MÓDULO:

Desarrollo de Aplicaciones Web Entorno Cliente

2º DAW

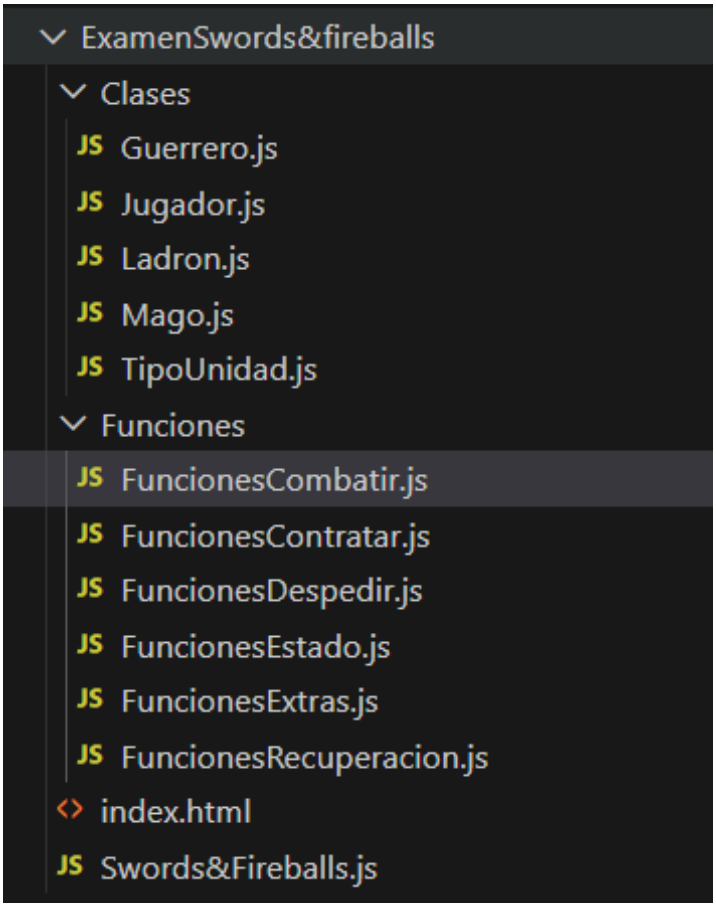
Autor/es:

Jessica Paola Hernández Rivera

INDICE

Estructura del codigo

Para poder organizarme mejor y poder llevar un orden de todos los ficheros que estaba utilizando, cree carpetas donde separo los contenidos. Tengo una carpeta donde están todas los ficheros con las funciones para cada caso específico, de manera adyacente, tengo otra carpeta donde se encuentran todas las clases que logre identificar y necesitar para poder realizar el juego. Y en este mismo nivel, se encuentra el fichero Swords&Fireballs que es donde se encuentra la lógica del juego junto al fichero index.html que es quien permite cargar el modulo del fichero anterior y así poder cargar el programa.



Swords&Fireballs

Este es el fichero principal, pues es quien carga toda la lógica del juego y la interacción con el usuario. Contiene las importaciones de cada fichero necesario, las validaciones y un switch, donde se maneja cada una de las acciones del case.

Me recomendaste el hacer este fichero como la clase Juego, lo pensé y con lo que ya tenía, saque ideas y apuntes para cambiarlo a una clase, pero la verdad me sentía mucho más cómoda con llevar el fichero sin necesidad de que fuera una clase y no me quería arriesgar a la hora de hacer modificaciones y que quizás algo me fallara.

Pero si lo tendré en cuenta, porque más adelante me gustaría poder mejorar la versión y poder implementar lo que tengo, para convertirlo en una clase.

```
//****************FUNCIONES DE ESTE FICHERO*********************//
// Si convertimos este fichero en clase, tener en cuenta:
// Tendriamos, un contructor donde inicializamos las variables.
// Una funcion iniciar juego, que esta funcion tendra llamada a las demas funciones, seria la funcion principal (con la llamada
// PedirDificultad -> Que devolveria la dificultad que ha elegido el usuario.
// Como et VictoriasParaGanar y let recuperacionTexto = jugador.getRecuperacion == true ? 'Si' : 'No'; lo hago a base de respue
// Una funcion MenuPrincipal() donde contenga el menu y a base de la respuesta haga cada una de las 5 opciones en el switch
// El switch del menu principal tendria que tener una llamada a cada una de las funciones de cada case.
// Cada case del switch se convertiria en una funcion aparte.
```

Lo que si hice, fue cambiar el switch con las opciones, inicialmente tenia un bloque de codigo grandísimo, y eso que solo tenia 2 case hechos, y seguí tu recomendación de meterlo en funciones, lo cual me aclaro muchísimo la mente y me ayudo a intentar organizarlo todo por medio de llamadas a funciones, lo cual ha sido una facilidad de poder ir creando los bloques de codigo e intentar hacerlo todo de manera más automatizada.

