Name: Muhammad Shoaib Akhter Qadri

Enrollment No: 02-131212-009

Reg No: 79290

SCENERIO:

StoreJinnie is an online store is looking to implement a new system to manage their inventory, orders, and shipping processes. Following are the main Modules of this system.

• Authentication/Authorization

• User Management: The system will have multiple roles for users. Super admin can manage all other users. • Inventory management: The system will keep track of the store's inventory, including the quantity of each item, location in the warehouse, and restocking notifications. When a new order is received, the system will automatically update the inventory and notify the warehouse manager of any items that need to be restocked.

• Order management: The system will manage customer orders, including payment processing, order confirmation, and order fulfillment. The system will automatically notify the customer of their order confirmation and shipment tracking number.

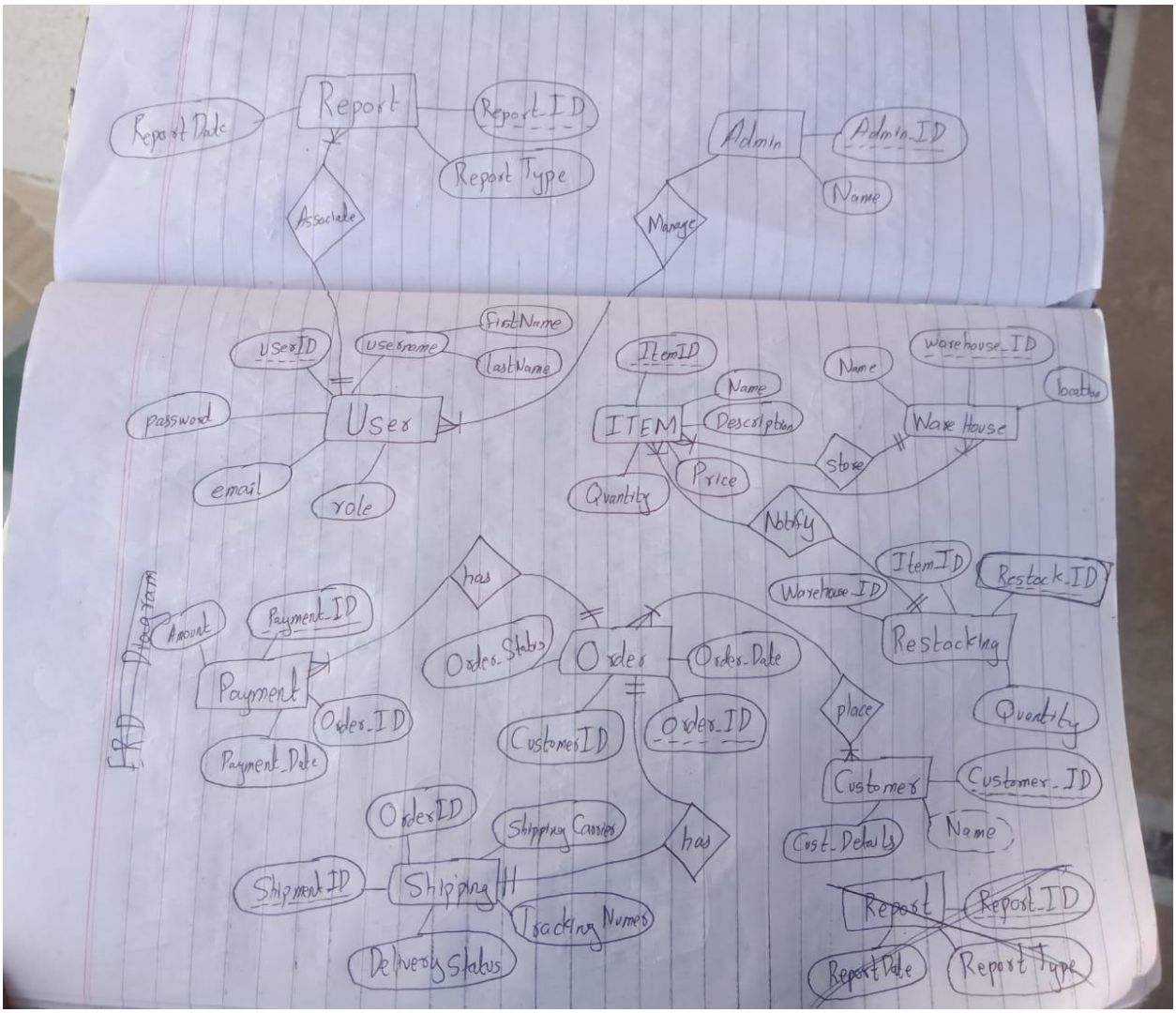
• Shipping management: The system will manage the shipping process, including selecting the best shipping carrier based on price and delivery time, generating shipping labels, and tracking shipments. The system will automatically update the customer of the shipping status and any delays.

