Certified Tech Developer
The Ultimate Degree

Mejora contínua y refactorización

Todos estos conceptos conviven a diario con las personas que programan. La buena calidad es el resultado de iterar sobre lo mismo buscando mejoras hasta no encontrarlas más. Si bien eso podría no terminar nunca, es importante para no crear deuda técnica en nuestro trabajo.

Deuda Técnica.

Alrededor de seis horas a la semana invierten los programadores en corregir los errores y deficiencias conocidas como deudas técnicas, es decir, pierden casi un día de trabajo a la semana para solventar problemas derivados de un código pobre.

La deuda técnica es un concepto para definir el costo de mantener y arreglar un software mal construido, a menudo por hacerlo rápido o por no haber llevado a cabo un buen control de calidad. La consecuencia de esto son errores e interrupciones en el funcionamiento, lo que hace que se tenga que intervenir para solventarlos y dediquen parte de su jornada a reparar en lugar de a desarrollar.
fuente:
https://www.xataka.com/pro/deuda-tecnica-lastre-para-tecnologicas-estudio-senala-que-informaticos-pierden-casi-dia-trabajo-a-semana-para-solventarlas

Refactoring

Este término se usa para describir la modificación del código fuente sin cambiar su comportamiento. Se realiza a menudo como parte del proceso de desarrollo del software, los desarrolladores alternan la inserción de nuevas funcionalidades con la refactorización del código para mejorar su consistencia interna y su claridad.

La refactorización es la parte del mantenimiento del código que no arregla errores ni añade funcionalidad. El objetivo, por el contrario, es mejorar la facilidad de

comprensión del código o cambiar su estructura y diseño y eliminar código muerto, para facilitar el mantenimiento en el futuro.

La refactorización debe ser realizada como un paso separado, para poder comprobar con mayor facilidad que no se han introducido errores al llevarla a cabo. Al final de la refactorización, cualquier cambio en el comportamiento es claramente un *bug*.

fuente: https://es.wikipedia.org/wiki/Refactorizaci%C3%B3n

Como vimos en el Parcial 2 todo nuestro programa estaba en un solo archivo, para separarlo en distintos dominios (secciones que comparten estructuras o patrones) conocimos los módulos. Éstos nos permiten separar código en distintos archivos, de esta manera podemos atender los problemas de a uno por vez, ¿recuerdan la descomposición cuando vimos Pensamiento Computacional?, eso mismo.
Por un lado están los datos, el array con objetos, que en la tecnología web son transmitidos con formato JSON. Casi igual a un array de objetos, pero todo es un gran String:

```
[
  {
    "id": 1,
    "ciclista": "Maure Benko",
    "marca": "Specialized",
    "rodado": 24,
    "peso": 9.248,
    "largo": 116.17,
    "dopaje": true
  },
  {
    "id": 2,
    "ciclista": "Burt Alday",
    "marca": "Giant",
    "rodado": 28,
    "peso": 7.5,
    "largo": 99.36,
    "dopaje": false
  }
]
```

Por otro lado está el Objeto que va a manejar esos datos con los distintos métodos o funcionalidades, que también separamos llevando a un nuevo archivo:

```
const inmobiliaria = {
 // A
 departamentos: require("./datos"),
 // B
 listarDepartamentos: function (departamentos) {
 },
 // C
 departamentosDisponibles: function () {
 },
 // D
 buscarPorId: function (id) {
 },
 // E
 buscarPorPrecio: function (precio) {
 },
 // F
 precioConImpuesto: function (porcentaje) {
 },
 // G
 simplificarPropietarios: function () {
 },
};
module.exports = inmobiliaria;
```

¿Notan cómo este archivo depende del archivo datos? Lo requiere para poder trabajar con ellos.

A su vez en el final exportamos este objeto para que otro archivo lo pueda requerir.

