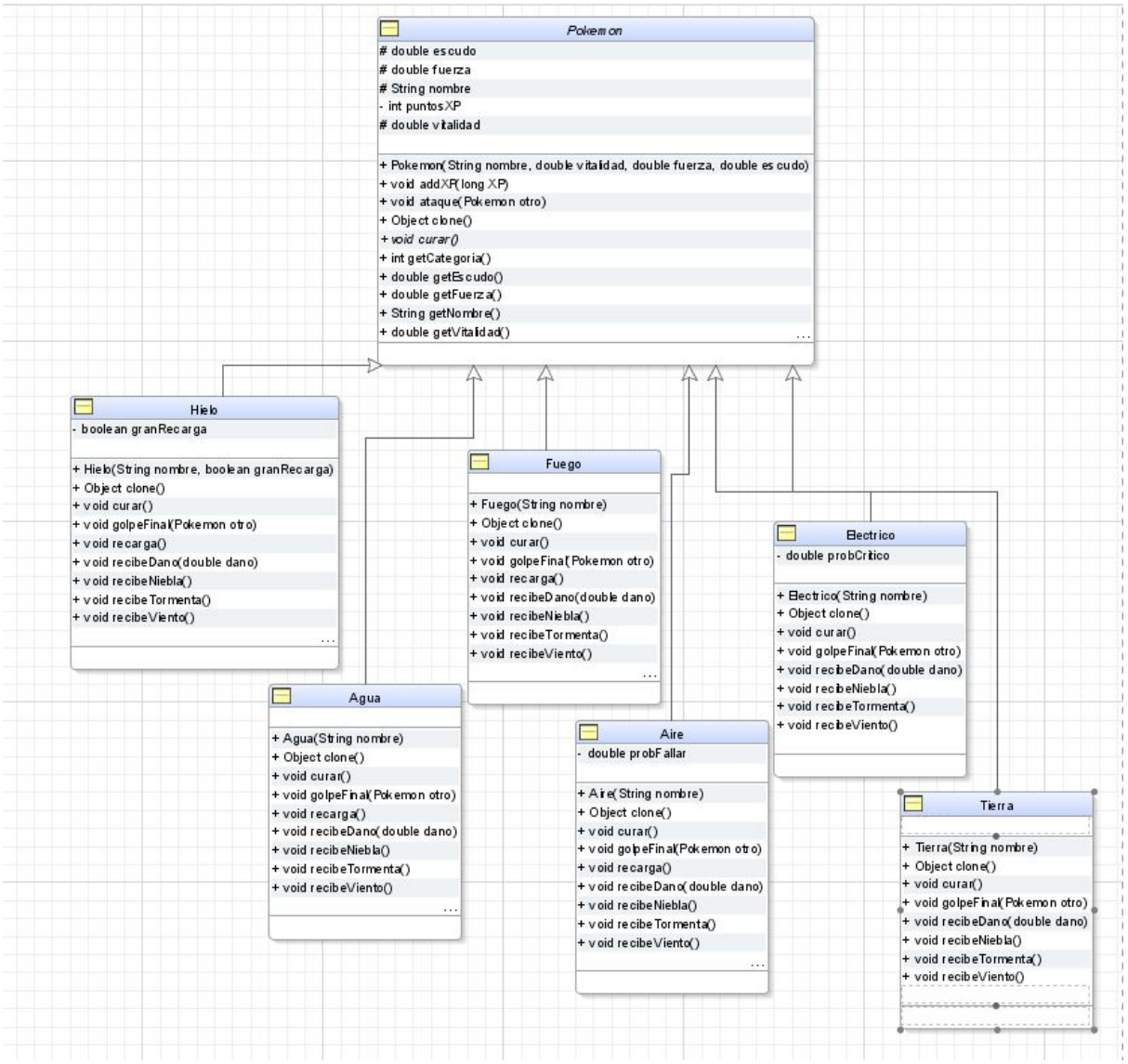
Integrantes: Fidelibus, Gabriel
Izurieta, Luciano
Casamayou, Ignacio