Por lo cual, el bloque que maneja las opciones de codigo en este fichero, queda así:

```
switch(accionJugador) {
    case 1:

        contratarTropas(jugador);

        break;
    case 2:

        if(jugador.getTropasJugador.length > 0) {
            despedirTropas(jugador);
        } else {
            alert(`No tienes unidades para despedir.`);
        }

        break;

    case 3:

        //PRUEBA jugador.getTropasJugador.forEach((tropa, indice) => tropa.setKO = true);
        //Si el jugador tiene ejercito y al menos una unidad disponible con > 0 PVs
        if(jugador.getTropasJugador.length != 0 && tieneUnidadesConVida(jugador.getTropasJugador)) {
            combatir(jugador);
        } else {
            alert(`Necesitas al menos una unidad con PVs > 0 para combatir`);
        }

        break;

    case 4:
```

```
            break;

        case 4:
            if ((jugador.getVictorias != 0 || jugador.getDerrotas != 0) && jugador.getUsoRecuperacio
                recuperacionTropas(jugador);

            } else {
                alert(`La recuperación solo está disponible después de un combate y una sola vez.`);
            }
            break;

        case 5:
            let mensajeEstado = mostrarEstado(jugador);
            alert(mensajeEstado);
            break;
        case 0:
            alert(`Hasta la proxima`);
            salir = true;
            break;

        default:
    }

}

// Una vez salga de este while de menu de acciones, es porque ha ganado, perdido o se ha quedado sin
if (jugador.getVictorias === VictoriasParaGanar) {
    alert(`Has ganado la partida`);
} else if (jugador.getDerrotas === DerrotasParaPerder) {
    alert(`La CPU ha ganado la partida`);
} else if (jugador.getIntentosContratacion === 0 && jugador.getTropasJugador.length == 0) {
    alert(`Has gastado todos los intentos y no tienes tropas. Creo que esto no es lo tuyo...`);
}
```

Clases

Con respecto a las clases, inicialmente tenia solo las clases de los personajes (Mago, Guerrero y Ladron), cada clase con sus respectivos atributos y metodos, pero mencionaste en clase del porque no hacerlo con Herencia si las unidades compartían atributos y acciones, entonces esa tarde empecé a implementar una cuarta clase, que era la clase "TipoUnidad", porque efectivamente es mucho mas optimo el tener una clase Padre y que las unidades hereden de esta clase, no me pareció complicado hacerlo de esta manera y me gustó muchísimo más, de manera que, si tengo que cambiar algo que tienen en común, simplemente tengo que modificar cosas desde la clase Padre.

Es verdad que las acciones que realizan como atacar, o recibir, se tienen que sobrescribir en esta clase, eso sí, dependiendo de la tropa, porque la clase Mago y la clase Guerrero tienen habilidades especiales a la hora de atacar, pero el ladrón tiene su habilidad especial cuando recibe el daño.

Clase Padre:

```
export class TipoUnidad {

    //Constructor
    // *Solo paso por parametro el nombre para diferenciarlos.
    // *La vidaMin y vidaMax, porque dependiendo de la unidad tienen rangos de vida diferentes.
    // *Al igual que la vida, cada uno tiene coste de contratacion y ganancia de retirarlos diferentes.
    // *El ataque lo incializo dentro del contructor, porque todos tienen un ataque entre 10 y 20.
    constructor (nombre, costeContratacion, gananciaRetirarlo, vidaMin, vidaMax) {
        this.nombre = nombre;
        this.ataque = Math.floor(Math.random() * (20 - 10 + 1)) + 10;
        this.costeContratacion = costeContratacion;
        this.gananciaRetirarlo = gananciaRetirarlo;
        this.puntosDeVidaMax = Math.floor(Math.random() * (vidaMax - vidaMin + 1)) + vidaMin; //Para poder generar la recuperacio
        this.puntosDeVida = this.puntosDeVidaMax; //Este si que puedo modificarlo.
        this.ko = false; //Significa que aun no ha sido derrotado
    }

    //***** Metodos de la clase *****//
    // Metodo Atacar, se sobreescribira en cada una de las clases hijas, dependiendo de sus capacidades especiales.
    atacar () {
        return this.ataque;
    }

    // Metodo ventajaTipo, este metodo lo heredan las hijas, pero es para que a la hora del combate saber si la unidad
    // que esta peleando tiene ventaja sobre la rival, me devuelve true si es asi y en el caso contrario false.
    ventajaTipo (nombreRival) {

        let tieneVentaja = false;

        let ventajas = new Map([['Mago', 'Guerrero'], ['Guerrero', 'Ladron'], ['Ladron', 'Mago']]);
```

A la hora de estar creando el combate, me surgieron muchas dudas y conflictos sobre como manejar las ventajas de tipo con respecto a las unidades, tenía claro que quería hacerlo con un mapa, ya que lo hice así en el ejercicio del piedra, papel o tijera y me parece una manera muy sencilla de controlarlo y manejarlo. Pero no estaba segura de si hacerlo en una funcion aparte o hacer que este método exista en la clase Padre, para poder controlarlo a partir de cada uno de los objetos y así poder manejar si aplicar el daño extra o no.

