Abstract

As this assignment is an iteration upon our last hand-in and I will therefore *not* include diagrams or considerations from the last iteration unless they have been updated in a meaningful way. In the following abstract I will, in broad strokes, describe what has changed.

The overall 10 user stories have not changed, as they were beyond the scope of our current level. The acceptance criteria will therefore be applied to the 9 "sequenced" user stories, from UML I, seeing as they were implementable and more in alignment with the assignment.

The Domain and Class diagram has been updated and is included in this document. A sequence diagram describing what happens when SearchPizza() method is invoked have been added as well.

The functionality of *PizzaMenu* itself was close to fulfilling the implementation criteria for this assignment but has been changed from using *List<T>* to *Dictionary<K, T>* per assignment instructions.

However, there weren't any customer-handling in PizzaStoreV5.1. There has been added a *Customer* and *CustomerFile* class, to handle customers and their information. This has been done using List<T>. The log has been updated to hold customer information, after each order.

Further, the *Store* class has been made to hold most methods needed to handle Console I/O.

Many access modifiers have been changed to public and many classes and methods have been made static. I'll elaborate further in the discussion.

Finally, I've added basic unit test for *CustomerFile* and *PizzaMenu*.

All in all, the current version should fulfill the criteria set for the implementation portion of the assignment.

Discussion and considerations

Generally, many code-design choices were made with two functionalities in mind. First, the program has a lot of Console I/O and this complicates the data storage, as it relies on user input meaning as little as possible should be hardcoded into the constructors, instance fields etc. Second, I needed to have some way to store data from compile time to termination. Seeing as storing data in an external file like .txt is quite possible but would impact speed and be a compromise between running a mock-server table within the IDE (VS2022) and containing the data within the solution itself, the latter was chosen to keep the complexity as low as possible. The consequence of these design choices has been keeping many access modifiers as public and many class instances, methods etc. as static to keep them available to the Console user and store the input data. It has also led to some less than beautiful ways to display data, seeing the need to store data within an object in a configuration warping its purpose. While these could be worked out and the solution itself is not necessarily complex, it would be time-consuming while not demonstrating any special or new competency nor giving me any valuable experience. Bearing in mind, that we'll soon move away from Console as "GUI", the current solution is acceptable to me.

On *Customer* and *CustomerFile* class: The assignment forced the use of a *List<T>* and wanted a search by name. I consciously disregarded the "search by name" part and gave each customer a unique Id. This is due to having multiple customers, with an increasing chance of two customers having the same name. If Anders Andersen orders pizza, it might be either Anders Andersen of Aabenraa or Anders Andersen of Aarhus, rendering the indexation meaningless. Furthermore, seeing as the whole program runs heavily on Console input, the risk of spelling mistakes, special characters (like René Niño) etc. would add unnecessary points of possible errors. Most people can flawlessly type 4 digits. I implemented the List portion, understanding the need to demonstrate ability to implement and utilize this. I would have preferred a Dictionary, which would in my mind have been a more optimal solution, using the unique customer ID as key for the customer object.

On *Store* class: I've chosen to move the main portion of the I/O handling to this class, keeping *Program* as clean as possible.

On *Tests*: Again, the problem of having the Console as GUI, would lead to writing the program as in a Console I/O and a non-Console I/O (hardcoded) version. I might be limited by my abilities, but the only solution for providing a test, that didn't entail a reviewer to try all possible and impossible combinations in the Console, I went for a very basic unit test to automate the testing procedure. The tests only run on *PizzaMenu* and *CustomerFile*, per the assignment's criteria. The test uses the *StringWriter* and *StringReader* classes from the .NET library and Console methods, which have not been a part of the curriculum so far, but I deem them simple enough to be acceptable. A curious problem has occurred though, since it seems that the line break provided by *Console.WriteLine()* is not the same as the usual "\n", at least not interpreted by *Assert.AreEqual()*. While a little refactoring and string structurization solved the problem, it consumed too much time, comparing and writing "string text" instead of writing real code, so I kept the test limited to the two mandatory classes, i.e., a choice of quality over quantity/volume.