• Reporting and analytics: The system will provide detailed reporting and analytics on sales, inventory levels, and order fulfillment performance. The system will allow the store owner to make data-driven decisions to improve their business operations.

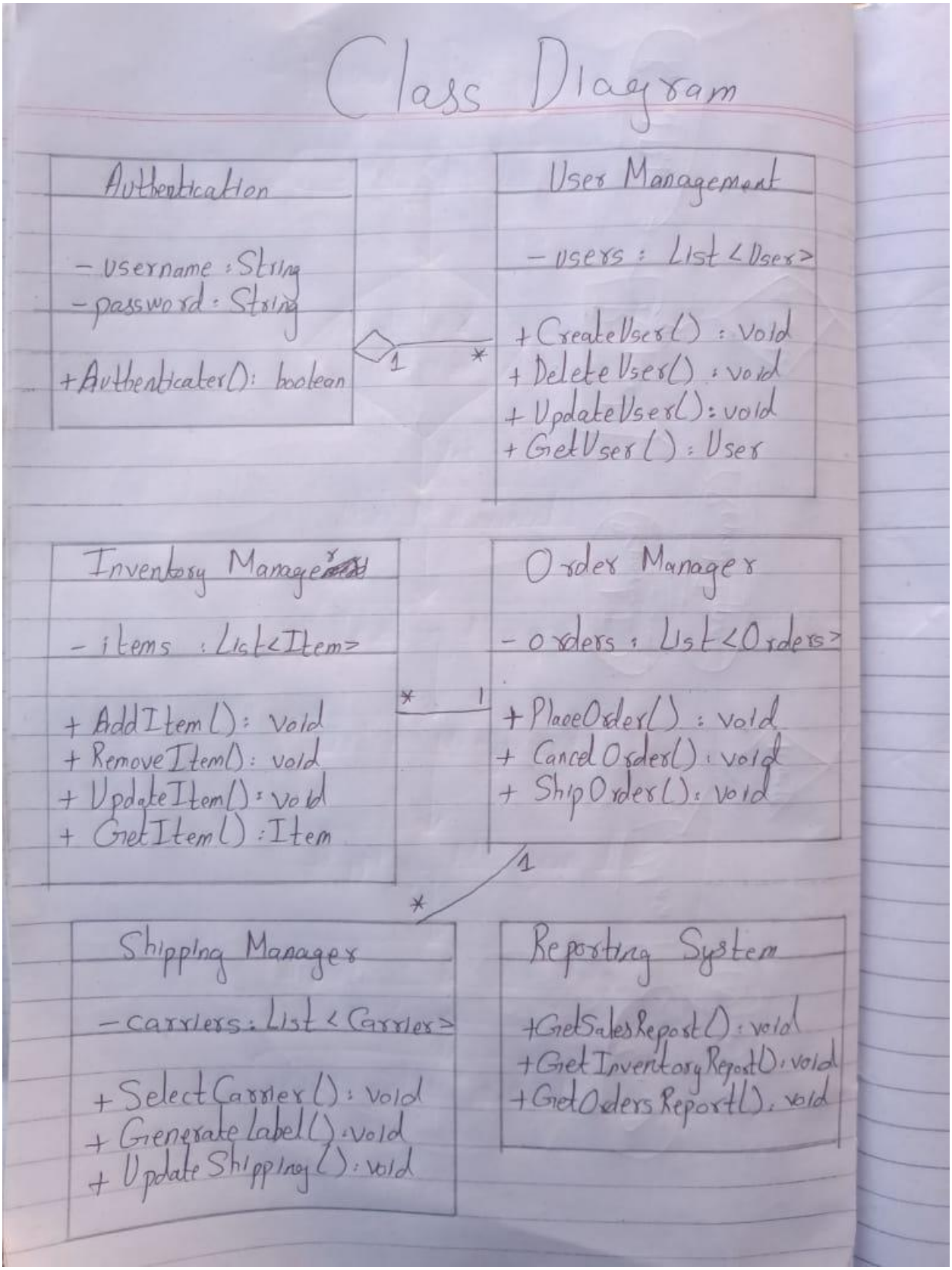
Q1 Design Data flow Diagram, Class Diagram and ERD diagram for the given Scenario.

