CPSC 3720
Group 5

Design specification

Brendan Giberson
Andrew Gresham
Cole McKnight

Index

Design Goals

(In Descending priority)

1. In compliance with the nonfunctional requirement of usability, the interface should be user friendly, simple, and accessible. One of the main points of online ordering is providing a convenient way for the customer to order pizza. A poor interface could lead to a frustrated user instead of a happy customer.

2. In compliance with the nonfunctional requirement of reliability, the system should have high uptime and hold secure under heavy traffic, such as during big football games. The decision to order pizza is often for convenience, and if the user cannot place the order due to unreliability in the system, they'll likely just change their dining plans.

3. In compliance with the nonfunctional requirement of performance, the system should have quick runtime and provide immediate feedback. Long delays between entering info and receiving a response will frustrate the potential customer.

4. In compliance with the nonfunctional requirement of being legal, the system should comply with all legal requirements. The user's payment info is not saved, and any personal information such as their address will only be stored upon their request. Lower priority even though necessary because it mostly requires the absence of certain steps rather than the presence of them.

Hardware Mapping Strategy

Outside of Polaski's Pizza, potentially thousands of different computers will be used to order pizza. They will access the software, which will be stored in servers located on the manager's computer, through the existing Polaski Pizza website.

Different Subsystems of software will be installed on different hardware in Polaski's Pizza. The following table should help illustrate the strategy we plan on following.

Hardware	**Location**	**Control Object**	**Asso. Subsystems**
Cash Register	Counter	RegisterControl	ReceiveCustomerPaymentSubsystem
Receipt/Delivery Ticket Printer	Counter	PrinterControl	ProcessOrderSubsystem ReceiveCustomerPaymentSubsystem
Credit Card Machine	Counter	CardMachineControl	ReceiveCustomerPaymentSubsystem
Kitchen Management Computer	Kitchen	KitchenComputerControl	SetInventoryThresholdSubsystem EditItemQuantitySubsystem ProcessOrderSubsystem AssignOrderSubsystem
Display Screen	Kitchen	DisplayScreenControl	ProcessOrderSubsystem AssignOrderSubsystem
Manager's Computer	Back Office	ManagerComputerControl	ViewBusinessReportsSubsystem CreateEmployeeAccountSubsystem DeleteEmployeeAccountSubsystem
Delivery Driver's Phone	Delivery Driver	DeliveryPhoneControl	ProcessDeliverySubsytem ContactCustomerSubsystem

Persistent Data Management Strategy

Our data will be managed in three mediums: object-oriented databases, relational databases, and flat files. The object-oriented databases will store persistent data that is complex, or data that will need to be accessed by multiple nodes. User accounts and inventory items are good examples. Relational databases will be used to store large data sets, such as business reports. The flat files will store all of the temporary data that is not large in size. Examples include the current assignments list, the kitchen queue, or delivery queue. This strategy places data in the best medium to hold it, and therefore maximizes efficiency.

Boundary Condition Strategies

Start-up:
The system is started when the hardware it is installed on is powered on. For computers with multiple applications, the system is started when the user runs the program. For customers' computers, the system is started through the Polaski Pizza Website.

Shutdown:
The system is shut down when the user exits the window that it is running in.

Crashes:
The system will utilize the correct amount of resources necessary to handle the maximum expected traffic for a specified period of time. If the traffic becomes near the limits of the server(s), the program will not allow any additional traffic. More servers will be able to be added in case traffic greatly increases.

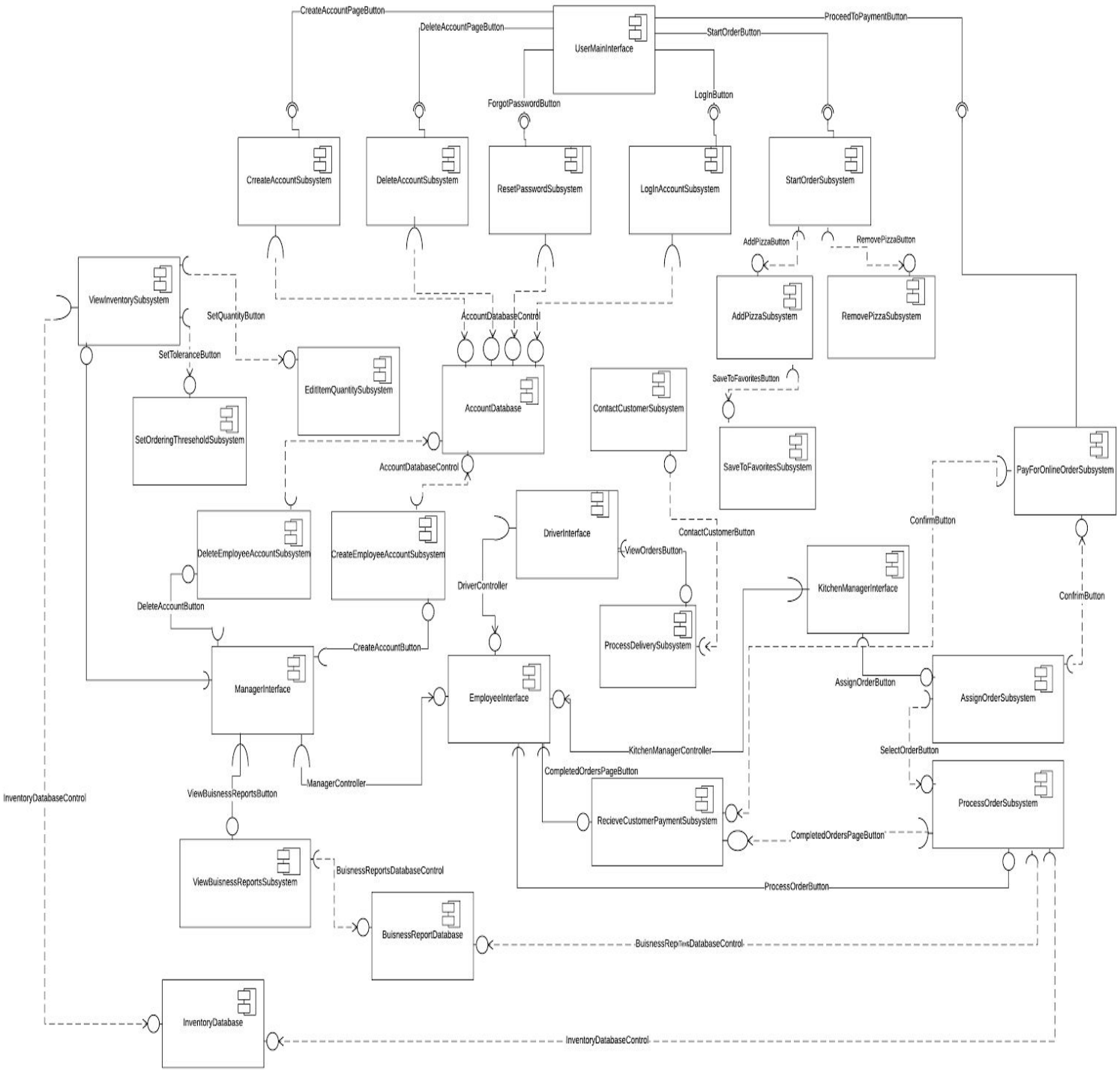
Network Outages:
The system will handle network outages by having all hardware that is in Polaski's Pizza hardwired to the necessary nodes, so that if the Wifi goes out, pizza can still be sold in-house. This will mean online orders cannot be made, but that is an inevitable consequence of a network outage.

Power Outages:
In the event of a power outage, still running a functioning POS system throughout the restaurant will be near impossible. We recommend backup generators to be installed to keep the restaurant powered up. After the system is rebooted, it should function normally.

Hardware Failure:
If hardware fails, little can be done to restore its functionality in the system. In the event of hardware failure, certain aspects of the "legacy" system should be implemented(i.e. writing down orders), until functionality can be restored. The system will allow for certain legacy measures to be implemented, while still keeping other parts of the system functional.

Software Fault:
The system will be thoroughly tested to minimize faults. In the event of a fault that's not already handled, the system will isolate the fault to the current subsystem to prevent connected parts of the system from crashing as well, and then will print an error message stating the location, date, and time of the error. This will allow any bugs to be fixed quickly.

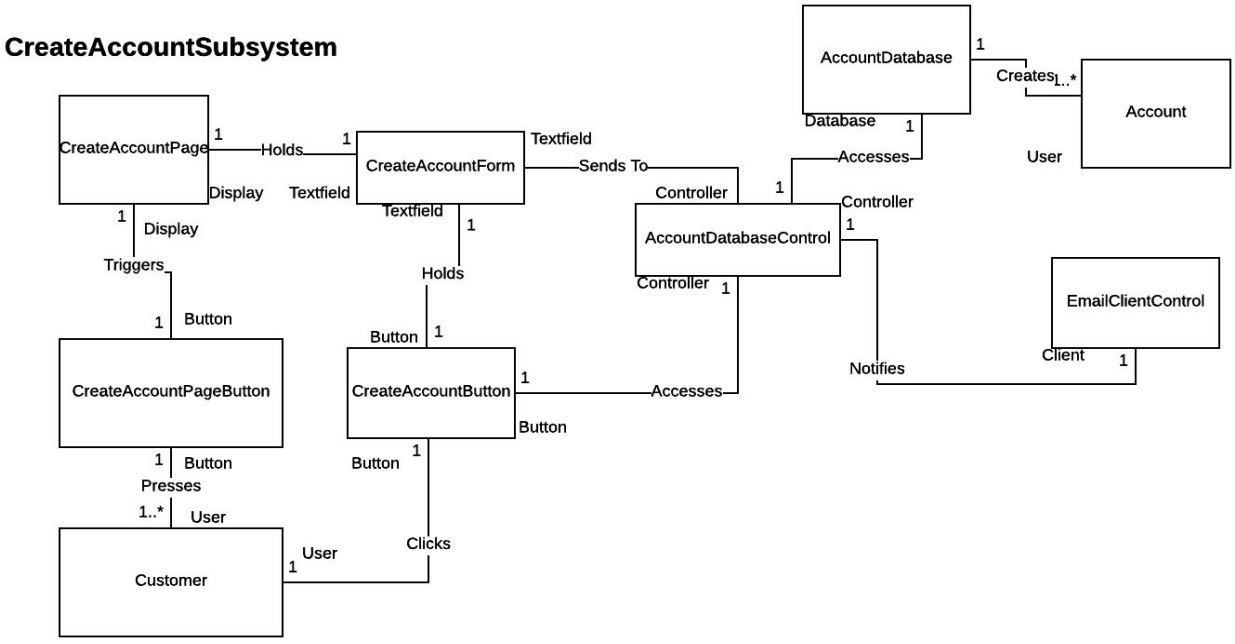
Subsystem Decomposition



Subsystem Descriptions and Association Diagrams

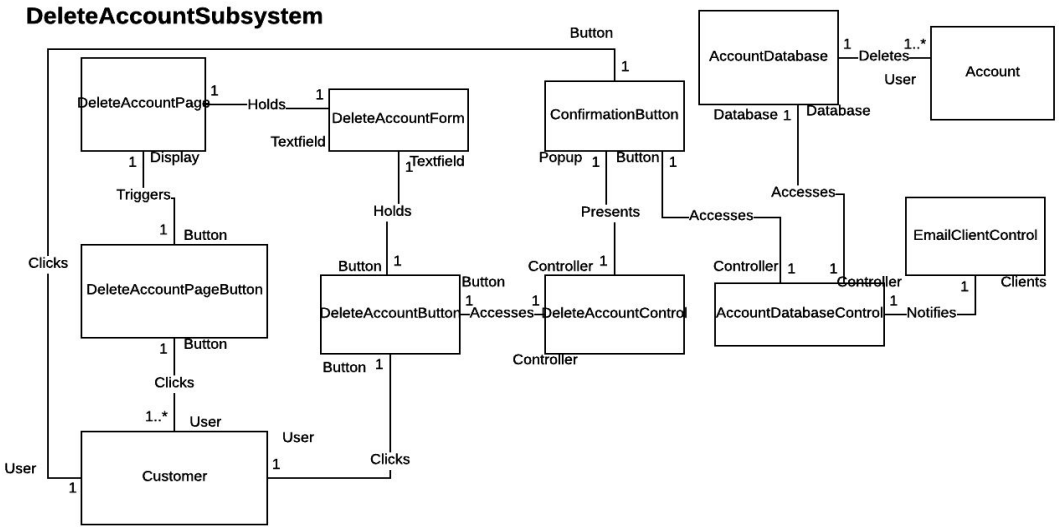
CreateAccountSubsystem

This subsystem exists as a way for a user to create an account for online ordering of Polaski's Pizza, making future ordering and checkout much simpler. This service requires access to the account database as well as the email client controller. It stores a new account with all the customers information to the database and emails the new customer to confirm.



DeleteAccountSubsystem

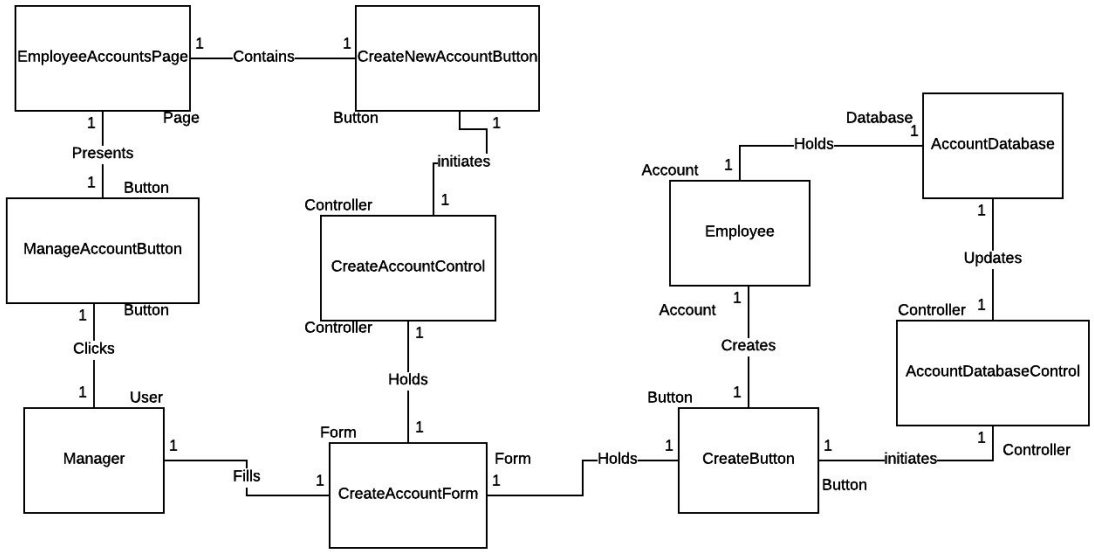
This subsystem exists as a way for a customer to delete their online account. The system requires access to the account database as well as the email client controller. It removes the customer account from the database and emails the customer confirming their change.



CreateEmployeeAccountSubsystem

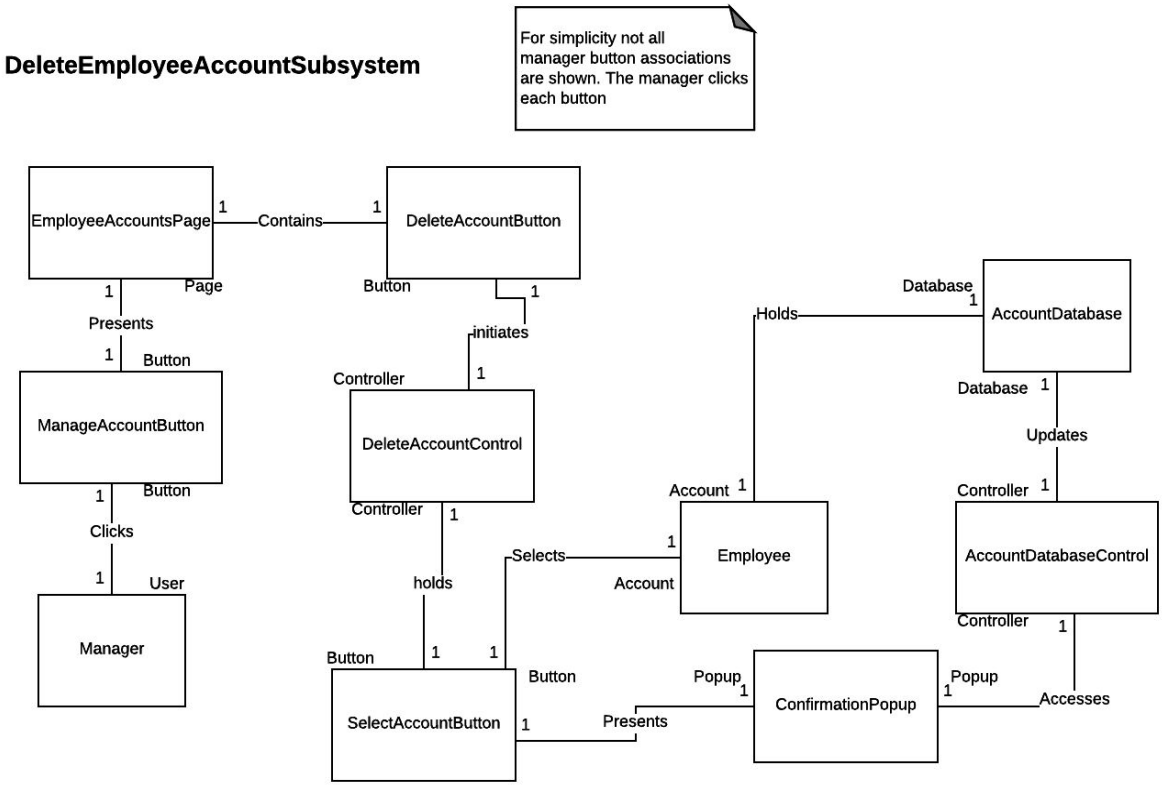
This subsystem allows the manager to create an account for a newly hired employee. The account is saved to the account database, upon filling out the employees information and clicking the create button. Employees can then log into their accounts to access a multiplicity of workplace related tasks.

CreateEmployeeAccountSubsystem



DeleteEmployeeAccountSubsystem

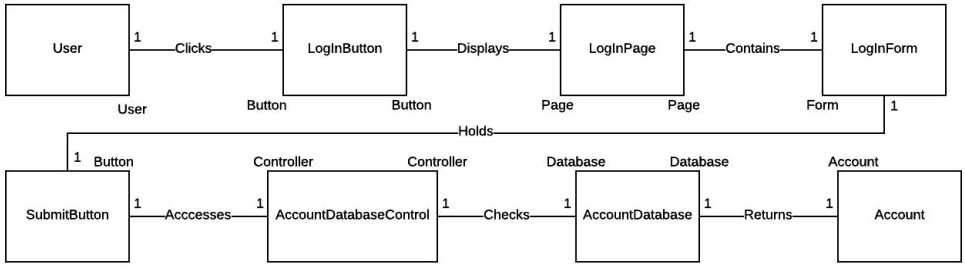
This subsystem allows the manager to delete an employee's account whom has been terminated/ is no longer working. The manager chooses the account to delete and upon confirming the delete the subsystem updates the account database.



LogInAccountSubsystem

This subsystem exists as a way for users to login to their accounts before ordering. The user's account contains favorite orders, their address, and any saved payment methods. The subsystem accesses the account database to ensure proper login credentials and return the user's account.

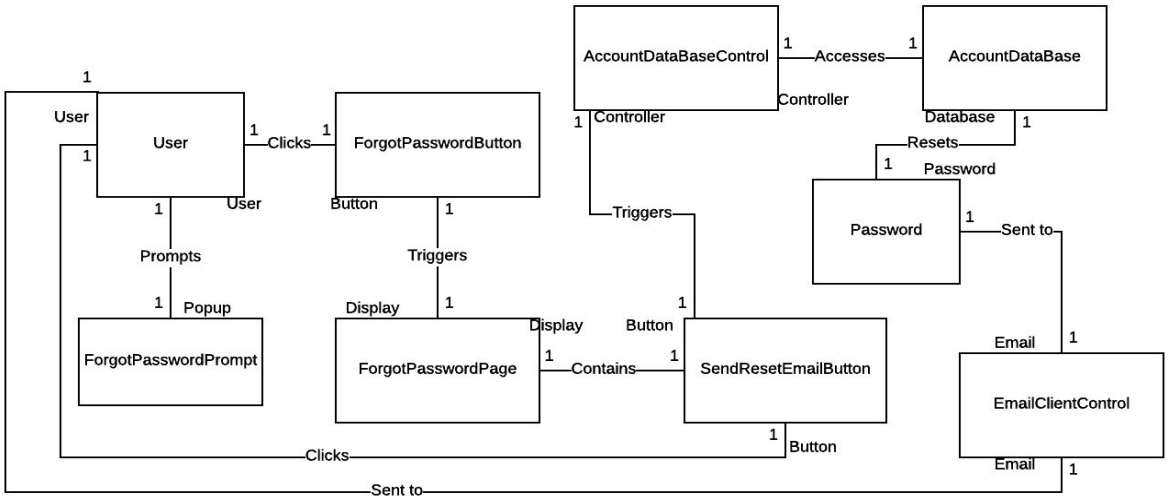
LogInAccountSubsystem



ResetPasswordSubsystem

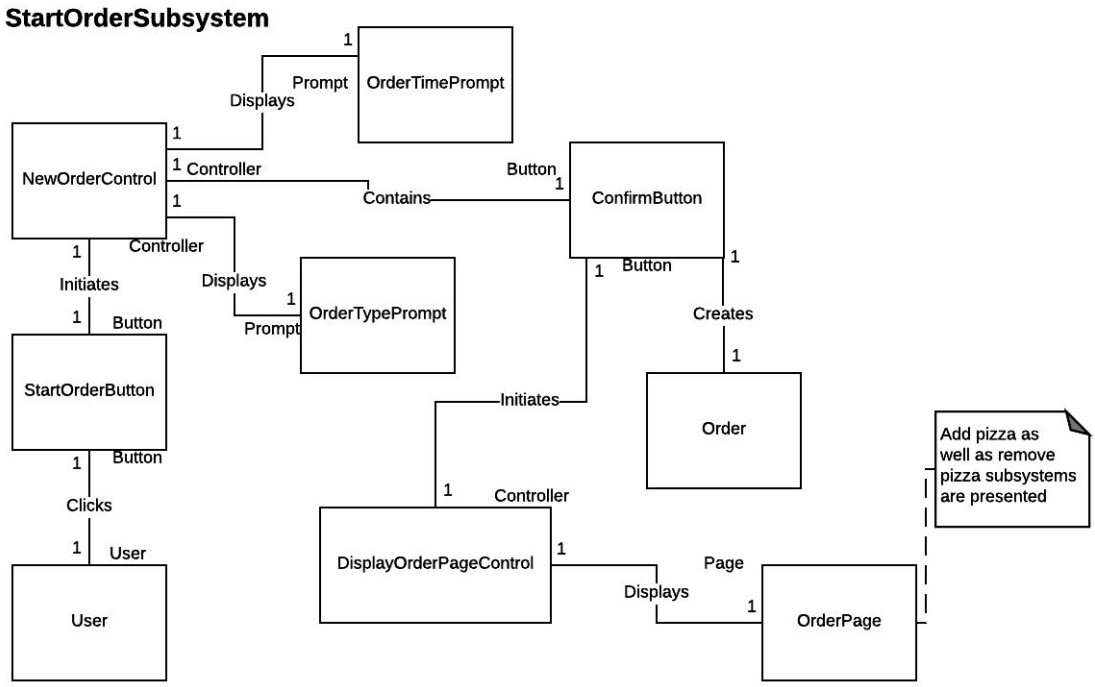
This subsystem provides the user with a way to recover their password in which they may have forgotten. The system resets the user's password. An email is sent to the customer's email account containing a new temporary password. This system requires access to the account database as well as the email client controller.

ResetPasswordSubsystem



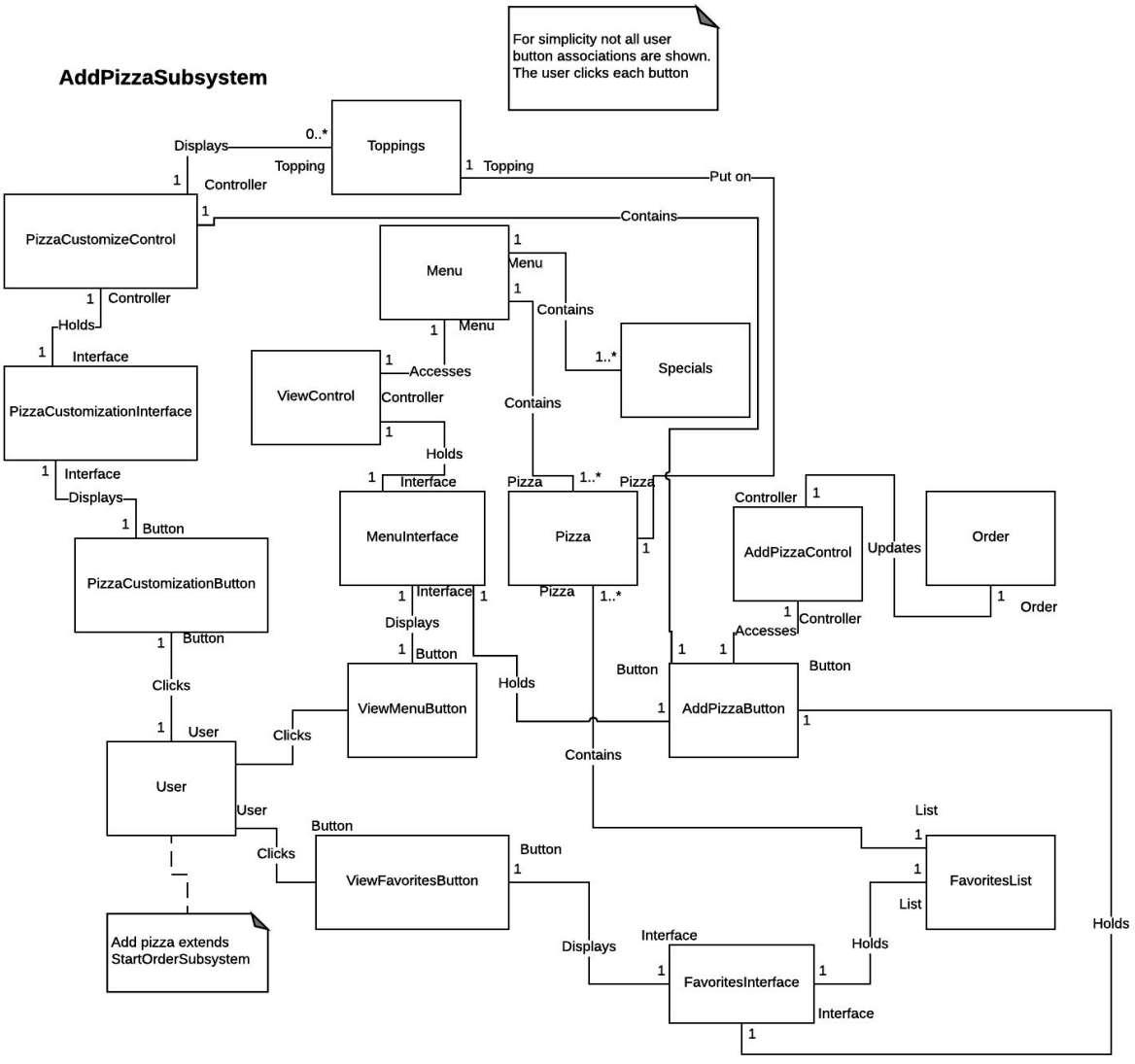
StartOrderSubsystem

This subsystem contains a way for an online user to start an order. The user is given the options for delivery, carryout, or dine in . They are also given the option to choose a time in which they would like their order prepared. This subsystem then presents the AddPizzaSubsystem allowing the user to add items to their order.



AddPizzaSubsystem

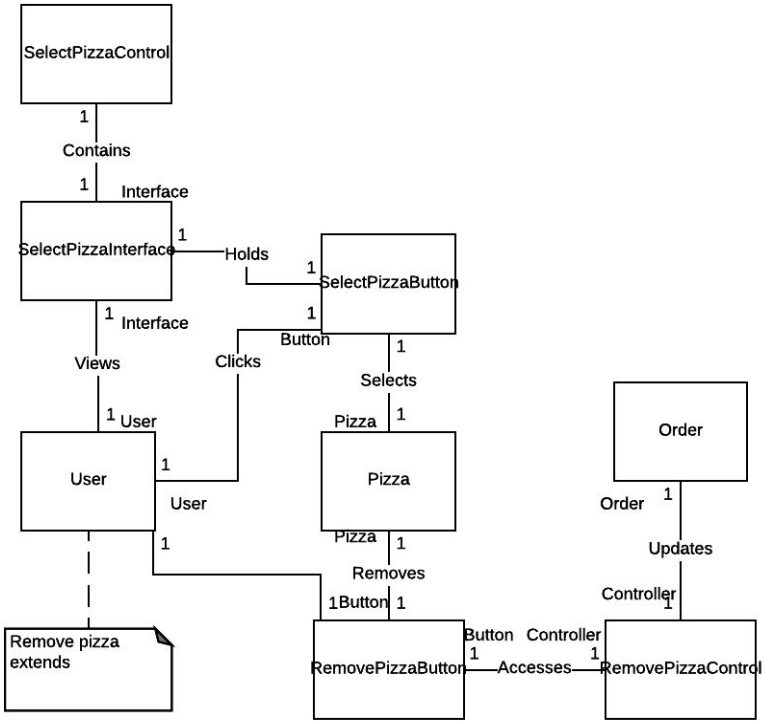
This subsystem is presented after the StartOrderSubsystem is completed. It allows for a user to view the menu containing the daily specials. The user can then select an item to add to their order. A user also has the ability to create their own pizza and choose which toppings they desire.



RemovePizzaSubsystem

This subsystem is contained within the StartOrderSubsystem. Upon the click of a button it allows the user to remove an item from their order updating the contents of their cart.

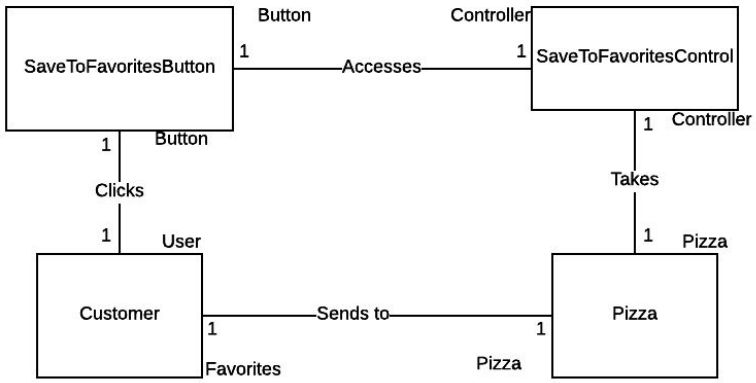
RemovePizzaSubsystem



SaveToFavoritesSubsystem

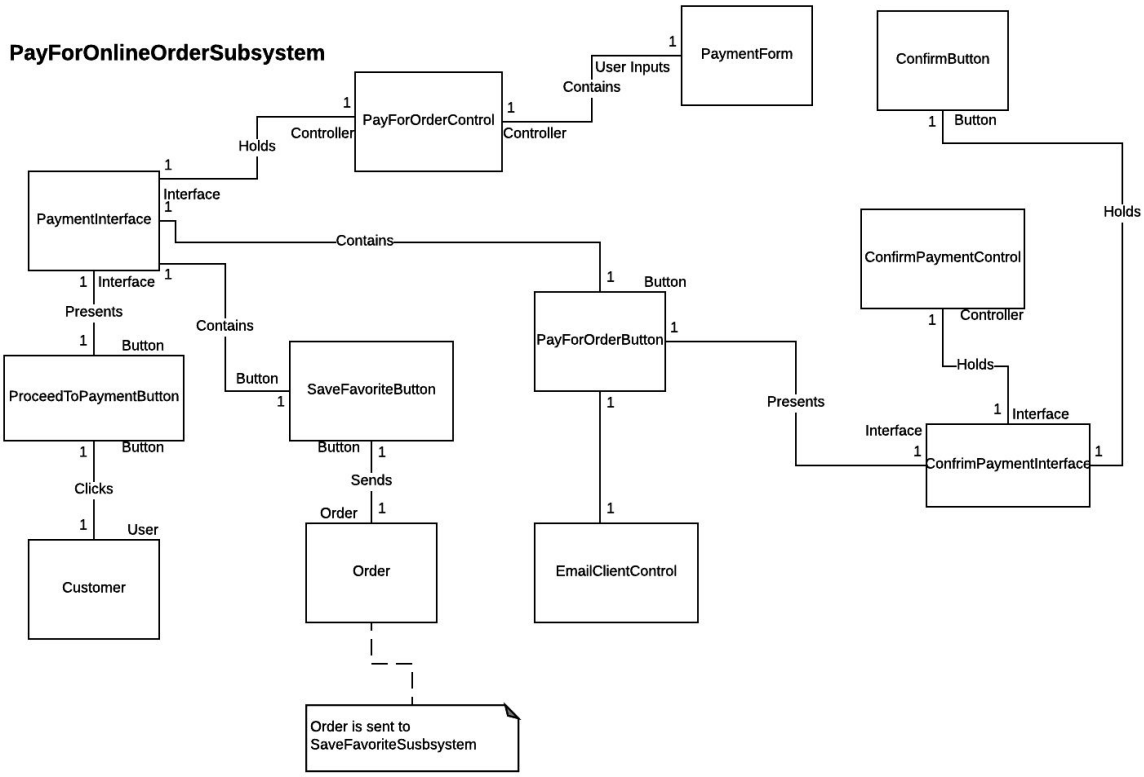
This subsystem allows users to save a pizza to their favorites list for future reordering. The subsystems list can be accessed from the AddPizzaSubsystem.

SaveToFavoritesSubsystem



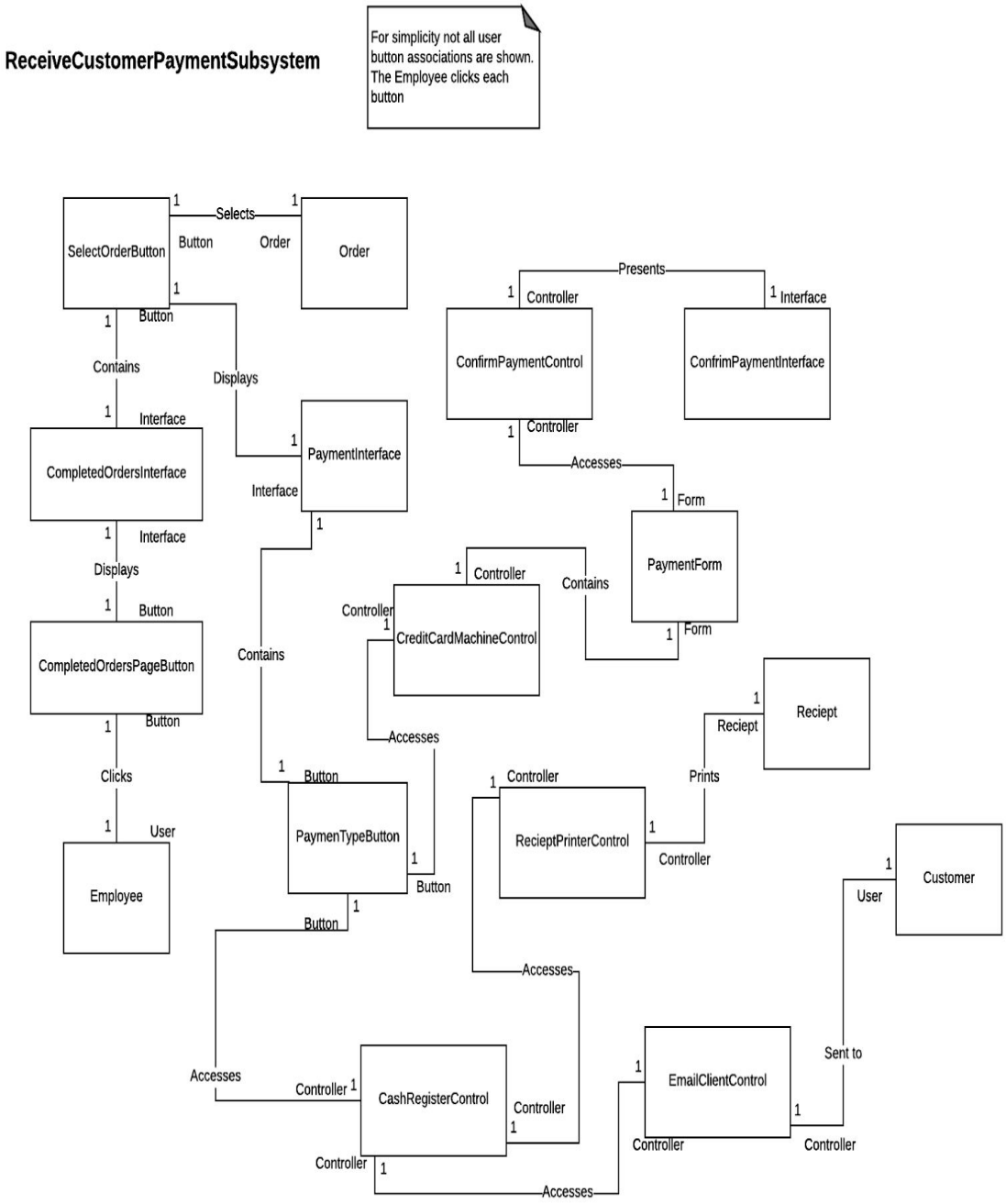
PayForOnlineOrderSystem

This subsystem acts as a way for the user to process and pay for the contents of their order. A user chooses to proceed to payment where they are prompted with the choice of paying with cash or a credit card. If paying with a card the user is asked to input their information . Successful payment transfers money to directly to Polaski's Pizza bank account. The system requires access to bank accounts using the controllers. It sends the order to Polaski's Pizza for processing.



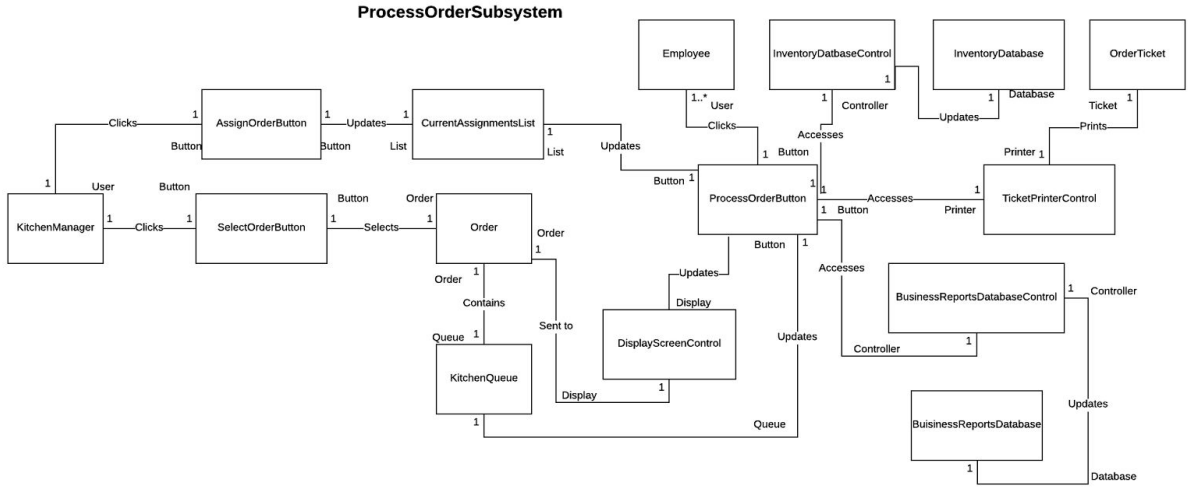
ReceiveCustomerPaymentSubsystem

This subsystem acts as a way for customers to pay for their orders in store. The employee accesses the completed orders to choose which order to pay for. Upon selecting cash or card the subsystem will either open the cash register or access the credit card machine for the user to operate. A receipt is printed and also emailed to the customer upon request.



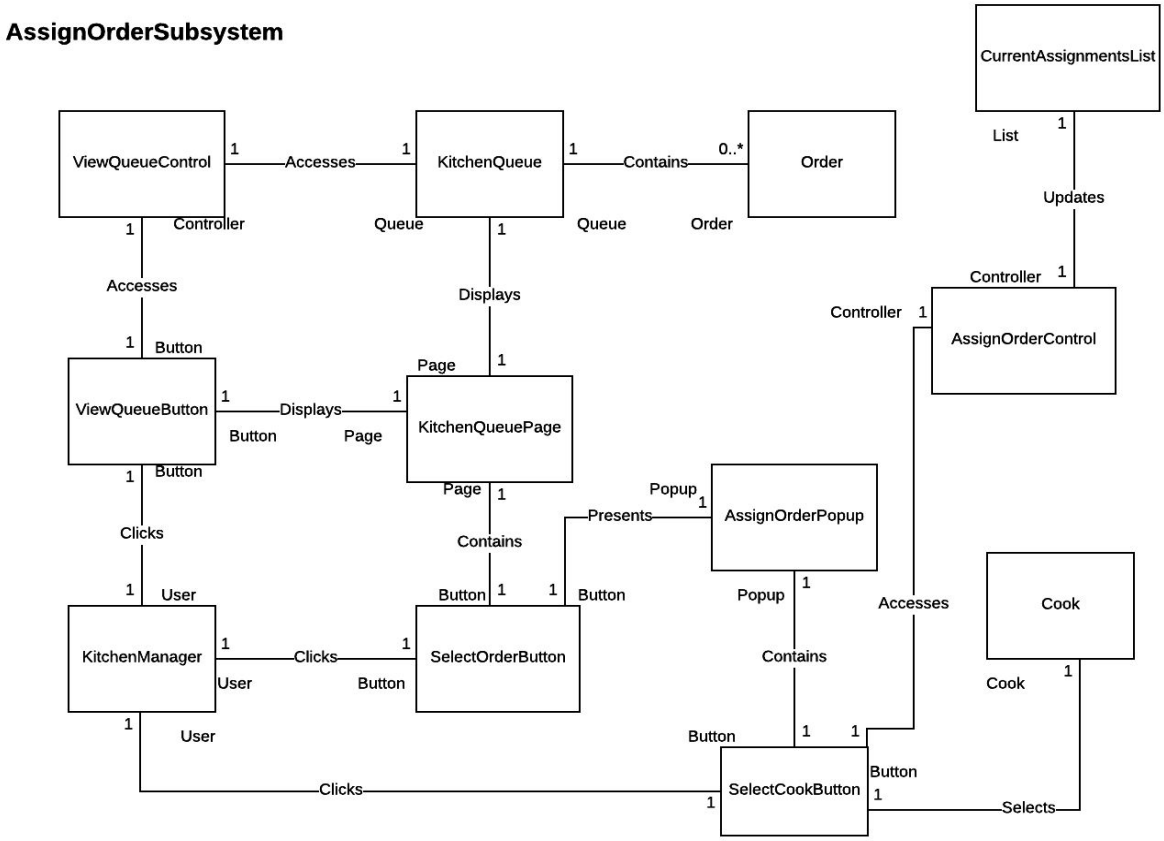
ProcessOrderSubsystem

This subsystem extends the assign order subsystem. The manager assigns an order to an employee, displaying it to the screen. When complete the employee chooses to process order updating: the kitchen queue, current assignments list, display screen, and the business and inventory reports. The ticket is then printed and placed on a pizza. This subsystem requires access to the printer control as well as the business reports database.



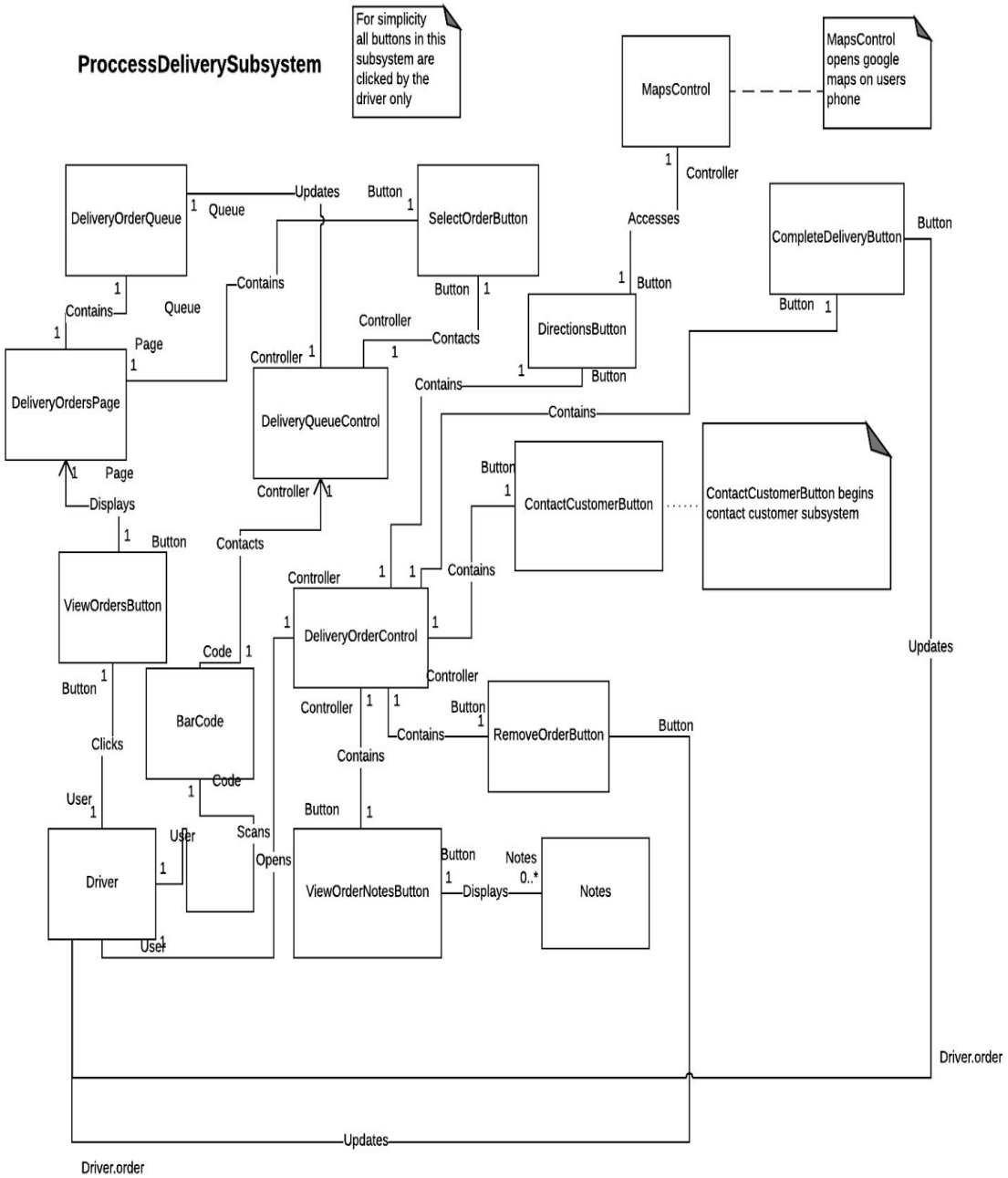
AssignOrderSubsystem

This subsystem allows the manager to assign an order to a cook displaying the order on the queue. The manager is able to see orders as well as a list of cooks , allowing easy assignment to the desired cook. This subsystem is the prerequisite to the ProcessOrderSubsystem.



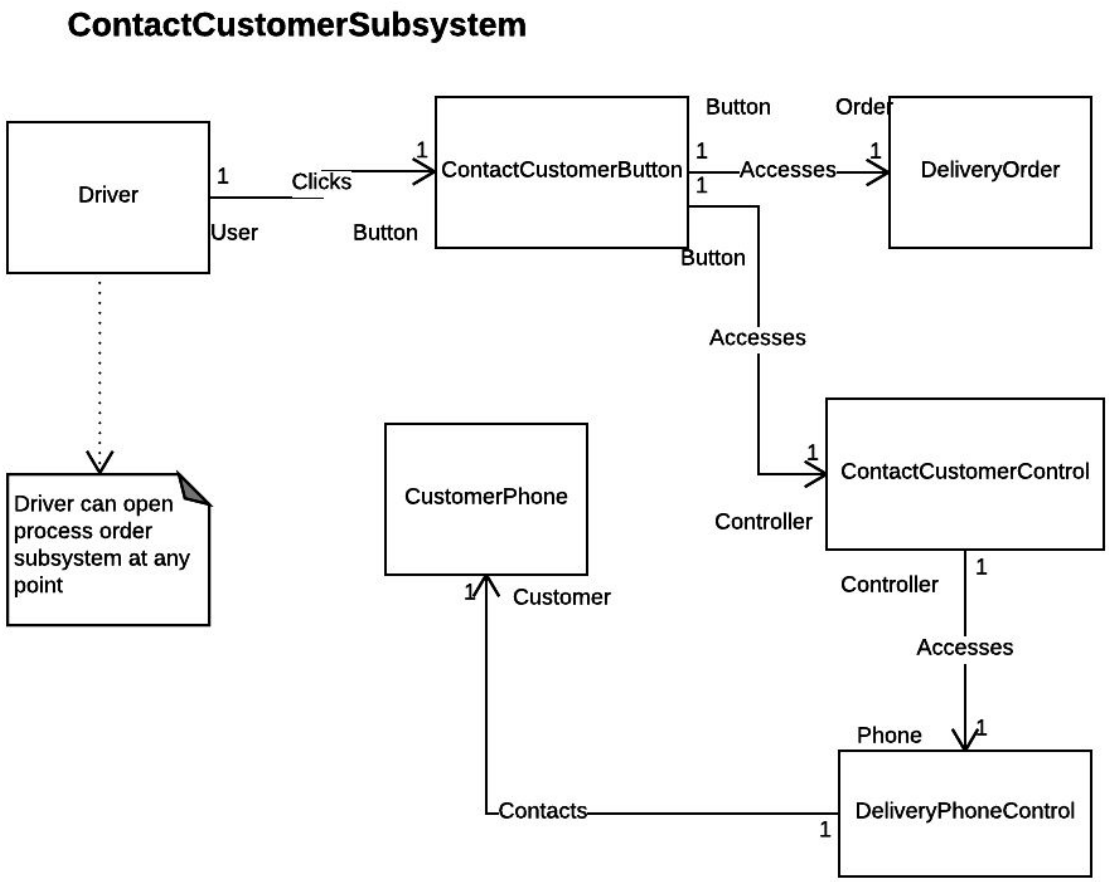
ProcessDeliverySubsystem

This subsystem occurs after an order is processed. It is a display for delivery drivers showing directions, delivery notes, customer contact, and the customer's order. The driver scans a receipt printed out by the ticket controller and then has the ability to scan it on their phone reading in what order they are taking care of. This subsystem requires outside access to google maps. It contains the contact customer subsystem.



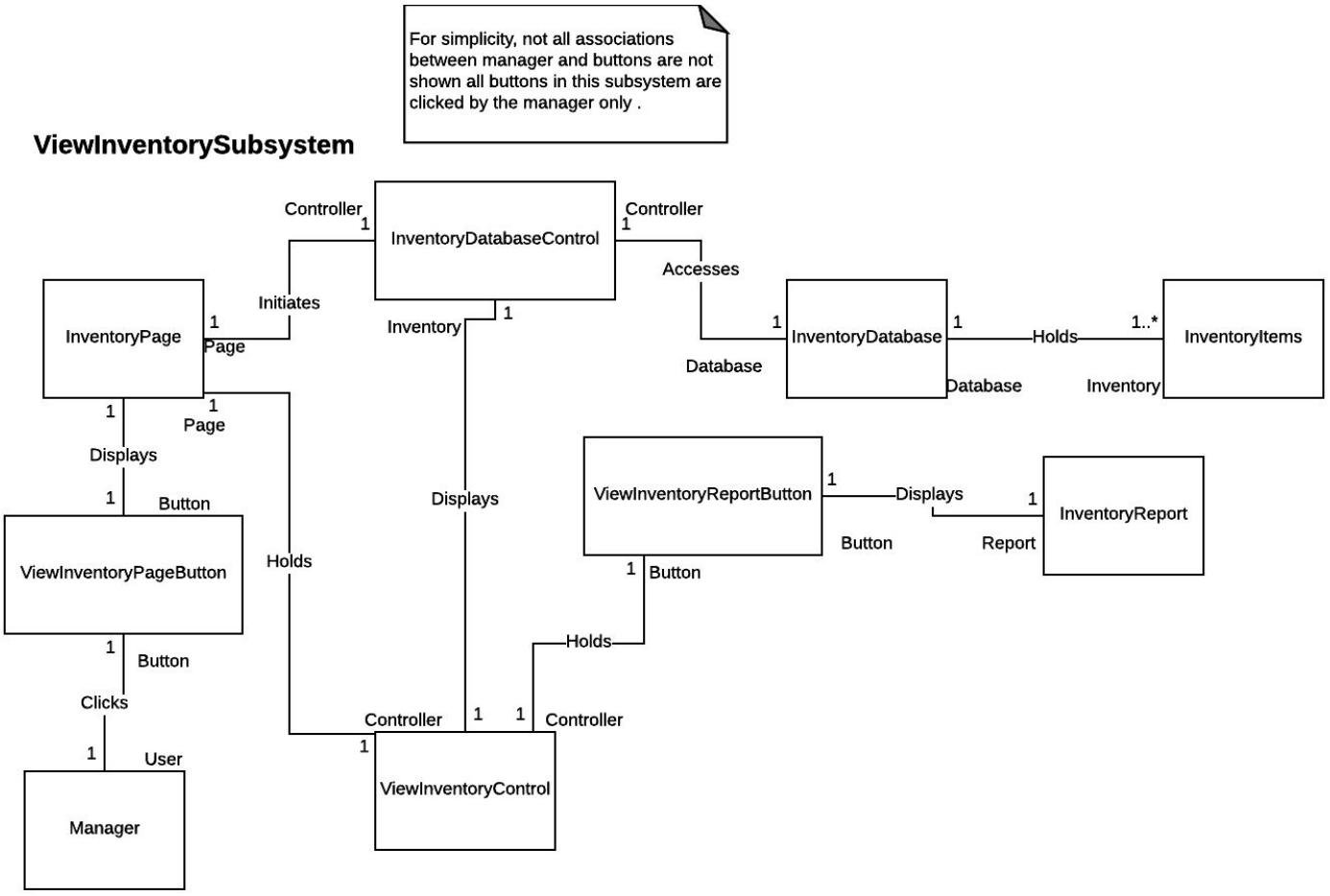
ContactCustomerSubsystem

This subsystem extends the ProcessDeliverySubsystem. It exists as a way for the delivery driver to get in touch with a customer if need be. The system uses the employee phone to contact the customer without displaying the customer's phone number.



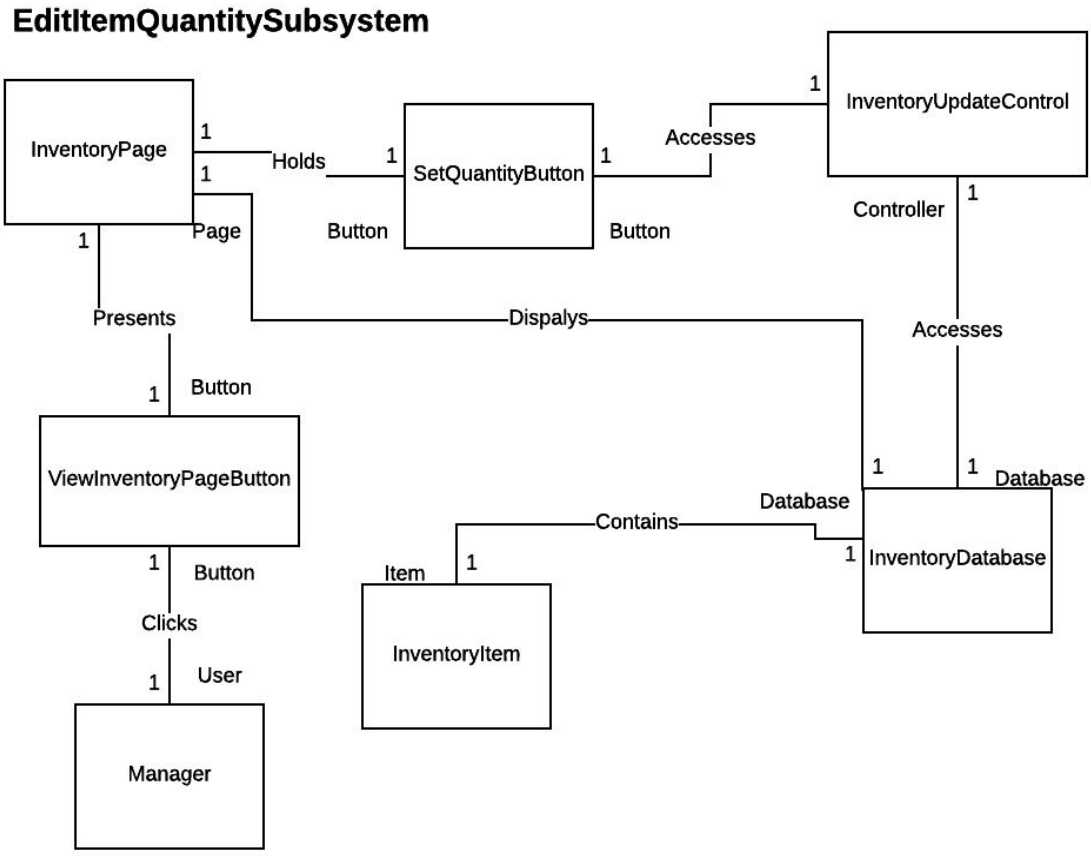
ViewInventorySubsystem

This subsystem exists as a way for the manager to view current inventory. Upon viewing inventory the manager can choose to create an inventory report which requires access to the inventory database , creating and displaying the inventory report.



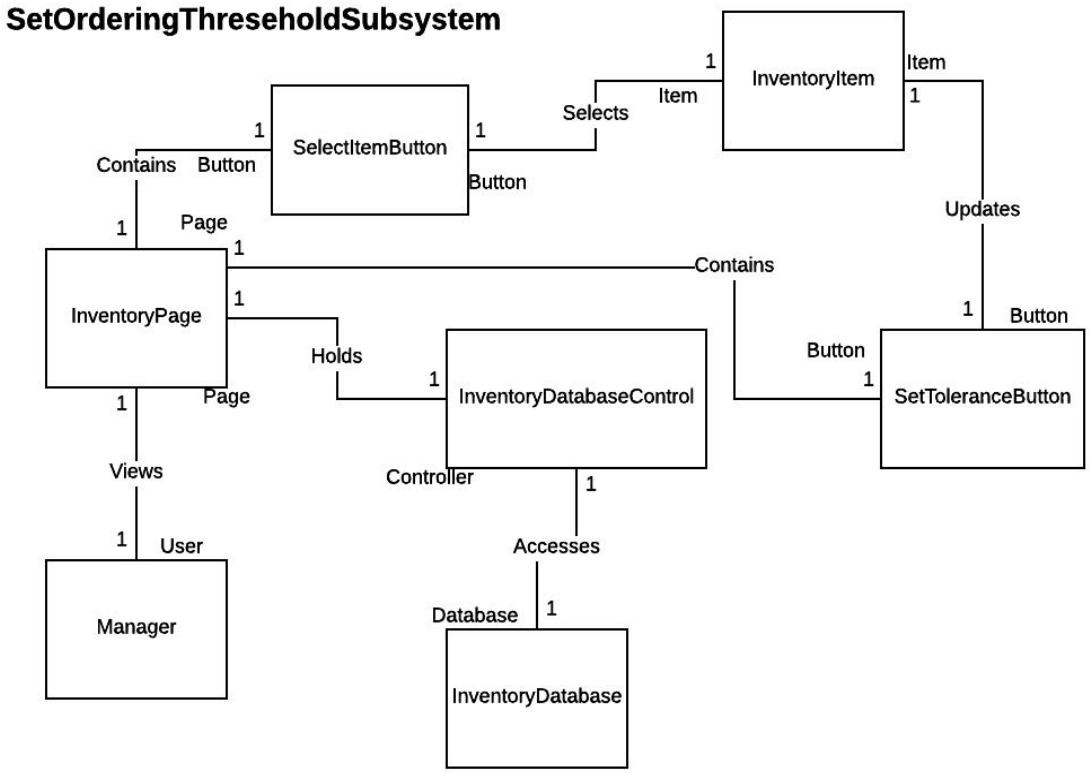
EditItemQuantitySubsystem

This subsystem extends ViewInventorySubsystem. It allows the manager to edit the quantity of any item in inventory. Upon an order arriving the manager can update the item's quantities. The subsystem then accesses the inventory database saving all changes made.



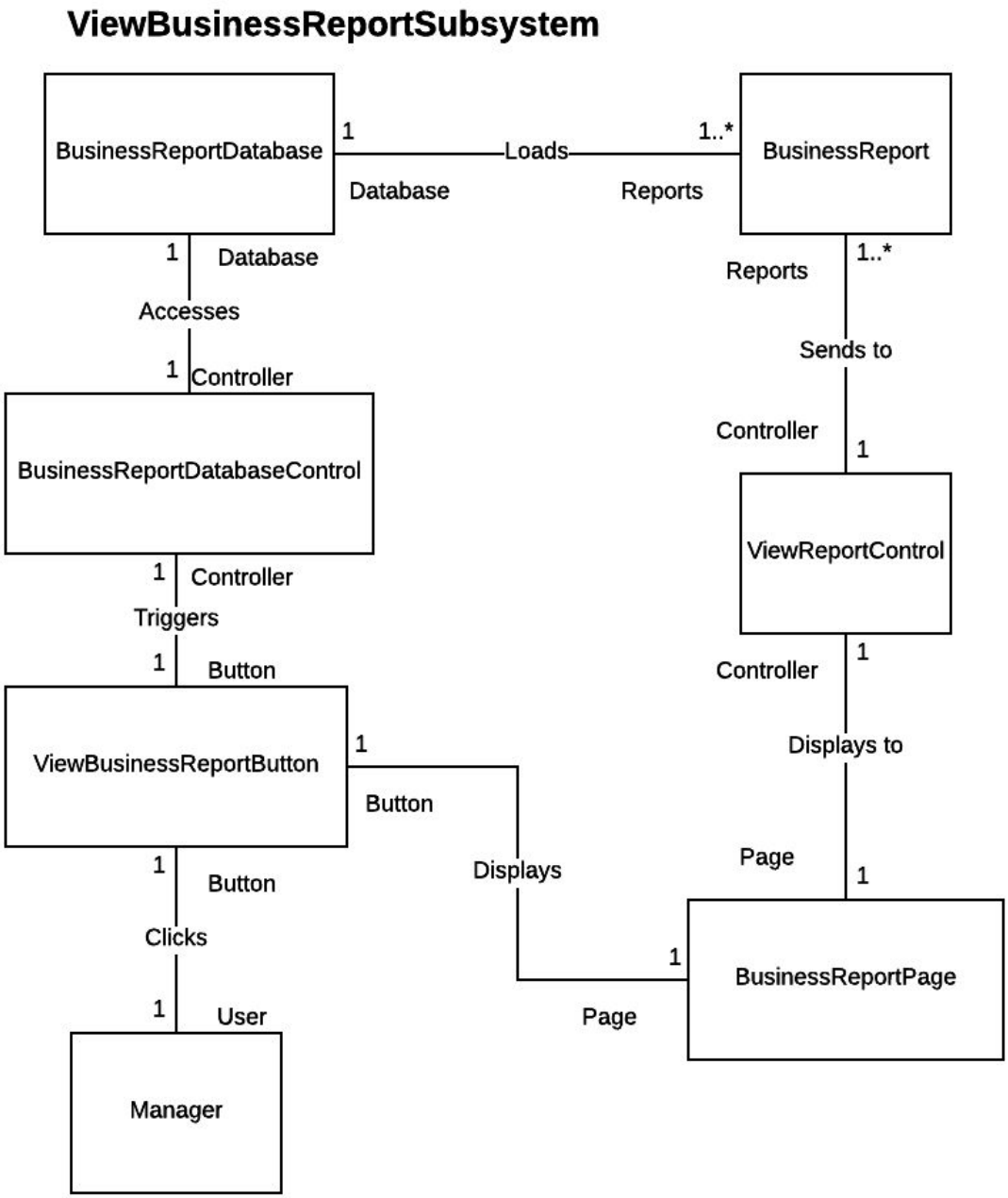
SetOrderingThresholdSubsystem

This subsystem allows the manager to update the inventory items with a set threshold. This threshold will act as a limitation on the quantity of an item the user can add to their order. The threshold is saved to the actual item itself. The subsystem requires access to the inventory database.



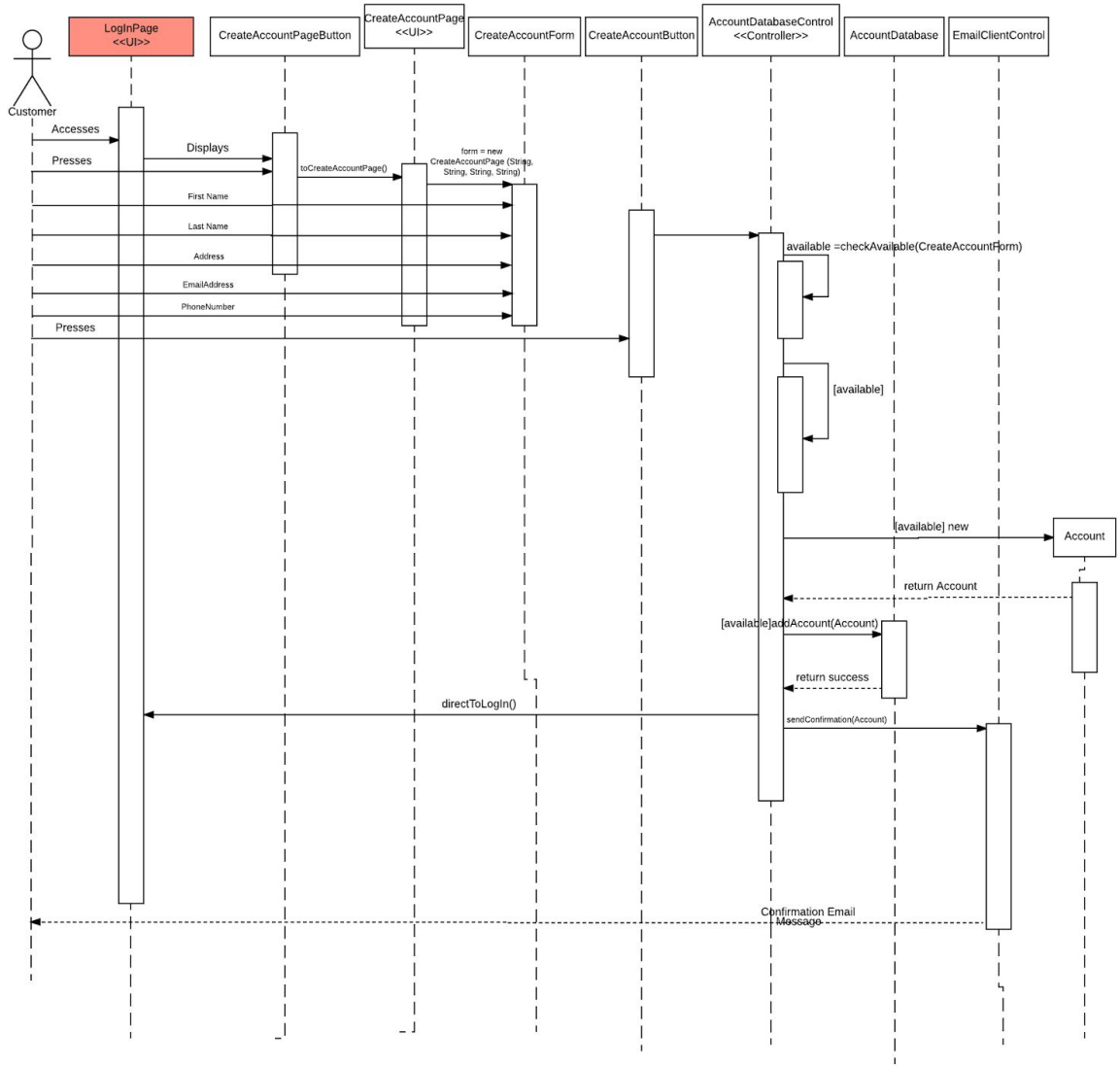
ViewBusinessReportSubsystem

This subsystem exists as a way for the manager to view various business reports calculated by the subsystem. It requires access to the business reports database in order to pull the data to be calculated. It returns and presents a business report for the manager.

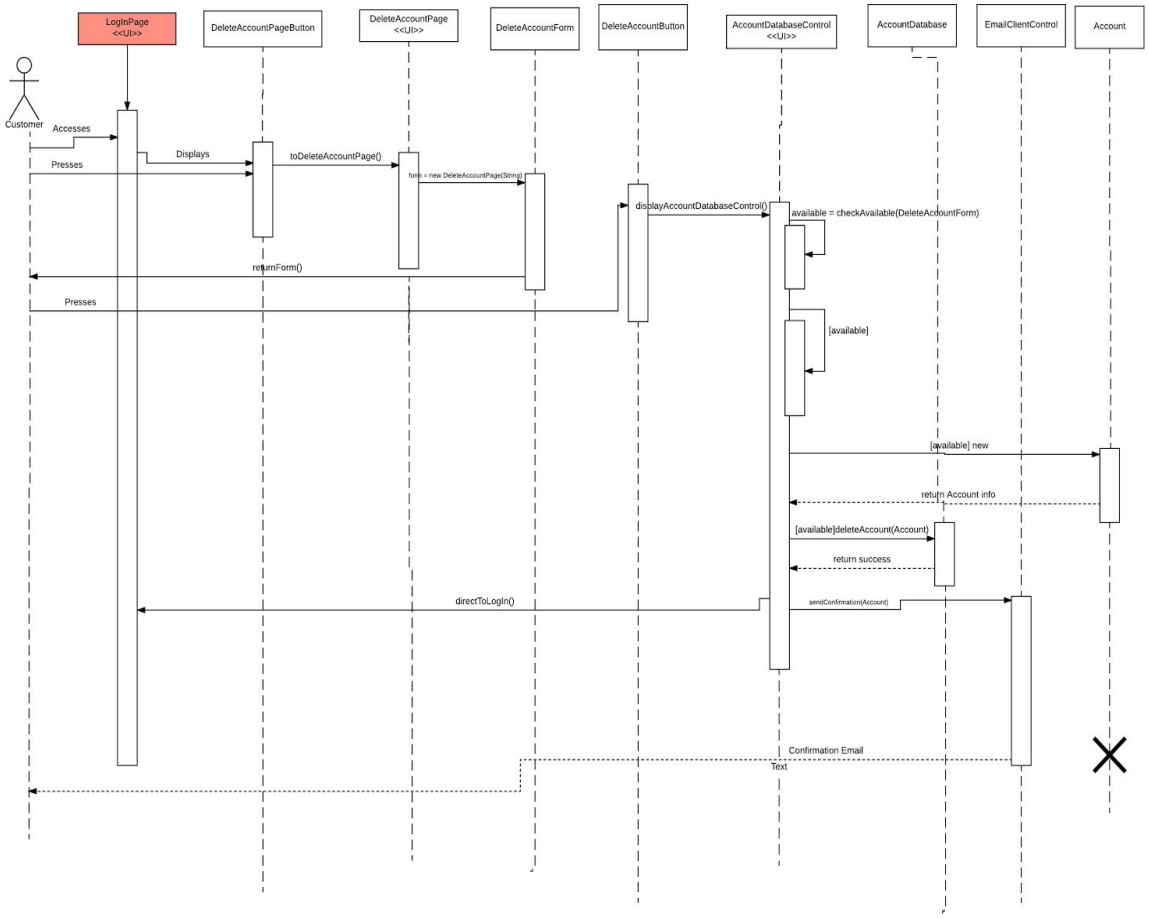


Sequence Diagrams

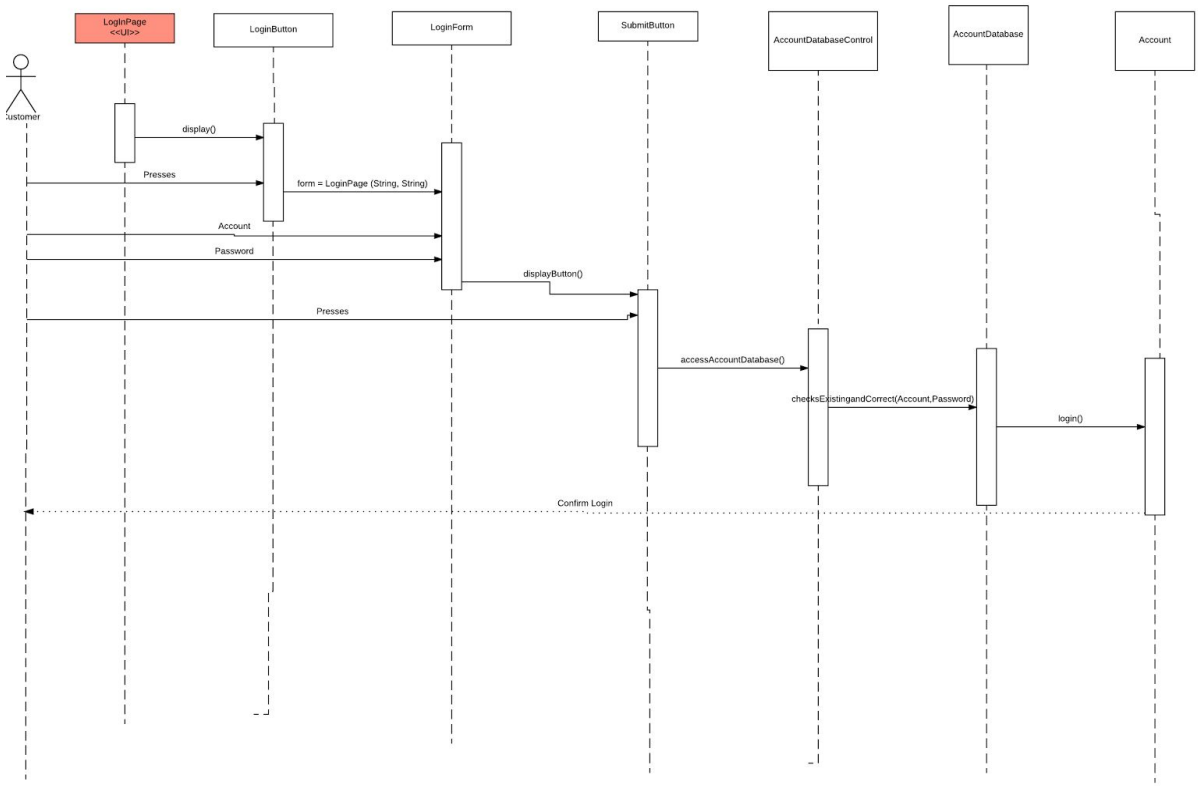
Create Account



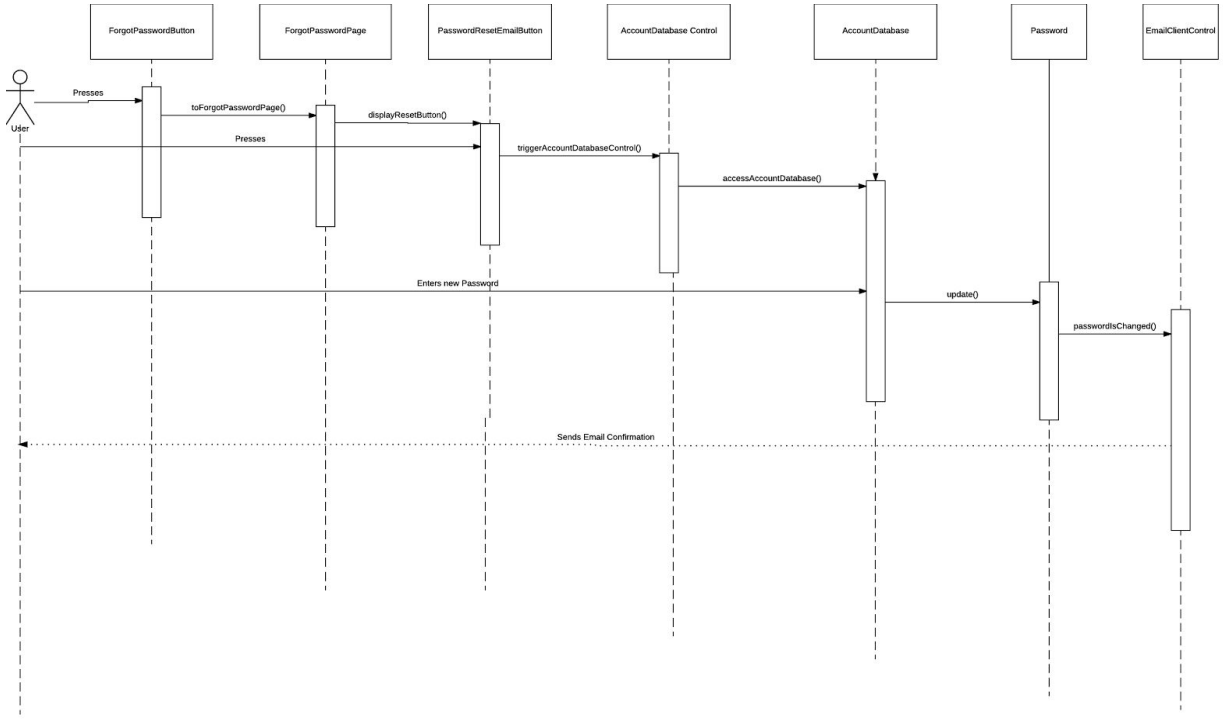
Delete Account



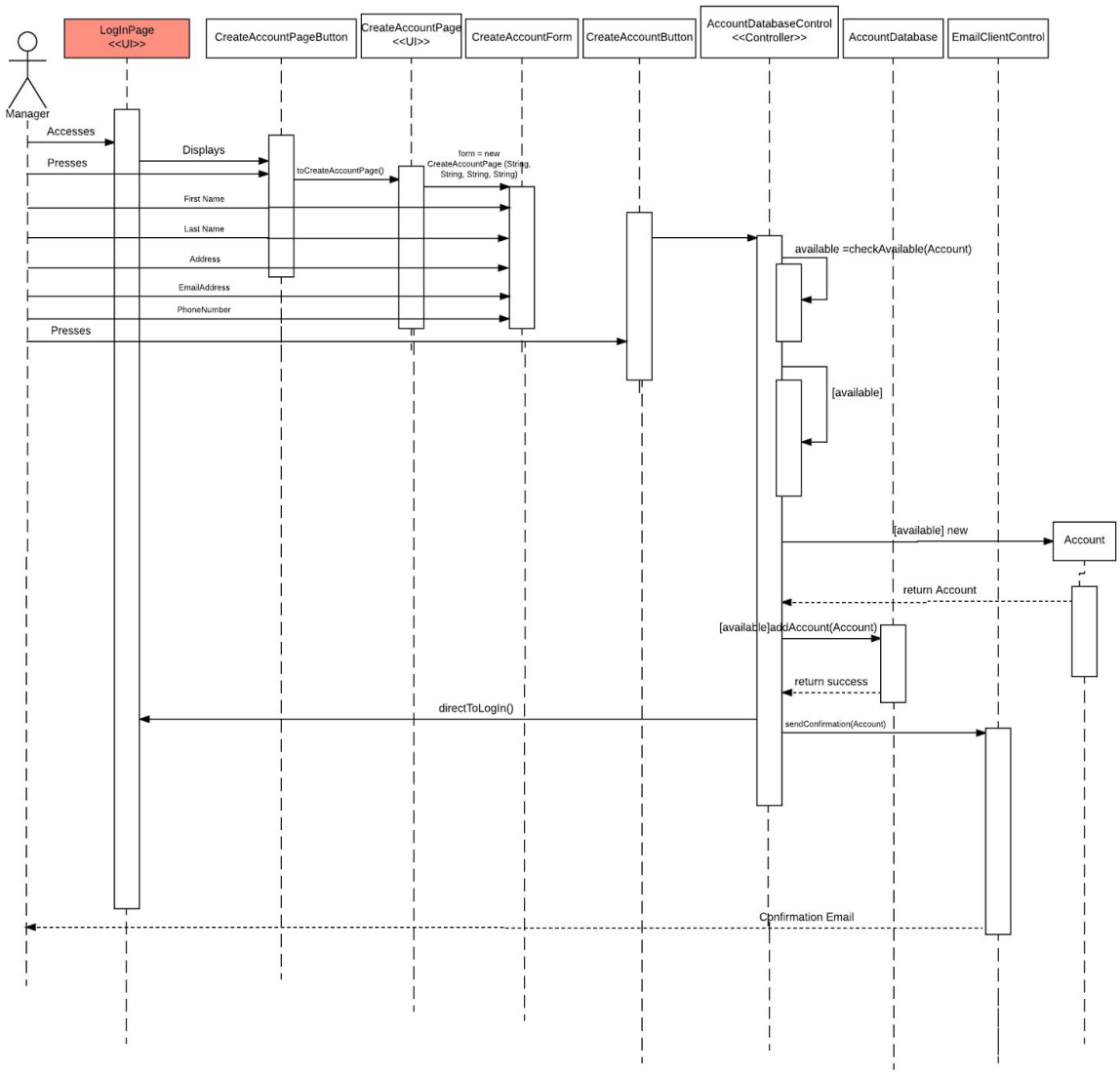
Log In to Account



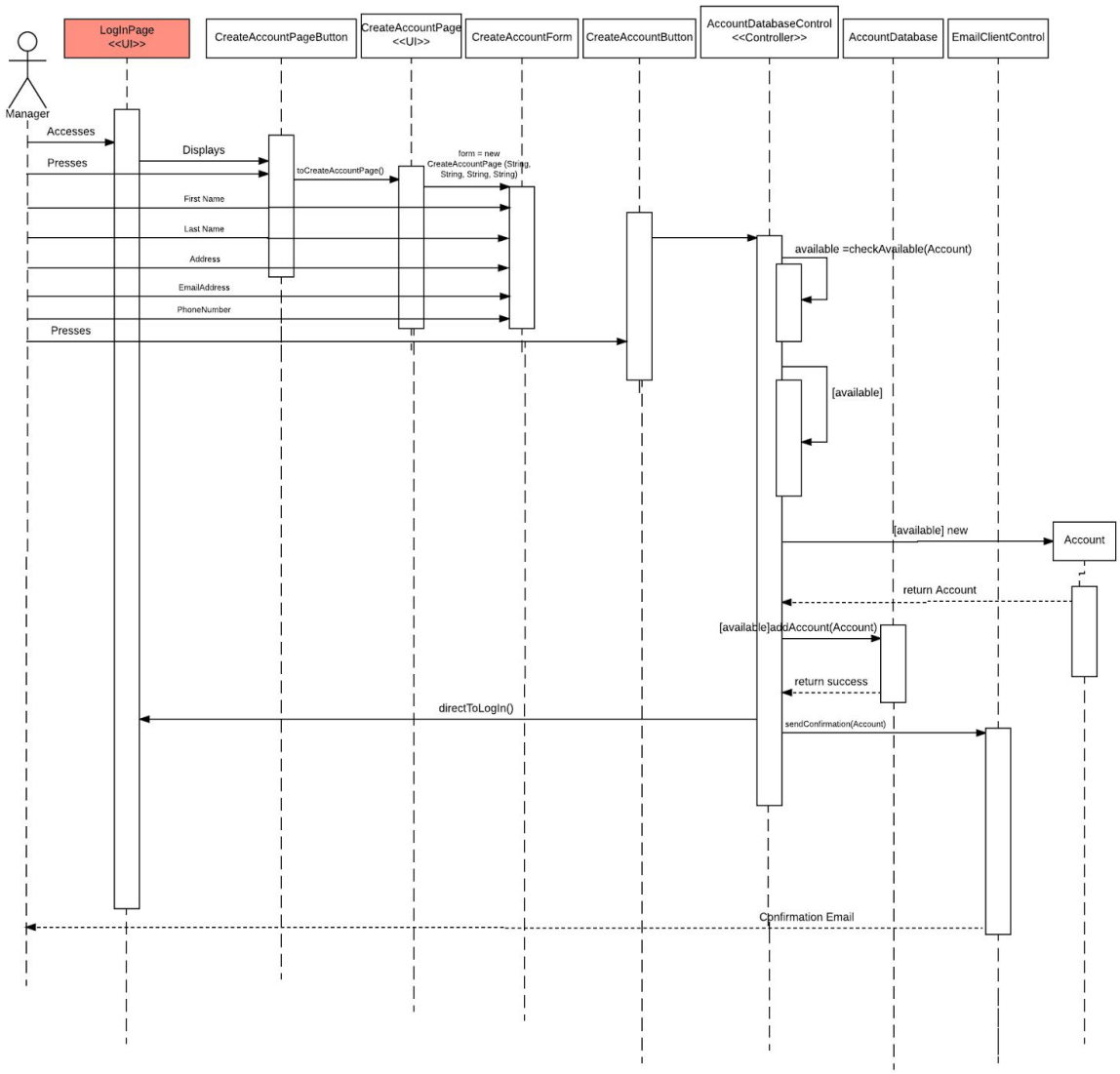
Reset Password



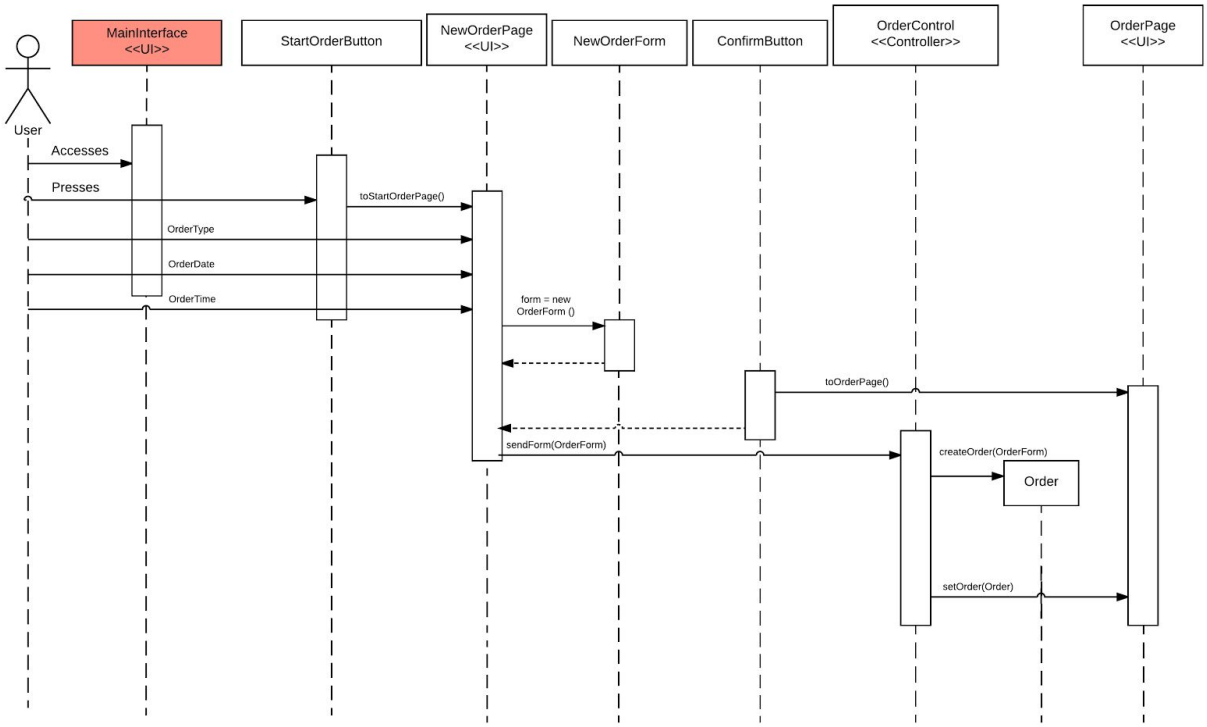
Create an Employee Account



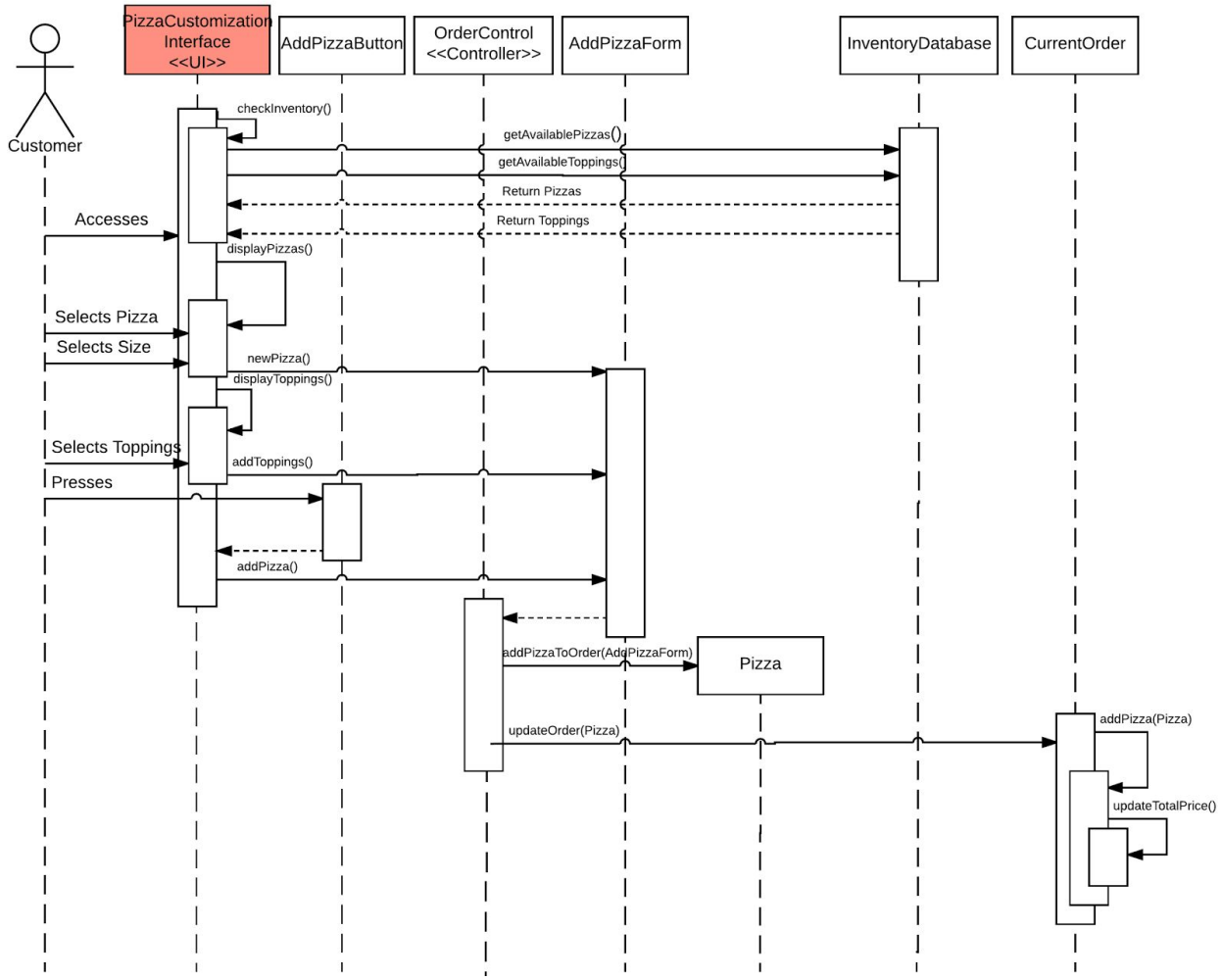
Delete an Employee Account



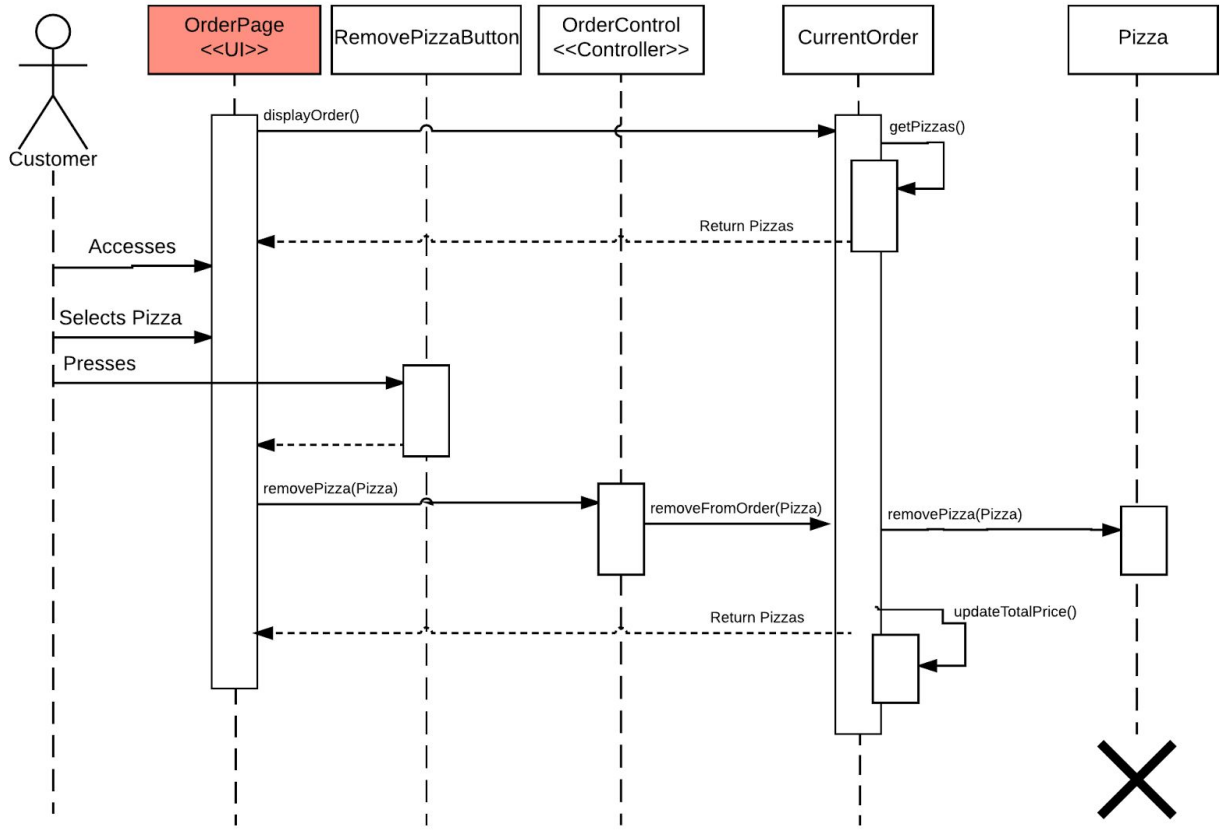
Start an Order



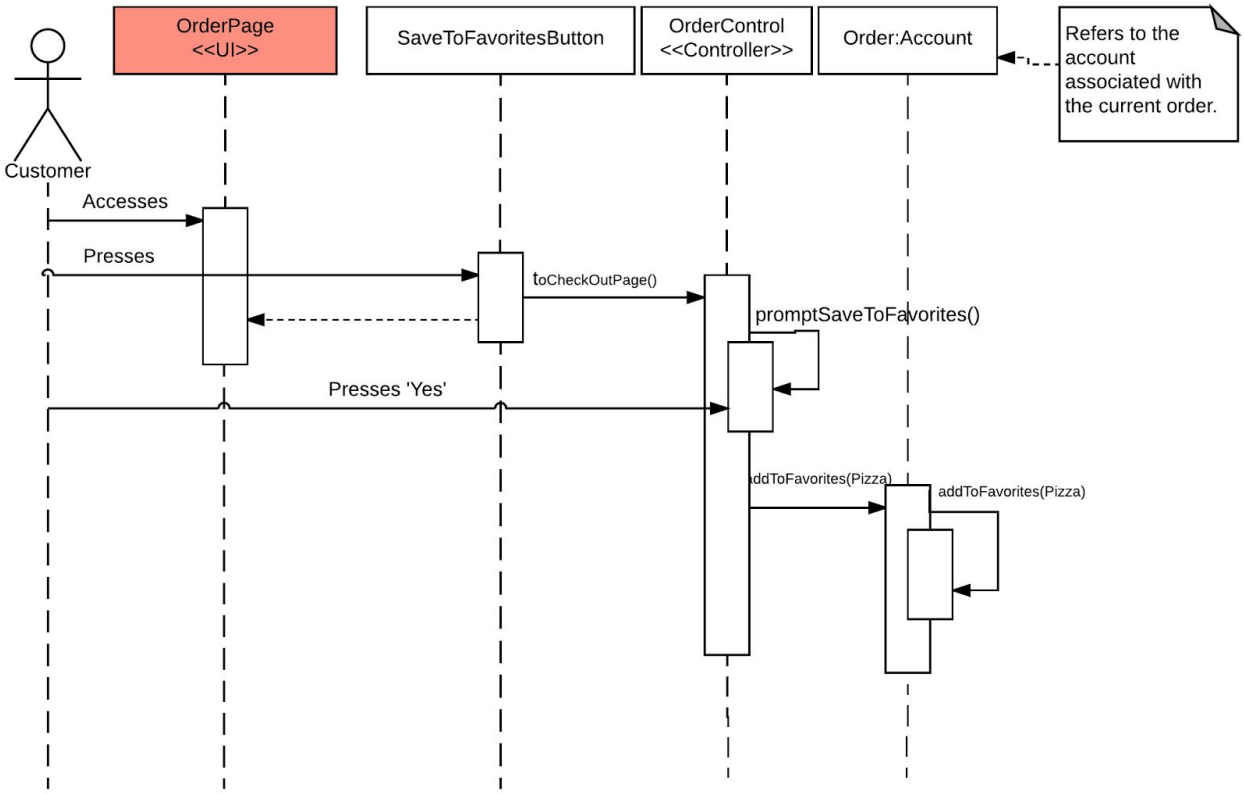
Add a Pizza to an Order



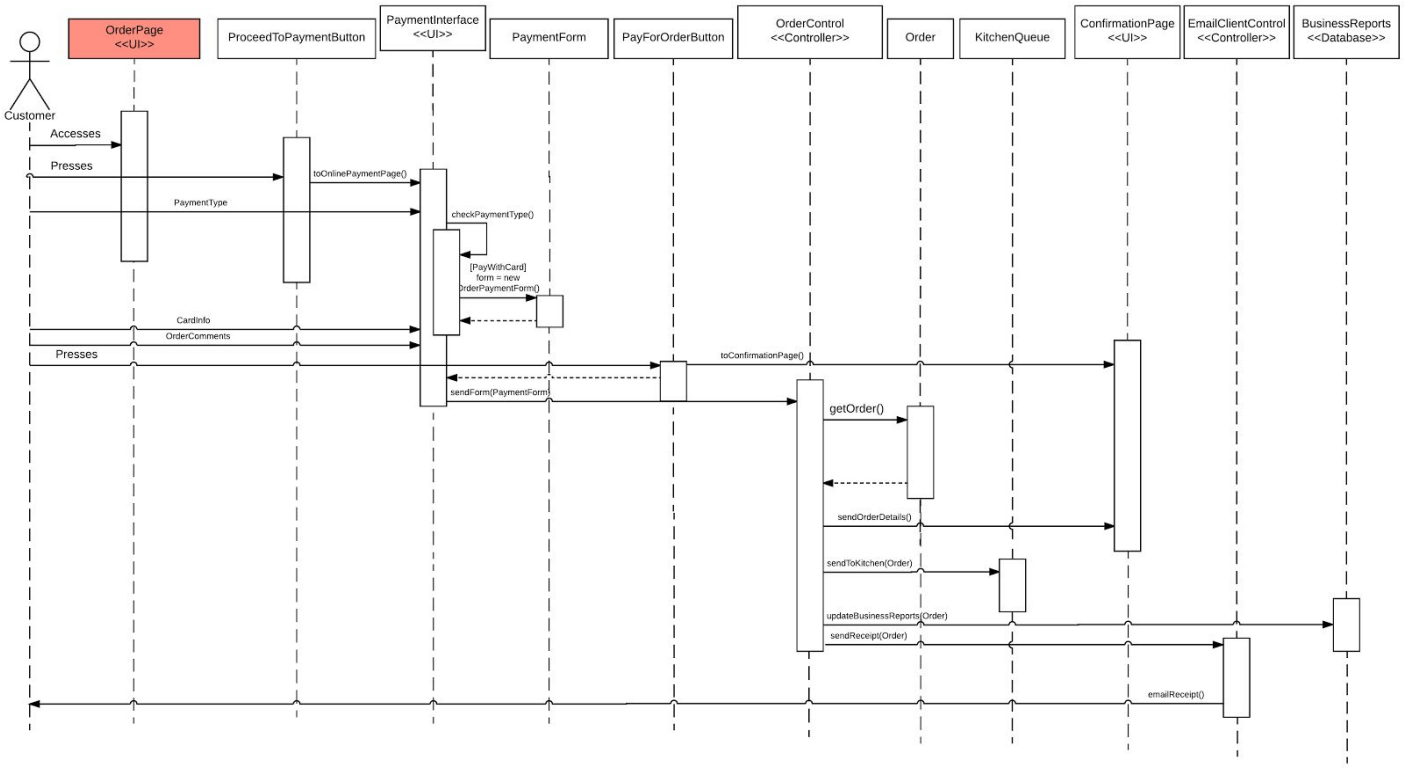
Remove a Pizza from an Order



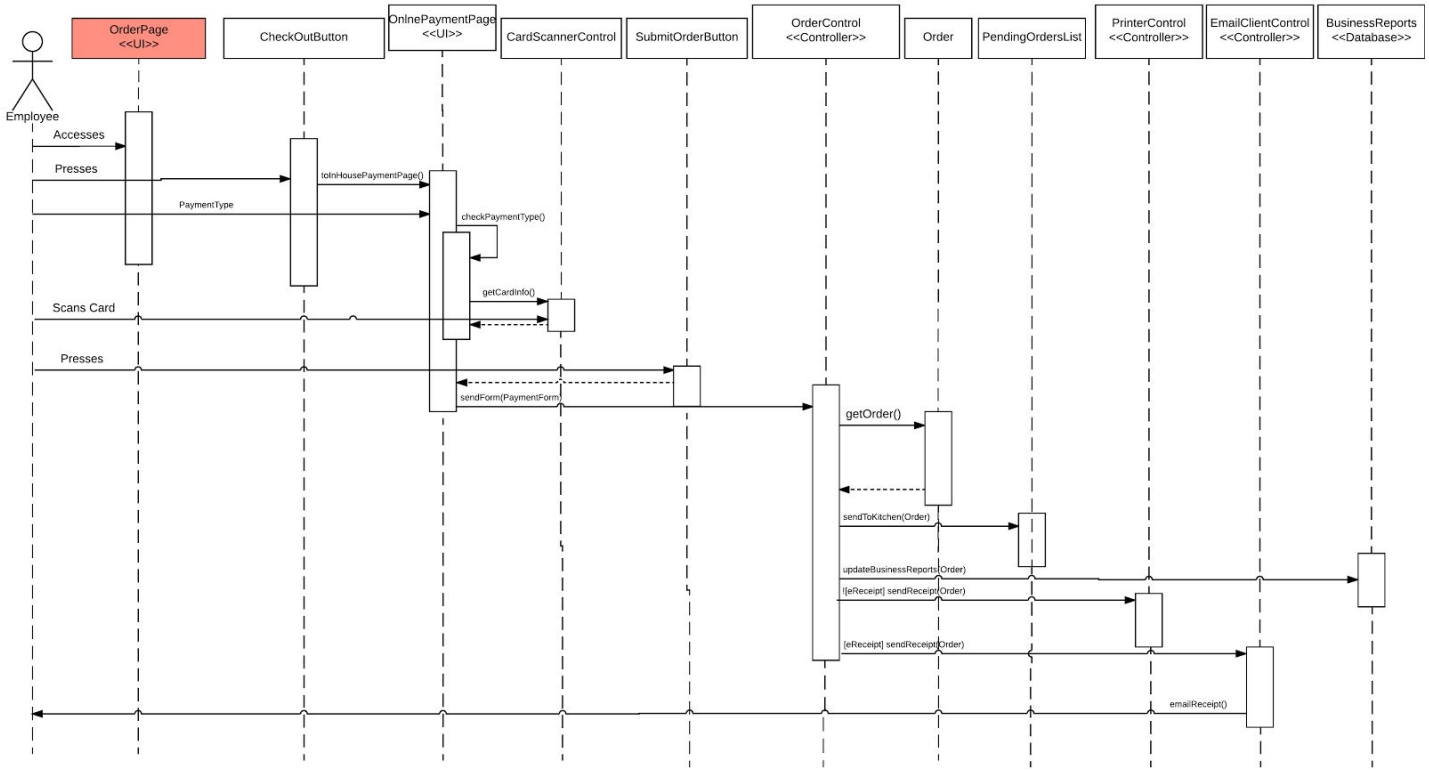
Save to Favorites



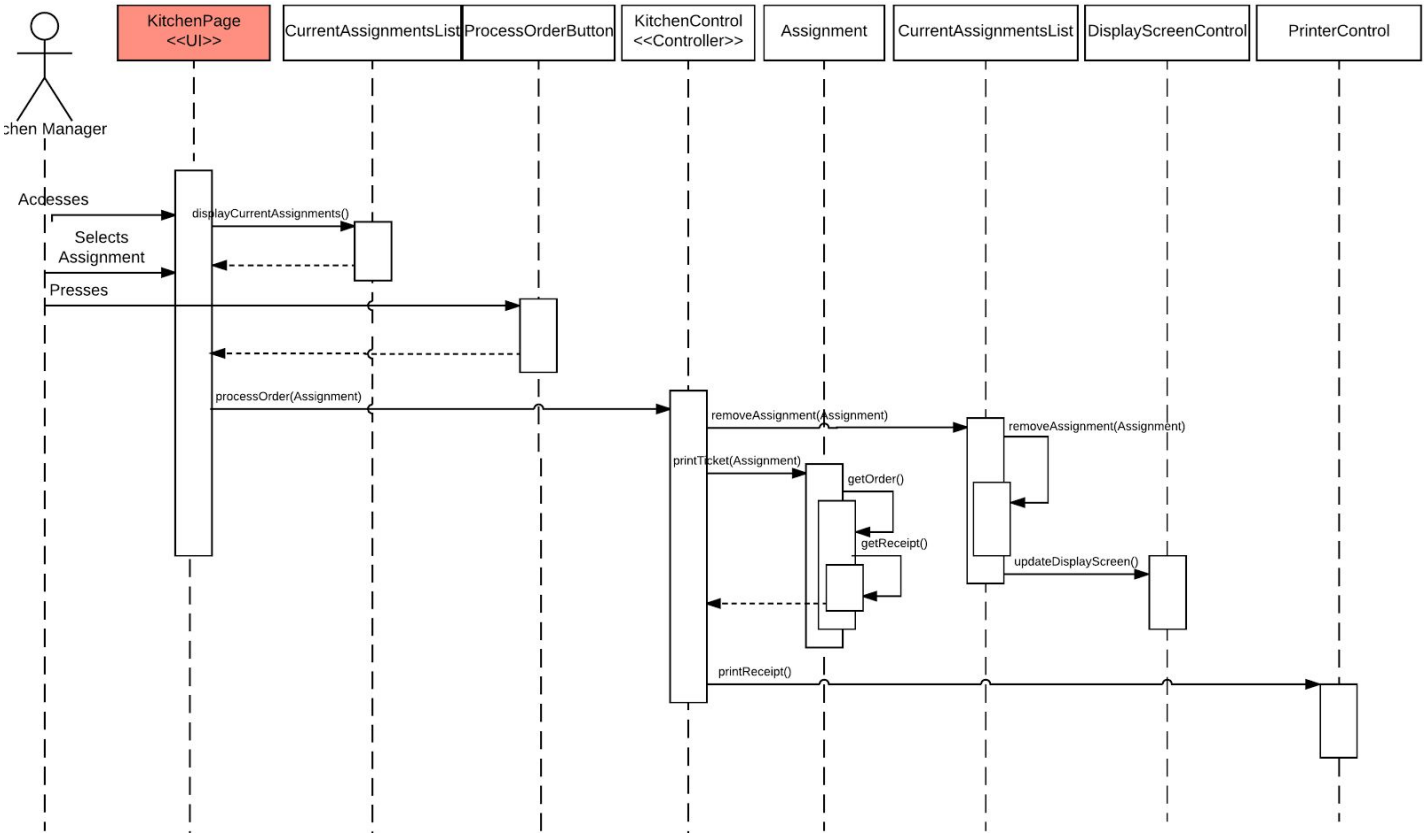
Pay for an Online Order



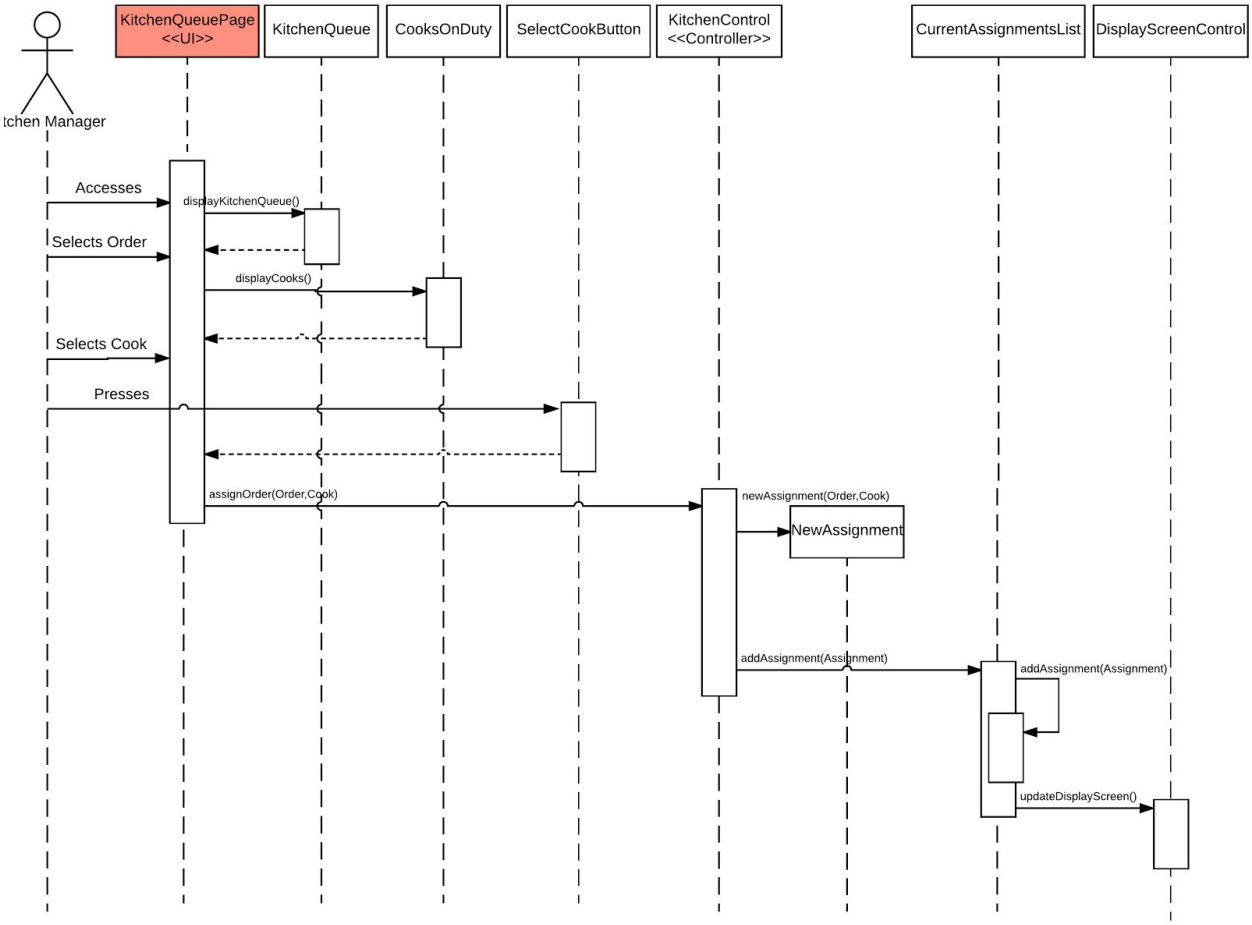
Receive a Customer's Payment



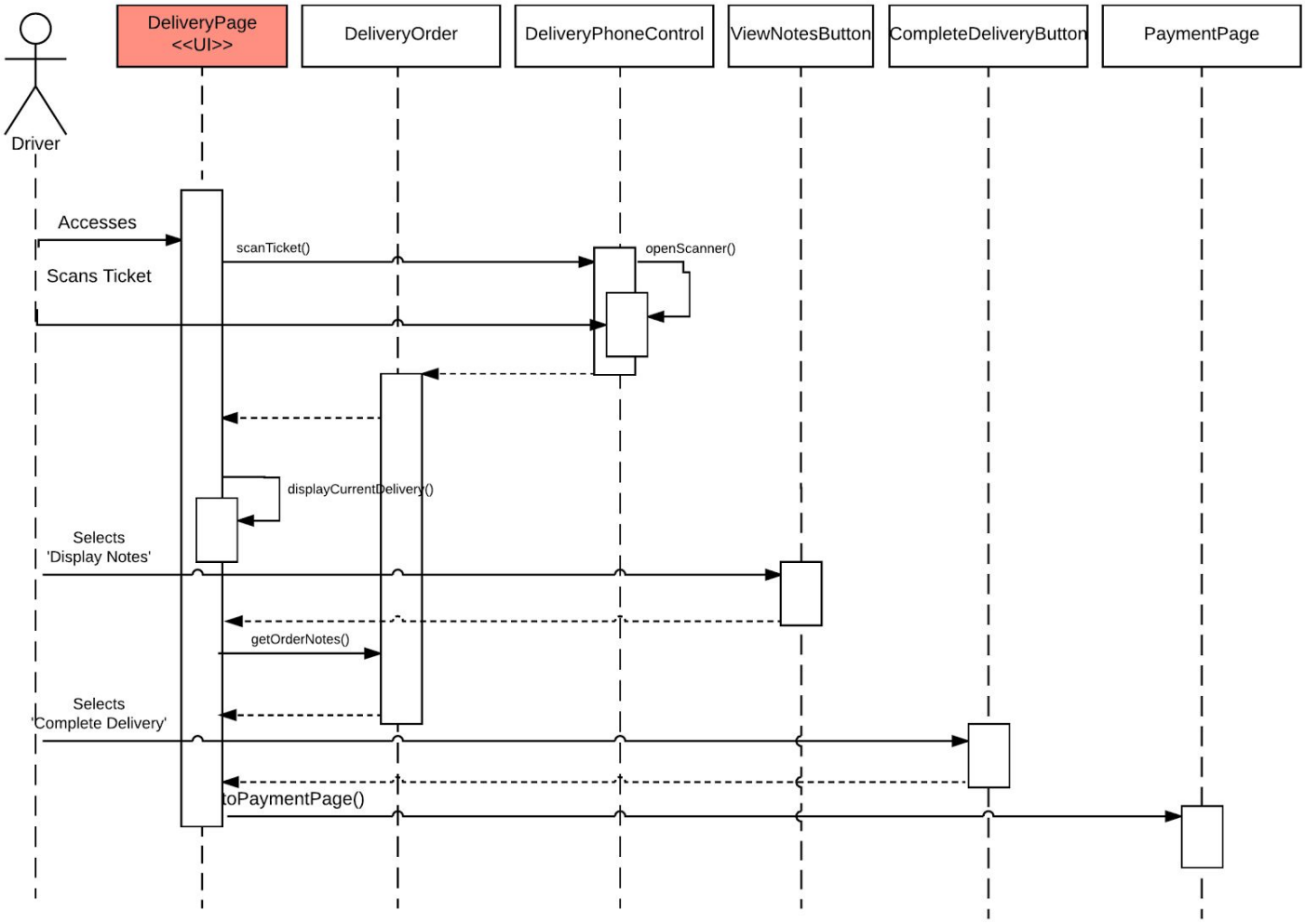
Process an Order



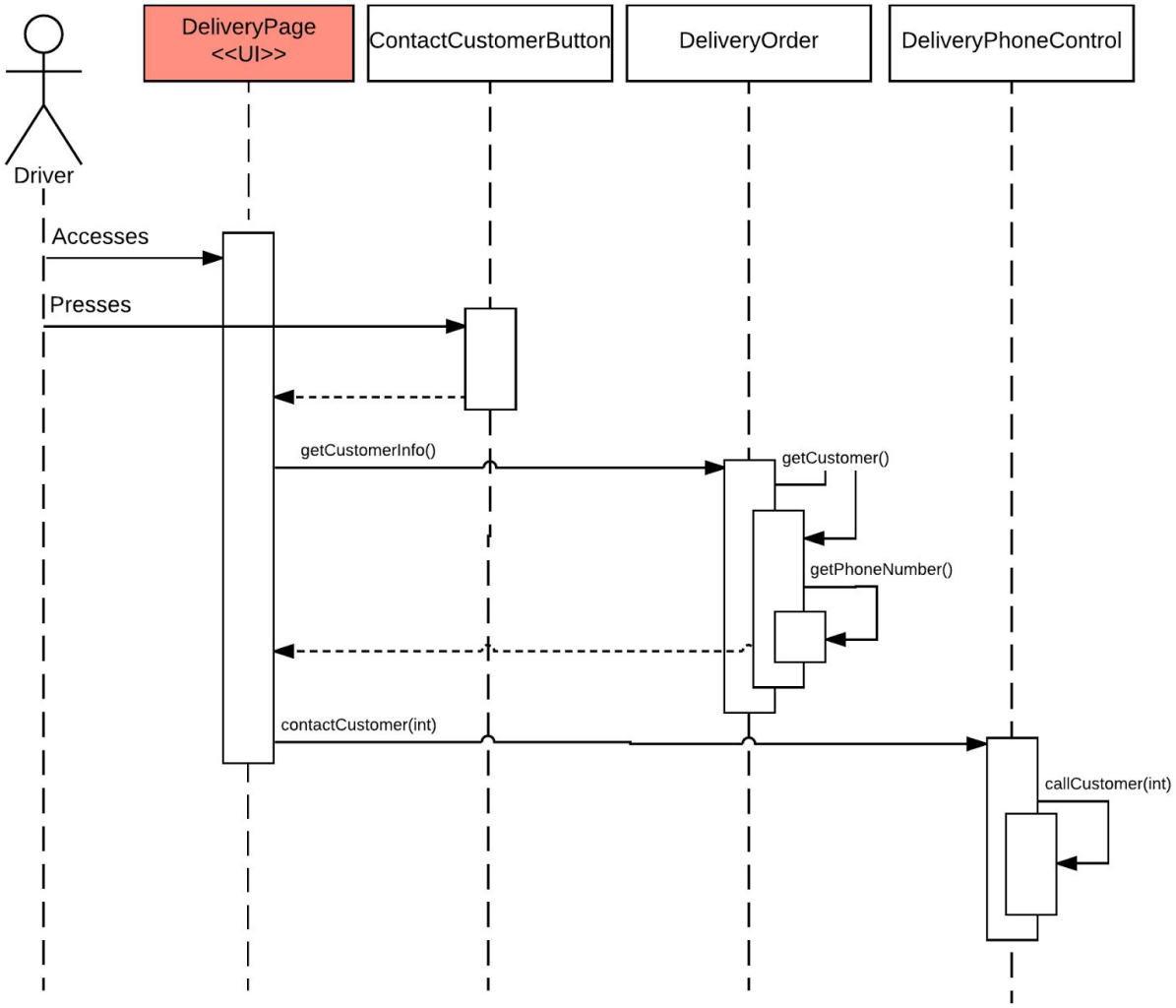
Assign an Order



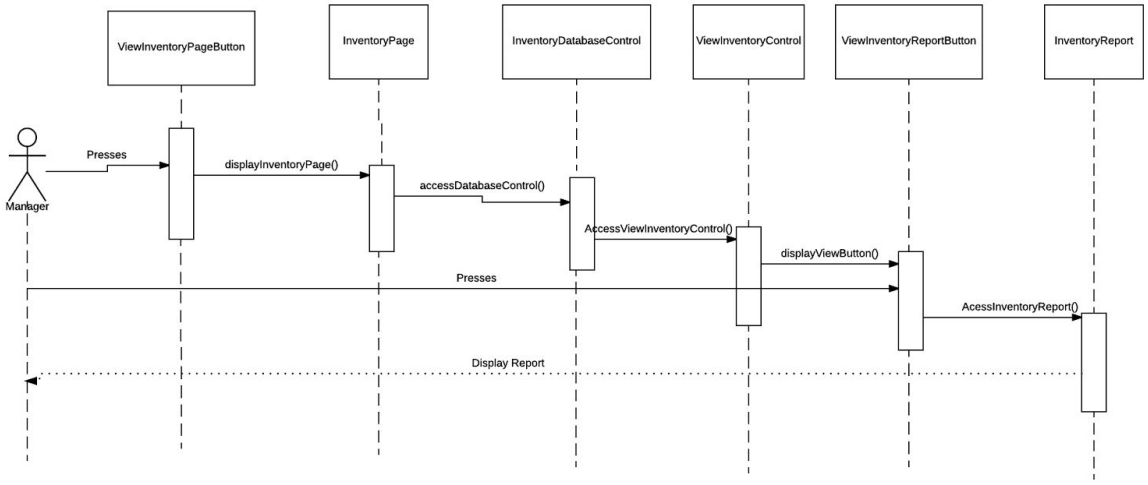
Process a Delivery



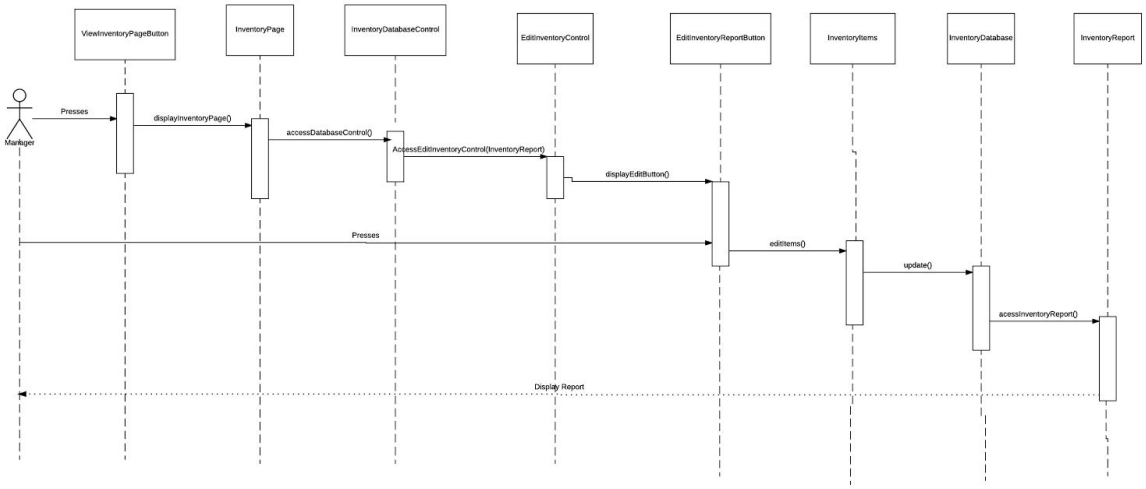
Contact Customer



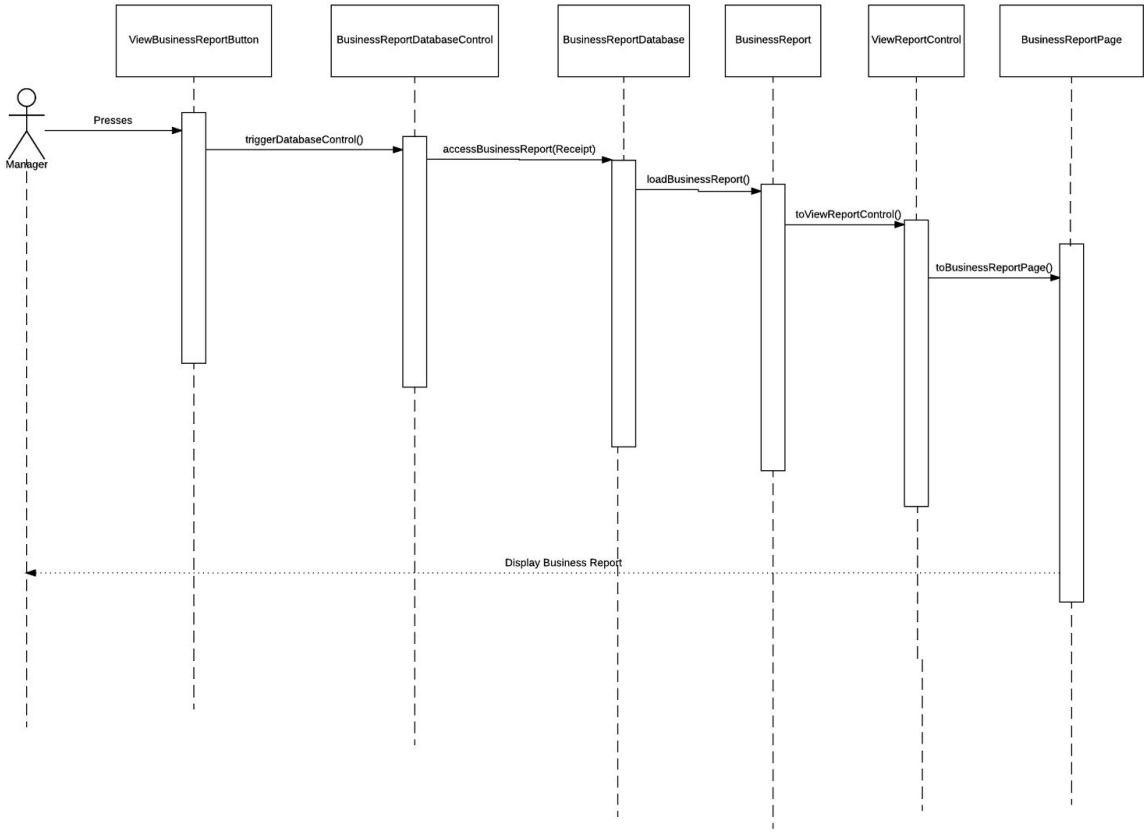
View Inventory Report



Edit Inventory Report



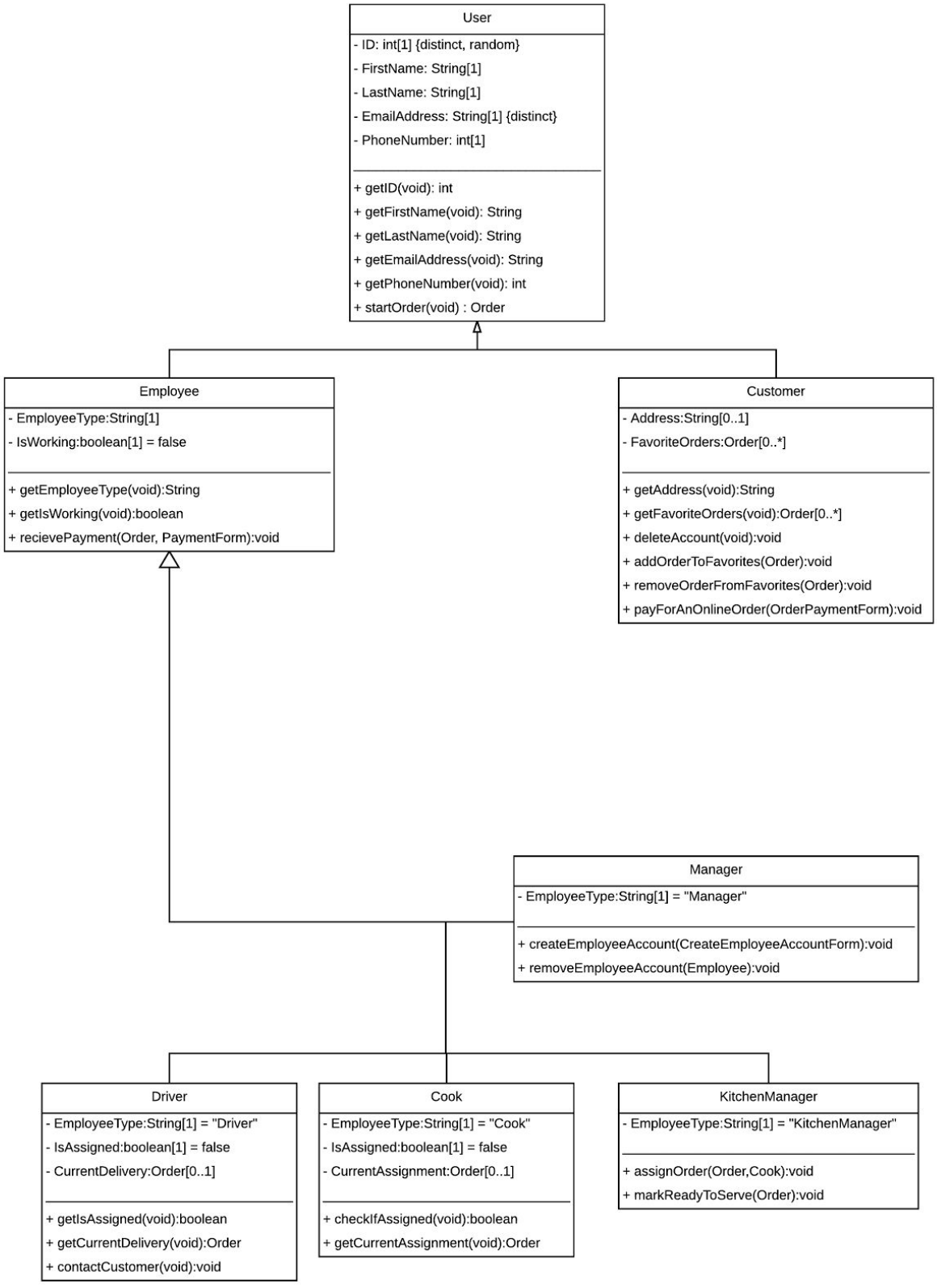
View Business Report



Class Diagrams

1. **Entity Objects**
2. **Databases**
3. **Control Objects**

User Accounts: These classes exist so users can be differentiated/given special functionality.



User Accounts(continued)

Invariants:

- ID must be distinct and random.
- Email Address must be distinct.

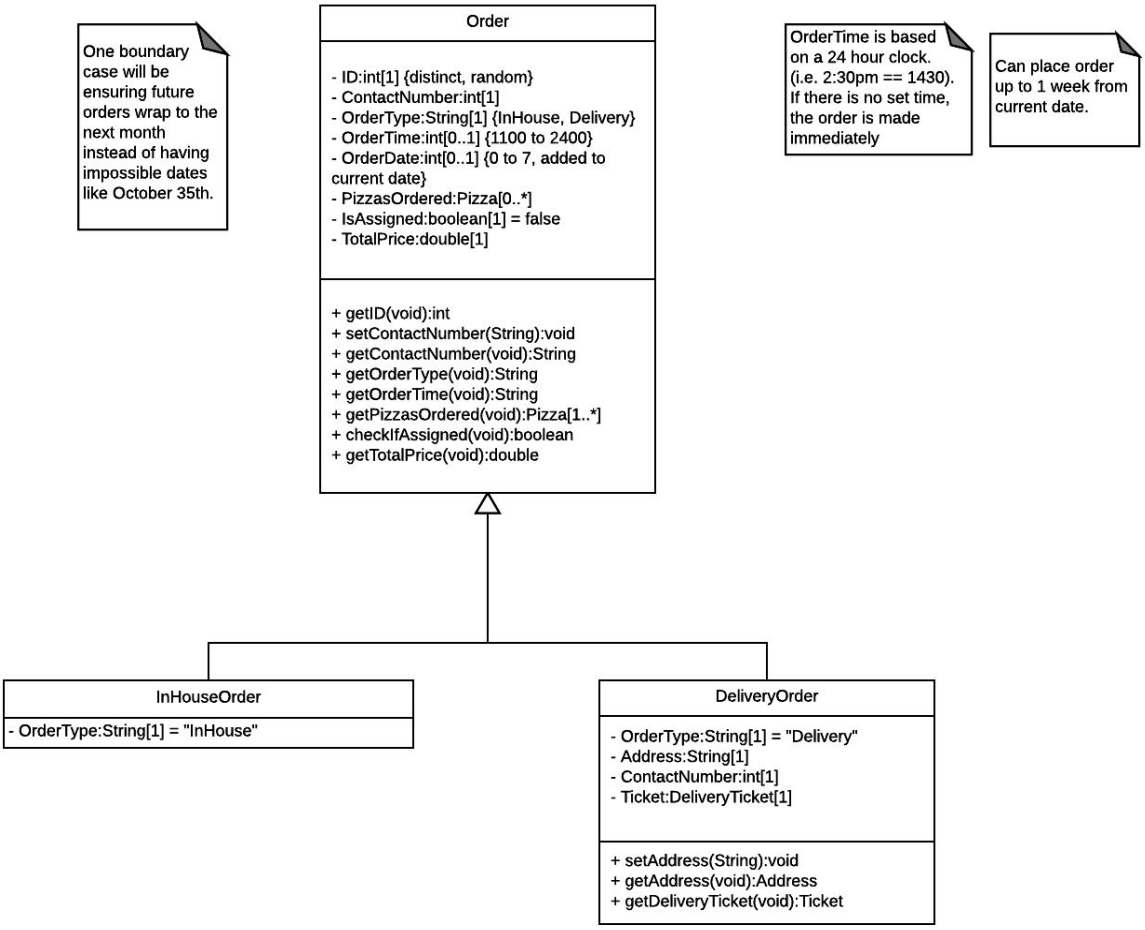
Preconditions:

- Employee:receivePayment(Order,PaymentForm) - Order != NULL && PaymentForm != NULL
- Customer:payForAnOnlineOrder(PaymentForm) - PaymentForm != NULL
- Customer:removeOrderFromPayment(PaymentForm) - PaymentForm != NULL
- Manager:removeEmployeeAccount(Employee) - Employee != NULL
- Driver:contactCustomer(int) - The customer's phone number must be valid.

Postconditions:

- startOrder(void) - An order is scheduled to be made at an acceptable date and time.
- Customer:payForAnOnlineOrder(PaymentForm) - The customer will be sent a confirmation/receipt email.
- Manager:createEmployeeAccount(CreateEmployeeAccountForm) - A new employee account will be created - EmployeeAccount != NULL
- Driver:getCurrentDelivery(void) - Driver:isAssigned == true
- Cook:getCurrentAssignment(void) - Cook:isAssigned == true

Order: These objects represent the orders that are placed at Polaski's Pizza.



Invariants:

- ID must be distinct and random.

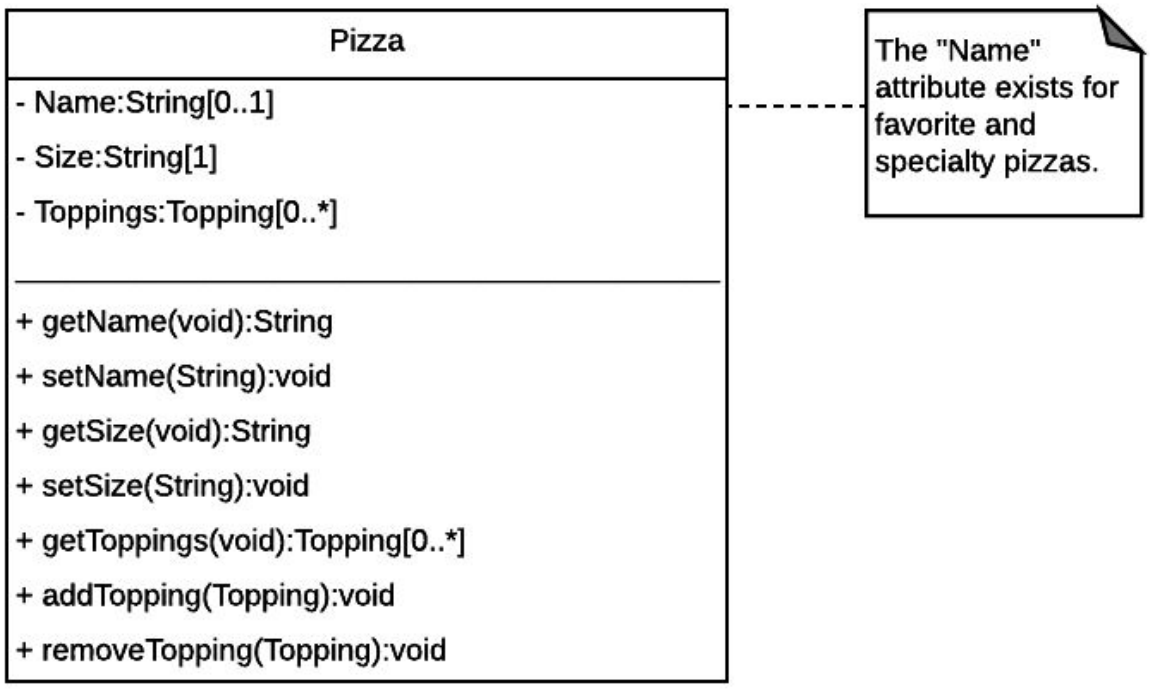
Preconditions:

- OrderType == InHouse || OrderType == Delivery
- 1100 <= OrderTime <= 2400
- 0 <= OrderDate <= 7
- DeliveryOrder:setAddress(String) - The address must exist and be in delivery range.

Postconditions:

- setContactNumber(int) - Order:ContactNumber == int
- DeliveryOrder:setAddress(String) - DeliveryOrder:Address == String

Pizza: Contains the attributes of a pizza.



Invariants: N/A

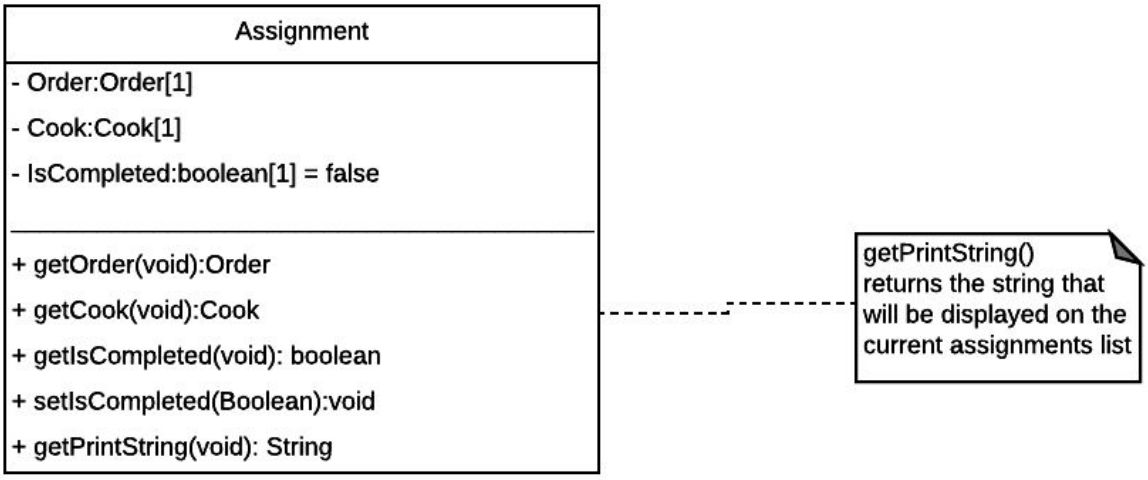
Preconditions:

- setSize(String) - String = “Small”, “Medium”, or “Large”
- addTopping(Topping) - Topping must be in inventory.
- removeTopping(Topping) - Topping must already be on the pizza.

Postconditions:

- setName(String) - Pizza:Name == String
- setSize(String) - Pizza:Size == String
- addTopping(Topping) - Pizza:Toppings += Topping
- removeTopping(Topping) - Pizza:Toppings -= Topping

Assignment: A pairing of an order, and the cook assigned to make it



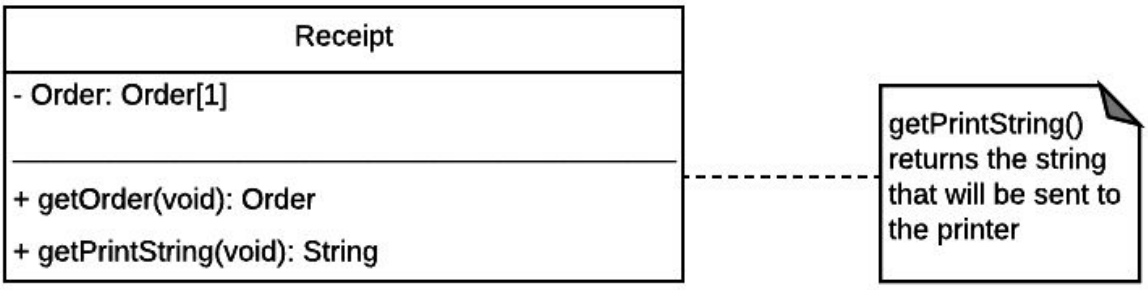
Invariants:

- Order == #Order
- Cook == #Cook

Preconditions: N/A

Postconditions: N/A

Receipt: Represents the receipt for an order. Can be emailed or printed out.



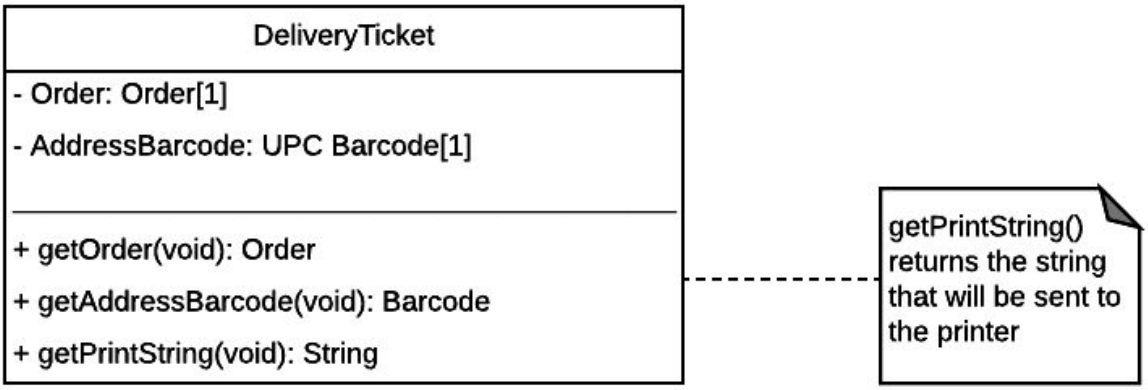
Invariants:

- Order = #Order

Preconditions: N/A

Postconditions: N/A

DeliveryTicket: A receipt with a UPC barcode, so the driver can get directions to the customer's house.



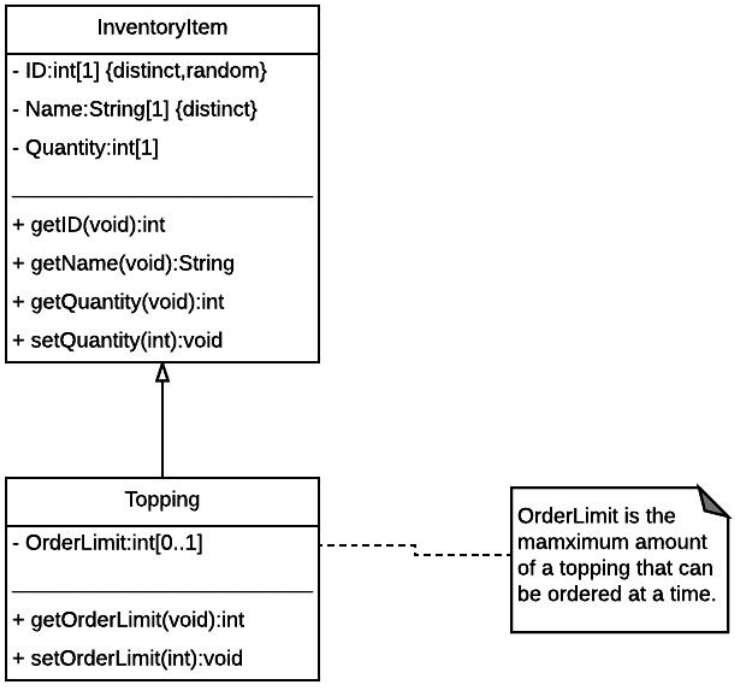
Invariants:

- Order = #Order

Preconditions: N/A

Postconditions: N/A

InventoryItems: Represents a commodity in Polaski Pizza's inventory, so the quantities of each item can be easily tracked and modified. Includes pizza toppings.



Invariants:

- ID is distinct and random.
- Name is distinct.

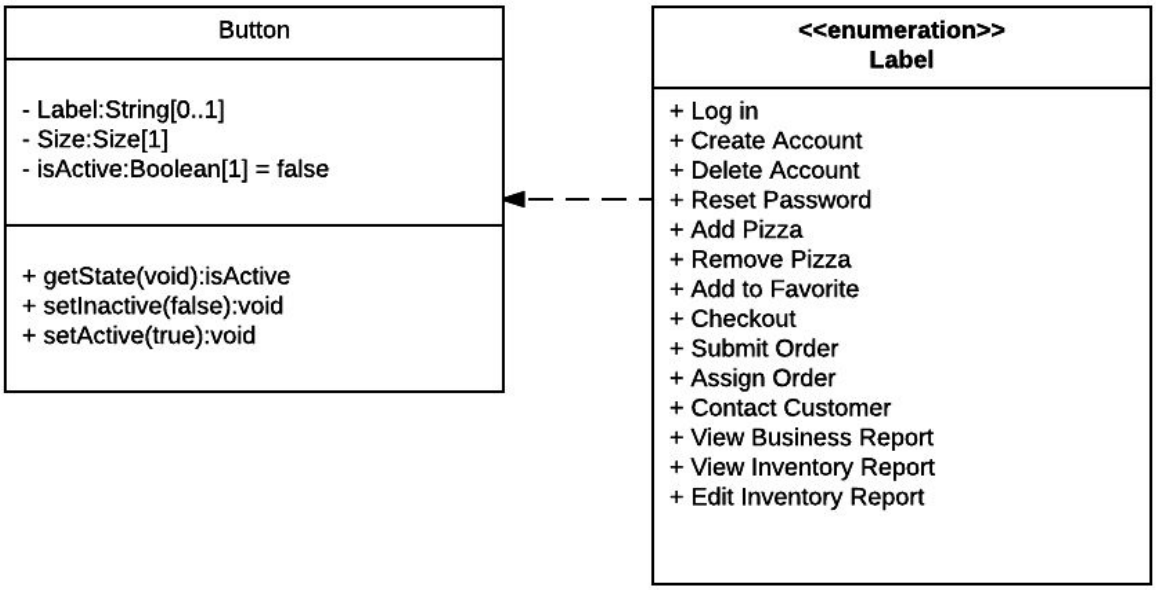
Preconditions:

- setQuantity(int) - int >= 0.
- Topping:setOrderLimit(int) - int > 0.

Postconditions:

- getQuantity(void) must return an int >= 0.
- Topping:getOrderLimit(void) must return an int > 0.

Button: Class exists so that the user can interact with the interface, since using a terminal to navigate the menus would be extremely inconvenient.



Invariants:

- Label == #Label
- Size == #Size

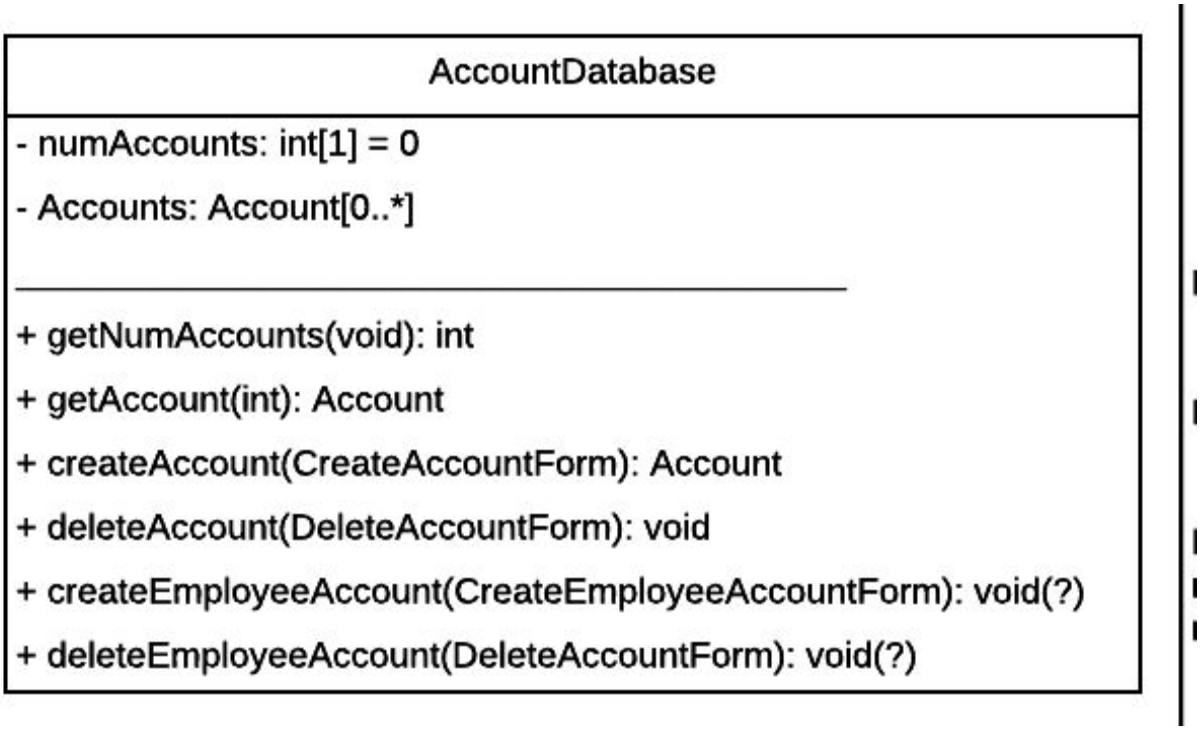
Preconditions:

- Size >= 0

Postconditions:

- getState() != #getState()

AccountDatabase: Class exists to store accounts, which is a persistent data type that holds customer information such as their name and phone number.



Invariants:

- Account == #Account

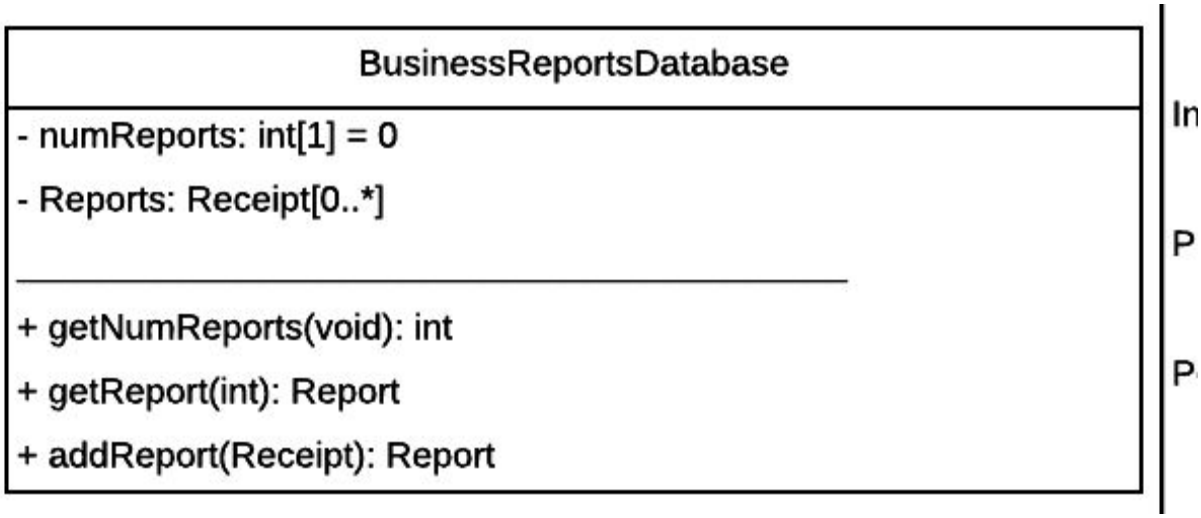
Preconditions:

- numAccounts >= 0

Postconditions:

- numAccounts >= #numAccounts-1 && numAccounts <= #numaccounts+1

BusinessReportsDatabase: Class exists to store business reports, which is a persistent data type that holds order information included in receipts.



Invariants:

- Receipt == #Receipt

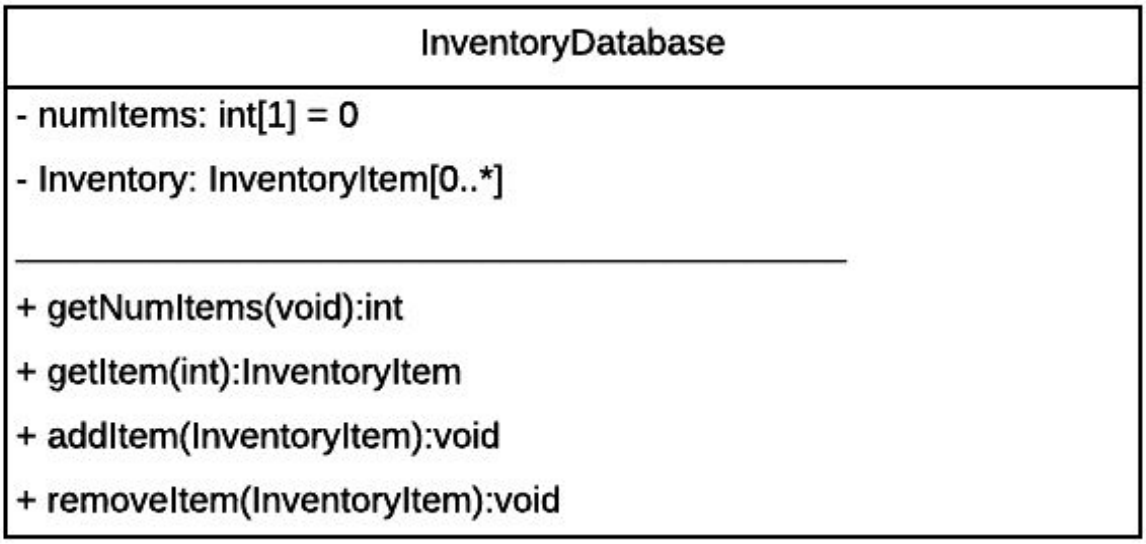
Preconditions:

- Receipt != NULL

Postconditions:

- numReports >= #numReports

InventoryDatabase: Class Exists to store inventory reports, which is a persistent data type that holds information regarding the ingredients, such as how many of each item are left and their projected expiration dates.



Invariants:

- None

Preconditions:

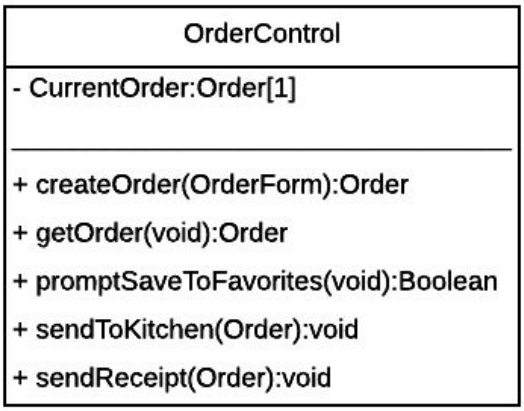
- numItems >= 0

Postconditions:

- numItems >= 0 (but can be different than at the function call)

Control Objects

OrderControl: Manages all the operations involved in placing an order.



Invariants:

- CurrentOrder == #CurrentOrder

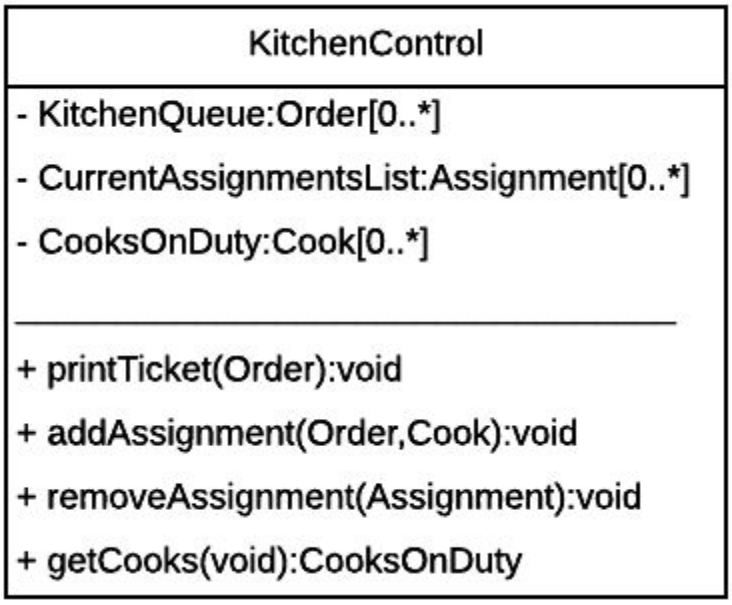
Preconditions:

- createOrder(OrderForm) - The form must have a valid date and time.
- updateOrder(Pizza) - The toppings on the pizza must not exceed any ordering limits.
- sendToKitchen(Order) - CurrentOrder:TotaPrice > 0.00 (Order must not be empty)
- sendReceipt(Order) - The payment of the order must be confirmed.

Postconditions:

- updateOrder(Pizza) - the Pizza's attributes will be updated.
- sendToKitchen(Order) - The Order will be sent to the Kitchen Queue.
- sendReceipt(Order) - The associated Receipt will be sent to the printer.

KitchenControl: Manages all the operations of an order as it is processed in the kitchen.



Invariants: N/A

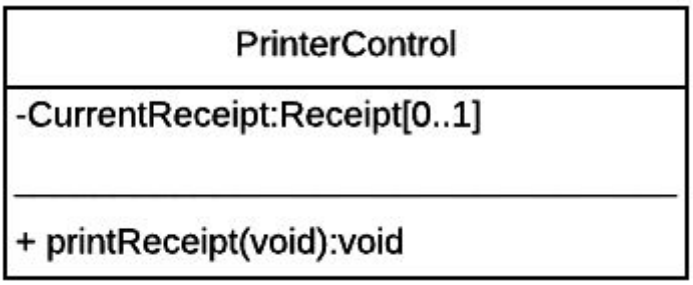
Preconditions:

- printTicket(Order) - Order:OrderType = “Delivery”
- addAssignment(Order,Cook) - Cook:IsAssigned = false (The cook must not already be making an order.)
- removeAssignment(Assignment)
- getCooks(void) - The cooks must all be on shift.

Postconditions:

- printTicket(Order) - The ticket will be sent to the receipt printer.
- addAssignment(Order,Cook) - A new assignment will be made and will be sent to the receipt printer.
- removeAssignment(Assignment) - CurrentAssignmentsList -= Assignment.

PrinterControl: The control object associated with the printer.



Invariants:

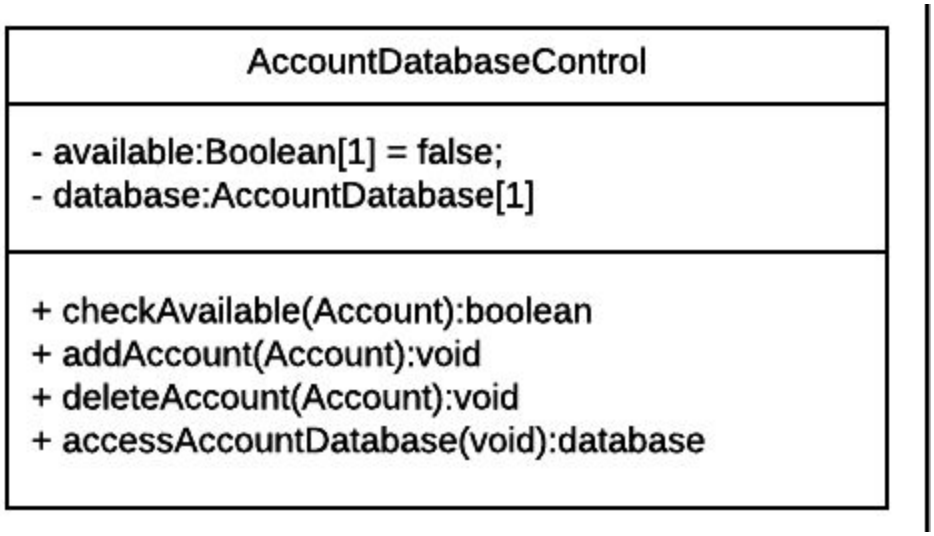
- CurrentReceipt == #CurrentReceipt

Preconditions: N/A

PostConditions:

- printReceipt(void) - The receipt is printed.

AccountDatabaseControl: Class Exists to interact with attributes in the AccountDatabase class. It's necessary to realize the use cases that create and delete accounts.



Invariants:

- numAccounts >= 0

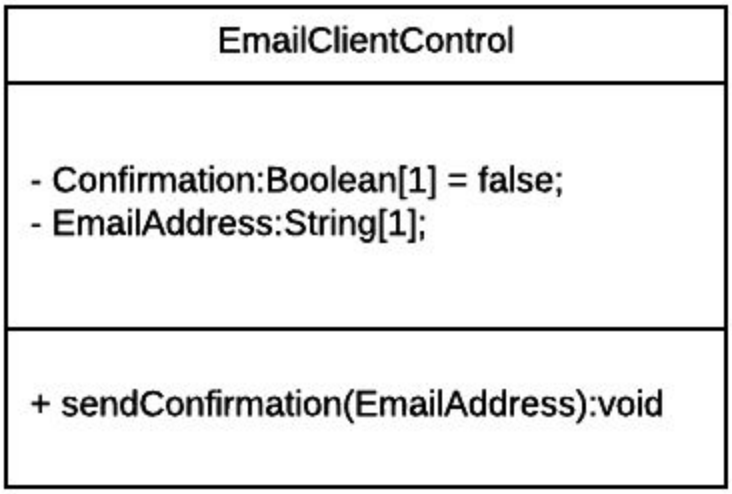
Preconditions:

- None

Postconditions:

- numAccounts >= #numAccounts -1 && numAccounts <= #numaccounts +1

EmailClientControl: Class Exists to inform the user of any sucessful operations, such as creating an account or ordering a pizza, without bloating the other classes.



Invariants:

- emailAddress == #emailAddress

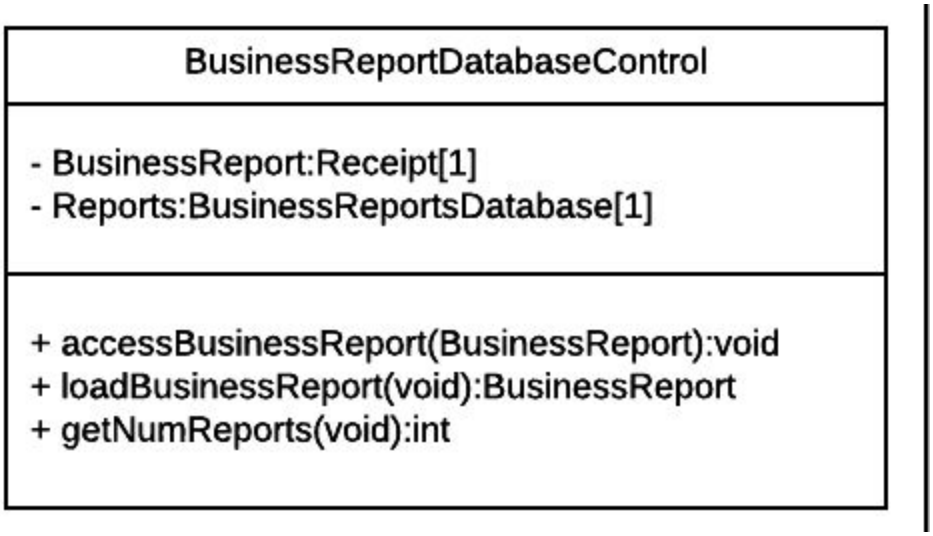
Preconditions:

- Confirmation == true

Postconditions:

- None

BusinessReportDatabaseControl: Class Exists to interact with attributes in the BussinessReportsDatabase class. It's necessary to implement the use cases that views business reports.



Invariants:

- BusinessReport == #BusinessReport

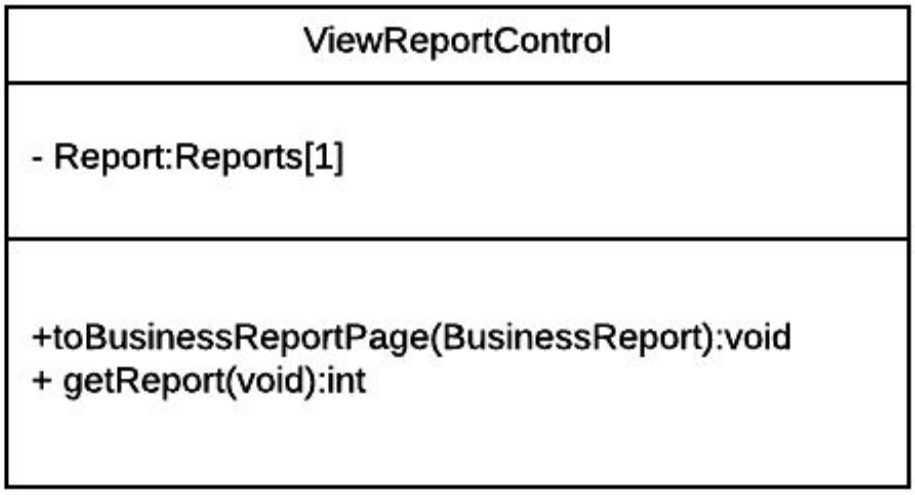
Preconditions:

- BusinessReport != NULL

Postconditions:

- None

ViewReportControl: Class Exists to activate the methods in the business reports database class. It's also necessary to implement the use cases that views business reports.



Invariants:

- Report == #Report

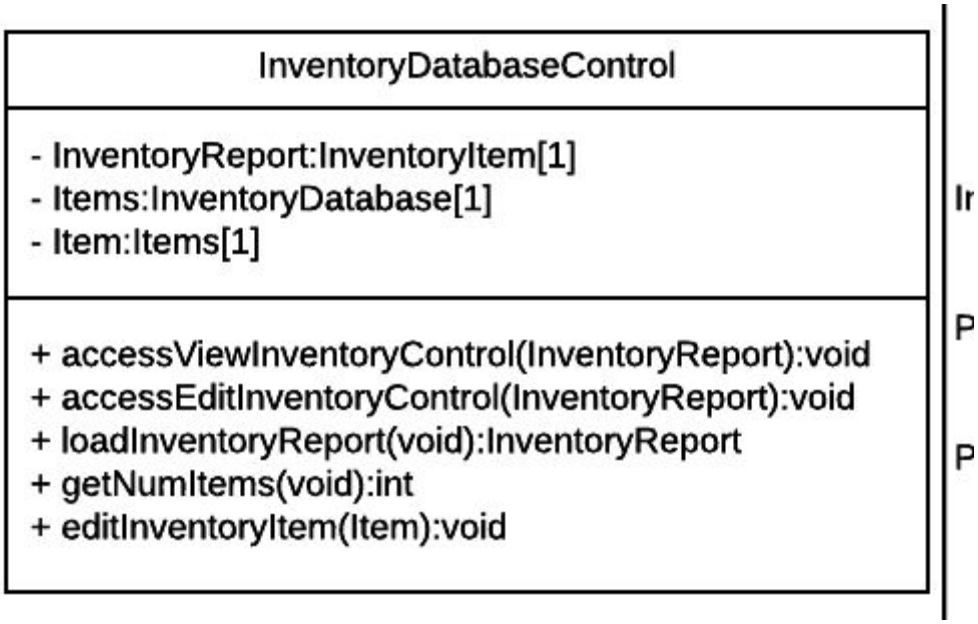
Preconditions:

- BusinessReport != NULL

Postconditions:

- None

InventoryDatabaseControl: Class Exists to interact with attributes in the InventoryDatabase class. It's necessary to implement the use cases "view inventory report" and "edit inventory report."



Invariants:

- None

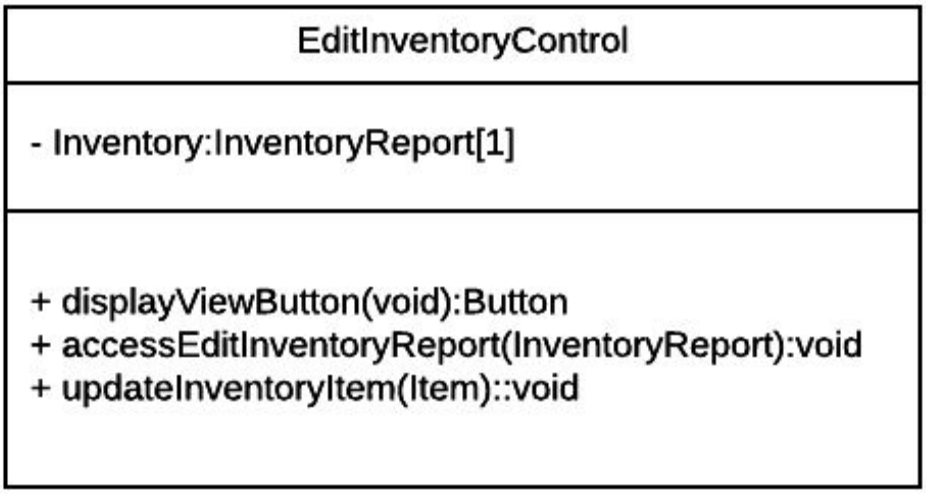
Preconditions:

- InventoryReport != NULL

Postconditions:

- numItems >= 0

EditInventoryControl: Class Exists to activate the methods in the InventoryDatabase class. It's also necessary to implement the use case "edit inventory report"



Invariants:

- numItems >= 0

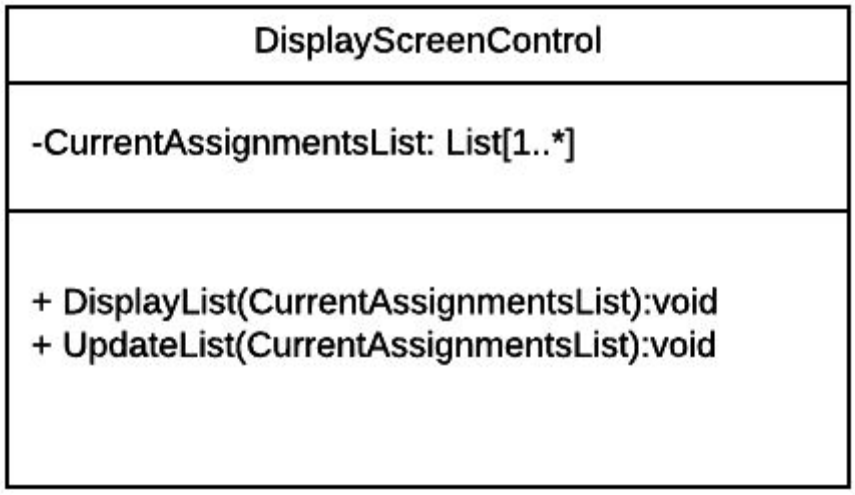
Preconditions:

- Inventory != NULL

Postconditions:

- None

DisplayScreenControl: Class Exists to access and display the current assignments to the kitchen screen.



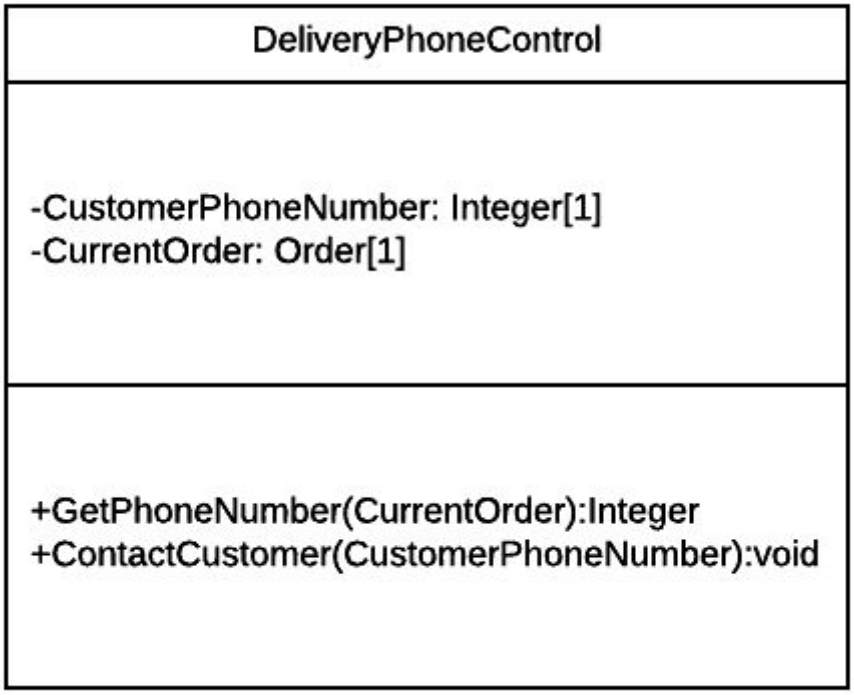
Invariants:
None
Preconditions:

- DisplayList >= 1

Postconditions:

- UpdateList >= 1

DeliveryPhoneControl: Class Exists to access the customer's phone number and contact them if need be. It is necessary to implement the "contact customer" use case.



Invariants:

- CustomerPhoneNumber == #CustomerPhoneNumber
- Order == #Order

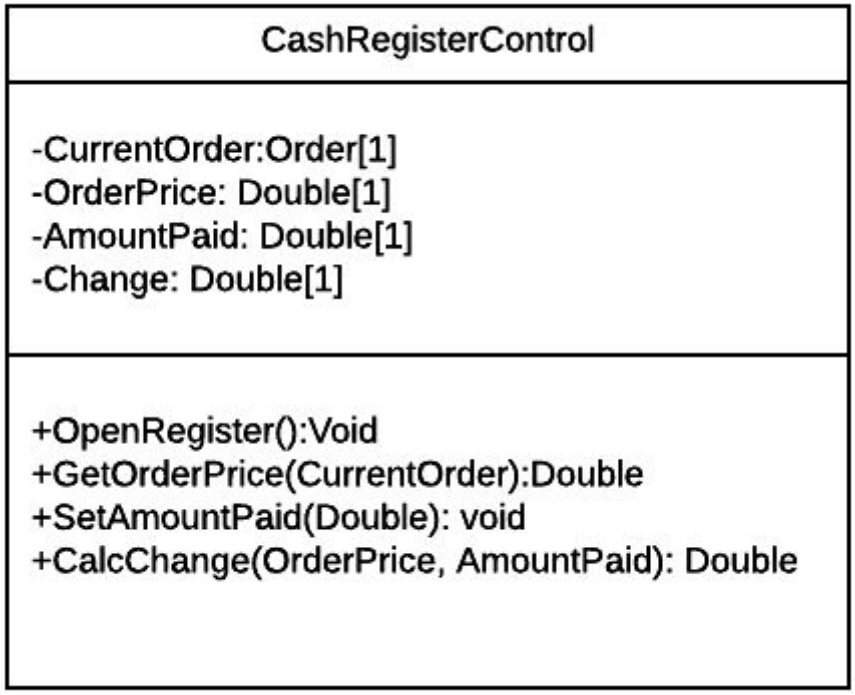
Preconditions:

- CustomerPhoneNumber.size.equals(10)
- CurrentOrder != NULL

Postconditions:

- None

CashRegisterControl: Class Exists to access the cash register and compute the amount of change to be given to a customer. It is necessary to implement the “receive customer payment” use case.



Invariants:
None

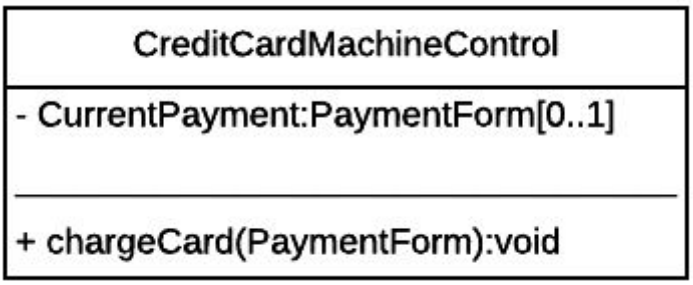
Preconditions:

- CurrentOrder != NULL
- OrderPrice > 0
- AmountPaid >= OrderPrice

Postconditions:

- Change < AmountPaid
- Change = OrderPrice - AmountPaid

CreditCardMachineContol: Charges the customer's card.



Invariants:

- CurrentPayment == #CurrentPayment

Preconditions:

- chargeCard(PaymentForm) - The payment form must contain the information of a valid card.

Postconditions:

- chargeCard(PaymentForm) - The customer's card is charged and a statement is sent to the bank.

Access Control Matrix

Due to how our subsystems were constructed, an access control matrix for each individual subsystem would consist of only one actor. We therefore thought it was better to have a single matrix that includes every actor and the major system objects that they interact with.

Objects Actors	Report	Account	Order	Delivery
Manager	viewBusinessReport() viewInventoryReport() editInventoryReport()	createEmployeeAccount() deleteEmployeeAccount() logInToAccount()	processOrder() assignOrder() getAssignedOrder() viewOrder()	x
Kitchen Manager	x	logInToAccount()	processOrder() assignOrder() getAssignedOrder() viewOrder()	x
Cook	x	logInToAccount()	viewOrder() getAssignedOrder()	x
Driver	x	logInToAccount()	viewOrder() getAssignedOrder()	assignDelivery()
Customer	x	createAccount() deleteAccount() logInToAccount() resetPassword()	placeOrder() payForOrder() addPizza() removePizza() viewOrder() saveOrdertoFavorites()	x