My considerations for the diagrams will fall in with the diagrams themselves.

While implementing as much as possible of the curriculum so far, there has not been a reasonable opportunity to implement either interfaces or inherited classes. Exceptions have been handled in the manipulation of user input, but has been implemented in *Pizza*, when giving parameters for an instance.

UML II

The assignment is stated as such:

1. User Stories	Update your Product Backlog with new or updated User Stories. The Product Backlog should contain US about orders, customers and more: ● Add Acceptance Criteria
2. UML I – Domain and Design Class Diagram	Update your diagrams from the assignment UML 1 regarding the pizza menu, orders, and customers.
3. UML II – Sequence Diagram	Create a sequence diagram for, what happened when you call the SearchPizza method.
4. Implementation	Implement the part of the system related to the customer administration (using a collection of type List) and to the pizza menu administration (using a collection of type Dictionary).

This version of the system should include:
Customer administration (to be implemented)

- **The system should be able to make a list of Customers**
- **The system should be able to create a customer, delete a customer and update a customer.**
- **The system should be able to search for a Customer based on name.**

Pizza menu administration (to be implemented)

- **The system should be able to create a Pizza, delete a Pizza and update a Pizza.**
- **The system should be able to print out on the screen a list of Pizzas.**
- **The system should be able to search for a Pizza based on relevant and given criteria (Pizza no).**

User dialog (extra)

- ***The system should be able to print out on the screen a menu***
- ***The system should handle errors in the input.***

User Stories, Acceptance Criteria and Backlog

We left the last iteration with the following user stories:

User stories:

1. **The manager wants the customer to be able to order a pizza online, with the ability to customize the toppings to streamline and improve customer experience.**
2. **The manager wants to have an overview and ability to retrieve data from orders iot. ease accounting.**
3. **The manager wants to get statistics over orders iot. min/max profit and gauge future procurement of raw materials (e.g., amount of mushrooms).**
4. **The manager wants the customer to have the ability to choose between eat-in, takeaway and delivery.**
5. **The manager wants the pizza-order system to generate a unique ID pr order, iot. track orders**
6. **The manager wants to have the customers be able to pay via "Mobile Pay" or Credit Card and have a "paid/not paid" displayed on the order.**
7. **The pizzamakers want to view paid orders and have them prioritized on a first-come-first-serve basis iot. bring down from-order-to-table time.**
8. **The staff wants to be able to enter orders for customers who want to phone in their orders or order in the restaurant iot. have all orders within the system.**
9. **The staff wants to be able to review orders iot. verify e.g., abnormal order (10 times anchovies on a veggie pizza) or ask for payment.**
10. **The staff wants to be able to review customer info iot. plan for deliveries either to table, over the counter or customer address.**

We sequenced the user stories to something a bit more manageable and ended with the following nine user stories.

The "PizzaStoreV5" program can, at current iteration complete the following tasks:

1. **The order system must be able to display the menu.**
2. **The order system must be able to take user input for an order with one or more pizzas.**
3. **The order system must be able to take user input for extra topping.**
4. **The order system must be able to take input from #2 and #3 and generate a price.**
5. **The order system must be able to assign a unique ID to each order.**
6. **The order system must be able to display the day's revenue when prompted.**

Leading to the following backlog:

7. **The order must be able to hold basic customer information given by staff.**
8. **The order must be able to change property from "not paid" to "paid."**
9. **The order must be able to change property from "not ready" to "ready."**

Taking the assignment into account and knowing what code has been implemented, an updated version of the sequenced User Stories ends up as

