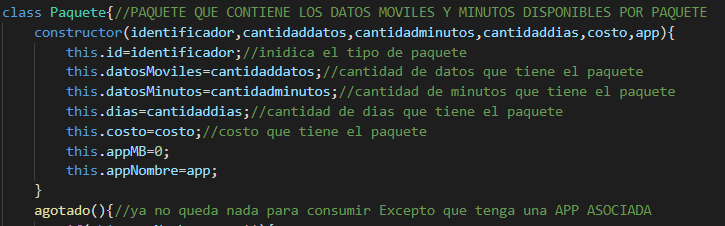
POO2-Andre Cueva

Trabajo final

**Conceptos aplicados:**

Aplique JAVASCRIPT con objetos utilizando **class NombreObjeto, los cuales tendrán un estado y un comportamiento**.

Los objetos tiene un constructor, los cuales se usan para crear e inicializar el objeto creado a partir de una clase.



El Paquete tiene un constructor, únicamente en caso de tener uno mas se lanza un error. También tienen métodos que pueden acceder al id del objeto usando **this.id** y modificarlo pero también pueden realizar una devolución usando return.Todo depende de que realice el método que implementes.

Los objetos creados son:

const { Compania, Paquete, Cliente, Cuenta, SistemaExterno, HistorialDeConsumos, ContratarPaquete, ReglasContratarPaquete, ClienteSaldoSuficiente, ClienteSinPaqueteContratado } = require('../src/ejemploAndre');

require() busca el archivo que esta entre paréntesis, en este caso esta trayendo los objetos del archivo ejemploAndre.js.

module.exports = {//INDICA QUE CLASES SERAN EXPORTADAS PARA SER USADAS EN TESTS POR EJEMPLO

    Compania, Paquete, Cliente, Cuenta, SistemaExterno, HistorialDeConsumos, ContratarPaquete, ReglasContratarPaquete, ClienteSaldoSuficiente, ClienteSinPaqueteContratado

  };

Module.exports ={}; exporta funciones,modelo,etc.

suite('suite de ejemplo de Compania Telefonica', () => {

Suite agrupa tests, lo que esta entre comillas es su nombre y ()=> indica inicio de sesión

Dentro de suite puede definir varios test.

Antes que todo debemos usar ciertos comandos para correr los test:

Npm install

Npm run test

**También ubicarnos en la carpeta donde estén los tests**

**Test** son situaciones que planteamos para poner a prueba el modelo en este caso CompaniaTelefonica, y ir analizando situaciones distintas.

Un test cuando aparece por consola [ERROR] significa su assert fallo, con lo cual puede ser porque algo este mal implementado ,mal escrito o variables no declaradas. Si sale [WIP] se refiere a un test que no está sin definir (no hace nada).

Declaraciones de **variables** utilice:

Var a =5; se puede cambiar el valor del mismo.

Const valor=20; no se puede cambiar el valor.

Let c=45; se puede cambiar su valor.

Utilice **throw ‘inesperado’** [error ] en algo que no se esperaba entonces lanzas un throw en ese caso, lo podes usar para correr test que sabes que son falsos y comprobar que lanzo un throw en ese caso.

Los **assert** que utilize fueron:

Assert.isTrue(a) indicando que a es VERDADERO

Assert.isFalse(c) indicando que c es FALSO

Assert.that(a).isEqualTo(b) indicando que a es igual que b.

Assert.that(a).raises(‘no se puede arreglar’) indicando que se lanzo un throw(‘no se puede arreglar’)

Assert.that(a).doesNotRaise(‘no puede hacerlo’) indicando que no se lanzo un throw(‘no puede hacerlo’)

El **before()** hace todo lo que este dentro suyo, por cada test que pasa mantenga su valor con el que fue instanciado en inicialmente.

Para indicar que cierta variable sea un objeto utlizaremos new NombreObjeto()

telefonica=new Compania();

En el caso de la compañía no coloque parámetros al constructor.