Answer:

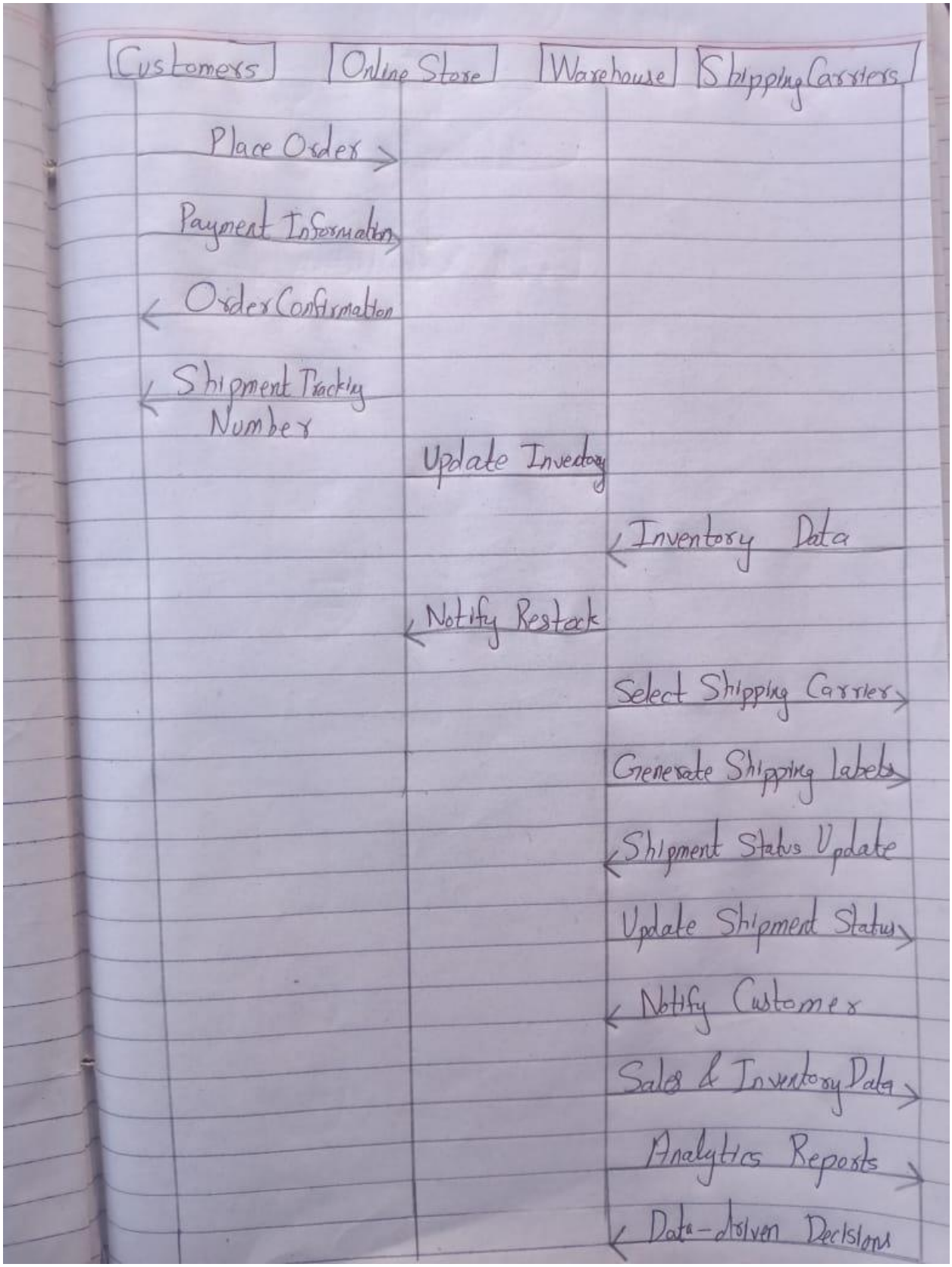
ERD diagram:



Class diagram:



Data FLow diagram:



Q2 Identify and Implement best suited design patterns for the given scenario with Reasons (Note: Multiple Design patterns can be implemented) :

Answer:

Authentication/Authorization:

Design Patterns:

We use these Design Patterns for Authentication and Authorization in which are following:

- Singleton design pattern,
- Prototype design pattern,
- Abstract Design Pattern,
- Decorator Design Pattern,
- Façade Design Pattern,
- Flyweight Design Pattern

Source Code:

This Login and Registration form is made by using MERN technology and the jwt web token and bcrypt is also used.

Here is the source code of design pattern:

Singleton Design Pattern:

```
class AuthenticationManager {
  constructor() {
    if (AuthenticationManager.instance) {
      return AuthenticationManager.instance; }
    AuthenticationManager.instance = this;
    this.loggedInUser = null; }
```

```
  login(username, password) {
    if (username === "admin" && password === "admin") {
      this.loggedInUser = {
        username: username,
        role: "admin"; };
      console.log("User logged in successfully");
      return true;
    } else {
      console.log("Invalid credentials");
      return false;
    }
  }
  logout() {
    this.loggedInUser = null;
    console.log("User logged out successfully");
  }
  getLoggedInUser() {
    return this.loggedInUser;
  }
}
```

Prototype Design Pattern:

```
const User = function (username, password) {
  this.username = username;
  this.password = password;
};
User.prototype.validateCredentials = function () {
  if (this.username && this.password) {
    return true;
  } else {
    return false;
  }
};
```

Abstract Design Pattern:

```
class Authenticator {
  authenticate(user) {}
}
class LoginAuthenticator extends Authenticator {
  authenticate(user) {
    const authenticationManager = new AuthenticationManager();
    if (user.validateCredentials()) {
      return authenticationManager.login(user.username, user.password);
    } else {
      console.log("Invalid credentials");
```

```
      return false;
    }
  }
}
class LogoutAuthenticator extends Authenticator {
  authenticate() {
    const authenticationManager = new AuthenticationManager();
    authenticationManager.logout();
    return true;
  }
}
```

Decorator Design Pattern:

```
class LoggingAuthenticatorDecorator extends Authenticator {
  constructor(authenticator) {
    super();
    this.authenticator = authenticator;
  }
  authenticate(user) {
    console.log("Authenticating user: ", user.username);
    const result = this.authenticator.authenticate(user);
    console.log("Authentication result: ", result);
    return result;
  }
```

```
}
```

Façade Design Pattern:

```
class AuthenticationFacade {
  constructor() {
    this.authenticationManager = new AuthenticationManager();
    this.loginAuthenticator = new LoginAuthenticator();
    this.logoutAuthenticator = new LogoutAuthenticator();
    this.loggingAuthenticatorDecorator = new LoggingAuthenticatorDecorator(
      this.loginAuthenticator
    );
  }
  login(username, password) {
    const user = new User(username, password);
    return this.loggingAuthenticatorDecorator.authenticate(user);
  }
  logout() {
    return this.logoutAuthenticator.authenticate();
  }
  getLoggedInUser() {
    return this.authenticationManager.getLoggedInUser();
  }
}
```

