	Carátula para entrega de prácticas	
Facultad de Ingeniería		Laboratorios de docencia

Laboratorio de Computación Salas A y B

Profesor(a): Dávila Pérez René Adrián

Asignatura: Programación Orientada a Objetos

Grupo: 1

No de Práctica(s): Proyecto 3

Integrante(s): 322089020

322089817

322151194

425091586

322085390

No. de lista o brigada: Equipo 4

Semestre: 2026-1

Fecha de entrega: 1 de diciembre de 2025

Observaciones:

CALIFICACIÓN: _____

Índice

1. Introducción

1.1. Planteamiento del Problema

El desarrollo de software orientado a objetos requiere la implementación de modelos que no solo representen entidades estáticas, sino que también gestionen interacciones complejas y flujos de estados dinámicos. Para este proyecto, se plantea el desafío de replicar la lógica subyacente de un sistema de batalla por turnos, basado en los videojuegos de Pokémon.

El problema central consiste en diseñar una arquitectura de software capaz de gestionar múltiples variables simultáneas: el cálculo de daño basado en una matriz de tipos elementales (debilidades y resistencias), la administración de turnos basada en estadísticas de velocidad, y la gestión de inventarios y estados alterados del personaje. Es necesario desarrollar una solución modular bajo el patrón Modelo-Vista-Controlador (MVC) que permita desacoplar la lógica del juego de la interfaz de usuario, garantizando un sistema robusto, extensible y mantenible.

1.2. Motivación

La simulación de sistemas complejos es una de las aplicaciones prácticas más efectivas para consolidar los conocimientos de programación orientada a objetos y estructuras de datos. Este proyecto surge de la necesidad de aplicar conceptos teóricos vistos en el curso —como polimorfismo, abstracción, herencia y manejo de colecciones— en un escenario práctico y reconocible.

Un simulador de batallas no es trivial; es un escenario ideal para enfrentar problemas comunes en el desarrollo profesional, tales como la implementación de patrones de diseño (como *Strategy* para los ítems) y la optimización de algoritmos de búsqueda (para la tabla de tipos). Además, la motivación se extiende a la creación de una experiencia de usuario interactiva en consola, superando la ejecución lineal de scripts simples para crear una aplicación que responde dinámicamente a las decisiones del usuario.

1.3. Objetivos

Objetivo General:

Desarrollar una aplicación de consola en lenguaje Dart que simule fielmente un combate Pokémon, implementando una arquitectura Modelo-Vista-Controlador (MVC) para gestionar la lógica de turnos, el cálculo de daño elemental y la interacción con el usuario.

Objetivos Específicos:

- **Implementar una matriz de efectividad de tipos:** Desarrollar una estructura de datos eficiente que permita calcular multiplicadores de daño (x2, x0.5, x0) para los 18 tipos elementales, evitando el uso excesivo de condicionales.
- **Diseñar un sistema de turnos dinámico:** Programar un algoritmo de control de flujo que determine el orden de actuación basándose en la velocidad de las entidades y que se adapte a modificaciones en tiempo real (como estados de parálisis).
- **Desarrollar un modelo flexible y escalable:** Hacer uso de herencia para permitir la instanciación genérica de Pokémon y la gestión de estados alterados (veneno, quemadura, congelamiento).
- **Integrar interactividad y gestión de recursos:** Habilitar un menú interactivo que permita al usuario elegir estratégicamente entre atacar o utilizar objetos del inventario para recuperar salud o curar estados.

2. Marco Teórico (Conceptos):

Para el desarrollo del simulador de videojuego Pokémon (Proyecto 3), se implementaron herramientas fundamentales de la Programación Orientada a Objetos para aprovechar la modularidad, reutilización de código. A continuación, se detalla la aplicación práctica de cada concepto:

2.1. Abstracción de Clases

La abstracción permite representar entidades complejas del mundo real como modelos de software simplificados, centrados en los atributos y comportamientos relevantes para el contexto del sistema [2].

Implementación: Se definieron las clases `Pokemon` y `Ataque`. La clase `Pokemon` abstrae las estadísticas vitales como *Hit Points* (HP), Velocidad y Nivel, ocultando complejidad innecesaria.

2.2. Herencia

La herencia es el mecanismo por el cual una clase se deriva de otra, adquiriendo sus atributos y métodos, lo que permite la reutilización de código y establece una jerarquía de tipos [1].

Implementación: Para satisfacer el requerimiento de "ítems de curación", se diseñó una jerarquía de herencia donde la clase abstracta `Item` sirve como clase base. De ella derivan las clases concretas `Pocion` y `CuraTotal`. Esto permitió compartir el atributo `nombre` y obligar a la implementación del comportamiento específico en las subclases.

2.3. Polimorfismo

El polimorfismo permite que objetos de diferentes clases respondan al mismo mensaje de manera distinta, facilitando la extensibilidad del sistema [2].

Implementación: El método abstracto `usar(Pokemon p, CombateView view)` definido en la clase `Item` es polimórfico.

- En la clase `Pocion`, el método incrementa la propiedad `vida`.
- En la clase `CuraTotal`, el método altera la propiedad `estado` a `Estado.sano`.

El controlador del juego (`CombateController`) invoca este método sin necesidad de conocer el tipo exacto del objeto, simplificando la lógica del menú "Mochila".

2.4. Encapsulamiento y Arquitectura MVC

El encapsulamiento oculta los detalles internos de implementación y protege la integridad de los datos, exponiendo solo una interfaz pública segura [3].

