Implementación objeto Manager

Contenido

1. Introducción

En este documento vamos a realizar un ejemplo de administración de relaciones entre objetos, para ello, vamos a crear un objeto `Manager` para que administre los productos que tenemos en **MiTienda**. Este objeto va a organizar en categorías cada uno de los productos que tenemos.

Para mantener los objetos vamos a utilizar dos colecciones, y la relación entre categorías y productos lo haremos mediante objetos literales, emulando las implementaciones que se hacen en bases de datos NoSQL. Para más detalle en el apéndice con la relación entre objetos.

Vamos a implementar la aplicación de tres formas diferentes:

- Utilizando como colección arrays.
- Utilizando como colección mapas.
- Implementado el patrón de diseño **Flyweight**.

1.1. Pasos iniciales

Empezamos creando el fichero *js/mitienda/manager/manager.js* que utilizaremos para implementar el objeto.

En este fichero importaremos las excepciones genéricas.

```
import {
  BaseException,
  InvalidAccessConstructorException,
  EmptyValueException,
  InvalidValueException,
  AbstractClassException,
} from '../exceptions.js';
```

2. Entidades

Vamos a crear la una nueva entidad para trabajar con las categorías denominada `Category`. Además, queremos ampliar el funcionamiento de los productos para que tengan un atributo `url`.

2.1. Objeto `Category`

Este objeto permite relacionar productos. Como propiedades obligatorias tenemos `title` y `url`.

```
class Category {
  #title;
  #url;

  constructor(title = 'Anon', url =
'https://via.placeholder.com/258x172.jpg?text=SinNombre') {
    if (!new.target) throw new InvalidAccessConstructorException();
    if (!title) throw new EmptyValueException('title');
    if (!url) throw new EmptyValueException('url');

    this.#title = title;
    this.#url = url;
    this.description = '';

    Object.defineProperty(this, 'title', {
      enumerable: true,
      get() {
        return this.#title;
      },
      set(value) {
        if (!value) throw new EmptyValueException('title');
        this.#title = value;
      },
    });

    Object.defineProperty(this, 'url', {
      enumerable: true,
      get() {
        return this.#url;
      },
      set(value) {
        if (!value) throw new EmptyValueException('url');
        this.#url = value;
      },
    });
  }
}
```

Exportamos la nueva clase.

```
export {
  Product, Laptop, Camera, Smartphone, Tablet, Category,
};
```

Importamos la nueva clase junto con el resto de las entidades en el fichero *manager*.

```
import {
  Product, Laptop, Camera, Smartphone, Tablet, Category,
} from '../entities/products.js';
```

2.2. Ampliación de `Product`

Añadimos una nueva propiedad a la clase `Product` para asignar la *url* de una imagen.

```
#url;
```

Y añadimos los getter y setter sobre el campo privado como una propiedad enumerable en el construstor.

```
Object.defineProperty(this, 'url', {
  enumerable: true,
  get() {
    return this.#url;
  },
  set(value) {
    if (!value) throw new EmptyValueException('url');
    this.#url = value;
  },
});
```

3. Implementación de `Manager` con arrays

La base de la implementación es que vamos a tener dos arrays para contener los productos y las categorías. Un array es una colección que nos permite tener una relación de orden entre sus elementos en base a su posición. Como hemos visto en las listas ordenadas, es relativamente fácil mantenerlos ordenados para mostrar un resultado, siempre que reordenemos en cada inserción de un nuevo elemento.

Las relaciones entre categorías y productos la vamos a implementar utilizando un **modelo de relación por referencia**. Al añadir una categoría crearemos un objeto literal con dos propiedades, la primera `category` contendrá el objeto de tipo `Category`, y la segunda `products`, será un array con las referencias que nos permita localizar los productos en el array de productos a través de la propiedad serial.

```
{
  category: new Category(...),
  products: ['11111', '22222','33333']
}
```

3.1. Excepciones

Creamos la excepción base sobre la que heredaran el resto de las excepciones del *Manager*.

```
class ManagerException extends BaseException {
  constructor(message = 'Error: Manager Exception.', fileName,
lineNumber) {
    super(message, fileName, lineNumber);
    this.name = 'ManagerException';
  }
}
```

3.2. Estructura de *Singleton*

Preparamos la estructura para el *Singleton*.

```
const Manager = (function () {
  let instantiated;

  function init() {

  }
  return {
    getInstance() {
      if (!instantiated) {
        instantiated = init();
      }
      return instantiated;
    },
  };
}());
```

3.3. Objeto `Manager`

En el cuerpo de la función anónima creamos el objeto `Manager`. Utilizamos dos arrays, uno para las categorías y otro para los productos que vamos a relacionar.

```
class Manager {
  #categories = [];
  #products = [];

  constructor() {
    if (!new.target) throw new InvalidAccessConstructorException();
  }
}
```

El método `init()` creamos el objeto y lo devolvemos congelado.

```
function init() {
  const manager = new Manager();
  Object.freeze(manager);
  return manager;
}
```

3.4. Método addCategory()

El método almacena un objeto `Category`, está preparado para recibir una secuencia de objetos. Las categorías no se guardan directamente, sino que se utiliza un objeto literal con una propiedad array para los productos que estén relacionados con la categoría. Por último, las categorías quedan ordenadas en el array por el título.

```
addCategory(...categories) {
  for (const category of categories) {
    if (!(category instanceof Category)) {
      throw new ObjecManagerException('category', 'Category');
    }
    const position = this.#getCategoryPosition(category);
    if (position === -1) {
      this.#categories.push({
        category,
        products: [],
      });
      this.#categories.sort(this.#sortCategoriesFunc);
    } else {
      throw new CategoryExistsException(category);
    }
  }
  return this;
}
```

Como podemos observar, utilizamos un método privado para buscar el índice de una categoría en el array, el cual nos indica si la categoría ya está registrada en el manager.

```
#getCategoryPosition(category){
  return this.#categories.findIndex(x => x.category.title === category.ti
tle);
}
```

Hemos utilizado una función que ordena las categorías introducidas por el título.

```
#sortCategoriesFunc = (catA, catB) => (
  (catA.category.title.toLocaleLowerCase() <
catB.category.title.toLocaleLowerCase())
    ? -1 : 1
);
```