Flyweight Design Pattern:

```
const AuthenticationFlyweightFactory = (function () {
  const authenticators = {};
  function createAuthenticator(type) {
    if (authenticators[type]) {
      return authenticators[type];
    } else {
      let authenticator;
      switch (type) {
        case "login":
          authenticator = new LoginAuthenticator();
          break;
        case "logout":
          authenticator = new LogoutAuthenticator();
          break;
        default:
          throw new Error("Invalid authenticator type");
      }
      authenticators[type] = authenticator;
      return authenticator;
    }
  }
  return {
    createAuthenticator,
```

```
  };
})();
const authenticationFacade = new AuthenticationFacade();
authenticationFacade.login("admin", "admin");
console.log(authenticationFacade.getLoggedInUser());
authenticationFacade.logout();
console.log(authenticationFacade.getLoggedInUser());
```

Web Pages (Add to Cart , Inventory, Orders, Shipping Processes, Reporting and analytics):

Here is also used multiple design patterns in above categories in which are following:

- Singleton design pattern,
- Prototype design pattern,
- Abstract Design Pattern,
- Decorator Design Pattern,
- Façade Design Pattern,
- Flyweight Design Pattern

Singleton design pattern

```
const Cart = (() => {
  let instance;
  function createInstance() {
    const cart = [];
    function addItem(item) {
      cart.push(item);
```

```
    console.log(`Item ${item.name} added to cart.`);
  }
  function removeItem(item) {
    const index = cart.indexOf(item);
    if (index !== -1) {
      cart.splice(index, 1);
      console.log(`Item ${item.name} removed from cart.`);
    }
  }

  function getCart() {
    return cart;
  }
  return {
    addItem,
    removeItem,
    getCart
  };
}
return {
  getInstance: () => {
    if (!instance) {
      instance = createInstance();
    }
```

```
    return instance;
  }
};
})();
```

Prototype design pattern

```
class Product {
  constructor(name, price, description) {
    this.name = name;
    this.price = price;
    this.description = description;
  }

  clone() {
    return new Product(this.name, this.price, this.description);
  }
}

const productPrototype = new Product('', 0, '');
```

Abstract design pattern

```
class Inventory {
  constructor() {
    if (new.target === Inventory) {
```

```
    throw new TypeError('Cannot instantiate abstract class Inventory');
  }
  this.products = [];
}

addProduct(product) {
  this.products.push(product);
  console.log(`Product ${product.name} added to inventory.`);
}

removeProduct(product) {
  const index = this.products.indexOf(product);
  if (index !== -1) {
    this.products.splice(index, 1);
    console.log(`Product ${product.name} removed from inventory.`);
  }
}

getProduct(name) {
  return this.products.find(p => p.name === name);
}

getAllProducts() {
  return this.products;
```

```
  }
}

class WarehouseInventory extends Inventory {
  constructor() {
    super();
  }

  restockProduct(product, quantity) {
    const p = this.getProduct(product.name);
    if (p) {
      p.quantity += quantity;
      console.log(`Product ${p.name} restocked by ${quantity} units.`);
    }
  }
}
```

Decorator design pattern

```
class Order {
  constructor(customer, products) {
    this.customer = customer;
    this.products = products;
  }
```

```
  getTotalPrice() {
    return this.products.reduce((acc, p) => acc + p.price, 0);
  }
}

class ShippingOrder {
  constructor(order) {
    this.order = order;
  }

  getTotalPrice() {
    return this.order.getTotalPrice() + 5; // shipping fee
  }

  ship() {
    console.log('Order shipped.');
  }
}
```

Facade design pattern

```
class OrderFacade {
  constructor(customer, product) {
    this.customer = customer;
    this.product = product;
```

```
  }

  placeOrder() {
    const order = new Order(this.customer, [this.product]);
    const shippingOrder = new ShippingOrder(order);

    console.log('Order placed.');
    console.log(`Total price: ${shippingOrder.getTotalPrice()}`);
    shippingOrder.ship();
  }
}
```

