Krystina Poling

Flower Shop Ordering System

Software Design Document

Last Update: 04/20/2018

Table of Contents

[1 INTRODUCTION 3](#_Toc513210709)

[1.1 Purpose 3](#_Toc513210710)

[1.2 Scope 3](#_Toc513210711)

[1.3 Overview 3](#_Toc513210712)

[2 SYSTEM OVERVIEW 4](#_Toc513210713)

[3 SYSTEM ARCHITECTURE 5](#_Toc513210714)

[3.1 Architectural Design 5](#_Toc513210715)

[3.2 Decomposition Description 5](#_Toc513210716)

[3.3 Exception Handling 5](#_Toc513210717)

[3.4 Design Rationale 6](#_Toc513210718)

[4 DATA DESIGN 6](#_Toc513210719)

[4.1 Data Description 6](#_Toc513210720)

[4.2 Data Dictionary 8](#_Toc513210721)

[5 COMPONENT DESIGN 11](#_Toc513210722)

[6 HUMAN INTERFACE DESIGN 25](#_Toc513210723)

[6.1 Overview of User Interface 25](#_Toc513210724)

[6.2 Screen Images 27](#_Toc513210725)

[6.3 Screen Objects and Actions 28](#_Toc513210726)

[7 REQUIREMENTS MATRIX 29](#_Toc513210727)

# INTRODUCTION

## Purpose

This software design document (SDD) describes the detailed structure of the architecture that will be used for the Flower Shop Ordering System (FSOS) and the precise system design details required to satisfy the stakeholder’s requirements. This document is specifically for use during implementation of the system and the primary audiences for this document are the software developers.

## Scope

The Flower Shop Ordering System described in this document will provide authorized users (Flower Store Employees) the ability to create and track different customer accounts and their orders through a Graphical User Interface (GUI). When a new customer wants to place an order the system will allow the user the ability to create a new customer account, which will prompt for customer data and generate a new account. The system will then allow the user to add orders to the account, one order at a time providing the information about each order: pickup date, size of order, message, etc. Once orders are passed into the system the user will have the ability to edit order information at any time, add another order, or cancel an order from the customer’s account. After all account and customer order data has been inputted into the GUI the user will then have the ability to save the customer’s account information to a file that will be stored on the computer’s hard drive. In addition, the FSOS will also give users the ability to lookup already created customer accounts and the price for each order in the account as calculated by the system. Users will be able to pull up information about total price for all orders placed (since account was opened) and the number of orders placed for an account for the currently loaded account. Lastly, the FSOS will give users the ability to cancel an order to the currently opened account, and delete the currently open account which will also delete the file.

The system is meant to be used by only authorized users, however at the current time user authentication functionality is outside the scope of the system described in this design document.

## Overview

The Software Design Document is divided into 8 major sections with various subsections. The sections of the Software Design Document are:

1. Introduction

* Purpose of this SDD and brief description and scope of the FSOS

1. System Overview

* General description of the context and design of the FSOS

1. System Architecture

* Complete architectural design of the system including class diagram

1. Data Design

* Information about the different data structures to be used, and data storage formats

1. Component Design

* Detailed description of the functioning and responsibilities of each component of the design

1. Human Interface Design

* Complete information about how the user interface of the FSOS will work and how it will look like

1. Requirements Matrix

* Description of the FSOS requirements, including requirements matrix

1. Appendices

# SYSTEM OVERVIEW

This system is broken down into two logical components, each with a specific role:

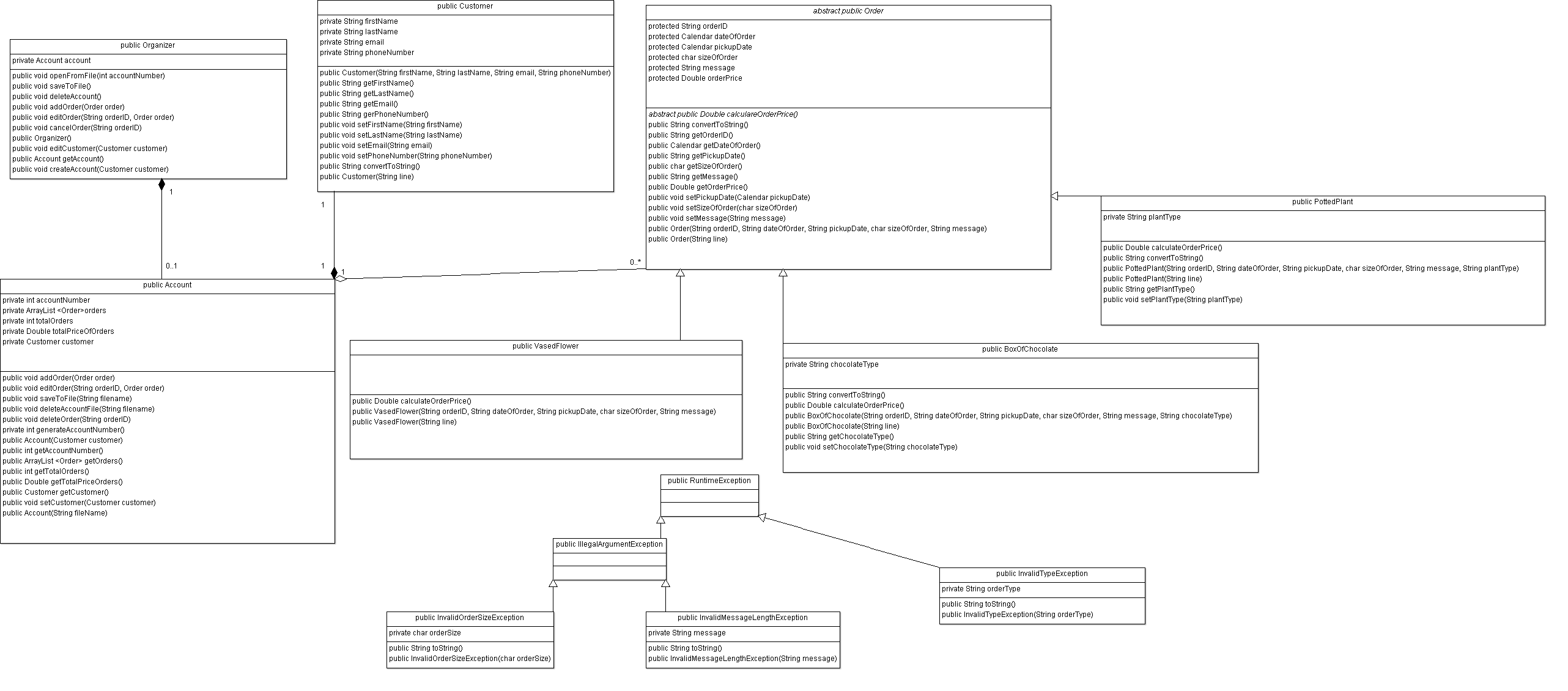
* Graphical User Interface (GUI): The sole interface to the user; it will receive input from the user and provide feedback upon the executions of the core business package.
* Core Business Package: The business logic of the system, the core business package is comprised of several java classes that will be responsible for processing user input and delivering the correct outputs to the GUI.

Technologies Used Include:

* NetBeans IDE
* Java
* XML flags
* Computer Hard Drive
  + The system will need use of the computer’s hard drive in which it is being installed on. The system will be saving fleet information in text files to the computer’s hard drive and will not be dependent on a database.

# SYSTEM ARCHITECTURE

## Architectural Design



## Decomposition Description

The FSOS is composed of seven main classes that essentially work together to provide the business logic necessary to make the system functional and meet all the requirements of the stakeholders. When a user first opens the system the Main class containing the GUI will automatically construct an Organizer object. The Organizer class will then use either its createAccount() method or openFromFile() to construct a new account object. The new account object is kept in Organizer’s account attribute, and when closed or deleted the account attribute gets null. Both Account constructors first create a new Customer object and then use that data to construct a new Account. Once the system has successfully created a new Account object the Account class will then interact with the Order class to begin creating Order objects and populating the Array List called orders in the Account class. The Order class is the parent class to three subclasses which all inherit the attributes and methods contained in the Order class. The three subclasses that inherit from the parent class Order are VasedFlower, BoxOfChocolate, and PottedPlant.