Al final he decidido colocar este método en la clase Padre, para poder controlarlo a partir del mismo objeto quien estaría llamando a este método, ya que al final cada objeto sabe su tipo por medio del nombre, entonces solo tengo que saber el del rival para comprobar si existe una ventaja de tipo. Aparte, como tengo que concatenar el mensaje "[Tipo de ventaja]" a la hora de mostrar la información al usuario, me resultaba mucho más útil que con un true poder agregar el mensaje si existe, que tener que estar comprobando o pasando dos tropas diferentes en una funcion externa.

Aquí es donde entran estos 2 metodos en la clase Padre:

```
// Metodo ventajaTipo, este metodo lo heredan las hijas, pero es para que a la hora del combate saber si la unidad
// que esta peleando tiene ventaja sobre la rival, me devuelve true si es asi y en el caso contrario false.
ventajaTipo (nombreRival) {

    let tieneVentaja = false;

    let ventajas = new Map([['Mago', 'Guerrero'], ['Guerrero', 'Ladron'], ['Ladron', 'Mago']]);

    if(ventajas.get(this.getNombre) === nombreRival) {
        tieneVentaja = true;
    }

    return tieneVentaja;

}

// Ahora, a base del metodo anterior, si nos devuelve true, al daño base de la tropa aplicamos
// lo multiplicamos por 1.5 de daño.
calcularDañoVentaja (dañoBase, nombreRival) {

    if (this.ventajaTipo(nombreRival)) {
        dañoBase *= 1.5;
    }

    return dañoBase;
}
```

Así a la hora del combate, puedo preguntar con respecto a la tropa que esta atacando o viceversa, ¿Tengo ventaja sobre la tropa rival? De ser cierto, en la siguiente funcion (que utiliza la funcion ventajaTipo para verificar) agrega 1.5 de daño y devuelve ese daño alterado. De ser falso simplemente devolverá el daño que le hemos pasado por parametro, por cual a la hora del combate

tengo una variable donde diga, si es cierto entonces pon este mensaje y si no, déjalo vacío y no tengo que estarme preocupando cuando debo de poner el mensaje manualmente.

Las clases hijas (Mago, Guerrero y Ladron), no tienen más que el constructor con sus respectivas características y sus habilidades especiales, sobrescribiendo los metodos que sean necesarios. La clase Padre contiene todos los Getters y Setters de los atributos, por lo tanto sus hijas tambien lo tienen.

Clase Jugador.

Con respecto a esta clase, me pareció lógico implementarla, mas que nada para poder guardar cada atributo del usuario y/o porque a lo mejor, en futuras versiones no solo estaríamos trabajando con un jugador, o controlando la lógica de un jugador, podríamos tener más. Es esta clase no contiene mas que los atributos como oro, victorias, derrotas, etc..

Lo que sí, tiene su propio array donde se guardan cada una de las tropas que decida tener. Inicialmente no sabia muy bien si sería una buena opción crear una clase aparte para guardar las tropas, pero es verdad que lo mencionaste que era mucho más lógico trabajar sobre un array, así que decidí dejarlo así, porque efectivamente tiene muchísima mas lógica. Este clase, al igual que todas, tienen sus respectivos Getters y Setters.

```
export class Jugador {

    //Variables del usuario.
    //let recuperacionTexto = recuperacion == true ? 'Si' : 'No';

    //Contructor
    constructor () {
        this.oroJugador = 5000;
        this.victorias = 0;
        this.derrotas = 0;
        this.intentosContratar = 6;
        this.recuperacion = false;
        this.tropasJugador = [];
        this.maxTropas = 5;
        this.usoRecuperacion = false; //No podra hacer uso de la recuperacion a menos que haya hech
    }

    //GETTERS
    get getOroJugador() {
        return this.oroJugador;
    }

    get getVictorias() {
        return this.victorias;
    }

    get getDerrotas() {
        return this.derrotas;
    }

    get getIntentosContratacion () {
        return this.intentosContratar;
    }
```