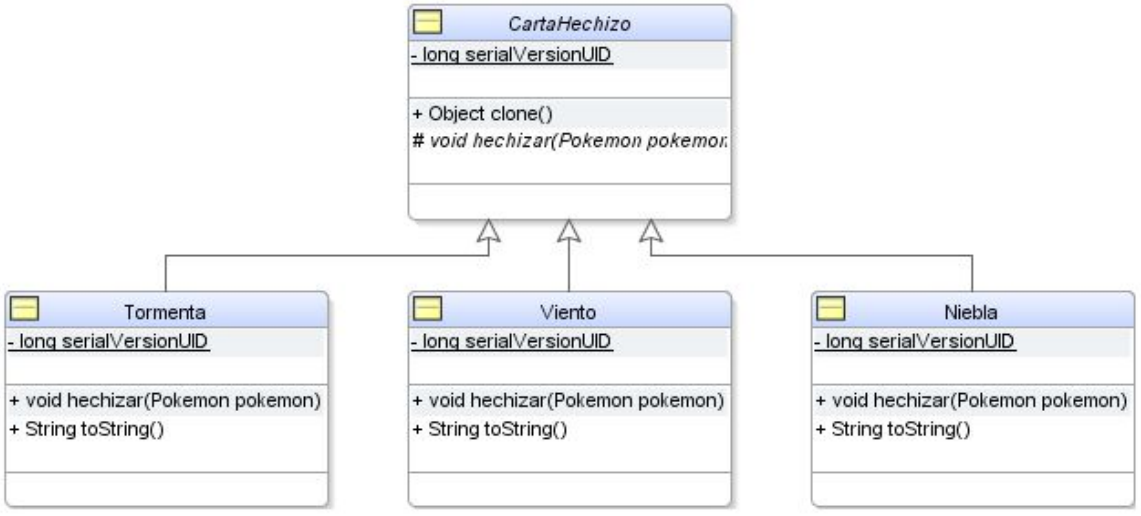
Fecha de entrega: 25/06/2020

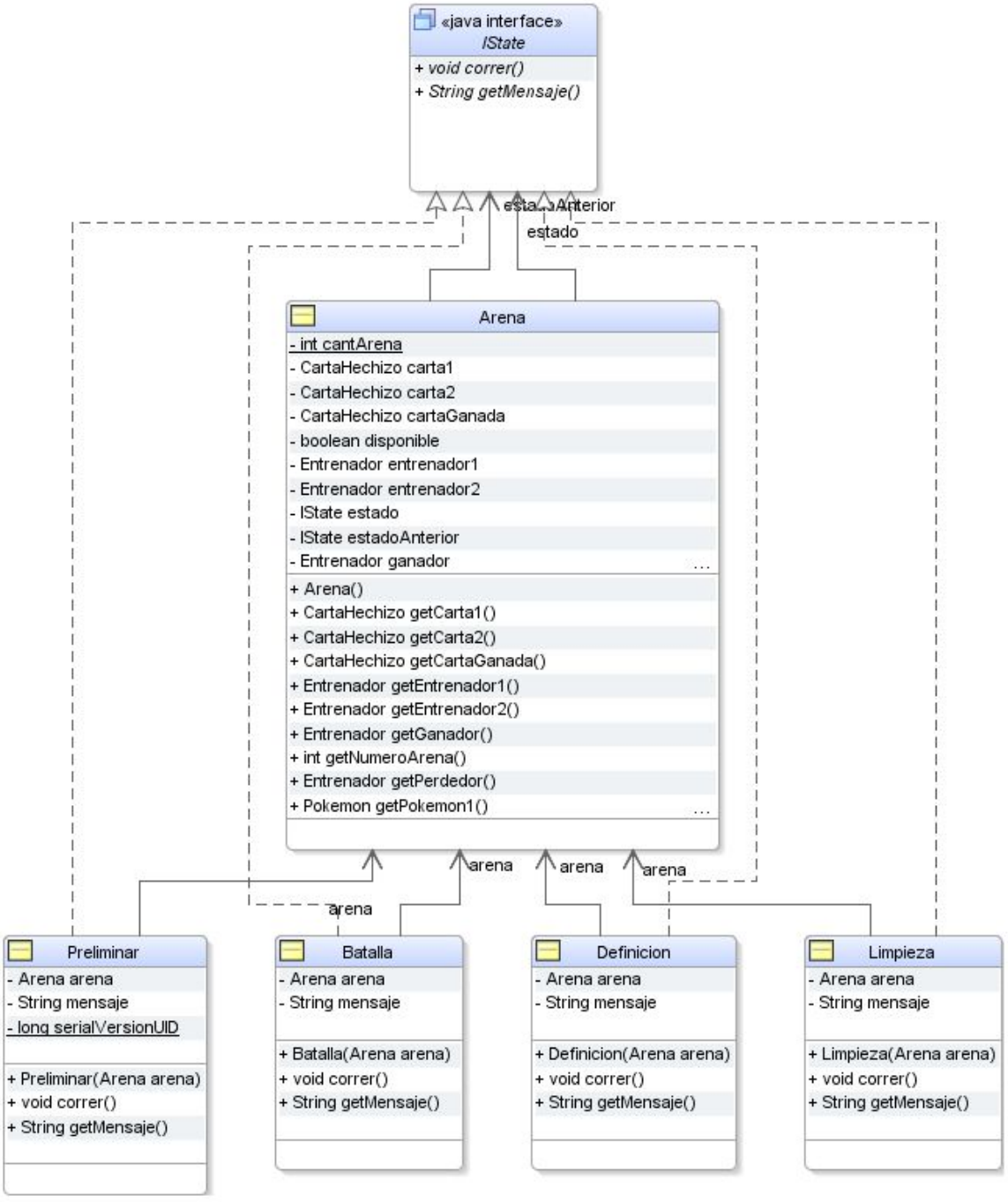
Funcionamiento general del programa

Diagramas UML




Pokemon
double escudo
double fuerza
String nombre
- int puntosXP
double vitalidad
+ Pokemon(String nombre, double vitalidad, double fuerza, double escudo)
+ void addXP(long XP)
+ void ataque(Pokemon otro)
+ Object clone()
+ void curar()
+ int getCategoria()
+ double getEscudo()
+ double getFuerza()
+ String getNombre()
+ double getVitalidad()
Hielo
- boolean granRecarga
+ Hielo(String nombre, boolean granRecarga)
+ Object clone()
+ void curar()
+ void golpeFinal(Pokemon otro)
+ void recarga()
+ void recibeDano(double dano)
+ void recibeNiebla()
+ void recibeTormenta()
+ void recibeViento()
Fuego
+ Fuego(String nombre)
+ Object clone()
+ void curar()
+ void golpeFinal(Pokemon otro)
+ void recarga()
+ void recibeDano(double dano)
+ void recibeNiebla()
+ void recibeTormenta()
+ void recibeViento()
Electrico
- double probCritico
+ Electrico(String nombre)
+ Object clone()
+ void curar()
+ void golpeFinal(Pokemon otro)
+ void recibeDano(double dano)
+ void recibeNiebla()
+ void recibeTormenta()
+ void recibeViento()
Agua
+ Agua(String nombre)
+ Object clone()
+ void curar()
+ void golpeFinal(Pokemon otro)
+ void recarga()
+ void recibeDano(double dano)
+ void recibeNiebla()
+ void recibeTormenta()
+ void recibeViento()
Aire
- double probFallar
+ Aire(String nombre)
+ Object clone()
+ void curar()
+ void golpeFinal(Pokemon otro)
+ void recarga()
+ void recibeDano(double dano)
+ void recibeNiebla()
+ void recibeTormenta()
+ void recibeViento()
Tierra
+ Tierra(String nombre)
+ Object clone()
+ void curar()
+ void golpeFinal(Pokemon otro)
+ void recibeDano(double dano)
+ void recibeNiebla()
+ void recibeTormenta()
+ void recibeViento()


CartaHechizo
- long serialVersionUID
+ Object clone()
# void hechizar(Pokemon pokemon
Tormenta
- long serialVersionUID
+ void hechizar(Pokemon pokemon)
+ String toString()
Viento
- long serialVersionUID
+ void hechizar(Pokemon pokemon)
+ String toString()
