**Patrón Strategy o Policy**

**Información general del proyecto**

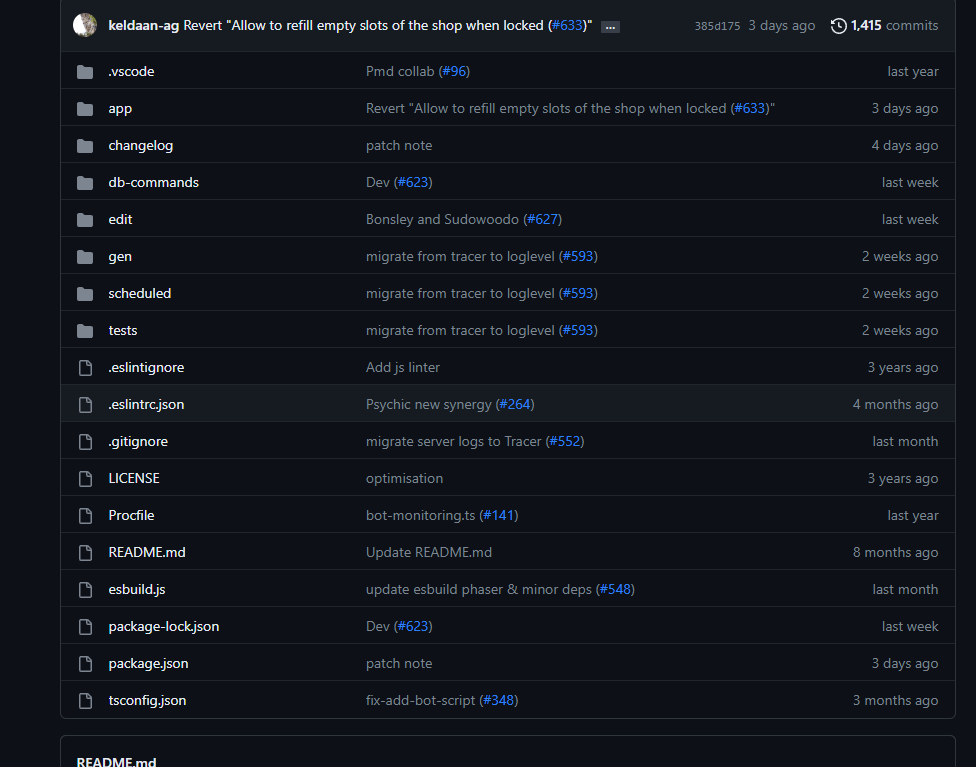
**¿Para qué sirve?**

El propósito del proyecto es divertir, no busca resolver una problemática, consiste en un videojuego basado en la franquicia de Pokémon que busca ser de “fans para fans”. El proyecto se aleja del modelo clásico de juego, brindando una experiencia de juego totalmente diferente.

**¿Cuál es la estructura general del diseño?**

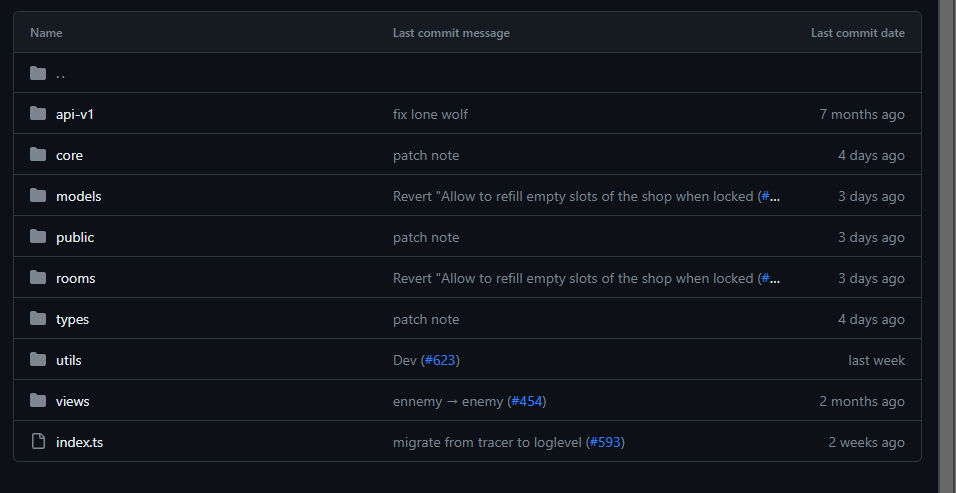
El proyecto posee una amplia variedad de elementos de su diseño, dispersos a través de distintas carpetas. Se abordará principalmente lo que concierne a la jugabilidad, especialmente, se omitirán aspectos como menús o lobbies.

