HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

School of Information and communications technology

Software Design Document

Version 1.3

AIMS

Subject: IT4490

Vũ Việt Anh – 20215261

*Hanoi,* *<month, year>*

*<All notations inside the angle bracket are not part of this document, for its purpose is for extra instruction. When using this document, please erase all these notations and/or replace them with corresponding content as instructed>*

*<This document, written by Prof. NGUYEN Thi Thu Trang, is used as a case study for student with related courses. Any modifications and/or utilization without the consent of the author is strictly forbidden>*

Table of Contents

Table of Contents 1

1 Introduction 5

1.1 Objective 5

1.2 Scope 5

1.3 Glossary 5

1.4 References 5

2 Overall Description 7

2.1 General Overview 7

2.2 Assumptions/Constraints/Risks 7

2.2.1 Assumptions 7

2.2.2 Constraints 7

2.2.3 Risks 8

3 System Architecture and Architecture Design 9

3.1 Architectural Patterns 9

3.2 Interaction Diagrams 9

3.3 Analysis Class Diagrams 9

3.4 Unified Analysis Class Diagram 9

3.5 Security Software Architecture 9

4 Detailed Design 10

4.1 User Interface Design 10

4.1.1 Screen Configuration Standardization 10

4.1.2 Screen Transition Diagrams 10

4.1.3 Screen Specifications 10

4.2 Data Modeling 10

4.2.1 Conceptual Data Modeling 10

4.2.2 Database Design 10

4.3 Non-Database Management System Files 11

4.4 Class Design 11

4.4.1 General Class Diagram 11

4.4.2 Class Diagrams 11

4.4.3 Class Design 11

5 Design Considerations 13

5.1 Goals and Guidelines 13

5.2 Architectural Strategies 13

5.3 Coupling and Cohesion 14

5.4 Design Principles 14

5.5 Design Patterns 14

**List of Figures**

[Figure 1: Place order 13](#_Toc180600818)

[Figure 2: UC Pay Order 14](#_Toc180600819)

[Figure 3: UC Place Rush Order 15](#_Toc180600820)

[Figure 4: Analysic Class Diagram for UC Place Order 16](#_Toc180600821)

[Figure 5: Analysic Class Diagram for UC Pay Order 17](#_Toc180600822)

[Figure 6: Analysic Class Diagram for UC Place Rush Order 18](#_Toc180600823)

[Figure 7: Unified Analysis Class Diagram 18](#_Toc180600824)

[Figure 8: General Class Diagram 22](#_Toc180600825)

[Figure 9: Class diagram for package view.cart 23](#_Toc180600826)

[Figure 10: Class Diagram for Package view.fail 23](#_Toc180600827)

[Figure 11: Class Diagram for Package view.order 24](#_Toc180600828)

[Figure 12: Class Diagram for Package view.payment 24](#_Toc180600829)

[Figure 13: Class Diagram for Package controller 25](#_Toc180600830)

[Figure 14: Class Diagram for Package entity.cart 25](#_Toc180600831)

[Figure 15: Class Diagram for Package entity.order 26](#_Toc180600832)

[Figure 16: Class Diagram for Package entity.payment 26](#_Toc180600833)

[Figure 17: Class "PayOrderController" 27](#_Toc180600834)

[Figure 18: Class "PlaceOrderController" 28](#_Toc180600835)

[Figure 19: Class "PlaceRushOrderController" 30](#_Toc180600836)

[Figure 20: Class "Cart" 31](#_Toc180600837)

[Figure 21: Class "Product" 32](#_Toc180600838)

[Figure 22: Class " DeliveryInfo " 34](#_Toc180600839)

[Figure 23: Class " Order" 35](#_Toc180600840)

[Figure 24: Class "Invoice" 37](#_Toc180600841)

[Figure 25: Class "TransactionInfo" 39](#_Toc180600842)

[Figure 26: Class "CartScreen" 41](#_Toc180600843)

[Figure 27: Class "FailedRushOrderScreen" 42](#_Toc180600844)

[Figure 28: Class "FailedPaymentScreen" 43](#_Toc180600845)

[Figure 29: Class "AdditionalInfoBoundary" 44](#_Toc180600846)

[Figure 30: Class "DeliveryForm" 45](#_Toc180600847)

[Figure 31: Class "PlaceRushOrderScreen" 46](#_Toc180600848)

[Figure 32: Class "InvoiceScreen" 47](#_Toc180600849)

[Figure 33: Class "VNPayBoundary" 48](#_Toc180600850)

**List of Tables**

[Table 1. Example of table design 21](#_Toc180600851)

[**Table 2. Attribute design** 28](#_Toc180600852)

[**Table 3. Operation design** 28](#_Toc180600853)

[**Table 4. Attribute design** 29](#_Toc180600854)

[**Table 5. Operation design** 30](#_Toc180600855)

[**Table 6. Attribute design** 31](#_Toc180600856)

[**Table 7. Operation design** 31](#_Toc180600857)

[**Table 8. Attribute design** 32](#_Toc180600858)

[**Table 9. Operation design** 32](#_Toc180600859)

[**Table 10. Attribute design** 33](#_Toc180600860)

[**Table 11. Operation design** 34](#_Toc180600861)

[**Table 12. Attribute design** 35](#_Toc180600862)

[**Table 13. Operation design** 35](#_Toc180600863)

[**Table 14. Attribute design** 36](#_Toc180600864)

[**Table 15. Operation design** 37](#_Toc180600865)

[**Table 16. Attribute design** 38](#_Toc180600866)

[**Table 17. Operation design** 39](#_Toc180600867)

[**Table 18. Attribute design** 40](#_Toc180600868)

[**Table 19. Operation design** 41](#_Toc180600869)

[**Table 20. Attribute design** 42](#_Toc180600870)

[**Table 21. Operation design** 42](#_Toc180600871)

[**Table 22. Attribute design** 43](#_Toc180600872)

[**Table 23. Operation design** 43](#_Toc180600873)

[**Table 24. Attribute design** 44](#_Toc180600874)

[**Table 25. Operation design** 44](#_Toc180600875)

[**Table 26. Attribute design** 45](#_Toc180600876)

[**Table 27. Operation design** 45](#_Toc180600877)

[**Table 28. Attribute design** 46](#_Toc180600878)

[**Table 29. Operation design** 46](#_Toc180600879)

[**Table 30. Attribute design** 47](#_Toc180600880)

[**Table 31. Operation design** 47](#_Toc180600881)

[**Table 32. Attribute design** 48](#_Toc180600882)

[**Table 33. Operation design** 48](#_Toc180600883)

[**Table 34. Attribute design** 49](#_Toc180600884)

[**Table 35. Operation design** 49](#_Toc180600885)

# Introduction

*<The following subsections of the Software Design Document (SDD) document should provide an overview of the entire SDD.>*

## Objective

This document provides an analysis and modeling of the use cases for the AIMS – An Internet Media Store software. The document outlines the purpose and features of the system, as well as the constraints the system must meet to respond to external stimuli.

This document is intended for stakeholders and software developers.

## Scope

AIMS is a cross-platform system that operates 24/7, allowing new users to quickly get familiar with it. The system is capable of serving multiple customers simultaneously without significant performance degradation.