Al cliente de la compañía si tiene parámetros:

cliente=new Cliente(222,'alejandra');

Como cliente ahora es un objeto Cliente esta indicando que tiene el numero 222 y su nombre es Alejandra.

class Cliente{//CLIENTE ES LA PERSON QUE PUEDE TENER UN NUMERO Y UNA CUENTA, DEBIDO A QUE ES UN TELEFONO PREPAGO

    constructor(telefono,nombre){

        this.nombre=nombre;            //NOMBRE DEL CLIENTE

        this.numeroTelefono=telefono;//NUMERO DEL CLIENTE

        this.cuenta=new Cuenta();//CUENTA DEL CLIENTE

    }

Lo que vemos en la imagen anterior es que el constructor de Cliente recibe un teléfono y nombre. Los cuales son los que se le asignaron a la variable **cliente.**

También utilice metodo **filter(),** el cual crea un array con los elementos que cumplan con la condición que pida por parametro, devolviendo un array con esa condición:

**words.filter(word => word.length > 6)**

También se utilizo el método **sort(),** que ordena segundo lo que se pida:

**arr.sort([compareFunction])**

Lo que utilice en varias situaciones es el array:

this.reglasParaObtenerPaquete=[];

El cual me permite tener una lista de x elementos.Esto me permitió usar filter() y sort() en los casos que tenia que ver los consumos realizados en ciertas fechas usando un SISTEMAEXTERNO, que es quien tiene los consumos y la fecha inicial y final del mismo consumo hecho.

Algo también pero no menos importante fue el uso de **if else**, los cuales se trataron de evitar su uso a toda costa, porque implicaría una sobrecarga de if innecesarios. Estamos trabajando con objetos entonces porque no crear métodos que eviten que usemos if. Esto implica el uso de polimorfismo y mejorar el modelo. Entonces siempre que podía usar if, pensaba alguna manera de usar polimorfismo y así evitar el uso de if. Esto permite crear el código mas eficiente y optimo.

If(condicion)

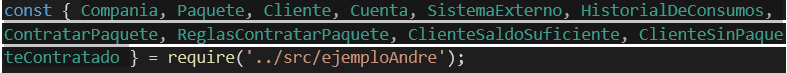
{}

Else{}

Al **if** le indico la condición y en el caso de ser verdadera realiza lo que este entre {} caso contrario usa lo que esta entre {} del **else.**

**Evolución Trabajo**

**A**l principio se me dificulto algunas clases porque no sabia si eran realmente necesarias. Otras era bastante visibles respecto al enunciado. Al final realice el uso de las siguientes:



Interprete que una Compania tiene ciertos números y estos están asociados a una persona. Pero esta persona tiene que ser cliente y para que sea cliente tiene que tener dinero en una cuenta que le permite la compra de algún paquete disponible por la empresa.

Con lo cual un cliente tiene asociado una cuenta, para ello coloque cuatro tipos de paquetes, una cuenta vacia que será cuentaCero, la cual revisa las reglas y se agrega con regla inicial. Luego se cargaSaldo a la cuenta del cliente. La cuenta indica únicamente que id de paquete tiene asociado y la cantidad de dinero que tiene el cliente. Una vez cargado dinero puede contratarPaquete si es que no tiene algún paquete o si tiene saldo suficiente para hacerlo. Luego telefónica usa descontar para quitar los consumos realizados por el cliente al paquete obtenido. Puede ser que en algún momento no pueda consumir mas en ese caso esta agotado el paquete.agotado() también puede ser que no puedas usar mas el paquete debido a que el tiempo de uso termino entonces paquete.vencido().Tambien hay una lista de los historiales de consumos realizados por los números telefónicos.

Me toco agregar una app con Mb ilimitados, lo que hice fue cada vez que hay un consumo llame a telefónica.descontar() y le digo que cantidad consumio con no hay limite directamente se agrega cuanto consumio únicamente. Y también hacer refactoring de agotado debido a que no importa que se acabe los MB y minutos podes usar la app aun.

Lo primero que fui haciendo fue un modelo las clases que podría usar y luego que termine las clases que usaría, fui pensando tests simples y cada vez un poco mas abarcativos. Me ocurrió varias veces que lo que colocaba en los tests era hacer únicamente verificaciones y no la compra de paquetes por lo tanto lo que mas me costo fue pensar en tests. Luego de hacer varios test, me di cuenta que tenia que cambiar cosas del modelo y las fui cambiando a medida que hacia los test y al veces otros test se rompían. Esto hacia que tenga que pensar denuevo como realice el modelo y hizo que me llevara bastante tiempo.

**Conclusión:**

La verdad me gusto mucho el realizar un proyecto y usar polimorfismo, fue la primera vez que lo implementamos en la carrera. También tuve que hacer bastante refactoring durante todo el proyecto, lo que al principio me molesto pero al final, me di cuenta que es para hacer un mejor código. La verdad fue un interesante desafio y complejo. Me gusto mucho el uso de test para comprobar el modelo y el uso del mismo pero me costo mucho entenderlos y pensarlos. Una de las cosas que también me costo pero nose si viene al tema, es los nombres de los tests. La verdad costo muchísimo pensar los nombres y al veces me parecía que repetía. Pero a pesar de todo fue una experiencia única e interesante.