Niebla
- long serialVersionUID
+ void hechizar(Pokemon pokemon)
+ String toString()


«java interface»
IState
+ void correr()
+ String getMensaje()
estadoAnterior
estado
Arena
- int cantArena
- CartaHechizo carta1
- CartaHechizo carta2
- CartaHechizo cartaGanada
- boolean disponible
- Entrenador entrenador1
- Entrenador entrenador2
- IState estado
- IState estadoAnterior
- Entrenador ganador
+ Arena()
+ CartaHechizo getCarta1()
+ CartaHechizo getCarta2()
+ CartaHechizo getCartaGanada()
+ Entrenador getEntrenador1()
+ Entrenador getEntrenador2()
+ Entrenador getGanador()
+ int getNumeroArena()
+ Entrenador getPerdedor()
+ Pokemon getPokemon1()
arena
arena
arena
arena
Preliminar
- Arena arena
- String mensaje
- long serialVersionUID
+ Preliminar(Arena arena)
+ void correr()
+ String getMensaje()
Batalla
- Arena arena
- String mensaje
+ Batalla(Arena arena)
+ void correr()
+ String getMensaje()
Definicion
- Arena arena
- String mensaje
+ Definicion(Arena arena)
+ void correr()
+ String getMensaje()
Limpieza
- Arena arena
- String mensaje
+ Limpieza(Arena arena)
+ void correr()
+ String getMensaje()

Clases (30 en total)

PATRÓN MVC - SWING (2) *

- Controlador
- Ventana (Vista)

PERSISTENCIA (2) *

- IPersistencia (Interface)
- PersistenciaBIN

PATRÓN STATE (5) *

- IState (Interface)
- Preliminar
- Batalla
- Definición
- Limpieza

CONCURRENCIA (2) *

- Arena (Recurso compartido)
- Enfrentamiento (Hilo)

PATRÓN FACTORY (1) *

- PokemonFactory

POKEMONES (7)

- Clase padre Pokémon
- Agua
- Aire
- Eléctrico
- Fuego
- Hielo
- Tierra

TIPOS DE CARTAS (3)

- Niebla
- Viento
- Tormenta

INTERFACES (2)

- Inclasificable
- IHechizable *

EXCEPCIONES (3)

- LimiteHechizosException
- NoTieneCartasException
- CloneNotSupportedException

OTRAS (3)

- CartaHechizo *

●Entrenador

●Torneo

(*) Marcador para las nuevas clases implementadas en la 2da parte del TP.

Clase Entrenador

Clase que implementa las interfaces IClasificable, Cloneable, Serializable y Comparable<Entrenador>. Contiene conjuntos de Pokemones y cartas en forma de ArrayList.