## Exception Handling

The system will introduce three new RunTimeExceptions; one InvalidTypeException, and two more of subclass IllegalArugmentException which will be InvalidOrderSizeException and InvalidMessageLengthException. The following will explain each of the new exceptions being introduced into the system and how and where they will be used:

* InvalidTypeException: When a user tries to create a new order, edit an order, or load an order from a file the system will validate data has a valid order type. The order typed will be checked within both BoxOfChocolate construstors and PottedPlant construstors. Therefore, chocolate type must be dark, milk, or white and plant type needs to be aloe, fern, or cactus. When a Box of Chocolate order or Potted Plant order finds invalid data the new order object will not be created and the InvalidTypeException will be thrown with an appropriate message. The Organizer class will catch the exception in the addOrder, editOrder, or openFromFile methods and write the error message to the console, then re-throw it so that the UI gets the exception and communicates the problem to the user.
* InvalidOrderSizeException: For every new order that is created, edited, or loaded from a file an initial order size value must be given (‘S’, ‘M’, ‘L’). The Order class has two constructors which will both check for invalid order size input. One constructor will check orders loaded from a file and the other constructor will check order sizes that are inputted directly into the UI by the user. When one of the Order constructors find invalid input a new InvalidOrderSizeException will be thrown and a new order object will not be created. The Organizer class will catch the exception in the addOrder, editOrder, or openFromFile methods and write the error message to console, then re-throw it so that the UI gets the exception and communicates the problem to the user.
* InvalidMessageLengthException: When assigning a value for initial order message or changing an existing message value, if it’s greater than 20 characters the Order class will throw an InvalidMessageLengthException from one of its two constructors. The Organizer class will catch the exception in its addOrder, editOrder, or openFromFile methods and re-throw the new InvalidMessageLengthException. However, the error message thrown to the UI must say what the allowed message size is versus how many characters the one the user submitted was.

## Design Rationale

The FSOS architectural design was chosen because the system is to be used on only one work station so it’s consider to be a small scale project. The system is designed without a database because the user’s hard drive will be used to store and load the system’s data. However, the seven classes chosen can be easily scaled up in the future and altered to incorporate a database, if the Flower Shop owner decides they want to be able to use the system on multiple computers later. In addition, while the FSOS is meant to be used by only approved users the system does not implement a user authentication functions but by incorporating an Organizer class it allows the developers to add that functionality in at a later date.

# DATA DESIGN

## Data Description

FSOS will not be implementing a database so the information domain of the system will all be saved to text files in the FSOS’s current directory located on the user’s hard drive. Each text file’s name will be generated from the customer’s account number. For implementing the functionality of saving data to file the FSOS will follow the principle of separation of concerns. That means that one class will not need to know anything about the “guts” of another class.

Saving and loading data from text files will be implemented in the following ways:

1. Saving Account Data to File
   * The saveToFile method in Organizer will call the saveToFile method in Account to create an xml file. The xml file will be created with a filename based on the account’s account number generated by the system. Once the file is created the Account’s saveToFile method will then write out the account level information using the proper xml format then write out the Customer level information using the Customer’s convertToString method. Next, each order’s information will be retrieved from the Account’s order array list and written to the xml file using the convertToString method found in Order and Order’s child classes. The final format will appear as the following:

<account>

<accounNumbert>accountNumber</accountNumber>

<totalOrders>totalOrders</totalOrders>

<totalPriceOfOrders>totalPriceOfOrders</totalPriceOfOrders>

</account>

<customer>

<firstName>firstName</firstName>

<lastName>lastName</lastName>

<email>email</email>

<phoneNumber>phoneNumber</phoneNumber>

</customer>

<VasedFlower>

<orderID>orderID</orderID>

<dateOfOrder>dateOfOrder</dateOfOrder>

<pickupDate>pickupDate</pickupDate>

<sizeOfOrder>sizeOfOrder</sizeOfOrder>

<message>message</message>

<orderPrice>orderPrice</orderPrice>

</VasedFlower>

<BoxOfChocolate>

…….

……..

………

<chocolateType>chocolateType</chocolateType>

</BoxOfChocolate>

<PottedPlant>

……

……

…….

<plantType>plantType</plantType>

</PottedPlant>

1. Loading Account Data from File

* The openFromFile method in Organizer will open an account text file based on the account number parameter passed into the UI by the user. Organizer will then call the Account constructor “Account(String filename)”, the filename parameter will be determined by the account number passed by openFromFile method in Organizer. The Account constructor will then read the first line and set the Account level attributes until xml flag </account> is reached. Next, a new customer object will be created based on the lines found between xml flags <customer> and </customer> using the Customer’s “Customer(String line)” constructor. Lastly, the remaining lines will be read and new Order objects will be created using Order’s “Order(String line)” constructor. After each new order object is created it will be saved to Account’s order array list.

## Data Dictionary

**Object: Account**

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Data Type** | **Purpose** |
| accountNumber | int | Store Account Number |
| orders | Array List | Stores a list of order objects |
| totalOrders | int | Stores total number of orders placed on account |
| totalPriceOfOrders | Double | Stores the total price of all orders placed on account |
| customer | Customer | Stores customer object |
| **Methods** | **Parameters** | **Purpose** |
| Account | Customer customer | Constructor for new account object |
| Account | String fileName | Constructor for new account object |
| addOrder | Order order | Adds an order object to the Account |
| deleteAccountFile | String filename | Deletes Account’s xml file stored on hard drive |
| deleteOrder | String orderID | Deletes order object and the order object from array list orders |
| editOrder | String orderID, Order order | Edits an existing order |
| getAccountNumber |  | Retrieves Account’s account number |
| getCustomer |  | Retrieves Customer information |
| getOrders |  | Retrieves the array list of orders |
| getTotalOrders |  | Retrieves the total number of orders assigned to the account |
| getTotalPriceOrders |  | Retrieves the total price for all orders assigned to the account |
| generateAccountNumber |  | Generates the Account’s account number |
| setCustomer | Customer customer | Sets values for the Account’s customer object |
| **Object: Customer** |  |  |

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Data Type** | **Purpose** |
| email | String | Stores customer’s email address |
| firstName | String | Stores customer’s first name |
| lastName | String | Stores customer’s last name |
| phoneNumber | String | Stores customer’s phone number |
| **Methods** | **Parameters** | **Purpose** |
| convertToString |  | Converts customer attributes to xml format String |
| Customer | String firstName, String lastName, String email, String phoneNumber | Constructor for new customer object |
| Customer | String line | Constructor for new customer object |
| getEmail |  | Retrieves customer’s email address |
| getFirstName |  | Retrieves customer’s first name |
| getLastName |  | Retrieves customer’s last name |
| getPhoneNumber |  | Retrieves customer’s phone number |
| setEmail | String email | Sets customer’s email address |
| setFirstName | String firstName | Sets customer’s first name |
| setLastName | String lastName | Sets customer’s last name |
| setPhoneNumber | String phoneNumber | Sets customer’s phone number |
| **Object: Order** |  |  |

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Data Type** | **Purpose** |
| dateOfOrder | Calendar | Stores date order was originally created |
| message | String | Stores a message to be put on the order (20 character max.) |
| orderID | String | Stores order ID number |
| orderPrice | Double | Stores total amount of order |
| pickupDate | Calendar | Stores date order will be picked up |
| sizeOfOrder | char | Stores size of order (S, M, L) |
| **Methods** | **Parameters** | **Purpose** |
| calculateOrderPrice |  | Abstract method: Sums total price for the order placed |
| convertToString |  | Converts order attributes to xml format String |
| getDateOfOrder |  | Retrieve date order was first placed |
| getMessage |  | Retrieve customers message for order |
| getOrderID |  | Retrieve order’s order ID |
| getOrderPrice |  | Retrieve total price for order |
| getPickupDate |  | Retrieve date order will be picked up |
| getSizeOfOrder |  | Retrieve size of order (S, M, L) |
| setMessage | String message | Sets customer’s message for order |
| setPickupDate | Calendar pickupDate | Sets date order will be picked up |
| setSizeOfOrder | char sizeOfOrder | Sets size of order (S, M, L) |
| **Object: Organizer** |  |  |

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Data Type** | **Purpose** |
| account | Account | Stores Account Object |
| **Methods** | **Parameters** | **Purpose** |
| addOrder | Order order | Initiates the process of adding a new order to Account |
| cancelOrder | String orderID | Initiates the process of deleting an order |
| createAccount | Customer customer | Initiates the process of creating an Account |
| deleteAccount |  | Initiates the process of deleting an account |
| editCustomer | Customer customer | Initiates the process of editing information at the Customer class level |
| editOrder | String orderID, Order order | Initiates the process of editing an order’s information |
| getAccount |  | Retrieves an account object |
| openFromFile | int accountNumber | Loads a file from the computer’s hard drive |
| Organizer |  | Constructor to create a organizer object |
| saveToFile |  | Initiates the process of saving a accounts information to a xml file |
| **Child Object to Order: BoxOfChocolate** | | |

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Data Type** | **Purpose** |
| chocolateType | String | Stores type of chocolate (dark, milk, white) |
| **Methods** | **Parameters** | **Purpose** |
| BoxOfChocolate | String orderID, Calendar dateOfOrder, Calendar pickupDate, char sizeOfOrder, String message, String chocolateType | Constructor to create a new box of chocolate object |
| BoxOfChocolate | String line | Constructor to create a new box of chocolate object |
| calculateOrderPrice |  | Calculates total price for order |
| convertToString |  | Converts box of chocolate attributes to xml format String |
| getChocolateType |  | Retrieves order’s chocolate type |
| setChocolateType | String chocolateType | Sets order’s chocolate type |
| **Child Object to Order: PottedPlant** | | |

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Data Type** | **Purpose** |
| plantType | String | Stores plant type (aloe, fern, cactus) |
| **Methods** | **Parameters** | **Purpose** |
| calculateOrderPrice |  | Calculates total price for order |
| convertToString |  | Converts potted plant attributed to xml format String |
| getPlantType |  | Retrieves order’s plant type |
| PottedPlant | String orderID, Calendar dateOfOrder, Calendar pickupDate,  Char sizeOfOrder, String message, String plantType | Constructor to create a new potted plant object |
| PottedPlant | String line | Constructor to create a new potted plant object |
| setPlantType | String plantType | Sets the order’s plant type (aloe, fern, cactus) |
| **Child Object to Order: VasedFlower** | | |

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Data Type** | **Purpose** |
| \*\* Inherits all attributes from Order |  |  |
| **Methods** | **Parameters** | **Purpose** |
| calculateOrderPrice |  | Calculates total price for order |
| VasedFlower | String orderID, Calendar dateOfOrder, Calendar pickupDate, char sizeOfOrder, String message | Constructor to create a new vased flower object |
| VasedFlower | String line | Constructor to create a new vased flower object |

# COMPONENT DESIGN

**Class Name:** Organizer

**Class Description/Purpose:** The Organizer class is the highest-level class of the system. It contains the complete interface for the User Interface to interact with the business logic of the system.