Flyweight design pattern

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

  generateLabel(order) {
    console.log(`Label generated by ${this.name} for order ${order}`);
  }
}

class ShippingCarrierFactory {
```

```
  constructor() {
    this.carriers = {};
  }

  getShippingCarrier(name) {
    if (!this.carriers[name]) {
      this.carriers[name] = new ShippingCarrier(name);
```

Abstract Factory pattern to create different types of reports:

```
class ReportFactory {
  createReport(reportType) {
    switch(reportType) {
      case 'sales':
        return new SalesReport();
      case 'inventory':
        return new InventoryReport();
      case 'order':
        return new OrderReport();
      default:
        throw new Error(`Invalid report type: ${reportType}`);
    }
  }
}
```

Factory method pattern to create instances of report data objects:

```
class ReportDataFactory {
  createReportData(reportType, startDate, endDate) {
    switch(reportType) {
      case 'sales':
        return new SalesReportData(startDate, endDate);
      case 'inventory':
        return new InventoryReportData(startDate, endDate);
      case 'order':
        return new OrderReportData(startDate, endDate);
      default:
        throw new Error(`Invalid report type: ${reportType}`);
    }
  }
}
```

Singleton pattern to ensure only one instance of the reporting service exists:

```
class ReportingService {
  constructor(reportFactory, reportDataFactory) {
    if (ReportingService.instance) {
      return ReportingService.instance;
    }
    this.reportFactory = reportFactory;
```

```
    this.reportDataFactory = reportDataFactory;
    ReportingService.instance = this;
  }

  generateReport(reportType, startDate, endDate) {
    const report = this.reportFactory.createReport(reportType);
    const reportData = this.reportDataFactory.createReportData(reportType,
startDate, endDate);
    report.generate(reportData);
  }
}
```

Decorator pattern to add additional functionality to the report objects:

```
class ReportDecorator {
  constructor(report) {
    this.report = report;
  }

  generate(reportData) {
    this.report.generate(reportData);
    this.addFooter();
  }

  addFooter() {
```

```
    console.log('Footer added to report.');
  }
}
```

Concrete report classes:

```
class SalesReport {
  generate(reportData) {
    console.log(`Generating sales report for ${reportData.startDate} - ${reportData.endDate}`);
    // Code to generate report
  }
}

class InventoryReport {
  generate(reportData) {
    console.log(`Generating inventory report for ${reportData.startDate} - ${reportData.endDate}`);
    // Code to generate report
  }
}

class OrderReport {
  generate(reportData) {
    console.log(`Generating order report for ${reportData.startDate} - ${reportData.endDate}`);
```

```
    // Code to generate report
  }
}
```

Concrete report data classes:

```
class SalesReportData {
  constructor(startDate, endDate) {
    this.startDate = startDate;
    this.endDate = endDate;
  }
}

class InventoryReportData {
  constructor(startDate, endDate) {
    this.startDate = startDate;
    this.endDate = endDate;
  }
}

class OrderReportData {
  constructor(startDate, endDate) {
    this.startDate = startDate;
    this.endDate = endDate;
  }}
```

Example for Usage

```
const reportFactory = new ReportFactory();

const reportDataFactory = new ReportDataFactory();

const reportingService = new ReportingService(reportFactory, reportDataFactory);
```

Generate sales report for the current month