Implementación: Se adoptó el patrón de arquitectura Modelo-Vista-Controlador (MVC):

- **Modelo:** Clases de datos como `Pokemon` y `TablaTipos`.
- **Vista:** La interfaz `CombateView` encapsula todas las operaciones de entrada/salida (I/O), separando la lógica de presentación de la lógica de negocio.
- **Controlador:** La clase `CombateController` gestiona el flujo de la batalla.

2.5. Miembros Estáticos (Static)

Los miembros estáticos pertenecen a la clase en lugar de a una instancia particular, permitiendo el acceso a utilidades y constantes compartidas [1].

Implementación: La clase `TablaTipos` utiliza un mapa estático (`efectividad`) y un método estático (`obtenerMultiplicador`) para calcular las debilidades y resistencias (x2.0, x0.5, x0.0). Esto permite consultar la tabla de tipos globalmente sin instanciar objetos, optimizando el uso de memoria según la Figura 1 del planteamiento.

3. Desarrollo

3.1. Explicación del Código

Paso 1: Matriz de Efectividad

Requerimiento: El sistema requiere implementar una lógica de batalla fiel a la "Figura 1"de los requerimientos, donde cada tipo elemental tiene debilidades (x2), resistencias (x0.5) e inmunidades (x0). El código original utilizaba condicionales anidados (`if/else`) que solo

cubrían un caso específico (Fuego vs Hierba), lo cual no era adecuado para manejar los 18 tipos existentes de forma ordenada.

Implementación realizada: Se implementó la clase `TablaTipos` para manejar todas las combinaciones de daño. Se utilizó una estructura de datos tipo mapa (`Map`) que funciona como una tabla de doble entrada. Esto permite buscar el multiplicador de daño accediendo directamente mediante el tipo de ataque y el tipo de defensor. De esta forma, el programa obtiene el resultado (2.0, 0.5, etc.) de manera directa y rápida sin necesidad de escribir múltiples condiciones lógicas, facilitando agregar más tipos si fuera necesario.

Paso 2: Modelo

Requerimiento: Se requería que la aplicación soportara múltiples instancias de Pokémon de diferentes tipos sin tener que crear una clase nueva por cada uno (como `class PokemonFuego`). Además, como puntos extra, se solicitó la inclusión de mecánicas de estados alterados (Quemado, Paralizado, Congelado, Envenenado) que afectaran la vida o la capacidad de movimiento del Pokémon.

Implementación realizada: Se modificó la clase `Pokemon` para que sea más general y flexible. Ahora recibe sus propiedades (tipo, movimientos, vida) directamente al momento de crearse, eliminando la necesidad de crear subclases específicas. Para manejar los estados, se implementó un listado fijo (`enum Estado`) que evita errores de texto. Se agregaron funciones dentro de la clase: `puedeMoverse()`, que verifica si el Pokémon puede atacar antes de su turno, y `aplicarEfectoDeEstado()`, que calcula y resta la vida por daño continuo al finalizar el turno.

Paso 3: Menú

Requerimiento: Según la "Figura 2", el simulador no debía ejecutarse automáticamente hasta el final, sino que debía pausar la ejecución en cada turno para permitir al usuario tomar decisiones. El sistema debía presentar un menú claro con las opciones "LUCHAz

"MOCHILA", y responder a lo que escriba el usuario para navegar por estas opciones.

Implementación realizada: Se integró la librería `dart:io` para permitir que el programa lea lo que el usuario escribe en la consola. Se modificó el ciclo principal del controlador para que funcione paso a paso. Ahora, el código muestra el menú en pantalla, detiene la ejecución esperando una respuesta, verifica si se escribió "1.º "2", y ejecuta la acción correspondiente (mostrar la lista de ataques o abrir el inventario) según la elección del jugador.

Paso 4: Inventario

Requerimiento: Como parte de los puntos extra, el sistema debía permitir al usuario utilizar objetos durante la batalla para recuperar puntos de salud o curar estados alterados. Esto implicaba que .ªtacar"no fuera la única opción y que usar un objeto gastara el turno del jugador sin hacer daño al rival.

Implementación realizada: Se implementó una clase base llamada `Item` para definir cómo deben comportarse los objetos. A partir de ella se crearon las clases `Pocion` (que suma vida hasta un tope máximo) y `CuraTotal` (que limpia el estado del Pokémon). El controlador ahora maneja una lista de estos objetos y, cuando el usuario elige uno, simplemente ejecuta su función de uso y descuenta el objeto del inventario.

Paso 5: Lógica de Turnos

Requerimiento: El sistema de batalla debía determinar el orden de actuación basándose en la velocidad de los Pokémon: el más rápido ataca primero. Sin embargo, este cálculo debía considerar cambios durante la batalla, como la reducción de velocidad por estar "Paralizado", y debía manejar el caso donde el jugador usa un objeto (cediendo el turno al rival).

Implementación realizada: Se implementó una lógica que compara las velocidades de ambos Pokémon en cada turno. El programa verifica la velocidad actual (considerando si el estado la reduce) y decide el orden. Si el jugador ataca, se compara su velocidad con la

del rival; si el jugador usa un ítem, se programó para que el rival ataque automáticamente primero. También se agregaron verificaciones para asegurar que, si un Pokémon pierde toda su vida tras el primer ataque, el turno termine inmediatamente.