Al momento de ingresar a la arena el entrenador tiene la obligación, para continuar en el torneo, de tener pokemones disponibles. De ser así al momento de ser elegido para enfrentarse a otro entrenador se seleccionara aleatoriamente un Pokémon para la pelea. En caso contrario el entrenador es eliminado. También es aleatorio el uso de cartas de hechizo. Si el entrenador usa una carta esta se selecciona mediante el método sacarCartaRandom(), el cual puede llegar a retornar una carta de tipo ICartaHechizo o lanzar dos tipos de excepciones: LimiteHechizosException (cuando ya se alcanzó el límite de cartas utilizables por el entrenador en el torneo) y NoTieneCartasException (cuando el entrenador no tiene ninguna carta disponible). Ambas excepciones se extienden de la clase Exception.

La clonación del entrenador es profunda y no siempre se puede realizar, debido a que hay pokemones que no son clonables. Al ganar en la arena, el entrenador sumará créditos y su pokemon experiencia, mientras que al perder, el entrenador se elimina de la lista de entrenadores del torneo.

Clase Pokemón

Clase abstracta que implementa las interfaces IClasificable, Cloneable, IHechizable, Comparable<Pokemon> y Serializable, de la cual se extienden los diferentes tipos de Pokemones.

En un principio se planteó la posibilidad de implementar el patrón Decorator para diferenciar los tipos de Pokemones e ir agregando funcionalidades a la clase principal, pero luego de analizarlo con más detenimiento simplemente usando herencia no se daba una explotación de clases que justifique su uso. Está clase se encarga de gestionar los atributos, de la estructura del ataque y de la implementación del golpe inicial.

Cada instancia de la clase tiene un estado inicial en común, y este se va a ir modificando a medida que peleen en la arena. Los atributos que representan a los Pokemones son: nombre, vitalidad, fuerza y escudo.

La forma de ataque está implementada mediante el patrón Template. Este está formado por un algoritmo general con tres pasos: golpe inicial, recarga y golpe final.

Está garantizado, mediante las pre condiciones, que el ataque siempre será hacia una variable de tipo Pokemon (o de sus subclases) no nula.

Todas las subclases que se extienden de la clase Pokémon son clases concretas.

Método public void ataque(Pokemon otro)

- golpeInicial: Primer paso del algoritmo general. Se emplea un golpe inicial que es genérico para todos los Pokemones. Este está implementado en la clase abstracta Pokemon, y

por lo tanto, ninguna subclase necesita sobreescribirlo. Este golpe consiste en dañar al Pokémon enemigo por los puntos de fuerza del Pokémon atacante, y luego este atributo será reducido a la mitad.

❏ recarga: Segundo paso del algoritmo general. En la clase padre está implementado el método genérico para los tipos de pokemones que no recargan, indicándose mediante una salida por pantalla. Los que sí lo hacen sobrescriben este método mediante un hook.

Lista de tipos de Pokemones con sus respectivas recargas:

★ Aire: Su fuerza se regenera 125% y su escudo 100%

★ Agua: Su fuerza se regenera 10% y su vitalidad 10%

★ Fuego: Su fuerza se regenera 10% y su vitalidad 10%

★ Hielo*: Su fuerza se establece en 400 puntos si posee el atributo granRecarga.En caso contrario su fuerza y su vitalidad se regeneran en 10%.

★ Eléctrico: No recarga.

★ Tierra: No recarga.

*Hielo: El atributo granRecarga es un boolean que se envía como parámetro al constructor de este tipo de Pokemon.

❏ golpeFinal: Tercer y último paso del algoritmo general. Este método está declarado como abstracto, por lo cual cada subclase tiene la obligación de implementarlo. Todos los tipos de Pokemones realizan golpes finales diferentes, donde algunos tienen probabilidades de fallar o probabilidad de un golpe crítico (daño aumentado proporcional a su fuerza).

Lista de tipos de Pokemones con sus respectivos golpes finales:

★ Aire*: Tiene una probabilidad de fallar de 10%. Si no lo hace, dañará al enemigo con el total de la fuerza del Pokémon.

★ Agua: Daña al enemigo con el total de su fuerza y luego este atributo se reduce a la mitad. Idéntico al golpeInicial.

★ Fuego: Daña al enemigo con el 125% de su fuerza actual, y luego este atributo se vuelve nulo.

★ Hielo: Daña al enemigo con el 90% de su fuerza. Si el Pokémon tiene la habilidad de granRecarga, el atributo fuerza vuelve a su valor base (100 puntos).

★ Eléctrico*: Tiene una probabilidad de crítico del 25%. Si lo hace dañará al enemigo con el doble de su fuerza actual, y sinó lo hará con el valor de su fuerza actual.

★ Tierra: Daña al enemigo con el 300% de su fuerza.

*Aire: La probabilidad de fallar (10%) está dada por un atributo privado double probFallar.

*Eléctrico: La probabilidad de golpe crítico (25%) está dada por un atributo privado double probCritico.

Al realizar un Pokémon el golpe inicial o golpe final, el Pokémon que recibe el ataque asimilará ese daño de diferentes maneras, dependiendo de su tipo. Está funcionalidad está

determinada por el método abstracto recibeDano (double dano), que sobreescribe cada subclase.
Lista de tipos de Pokemones con sus respectivos recibeDano.

