Table of Contents

1. The use case model – Takeaway Restaurant

1.1. Description of actors

The subject of our project is the business domain of a takeaway restaurant and its online food ordering system.

The most important components of our system are the actors, use cases and their relationships.

Actors are external entities to the system. They can be divided into two main groups: primary and secondary actors.

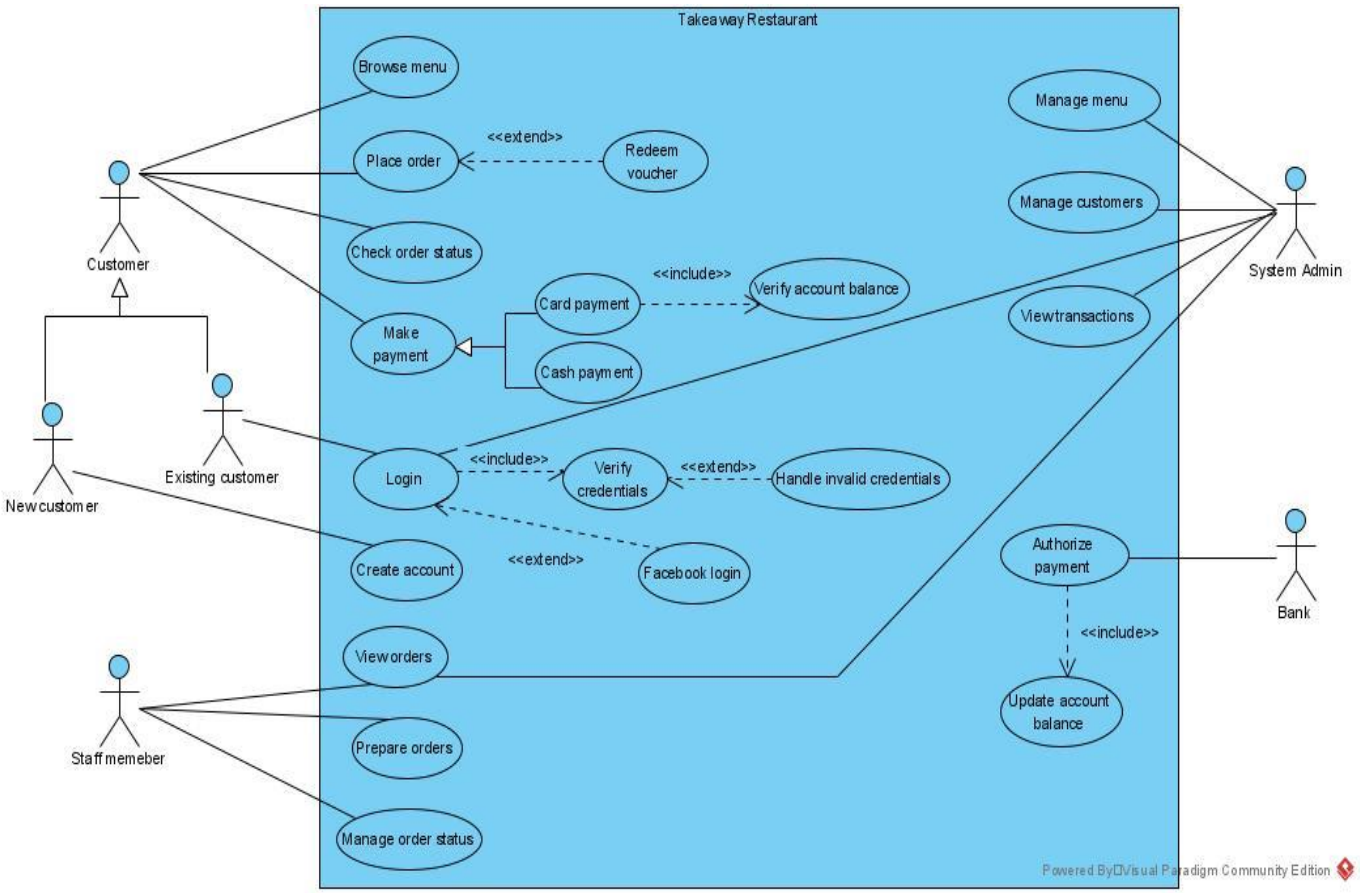
The **primary actors** of the system are users who execute main functionality. Our primary users are the ***customers*** who place food orders and the ***staff members*** of the restaurant who fulfil the customers' orders.
New customers are able to register themselves in the system and existing users can log in with their previously registered credentials (email address and password).

Customers login credentials will be verified each time they log in and in the case of invalid login data provided, an appropriate error message will appear.
All customers can **browse the menu**, **place orders**, **make payments** and **check the status** of their orders.
Staff members of the restaurant also act as primary users. They have the capability to **view the orders, fulfil them and manage the status of each order** by sending appropriate messages to the customers ("Your order is ready for collection in 20 minutes.").

Secondary actors are entities who execute secondary functions in the system. One of our secondary actors is the ***system administrator***, who has the capability to create, update and delete food items in the restaurants' menu, can manage the customer database (create, read, update and delete data related to the customers) and can view transactions related to customer orders.

With regards to processing credit/debit card payments, we can identify the ***customer's bank*** as a secondary actor as well. The bank plays a crucial role in the ordering process when the system sends a payment authorization request. At that point, the bank checks the customer's bank account balance and either authorizes the payment or refuses it.

1.2. Use case diagram with actors, use cases and relationships



The system boundary is the software that handles all customer orders.

An approximate ranking of use cases:

Rank	Use Case	Reasoning
High	Browse menu	First step of a successful sale is that customer can have a look at the products on sale
	Place order	This is the core functionality of the entire system
	Make payment	Crucial step for a successfully finished sale
	Create account	Very important step, as only registered customer are allowed to place an order
Medium	Check order status	Important part of customer satisfaction
	Prepare orders, Manage order status	
Low	Manage menu, Manage Customers	Less effect on architecture

2. Detailed use case description of a selected use case

1) **Use case name: Place Order**
2) **Scope:** A customer wishes to place a food order using the restaurant's online ordering system
3) **Actor:** customer
4) **Description:** This use case allows the customer to place a food order
5) **Flow description:**
    a) **Preconditions:** The system is idle, all related functions are available and customer logged in successfully
    b) **Activation:** This use case starts when the customer picks an item from the restaurant's menu
    c) **Main flow of events:**
        1. Customer can browse the restaurant's menu
        2. Customer selects a food item from the menu
        3. The system displays the price of the selected item
        4. The system prompts the customer to enter the required quantity of the selected item
        5. The system offers choice of payment type: cash or card payment
        6. Customer chooses cash payment
        7. System confirms that order was placed successfully and issues a receipt
        8. The system sends the order to the staff members of the restaurant for processing.
    d) **Alternate flow of events:**
        1. If at point 6 above the customer chose card payment, the system provides a form where customer enters payment card details.
        2. The system sends a request to the customer's bank to authorize the transaction.
        3. The transaction is successfully authorized by the bank
        4. Flow of events continues from point 7 above, the system confirms that order was placed successfully and issues a receipt

5. The system sends the order to the staff member of the restaurant for processing.

e) Another alternate flow of events:

1. If at step 4 in the main flow the customer provided invalid number for quantity (ie. 0), the system displays an error message and prompts customer to provide valid data for quantity.
2. When customer enters valid data, flow of events resumes back to the main flow.

f) Exceptional flow of events:

1. If the customer chose card payment at the 5th step of the main flow and the customer's bank did not authorize the transaction
2. The system displays a message to the customer outlining that the order cannot be processed this time.
3. The system displays the restaurant's home page.
4. Use case ends.

g) Termination: The system stores all data related to the fulfilled order.

h) Post-conditions: The system goes back to available state, ready to process another order.

3. Conceptual class diagram

The conceptual class diagram of the proposed online takeaway ordering system consists of 11 main classes and includes the relationships, attributes, basic operations and multiplicity between these classes.

Customer class keeps basic data about each registered customer and has a composition type relationship with the CustomerStatus class.

Customer and CustomerAccount classes also have a composition type relationship and a one to one type of multiplicity between them (one customer can only have one account and one account can only belong to one customer).

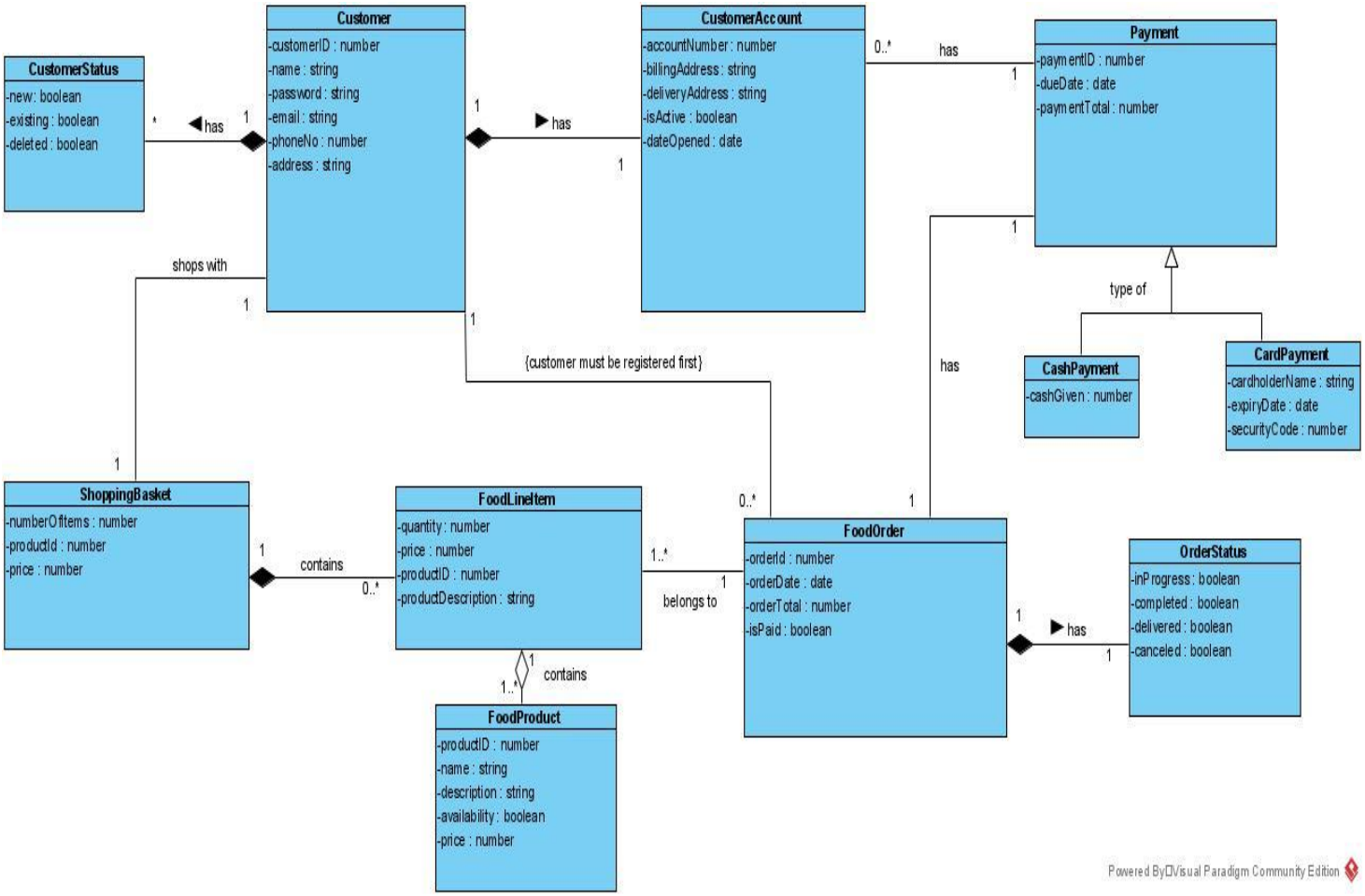
A customer shops with one ShoppingBasket, and a basket can only belong to one customer, marking their multiplicity as a one to one type. ShoppingBasket class has a direct relationship with FoodLineItem class, where a FoodLineItem cannot exist without a ShoppingBasket.

FoodProduct class an aggregation type of relationship with FoodLineItem class, as a food item can exist without being a part of a particular FoodLineItem.

FoodOrder class is one of the most important classes in the system, and its main functionality is to keep data about each order the customer places.

OrderStatus has a direct composition type relationship with FoodOrder class, as an order status cannot exist without an order.

Payment class has a direct connection to FoodOrder class and is a generalised class; two other classes inherit from it: CashPayment and CardPayment.


Customer
-customerID : number
-name : string
-password : string
-email : string
-phoneNo : number
-address : string
CustomerStatus
-new: boolean
-existing : boolean
-deleted : boolean
has
CustomerAccount
-accountNumber : number
-billingAddress : string
-deliveryAddress : string
-isActive : boolean
-dateOpened : date
Payment
-paymentID : number
-dueDate : date
-paymentTotal : number
shops with
{customer must be registered first}
type of
CashPayment
-cashGiven : number
CardPayment
-cardholderName : string
-expiryDate : date
-securityCode : number
ShoppingBasket
-numberOfItems : number
-productId : number
-price : number
contains
FoodLineItem
-quantity : number
-price : number
-productID : number
-productDescription : string
belongs to
FoodOrder
-orderId : number
-orderDate : date
-orderTotal : number
-isPaid : boolean
OrderStatus
-inProgress : boolean
-completed : boolean
-delivered : boolean
-canceled : boolean
FoodProduct
-productID : number
-name : string
-description : string
-availability : boolean
-price : number
Powered By Visual Paradigm Community Edition

4. Glossary of terms

Name	Aliases	Labels	Description
Browse menu		Use Case	Describes the process where customers can look up and check the restaurant's menu.
View orders		Use Case	Describes an option where staff members of the restaurant can look up the orders customers placed.
Manage menu		Use Case	Describes functionality where the system administrators can add/modify/delete food items on the restaurant's menu.
Manage customers		Use Case	Describes the options to add/modify/delete customer data within the database.
Place order		Use Case	Describes the main functionality of the system, where customers can order a food item from the restaurant's menu.
Redeem voucher		Use Case	Describes the option where the user can enter a special offer code to get a discount.
Check order status		Use Case	Describes the option where customers can look up the current status of their order.
Manage order status		Use Case	Describes an option where staff members can alter the status of an order: "received", "in progress", "completed".
FoodLineItem		Type	A type that represents data about an item sold.
Payment		Type	A class that holds data about the payment a customer makes when placing an order.
FoodOrder		Type	A type that represents a placed order and its basic details, including its ID, date placed, and the total value of that order.
Customer		Type	Describes users who register themselves as customers in the system.
CustomerAccount		Type	A type that holds relevant data about registered customers.
ShoppingBasket		Type	A type that represents all the food items chosen by the customer, alongside with their quantity and price.

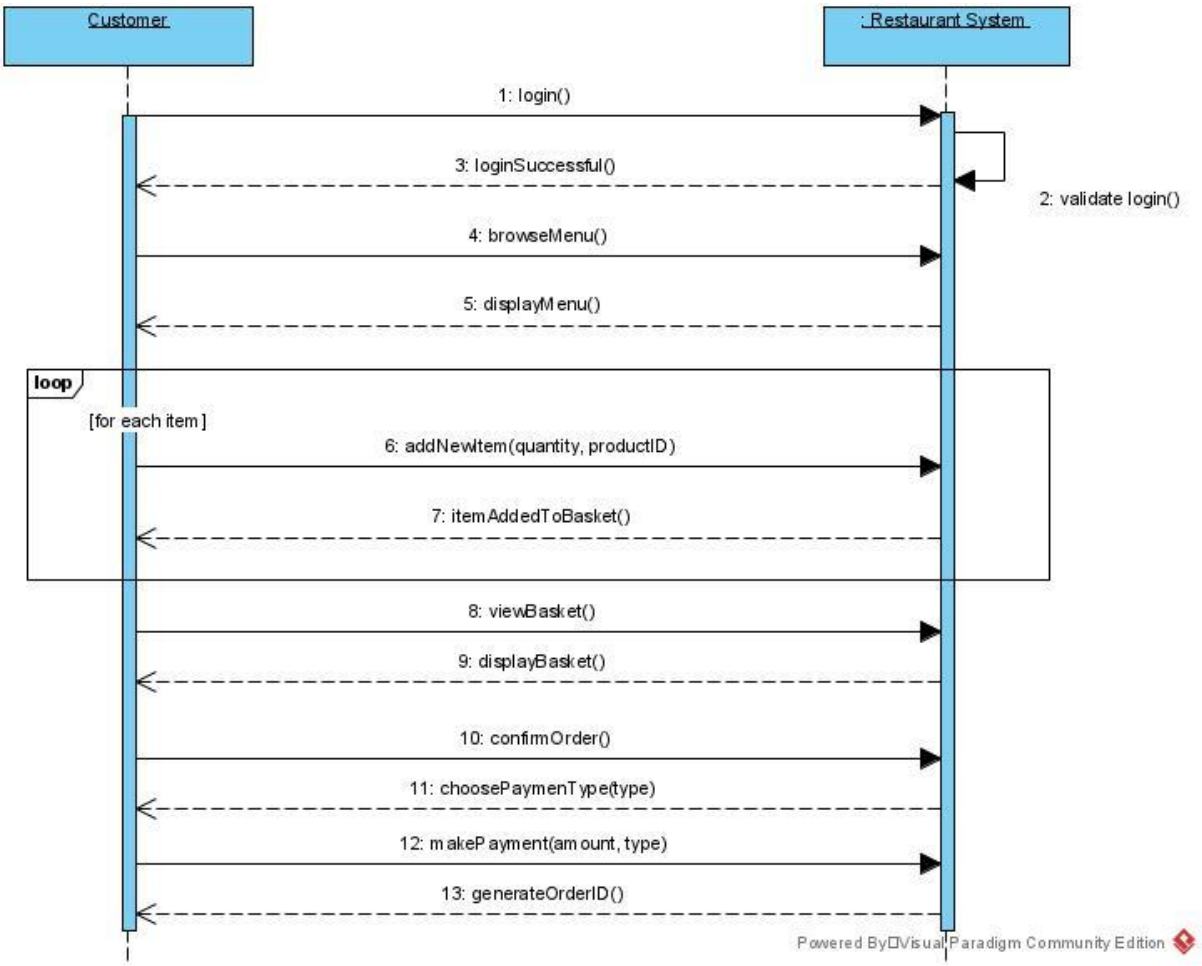
5. System sequence diagram

A system sequence diagram shows the progression of events over time for a particular scenario of a use case.

The most important system events and the corresponding system operations with regards to the use case are depicted on a system sequence diagram below.

Our highest priority use case is “Place Order” and our sequence diagram shows the steps of a successfully placed order.

The diagram shows synchronous messages from the customer and the appropriate responses from the system, including a self-message during the login process.
Adding new items to an order happens iteratively, ensuring that the system confirms the addition of each item separately.



6. Contracts for two system operations

Contracts help define system behaviour by describing the effects of system operations upon the system.

The most prominent system operations of “Place Order” use case are addNewItem and makePayment.

In both cases the customer generates a system event by taking a particular action and the restaurant’s order system reacts by executing the appropriate system operation.

Contract	
Name	addNewItem(quantity: integer, productID:integer)
Responsibility	Captures the addition of a new food item to the ShoppingBasket, displays the description, productID and the price of the chosen item
Use Case	Place Order
Type	System
Pre-condition	ProductID exist in the system
Post-conditions	If new order, an instance of ShoppingBasket class is created
	If new order, an instance of FoodOrder class is created
	An instance of FoodLineItem class is created
	An association is formed between ShoppingBasket and FoodLineItem
	FoodLineItem quantity is set to quantity
	FoodLineItem is associated with FoodProduct, based on matching productID

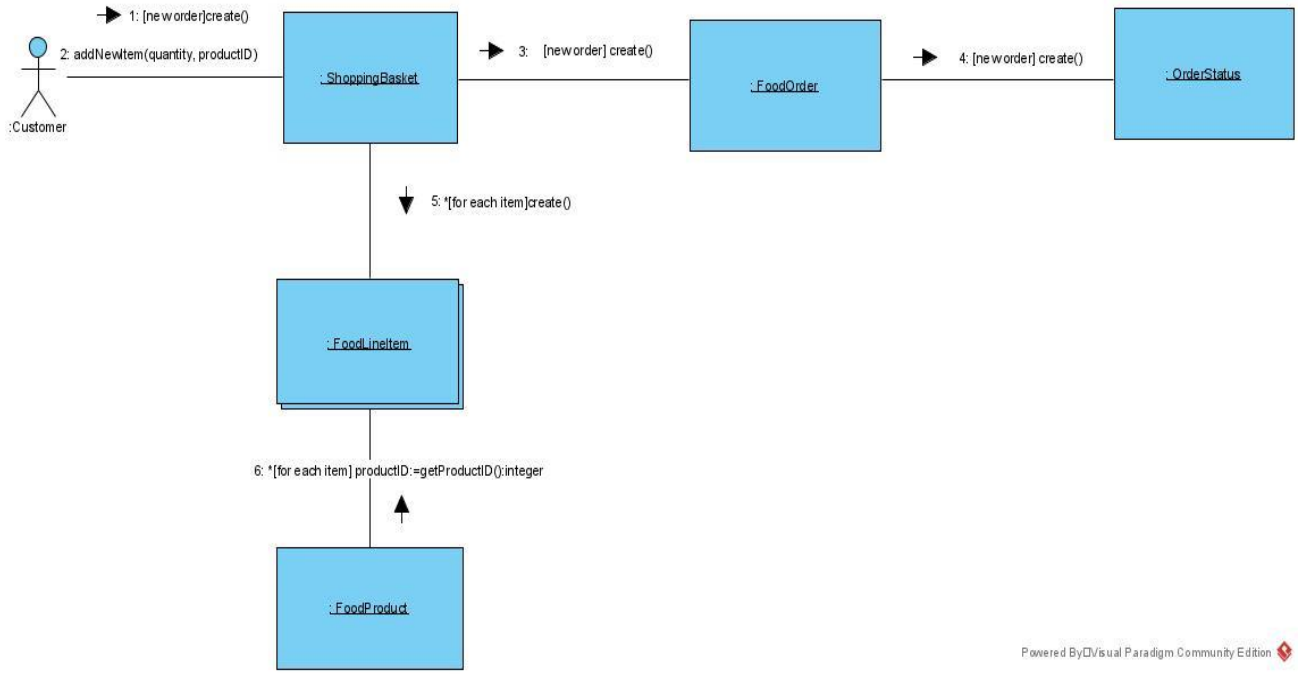
Contract	
Name	makePayment(amount: double, type: String)
Responsibility	Displays the total to be paid, creates a payment instance
Use Case	Place Order
Type	System
Pre-condition	FoodOrder was successfully created containing all required FoodLineItem instances
Post-conditions	An instance of Payment class is created
	An instance of CashPayment/CardPayment is created
	An association is formed with FoodOrder class and Payment class
	paymentTotal is set to orderTotal
	OrderStatus is set to inProgress

7. Communication diagrams with design patterns

Communication diagrams are a subtype of interaction diagrams and are used to describe the collaboration messages between object instances. Communication diagrams are built using design patterns and initiated with a starting message as shown below.

7.1. Communication diagram for addNewItem() operation using creator pattern

Object oriented software engineering uses a set of general principles to guide the creation process of software products.
These general principles are often referred to as design patterns.
The most suitable design pattern to use with the addNewItem() system operation of the proposed takeaway restaurant system is the *creator pattern.*
Creator pattern is used when one class aggregates or closely uses another class, or one class initializes data that will be passed to another class.
In our online ordering system, ShoppingBasket class closely uses FoodLineItem class, and FoodOrder class also closely uses OrderStatus class.
Furthermore, FoodProduct class passes vital data (productID) to FoodItem class.

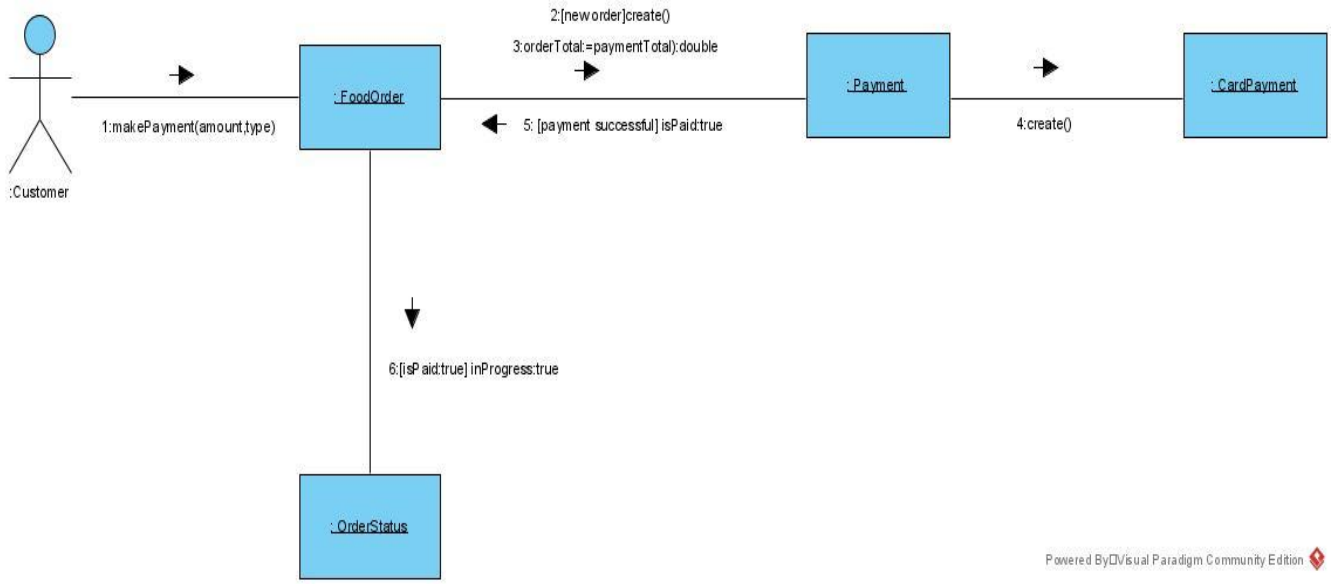


7.2. Communication diagram for makePayment() operation using the expert pattern

The most appropriate design pattern to describe the makePayment() operation is the *expert pattern.*

Expert pattern is used when the responsibility to fulfil a particular task is assigned to the class that has all the information needed.

FoodOrder class will have all the necessary information to pass to the Payment class, including the total amount payable.
Also, Payment class can pass down all the required data to the CardPayment class.



8. References

1. Dennis, A., Haley Wixom, B. and Tegarden, D., 2015. *Systems Analysis and Design with UML*. 5th ed. New York, US: John Wiley & Sons, Inc.,
2. Visual Paradigm. 2021. Visual Paradigm. [ONLINE] Available at: https://www.visual-paradigm.com/. [Accessed 30 March 2021].