La carpeta principal del proyecto contiene las siguientes carpetas:



*Imagen 2.1*

La implementación concreta del juego se encuentra dentro de la carpeta app, la cual contiene las siguientes carpetas:



*Imagen 2.2*

**Core:** Contiene la lógica del juego

**Models:** contiene la lógica para poder guardar y acceder a información de los usuarios y sus Pokémon.

**Public:** Contiene muchos elementos visuales como iconos y sonidos, además, contiene información sobre los Pokémon que poseen un tipo especifico de habilidad.

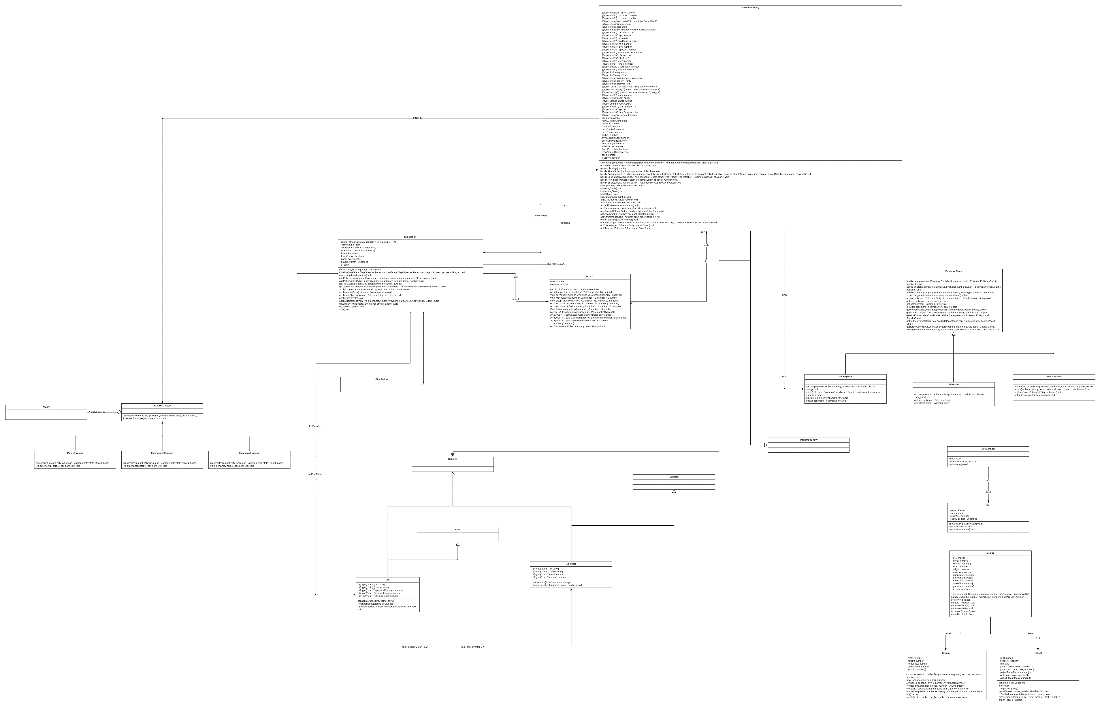
**Rooms:** Contiene los diferentes lobbies en los que puede estar un usuario

**Types:** En esta carpeta se encuentra la lógica principal para el acceso a la aplicación y los tipos de usuarios disponibles. También cuenta con archivos de configuración para ciertos aspectos del juego. Por último, cuenta con una basta implementación de Pokémon y sus características, pero para ser desplegados principalmente dentro de los menús del juego. Sin embargo, también se encuentra la clase Ability, la cual si es usada dentro de la lógica del juego

**Utils:** Contiene funciones de apoyo, algunas por ejemplo se usan para hacer operaciones sobre arrays y otras se usan para implementar funciones con math.random.

**Views:** Contiene el archivo HTML principal

Dentro de la carpeta core, se encuentran los aspectos principales de la jugabilidad:



*Imagen 2.3 UML*

La jerarquía del BotManager hace que el usuario pueda jugar contra bots cuando no hay suficientes jugadores disponibles

La jerarquía de Design despliega elementos adicionales para el diseño del tablero.

Las demás clases corresponden a la lógica principal con la que funciona el juego, se omiten la especificación de algunas clases que provienen desde otros paquetes, así como todas las clases que heredan de “attackStrategy” para no sobrecargar el diagrama.

**¿Qué grandes retos de diseño enfrenta?**

Hoy en día, Pokémon es una franquicia que cuenta con 1015 especies diferentes de monstruos, los cuales son los personajes jugables, y 742 ataques, los cuales son realizados por los personajes. Incorporar esta cantidad tan alta de elementos únicos trae un gran reto al diseño, pues cada monstruo y ataque posee características únicas que no pueden ser implementados de manera sencilla en jerarquías de clases.