#	User Story	Acceptance Criteria
1	The Store must be able to display the menu.	A. The menu should be visible on the store interface. B. All menu items should be listed with their respective prices.
2	The Store must be able to take user input for an order with one or more pizzas.	C. The system must allow users to input one or more pizzas for an order. D. Each pizza selected should be clearly listed in the order summary.
3	The Store must be able to take user input for extra topping.	E. Users should have the option to add extra toppings to their selected pizzas. F. The system should display the additional cost for each extra topping. G. Users should receive confirmation of added extra toppings.
4	The Store must be able to take input from #2 and #3 and generate a price.	H. The system should calculate the total price based on the selected pizzas and extra toppings. I. The price displayed should be accurate and reflect any changes in the order (e.g., added toppings).
5	The Store must be able to assign a unique ID to each order.	J. Every order should be assigned a unique ID. K. The ID should be displayed in the order confirmation and be retrievable for reference in the log.
6	The Store must be able to display the day's revenue when prompted.	L. The system should provide an option to display the total revenue for the current day. M. The displayed revenue should accurately reflect all paid orders made on that day.
7	The Store must be able to contain basic customer information given by staff.	N. Staff should be able to input basic customer information. O. The information should include (at least) the customer's name, address (including postal code and city), phone number.
8	The Store must be able to search and if found, display a customer's information	P. The system should allow staff to search for a customer's information using a unique identifier (i.e., customer ID). Q. If found, the customer's information should be displayed clearly and comprehensively.

9	**The Store must be able to search the Menu for a pizza and if found, display the pizza**	**R. The system should enable users to search for a specific pizza on the menu.** **S. The system should be able to search for either number, name, or price – all pizzas with the same price should be displayed.** **T. If found, the pizza details (number, name, price) should be displayed.**
10	**The Store must be able to change an order property from “not paid” to “paid.”**	**U. The system should allow staff to change the payment status of an order from "not paid" to "paid."** **V. A clear confirmation should be provided upon successful payment status update.**
11	**The Store must be able to change an order property from “not ready” to “ready.”**	**W. Staff should be able to update the status of an order from "not ready" to "ready."** **X. The system should confirm the successful change in the order status.**

Reviewing the user stories and their acceptance criteria(s), US #1-9 will be marked as -Done- while the updated backlog will be:

Big Mamma Pizzeria – Backlog 12. Nov 2023		
1	**The Store must be able to change an order property from “not paid” to “paid.”**	**A. The system should allow staff to change the payment status of an order from "not paid" to "paid."** **B. A clear confirmation should be provided upon successful payment status update.**
2	**The Store must be able to change an order property from “not ready” to “ready.”**	**C. Staff should be able to update the status of an order from "not ready" to "ready."** **D. The system should confirm the successful change in the order status.**