Encapsulación: En el programa, la encapsulación se refleja en la manera en que cada componente administra su propia lógica interna sin exponer detalles innecesarios. La función `main` únicamente define el flujo general, mientras que el manejo del tiempo de espera y la ejecución diferida se encuentra dentro de `Future.delayed`. Esta separación permite mantener un control adecuado sobre las partes del programa y favorece la organización del código, evitando dependencias excesivas entre módulos.

Abstracción: La abstracción se aplica mediante el uso de estructuras como `Future` y `Duration`, las cuales ocultan la complejidad del manejo del *Event Loop*. El programador sólo debe especificar qué tarea desea ejecutar y en cuánto tiempo, sin preocuparse por los detalles internos del procesamiento asíncrono. Esto permite trabajar con procesos complejos mediante interfaces simples y comprensibles.

Composición: El programa implementa composición al integrar objetos de apoyo dentro del flujo principal sin necesidad de recurrir a herencia. Componentes como `Duration(seconds: 2)` y `Future.delayed` se utilizan directamente como partes funcionales del programa. Esta técnica favorece la reutilización de componentes ya existentes y permite construir comportamientos más completos sin duplicar código.

Polimorfismo: El polimorfismo aparece cuando se pasa una función callback dentro de `Future.delayed`. Este callback puede representar distintos comportamientos, lo que permite que la misma estructura ejecute acciones diferentes según la función proporcionada. Gracias a esto, el diseño se vuelve flexible y adaptable sin modificar la estructura principal.

Responsabilidad Única: Cada elemento del programa cumple una única responsabilidad. La función `main` se encarga del flujo general, `Future.delayed` administra la tarea programada y el *Event Loop* controla la ejecución en segundo plano. Esta distribución clara evita que un módulo realice tareas adicionales y facilita la lectura y mantenimiento del

código.

Arquitectura MVC: Aunque el programa es corto, en su estructura puede identificarse una organización compatible con el patrón *Modelo–Vista–Controlador* (MVC). La función `main` actúa como el *Controlador*, ya que coordina el flujo del programa e indica cuándo deben ejecutarse las tareas. El manejo del tiempo mediante `Future` y `Duration` funciona como el *Modelo*, puesto que encapsula la lógica interna relacionada con los procesos asíncronos. Finalmente, las salidas impresas con `print` representan la *Vista*, encargada de mostrar la información al usuario. Esta separación implícita de responsabilidades contribuye a un diseño organizado, modular y fácil de extender.

3.2. Diagramas UML

4. Resultados relevantes

5. Conclusiones

El desarrollo de este código permitió consolidar los conocimientos fundamentales de la programación orientada a objetos, aplicando principios como la encapsulación, la abstracción, la composición y el polimorfismo dentro de un ejemplo práctico y funcional. Asimismo, el análisis de su estructura facilitó la comprensión de cómo estos conceptos se integran naturalmente en el manejo de tareas asíncronas en Dart. De igual manera, el estudio del programa posibilitó relacionarlo con los fundamentos del modelado mediante diagramas UML, ya que fue posible identificar responsabilidades, flujos de interacción y la separación implícita en capas compatible con la arquitectura MVC. En conjunto, este ejercicio no sólo reforzó aspectos teóricos de la POO y de los modelos de diseño, sino que también permitió comprender de forma más profunda el comportamiento interno del código, su organización y los principios que lo hacen claro, modular y escalable.