★ Aire: Tiene una probabilidad de esquivar de 20%. Si lo hace no recibirá daño, sinó se le descontará la cantidad total del daño al escudo, y cuando este sea 0, comenzará a decrementar su vitalidad.

★ Agua: Se le descontará la cantidad total del daño al escudo, y cuando este sea 0, comenzará a decrementar su vitalidad.

★ Fuego: Su escudo y su vitalidad bajarán en proporción a la mitad del daño recibido.

★ Hielo : Su escudo se decrementará en proporción de un 75% del daño recibido, y en un 25% la vitalidad.

★ Eléctrico: Su escudo se decrementará en proporción de un 10% del daño recibido, y en un 90% la vitalidad.

★ Tierra: Su escudo se decrementará en proporción de un 80% del daño recibido, y en un 20% la vitalidad.

En todos los casos, en el momento en que el escudo sea nulo, el daño afectará en forma total a la vitalidad.

Otra funcionalidad que se suma a ésta clase es la de recibir una maldición por parte de una carta hechizo. La forma implementada es mediante el patrón Double Dispatch, donde se implementan tres métodos de la interface IHechizable: recibeNiebla, recibeViento y recibeTormenta. Estos métodos deberán ser implementados en cada tipo de Pokémon, para definir cómo les afectará el hechizo.

Cada clase es invulnerable a un hechizo específico.

Lista de tipos de Pokemones con sus respectivos recibeNiebla.

★ Aire: Se duplica la probabilidad de fallar.
★ Agua: La fuerza se reduce en un 80%
★ Fuego: No le afecta.
★ Hielo : La fuerza se reduce en un 30%
★ Eléctrico: Probabilidad de crítico se reduce un 60%
★ Tierra: No afecta.

Lista de tipos de Pokemones con sus respectivos recibeViento.

★ Aire: No afecta.
★ Agua: El escudo se reduce en un 15%
★ Fuego: El escudo se reduce en un 50%
★ Hielo : No afecta.
★ Eléctrico: La vitalidad se reduce en un 15%
★ Tierra: El escudo se reduce en un 33%

Lista de tipos de Pokemones con sus respectivos recibeTormenta.

- ★ Aire: La vitalidad se reduce en un 20%
- ★ Agua: No afecta
- ★ Fuego: La vitalidad se reduce en un 10%
- ★ Hielo : El escudo se reduce en un 25%
- ★ Eléctrico: No afecta.
- ★ Tierra: La vitalidad se reduce en un 15%

Cuando un hechizo no le afecta a un tipo de Pokémon en particular, se informa por pantalla en el método sobrescrito.

Cuando un hechizo es aplicado sobre un Pokémon, este quedará hechizado durante todo el Torneo.

Ésta clase también contiene un método abstracto curar, en el cual establece la salud del Pokémon en su valor base.

Con respecto a la clonación, la clase Pokémon realiza la sobrescritura del método clone() de la clase Object, para hacerlo público.

La creación de cada Pokémon es delegada a la clase PokemonFactory, en el cual se le envía el nombre y el tipo como parámetros String y devuelve como resultado el objeto Pokémon ya generado. En el caso del tipo Pokémon Hielo, donde se diferencia el Pokémon Hielo con SuperRecarga, del que no, esto se determina según un número random. Esto hace que al crear un Pokémon de Hielo haya un 50% de probabilidad de que admita SuperRecarga, y otro 50% de que no.

Tipos de Pokemones clonables y no clonables:

CLONABLE	NOCLONABLE
AGUA	AIRE
HIELO	
FUEGO	TIERRA
ELECTRICO	

En los tipos no clonables se lanza una nueva excepción de tipo CloneNotSupportedException, donde se envía un mensaje informando que la clase no puede realizar la clonación. En los que siempre son clonables, en cambio, se trata la excepción dentro del método clone(), eliminando la necesidad de la sentencia throws en su firma para evitar que futuras clases que extiendan de estos tipos puedan dejar de ser clonables.

Tabla de valores iniciales de los diferentes tipos de Pokemones:

Tipo	Vitalidad	Escudo	Fuerza
Aire	500	40	40
Agua	500	100	120
Electrico	600	50	80
Fuego	530	200	80
Hielo	500	120	100
Tierra	700	150	20

Clase CartaHechizo

Clase abstracta que implementa la interfaz Serializable y que se encarga de definir el método para realizar el doble dispatch entre los tipos de pokemones y de cartas. Mediante su método void hechizar(Pokemon pokemon) se logra hechizar a un Pokémon. Dicho método es desarrollado en cada una de las clases que se extienden de esta clase (Niebla, Viento y Tormenta). La implementación funciona debido al polimorfismo, ya que por cada tipo de Pokémon se ejecutará un método distinto. Esta selección del método a ejecutar se da en

tiempo de ejecución. A su vez, cada clase que se extiende de esta también implementa Cloneable, lo cual indica que todas las cartas son clonables.

Interfaz IClasificable

Interfaz que se encarga de hacer que tanto los Pokemon como los entrenadores sean clasificables. En el primer caso su categoría depende de sus puntos de experiencia, mientras que en el segundo es la suma de las categorías de todos sus Pokemon vivos. Cuando un Pokemon muere y es eliminado de la lista de Pokemones de un entrenador, la categoría de este último se reduce.