Domain System Diagram

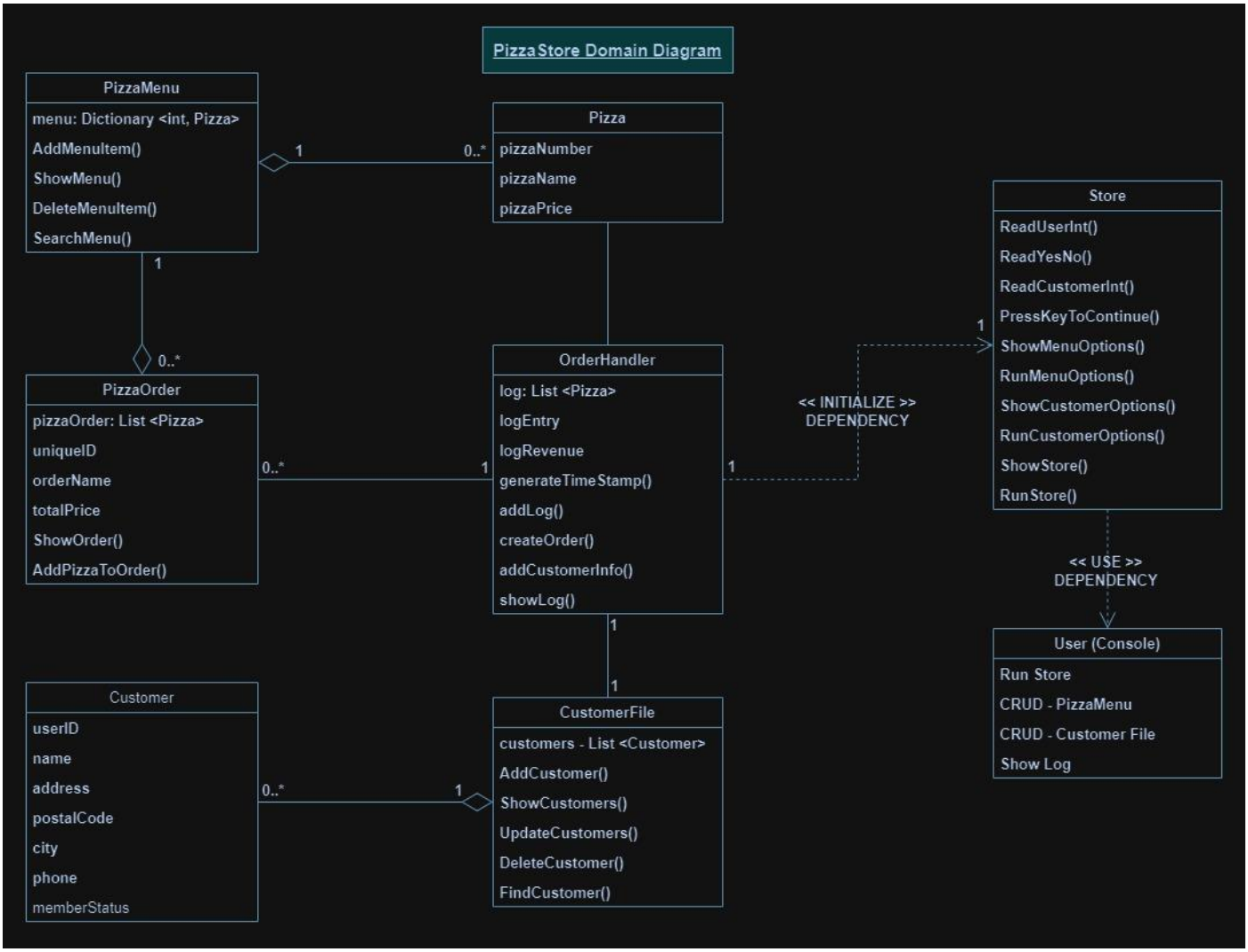

Figure 1. Domain System Diagram

As the Abstract described, the biggest change from the last iteration is the added classes *Customer*, *CustomerFile* and *Store*, changes internally to *PizzaMenu* and *OrderHandler*, while the systems overall internal associations etc. have had a slight change.