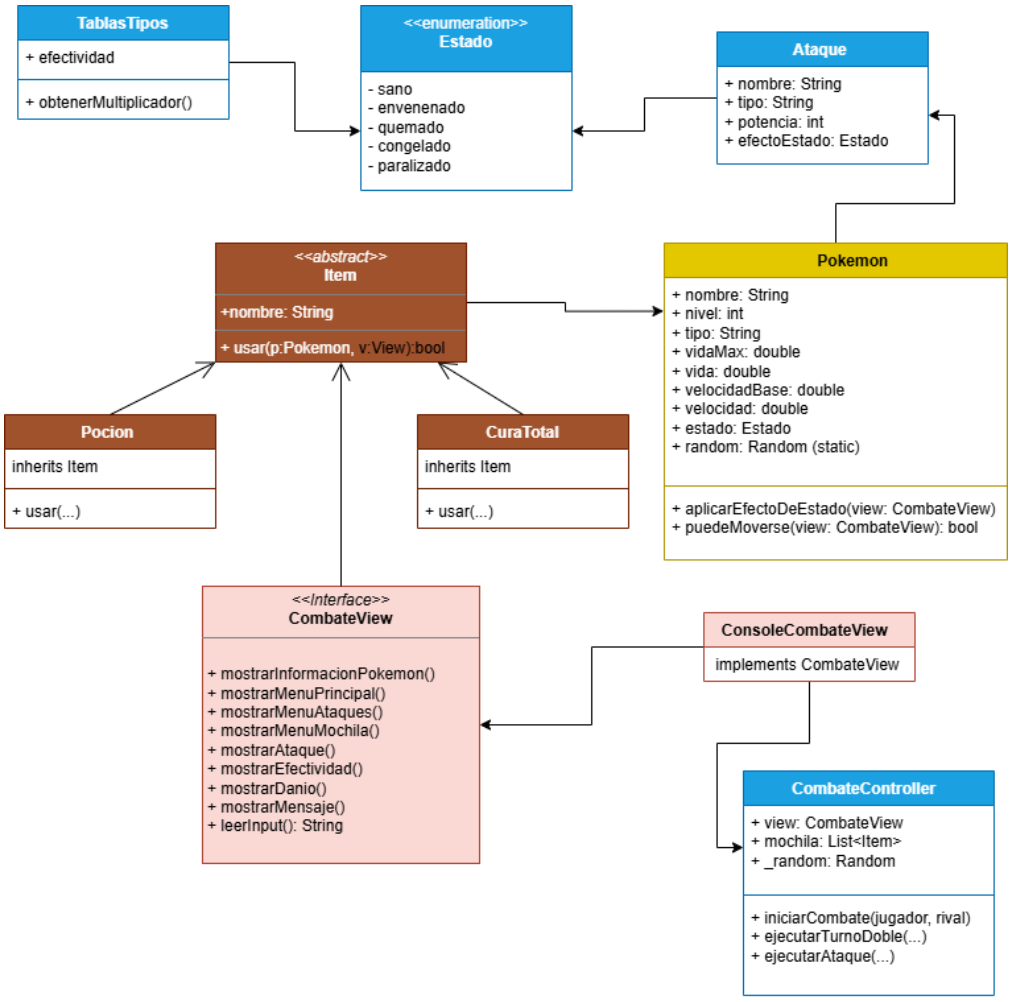

Figura 1: Diagrama estatico de clases.

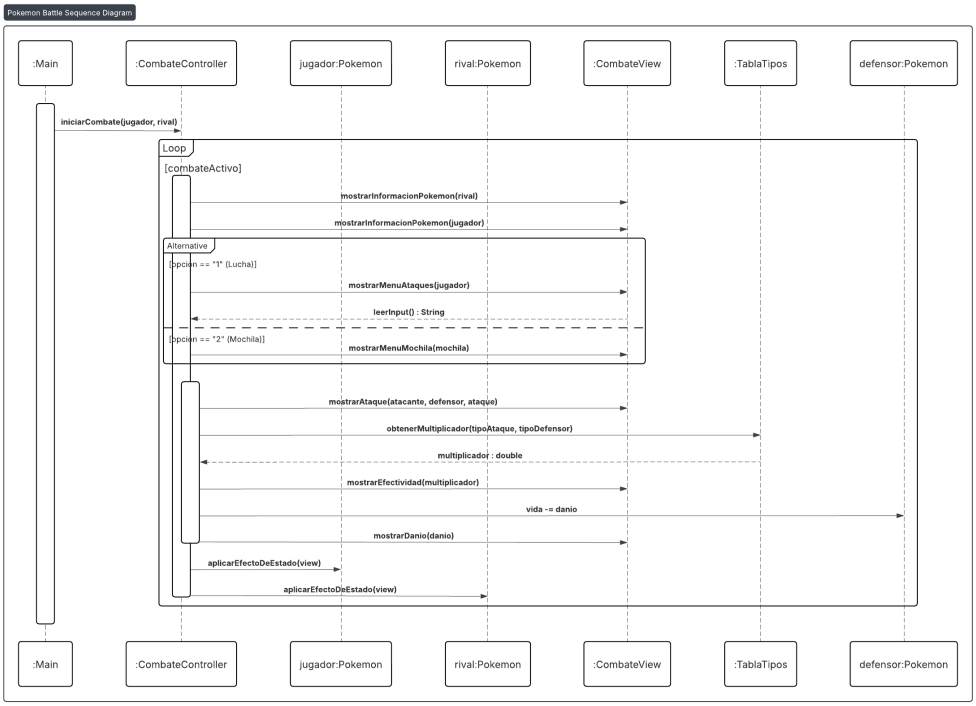

Figura 2: Diagrama dinamico de secuencia.