Funciones

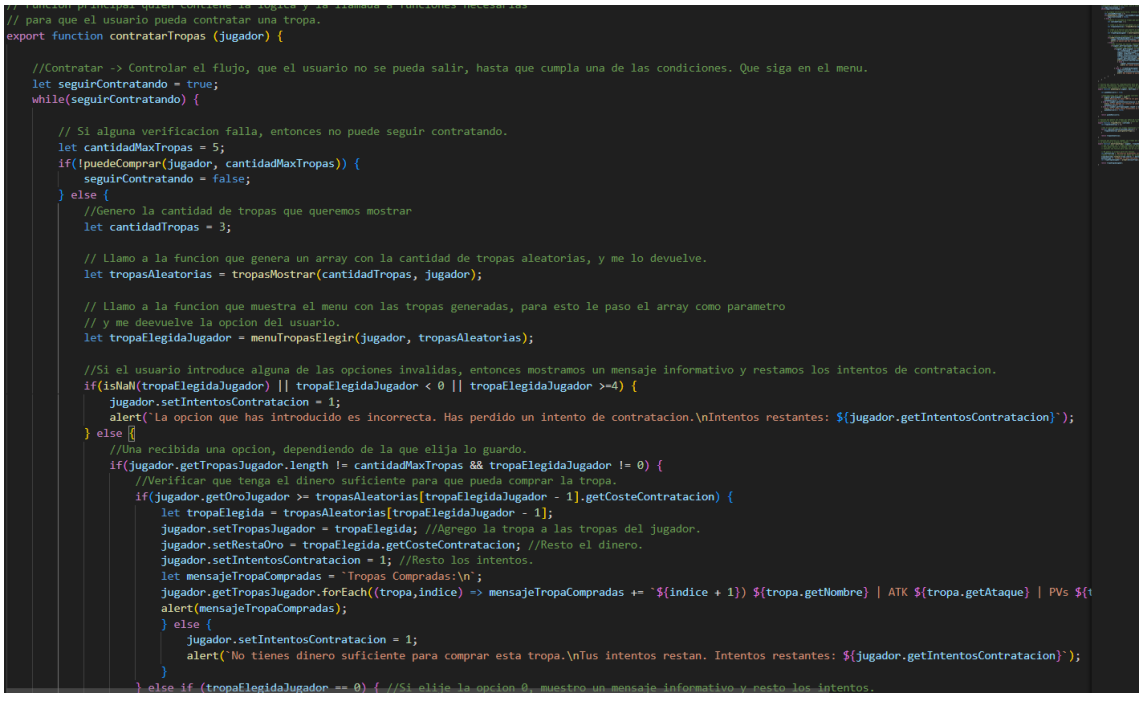
En esta carpeta es donde realmente se encuentra una cantidad de codigo. Al darme cuenta de que cada case más que todo el de combatir, tenia una cantidad de comprobaciones y acciones decidí crear un fichero con funciones especificas para cada caso. Deje el fichero “Funciones Extras” que es donde inicialmente tenia todas las funciones, pero repito, eran una cantidad de funciones y líneas de codigo que opte por separarlas dependiendo del case.

Nose si sea la mejor opción porque es verdad que a la hora de mostrar los mensajes y estar mostrando las tropas he repetido varias veces ese mismo proceso, pero es verdad que a la hora de estar enfocado en un caso, se te pueden escapar ciertos detalles.

Funciones Contratar

En cada fichero de funciones, existe la función principal que es quien se encarga de manejar toda la lógica de la opción en concreto, en este caso contratar. He separado trozos y creado funciones externas (pero en el mismo fichero) para en la funcion principal poder llamarlas, porque me parece mas ordenado y me entiendo mucho mejor, me cuesta menos leerlo.

La funcion principal es esta:



```
// para que el usuario pueda contratar una tropa.
export function contratarTropas (jugador) {

    //Contratar -> Controlar el flujo, que el usuario no se pueda salir, hasta que cumpla una de las condiciones. Que siga en el menu.
    let seguirContratando = true;
    while(seguirContratando) {

        // Si alguna verificacion falla, entonces no puede seguir contratando.
        let cantidadMaxTropas = 5;
        if(!puedeComprar(jugador, cantidadMaxTropas)) {
            seguirContratando = false;
        } else {
            //Genero la cantidad de tropas que queremos mostrar
            let cantidadTropas = 3;

            // Llamo a la funcion que genera un array con la cantidad de tropas aleatorias, y me lo devuelve.
            let tropasAleatorias = tropasMostrar(cantidadTropas, jugador);

            // Llamo a la funcion que muestra el menu con las tropas generadas, para esto le paso el array como parametro
            // y me deevuelve la opcion del usuario.
            let tropaElegidaJugador = menuTropasElegir(jugador, tropasAleatorias);

            //Si el usuario introduce alguna de las opciones invalidas, entonces mostramos un mensaje informativo y restamos los intentos de contratacion.
            if(isNaN(tropaElegidaJugador) || tropaElegidaJugador < 0 || tropaElegidaJugador >=4) {
                jugador.setIntentosContratacion = 1;
                alert(`La opcion que has introducido es incorrecta. Has perdido un intento de contratacion.\nIntentos restantes: ${jugador.getIntentosContratacion}`);
            } else {
                //Una recibida una opcion, dependiendo de la que elija lo guardo.
                if(jugador.getTropasJugador.length != cantidadMaxTropas && tropaElegidaJugador != 0) {
                    //Verificar que tenga el dinero suficiente para que pueda comprar la tropa.
                    if(jugador.getOroJugador >= tropasAleatorias[tropaElegidaJugador - 1].getCosteContratacion) {
                        let tropaElegida = tropasAleatorias[tropaElegidaJugador - 1];
                        jugador.setTropasJugador = tropaElegida; //Agrego la tropa a las tropas del jugador.
                        jugador.setRestaOro = tropaElegida.getCosteContratacion; //Resto el dinero.
                        jugador.setIntentosContratacion = 1; //Resto los intentos.
                        let mensajeTropaCompradas = `Tropas Compradas:\n`;
                        jugador.getTropasJugador.forEach((tropa,indice) => mensajeTropaCompradas += `${indice + 1}) ${tropa.getNombre} | ATK ${tropa.getAtaque} | PVs ${t
                        alert(mensajeTropaCompradas);
                        } else {
                            jugador.setIntentosContratacion = 1;
                            alert(`No tienes dinero suficiente para comprar esta tropa.\nTus intentos restan. Intentos restantes: ${jugador.getIntentosContratacion}`);
                        }
                    } else if (tropaElegidaJugador == 0) { //Si elije la opcion 0, muestro un mensaje informativo y resto los intentos.
```