Por último, hemos definido las siguientes excepciones para verificar que introducimos categorías y que no exista todavía la categoría en el array.

```
class ObjecManagerException extends ManagerException {
  constructor(param, className, fileName, lineNumber) {
    super(`Error: The ${param} is not a ${className}`, fileName,
lineNumber);
    this.param = param;
    this.param = className;
    this.name = 'ObjecManagerException';
  }
}

class CategoryExistsException extends ManagerException {
  constructor(category, fileName, lineNumber) {
    super(`Error: The ${category.title} already exists in the manager.`,
fileName, lineNumber);
    this.category = category;
    this.name = 'CategoryExistsException';
  }
}
```

3.5. Método addProduct()

Seguimos la misma lógica para añadir productos que en el caso anterior, pero insertamos el producto directamente en el array.

```
addProduct(...products) {
  for (const product of products) {
    if (!(product instanceof Product)) {
      throw new ObjecManagerException('product', 'Product');
    }
    const position = this.#getProductPosition(product);
    if (position === -1) {
      this.#products.push(product);
      this.#products.sort(this.#sortProductsFunc);
    } else {
      throw new ProductExistsException(product);
    }
  }
  return this;
}
```

El método privado para obtener el índice de un producto en el array.

```
#getProductPosition(product){
  return this.#products.findIndex(x => x.serial === product.serial);
}
```

La ordenación es por marca y luego por modelo.

```
#sortProductsFunc = (productA, productB) => {
  if (productA.brand.toLocaleLowerCase() <
productB.brand.toLocaleLowerCase()) {
    return -1;
  } if (productA.brand.toLocaleLowerCase() >
productB.brand.toLocaleLowerCase()) {
    return 1;
  }
  return (productA.model.toLocaleLowerCase() <
productB.model.toLocaleLowerCase()) ? -1 : 1;
};
```

Añadimos la excepción para los productos que ya existan basado en su número de serie.

```
class ProductExistsException extends ManagerException {
  constructor(product, fileName, lineNumber) {
    super(`Error: The ${product.serial} already exists in the manager.`,
fileName, lineNumber);
    this.product = product;
    this.name = 'ProductExistsException';
  }
}
```

3.6. Método addProductInCategory()

El método crea las relaciones entre las categorías y los productos. Admiten un número no determinado de productos, siendo el primer parámetro la categoría.

```
addProductInCategory(category, ...products) {
  if (!(category instanceof Category)) {
    throw new ObjecManagerException('category', 'Category');
  }
  let pCategory = this.#getCategoryPosition(category);
  if (pCategory === -1) {
    this.addCategory(category);
    pCategory = this.#getCategoryPosition(category);
  }

  for (const product of products) {
    if (!(product instanceof Product)) {
      throw new ObjecManagerException('product', 'product');
    }
    let pProduct = this.#getProductPosition(product);
    if (pProduct === -1) {
      this.addProduct(product);
      pProduct = this.#getProductPosition(product);
    }
    const position = this.#getProductPositionInCategory(product,
this.#categories[pCategory]);
    if (position === -1) {
      this.#categories[pCategory].products.push(this.#products[pProduct])
;
      this.#categories[pCategory].products.sort(this.#sortProductsInCateg
oryFunc);
    } else {
      throw new ProductExistInCategoryException(product, category);
    }
  }
  return this;
}
```

Tenemos un método privado para obtener la posición del producto en la categoría.

```
#getProductPositionInCategory(product, category){
  return category.products.findIndex(x => x.serial === product.serial);
}
```

Además, mantenemos una función de ordenación para mostrar los resultados ordenados por precio.

```
#sortProductsInCategoryFunc = (productA, productB) => (
  (productA.price > productB.price) ? -1 : 1
);
```

Añadimos una excepción que indique que el producto ya existe en la categoría.

```
class ProductExistInCategoryException extends ManagerException {
  constructor(product, category, fileName, lineNumber) {
    super(`Error: The ${product.serial} already exist in
${category.title}.`, fileName, lineNumber);
    this.category = category;
    this.product = product;
    this.name = 'ProductExistInCategoryException';
  }
}
```

3.7. Iterables de los array

Hacemos que ambos arrays sean iterables mediante una propiedad. En ambos casos devolvemos un objeto que implementa `[Symbol.iterator]()`. Como queremos que sean enumerables lo hacemos en el constructor mediane la definición de una propiedad.

```
Object.defineProperty(this, 'categories', {
  enumerable: true,
  get() {
    const array = this.#categories;
    return {
      * [Symbol.iterator]() {
        for (const arrayCat of array) {
          yield arrayCat.category;
        }
      },
    };
  },
});

Object.defineProperty(this, 'products', {
  enumerable: true,
  get() {
    const array = this.#products;
    return {
      * [Symbol.iterator]() {
        for (const product of array) {
          yield product;
        }
      },
    };
  },
});
```

3.8. Iterador de productos por categoría

Creamos un método que dado un objeto `Category` nos devuelva un iterador con los productos de dicho objeto.

```
* getCategoryProducts(category){
  if (!(category instanceof Category)) {
    throw new ObjecManagerException ('category', 'Category');
  }
  let position = this.#getCategoryPosition(category);
  if (position !== -1){
    let nextIndex = 0;
    let array = this.#categories[position].products;
    for (let product of array){
      yield product;
    }
  } else{
    throw new CategoryNotExistException(category);
  }
}
```

Añadimos la excepción en caso de que la categoría no exista.

```
class ProductExistInCategoryException extends ManagerException {
  constructor (product, category, fileName, lineNumber){
    super(`Error: The ${product.serial} already exist in
${category.title}.`, fileName, lineNumber);
    this.category = category;
    this.product = product;
    this.name = 'ProductExistInCategoryException';
  }
}
```

3.9. Método toString()

Creamos un método `toString()` sobre los iteradores de los array y lo añadimos al prototipo.

```
toString(separator = '\n') {
  let str = '';
  for (const category of this.categories) {
    str += category.title + separator;
    for (const product of this.getCategoryProducts(category)) {
      // console.log(product.value.toString());
      str += product.toString() + separator;
    }
  }
  return str;
}
```

3.10. Exportamos

Exportamos el objeto `Manager` y las entidades.

```
export default Manager;
export {
  Product, Laptop, Camera, Smartphone, Tablet, Category,
} from '../entities/products.js';
```

3.11. Testeo de inserción

Probamos los métodos creados hasta ahora. Creamos los objetos en un fichero específico para el testeo. Primero en el módulo de testeo debemos importar `Manager` y `Category`, el resto de entidades están ya disponibles.

```
import Manager, { Category } from './manager/manager.js';
```

Creamos una función para testear las inserciones de categorías, productos y la relación entre ellos y mostramos el contenido del manager.

```
const category1 = new Category('Promociones',
'https://via.placeholder.com/258x172.jpg?text=Promociones');
const category2 = new Category('Outlet',
'https://via.placeholder.com/258x172.jpg?text=Outlet');
const category3 = new Category('Ofertas especiales',
'https://via.placeholder.com/258x172.jpg?text=Ofertas+especiales');
const category4 = new Category('Reacondicionados',
'https://via.placeholder.com/258x172.jpg?text=Reacondicionados');
category1.description = 'Productos en promoción.';
category2.description = 'Outlet de productos con grandes descuentos.';
category3.description = 'Ofertas actuales.';
category4.description = 'Productos reacondicionados o seminuevos.';
const errorCategory = new Category('Reacondicionados',
'https://via.placeholder.com/258x172.jpg?text=Reacondicionados');

const product1 = new Laptop(1, 'brand1', 'model1', 1100);
const product2 = new Camera(2, 'brand1', 'model2', 1200);
const product3 = new Smartphone(3, 'brand1', 'model3', 1300);
const product4 = new Tablet(4, 'brand1', 'model4', 1400);
const product5 = new Laptop(5, 'brand1', 'model5', 1500);
const product6 = new Laptop(6, 'brand2', 'model1', 2100);
const product7 = new Camera(7, 'brand2', 'model2', 2200);
const product8 = new Tablet(8, 'brand2', 'model3', 2300);
const product9 = new Smartphone(9, 'brand2', 'model4', 2400);
const product10 = new Laptop(10, 'brand2', 'model5', 2500);
const product11 = new Laptop(11, 'brand3', 'model1', 3100);
const product12 = new Camera(12, 'brand3', 'model2', 3200);
const product13 = new Tablet(13, 'brand3', 'model3', 3300);
const product14 = new Smartphone(14, 'brand3', 'model4', 3400);
const product15 = new Laptop(15, 'brand3', 'model5', 3500);
const product16 = new Laptop(16, 'brand4', 'model1', 4100);
const product17 = new Camera(17, 'brand4', 'model2', 4200);
const product18 = new Tablet(18, 'brand4', 'model3', 4300);
```

```
const product19 = new Tablet(19, 'brand4', 'model4', 4400);
const product20 = new Laptop(20, 'brand4', 'model5', 4500);

function testCreateObjects() {
  const manager = Manager.getInstance();

  console.log('Testeo: Objetos categorias');
  manager.addCategory(category2, category1, category4);
  try {
    manager.addCategory(errorCategory);
  } catch (error) {
    console.log(error.toString());
  }

  console.log('Testeo: Objetos productos');
  manager.addProduct(product2, product3, product4, product5);
  manager.addProduct(product7, product8, product9, product10);
  manager.addProduct(product12, product13, product14, product15);
  manager.addProduct(product17, product18, product19, product20);

  manager.addProductInCategory(category1, product1, product2, product3,
product4, product5);
  manager.addProductInCategory(category2, product6, product7, product8,
product9, product10);
  manager.addProductInCategory(category3, product11, product12,
product13, product14, product15);
  manager.addProductInCategory(category4, product16, product17,
product18, product19, product20);
  try {
    manager.addProductInCategory(category4, product16);
  } catch (error) {
    console.log(error.toString());
  }

  console.log('Contenido del carrito');
  console.log(manager.toString('\n'));
}
```

3.12. Método removeCategory()

Creamos un método que elimina una categoría del array.

```
removeCategory(...categories) {
  for (const category of categories) {
    if (!(category instanceof Category)) {
      throw new ObjecManagerException('category', 'Category');
    }
    const position = this.#getCategoryPosition(category);
    if (position !== -1) {
      this.#categories.splice(position, 1);
    } else {
      throw new CategoryNotExistException(category);
    }
  }
  return this;
}
```

Añadimos una excepción para indicar que queremos borrar una categoría que no está registrada.

```
class CategoryNotExistException extends ManagerException {
  constructor(category, fileName, lineNumber) {
    super(`Error: The ${category.title} doesn't exist in the manager.`,
fileName, lineNumber);
    this.category = category;
    this.name = 'CategoryNotExistException';
  }
}
```

3.13. Método removeProduct()

Creamos un método para eliminar un producto del array de productos. También debemos borrar las referencias creadas en las categorías.

```
removeProduct(...products) {
  for (const product of products) {
    if (!(product instanceof Product)) {
      throw new ObjecManagerException('product', 'product');
    }
    const position = this.#getProductPosition(product);
    if (position !== -1) {
      const storedProduct = this.#products[position];
      for (const category of this.#categories) {
        const pProduct =
this.#getProductPositionInCategory(storedProduct, category);
        if (pProduct !== -1) {
          category.products.splice(pProduct, 1);
        }
      }
      this.#products.splice(position, 1);
    } else {
      throw new ProductNotExistInManagerException(product);
    }
  }
  return this;
}
```

Añadimos una excepción para indicar que no existe el producto.

```
class ProductNotExistInManagerException extends ManagerException {
  constructor(product, fileName, lineNumber) {
    super(`Error: The ${product.serial} doesn't exist in the manager.`,
fileName, lineNumber);
    this.product = product;
    this.name = 'ProductNotExistInManagerException';
  }
}
```

3.14. Método removeProductCategory()

El último método de borrado debe eliminar una relación entre productos y categorías.

```
removeProductInCategory(category){
  if (!(category instanceof Category)) {
    throw new ObjecManagerException ('category', 'Category');
  }
  let pCategory = this.#getCategoryPosition(category);
  if (pCategory !== -1){
    for (let i = 1; i < arguments.length; i++){
      let product = arguments[i];
      if (!(product instanceof Product)) {
        throw new ObjecManagerException ('product', 'product');
      }
      let pProduct = this.#getProductPositionInCategory(product,
this.#categories[pCategory]);
      if (pProduct !== -1){
        this.#categories[pCategory].products.splice(pProduct,1);
      } else {
        throw new ProductNotExistInCategoryException(product,
this.#categories[pCategory].category);
      }
    }
  } else{
    throw new CategoryNotExistException(category);
  }
  return this;
}
```

Necesitamos una excepción que indique que un producto no está registrado en la categoría.

```
class ProductNotExistInCategoryException extends ManagerException {
  constructor(product, category, fileName, lineNumber) {
    super(`Error: The ${product.serial} doesn't exist in
${category.title}.`, fileName, lineNumber);
    this.category = category;
    this.product = product;
    this.name = 'ProductNotExistInCategoryException';
  }
}
```

3.15. Testeo de borrado

Añadimos al testeo los métodos de borrado.

```
function testRemoveObjects(){
  let manager = Manager.getInstance();

  $$result.logBold("Test: Borrado de objetos");
  $$result.logBold("Test: Borrado de productos");
  $$result.log("Productos: p1 y p12");
  manager.removeProduct(product1, product12);
  $$result.logBold("Test: Borrado de productos en categorías");
  $$result.log("Productos: p2 y p3 en c1");
  manager.removeProductInCategory(category1, product3, product2);
  try{
    manager.removeProductInCategory(category1, product19);
  } catch(error){
    console.log(error.toString());
  }

  $$result.logBold("Test: Borrado de categoría");
  $$result.log("Categoría: c2");
  manager.removeCategory(category2);

  try{
    manager.removeCategory(new Category('ErrorCategory',
'img/error.jpg'));
  } catch(error){
    console.log(error.toString());
  }

  $$result.logBold("Contenido del carrito");
  $$result.log(manager.toString('<br>'));
}
```

3.16. Iteradores de tipo de productos

Vamos a crear un iterador que, dado un tipo de producto, obtengamos un iterador de los productos que cumplan el criterio. Además, pasaremos una cadena de texto para indicar la propiedad por la que ordenaremos los resultados.

Primero creamos un objeto literal privado con las funciones de ordenación que necesitamos.

```
#order = {
  serial: (productA, productB) => {return productA.serial < productB.seri
al? -1 : 1},
  brand: (productA, productB) => {return productA.brand < productB.brand?
 -1 : 1},
  model: (productA, productB) => {return productA.model < productB.model?
 -1 : 1},
  price: (productA, productB) => {return productA.price < productB.price?
 -1 : 1},
}
```

El siguiente método crea un array filtrando en función del tipo de instancia, para después ordenarlos por el campo si disponemos de la función de ordenación.

```
* getTypeProducts(type, field) {
  const array = this.#products.filter((product) => product instanceof
type);
  if (this.#order[field]) {
    array.sort(this.#order[field]);
  }

  for (const product of array) {
    yield product;
  }
}
```

Para testearlo añadimos el código a la función de testeo.

```
function testListObjects() {
  const manager = Manager.getInstance();

  console.log('Listado Laptop ordenado por brand');
  for (const product of manager.getTypeProducts(Laptop, 'brand')) {
    console.log(product.toString());
  }
}
```

3.17. Limpiado del `Manager`

El siguiente método reinicia los objetos del `Manager`.

```
clear (){
  this.#categories.length = 0;
  this.#products.length = 0;
}
```

4. Implementación de `Manager` con mapas

La implementación con `Map` agiliza las búsquedas de los objetos en la colección al estar indexado por la clave. En nuestro caso, vamos a crear dos mapas uno para productos indizado por la propiedad `serial` y otro para categorías por `title`.

Vamos a reescribir los métodos.

4.1. Campos privados

Pasan a ser dos objetos de tipo `Map`.

```
#categories = new Map();
#products = new Map();
```

4.2. Método addCategory()

Reescribimos el método para utilizar la colección como un `Map`. En la relación con los productos también vamos a utilizar un `Map`.

```
addCategory(...categories) {
  for (const category of categories) {
    if (!(category instanceof Category)) {
      throw new ObjecManagerException('category', 'Category');
    }
    if (!this.#categories.has(category.title)) {
      this.#categories.set(category.title, {
        category,
        products: new Map(),
      });
    } else {
      throw new CategoryExistsException(category);
    }
  }
  return this;
}
```

Podemos borrar:

- El método privado `#getCategoryPosition()`.
- El campo privado `#sortCategoriesFunc`.

4.3. Método addProduct()

Modificamos el método para utilizar `Map`.

```
addProduct(...products) {
  for (const product of products) {
    if (!(product instanceof Product)) {
      throw new ObjecManagerException('product', 'Product');
    }
    if (!this.#products.has(product.serial)) {
      this.#products.set(product.serial, product);
    } else {
      throw new ProductExistsException(product);
    }
  }
  return this;
}
```

Podemos borrar:

- El método privado `#getProductPosition()`.
- El campo privado `#sortProductsFunc`.

4.4. Método addProductInCategory()

Para este caso tenemos que recordar que al recuperar la categoría del mapa, lo que obtenemos es el objeto literal en lugar de la categoría directamente.

```
addProductInCategory(category, ...products) {
  if (!(category instanceof Category)) {
    throw new ObjecManagerException('category', 'Category');
  }
  if (!this.#categories.has(category.title)) {
    this.addCategory(category);
  }
  const storedCategory = this.#categories.get(category.title);
  for (const product of products) {
    if (!(product instanceof Product)) {
      throw new ObjecManagerException('product', 'product');
    }
    if (!this.#products.has(product.serial)) {
      this.addProduct(product);
    }
    const storedProduct = this.#products.get(product.serial);
    if (!storedCategory.products.has(product.serial)) {
      storedCategory.products.set(product.serial, storedProduct);
    } else {
      throw new ProductExistInCategoryException(product, category);
    }
  }
  return this;
}
```

Podemos borrar:

- El método privado `#getProductPositionInCategory()`.
- El campo privado `# sortProductsInCategoryFunc`.

4.5. Iterables

Implementamos los getter con mapas. Primero obtenemos las categorías.

```
get() {
  const values = this.#categories.values();
  return {
    * [Symbol.iterator]() {
      for (const storedCategory of values) {
        yield storedCategory.category;
      }
    },
  };
},
```

Repetimos con los productos.

```
get() {
  const values = this.#products.values();
  return {
    * [Symbol.iterator]() {
      for (const product of values) {
        yield product;
      }
    },
  };
},
```

4.6. Iterador de productos por categoría

Lo transformamos para trabajar con los mapas.

```
* getCategoryProducts(category) {
  if (!(category instanceof Category)) {
    throw new ObjecManagerException('category', 'Category');
  }
  if (this.#categories.has(category.title)) {
    const storedCategory = this.#categories.get(category.title);
    const values = storedCategory.products.values();
    for (const product of values) {
      yield product;
    }
  } else {
    throw new CategoryNotExistException(category);
  }
}
```

4.7. Método removeCategory()

```
removeCategory(...categories) {
  for (const category of categories) {
    if (!(category instanceof Category)) {
      throw new ObjecManagerException('category', 'Category');
    }
    if (this.#categories.has(category.title)) {
      this.#categories.delete(category.title);
    } else {
      throw new CategoryNotExistException(category);
    }
  }
  return this;
}
```

4.8. Método removeProduct()

```
removeProduct(...products) {
  for (const product of products) {
    if (!(product instanceof Product)) {
      throw new ObjecManagerException('product', 'product');
    }
    if (this.#products.has(product.serial)) {
      for (const category of this.#categories.values()) {
        if (category.products.has(product.serial)) {
          category.products.delete(product.serial);
        }
      }
      this.#products.delete(product.serial);
    } else {
      throw new ProductNotExistInManagerException(product);
    }
  }
  return this;
}
```

4.9. Método removeProductCategory()

```
removeProductInCategory(category, ...products) {
  if (!(category instanceof Category)) {
    throw new ObjecManagerException('category', 'Category');
  }
  if (this.#categories.has(category.title)) {
    const storedCategory = this.#categories.get(category.title);
    for (let i = 1; i < arguments.length; i++) {
      const product = arguments[i];
      if (!(product instanceof Product)) {
        throw new ObjecManagerException('product', 'product');
      }
      if (storedCategory.products.has(product.serial)) {
        storedCategory.products.delete(product.serial);
      } else {
        throw new ProductNotExistInCategoryException(product,
storedProduct.category);
      }
    }
  } else {
    throw new CategoryNotExistException(category);
  }
  return this;
}
```

4.10. Iteradores de tipo de productos

```
* getTypeProducts(type, field) {
  const array = [...this.#products.values()].filter((product) => product
instanceof type);
  if (this.#order[field]) {
    array.sort(this.#order[field]);
  }

  for (const product of array) {
    yield product;
  }
}
```

4.11. Limpiado del `Manager`

```
clear() {
  this.#categories.clear();
  this.#products.clear();
}
```

5. Comparativa Array vs Map

El rendimiento de un array para la **búsqueda de objetos** tiene un rendimiento bastante malo, al tener que recorrer cada una de las posiciones. En este caso, los mapas tienen un rendimiento mucho mejor al estar indizados.

El problema de los mapas es a la hora de **mostrar los resultados** de la colección en un determinado orden, ya que el mapa no ordena su contenido, tan solo se hace en función del orden de inserción. Ordenar los datos implica transformar un iterador en array, ordenarlo, y recorrerlo para mostrar el resultado. El rendimiento de esta tarea cae en picado en función del crecimiento de los datos de la colección.

Debemos utilizar un array cuando tengamos una relación de orden entre los datos de la colección.

Debemos utilizar un mapa cuando tengamos que maximizar el rendimiento en la búsqueda de los datos de la colección.

En el siguiente artículo discute la posibilidad de implementar una clase `SortedMap`, que extiende la funcionalidad de los objetos `Map` mediante el uso de arrays ordenados. La clase admite en el constructor una función para mantener los datos de la colección.

https://www.todojs.com/objeto-map-ordenado/

La implementación de esta clase queda fuera del ámbito del curso, aunque podemos encontrar una breve explicación en el apéndice del documento.

Debemos tener en cuenta que en el backend de una aplicación podremos disponer de una base de datos que nos permita recuperar los datos de forma organiza y con el rendimiento deseado.

5.1. Implementación con `SortedMap`

Primero debemos generar el módulo con la clase en el mismo directorio del manager. Importamos la clase.

```
import {SortedMap} from './sortedmap.js';
```

Creamos las colecciones con la nueva clase. Para las categorías ordenamos en base al título de la categoría. Para los productos ordenamos por marca y después por modelo.

```
#categories = new SortedMap((categoryA, categoryB) =>
categoryA[0].localeCompare(categoryB[0]));
#products = new SortedMap((productA, productB) => {
  if (productA[1].brand.toLocaleLowerCase() <
productB[1].brand.toLocaleLowerCase()) {
    return -1;
  } if (productA[1].brand.toLocaleLowerCase() >
productB[1].brand.toLocaleLowerCase()) {
    return 1;
  }
  return (productA[1].model.toLocaleLowerCase() <
productB[1].model.toLocaleLowerCase()) ? -1 : 1;
});
```

Modificamos el mapa de los productos de cada categoría. En esta ocasión ordenamos por precio.

```
products: new SortedMap((productA, productB) => ((productA[1].price <
productB[1].price) ? -1 : 1)),
```

Para el método de obtención de productos por categoría, alteramos la firma para definir un nuevo parámetro que recoja una función de ordenación.

```
* getCategoryProducts(category, ordered)
```

Al obtener los valores de la colección comprobamos si existe la función de ordenación para ordenarlos a partir de ella.

```
const values = (ordered)
  ? storedCategory.products.values(ordered)
  : storedCategory.products.values();
```

6. Patrón de diseño Flyweight

El patrón de diseño Flyweight es un patrón estructural que se utiliza para optimizar el rendimiento y la eficiencia de una aplicación al reducir la redundancia de datos. En el contexto de JavaScript, el patrón Flyweight se implementa para minimizar el uso de recursos al compartir objetos comunes entre múltiples instancias, en lugar de crear nuevas instancias para cada caso similar.

En términos sencillos, el patrón Flyweight divide un objeto en dos partes: la parte intrínseca y la parte extrínseca. La parte intrínseca contiene el estado compartido y es almacenada de forma centralizada, mientras que la parte extrínseca contiene el estado único para cada instancia y se pasa como parámetro cuando es necesario.

Ventajas del patrón:

- **Optimización de recursos**: Al compartir instancias de objetos similares, se reduce la cantidad de memoria utilizada, ya que se evita la duplicación de datos comunes.
- **Mejora del rendimiento**: Al reducir la carga de memoria y el número de instancias, se mejora el rendimiento de la aplicación, especialmente en escenarios con grandes cantidades de objetos similares.
- **Flexibilidad**: Permite separar el estado intrínseco (compartido) y extrínseco (único), lo que facilita la adaptación a cambios en los requisitos y facilita la reutilización de código.

Inconvenientes del patrón:

- **Complejidad adicional**: La implementación del patrón Flyweight puede introducir cierta complejidad en el código, ya que se necesita dividir el estado intrínseco y extrínseco y gestionar adecuadamente las instancias compartidas.

Requisitos de diseño específicos: No es adecuado para todos los escenarios. Funciona mejor cuando hay una gran cantidad de objetos similares y se puede identificar claramente el estado intrínseco y extrínseco.

Situaciones adecuadas para el uso del patrón:

- **Gran cantidad de objetos**: Se beneficia especialmente en situaciones donde hay un gran número de objetos similares que comparten propiedades comunes.
- **Consumo de memoria crítico**: Es útil cuando el consumo de memoria es una preocupación y se busca optimizar la eficiencia en este aspecto.
- **Aplicaciones gráficas o juegos**: En aplicaciones que manejan gráficos o juegos, donde hay muchos elementos visuales que comparten características comunes, el patrón Flyweight puede ayudar a reducir la carga de memoria.
- **Sistemas donde el rendimiento es clave**: En sistemas donde el rendimiento es esencial y se busca minimizar la sobrecarga de recursos.

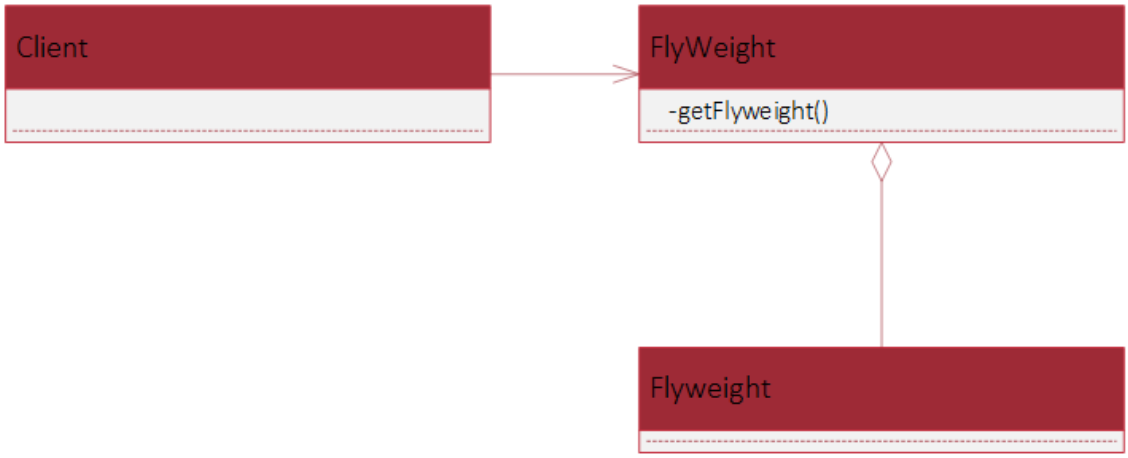
En resumen, el patrón de diseño Flyweight es beneficioso en situaciones específicas donde hay una cantidad significativa de objetos similares y se busca optimizar el uso de recursos. Sin embargo, su implementación debe evaluarse cuidadosamente en función de los requisitos específicos de cada caso, ya que puede agregar complejidad innecesaria en escenarios más simples.

6.1. Ejemplo de código

El siguiente diagrama UML muestra la implementación del patrón. El patrón se basa en otro patrón de diseño, el **Factory**. Disponemos de una clase `FlyweightFactory` que es capaz de crear los objetos de entidad `Flyweight` que necesitemos, además lleva un registro de todos los que se han creado hasta el momento, por eso se trata de una agregación de objetos de entidad.

La clase cliente `Client` no debe crear objetos de entidad directamente, sino que los solicita a `FlyweightFactory`. Si el objeto de entidad solicitado no existe, `FlyweightFactory` lo creará y lo registrará. En el caso de que la entidad estuviera registrada previamente, devuelve la entidad que ya había sido creada. De esta forma, no replicamos instancias con el mismo contenido.

La clase `FlyweightFactory`, además de la factoría, puede disponer de más métodos de negocio para implementar la funcionalidad de la aplicación.



El siguiente ejemplo muestra un ejemplo de implementación.

```
// Objeto Flyweight
class Flyweight {
  constructor(sharedState) {
    this.sharedState = sharedState;
  }
}

// Fabrica de Flyweights
class FlyweightFactory {
  constructor() {
    this.flyweights = {};
  }

  getFlyweight(sharedState) {
    if (!this.flyweights[sharedState]) {
      this.flyweights[sharedState] = new Flyweight(sharedState);
    }
    return this.flyweights[sharedState];
  }
}

// Cliente
class Client {
  constructor(flyweightFactory) {
    this.flyweightFactory = flyweightFactory;
    this.instances = [];
  }

  createInstance(uniqueState, sharedState) {
    const flyweight = this.flyweightFactory.getFlyweight(sharedState);
    this.instances.push({ uniqueState, flyweight });
  }
```

```
}

// Uso
const flyweightFactory = new FlyweightFactory();
const client = new Client(flyweightFactory);

client.createInstance("A", "shared");
client.createInstance("B", "shared");
client.createInstance("C", "shared");

console.log(client.instances);
```

6.2. Implementación de `Manager` con Flyweight

Comenzamos creando el método de creación de entidades `Category`. Tenemos que recordar que las entidades no están almacenadas directamente en la colección, sino que utilizamos un objeto literal para relacionar la categoría con los productos.

```
getCategory(title = 'Anon', url =
'https://via.placeholder.com/258x172.jpg?text=SinNombre') {
  let cat = this.#categories.get(title);
  if (!cat) {
    cat = new Category(title, url);
  } else {
    cat = cat.category;
  }
  return cat;
}
```

Continuamos con el método de creación de productos. Los productos se encuentran directamente en la colección.

```
getProduct(serial, brand, model, price, type) {
  let product = this.#products.get(serial);
  if (!product){
    product = new this.#productConstructors[type](serial, brand, model,
price);
  }
  return product;
}
```

Estamos trabajando con subclases por lo que debemos de indicar el tipo de clase que queremos instanciar, lo haremos a través del nombre de la clase. Definimos un parámetro `type` para indicar el tipo de subclase.

Los nombres de las subclases los habilitaremos en el objeto de Singleton.

```
return {
  getInstance() {
    if (!instantiated) {
      instantiated = init();
    }
    return instantiated;
  },
  Laptop: Laptop.name,
  Camera: Camera.name,
  Smartphone: Smartphone.name,
  Tablet: Tablet.name
};
```

Por último, en lugar de utilizar una estructura `switch` para seleccionar el tipo, en general poco eficiente, lo hemos refactorizado. Hemos creado un objeto literal como propiedad privada con los constructores de las subclases.

```
#productConstructors = {
  Laptop,
  Camera,
  Smartphone,
  Tablet,
};
```

En el momento de la creación de las entidades accederemos al objeto literal a través de la notación corchete y pasando el string con el nombre de la clase para recuperar el constructor, y así instanciar el objeto.

```
product = new this.#productConstructors[type](serial, brand, model,
price);
```

6.3. Testeo

Cambiamos las importaciones del testeo para solo utilizar las clases que necesitamos. Las clases de las entidades solo serán necesarias para chequear tipos de objetos, o para crear un listado de los productos del manager de un determinado tipo. Estas clases no serán utilizadas para instanciar objetos.

```
import ShoppingCart from './shoppingcart/shoppingcart.js';
import Manager, {
  Laptop, Smartphone, Tablet, Camera, Product,
} from './manager/manager.js';
```

El siguiente código muestra cómo podemos instanciar un objeto de una clase a partir de la factoría, ya sea para categorías o para productos.

```
const category1 = Manager.getInstance().getCategory('Promociones',
'https://via.placeholder.com/258x172.jpg?text=Promociones');
const product1 = Manager.getInstance().getProduct(1, 'brand1', 'model1',
1100, Manager.Laptop);
```

Apéndice I: Relación entre objetos

Las bases de datos **NOSQL** son aquellas que utilizan almacenes de objetos para contener la información, en lugar de utilizar relaciones entre tablas como ocurre con las relacionales. Un ejemplo es **MongoDB** la cual almacena documento en un formato parecido a **JSON**. Como veremos en próximas unidades, un documento **JSON** es un *objeto literal* traducido a *string* para que pueda ser portable. Estos documentos son fácilmente interpretables por un humano, por lo que no necesitan ser procesados. Por último, pueden ser transferibles a través de red para comunicar componentes de una aplicación, o de forma más habitual, entre un cliente y servidor.

Un ejemplo de documento JSON podría utilizarse para representar un usuario. El siguiente ejemplo vemos cómo podemos utilizar esta representación para un usuario con una propiedad *_id*, *name*, *contact*, y *dob* con la fecha de nacimiento.

```
{
    "_id": "52ffc33cd85242f436000001",
    "name": "Tom Hanks",
    "contact": "987654321",
    "dob": "01-01-1991"
 }
```

Otro ejemplo podría utilizase para representar una dirección.

```
{
    "_id": "52ffc4a5d85242602e000000",
    "building": "22 A, Indiana Apt",
    "pincode": 123456,
    "city": "Los Angeles",
    "state": "California"
}
```

Como vemos, ambos siguen una estructura similar a la de un objeto literal. Ambos, JSON y los objetos literales, son formatos intercambiables.

MongoDB utiliza los dos modelos de relación de objetos citados. Veamos unos ejemplos.

Modelo embebido o de relación embebida

En este modelo, los objetos subordinados están embebidos dentro del objeto principal. Un ejemplo es el siguiente, donde tenemos un usuario y un array con todas sus posibles direcciones.

```
{
    "_id": "52ffc33cd85242f436000001",
    "contact": "987654321",
    "dob": "01-01-1991",
    "name": "Tom Benzamin",
    "address": [
       {
          "building": "22 A, Indiana Apt",
          "pincode": 123456,
          "city": "Los Angeles",
          "state": "California"
       },
```

```
        {
            "building": "170 A, Acropolis Apt",
            "pincode": 456789,
            "city": "Chicago",
            "state": "Illinois"
        }
    ]
}
```

Esto nos crea la **ventaja** de que si queremos acceder al contenido se encuentra disponible inmediatamente. No tenemos que hacer ningún paso extra para recuperarlo. El **inconveniente** viene en el mantenimiento de los datos y su repetición. Dos usuarios podrían tener la misma dirección, si queremos modificar un dato de la dirección tendríamos que ir objeto tras objeto comprobando si es la dirección que estamos buscando para actualizar el dato.

Modelo relación por referencia

En este caso los objetos subordinados residen en otro documento o estructura, y son referenciados mediante una propiedad con forma de identificador. En este ejemplo, las direcciones son referenciadas por un identificador.

```
{
    "_id":"52ffc33cd85242f436000001",
    "contact": "987654321",
    "dob": "01-01-1991",
    "name": "Tom Benzamin",
    "address_ids": [
        "52ffc4a5d85242602e000000",
        "52ffc4a5d85242602e000001"
    ]
}
```

La **ventaja** de este modelo es el mantenimiento, ya que los datos están centralizados, pero el **inconveniente** es que tendremos que realizar varios pasos para encontrar la información, el usuario, y una vez encontrado, habría que buscar todas las direcciones que le pueden pertenecer a través de los *id* aportados por el objeto del usuario.

Nunca hay soluciones perfectas, todo depende de las necesidades de nuestra aplicación, y qué beneficia más a nuestra lógica de negocio.

Apéndice II: Clase SortedMap

La implementación de la clase se basa en disponer de una función que permita ordenar los datos recibidos en la colección en cada inserción, además de mantener un array que facilite su recuperación de forma ordenada, el array almacena los pares clave y valor en un array, tal y como se utiliza en `Map`.

Propiedades ocultas

La clase utilizar 4 `Symbol` para implementar su funcionalidad.