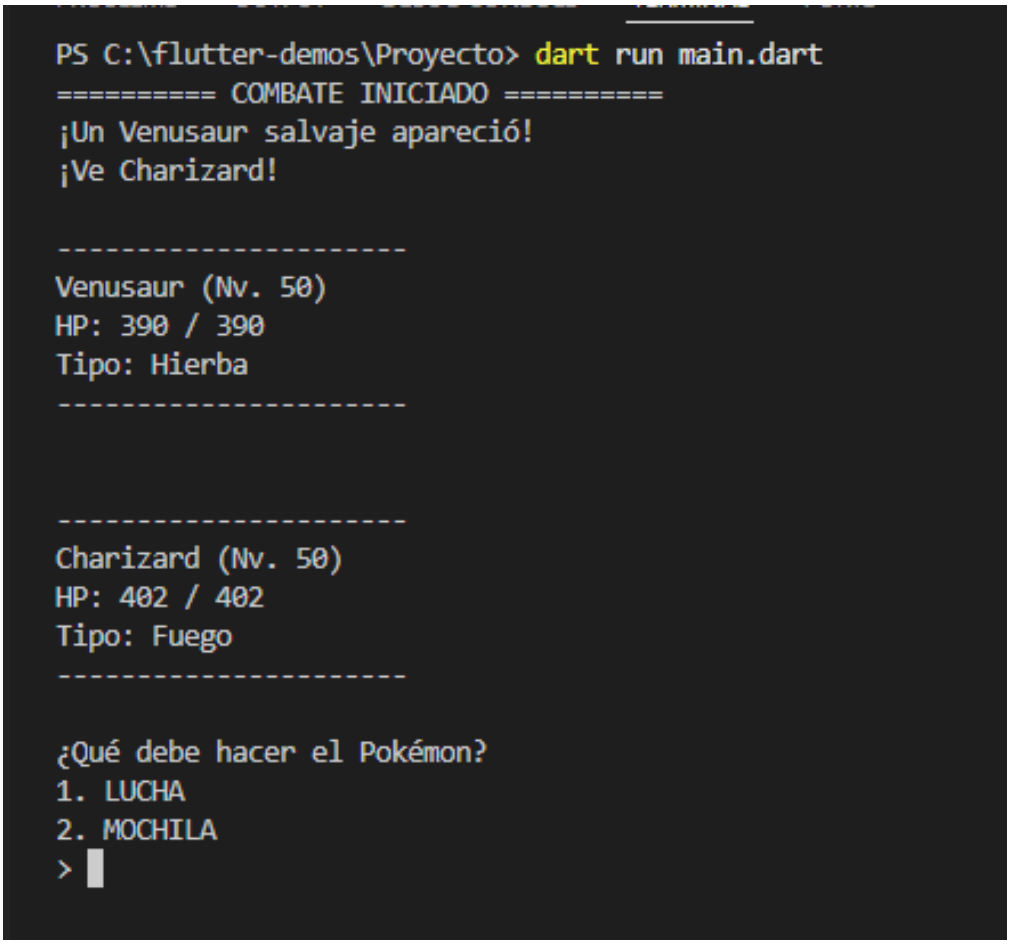

Figura 3: Menú al iniciar la aplicación (Status de batalla inicial)