**Enlace del proyecto**

<https://github.com/keldaanInteractive/pokemonAutoChess>

**Información y estructura del fragmento del proyecto donde aparece el patrón**

Diagrama

Descripción generada automáticamente

*Imagen 2.1 Implementación del patrón*

El patrón aparece de manera general en la carpeta que se encuentra en la ruta “pokemonAutoChess/app/core” y en la carpeta de la ruta “app/types/enum/Ability.ts”. En particular, los archivos relevantes dentro del fragmento son:

* app/core/pokemon-entity.ts: Contiene la creación de los Pokémon, la cual hace uso del patrón.
* App/types/enum/Ability.ts : Es el encargado de asignar un ataque a un Pokémon rápidamente
* app/core/attack-strategy.ts: Contiene la mayor parte de la lógica del patrón:

**Información general sobre el patrón: qué patrón es y para qué se usa usualmente.**

El patrón strategy, también llamado policy, es una forma de organizar y estructurar el código de manera que se pueda seleccionar entre diferentes algoritmos para realizar una determinada tarea. En lugar de tener un único algoritmo implementado directamente en el código, se puede encapsular cada algoritmo en su propia clase y luego intercambiarlos según sea necesario.

El objetivo principal del patrón es permitir la flexibilidad y extensibilidad en el diseño de software. Al separar cada algoritmo en una clase separada, se puede agregar fácilmente nuevos algoritmos sin modificar el código existente. Además, se puede cambiar dinámicamente el comportamiento de un objeto seleccionando un algoritmo diferente en tiempo de ejecución.

La implementación básica del patrón presente en el libro de GoF es la siguiente:

Diagrama

Descripción generada automáticamente

*Imagen 3.1 Diagrama del patrón strategy*

Se declara una interfaz común para todos los algoritmos admitidos (Strategy). El Context utiliza esta interfaz para llamar al algoritmo definido por una ConcreteStrategy.

Las ConcreteStrategy implementan el algoritmo utilizando la interfaz Strategy.

El Context se configura con un objeto ConcreteStrategy. Mantiene una referencia a un objeto Strategy y puede definir una interfaz que permita que Strategy acceda a sus datos.

**Información del patrón aplicado al proyecto**

En este caso, la clase que cumple el papel de la clase “Strategy” del patrón es la clase AttackStrategy, la cual sirve como plantilla para otras clases que la extienden.



*Imagen 4.1 Clase principal*

Las clases que la extienden vendrían a ser el equivalente a las ConcreteStrategies del modelo original. El archivo que se encuentra en la ruta: app/core/attack-strategy.ts contiene la clase principal y todas las clases que la extienden, de las cuales solo se mostraran tres en este documento para fines prácticos.



*Imagen 4.2 primera Implementación*



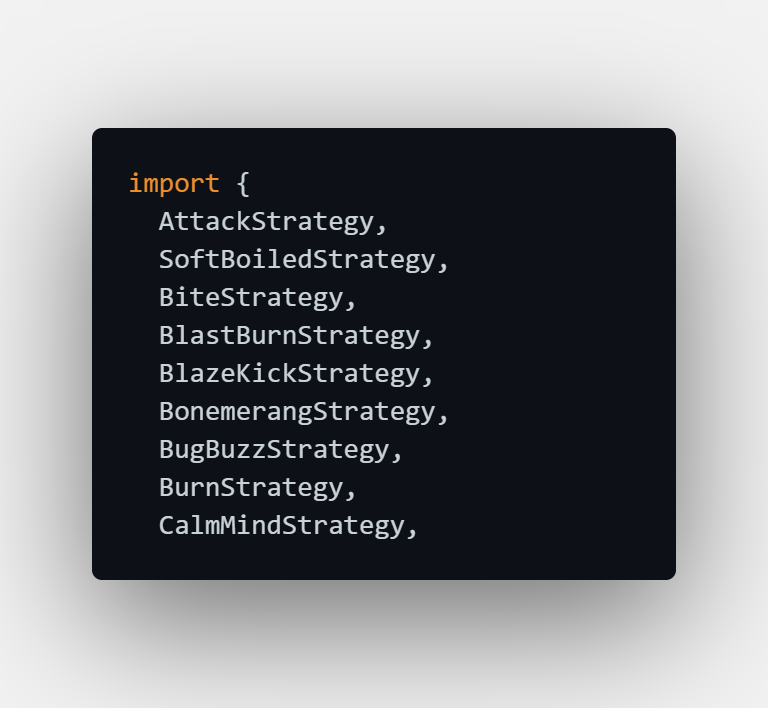
*Imagen 4.3 segunda Implementación*



**Imagen 4.4 tercera Implementación**

Se puede observar que las tres clases extienden la clase principal en el fragmento “extends AttackStrategy”

Seguidamente, la parte del proyecto que funge como el context se encuentra en la ruta “app/types/enum/Ability.ts”. Donde primero se importan todos los algoritmos, luego se hace una enumeración de estos y luego se crea un inicializador de objetos que contiene todos los algoritmos.