The AIMS system allows product managers to add, view, edit, and delete products. Each product must have information such as the product name, type, value, current price, and special information depending on the product type (e.g., books, CDs, DVDs). Product managers are allowed to change the product price up to 2 times per day, and the product price must be within 30% to 150% of the product's value.

Administrators are allowed to manage all the users of the systems, comprising product managers, and manage products.

The AIMS system stores the history of adding, editing, and deleting products and notifies administrators of invalid actions.

Customers can search and view product listings, sort products by price, or add products to the shopping cart. The shopping cart displays product information, pricing, and shipping fees. Customers can choose between standard or express shipping options.

To successfully place an order, customers need to update shipping information and make a payment. The payment method is through credit cards, and the system supports payments in the local currency (VND). After successful payment, the system displays a transaction code and stores the order information.

Administrators have the right to approve or reject orders. An order can be canceled after payment, and the money will be refunded to the customer.

In the simulation system, customers do not need to create an account to place an order. The system uses virtual credit cards for payment and provides APIs to manage virtual credit cards.

## Definitions, Acronyms, and Abbreviations

* **AIMS**: Automated Inventory Management System.
* **UC**: Use Case.
* **VNPay**: Third-party payment gateway for processing transactions.
* **Rush order:** refers to a request for expedited processing of an order, meaning the order needs to be completed and delivered faster than usual.

## References

|  |  |
| --- | --- |
| [1] | Centers for Medicare & Medicaid Services, "System Design Document Template," [Online]. Available: https://www.cms.gov/Research-Statistics-Data-and-Systems/CMS-Information-Technology/XLC/Downloads/SystemDesignDocument.docx. |

# Overall Description

<*This section describes the principles and strategies to be used as guidelines when designing and implementing the system.>*

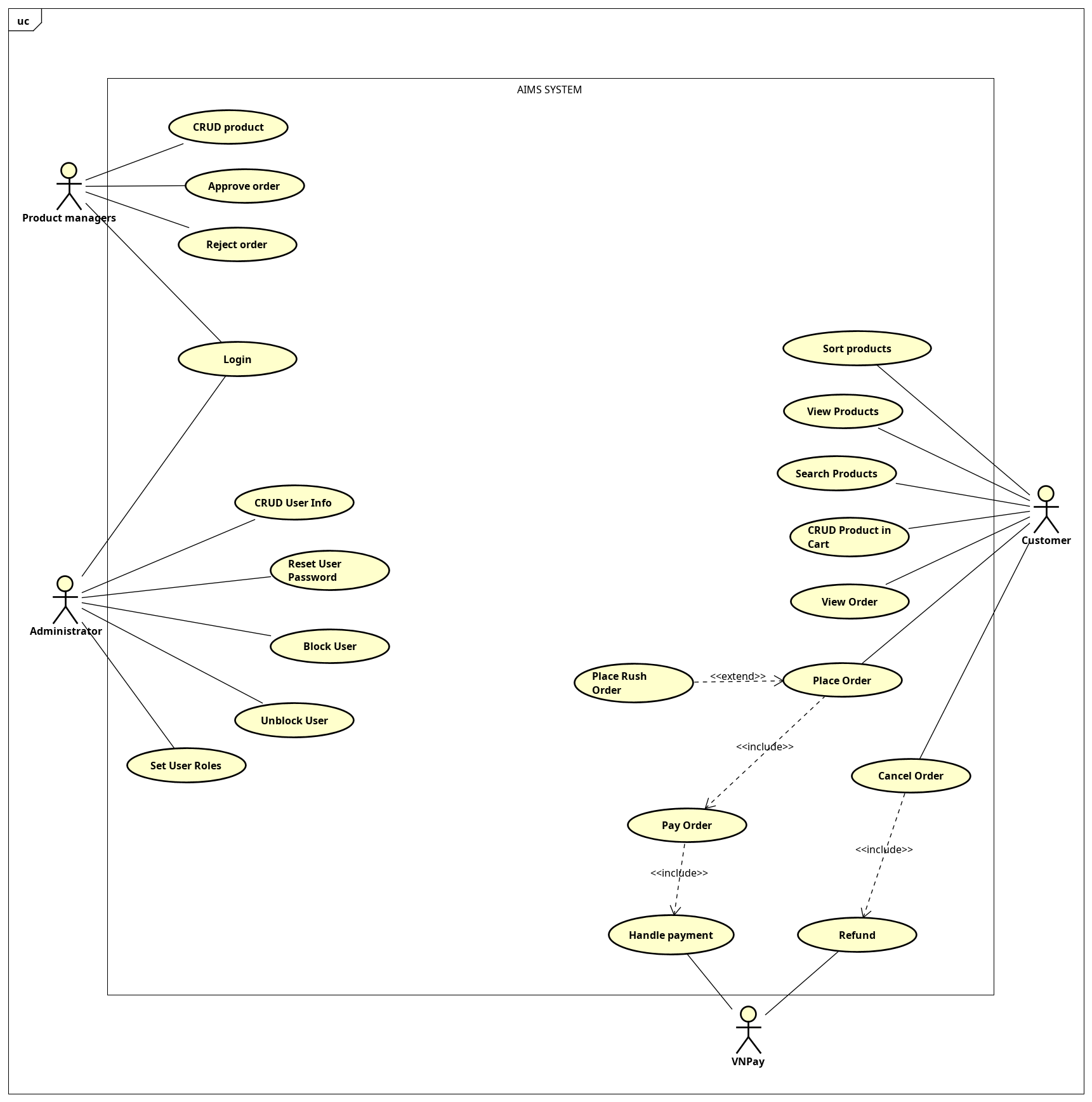
## General Overview

The AIMS system is designed to operate as a cross-platform system running 24/7, with a primary focus on usability and scalability. The system architecture allows for multiple concurrent users without a noticeable impact on performance. It serves both administrators and customers, providing interfaces for managing products, handling transactions, and processing payments.

The core functionalities include:

* **Product Management**: Product managers can add, view, edit, and delete products. Each product has detailed information, including specific attributes like name, type, value, and special characteristics depending on the product category (books, CDs, DVDs). Prices can be updated up to twice per day and must remain within 30% to 150% of the product's value.
* **Customer Interaction**: Customers can search for products, sort them by price, and manage a shopping cart. They can select between standard and express shipping options and complete transactions through credit card payments.
* **Order Management**: Administrators can approve, reject, or cancel orders, with a full audit trail of product changes. Order cancellations trigger automatic refunds.

Key architectural goals include ease of use, robust concurrency handling, and integration with virtual credit card APIs for simulation purposes.



## Assumptions/Constraints/Risks

### Assumptions

 The system operates in a mixed hardware environment (e.g., mobile platforms).

 Users have basic internet access and devices capable of handling online shopping.

 All transactions will be processed using credit cards in the local currency (VND).

 The system assumes stable internet connections for transactions to succeed without errors.

 Future functionalities might be required to handle international payments, but this is not within the current scope.

### Constraints

 **Hardware/Software Environment**: The system must be compatible with multiple platforms (web and mobile) and operate within modern web browsers.