**Class Modifiers:** Public

**Class Inheritance:** N/A

**Class Attributes:**

private Account account - Stores Account Object

**Exceptions Thrown:** InvalidOrderSizeException, InvalidMessageLengthException, InvalidTypeException

**Class Methods:**

Constructors: There is only one way to create a new Organizer object. As soon as the system is opened the Main class containing the UI will automatically create a new Organizer object because the UI will need that class in order to reach the business logic of the system and process any user input.

public Organizer()

* Construct a organizer object

Methods:

public createAccount(Customer customer) //Initiates the process of creating an Account

Constructs a new Account object for attribute account using Account(Customer customer)

public openFromFile(int accountNumber) //Loads a file from the computer’s hard drive

Constructs a new Account object for attribute account using Account(String filename)

Catch and re-throw InvalidOrderSizeException, InvalidMessageLengthException, or InvalidTypeException to UI

public deleteAccount() //Deletes an account file

Call Account’s deleteAccountFile(String filename)

Set account attribute to null

public editCustomer(Customer customer) //edits information at the Customer class level

Call Customer set methods

public getAccount() //Retrieves an account object

Retrieve Account object

public addOrder(Order order) //Initiates the process of adding a new order to Account

Call Account’s addOrder(Order order)

Catch and re-throw InvalidOrderSizeException, InvalidMessageLengthException, or InvalidTypeException to UI

public cancelOrder(String orderID) //Initiates the process of deleting an order

Call Account’s deleteOrder(String orderID)

public editOrder(String orderID, Order order) //Initiates the process of editing an order’s information

Call Account’s editOrder(String orderID, Order order)

public saveToFile() //Initiates the process of saving a accounts information to a xml file

Call Account’s saveToFile(String filename)

**Class Name:** Account

**Class Description/Purpose:** Account class is used to add, edit, or delete orders from an account. It is also responsible for saving, deleting, and updating account xml files saved to the user’s hard drive.

**Class Modifiers:** Public

**Class Inheritance:** N/A

**Class Attributes:**

private int accountNumber- Store Account Number

private ArrayList<Order>orders- Stores a list of order objects

private int totalOrder- Stores total number of orders placed on account

private Double totalPriceOfOrders- Stores the total price of all orders placed on account

private Customer customer- Stores customer object

**Exceptions Thrown:** N/A

**Class Methods:**

Constructors: there are two ways to create a new Account object. If the user selects to create a new Account, then constructor Account(Customer customer) is called. If the user opens a account from a file, then constructor Account(String fileName) is called.

public Account(Customer customer) //Constructor for new account object

//Constructs a new Customer object for attribute customer using Customer’s getter methods

this.customer = new Customer(customer.getFirstName(), customer.getLasrName(), customer.getEmail(), customer.getPhoneNumber())

Call method generateAccountNumber

public Account(String filename) //Constructor for new account object from file

Read the first line of the file filename and load the Account into the associated attributes

Check for substring <accountNumber> and assign value to accountNumber attribute

Check for substring <totalOrders> and assign value to totalOrders attribute

Check for substring <totalPriceOfOrders> and assign value to totalPriceOfOrders attribute

Loop through all of the other lines of the file

For each line check the beginning

If the beginning is <Customer>

Create a new Customer object, pass in the current line from the file

load the Customer into the associated attributes

continue Loop

If the beginning is <PottedPlant>

Create a new PottedPlant object, pass in the current line from the file

load the PottedPlant into the associated attributes

add orders to orders array

continue Loop

If the beginning is <VasedFlower>

Create a new VasedFlower object, pass in the current line from the file

load the VasedFlower into the associated attributes

add orders to orders array

continue Loop

If the beginning is <BoxOfChocolates>

Create a new BoxOfChocolates object, pass in the current line from the file

load the BoxOfChocolates into the associated attributes

Add order to orders array

continue Loop

Exit Loop when no more orders exist

Methods:

public addOrder(Order order) /\*Adds an order object to the Account (see class diagram for parameters and order) \*/

Construct new Order object

Increase totalOrders attribute by 1

Add order price to totalPriceOfOrders attribute

public deleteAccountFile(String filename) //Deletes Account’s xml file stored on hard drive

Check if File exists

If file exists delete

public deleteOrder(String orderID) /\*Deletes order object and the order object from array list orders \*/

Loop through orders array list to find Orders object with matching parameter order ID

Subtract Orders orderPrice from Account’s totalPriceOfOrders attribute

Delete Order object

Subtract 1 from totalOrders attribute

public editOrder(String orderID, Order order) //Edits an existing order

Loop through orders array to find Orders object with matching parameter orderID

set new order attributes with Order setter methods

Exit Loop

// calculate new sum of order

Loop through orders array and recalculate each orders price using the Order’s caluculatePrice method

// calculate new sum of all orders belonging to account

Sum totalPriceOfOrders

public getAccountNumber() //Retrieves Account’s account number

get and return accountNumber

public getCustomer() //Retrieves Customer information

get and return customer object

public ArrayList<Order> getOrders() //Retrieves the array list of orders

get and return order ArrayList

public getTotalOrders() //Retrieves the total number of orders assigned to the account

get and return totaoOrders

public getTotalPriceOrders() //Retrieves the total price for all orders assigned to the account

get and return totalPriceOfOrders

private generateAccountNumber() //Generates the Account’s account number

get current time in milliseconds

return accountNumber

public setCustomer(Customer customer) //Sets values for the Account’s customer object

Take the parameter customer and assign it to customer attribute

**Class Name:** Customer

**Class Description/Purpose:** Every account must have a customer assigned to it. The Customer class keeps track of each customer’s name and contact information

**Class Modifiers:** Public

**Class Inheritance:** none

**Class Attributes:**

private String email- Stores customer’s email address

private String firstName- Stores customer’s first name

private String lastName- String Stores customer’s last name

private String phoneNumber- Stores customer’s phone number

**Exceptions Thrown:** N/A

**Class Methods:**

Constructors: there are two ways to create a new Customer object. If the user selects to create a new Account, then constructor Customer(String firstName, String lastName, String email, String phoneNumber) is called. If the user opens an account from a file, then constructor Customer(String line) is called.

//Constructor for new customer object

public Customer(String firstName, String lastName, String email, String phoneNumber)

// Customer variables

this.firstName = firstName;

this.lastName = lastName;

this.email = email;

this.phoneNumber = phoneNumber;

public Customer(String line) //Constructor for new customer object

Check for substring <firstName> and assign value to firstName attribute

Check for substring <lastName> and assign value to lastName attribute

Check for substring <email> and assign value to email attribute

Check for substring <phoneNumber> and assign value to phoneNumber attribute

Methods:

public convertToString() //Converts customer attributes to xml format String

return a string with tags and values for each attribute that has the following values and format

<customer>

<firstName>firstName</firstName>

<lastName>lastName</lastName>

<email>email</email>

<phoneNumber>phoneNumber</phoneNumber>

</customer>

public getEmail() //Retrieves customer’s email address

get and return emailAddress

public getFirstName() //Retrieves customer’s first name

get and return firstName

public getLastName() //Retrieves customer’s last name

get and return lastName

public getPhoneNumber() //Retrieves customer’s phone number

get and return phoneNumber

public setEmail(String email) //Sets customer’s email address

Take the parameter email and assign it to email attribute

public setFirstName(String firstName) //Sets customer’s first name

Take the parameter firstName and assign it to firstName attribute

public setLastName(String lastName) //Sets customer’s last name

Take the parameter lastName and assign it to lastName attribute

public setPhoneNumber(String phoneNumber) //Sets customer’s phone number

Take the parameter phoneNumber and assign it to phoneNumber attribute

**Class Name:** Order

**Class Description/Purpose:** Parent class used to allow VasedFlower, PottedPlant, and BoxOfChocolates to inherit all its attributes and methods. The Order class is used to track order information

**Class Modifiers:** abstract public

**Class Inheritance:** Parent to VasedFlower, PottedPlant, and BoxOfChocolates

**Class Attributes:**

private Calendar dateOfOrder- Stores date order was originally created

private String message- Stores a message to be put on the order (20 character max.)

private String orderID- Stores order ID number

private Double orderPrice- Stores total amount of order

private Calendar pickupDate- Stores date order will be picked up

private char sizeOfOrder- Stores size of order (S, M, L)

**Exceptions Thrown:** InvalidOrderSizeException, InvalidMessageLengthException, InvalidTypeException

**Class Methods:**

Constructors: Order is a parent class and will have two constructors that will be called by its three children classes. The first constructor Order(String orderID, String orderDate, String deadlineDate, char sizeOfOrder, String message) will be called when an order is entered manually into the UI and the second construstor Order(String line) will be called when loading an order from a file.

// Constructor for new order object

Order(String orderID, String orderDate, String deadlineDate, char sizeOfOrder, String message)

Initialize SimpleDate Formatter for dateOfOrder and pickupDate (“MM-dd-yyyy”)

//assign order attributes

this.orderID = orderID;

// convert date Strings into Calendar objects using the SimpleDate formatter

dateOfOrder.setTime(sdf.parse(orderDate));

pickupDate.setTime(sdf.parse(deadlineDate));

this.sizeOfOrder = sizeOfOrder;

// check size of order char

IF sizeOfOrder does not equal ‘S’, ‘M’, or ‘L’{

throw new InvalidOrderSizeException(sizeOfOrder);

}

this.message = message;

// check message length

int messageLength = message.length();

IF messageLength > 20){

throw new InvalidMessageLengthException(message);

}

public Order(String line) //Constructor to create a new order object

Initialize SimpleDate Formatter for dateOfOrder and pickupDate (“MM-dd-yyyy”)

Check for substring <orderID> and assign value to orderID attribute

Check for substring <dateOfOrder> and assign value to dateOfOrder attribute

Check for substring <pickupDate> and assign value to pickupDate attribute

Check for substring <sizeOfOrder>

IF sizeOfOrder does not equal S, M, L

throw InvalidOrderSizeException

ELSE assign value to sizeOfOrder attribute

Check for substring <messagee>

IF message length is greater than 20 char

throw InvalidMessageLengthException

ELSE assign value to message attribute

Check for substring <orderPrice> and assign value to orderPrice attribute

Methods:

abstract public calculateOrderPrice() //Abstract method: Sums total price for the order placed.

abstract method no code

public convertToString() //Converts order attributes to xml format String

// Initalize String to hold class name

String className = this.getClass().getSimpleName();

Initialize SimpleDateFormatter for dateOfOrder and pickUpDate (“MM-dd-yyyy”)

return a string with tags and values for each attribute that has the following values and format

<className>

<orderID>orderID</orderID>

<dateOfOrder>dateOfOrder</dateOfOrder>

<pickupDate>pickupDate</pickupDate>

<sizeOfOrder>sizeOfOrder</sizeOfOrder>

<message>message</message>

<orderPrice>orderPrice</orderPrice>

</className>

public getDateOfOrder() //Retrieve date order was first placed

get and return dateOfOrder

public getMessage() //Retrieve customers message for order

get and return message

public getOrderID() //Retrieve order’s order ID

get and return orderID

public getOrderPrice() //Retrieve total price for order

get and return orderPrice

public getPickupDate() //Retrieve date order will be picked up

get and return pickupDate

public getSizeOfOrder() //Retrieve size of order (S, M, L)

get and return sizeOfOrder

public setMessage(String message) //Sets customer’s message for order

IF message length is greater than 20 char

throw InvalidMessageLengthException

ELSE assign value to message attribute

public setPickupDate(Calendar pickupDate) //Sets date order will be picked up

Take the parameter pickupDate and assign it to attribute pickupDate

public setSizeOfOrder(char sizeOfOrder) //Sets size of order (S, M, L)

IF sizeOfOrder does not equal S, M, L

throw InvalidOrderSizeException

ELSE assign value to sizeOfOrder attribute

**Class Name:** VasedFlower

**Class Description/Purpose:** Used to set and get attributes for VasedFlower objects that are added to Account. In addition, it puts VasedFlower data in the correct format using xml flags so it can be saved to a text file.

**Class Modifiers:** public

**Class Inheritance:** Child of Order

**Class Attributes: Inherits all attributes from Order**

**Exceptions Thrown:** InvalidOrderSizeException, InvalidMessageLengthException

**Class Methods:**

Constructors: there are two ways to create a new VasedFlower object. If the user selects to create a new Account, and place a new vased flower order then constructor VasedFlower(String orderID, String dateOfOrder, String pickupDate, char sizeOfOrder, String message) is called. If the user opens an account from a file, then constructor VasedFlower(String line) is called.

//Constructor to create a new vased flower object

public VasedFlower(String orderID, String dateOfOrder, String pickupDate, char sizeOfOrder, String message)

//call parent’s constructor using super method

super(orderID, orderDate, deadlineDate, sizeOfOrder, message)

Call calculateOrderPrice() method to calculate and set value for orderPrice attribute

public VasedFlower(String line) // Constructor to create a new vased flower object

call parent’s constructor using super method- super(line)

Methods:

public calculateOrderPrice() Calculates total price for order

Initialize size variables Double small = 15.99, Double medium = 25.99, large = 35.99

Initialze addFee = Double 2.99

Find difference between pickupDate and orderOfDate

IF pickupDate >= 2

return size of order price + 2.99

ELSE return size of order price

**Class Name:** BoxOfChocolate

**Class Description/Purpose:** Used to set and get attributes for BoxOfChocolate objects that are added to Account. In addition, it puts BoxOfChocolate data in the correct format using xml flags so it can be saved to a text file.

**Class Modifiers:** public

**Class Inheritance:** Child of Order

**Class Attributes:**

private String chocolateType- Stores type of chocolate (dark, milk, or white)

**Exceptions Thrown:** InvalidOrderSizeException, InvalidMessageLengthException, InvalidTypeException

**Class Methods:**

Constructors: there are two ways to create a new BoxOfChocolate object. If the user selects to create a new Account, and place a new box of chocolate order then constructor BoxOfChocolate(String orderID, Calendar dateOfOrder, Calendar pickupDate, char sizeOfOrder, String message, String chocolateType) is called. If the user opens an account from a file, then constructor BoxOfChocolate(String line) is called.

//Constructor to create a new box of chocolate object

public BoxOfChocolate (String orderID, String dateOfOrder, String pickupDate, char sizeOfOrder, String message, String chocolateType)

//call parent’s constructor using super method

super(orderID, orderDate, deadlineDate, sizeOfOrder, message)

// check order type

IF chocolate type does not equal “milk”, “dark”, or “white”

throw InvalidTypeException

ELSE assign chocolateType parameter to chocolateType attribute

Call calculateOrderPrice() method to calculate and set value for orderPrice attribute

public BoxOfChocolate(String line) //Constructor to create a new box of chocolate object

call parent’s constructor using super method- super(line)

Check for substring <chocolateType>

IF chocolateType does not equal dark, milk, white

Throw InvalidTypeException

ELSE assign value to chocolateType attribute

Methods:

public convertToString() //Converts box of chocolate attributes to xml format String

return a string with tags and values for each attribute that has the following values and format

return parents convertToString method and add the type value to the string

public getChocolateType() //Retrieves order’s chocolate type

get and return chocolateType

public setChocolateType(String chocolateType) //Sets order’s chocolate type

IF chocolateType does not equal dark, milk, white

Throw InvalidTypeException

ELSE assign value to chocolate attribute

public calculateOrderPrice() //Calculates total price for order

Initialize size of the box price Double medium = 5.99 large = 10.99

Initialize type of chocolate price Double milk = 10, Double dark = 15,

Double white = 20

IF sizeOfOrder = S

return orderPrice = type of chocolate price

ELSE

return orderPrice = type of chocolate price + size of the box price

**Class Name:** PottedPlant

**Class Description/Purpose:** Used to set and get attributes for PottedPlant objects that are added to Account. In addition, it puts PottedPlant data in the correct format using xml flags so it can be saved to a text file.

**Class Modifiers:** public

**Class Inheritance:** Child of Order

**Class Attributes:**

private String plantType- Stores type of plant (aloe, fern, cactus)

**Exceptions Thrown:** InvalidOrderSizeException, InvalidMessageLengthException, InvalidTypeException

**Class Methods:**

Constructors: there are two ways to create a new PottedPlant object. If the user selects to create a new Account, and place a new potted plant order then constructor PottedPlant(String orderID, String dateOfOrder, String pickupDate, char sizeOfOrder, String message, String plantType) is called. If the user opens an account from a file, then constructor PottedPlant(String line) is called.

//Constructor to create a new potted plant object

public PottedPlant(String orderID, String dateOfOrder, String pickupDate, char sizeOfOrder, String message, String plantType)

//call parent’s constructor using super method

super(orderID, orderDate, deadlineDate, sizeOfOrder, message)

// check order type

IF plantType does not aloe,fern, or cactus

Throw InvalidTypeException

ELSE assign plantType parameter to plantType attribute

Call calculateOrderPrice() method to calculate and set value for orderPrice attribute

public PottedPlant(String line) //Constructor to create a new potted plant object

call parent’s constructor using super method- super(line)

Check for substring <plantType>

IF plantType does not equal fern, aloe, or cactus

Throw InvalidTypeException

ELSE assign value to message attribute

Methods:

public convertToString() //Converts potted plant attributed to xml format String

return parents convertToString method and add the type value to the string

public getPlantType() //Retrieves order’s plant type

get and return plantType

public setPlantType(String plantType) //Sets the order’s plant type (aloe, fern, cactus)

IF plantType does not equal fern, aloe, or cactus

Throw InvalidTypeException

ELSE assign value to plantType attribute

public calculateOrderPrice() //Calculates total price for order

if sizeOfOrder = S

return orderPrice = 19.99

if sizeOfOrder = M

return orderPrice = 29.99

else

return orderPrice = 39.99

**Class Name:** InvalidOrderSizeException

**Class Description/Purpose:** Ensures the user only input the correct data for the size of order they want. User can only enter S, M, or L and if something else is entered the order object will not be constructed or edited. The UI will also be sent a message letting the user know incorrect size input was entered.

**Class Modifiers:** public

**Class Inheritance:** child of IllegalArugment Exception (RunTimeException)

**Class Attributes:**

Private char orderSize- Stores the users input

**Exceptions Thrown:**

**Class Methods:**

Constructor: There is only one way to construct a new InvalidOrderSizeException and that is by the user or the file being opened trying to pass incorrect input for order size into the system.

public InvalidOrderSizeException(char orderSize) // Constructor for exception

incorrect orderSize parameter = orderSize attribute

Methods:

public String toString() // Returns error message to UI

Return “ The “ + orderSize + “is incorrect. Please enter S, M, or L for order size”

**Class Name:** InvalidMessageLengthException

**Class Description/Purpose:** Ensures the user only inputs the correct number of characters for the message they want to put with their order, message can be no longer than 20 char. When more than 20 characters are entered the order object will not be constructed or edited. The UI will also be sent a message letting the user know they are not allowed to input more than 20 characters.

**Class Modifiers:** public

**Class Inheritance:** child of IllegalArugmentException (RunTimeException)

**Class Attributes:**

private String message

**Exceptions Thrown:**

**Class Methods:**

Constructor: There is only one way to construct a new InvalidMessageLengthException and that is by the user or the file being opened trying to pass input for message that is longer then 20 characters.

//Constructor for exception

public InvalidMessageLengthException(char orderSize)

incorrect message parameter = message attribute

Methods:

public String toString() // Returns error message to UI

Return “Message: “+ message + “ is too long. Max character length is 20.”

**Class Name:** InvalidTypeException

**Class Description/Purpose:** Ensures that when the user is attempting to place a plant or chocolate order they input the correct data. When a user enters anything other than fern, aloe, or cactus for a plant order they will get an error. If the user enters anything other than milk, dark, or white for a chocolate order they will get an error. Invalid input will result in a new order object not be created and the UI will be sent an error message letting the user know they entered an incorrect order type.

**Class Modifiers:** public

**Class Inheritance:** child of RunTimeException

**Class Attributes:**

Private String orderType- Stores user input

**Exceptions Thrown:**

**Class Methods:**

Constructor: There is only one way to construct a new InvalidTypeException and that is by the user or the file being opened trying to pass an incorrect order type input when attempting to place a box of chocolates or potted plant order.

//Constructor for exception

public InvalidTypeException(String orderType)

incorrect orderType parameter = orderType attribute

Methods:

public String toString() // Returns error message to UI

Return “Wrong Order Type: “ + orderType + “Please enter correct order type.”

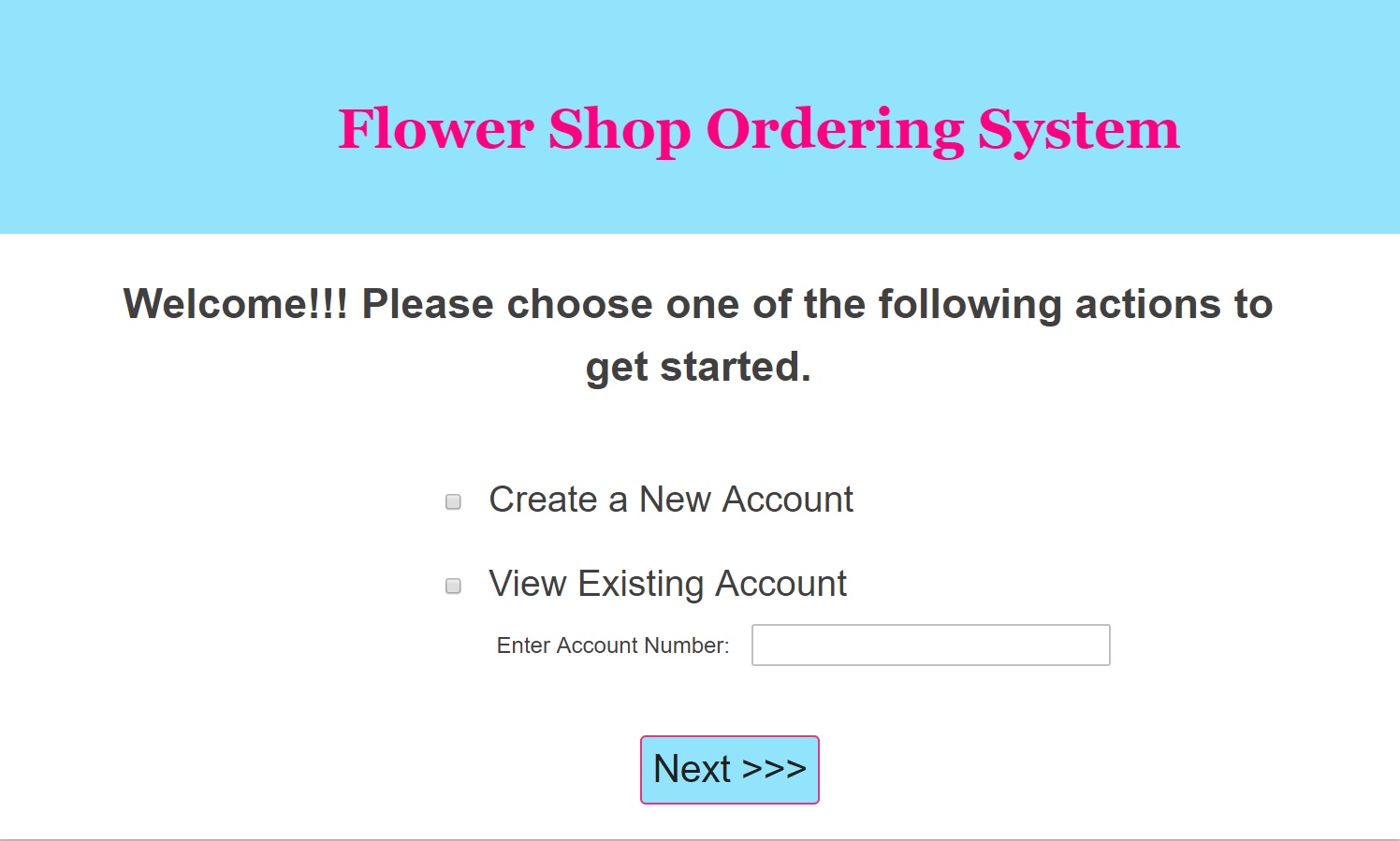
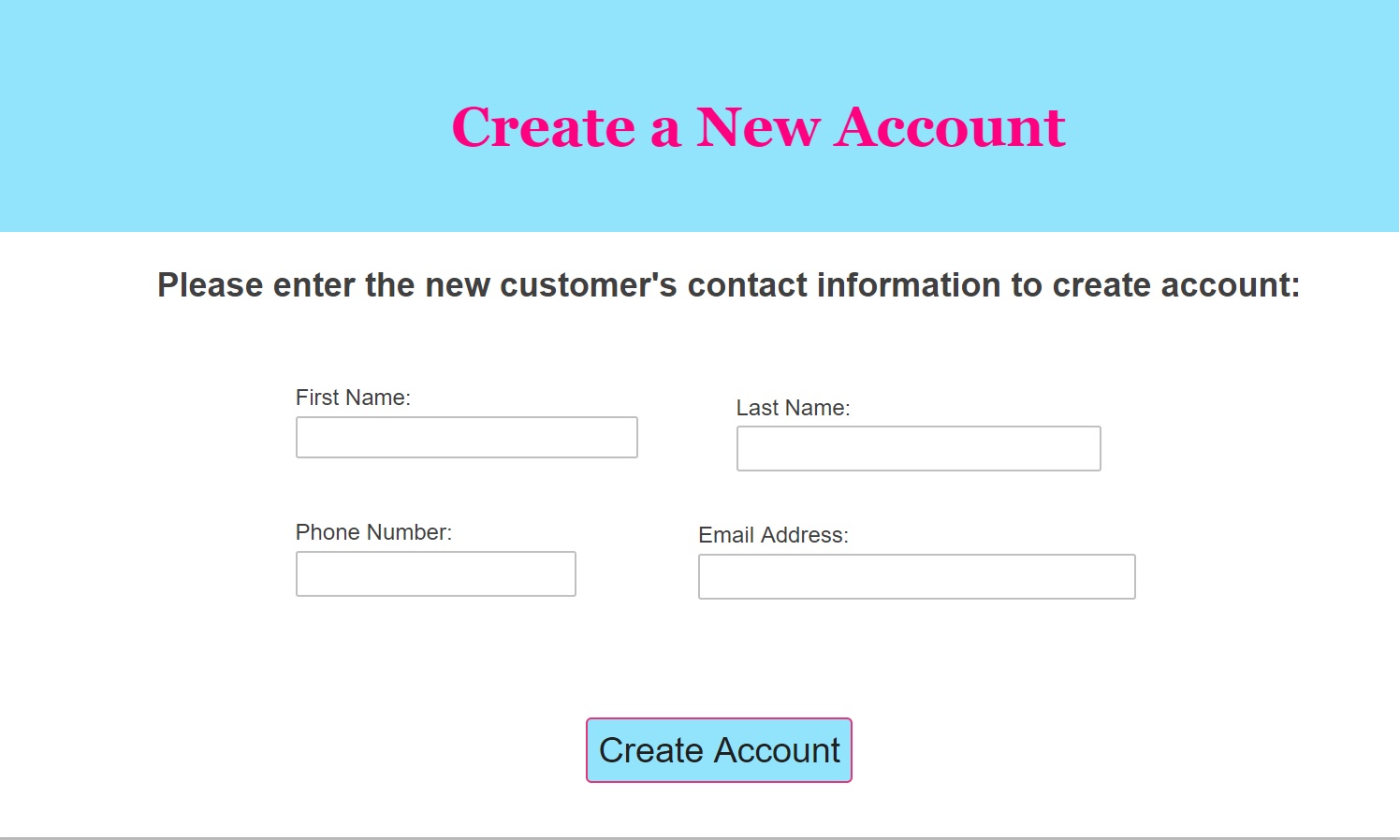
# HUMAN INTERFACE DESIGN

## Overview of User Interface

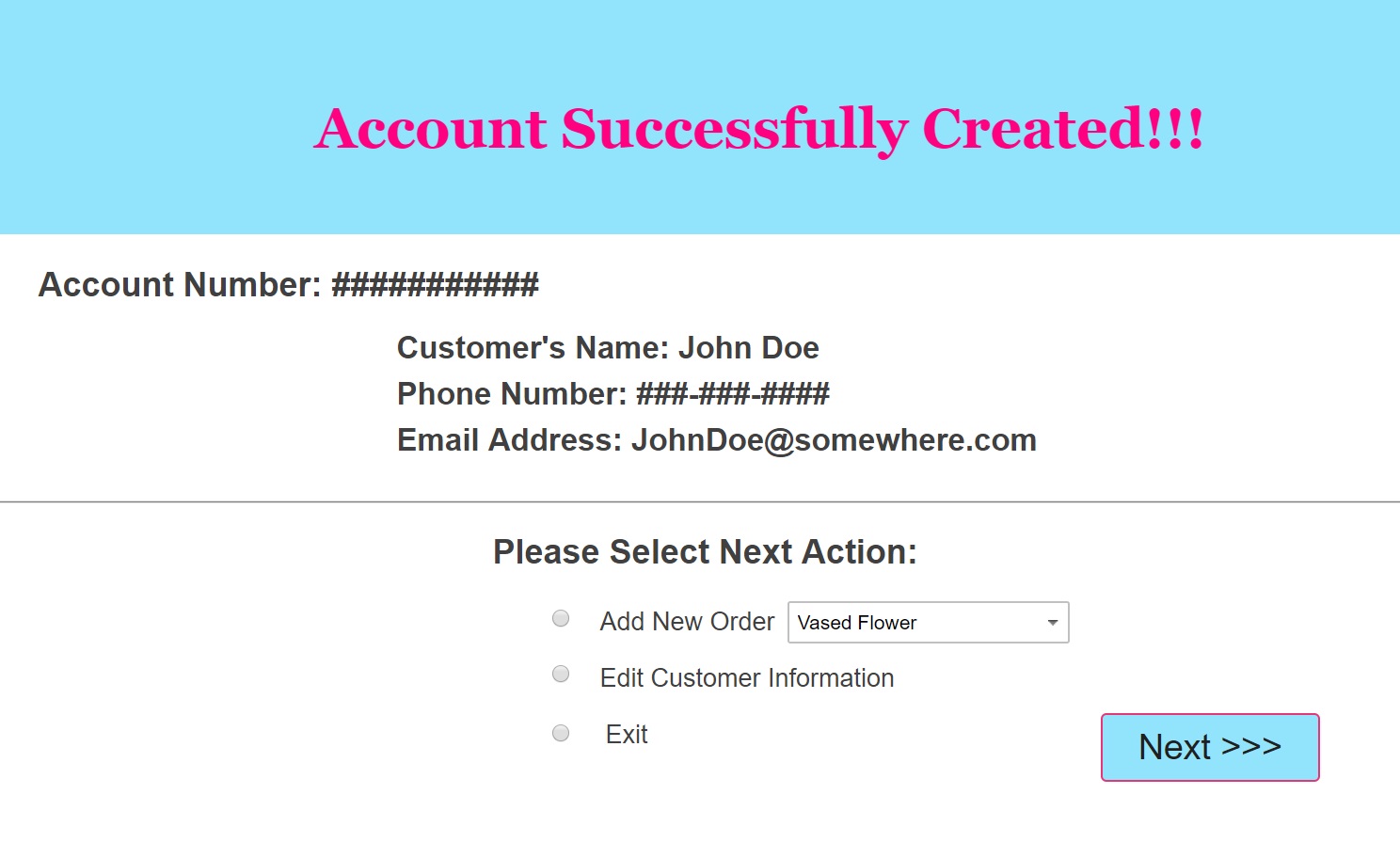
When the FSOS is opened the main screen will appear that will allow the user to either create a new user account or view an existing user account. Once the user makes their choice the UI will then send the user down one of two paths:

1. Create New Account: When user chooses to create a new account they will then be directed to the screen that asks the user for customers contact information. After the contact information has been inputted into the system the user will be forwarded to a confirmation page that lets the user know the new customer’s account has been created. From the account confirmation page the user can choose to add a new order to the new account, edit the customers contact information, or exit the system.
   1. New Order: Depending on the type of order the customer chooses, from the account confirmation page, determines which order form the UI will display (vased flower, box of chocolate or potted plant). The user will then fill out each field of the order form screen and once completed successfully the UI will display the order confirmation page. The order confirmation page will display the entire customer’s contact information and the information for the order just placed. Lastly, the user will be able to decide whether to: add another order, edit customer information, return to main page, or exit system.
   2. Edit Customer: UI will display the customer edit page where they will be able to edit the customers contact information. Once completed they will be directed to the view account page that will display all data associated with the customer’s account. From the view account page the user can: add a new order, edit customer, edit order, return to main page, or exit system.
2. View Customer Account: When the user wants to view an existing account they will first need to enter the existing account’s account number into the main page. After the account number has been entered the UI will display the view account page. The account page will output all the customer’s contact information and show all order data associated with the account in a scrollable table. The account page also gives the user the following choices: add a new order, edit customer, edit order, delete account, delete order, return to main page, save account to file, or exit system.
   1. New Order: UI takes user down same path as New Order path (discussed above).
   2. Edit Customer: UI takes user down same path as Edit Customer path (discussed above).
   3. Edit Order: UI will display the order edit page which will output all information associated with the order number (inputted on the view customer account page). The UI will then display the order information in a new order form that the customer can edit and resubmit. Once order has been edited the user will be directed to the order confirmation page.
   4. Delete Account: Deletes opened account from system and computer hard drive. UI will send user back to main page.
   5. Delete Order: The chosen order will disappear from the orders table and the user will remain on the same page. Until another action is chosen.
   6. Save Account to File: Saves all the users account information to a file on the computer’s hard drive and then takes the user back to the main page.

## Screen Images

Main Page Create New Account Page  

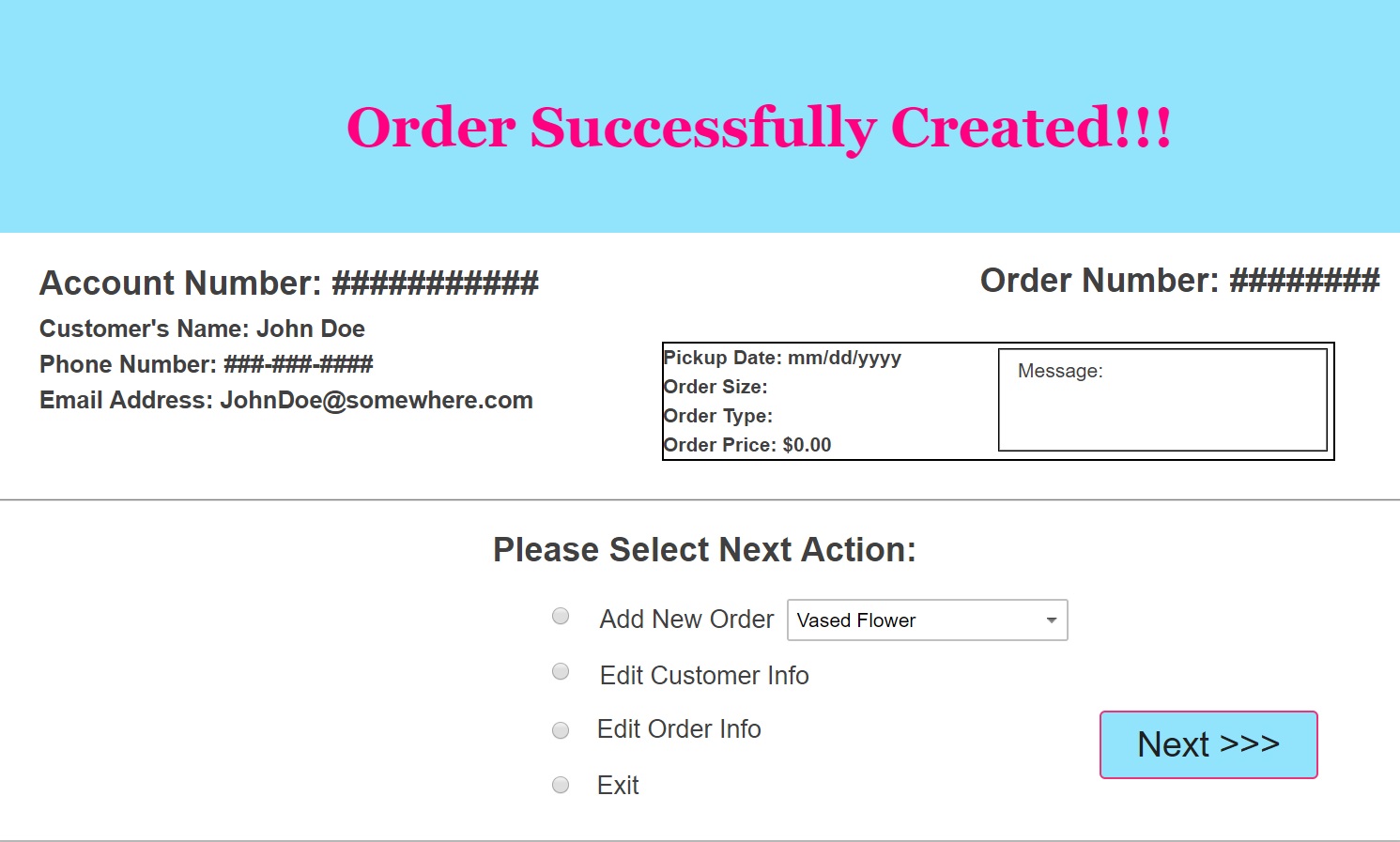
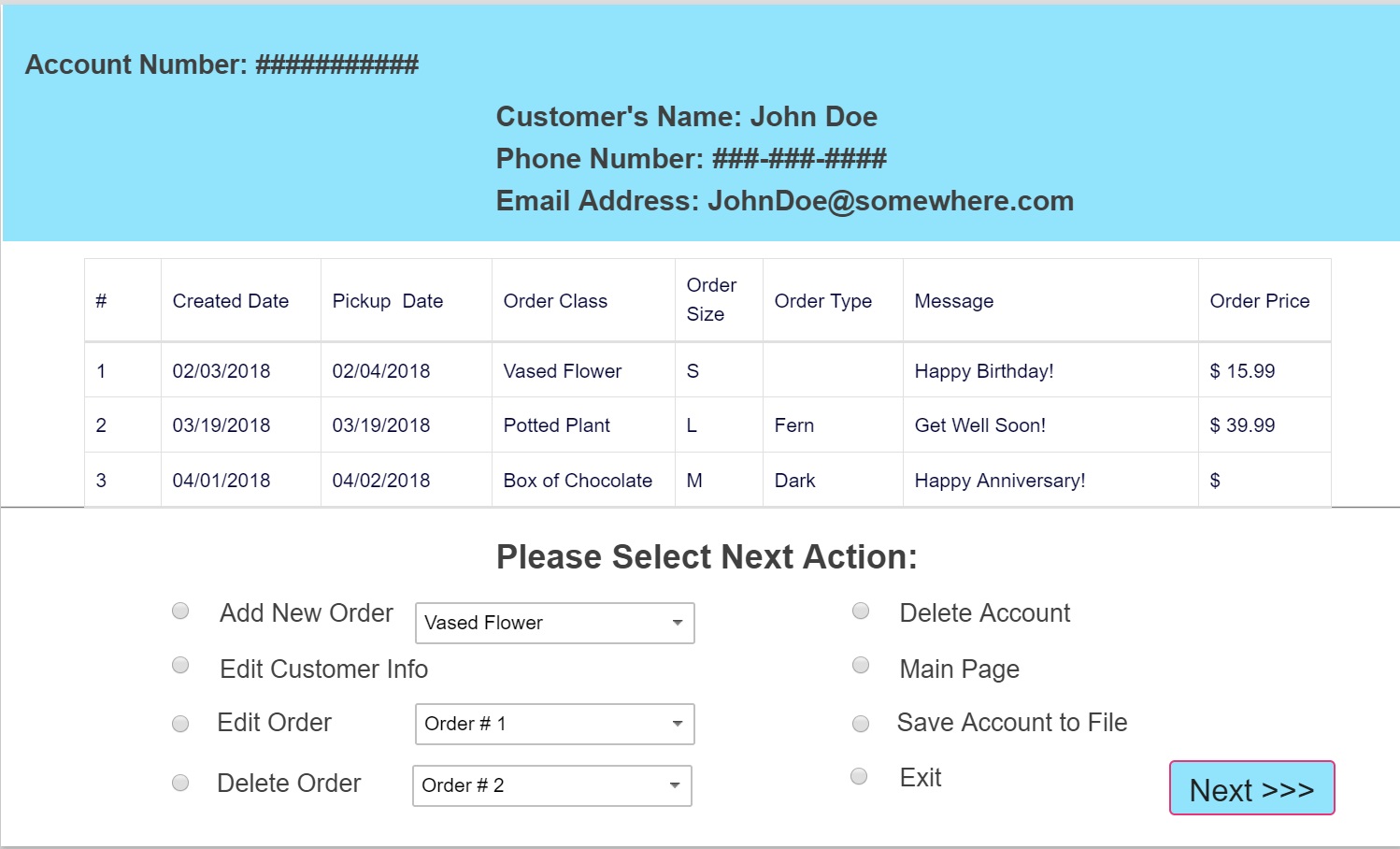
Account Confirmation Page Vased Flower Order Page

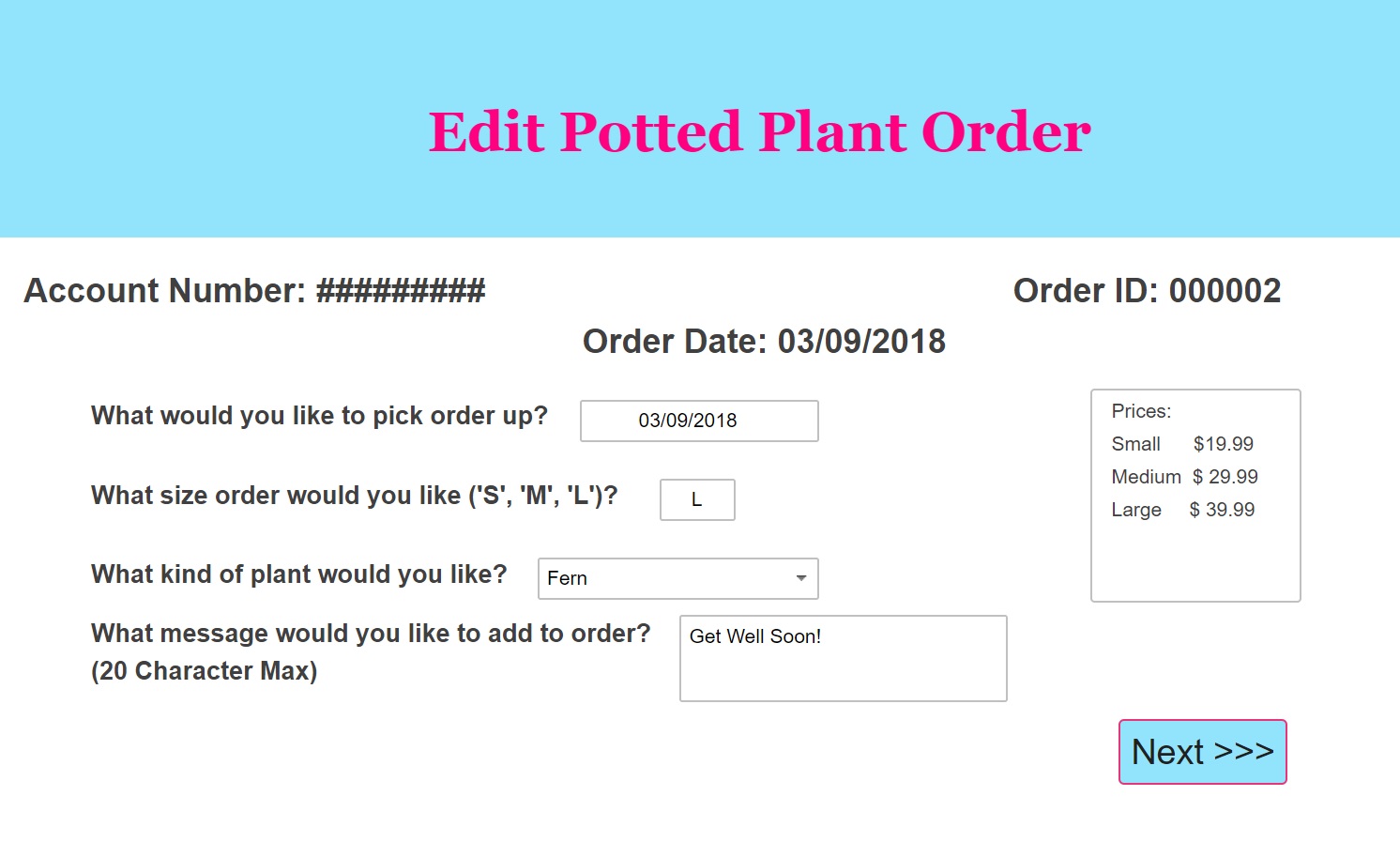
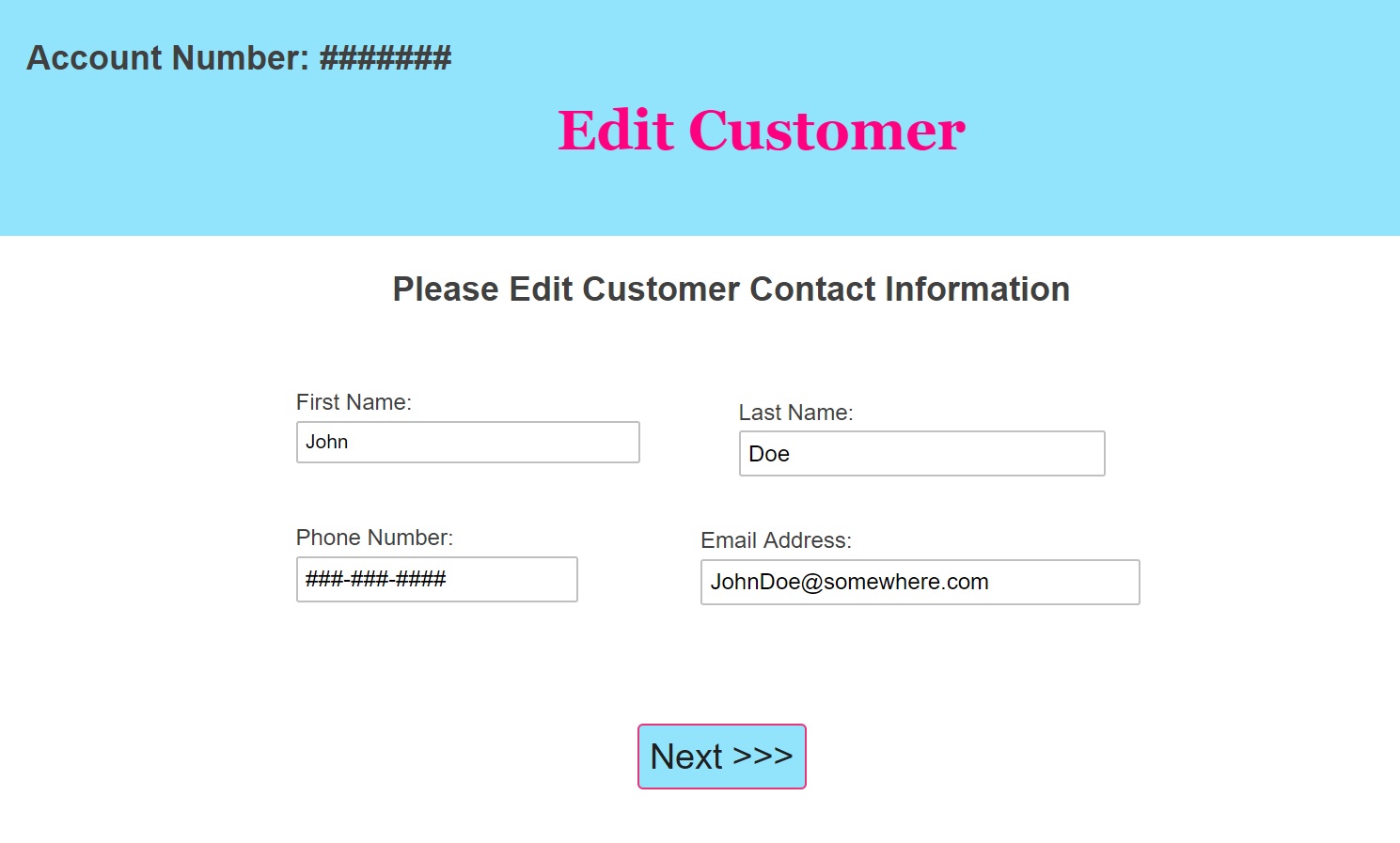
Box of Chocolate Order Page Potted Plant Order Page

Order Confirmation Page View Existing Account Page

Edit Order Page Edit Customer Page

## Screen Objects and Actions

Main Page: Contains two radio buttons and one text box. User will need to choose one of the radio buttons corresponding to the action they wish to take. If the user wants to view an existing account then the account number text box will also need to be filled out. The “Next” button at the bottom of the screen allows the user to move to the next screen corresponding to the action they choose.

New Account: Contains four text boxes that gather customer contact input from the user. The “Next” button at the bottom of the screen will not work until all text boxes have been completed.

Account Confirmation: Displays a confirmation message at the top of the screen letting the user know that the customer’s account was successfully created. The screen then displays the new customer’s contact information and account number. At the bottom of the screen are three radio buttons which allows the user to choose their next action. The “Next” button the bottom of the screen allows the user to move onto their next action.

Order Pages: There are three order pages (Vased Flower, Box of Chocolate, and Potted Plant) which all contain a order form the user must complete and a list of order prices. Once the order form is complete the “Next” button allows the user to move to the next screen.

Order Confirmation: Displays order confirmation message at top of screen, along with order details and customer contact information. At the bottom of the screen there are four radio buttons that allow the user to choose their next action. The “Next” button allows the user to move to the next screen.

View Existing Order: Displays customer contact information and account number at the top of the screen. Then all orders placed on the account are displayed in a scrollable table for the user to review. At the bottom of the page there are seven radio buttons that allow the user to choose their next action. The “Next” button allows the user to move to the next screen.

Edit Pages: Both edit pages (Customer and Order) appear just as the original forms but the text boxes are already completed. User can edit what information needs to be edited and leave the rest alone. Once editing is completed the “Next” button can be clicked by the user in order to move to the next screen.

# REQUIREMENTS MATRIX

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Req ID | Requirement Description | Class | Method | Attribute |
| 1 | Create Account | Organizer | createAccount | account |
| 2 | Create Order | Organizer  Account | addOrder  addOrder | orders |
| 3 | Create Customer | Organizer | createAccount | customer |
| 4 | Edit Order | Organizer  Account | editOrder  editOrder | orderID, order |
| 5 | Edit Customer | Organizer  Account | editCustomer  setCustomer | customer  customer |
| 6 | Load Account from File | Organizer | openFromFile | accountNumber |
| 7 | Save Account to File | Organizer  Account | saveToFile  saveToFile |  |
| 8 | Delete Order | Organizer  Account | cancelOrder  deleteOrder | orderID  orderID |
| 9 | Delete Account | Organizer  Account | deleteAccount  deleteAccountFile |  |
| 10 | Generate Account Number | Account | generateAccountNumber | accountNumber |
| 11 | Calculate Total Order Price | Order | calculateOrderPrice | orderPrice |
| 12 | Track Total Orders | Account | addOrder  deleteOrder | totalOrders |
| 13 | Track Total Price of all Orders | Account | addOrder  deleteOrder | totalPriceOfOrders |
| 14 | Throw Wrong Order Size Exception | InvalidOrderSizeException | toString | orderSize |
| 15 | Throw Wrong Message Size Exception | InvalidMessageLengthException | toString | message |
| 16 | Throw Wrong Order Type Exception | InvalidTypeException | toString | orderType |