Se que es larga, pero he separado aun así, cierta parte del codigo que me vendría mejor solo llamarla o guardarla en una variable dependiendo a lo recibe de ella, como por ejemplo, aparte tengo una funcion que me verifica si el usuario puede comprar, si me devuelve true, entonces ejecuto todo el codigo que se encuentra en la funcion contratar.

Luego tengo otra opción que me genera un array con tropas aleatorias dependiendo de la cantidad que le pase como parametro y me lo devuelve, y por ultima la funcion menuTropasElegir(), que se encarga de mostrar al usuario un mensaje con las tropas que se generaron aleatoriamente y me devuelve la respuesta del jugador.

A la hora de generar la funcion:

```
// Funcion que genera las tropas por medio de la funcion generarTropas() mas arriba, pero
// devuelve un array con la cantidad de tropas que pasemos como parametro.
export function tropasMostrar (cantidad) {
    let tropasAleatorias = [];

    //Con el buclo genero las tropas aleatorias y las guardo en el array.
    for(let indice=0;indice<cantidad;indice++) {
        tropasAleatorias.push(generarTropa());
    }

    return tropasAleatorias;
}
```

No estaba segura de si reutilizar una funcion que tenia en el fichero “FuncionesExtras”, que te genera una tropa aleatoriamente y te la devuelve, que es muy parecida a esta con la diferencia que en lugar de devolverte una tropa, te devuelve un array, pero preferí dejarlo así y utilizar esa funcion para ir guardando la tropa en el array, por si en algún caso específico necesitaba crear solo 1 tropa en concreto.

```
export function generarTropa () { //Devuelve 1 tropa por cada vez que se llama.

    let probabilidad = Math.floor(Math.random() * 100) + 1; //Probabilidad entre 0 y 1.
    if(probabilidad <=20) {
        let tropa1 = new Mago ();
        return tropa1;
    } else if (probabilidad > 20 && probabilidad <=50) {
        let tropa2 = new Ladron();
        return tropa2;
    } else {
        let tropa3 = new Guerrero();
        return tropa3;
    }
}
```

Funciones Despedir

Realizar este caso fue muy sencillo, al principio me estaba complicando un poco sobre como manejar cuando el usuario quiera eliminar alguna tropa que se encuentre en un índice de en medio, que se puede hacer, pero es verdad que en algún momento me podría arriesgar a que no se borrara o no lo reorganizara de manera correcta. Empece a buscar funciones en las presentaciones y encontré la de splice, que te reorganiza el array de manera automáticamente y me parece increíble. Simplemente que como esta funcion te devuelve un array, tienes que acceder por medio del índice 0 para tener la tropa que ha sido despedida, pero con esto, implementar este case fue mucho más sencillo.

En este fichero lo he separado en dos funciones, uno donde se encarga de mostrar el mensaje con sus tropas y controla las validaciones con respecto a la respuesta del usuario y otra, donde utiliza esta funcion y dependiendo de la tropa que quiera borrar el usuario, especifico el índice y la cantidad de tropas que quiero borrar.

```
export function despedirTropas(jugador) {

    let respuestaDespedir = menuDespedir(jugador);
    let tropaDespedida = [];

    if(respuestaDespedir != 0) {
        //Una vez hecha las comprobaciones, procedo a eliminar las tropas.
        tropaDespedida = jugador.getTropasJugador.splice(respuestaDespedir - 1, 1); //Lo elimino y lo guardo. //Splice me devuelve un a
        alert(tropaDespedida[0]);
        //Sumo el oro al jugador
        jugador.setSumaOro = tropaDespedida[0].getRetirarlo;
        alert(`Unidad retirada. Recuperas ${tropaDespedida[0].getRetirarlo} oro. Oro: ${jugador.getOroJugador}`);
    }


    return tropaDespedida;
}

// Funcion que muestra el menu al usuario de las tropas que tiene
// para que pueda despedir alguna de elllas.
// Devuelve la respuesta del jugador
export function menuDespedir(jugador) {

    let mensajeDespedir = `Elige índice para despedir (oro: ${jugador.getOroJugador})\n`;
    //Ahora al mensaje agrego la informacion de las tropas
    jugador.getTropasJugador.forEach((tropa,indice) => mensajeDespedir += `#${indice + 1}: ${tropa.getNombre} (ATK ${tropa.getAtaque} P

    //A base de esa cantidad, agrego al mensaje por medio de un for, las unidades vacias.
    for(let unidadVacia=jugador.getTropasJugador.length + 1;unidadVacia<=5;unidadVacia++) {
        mensajeDespedir += `#${unidadVacia}: [vacío]\n`;
    }

    //Y por ultimo concateno la ultima linea del mensaje.
    mensajeDespedir += `0 para cancelar:`;

    let respuestaDespedir = parseInt(prompt(mensajeDespedir));
```