Interfaz IHechizable

Interfaz que se encarga de contener los métodos necesarios para realizar el Double Dispatch. Está interfax es implementada por la clase abstracta Pokémon.

Clases LimiteHechizosException y NoTieneCartasException

Se extienden de la clase Exception propia de Java. Sus constructores (públicos) reciben un mensaje de tipo String el cual incluye el nombre del entrenador que solicitó usar una carta pero alcanzó su límite máximo de cartas en el torneo o no tiene cartas en su mazo, respectivamente. Este mensaje puede obtenerse usando e.getMessage(), siendo e una instancia de la clase.

Clase CloneNotSupportedException

Clase propia de Java. Se usa con un mensaje de tipo String pasado por constructor que especifica cuál es el tipo de Pokémon que no se puede clonar. Este mensaje puede obtenerse usando e.getMessage(), siendo e una instancia de la clase.

Clase Arena y Enfrentamiento (Concurrencia)

La clase Arena se extiende de Observable e implementa la interfaz Serializable.
En el torneo habrá una cantidad de arenas (4) que podrán estar abiertas simultáneamente. Cada una de estas será un recurso compartido (Monitor).

La arena contendrá un atributo numeroArena que la representará (para loguear sucesos en la vista) y varios atributos como: boolean disponible, ambos entrenadores con sus respectivos pokemones, cartas hechizos y el estado (Al instanciar una Arena su estado es Preliminar y su disponibilidad es true).

Está clase contiene el método iniciar en el que se le pasa por parámetros dos entrenadores, que por precondición son distintos de null. Este método es synchronized y forma parte de la concurrencia. Cuando un hilo bloquea el recurso compartido, el atributo disponible pasa a ser false y siempre se realizará un ciclo for de 4 iteraciones en el que se pasará por los 4 estados designados para el Patrón State (Preliminar, Batalla, Definición y Limpieza). A su vez, en cada iteración se realizará el llamado a los métodos notifyObservers(estado) y setChanged(), para informar a los observadores que hubo un

cambio en el estado de la arena. Una vez finalizado el último estado (Limpieza) se ejecuta el método notify() que informa a los hilos pendientes (si es que hay alguno) de que se liberó el recurso compartido para que alguno, aleatoriamente, tome el bloqueo de la arena.

Por otro lado, está la clase Enfrentamiento que se extiende de Thread e implementa la interfaz Serializable.

En el torneo, la cantidad de enfrentamientos que habrá por etapa será proporcional a la cantidad de Entrenadores aptos para pelear dividido 2. Aclaración: Aptos para pelear se refiere a que estén vivos (que estén en el ArrayList entrenadores del Torneo y que tengan pokemones disponibles.).

Está clase, al extenderse de Thread, debe implementar el método public void run() que definirá la forma de que el hilo selecciona el recurso compartido (arenas) disponible para llevar a cabo el enfrentamiento en la arena. En este método se obtiene la referencia al iterator de las arenas disponibles del torneo, y se comienza a buscar en este ArrayList la arena que esté disponible para pelear. Si no se encuentra ninguna (todas las arenas están llevando a cabo algún enfrentamiento de manera concurrente) se genera un número random entre 0 y la cantidad de arenas activas del torneo, y con este número se accede a la arena correspondiente en el arreglo de arenas para que luego se ejecute el método arena.iniciar() con los dos entrenadores ya pasados por constructor anteriormente. Está elección de la arena mediante un número random genera que el hilo que ejecuta el método de iniciar arena no pueda bloquear el recurso compartido hasta que el enfrentamiento que se estaba llevando a cabo termine y lo libere.

La cantidad de arenas activas por cada etapa estará determinada por el Torneo, que al final de cada una se pregunta si la cantidad de arenas es mayor a la cantidad de entrenadores vivos dividido 2. Si esto es así, entonces se elimina del ArrayList de Arenas pertenecientes al torneo la última arena. Al eliminarse a lo último de cada etapa te garantiza que la arena eliminada no está siendo esperada por ningún hilo para ser bloqueada.

Patrón State

La implementación de este patrón no fue de gran dificultad debido a que ya teníamos, de la primer parte del TP, las distintas partes bien definidas, lo cual, lo único que fue necesario realizar es dividir estas partes en los distintos estados de la Arena.

Para implementar este patrón, creamos una interfaz IState que contiene un método void correr() y un método String getMensaje(). Está interfaz es implementada por los 4 tipos de estados (Preliminar, Batalla, Definición y Limpieza) que sobreescriben este método para realizar determinadas acciones. Cada estado contiene una referencia a la arena, y es el encargado que, al finalizar, cambiar el estado por uno nuevo siguiente.

1. **Estado Preliminar**: Aquí es donde se eligen los pokemones que pelearán en el siguiente estado. Diferenciando de la primer parte, donde mandábamos por parámetro dos atributos booleanos que informaban si los entrenadores usaban o no cartas hechizo, ahora esta elección es de manera aleatoria dentro del estado.
   Una vez finalizado, se realiza un set estado de la arena a Batalla.