 **End-user Environment**: Users must be able to quickly navigate the system, requiring a simple, intuitive user interface.

 **Resource Availability**: System resources (e.g., database, processing power) must handle simultaneous multiple users without degradation in performance.

 **Standards Compliance**: The system must adhere to relevant security standards (e.g., PCI DSS for payment processing).

 **Interoperability**: APIs for handling virtual credit card payments must be compatible with existing systems and easily extensible.

 **Data Security**: Secure storage of transaction and user data is paramount, following encryption standards and data protection regulations.

 **Performance Requirements**: The system must provide a smooth experience with quick loading times and responsive interfaces, even under heavy traffic.

 **Network Communications**: Efficient handling of network traffic is necessary, especially during payment processing and order updates.

###  ****Verification and Validation****: Regular testing is required to ensure the system functions as expected across all supported platforms and under different user loads.

### Risks

*<Describe any risks associated with the system design and proposed mitigation strategies.>*

 **Payment Gateway Failures**: The system’s reliance on external payment services may cause downtime or transaction failures. Mitigation: Implementing fallback mechanisms and user notifications during payment issues.

 **High Traffic Impact**: During peak usage times, system performance may degrade. Mitigation: Ensure the system is scalable, using load balancing and caching where necessary.

 **Data Breaches**: Unauthorized access to sensitive data is a risk. Mitigation: Strong encryption and regular security audits will help mitigate this risk.

 **Pricing Errors**: Incorrect price adjustments beyond allowed limits may occur. Mitigation: Implement checks and notifications for product managers to prevent such errors.

# System Architecture and Architecture Design

This section describes the overall architecture and design steps for the system, outlining the principles and strategies used in the implementation.

## Architectural Patterns

The chosen architectural pattern for this system is **MVC (Model-View-Controller)**.

* **Model**: This layer is responsible for managing the data and business logic of the system. It communicates with the database and processes any data-related operations.
* **View**: The view layer is responsible for presenting the data to the user in a specific format. It handles the UI and user interactions without holding any business logic.
* **Controller**: The controller acts as an intermediary between the model and view layers. It listens to user inputs, processes requests, and sends responses back to the view.

**Reason for Choosing MVC**:

* **Separation of concerns**: MVC separates the logic of the system, allowing independent development of UI, business logic, and data management.
* **Scalability**: With the separation of components, the system can be scaled efficiently.
* **Maintainability**: Modularized structure makes it easier to update or modify the system as requirements evolve.
* **Reusability**: Components within MVC can be reused across different parts of the application or even in other projects.

## Interaction Diagrams

### Sequence Diagrams

#### UC Place Order

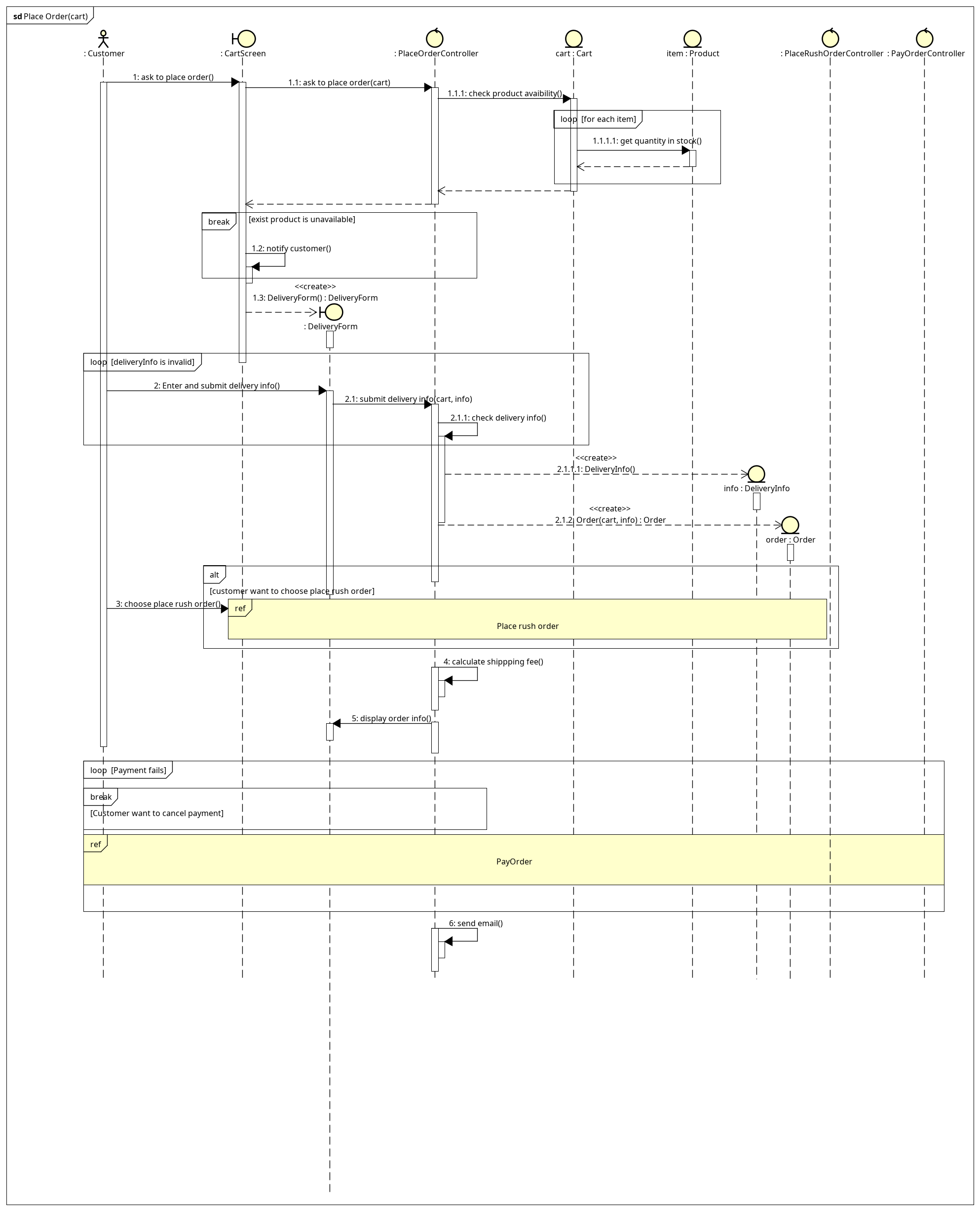


Figure 1: Place order

#### UC Pay Order

A diagram of a diagram

Description automatically generated

Figure 2: UC Pay Order

#### UC Place Rush Order

A diagram of a project

Description automatically generated

Figure 3: UC Place Rush Order

## Analysis Class Diagrams

#### UC Place Order

A diagram of a diagram

Description automatically generated with medium confidence

Figure 4: Analysic Class Diagram for UC Place Order

#### UC Pay Order

A diagram of a company

Description automatically generated

Figure 5: Analysic Class Diagram for UC Pay Order

#### UC Place Rush Order

A diagram of a company

Description automatically generated

Figure 6: Analysic Class Diagram for UC Place Rush Order

## Unified Analysis Class Diagram

A diagram of a computer flowchart

Description automatically generated

Figure 7: Unified Analysis Class Diagram

## Security Software Architecture

*<Describe the software components and configuration supporting the security and privacy of the system. Specify the architecture for (1) authentication to validate user identity before allowing access to the system;(2) authorization of users to perform functional activity once logged into the system, (3) encryption protocol to support the business risks and the nature of information, and (4) logging and auditing design, if required.>*