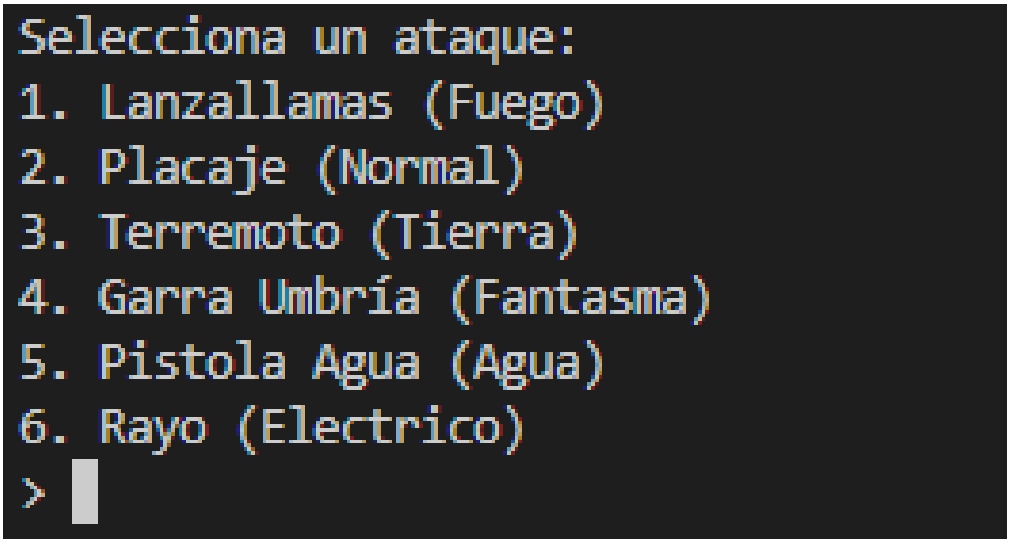

Figura 4: Menú de selección de ataque

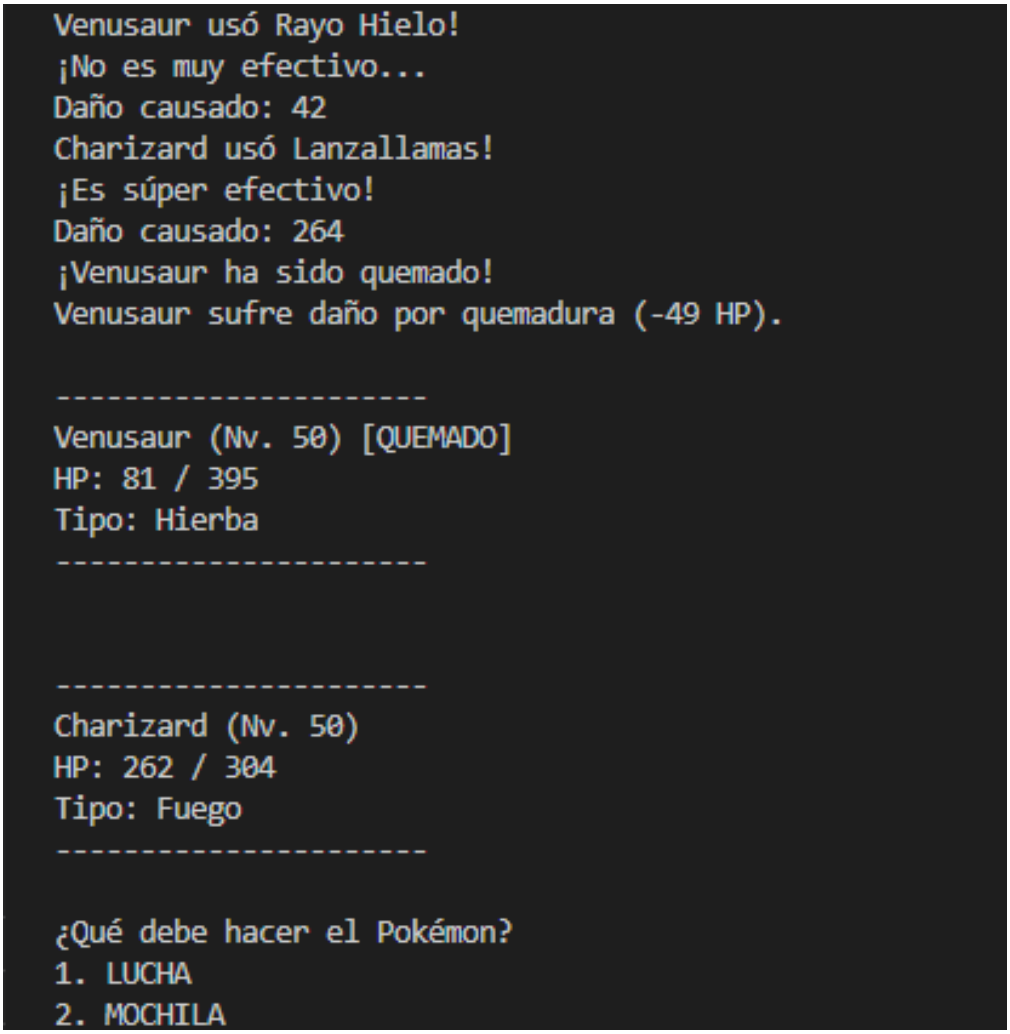

Figura 5: Status de la batalla después de realizar un movimiento (ataque)