```
const salesReport = reportingService.generateReport('sales', '2023-05-01', '2023-05-31');
```

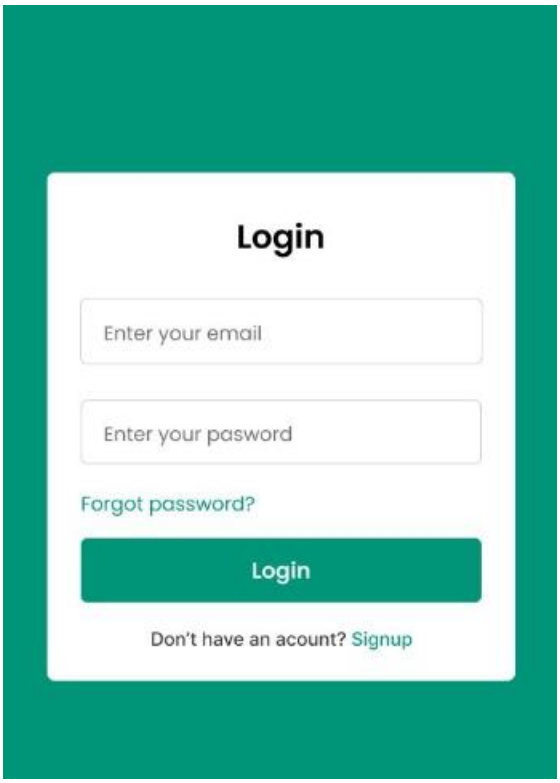
Generate inventory report for the current quarter

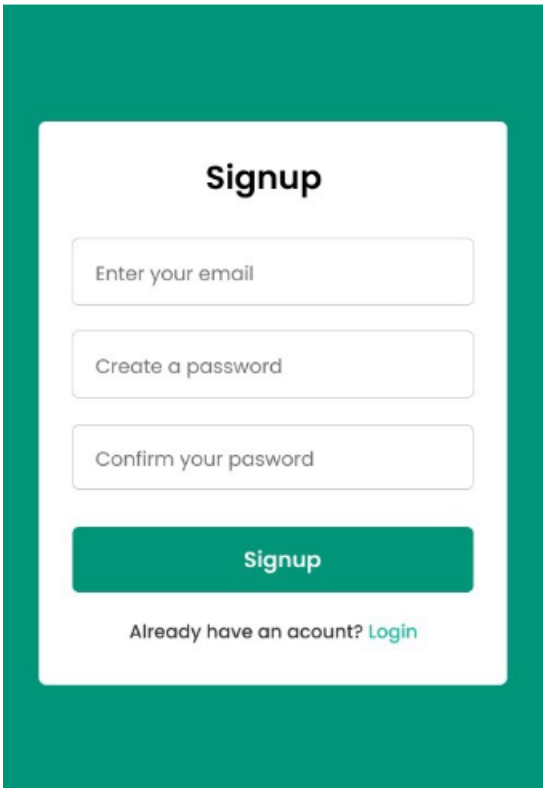
```
const inventoryReport = reportingService.generateReport('inventory', '2023-04-01', '2023-06-30');
```

Generate order report for the current year

```
const orderReport = reportingService.generateReport('order', '2023
```

Q3. Design UI for the system.





HOME SHOP OUR STORY WHERE TO BUY RECIPES NEWS CONTACT

A CHANCE TO GET:

- Great snacks containing organic ingredients
- Paleo snacks
- Dairy-free snacks
- Gluten-free snacks

SHOP NOW

HOME SHOP OUR STORY WHERE TO BUY RECIPES NEWS CONTACT

Search products...

SEARCH

CATEGORIES

- Cookie Bites
- Gifts
- Grain Free
- Nut Butter
- Oatmeal Cups
- Snacks
- Vegan Brittle

Default sorting

VIEW: 12 / 24 / ALL

NUT BUTTER

ALMOND BUTTER

$8.00

ADD TO CART

COOKIE BITES, GRAIN FREE

ALMOND BUTTER COOKIE BITES

$6.00 – $32.00

SELECT OPTIONS

GRAIN FREE, NUT BUTTER

BANANA BREAD NUT BUTTER

$14.00

ADD TO CART

Billing details

First name *: Szymon

Last name *: Barczak

Company name (optional): WP Desk

Country *: United Kingdom (UK)

Street address *: Test

Apartment, suite, unit etc. (optional)

Town / City *: Rotherham

State / County (optional): United Kingdom (UK)

Your order

Product	Total
Sweatshirt × 3	£126.00
Subtotal	£126.00
Shipping	My awesome shipping method: £10.00 The best shipping method you have ever seen!
Total	£136.00

Direct bank transfer

HOME SHOP OUR STORY WHERE TO BUY RECIPES

CONTACT US

Name *

First Last

Email *

What are you contacting us about?

Wholesale

Message *

********* BEST OF LUCK *********