*Imagen 4.5 Línea 1 app/types/enum/Ability.ts. parte recortada del algoritmo*



*Imagen 4.6 Línea 142 app/types/enum/Ability.ts. parte recortada del algoritmo*

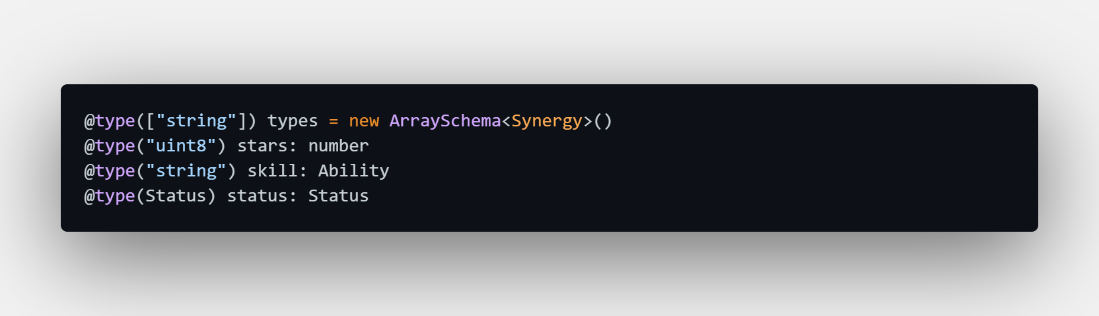


*Imagen 4.7 Línea 284 app/types/enum/Ability.ts. parte recortada del algoritmo*

Finalmente, el algoritmo correspondiente es asignado al Pokémon, del cual su clase se encuentra en app/core/pokemon-entity.ts, a través del Context, específicamente a través de AbilityStrategy, donde pokemon.skill es una instancia de la enumeración ability.



*Imagen 4.8 Línea 62 app/core/pokemon-entity.ts parte recortada del algoritmo*



*Imagen 4.9 Línea 53 app/core/pokemon-entity.ts parte recortada del algoritmo*



*Imagen 4.10 Línea 97 app/core/pokemon-entity.ts parte recortada del algoritmo*

**¿Por qué tiene sentido haber utilizado el patrón en ese punto del proyecto? ¿Qué ventajas tiene?**

Los monstruos en Pokémon cuentan con una inmensa cantidad de ataques posibles, cada uno con características únicas, algunos hacen más daño que otros, otros pueden causar disminución en los atributos de un enemigo o propios o cambiar el estado de otro Pokémon, confundiéndolo, quemándolo, congelándolo, etc. Toda esta variedad de movimiento hace razonable la implementación del patrón en este punto, pues, aunque es posible realizar el requerimiento sin usarlo, su uso hace que sea más fácil de escribir el código y hace que se pueda modificar más fácilmente en el futuro, lo cual es especialmente importante, pues con cada nueva entrega de la franquicia se implementan nuevos ataques, los cuales pueden ser implementados fácilmente usando el patrón.

**¿Qué desventajas tiene haber utilizado el patrón en ese punto del proyecto?**

La principal desventaja es la sobrecarga de memoria. Como se puede apreciar en la imagen 4.7, se crea una instancia para cada tipo de ataque y Pokémon, lo que conlleva a que casi todas las instancias no se usen y se desperdicie memoria.

**¿De qué otras formas se le ocurre que se podrían haber solucionado, en este caso particular, los problemas que resuelve el patrón?**

Claramente, el problema se hubiese podido solucionar utilizando varios condicionales para cada tipo de ataque y Pokémon. Sin embargo, esta sería una solución menos mantenible a largo plazo y más difícil de programar.

Cabe resaltar que el proyecto hace un buen uso de diferentes patrones de diseño además del Strategy, como lo es el patrón State (app/core/attacking-state.ts) y el patrón factory (app/models/pokemon-factory.ts), por lo que se puede apreciar que el proyecto busco desarrollarse bajo muy buenas prácticas, lo cual reduce la posibilidad de encontrar una mejor implementación de manera sencilla.

**Bibliografía**

Design Patterns Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides.