Identificación de patrones

Analiza el siguiente código en JavaScript y determina qué patrón de diseño se está utilizando y describirlos

```
class Database {
  constructor() {
    if (!Database.instance) {
      Database.instance = this;
    }
    return Database.instance;
  }

  query(sql) {
    console.log("Ejecutando consulta:", sql);
  }
}

const db1 = new Database();
const db2 = new Database();

console.log(db1 === db2); // Output: true
db1.query("SELECT * FROM users");
```

Patrón Singleton:

La clase Database implementa el patrón Singleton. Utiliza una variable estática instance y verifica en el constructor si ya existe una instancia previa. Si no existe, crea una nueva instancia y la asigna a instance. Si ya existe una instancia, devuelve esa instancia en lugar de crear una nueva.

Esto asegura que solo exista una instancia global de la clase Database. Cuando se llaman db1 y db2 como nuevas instancias de Database, en realidad se está obteniendo la misma instancia singleton. Esto se demuestra comparando db1 === db2, lo cual da true porque son la misma instancia. El patrón Singleton garantiza un punto de acceso global a un objeto, evitando la creación de múltiples instancias innecesarias.

```
class Logger {
  constructor() {
    this.logs = [];
  }

  log(message) {
    this.logs.push(message);
    console.log("Log registrado:", message);
  }

  static getInstance() {
    if (!Logger.instance) {
      Logger.instance = new Logger();
    }
    return Logger.instance;
  }
}

const logger1 = Logger.getInstance();
const logger2 = Logger.getInstance();

console.log(logger1 === logger2); // Output: true
logger1.log("Error: No se puede conectar al servidor");
```

Patrón Singleton:

El patrón Singleton garantiza que una clase sólo tenga una única instancia global y proporcione un punto de acceso a la misma. Esta clase tiene un constructor privado que impide crear instancias con el operador new, y un método estático que crea o devuelve la única instancia permitida.

En el código proporcionado, la clase Logger implementa el Singleton a través de su método estático getInstance(). Este verifica si ya existe una instancia de Logger asignada a la propiedad estática Logger.instance. Si no existe, crea una nueva instancia. Si ya existe, devuelve esa misma instancia en lugar de crear una nueva. Así, cuando se llama a Logger.getInstance(), siempre se

obtiene la misma instancia global singleton de Logger, ya sea creándola por primera vez o devolviendo la existente. Esto se demuestra comparando logger1 === logger2, que da true porque ambas variables tienen asignada la única instancia singleton de Logger.

```
class User {
  constructor(name) {
    this.name = name;
  }

  greet() {
    console.log("Hola, soy", this.name);
  }
}

class UserFactory {
  createUser(name) {
    return new User(name);
  }
}

const factory = new UserFactory();
const user1 = factory.createUser("Juan");
const user2 = factory.createUser("Maria");

user1.greet(); // Output: Hola, soy Juan
user2.greet(); // Output: Hola, soy Maria
```

Patrón Factory:

El patrón Factory es un patrón de diseño creacional que centraliza la creación de objetos en una clase "Fábrica". En lugar de crear instancias directamente, se delega esta responsabilidad a la clase Fábrica, la cual contiene un método factory encargado de instanciar los objetos.

En el código proporcionado, la clase UserFactory actúa como la Fábrica y tiene un método estático create() que crea e instancia objetos de la clase User. Para crear nuevos usuarios, se llama a UserFactory.create('nombre'), lo que devuelve una nueva instancia de User con ese nombre.

Esto encapsula la lógica de creación de objetos User dentro de UserFactory, separándola del código que utiliza esas instancias. Promueve el bajo acoplamiento y permite que las subclases alteren el tipo de objeto creado. Las variables user1 y user2 demuestran la creación de usuarios utilizando el método factory. La creación se centraliza en UserFactory.create(), abstrayendo los detalles de instanciación de User.