2. **Estado Batalla**: Aquí es donde se utilizan las cartas que fueron obtenidas en el estado anterior, y siguiendo con el formato antiguo, se elige mediante un número random qué pokémon atacará primero. La forma de ataque y contraAtaque está dada por el patrón Template, que está formado por un algoritmo general con tres pasos: golpe inicial, recarga y golpe final.
   Cabe destacar que al utilizar una carta, ésta es eliminada del mazo de su usuario.
   Una vez finalizado, se realiza un set estado de la arena a Definición.
3. **Estado Definición**: Aquí es donde se define al ganador en base a una fórmula que tiene en cuenta la vitalidad, la fuerza y el escudo de cada Pokémon al finalizar la pelea (Vitalidad + 2*Fuerza + 0.5*Escudo). En caso de empate gana el Entrenador 1.
   Al ganar el entrenador gana la carta utilizada por su contrincante (si es que usó una), gana créditos de acuerdo a la categoría de este último (multiplicada por 0.4), y su Pokemon gana 3 puntos de experiencia, mientras el Pokemon del perdedor gana solo un punto.
   Una vez finalizado, se realiza un set estado de la arena a Limpieza.
4. **Estado Limpieza**: Aquí es donde se genera la “limpieza” de la arena. Esto consiste en verificar si el Pokémon que ganó murió en batalla (vitalidad menor o igual a 0). En el caso de que así sea, el Pokémon es removido de la lista de pokemones del ganador, y sinó es curado a su vitalidad original.
   El entrenador ganador es añadido nuevamente al ArrayList de entrenadores del torneo.
   Una vez finalizado, se realiza un set estado de la arena a Preliminar, con la intención de que se siga utilizando la Arena para otros enfrentamientos.

En todos los estados, y por simplicidad para conectar con el observador (Controlador), se genera un String mensaje que va concatenando los distintos eventos de cada etapa, y cuando esta termina, el observador es capaz de obtener este mensaje mediante el método getMensaje() de cada Estado, a fin de informar todo lo sucedido en la totalidad del enfrentamiento.

Clase Torneo

Clase que es Serializable e Implementa el patrón Singleton para tener una sola instancia de la clase accesible desde cualquier parte del programa.

Se encarga de la estructura general del torneo y del manejo de las listas de arenas y entrenadores. Esta última tiene un clon no profundo llamado entrenadoresClon utilizado para no perder la referencia a cada entrenador a medida que van siendo eliminados.

Esta clase varió en su funcionalidad a lo que hacía en la primer parte del trabajo práctico. Sus atributos son los mismos, agregando algunos necesarios para las nuevas implementaciones. El torneo ahora se distribuye en etapas, donde al momento de instanciarlo se comienza en la etapa 1.

Contiene un método void correrTorneo() que es ejecutado desde el Controlador, y realiza determinadas acciones según la etapa en la que esté ubicado el torneo.
Si este se encuentra en una etapa de pelea, se seguirán los siguientes pasos:

1. Elegir el primer entrenador, siempre y cuando la cantidad de entrenadores vivos sea mayor a 2. Una vez elegido (de manera aleatoria), se elimina del ArrayList de entrenadores vivos del torneo.
2. Elegir el segundo entrenador, siempre y cuando la cantidad de entrenadores vivos sea mayor a 1. Una vez elegido (de manera aleatoria), se elimina del ArrayList de entrenadores vivos del torneo (idéntico a la elección del primer entrenador).

Aclaración: Si en ambos casos se elige un entrenador que no contiene pokemones vivos, entonces es eliminado del torneo y se continúa la búsqueda de un entrenador apto.

3. Crear una instancia de un nuevo Enfrentamiento donde está asegurado de que los dos entrenadores que se pasan por parámetro no son null. Este nuevo enfrentamiento es agregado a un arreglo del torneo.

Una vez creados todos los enfrentamientos de la etapa, se crea un iterador de enfrentamientos en el cual se realiza el .start a cada uno, para que comiencen las peleas de manera concurrente, según en la arena que le corresponda a cada una.
Una vez que terminaron todos los enfrentamientos, se limpia el arrayList de enfrentamientos y se avanza de etapa.
Si al finalizar la etapa, la cantidad de entrenadores es 1, entonces se da por finalizado el torneo y la etapa pasa a ser -1.

Etapas del torneo:

1. Alta de entrenadores.
2. Alta de pokemones.
3. o Superior: Enfrentamientos.

Etapa -1: Finalización del torneo.
Decidimos que los enfrentamientos sean como los veníamos haciendo, donde los entrenadores a pelear se eligen de manera aleatoria y no haciendo llaves.

Serialización

La persistencia que elegimos fue binaria. Hicimos esta elección ya que este tipo de persistencia nos brinda facilidad debido a que solo necesita que la clase implemente la interfaz Serializable.
La interfaz IPersistencia es la encargada de declarar todos los métodos necesarios (escribir,leer,abrir y cerrar Output e Input). La clase PersistenciaBIN implementa esta interfaz antes mencionada, y desarrolla todos los métodos para que la serialización se lleve a cabo.
El torneo, al finalizar cada etapa, es serializado. Esto permite que, si el torneo es interrumpido en una etapa que no sea la de finalización, sea reanudado desde esa etapa, restaurando el estado de la misma.