¿mas archivos? Si, el archivo que va a invocar a los distintos métodos y se va a encargar de mostrar de cara al usuario final los distintos resultados.

```
const v = "\x1b[32m%s\x1b[0m";
const o = "*".repeat(80) + "\n";
const oo = "*".repeat(25);

const nombre = "tu nombre aquí";
const tema = "el tema que te tocó";

const inmobiliaria = require("./objApp")
```

```
/*****************************/
/* Ejecución de las consignas */
/*****************************/
console.table([{ alumno: nombre, tema: tema }]);

console.log(v, "\n" + oo + "   B. listarDepartamentos");
// Ejecución aquí
inmobiliaria.listarDepartamentos(inmobiliaria.departamentos);
//console.log(inmobiliaria.departamentos)//nota mas baja
console.log(o);

console.log(v, oo + "   C. departamentosDisponibles");
// Ejecución aquí
const disponibles = inmobiliaria.departamentosDisponibles();
inmobiliaria.listarDepartamentos(disponibles);
console.log(o);

console.log(v, oo + " D. buscarPorId");
// Ejecución aquí
const unDpto = inmobiliaria.buscarPorId(7);
inmobiliaria.listarDepartamentos([unDpto]);
console.log(o);

console.log(v, oo + "  E. buscarPorPrecio");
// Ejecución aquí
const porPrecio = inmobiliaria.buscarPorPrecio(3000);
inmobiliaria.listarDepartamentos(porPrecio);
console.log(o);
```

(͡° ͜ʖ ͡°) mmm ah!

Entonces termina quedando el **modelo** de datos, la **vista** del usuario, y el responsable de **controlar** el comportamiento.
Esto es una arquitectura de software bastante común, **M**odel-**V**iew-**C**ontroller o **MVC**.

El sentido de separar en dominios es para no tener todo mezclado en la cabeza, en el cajón de las medias, medias, en el de las remeras, remeras. Como cuando en un array que se llama departamentos, guardamos solamente departamentos. Es obvio cuando lo planteamos así pero no tanto cuando tenemos todo en el pensamiento. Por este motivo ser ordenados con nuestro código es fundamental.

Módulos nativos

En los videos de Playground vimos cómo Node.js posee código modularizado que nos amplía el repertorio de herramientas. Como por ejemplo “fs” que es el módulo de File System, nos permite trabajar con el sistema de archivos de nuestro disco rígido. Básicamente leer y escribir en archivos.

Helpers

Se les dice así a porciones de código que nos **ayudan** (help) a realizar acciones repetitivas que incluyen más de una función. Recordemos que en una función creamos código que realiza una sola cosa, un helper puede ser un objeto que contenga varios métodos para ayudarnos por ejemplo a leer un archivo Json, convertirlo a un array y retornarlo, como así también a recibir un array de obj, convertirlo a json y escribirlo en un archivo. Hagamos uno:

Lo haremos primero asegurándonos que la lectura con fs funciona.

1. requerir al módulo fs
2. guardar en una variable el retorno fs.readFileSync(“nuestroArchivo.txt”, “utf8”)
3. mostrar por consola esa variable y comprobar su funcionamiento
4. hacer lo propio con fs.writeFileSync(“nuestroArchivo.txt”, data) donde data es un string cualquiera.
5. volver a leer y comprobar que el guardado funcionó correctamente

De este modo ya podemos estar tranquilos con esta etapa.

Sumamos dificultad agregando que el archivo a leer y escribir sea un JSON.

1. A la variable donde guardamos lo leído, la enviamos como argumento al método JSON.parse(variableConJson) y guardamos en nueva variable su resultado, un array con onbjetos literales de JS

2. Modificamos algún objeto, por ejemplo eliminando uno o cambiando alguna propiedad y pasamos como argumento a JSON.stringify(eseArrayModificado)

3. Finalmente guardamos y volvemos a leer para comprobar cambios.

Otra etapa de desarrollo incremental donde avanzamos a paso seguro sabiendo que cada etapa por separado funciona correctamente.

Ahora un poco de **refactorización**:

1. Separar en dos funciones leer y escribir esto que hicimos
2. comprobar que no se modifica el comportamiento del código.

Más refactoring:

1. crear un objeto que tenga estas dos funciones, métodos ahora
2. comprobar

¿cómo quedó?

¿dónde están los logs a la consola, dentro o afuera del objeto?

¿qué sería mas correcto?

Finalmente hacer de todo esto un módulo exportando dicho objeto sólo con lo indispensable. Para comprobar su funcionamiento deberemos requerirlo desde un archivo nuevo. Veamos:

1. Crear en un nuevo archivo el requerimiento de éste módulo.
2. Invocar al objeto para leer el archivo.
3. Realizar cambios en el array y guardar en disco con el otro método escribir().
4. Comprobar cambios.