Funciones Combatir.

Este caso si que me mantuvo muy entretenida. Me costó mucho más que nada porque al tener tantas comprobaciones por controlar y tantos mensajes por mostrar de manera correcta que no sabia ni por donde empezar. Intente hacerlo de tirón, separando cada cosa en funciones pero la verdad es que no me salía y me daba muchos errores, así implemente todo el codigo sin usar funciones ni pensar en alguna manera de reducir el codigo, porque necesitaba hacerlo así, siguiendo la lógica del combate para lograr tenerlo. Sabia que tenia que usar funciones, porque una vez que lo termine había mucho codigo repetido.

Lo que mas me costo de este bloque fue mostrar los mensajes correctamente. Me paso que con el ladrón a veces el mensaje de esquiva aparecía pero en el lado contrario, en el lado del rival. Como había hecho todo de golpe sin pruebas, no había caido que como el Ladron es un caso especial porque no tiene una habilidad al atacar, si no al recibir, entonces yo estaba generando el mensaje y colocándolo de manera incorrecta.

En verdad, las funciones como some(), find() y forEach() me parecen totalmente increíbles y las he utilizado muchísimo en el codigo, me han facilitado mucho el poder crear esta acción.

En la funcion principal primero genero las tropas aleatorias de la CPU, como en el fichero de la funcion contratar tengo una funcion que dependiendo la cantidad que le pase por parametro me devuelve un array con tropas aleatorias, entonces la vuelvo a reutilizar.

```
// Funcion que generara el ejercito de la CPU utilizaremos la funcion en el fichero FuncionesContratar,
// ya que genera un array con objetos de Mago, Guerrero o Ladron dependiendo de la cantidad.
export function generaTropasCPU () {

    let rangoMin = 3;
    let rangoMax = 5;
    let cantidad = Math.floor(Math.random() * (rangoMax - rangoMin + 1)) + rangoMin;

    let tropasAleatoriasCPU = tropasMostrar(cantidad);

    return tropasAleatoriasCPU;
}
```

Luego tengo una funcion en ese mismo fichero que la utilizo para poder mostrarle al usuario las tropas que tiene para combatir.

```
// Esta funcion mostrara al usuario las unidades que tiene disponible para conbatir.
export function muestraTropasCombatir (jugador, tropasAleatoriasCPU) {

    //Primera parte del mensaje
    let mensajeCombatir = `Vas a combatir. Tus unidades disponibles:\n`;

    jugador.getTropasJugador.forEach((tropa, indice) => mensajeCombatir += `${indice + 1}) ${tropa.getNombre} ATK ${tropa.getAtaque} PVs ${tropa.g

    //Se tiene que pasar la informacion que tiene sobre las tropas de la CPU para concatenarlo al mensaje.
    mensajeCombatir += `\nLa CPU tiene ${tropasAleatoriasCPU.length} unidades`;
    alert(mensajeCombatir);

    // Le muestro las tropas de jugador //PRUEBA PARA VER LAS TROPAS RIVALES.
    let muestraTropasCPU = `Unidades CPU a combatir:\n`;
    tropasAleatoriasCPU.forEach((tropa, indice) => muestraTropasCPU += `${indice + 1}) ${tropa.getNombre} ATK ${tropa.getAtaque} PVs ${tropa.getPu
    alert(muestraTropasCPU);
}
```

Ahora, tengo un while donde comprueba y dice, mientras ambos tengan, al menos una unidad con vida, entonces se realiza el combate. Para poder hacer esto, hice una funcion aparte, que por medio de la funcion some(), me devuelve true si al menos una de las tropas del array tiene el atributo KO a false y así es como controlo este bucle.

```
// Funcion que compruebe si el jugador tiene al menos una unidad con vida para poder combatir.
export function tieneUnidadesConVida (tropasArray) {

    // Mientras que el usuario tenga una unidad disponible sin estar KO, entonces
    // utilizo el metodo some, que me dice si algun objeto del array cumple con la condicion entonces
    // puede combatir. Si necesitara que todos y cada uno tienen que estar sin KO lo cambio
    // por la funcion every().
    let tieneVida = tropasArray.some((tropa) => tropa.getKO != true);

    return tieneVida;
}
```