# Detailed Design

## User Interface Design

*<Suppose that you design a Graphical User Interface (GUI)>*

### Screen Configuration Standardization

### Screen Transition Diagrams

### Screen Specifications

*<Screen images should be included in the screen specifications>*

## Data Modeling

### Conceptual Data Modeling

*<E-R Diagram image and description of entities and relationships>*

### Database Design

#### Database Management System

*<Specify what is the decision of Database Management System (DBMS) and give some description of the DBMS>*

#### Database Diagram

<

* *Show the process to design database from E-R diagram*
* *Show the diagram of DB design*

*>*

#### Database Detail Design

<

*Give a detail design of each element in the DB diagram. For instance, in a Relational DBMS, give a detail design for each Table and their constraints, illustrated in below table (PK: Primary Key, FK: Foreign Key).*

Table 1. Example of table design

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *#* | *PK* | *FK* | *Column name* | *Data type* | *Default value* | *Mandatory* | *Description* |
| 1 | x |  | ProductID |  |  |  |  |
| 2 |  | x | CategoryID |  |  |  |  |

*You may add indexing, trigger, view, etc.*

*Give the database script*>

## Non-Database Management System Files

*<Provide the detailed description of all non-DBMS files if any and include a narrative description of the usage of each file that identifies if the file is used for input, output, or both, and if the file is a temporary file. Also provide an indication of which modules read and write the file and include file structures (refer to the data dictionary). As appropriate, the file structure information should include the following:*

*• Record structures, record keys or indexes, and data elements referenced within the records*

*• Record length (fixed or maximum variable length) and blocking factors*

*• Access method (e.g., index sequential, virtual sequential, random access, etc.)*

*• Estimate of the file size or volume of data within the file, including overhead resulting from file access methods*

*• Definition of the update frequency of the file (If the file is part of an online transaction-based system, provide the estimated number of transactions per unit of time, and the statistical mean, mode, and distribution of those transactions.)*

*• Backup and recovery specifications>*

## Class Design

### General Class Diagram

<General class diagram which shows the whole class diagram of the software. This diagram may have packages, subsystems and classes. Classes in this diagram may not have all attributes and operations>