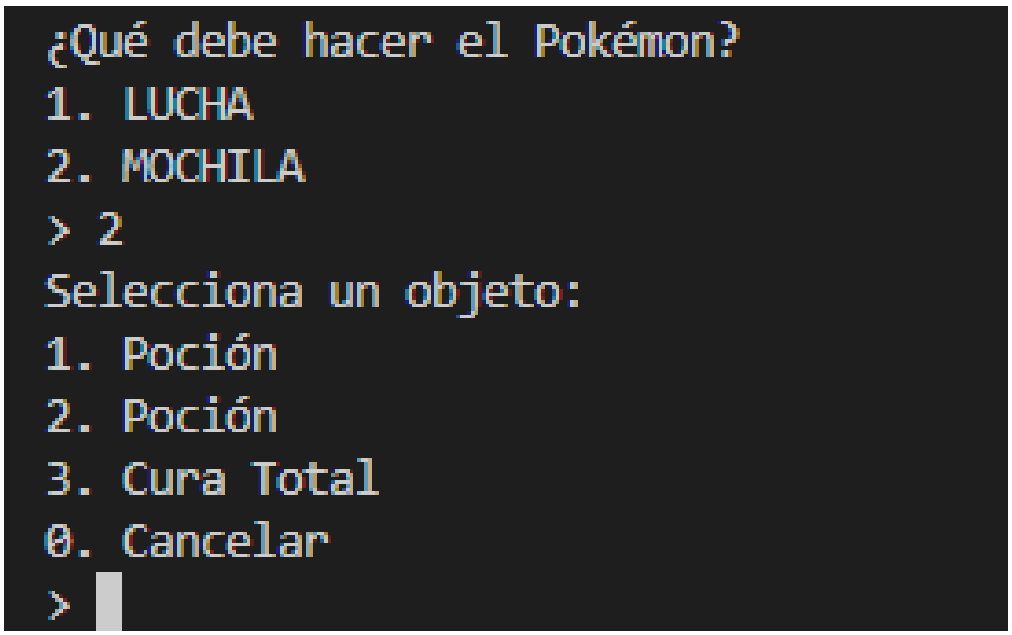

Figura 6: Menú de selección de objetos en mochila

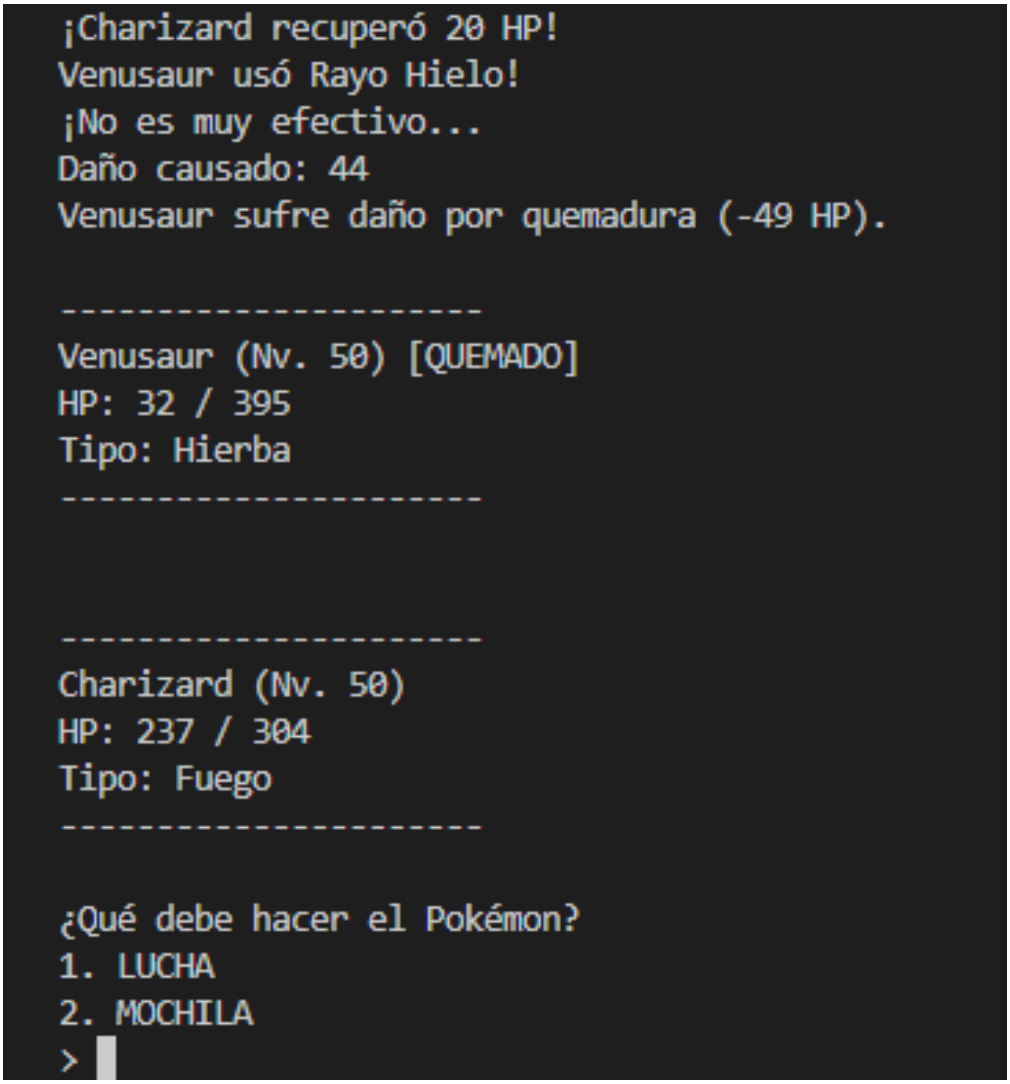

Figura 7: Status de batalla después de usar poción

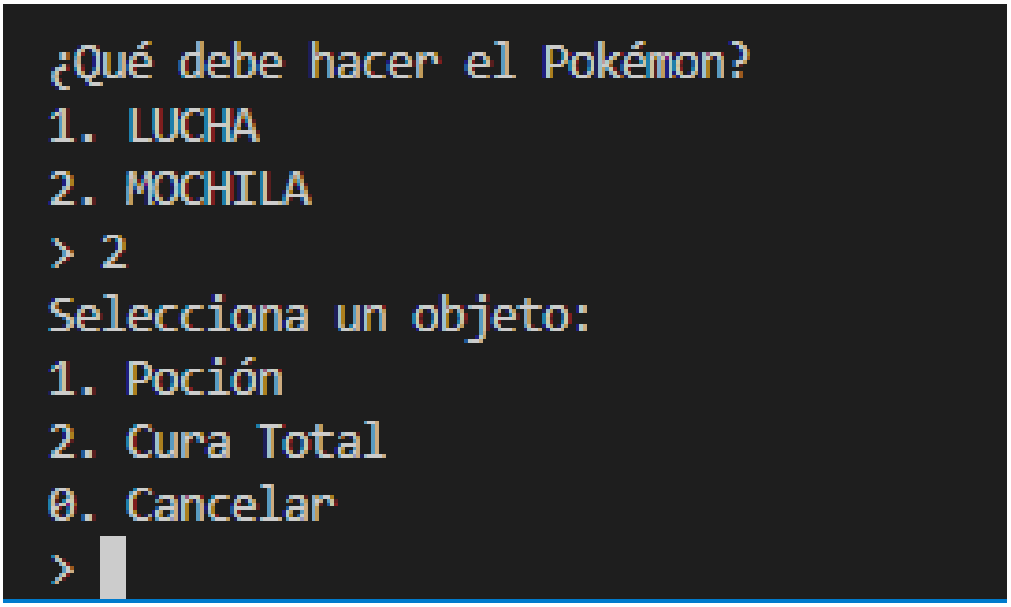

Figura 8: Status de mochila después de usar un objeto (consumibles)