- LIST: Almacenará el array para mantener la doble colección.

- LIST_DELETE: Se utiliza para implementar un método interno para el borrado de datos de la colección.
- LIST_FIND: Utilizado para realizar una búsqueda binaria y devolver la posición de una clave dentro del array.
- COMPARE_FUNCTION: contendrá la función de ordenación del mapa.

```
const LIST             = Symbol ();
const LIST_DELETE      = Symbol ();
const LIST_FIND        = Symbol ();
const COMPARE_FUNCTION = Symbol ();
```

Clase y constructor

La clase extiende de `Map`. El constructor chequea si tenemos una función de ordenación. Si es así, la empaqueta con el array de datos para insertar los datos iniciales. La razón es que el superconstructor invocará al método `set()` de manera interna, por lo que es en dicho método donde crearemos el array en la primera invocación.

```
class SortedMap extends Map {

  constructor (data, compareFunction) {
    if (typeof data === 'function') {
      compareFunction = data;
      data            = [];
    }
    if (compareFunction) {
      data.unshift ([ COMPARE_FUNCTION, compareFunction ]);
    }
    super (data);
  }
}
```

Método `set()`

El método chequea si existe el array `this[LIST]` para crearlo en la primera invocación. Junto con la creación se extrae los argumentos de entrada la función de comparación para almacenarla en la propiedad privada `this[COMPARE_FUNCTION]`.

El funcionamiento habitual es comprobar si la clave ya existe, en ese caso se eliminará del array con el método privado `this[LIST_DELETE]`, se añade al mapa y se añade en el array en la posición de ordenación calculada con `this[LIST_FIND]`.

```
set (key, value) {
  if (!this[ LIST ]) {
    this[ LIST ] = [];
    if (key === COMPARE_FUNCTION) {
      return this[ COMPARE_FUNCTION ] = value;
    }
  }
  if (this.has (key)) {
    this[ LIST_DELETE ] (key);
  }
  super.set (key, value);
  this[ LIST ].splice (this[ LIST_FIND ] (key, value), 0, [ key, value
]);

}
```

Método `delete()`

Borra un dato en la colección tanto en el mapa como en el array.

```
delete (key) {
  if (this.has (key)) {
    this[ LIST_DELETE ] (key);
    super.delete (key);
  }
}
```

Método `clear()`

Borra la colección en el array y en el mapa.

```
clear () {
  this[ LIST ] = [];
  super.clear ();
}
```

Colección iterable

Implementamos `[Symbol.iterator ] ()` a partir de la misma propiedad del array.

```
[ Symbol.iterator ] () {
  return this[ LIST ][ Symbol.iterator ] ();
}
```

Recorrer valores y claves de la colección

Los métodos `entries()`, `keys()` y `values()` quedan implementamos mapeando el array al estar ya ordenado. Como mejora, estos métodos aceptan una nueva función de ordenación, en el caso de recibirla realizan una copia del array y lo ordenan a partir de la función.

```
entries (ordered) {
  if (ordered)
    return [...this[ LIST ]].sort(ordered)[ Symbol.iterator ] ();
  else
  return this[ LIST ];
}

keys (ordered) {
  if (ordered)
    return [...this[ LIST ]].sort(ordered).map (x => x[ 0 ]);
  else
    return this[ LIST ].map (x => x[ 0 ]);
}

values (ordered) {
  if (ordered)
    return [...this[ LIST ]].sort(ordered).map (x => x[ 1 ]);
  else
    return this[ LIST ].map (x => x[ 1 ]);
}
```

Método `forEach()`

El método `forEach()` queda sobrescrito para iterar a partir del array invocando a la función de callback.

```
forEach (callbackFn, thisArg) {
  const cb = callbackFn.bind (thisArg || this);
  for (let n = 0; n < this[ LIST ].length; n++) {
    cb (this[ LIST ][ n ][ 1 ], this[ LIST ][ n ][ 0 ], this || thisArg);
  }
}
```

Método de búsqueda

El método privado `[LIST_FIND] ()` realiza una búsqueda binaria para localizar el dato en el array en lugar de recorrerlo de forma secuencial. Este algoritmo aprovecha que el array ya está ordenado, busca el punto medio dividiendo el array en dos partes, si el valor buscado es menor trabaja con la parte más baja, si por el contrario es mayor repite la acción con la parte más alta, así hasta encontrar el valor buscado. Devuelve la posición en el array del valor que estamos buscando.

```
[ LIST_FIND ] (key, value) {
  const list = this[ LIST ];
  if (!list.length) {
    return 0;
  }
  const compareFunction = this[ COMPARE_FUNCTION ] ||
                          ((x, y) => x[ 1 ] > y[ 1 ] ? 1 : x[ 1 ] < y[ 1
] ? -1 : 0);
  let start             = 0;
  let end               = list.length - 1;
  while (start < end && start < list.length && end > 0) {
    const position = start + Math.floor ((end - start) / 2);
    const result   = compareFunction (list[ position ], [ key, value ]);
    if (result === 0) {
      return position;
    }
    if (result < 0) {
      start = position + 1;
    } else if (result > 0) {
      end = position - 1;
    }
  }
  if (list[ start ] && compareFunction (list[ start ], [ key, value ]) >=
0) {
    return start
  }
  if (!list[ end ] || compareFunction (list[ end ], [ key, value ]) < 0)
{
    return end + 1;
  }
}
```

Método de borrado

El método privado `[ LIST_DELETE ] ()` acepta como argumento una clave para borrar el dato del array a partir de ella. El método se asegura que tanto la clave como el valor coinciden y a que se puede dar el caso que dos claves tengas asociado el mismo valor.

```
[ LIST_DELETE ] (key) {
  const value  = this.get (key);
  let position = this[ LIST_FIND ] (key, value);
  let current  = position;
  while (current < this[ LIST ].length &&
         this[ LIST ][ current ][ 1 ] === value &&
         this[ LIST ][ current ][ 0 ] !== key)
  {
    current++;
  }
  if (this[ LIST ][ current ][ 0 ] !== key) {
    current = position;
    while (current > 0 &&
           this[ LIST ][ current ][ 1 ] === value &&
           this[ LIST ][ current ][ 0 ] !== key)
    {
      current--;
    }
  }
  this[ LIST ].splice (current, 1);
}
```