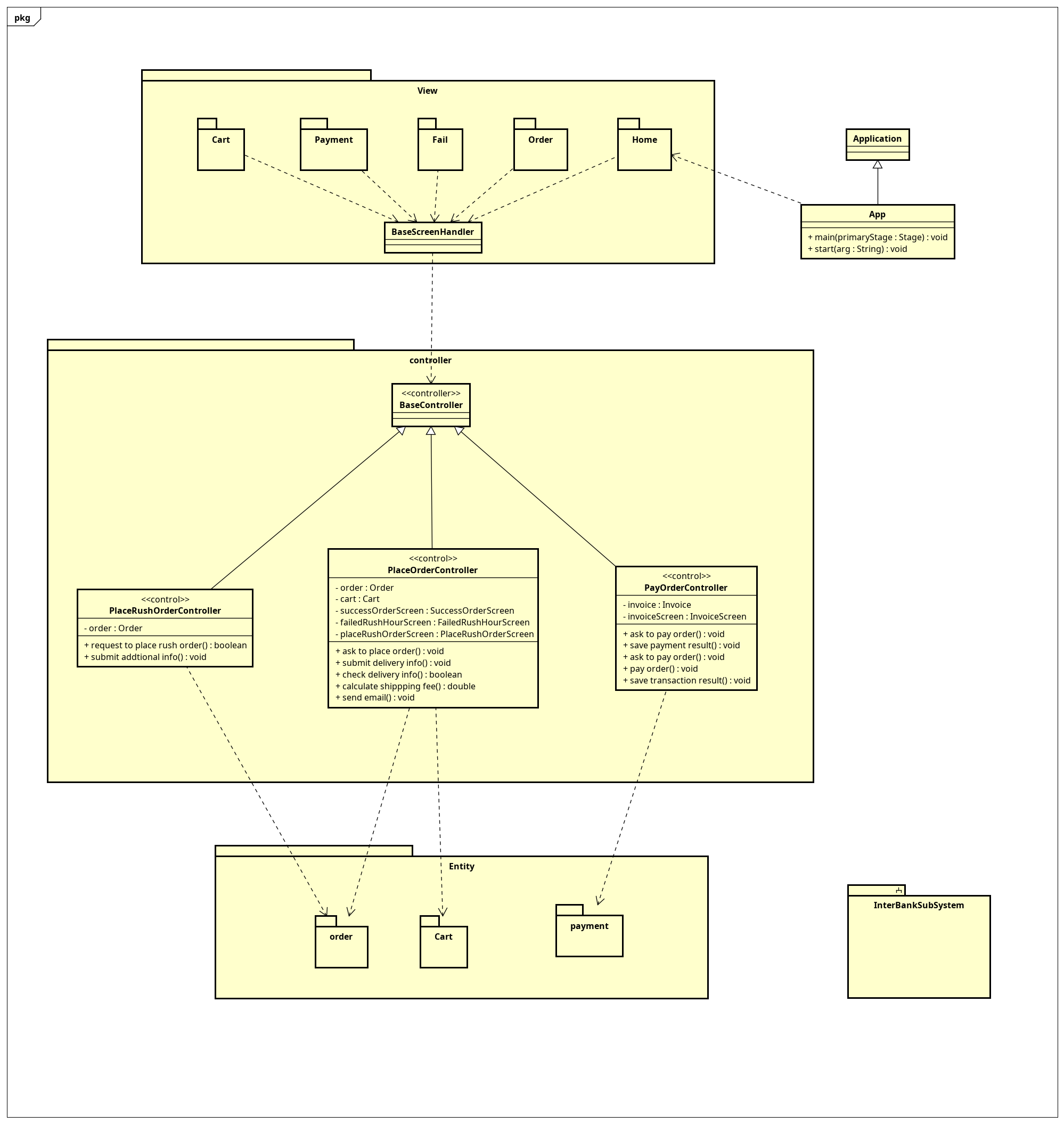


Figure 8: General Class Diagram

### Class Diagrams

#### Class Diagram for Package view.cart

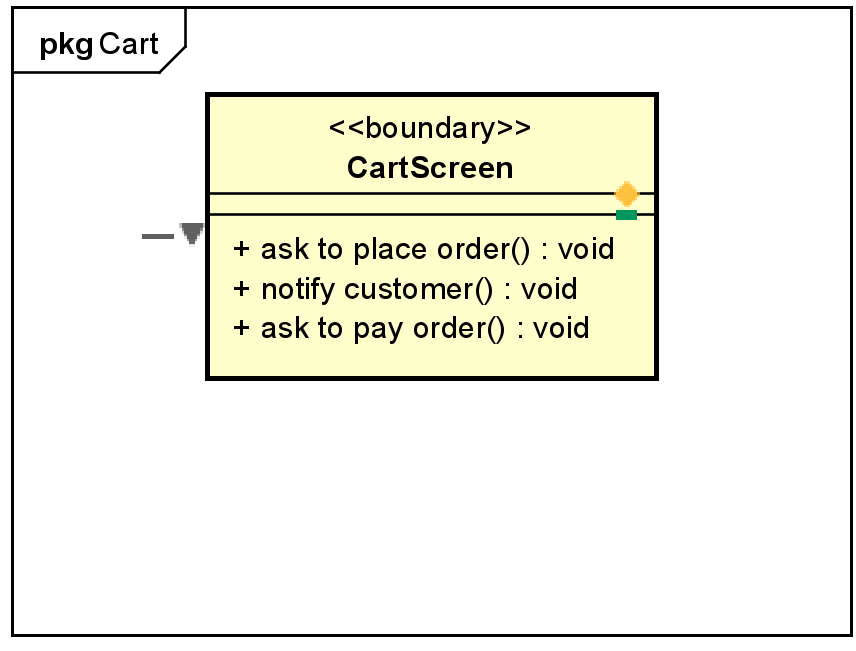


Figure 9: Class diagram for package view.cart

#### Class Diagram for Package view.fail

A screenshot of a computer

Description automatically generated

Figure 10: Class Diagram for Package view.fail

#### Class Diagram for Package view.order

A screenshot of a computer

Description automatically generated

Figure 11: Class Diagram for Package view.order

#### Class Diagram for Package view.payment

A close-up of a computer screen

Description automatically generated

Figure 12: Class Diagram for Package view.payment

#### Class Diagram for Package controller

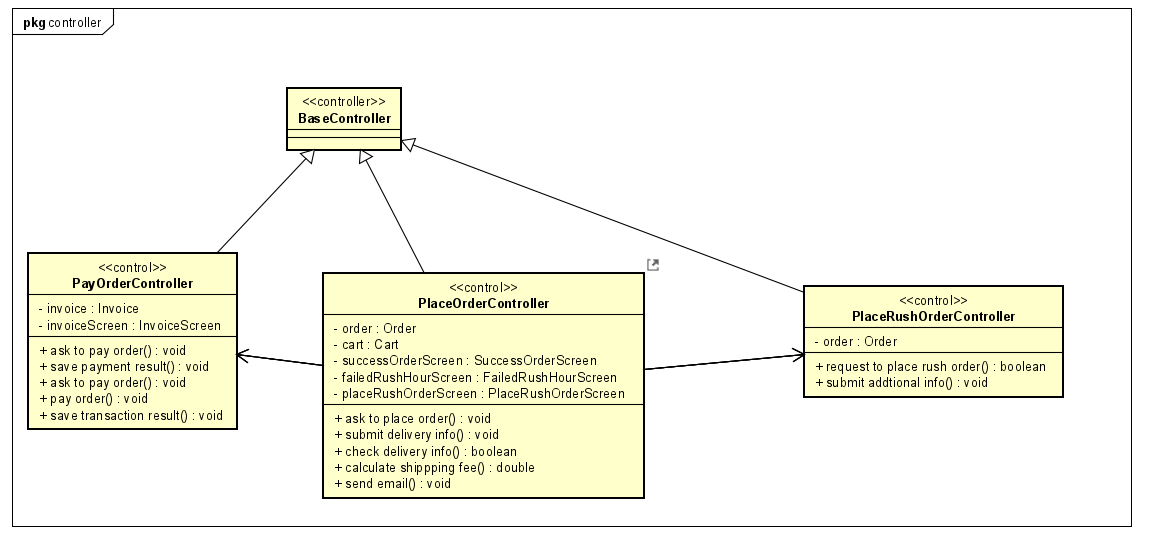


Figure 13: Class Diagram for Package controller

#### Class Diagram for Package entity.cart

A screenshot of a computer

Description automatically generated

Figure 14: Class Diagram for Package entity.cart

#### Class Diagram for Package entity.order

A screenshot of a computer

Description automatically generated

Figure 15: Class Diagram for Package entity.order

#### Class Diagram for Package entity.payment

A diagram of a transaction

Description automatically generated

Figure 16: Class Diagram for Package entity.payment

…

### Class Design

<Detail design for each class>

#### Class “PayOrderController”

A screenshot of a computer screen

Description automatically generated

Figure 17: Class "PayOrderController"

**Table 2. Attribute design**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *#* | *Name* | *Data type* | *Default value* | *Description* |
| 1 | invoice | Invoice | null | Represents the invoice for the order. |
| 2 | invoiceScreen | InvoiceScreen | null | Screen interface for displaying invoice details |

**Table 3. Operation design**

|  |  |  |  |
| --- | --- | --- | --- |
| *#* | *Name* | *Return type* | *Description (purpose)* |
| 1 | askToPayOrder() | void | Prompts the user to pay the order, typically via the InvoiceScreen. |
| 2 | payOrder() | void | Initiates the payment process for the order. |
| 3 | saveTransactionResult() | void | Saves the result of the payment transaction to the system. |

*Parameter*:

* askToPayOrder(): Không có
* payOrder(): Không có
* saveTransactionResult(): Không có

*Exception*:

* PaymentException if there is an error during the payment process.
* InvoiceNotFoundException if the invoice cannot be found during the transaction.

**Method**

* askToPayOrder(): This method uses invoiceScreen to prompt the user to initiate payment for the invoice.
* payOrder(): Uses invoice to complete the payment process.
* saveTransactionResult(): Stores the outcome of the payment, potentially interacting with other components such as a database.

#### Class “PlaceOrderController”

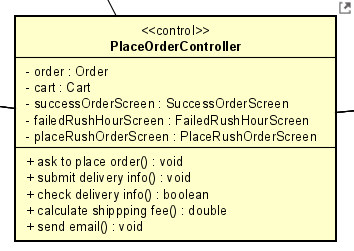


Figure 18: Class "PlaceOrderController"

**Table 4. Attribute design**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *#* | *Name* | *Data type* | *Default value* | *Description* |
| 1 | order | Order | N/A | Represents the order that needs to be processed. |
| 2 | cart | Cart | N/A | The shopping cart associated with the order. |
| 3 | successOrderScreen | SuccessOrderScreen | N/A | Screen displayed upon successful order placement. |
| 4 | failedRushHourScreen | FailedRushHourScreen | N/A | Screen displayed if order fails due to rush hour traffic. |
| 5 | placeRushOrderScreen | PlaceRushOrderScreen | N/A | Screen for placing rush orders. |

**Table 5. Operation design**

|  |  |  |  |
| --- | --- | --- | --- |
| *#* | *Name* | *Return type* | *Description (purpose)* |
| 1 | askToPlaceOrder() | void | Initiates the process of placing an order. |
| 2 | submitDeliveryInfo() | void | Submits delivery information for the order. |
| 3 | checkDeliveryInfo() | boolean | Checks if the delivery information is valid. |
| 4 | calculateShippingFee() | double | Calculates the shipping fee for the order based on delivery info. |
| 5 | sendEmail() | void | Sends a confirmation email after the order has been placed. |

*Parameter*:

* askToPlaceOrder(): Không có
* submitDeliveryInfo(): Không có
* checkDeliveryInfo(): Không có
* calculateShippingFee(): Không có
* sendEmail(): Không có

*Exception*:

* InvalidDeliveryInfoException if the delivery information is incorrect or incomplete.
* OrderFailureException if the order fails due to system or process issues.