Ahora, una vez dentro del bucle, por medio de la funcion find(), me devuelve el primer objeto que encuentre que cumpla la condición, en este caso que el atributo KO se encuentre en false. Saco la tropa del jugador, la tropa del CPU y a combatir.

```
// Seleccionamos la primer unidad viva del jugador y la primer unidad viva de la CPU.
// El metodo find() nos devuelve la primer ocurrencia que cumple la condicion que le especificamos
// Entonces nos devuelve la primer tropa que encuentra que no esta en KO.
let tropaJugador = jugador.getTropasJugador.find((tropa) => tropa.getKO  == false);
let tropaCPU = tropasAleatoriasCPU.find((tropa) => tropa.getKO == false);
let numTurnos = 1;
let turnoGanado = '';
```

Ahora aquí fui generando poco a poco los bloques, como mencione, inicialmente escribí todas las líneas de una, sin importar que se repitiera codigo, pero luego logre meter poco a poco todo en funciones.

Tengo funciones que me generan el mensaje de la habilidad al atacar, en el caso de que la tenga disponible.

```
// Funcion que devuelve el mensaje de la habilidad especial
// Recibe como parametro la tropa y devuelve su respectivo mensaje.
export function habilidadAtacarMensaje (tropa) {

    let mensajeHabilidadEspecial = '';

    // Si es Mago o Guerrero, al atacar tienen una habilidad especial que queremos informar al jugador.
    if(tropa.getNombre === "Mago" && tropa.getCuantaHabilidadEspecial > 0) {
        mensajeHabilidadEspecial = ` (${tropa.getNombreHabilidad})`;
    } else if (tropa.getNombre === "Guerrero" && tropa.getCuantaHabilidadEspecial > 0) {
        mensajeHabilidadEspecial = ` (${tropa.getNombreHabilidad})`;
    }

    return mensajeHabilidadEspecial;
}
```

Variables que me guardan el mensaje de “[Ventaja de Tipo]”, en el caso de que exista, funciones que me devuelven un daño final, en el caso de que haya una ventaja de tipo o este activa una habilidad especial, tengo varias funciones que me generan un mensaje en concretos, porque quería que el combate fuera muy informativo tal cual lo describiste y por ultimo tengo funciones donde manejo cuando una tropa ataca a otra, pero además, aquí es donde manejo el caso especial del ladrón, donde si la tropa a dañar es un Ladron, verifique si tiene la habilidad disponible activa y si ha salido la probabilidad de que la use, de ser así el mensaje se guarda con “¡Esquivado!”, si no, se queda vacío, en cualquiera de los dos casos se devuelve la variable con el mensaje.

```
// Funcion que realiza el ataque por medio del daño que se pasa como parametro a la tropa enemiga.
// Pero, tenemos que verificar si la tropa Rival es un ladron, de ser asi, tengo que verificar si ha
// tenido la posibilidad de esquivar el daño y devolver un mensaje con esta informacion.
export function realizarAtaque (tropaDañar, dañoHacer) {

    let mensajeEsquiva = "";

    if (tropaDañar.getNombre === "Ladron") {
        let dañoAntes = tropaDañar.getPuntosVida;
        tropaDañar.recibirDaño(dañoHacer);
        let dañoDespues = tropaDañar.getPuntosVida;

        if(dañoAntes == dañoDespues) {
            mensajeEsquiva= " ¡Esquivado!";
        }
    } else {
        tropaDañar.recibirDaño(dañoHacer);
    }

    return mensajeEsquiva;

}
```

Organizar el codigo de esta manera me ha ayudado mas que nada porque tanto el jugador, como la CPU realizan las mismas acciones y generan los mismo tipos de mensaje, así como tambien, en los dos casos pueden existir ladrones, así que se debe de manejar esta lógica. Yo me genere las variables necesarias, separadas, con cada tipo de mensaje informativo, si existe pues guardar el mensaje a mostrar y si no, esta vacío, así con solo concatenarlo todo a la hora de mostrar el mensaje final del jugador, tendrán que salir los mensajes correspondientes.

