Applying design principle on our website KontaNiben using Node.Js

The SOLID principles are a set of five object-oriented design principles aimed at making software designs more understandable, flexible, and maintainable. A complete function-based example code for an e-commerce project using Node.js is described here.

1. Single Responsibility Principle (SRP): A class should have only one reason to change, meaning that it should have only one responsibility.

2. Open/Closed Principle (OCP): Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

3. Liskov Substitution Principle (LSP): Objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.

4. Interface Segregation Principle (ISP): Clients should not be forced to depend on interfaces they do not use.

5. Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

Using these principles in a function-based example for an e-commerce website using Node.js is described below

Applying SOLID Principles in Node.js

~~~~
```javascript
// Single Responsibility Principle (SRP)
// Each function should have only one responsibility


// Function to add a product to the cart
function addToCart(product, quantity, cart) {
  const item = { product, quantity };
  cart.push(item);
}


// Open/Closed Principle (OCP)
// Code should be open for extension but closed for modification
// Function to calculate total price of items in the cart
function calculateTotalPrice(cart) {
~~~~

~~~
  let totalPrice = 0;
  for (let item of cart) {
    totalPrice += item.product.price * item.quantity;
  }
  return totalPrice;
}

// Liskov Substitution Principle (LSP)
// Subclasses should be substitutable for their base classes

// Interface Segregation Principle (ISP)
// Clients should not be forced to depend on interfaces they do not use

// Dependency Inversion Principle (DIP)
// High-level modules should not depend on low-level modules

// Example usage
const cart = [];
const product = { name: "Product 1", price: 10 };
addToCart(product, 2, cart);

const totalPrice = calculateTotalPrice(cart);
console.log("Total Price:", totalPrice);
```
~~~

In this example:

SRP: The `addToCart` function has the responsibility of adding a product to the cart, and the `calculateTotalPrice` function has the responsibility of calculating the total price of items in the cart.

OCP: If we want to extend the functionality, for example, by applying discounts, we can create a new function or class without modifying the existing functions.

LSP: Objects of the subclass, such as a product, can be easily replaced with objects of its superclass, ensuring compatibility.

ISP: Clients can use only the functions they need without being forced to depend on unnecessary interfaces.

DIP: The high-level functions (`addToCart`, `calculateTotalPrice`) do not directly depend on low-level details, such as data storage mechanisms or specific product implementations.

By following SOLID principles, we will be able to create more maintainable, flexible, and scalable code for our e-commerce project in Node.js.

In our e-commerce project, we want to manage products and orders efficiently. We'll create separate modules for handling products and orders, adhering to DRY and KISS principles by avoiding code repetition and keeping the code simple and easy to understand.

Product Management Module:

~~~
```javascript
// products.js
// Data structure to store products
let products = [];
// Function to add a new product
function addProduct(product) {
   products.push(product);
}
// Function to get all products
function getAllProducts() {
   return products;
}
module.exports = { addProduct, getAllProducts };
```
~~~

Order Management Module:

```

// Data structure to store orders
let orders = [];


// Function to place a new order
function placeOrder(order) {
  orders.push(order);
}


// Function to get all orders
function getAllOrders() {
   return orders;
}


module.exports = { placeOrder, getAllOrders };
```

Example Usage:

~~~
```javascript
// app.js


const { addProduct, getAllProducts } = require('./products');
const { placeOrder, getAllOrders } = require('./orders');


// Adding products
addProduct({ id: 1, name: "Product 1", price: 10 });
addProduct({ id: 2, name: "Product 2", price: 20 });


// Placing orders
~~~

```
placeOrder({ id: 1, products: [{ productId: 1, quantity: 2 }] });

placeOrder({ id: 2, products: [{ productId: 2, quantity: 1 }] });


// Getting all products and orders

console.log("All Products:", getAllProducts());

console.log("All Orders:", getAllOrders());
```

We have separate modules for product management (`products.js`) and order management (`orders.js`). Each module follows the DRY principle by encapsulating related functionality without repeating code. Functions within each module are kept simple and focused on specific tasks, adhering to the KISS principle. In the example usage (`app.js`), we demonstrate how to add products, place orders, and retrieve all products and orders. By structuring our e-commerce project in this way, we maintain code reusability, readability, and simplicity, making it easier to manage and extend the functionality in the future.