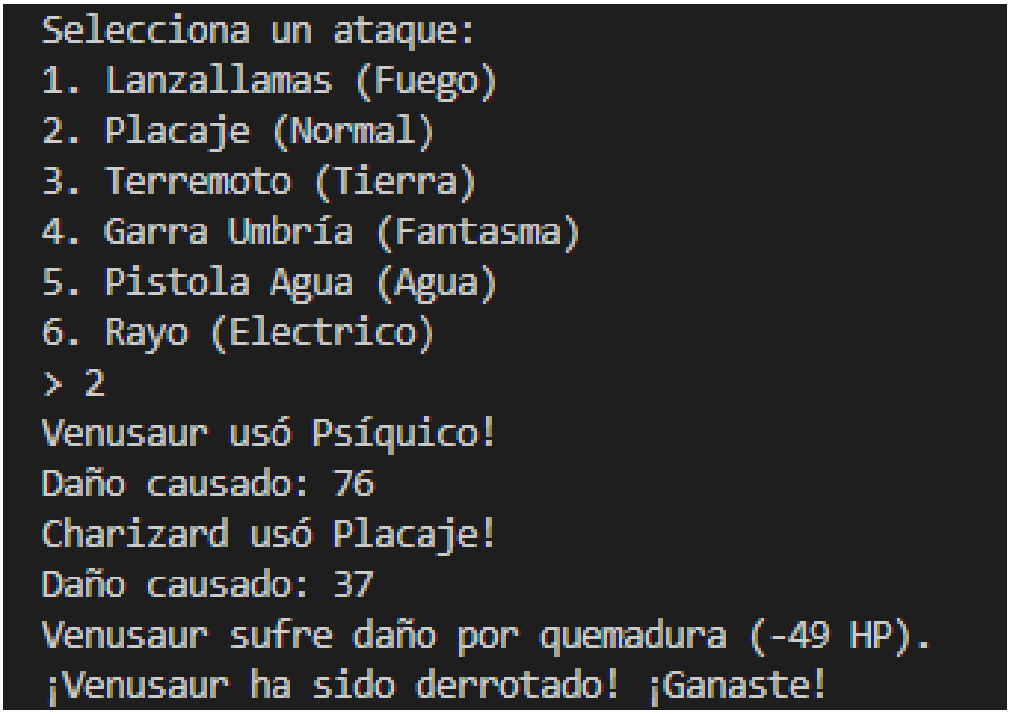

Figura 9: Último ataque realizado por el usuario, pokemón rival derrotado

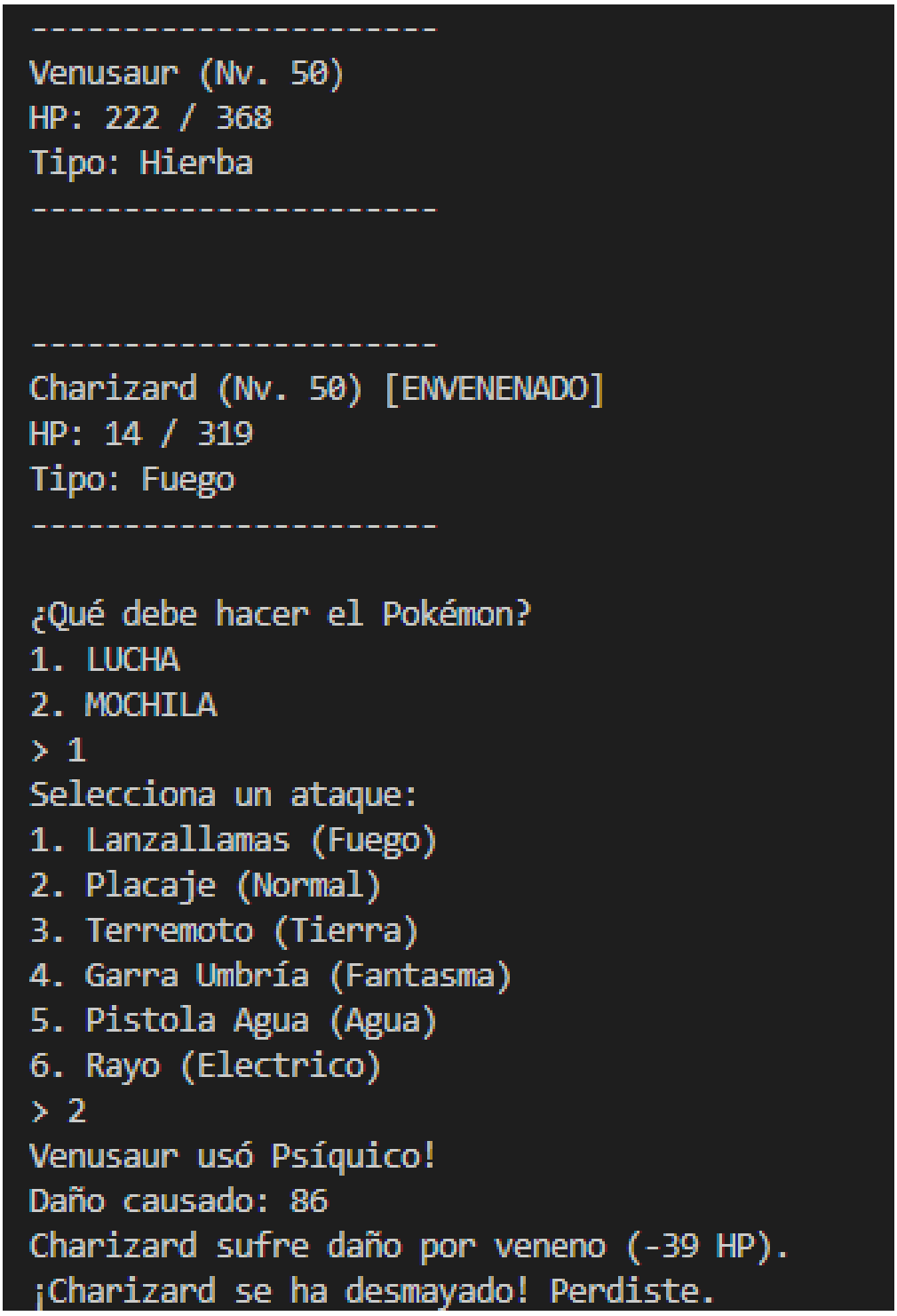

Figura 10: Status mostrado en caso de perder la batalla

Referencias

[1] Paul Deitel y Harvey Deitel. *Java: Cómo programar*. 9.ª ed. México: Pearson Educación, 2012.

[2] Luis Joyanes Aguilar. *Fundamentos de Programación: Algoritmos, estructura de datos y objetos*. 4.ª ed. Madrid: McGraw-Hill, 2008.

[3] Ian Sommerville. *Ingeniería de Software*. 9.ª ed. Boston: Addison-Wesley, 2011.