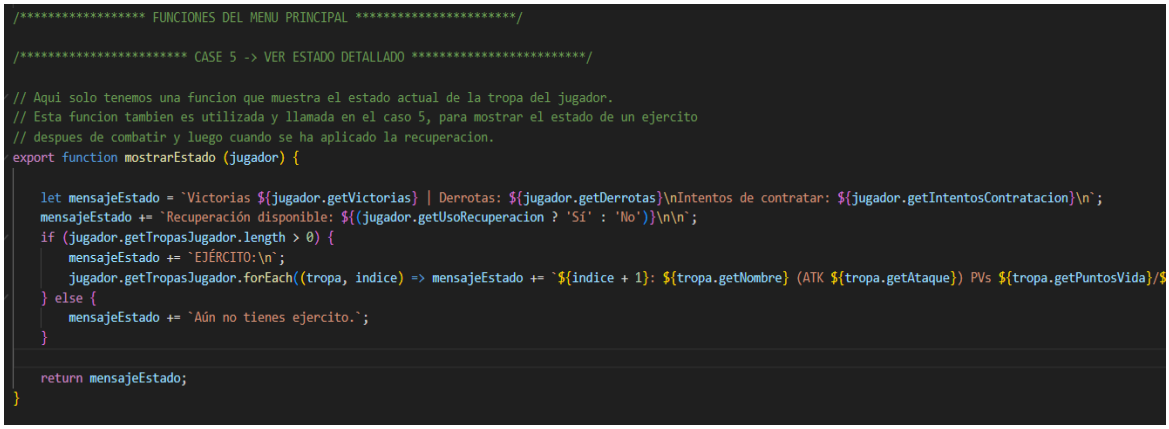
Funciones Recuperación

Este caso fue muy sencillo, desde un principio había generado en la clase Padre la existencia de esta acción y lo único que tuve que hacer es en las clases hijas sobrescribir el método, llamar al método de la clase Padre porque quería utilizar lo que estaba escrito en el, que es curar el 70% de la vida de las tropas, pero en cada una de las clases tenia que restablecer las habilidades especiales, entonces al tener ya esto controlado desde la clase Padre, lo único que tuve que hacer es utilizar la funcion forEach() y especificar que para cada tropa que recorra, llame al método de recuperar.

```
// Funcion principal que tendra toda la logica del caso de Recuperacion
export function recuperacionTropas (jugador) {

    let mensajeRecuperacion = `Tu compañia descansa: +70% vida a todos y habilidades especiales restauradas.`;

    // Llamamos a la funcion para ver el estado antes de que se restauren.
    let resumenTropas = mostrarEstado (jugador);
    alert(resumenTropas);

    // Mostramos la informacion al usuario
    alert(mensajeRecuperacion);

    // Y por medio de la funcion forEach hacemos que cada una las tropas
    // accedan a la funcion que heredan de la clase padre, para que puedan recuperar
    // un 70% de vida.
    jugador.getTropasJugador.forEach((tropa) => tropa.recuperarse());
    // Una vez dentro, volvemos a poner el uso de Recuperacion en false, despues de haber mostrado el mensaje.
    jugador.setUsoRecuperacion = false;

    // Mostramos un mensaje para ver el cambio
    jugador.restaurarIntentosContratacion = 6; //Volver a poner los intentos a 6.
    let tropasDespues = mostrarEstado(jugador);
    alert(tropasDespues);

}
```

Funciones Mostrar Estado

Este fue otro caso que no fue para nada complicado, era simplemente mostrar las tropas que tenia el usuario y su estado, en el caso que no tiene ejército, he implementado que le informe al usuario que aun no existe un ejército. Esta funcion la utiliza en el apartado anterior, para mostrar el antes y el después de las tropas al aplicar la recuperación.



```
/****************** FUNCIONES DEL MENU PRINCIPAL *********************/

/************************ CASE 5 -> VER ESTADO DETALLADO ************************/

// Aqui solo tenemos una funcion que muestra el estado actual de la tropa del jugador.
// Esta funcion tambien es utilizada y llamada en el caso 5, para mostrar el estado de un ejercito
// despues de combatir y luego cuando se ha aplicado la recuperacion.
export function mostrarEstado (jugador) {

    let mensajeEstado = `Victorias ${jugador.getVictorias} | Derrotas: ${jugador.getDerrotas}\nIntentos de contratar: ${jugador.getIntentosContratacion}\n`;
    mensajeEstado += `Recuperación disponible: ${(jugador.getUsoRecuperacion ? 'Sí' : 'No')}\n\n`;
    if (jugador.getTropasJugador.length > 0) {
        mensajeEstado += `EJÉRCITO:\n`;
        jugador.getTropasJugador.forEach((tropa, indice) => mensajeEstado += `${indice + 1}: ${tropa.getNombre} (ATK ${tropa.getAtaque}) PVs ${tropa.getPuntosVida}/$
    } else {
        mensajeEstado += `Aún no tienes ejercito.`;
    }

    return mensajeEstado;
}
```

Conclusiones

Para concluir, ha sido un ejercicio muy entretenido de desarrollar, he aprendido a usar e implementar funciones que no conocía y que me parecen muy útiles e interesantes, así como siempre intentar mantener todo automatizado y escalado por que no sabremos si el día de mañana las condiciones pueden cambiar, por lo cual esto me ha ayudado a tenerlo muy en cuenta.

He dejado implementado algunos mensajes extras, mas que todo para informarme de lo que estaba haciendo y como se estaba generando el mensaje y que a la hora de estar testeando (más que todo el combate) funcionara todo correctamente.

Tuve muy en cuenta las recomendaciones que mencionaste en clase y que me diste para mejorar el codigo, la cual me han ayudado a poder llevar a cabo el proyecto de manera mas organizada y legible.