Domain Class Diagram

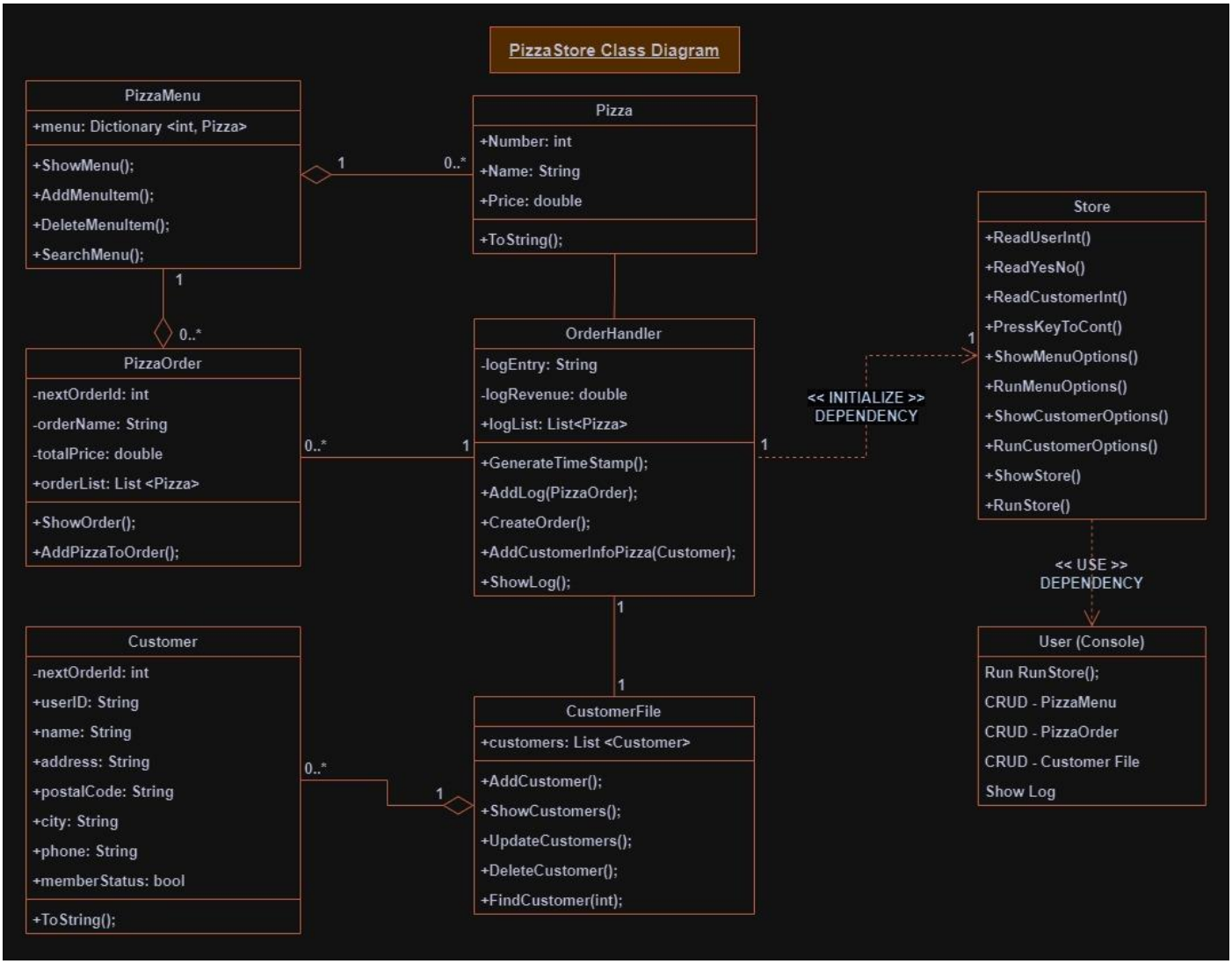

Figure 2. Domian Class Diagram

As stated in the discussion and considerations, a – terror inducing – large part of the attributes is public to be used by *Store*. This choice is made from necessity and not out of carelessness.

The relationship between *Pizza* – *PizzaMenu* and *Customer* – *CustomerFile* is set as an aggregation because a pizza menu without pizzas is possible, but useless. *PizzaMenu* and *CustomerFile* will be created during compilation along with *Store* and *OrderHandler*, so those four will exist throughout runtime. This is needed as the data they hold is crucial to the overall functioning of the program.

The relationship *PizzaMenu* – *PizzaOrder* is also set as an aggregation, because *PizzaOrder.AddPizzaToOrder()* needs input from *PizzaMenu* to fill *orderList* with *Pizza* objects. A *PizzaOrder* object can be created and exist without the existence of *PizzaMenu*, however it is useless. The dependency between *User (Console)* – *Store* is because *Store* needs to be started by the User and acts as a very rudimental GUI. The dependency between *Store* – *OrderHandler* is more symbolic and should be interpreted as relationship between "GUI" (*Store*) and the rest of the system. One *Store* can initialize one "system".

System Sequence Diagram

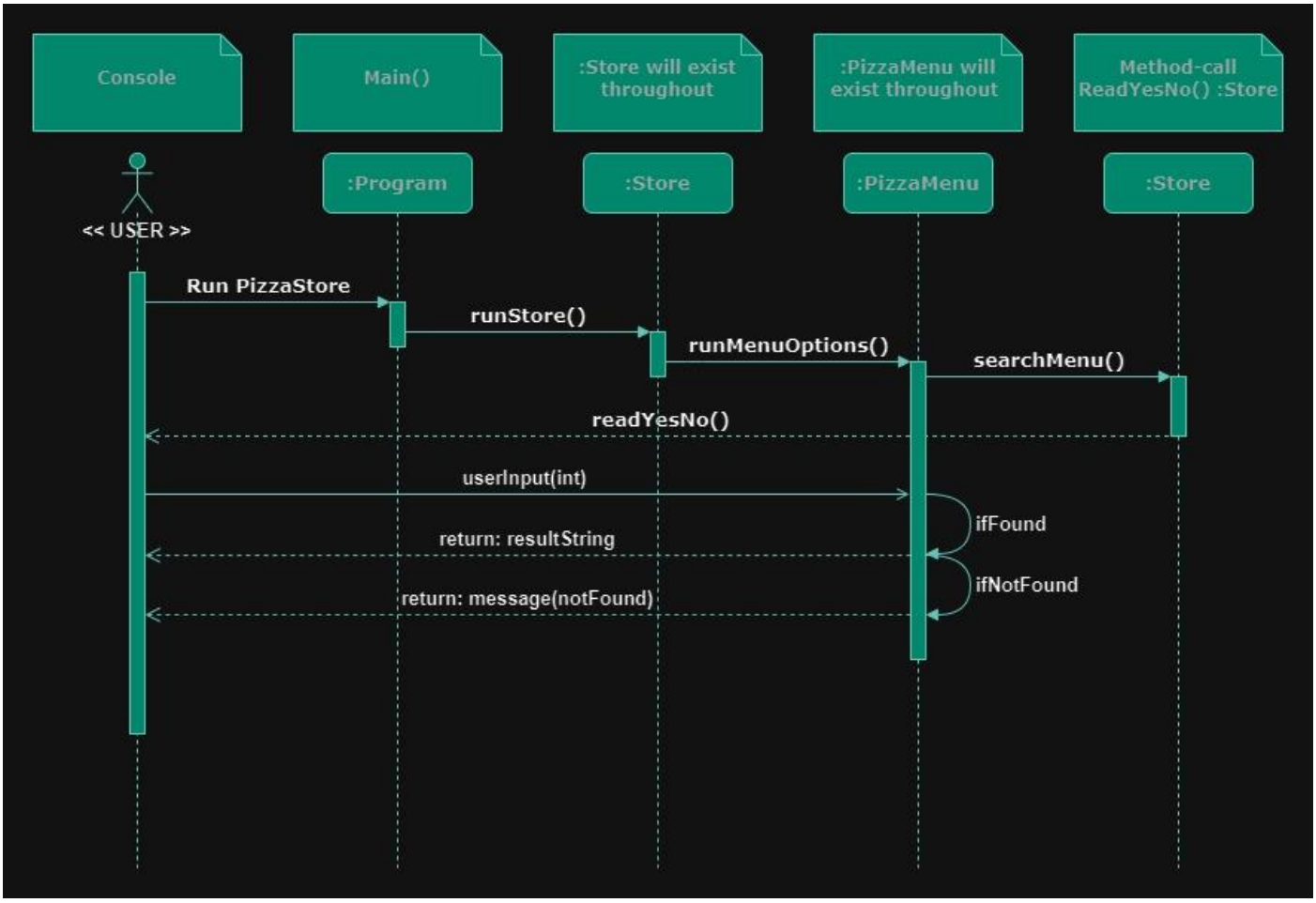

Figure 3. System Sequence Diagram

The SSD should be self-explanatory given its relative simplicity. The user runs *Program*, *Program* calls *Store.RunStore()*, which in turn (after user has elected the menu options) calls *Store.RunMenuOptions()*. After this, the user can choose "2. Search menu" option, which invokes *PizzaMenu.SearchPizza()*. The method itself invokes *Store.ReadYesNo()*, which takes the users input, puts it into lowercase and returns it as datatype String – it will return an empty string (instead of null) if user just presses enter. If the string is null or empty the *SearchMenu()* will display an error message. Else, it will loop through the dictionary, taking each *Pizza* and puts each object's state into variables and compares it to the search input, triggering an *if* statement if such condition returns true, displaying the found *Pizza* objects using its *ToString()* method. Otherwise, it will return a "not found" message to Console.

I've chosen the notations for the instance lengths, to only be the time needed to interact with the next step. I've chosen to display two instances of *Store*, since the first is the "main" menu and the second is a "submenu", even though *Store* will run from compilation to termination.