**Method**

* askToPlaceOrder(): This method interacts with the user through various screens (successOrderScreen, failedRushHourScreen, etc.) to initiate the order process.
* submitDeliveryInfo(): Takes delivery information from the user and submits it for the order.
* checkDeliveryInfo(): Validates the delivery information. If invalid, throws InvalidDeliveryInfoException.
* calculateShippingFee(): Calculates the shipping fee based on the location or rush hour conditions.
* sendEmail(): Sends a confirmation email to the user with details of the order once successfully placed.

#### Class “PlaceRushOrderController”

A yellow box with black text

Description automatically generated

Figure 19: Class "PlaceRushOrderController"

**Table 6. Attribute design**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *#* | *Name* | *Data type* | *Default value* | *Description* |
| 1 | order | Order | N/A | Represents the order that needs to be processed. |

**Table 7. Operation design**

|  |  |  |  |
| --- | --- | --- | --- |
| *#* | *Name* | *Return type* | *Description (purpose)* |
| 1 | requestToPlaceRushOrder() | boolean | Requests to place a rush order, returning true if successful, false if not. |
| 2 | submitAdditionalInfo() | void | Receives and submits additional information necessary for the order. |

*Parameter*:

* requestToPlaceRushOrder(): None
* submitAdditionalInfo(): None

*Exception*:

* OrderFailureException: If the order fails due to system or process issues.

**Method**

* requestToPlaceRushOrder(): This method initiates the process of placing a rush order and checks the conditions for the order. It returns true if successful; otherwise, it returns false.
* submitAdditionalInfo(): This method allows the user to input additional information for the order and forwards that information for processing.

#### Class “Cart”

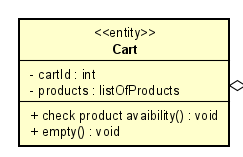


Figure 20: Class "Cart"

**Table 8. Attribute design**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *#* | *Name* | *Data type* | *Default value* | *Description* |
| 1 | cartId | int | N/A | Unique identifier for the shopping cart. |
| 2 | products | listOfProducts | N/A | List of products currently in the cart. |

**Table 9. Operation design**

|  |  |  |  |
| --- | --- | --- | --- |
| *#* | *Name* | *Return type* | *Description (purpose)* |
| 1 | checkProductAvailability() | void | Checks the availability of products in the cart. |
| 2 | empty() | void | Empties all products from the cart. |

*Parameter*:

* checkProductAvailability(): None
* empty(): None

*Exception*:

**Method**

* checkProductAvailability(): This method verifies whether the products in the cart are available for purchase, ensuring that stock is adequate.
* empty(): This method clears all products from the cart, resetting it for a new selection.

#### Class “Product”

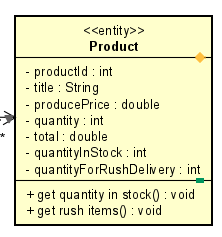


Figure 21: Class "Product"

**Table 10. Attribute design**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *#* | *Name* | *Data type* | *Default value* | *Description* |
| 1 | productId | int | N/A | Unique identifier for the product. |
| 2 | title | String | N/A | Name or title of the product. |
| 3 | producePrice | double | N/A | Price at which the product is sold. |
| 4 | quantity | int | N/A | Total quantity available for the product. |
| 5 | total | double | N/A | Total value of the products in stock based on price and quantity. |
| 6 | quantityInStock | int | N/A | Number of items currently in stock. |
| 7 | quantityForRushDelivery | int | N/A | Quantity allocated for rush delivery orders. |

**Table 11. Operation design**

|  |  |  |  |
| --- | --- | --- | --- |
| *#* | *Name* | *Return type* | *Description (purpose)* |
| 1 | getQuantityInStock() | void | Retrieves the current quantity of the product in stock. |
| 2 | getRushItems() | void | Retrieves the quantity of items available for rush delivery. |

*Parameter*:

* getQuantityInStock(): None
* getRushItems(): None

*Exception*:

* getQuantityInStock(): None
* getRushItems(): None

**Method**

* getQuantityInStock(): This method fetches and displays the current quantity of the product that is in stock, helping to manage inventory effectively.
* getRushItems(): This method provides the quantity of products that can be allocated for rush delivery, ensuring timely processing of urgent orders.

#### Class “DeliveryInfo”

A yellow box with black text

Description automatically generated

Figure 22: Class " DeliveryInfo "

**Table 12. Attribute design**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *#* | *Name* | *Data type* | *Default value* | *Description* |
| 1 | deliveryInfoId | int | N/A | Unique identifier for the delivery information. |
| 2 | name | String | N/A | Name of the recipient for the delivery. |
| 3 | address | String | N/A | Delivery address for the order. |
| 4 | deliveryInstruction | int | N/A | Instructions for delivery (e.g., specific handling requests). |
| 5 | phoneNumber | String | N/A | Contact number for the recipient. |

**Table 13. Operation design**

|  |  |  |  |
| --- | --- | --- | --- |
| *#* | *Name* | *Return type* | *Description (purpose)* |
| 1 | DeliveryInfo() | DeliveryInfo | Constructor for the DeliveryInfo class. |
| 2 | checkAddressSupportPlaceRushOrder() | void | Validates the delivery address for rush order placement. |

*Parameter*:

* DeliveryInfo(): None
* checkAddressSupportPlaceRushOrder(): None

*Exception*:

* DeliveryInfo(): None
* checkAddressSupportPlaceRushOrder(): None

**Method**

* DeliveryInfo(): This constructor initializes a new instance of the DeliveryInfo class with specified delivery details.
* checkAddressSupportPlaceRushOrder(): This method checks if the provided delivery address can support the placement of a rush order, ensuring that the address is valid and feasible for urgent delivery

#### Class “Order”

A yellow box with black text

Description automatically generated

Figure 23: Class " Order"

**Table 14. Attribute design**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *#* | *Name* | *Data type* | *Default value* | *Description* |
| 1 | orderId | int | null | Represents the unique identifier for the order. |
| 2 | deliveryInfo | DeliveryInfo | null | Contains the delivery details for the order. |
| 3 | cart | Cart | null | Represents the shopping cart associated with the order. |
| 4 | rushOrder | boolean | false | Indicates whether the order is expedited. |
| 5 | shippingFee | double | 0.0 | Represents the shipping fee associated with the order. |

**Table 15. Operation design**

|  |  |  |  |
| --- | --- | --- | --- |
| *#* | *Name* | *Return type* | *Description (purpose)* |
| 1 | Order(cart: Cart, deliveryInfo: DeliveryInfo) | Order | Constructor that initializes an order with a cart and delivery info. |
| 2 | updateWithRushInfo() | void | Updates the order to indicate it's a rush order. |
| 3 | updateAdditionalInformation() | void | Updates the order with additional information. |
| 4 | save() | void | Persists the order details to the system, such as a database. |

*Parameter*:

* Order(cart: Cart, deliveryInfo: DeliveryInfo): Không có
* updateWithRushInfo(): Không có
* updateAdditionalInformation(): Không có
* save(): Không có

*Exception*:

* OrderException if there is an error with the order process.
* DeliveryInfoNotFoundException if the delivery information is missing.
* CartEmptyException if the cart is empty at the time of order creation.

**Method**

* Order(cart: Cart, deliveryInfo: DeliveryInfo): Initializes a new order with the provided cart and delivery information.
* updateWithRushInfo(): Sets the rushOrder attribute to true if the order is expedited.
* updateAdditionalInformation(): Allows adding more details to the order without altering the existing attributes.
* save(): Saves the current state of the order into the system, possibly interacting with a database or other storage mechanisms.

#### Class “Invoice”

A screenshot of a computer code

Description automatically generated

Figure 24: Class "Invoice"

**Table 16. Attribute design**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *#* | *Name* | *Data type* | *Default value* | *Description* |
| 1 | invoiceId | int | null | Represents the unique identifier for the invoice. |
| 2 | order | Order | null | Contains the order associated with this invoice. |
| 3 | vat | double | 0.0 | Represents the Value Added Tax applied to the invoice. |
| 4 | transactionInfo | TransactionInfo | null | Contains details about the transaction related to the invoice. |
| 5 | subtotalBeforeVat | double | 0.0 | Subtotal amount before VAT is applied. |
| 6 | subtotal | double | 0.0 | Represents the subtotal amount after adjustments but before total. |
| 7 | total | double | 0.0 | Total amount due for the invoice including all applicable fees. |
| 8 | currency | String | "USD" | Represents the currency in which the invoice is issued. |

**Table 17. Operation design**

|  |  |  |  |
| --- | --- | --- | --- |
| *#* | *Name* | *Return type* | *Description (purpose)* |
| 1 | Invoice() | Invoice | Constructor that initializes an invoice with default values. |
| 2 | save() | void | Persists the invoice details to the system. |

*Parameter*:

* Invoice(): Không có
* save(): Không có

*Exception*:

* InvoiceException if there is an error with the invoice processing.
* OrderNotFoundException if the associated order cannot be found.

**Method**

* Invoice(): Initializes a new invoice, setting default values for its attributes.
* save(): Saves the current invoice details to the system, potentially interacting with a database or other storage mechanisms.

#### Class “TransactionInfo”

A screenshot of a computer

Description automatically generated

Figure 25: Class "TransactionInfo"

**Table 18. Attribute design**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *#* | *Name* | *Data type* | *Default value* | *Description* |
| 1 | transactionId | int | null | Represents the unique identifier for the transaction. |
| 2 | invoice | Invoice | null | Contains the invoice associated with this transaction. |
| 3 | time | String | null | Represents the time at which the transaction took place. |
| 4 | status | boolean | false | Indicates whether the transaction was successful or failed. |

**Table 19. Operation design**

|  |  |  |  |
| --- | --- | --- | --- |
| *#* | *Name* | *Return type* | *Description (purpose)* |
| 1 | TransactionInfo() | TransactionInfo | Constructor that initializes a transaction info with default values. |
| 2 | save() | void | Persists the transaction information to the system. |

*Parameter*:

* TransactionInfo(): Không có
* save(): Không có

*Exception*:

* TransactionInfoException if there is an error processing the transaction information.
* InvoiceNotFoundException if the associated invoice cannot be found..

**Method**

* TransactionInfo(): Initializes a new transaction info, setting default values for its attributes.
* save(): Saves the current transaction information to the system, potentially interacting with a database or other storage mechanisms.

#### Class “CartScreen”

A yellow box with black text

Description automatically generated

Figure 26: Class "CartScreen"

**Table 20. Attribute design**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *#* | *Name* | *Data type* | *Default value* | *Description* |
|  |  |  |  |  |

**Table 21. Operation design**

|  |  |  |  |
| --- | --- | --- | --- |
| *#* | *Name* | *Return type* | *Description (purpose)* |
| 1 | askToPlaceOrder() | void | Hiển thị thông báo để yêu cầu khách hàng đặt hàng. |
| 2 | notifyCustomer() | void | Thông báo cho khách hàng về trạng thái hiện tại của giỏ hàng. |
| 3 | askToPayOrder() | void | Hiển thị thông báo để yêu cầu khách hàng thanh toán đơn hàng. |

*Parameter*:

* askToPlaceOrder(): Không có
* notifyCustomer(): Không có
* askToPayOrder(): Không có

*Exception*:

**Method**

* askToPlaceOrder(): This method displays an interface to request the user to confirm the order placement.
* notifyCustomer(): This method notifies the customer about new information related to their shopping cart.
* askToPayOrder(): This method displays an interface requesting the customer to make a payment for their order.

#### Class “FailedRushOrderScreen”

A yellow rectangular sign with black text

Description automatically generated

Figure 27: Class "FailedRushOrderScreen"

**Table 22. Attribute design**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *#* | *Name* | *Data type* | *Default value* | *Description* |
|  |  |  |  |  |

**Table 23. Operation design**

|  |  |  |  |
| --- | --- | --- | --- |
| *#* | *Name* | *Return type* | *Description (purpose)* |
| 1 | FailedRushOrderScreen() | FailedRushOrderScreen | Constructor initializes a FailedRushOrderScreen instance. |

*Parameter*:

* FailedRushOrderScreen(): Không có

*Exception*:

**Method**

* FailedRushOrderScreen(): This initializes the screen that displays a notification when a rush order fails to be processed successfully.

#### Class “FailedPaymentScreen”

A yellow rectangular sign with black text

Description automatically generated

Figure 28: Class "FailedPaymentScreen"

**Table 24. Attribute design**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *#* | *Name* | *Data type* | *Default value* | *Description* |
|  |  |  |  |  |

**Table 25. Operation design**

|  |  |  |  |
| --- | --- | --- | --- |
| *#* | *Name* | *Return type* | *Description (purpose)* |
| 1 | FailedPaymentScreen() | FailedPaymentScreen | Constructor initializes a FailedPaymentScreen instance. |

*Parameter*:

* FailedRushOrderScreen(): Không có

*Exception*:

**Method**

* FailedPaymentScreen(): Initializes the screen that displays a notification when the payment is unsuccessful.

#### Class “AdditionalInfoBoundary”

A yellow rectangular sign with black text

Description automatically generated

Figure 29: Class "AdditionalInfoBoundary"

**Table 26. Attribute design**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *#* | *Name* | *Data type* | *Default value* | *Description* |
|  |  |  |  |  |

**Table 27. Operation design**

|  |  |  |  |
| --- | --- | --- | --- |
| *#* | *Name* | *Return type* | *Description (purpose)* |
| 1 | AdditionalInfoBoundary() | AdditionalInfoBoundary | Constructor initializes an AdditionalInfoBoundary instance. |
| 2 | submitAdditionalInfo() | int | Handles the submission of additional information. |

*Parameter*:

* AdditionalInfoBoundary(): Không có
* submitAdditionalInfo(): Không có

*Exception*:

**Method**

* AdditionalInfoBoundary(): Initializes the interface for collecting additional information.
* submitAdditionalInfo(): This method handles the submission of additional information and returns a code (or ID) related to the submitted information.

#### Class “DeliveryForm”

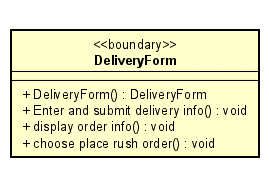
\ 

Figure 30: Class "DeliveryForm"

**Table 28. Attribute design**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *#* | *Name* | *Data type* | *Default value* | *Description* |
|  |  |  |  |  |

**Table 29. Operation design**

|  |  |  |  |
| --- | --- | --- | --- |
| *#* | *Name* | *Return type* | *Description (purpose)* |
| 1 | DeliveryForm() | DeliveryForm | Constructor initializes a DeliveryForm instance. |
| 2 | enterAndSubmitDeliveryInfo() | void | Handles the input and submission of delivery information. |
| 3 | displayOrderInfo() | void | Displays the details of the order. |
| 4 | choosePlaceRushOrder() | void | Allows the user to select options for a rush order. |

*Parameter*:

* DeliveryForm(): Không có
* enterAndSubmitDeliveryInfo(): Không có
* displayOrderInfo(): Không có
* choosePlaceRushOrder(): Không có

*Exception*:

**Method**

* DeliveryForm(): Initializes the interface for entering delivery information.
* enterAndSubmitDeliveryInfo(): This method handles the input and submission of delivery information from the user.
* displayOrderInfo(): Displays detailed information about the current order.
* choosePlaceRushOrder(): Allows the user to select options to place a rush order.

#### Class “PlaceRushOrderScreen”

\ A close-up of a code

Description automatically generated

Figure 31: Class "PlaceRushOrderScreen"

**Table 30. Attribute design**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *#* | *Name* | *Data type* | *Default value* | *Description* |
|  |  |  |  |  |

**Table 31. Operation design**

|  |  |  |  |
| --- | --- | --- | --- |
| *#* | *Name* | *Return type* | *Description (purpose)* |
| 1 | PlaceRushOrderScreen() | void | Constructor initializes a PlaceRushOrderScreen instance. |
| 2 | displayRushOrderInfo() | void | Displays the details of the rush order. |
| 3 | confirmToPlaceRushOrder() | void | Confirms the action to place a rush order. |

*Parameter*:

* PlaceRushOrderScreen(): No parameters.
* displayRushOrderInfo(): No parameters.
* confirmToPlaceRushOrder(): No parameters.

*Exception*:

**Method**

* PlaceRushOrderScreen(): Initializes the interface for placing a rush order.
* displayRushOrderInfo(): Displays detailed information about the rush order for the user.
* confirmToPlaceRushOrder(): Confirms the user's intent to place a rush order.

#### Class “InvoiceScreen”

\ A close-up of a screen

Description automatically generated

Figure 32: Class "InvoiceScreen"

**Table 32. Attribute design**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *#* | *Name* | *Data type* | *Default value* | *Description* |
|  |  |  |  |  |

**Table 33. Operation design**

|  |  |  |  |
| --- | --- | --- | --- |
| *#* | *Name* | *Return type* | *Description (purpose)* |
| 1 | InvoiceScreen() | InvoiceScreen | Constructor initializes an InvoiceScreen instance. |
| 2 | choosePaymentMethodAndAskToPay() | void | Prompts the user to choose a payment method and request payment. |

*Parameter*:

* InvoiceScreen(): No parameters.
* choosePaymentMethodAndAskToPay(): No parameters.

*Exception*:

**Method**

* InvoiceScreen(): Initializes the interface for displaying invoice details.
* choosePaymentMethodAndAskToPay(): Prompts the user to select a payment method and initiates the payment process.

#### Class “VNPayBoundary”

\ A yellow rectangular object with black text

Description automatically generated

Figure 33: Class "VNPayBoundary"

**Table 34. Attribute design**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *#* | *Name* | *Data type* | *Default value* | *Description* |
|  |  |  |  |  |

**Table 35. Operation design**

|  |  |  |  |
| --- | --- | --- | --- |
| *#* | *Name* | *Return type* | *Description (purpose)* |
| 1 | payOrder() | void | Initiates the payment process for an order. |
| 2 | redirect() | void | Redirects the user to the payment gateway or confirmation page. |

*Parameter*:

* payOrder(): No parameters.
* redirect(): No parameters.

*Exception*:

**Method**

* payOrder(): Handles the payment process for the order.
* redirect(): Manages the redirection logic after payment or for further actions.

# Design Considerations

***<Describe issues which need to be addressed or resolved before attempting to devise a complete design solution. Remember that, you have to refactor your source code to strictly follow the final design>***

## Goals and Guidelines

*<Describe any goals, guidelines, principles, or priorities which dominate or embody the design of the system and its software.*

*Examples of such goals might be: an emphasis on speed versus memory use; or working, looking, or “feeling” like an existing product.*

*Guidelines include coding guidelines and conventions.*

*For each such goal or guideline, describe the reason for its desirability unless it is implicitly obvious.*

*Describe any design policies and/or tactics that do not have sweeping architectural implications (meaning they would not significantly affect the overall organization of the system and its high-level structures), but which nonetheless affect the details of the interface and/or implementation of various aspects of the system (e.g., choice of which specific product to use)*>

## Architectural Strategies

*<Describe any design decisions and/or strategies that affect the overall organization of the system and its higher-level structures. These strategies should provide insight into the key abstractions and mechanisms used in the system architecture. Describe the reasoning employed for each decision and/or strategy (possibly referring to previously stated design goals and principles) and how any design goals or priorities were balanced or traded-off.*

*Examples of design decisions might concern (but are not limited to) things like the following:*

*• Use of a particular type of product (programming language, database, library, commercial off-the-shelf (COTS) product, etc.)*

*• Reuse of existing software components to implement various parts/features of the system*

*• Future plans for extending or enhancing the software*

*• User interface paradigms (or system input and output models)*

*• Hardware and/or software interface paradigms*

*• Error detection and recovery*

*• Memory management policies*

*• External databases and/or data storage management and persistence*

*• Distributed data or control over a network*

*• Generalized approaches to control*

*• Concurrency and synchronization*

*• Communication mechanisms*

*• Management of other resources*

>

## Coupling and Cohesion

*<Evaluate your design and describe which levels of coupling and cohesion that your design is at. Give proofs for your assumptions. Explain if there is any special design or exceptions>*

*<You may show the previous design from which you made improvements to get better levels of coupling and cohesion. You should clarify how and why you did these improvements>*

## Design Principles

*<Does your design follow the SOLID principles if there are new requirements/changing requirements in the future? Give proofs for your assumptions. Explain if there is any special design or exceptions>*

*<You may show the previous design from which you made improvements to get a better design, which follows SOLID principles in spite of additional requirements. You should clarify how and why you did these improvements>*

## Design Patterns

*<Do you use any design patterns for your design? If yes, describe detailly why you use those design patterns? Describe in detail on the solutions and how to implement each design pattern>*