Controlador

El controlador funciona como nexo entre la vista y el modelo. Se encarga de enviar la información provista por la ventana (nuevos entrenadores, pokemones, cambios de etapa, etc.) a la clase Torneo explicada anteriormente.
En su constructor se crea una instancia de la vista, se setean los distintos Listener para poder detectar las acciones del usuario en la ventana, y se obtiene la instancia de la clase Torneo, la cual dependiendo de la presencia de un archivo serializado puede ser nueva o restaurada de una ejecución anterior.
Una vez terminado el constructor, el controlador espera que el usuario presione los distintos botones, provocando cambios en el estado del torneo.
Además, esta clase implementa la interfaz Observer y tiene un conjunto de Arenas observables que, mediante sus métodos notifyObservers(), llaman al método update() del controlador el cual actualiza los paneles de cada Arena en la ventana.

Vista

La clase Ventana se extiende de JFrame e implementa las interfaces KeyListener y ListSelectionListener. Mediante esta clase se obtiene la interacción con el usuario.
La validación de datos es verificada por esta clase para facilidad de implementación.
Según la etapa del torneo en la que estemos, es el panel que aparecerá.
En el caso de la etapa 1, utilizamos un panel que permite generar entrenadores con cartas hechizo, con un máximo de 3 (una por tipo). Además, para que se habilite el botón que hace avanzar a la siguiente etapa, es necesario que la cantidad de entrenadores sea la que informa el Torneo mediante su atributo número Entrenadores. También se pueden eliminar entrenadores.
Para la etapa 2, utilizamos un panel que permite generar pokemones, escribiendo su nombre y seleccionando el atributo tipo del pokémon. Además es necesario seleccionar un entrenador para asignar el pokémon a crear. Para esto, se utiliza DefaultListModel donde el modelo mantiene una referencia a todos los entrenadores creados en la etapa 1, lo cual permite seleccionar uno. También se pueden eliminar pokemones de sus respectivos entrenadores.
Para la etapa 3 y posteriores, se crean 4 paneles que simulan las arenas (recursos compartidos) donde aparece en cada una la información del enfrentamiento actual. Para que se habilite el botón de siguiente etapa, es necesario que hayan finalizado todos los enfrentamientos.
Cuando la etapa es -1, entramos en la finalización del torneo, donde se utiliza un TextArea para mostrar los resultados de todos los enfrentamientos del torneo y un JList para las clasificaciones, dónde están ordenadas de mayor a menor, mediante un TreeSet.

En todas las etapas tenemos una sección Anuncios, donde informamos cuando se agrega y elimina correctamente un entrenador y un pokémon, cuando se lee el Torneo mediante un archivo, cuando se persiste al terminar una etapa, cuando comienza la etapa de Finalización y cuando el torneo termina anunciando al ganador.

Al cambiar de etapa utilizamos CardLayout. Esto nos permite tener varios paneles que se muestran de a uno a la vez en una misma posición, y que por etapa mostrará uno determinado, acorde a lo que se deba hacer en ella.

Conclusiones y dificultades

-En esta segunda etapa se tuvieron que implementar más patrones para ampliar las posibilidades de diseño y poder implementar una mejor interacción con el usuario mediante la ventana.
-La persistencia nos permitió guardar el torneo desde cualquier estado, evitando tener que cargar nuevamente los datos.
-En un principio el patrón State nos pareció innecesario debido a que el cambio de estado podría implementarse desde una sola clase, pero luego nos dimos cuenta que nos facilitaba informar en la ventana.
-La forma de generar los enfrentamientos se mantuvo de la primer parte del trabajo debido a que la concurrencia mejoraba la estructura.
-En un principio el panel izquierdo de la ventana estaba implementado por un BorderLayout, cosa que nos trajo inconvenientes al momento de cambiar de etapa. Esto se solucionó con el descubrimiento del CardLayout que ya se explicó anteriormente.
-Nos hubiese gustado ver en la ventana, en las etapas de batalla, los cambios de estado de las arenas en tiempo real y los hilos concurrentes bloqueando los recursos compartidos. Una solución hubiera sido implementar un método Sleep con tiempo aleatorio en el recurso compartido, pero esto rompería las convenciones de concurrencia explicadas por la cátedra.
-En la concurrencia utilizamos notify y no notifyAll debido a que sería innecesario despertar a todos los hilos, sabiendo que el acceso de cada uno no está condicionado de distintas maneras. No se implementó el método Wait dentro de un while con una condición debido a que el modificador synchronized no permite que más de un hilo acceda al recurso compartido.
-La cantidad de recursos compartidos (arenas) fue definida por nosotros debido a que necesitábamos un panel por cada una, y asignar una cantidad de paneles a la vista dinámicamente nos quitaría el control sobre la estética del programa final.