2020

Software Architectures and Design

SWE30003

LUKE MCWHA

Table of Contents

Please view source code for this project at: https://github.com/LukeMcwha/SWE30003

1. Design Changes

Changes to the project design have been minimal from the design to implementation. Many additions are to incorporate a user interface through a Command Processor. This allows a user to input commands to manipulate the ViewModel layer. Different user views are provided through the State Pattern. Different commands are allocated to each State creating different functionality between users.

For example, the customer can see the menu and their current order. They have commands that allow them to add or remove menu items from the order and commands that will allow them to confirm or pay for their order.

The kitchen marks an order as complete and moves the completed order back to the customer and another command to allow them to view all orders they need to make. The FOH can view completed orders and orders that have taken too long to make.

The FOH has commands to settle invoices with a customer, move reservations to a table and view order statistics

Below will demonstrate the structure of the View, Model and ViewModel. The View and Model are new and have not been provided in previous documents. A discussion on the ViewModel will be provided outlining design changes.

1.1 View Design

The design decision for this UI is a command processor. This is a non GUI interpretation of a UI where the user types commands to interact with the program.

See below the design UML documentation for this pattern.

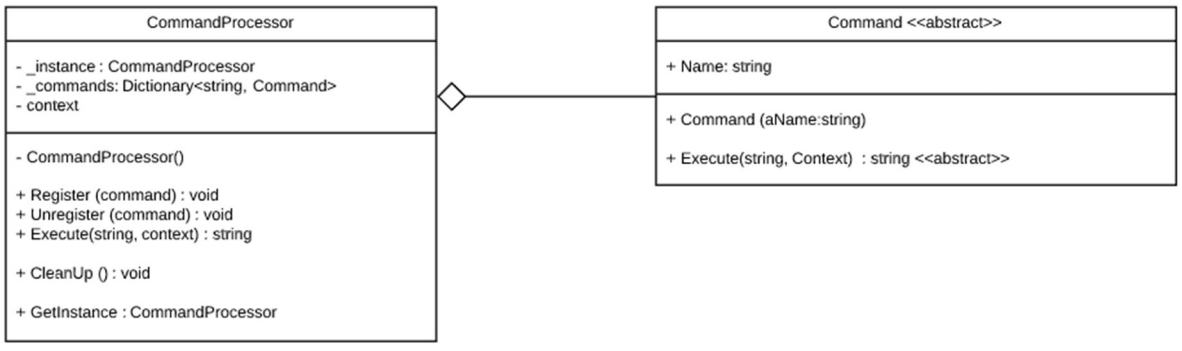


Figure 1. Command Processor Pattern and Command Pattern

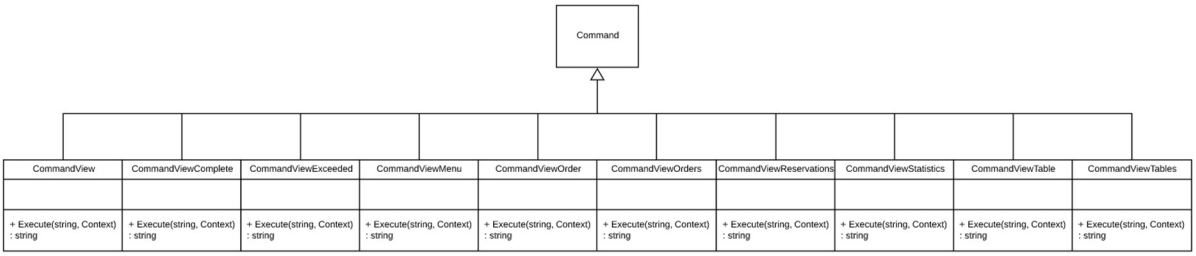


Figure 2. Command Pattern with Inherit Commands

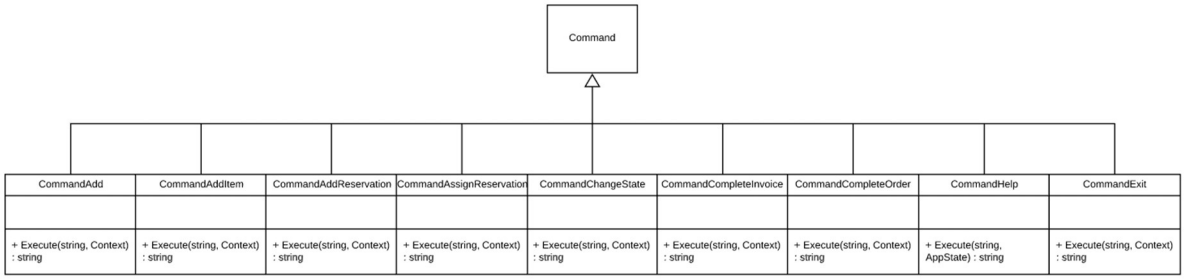


Figure 3. Command Pattern with Inherit Commands

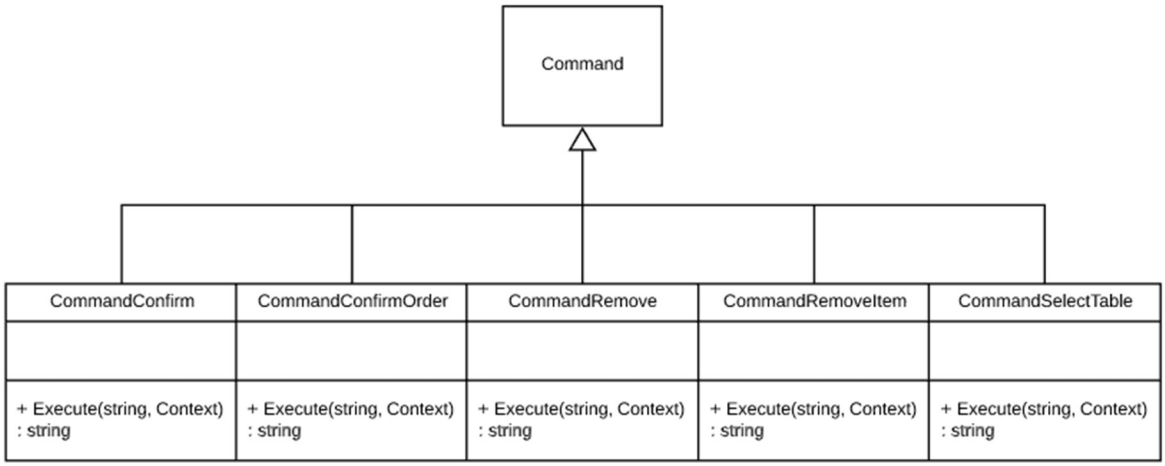


Figure 4. Command Pattern with Inherit Commands

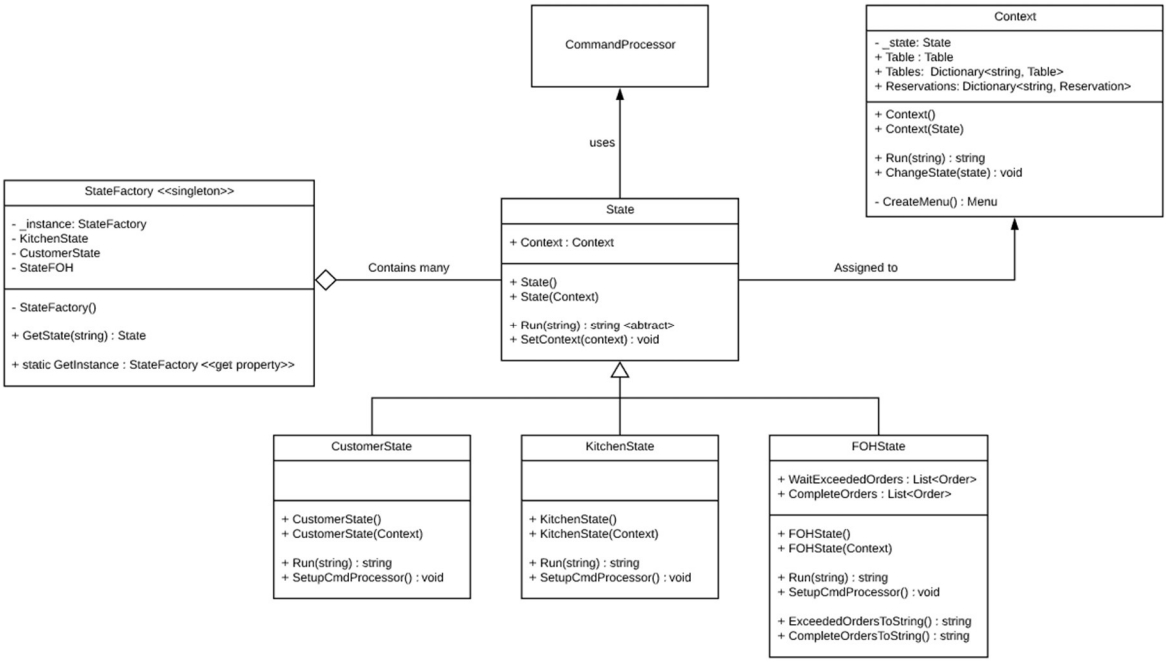


Figure 5. State Pattern and Factory Pattern in relation to a Command Processor

1.2. Model Design

The Model Design is small in relation to the project. As there are simplifications made to the use of a database within the implementation of the project there is also a simplified database model for data.

Accessing the data is by the _id field. This is a unique identifier given to each entry to the database.

Reservations for the future would be entered into the database and through various API queries the future reservations could be viewed or modified. Simplifications have been made and only reservations made within the app session can be viewed.

Invoices when paid for would be pushed to the database to record them as being complete. This is all being stored within the app session and database service.

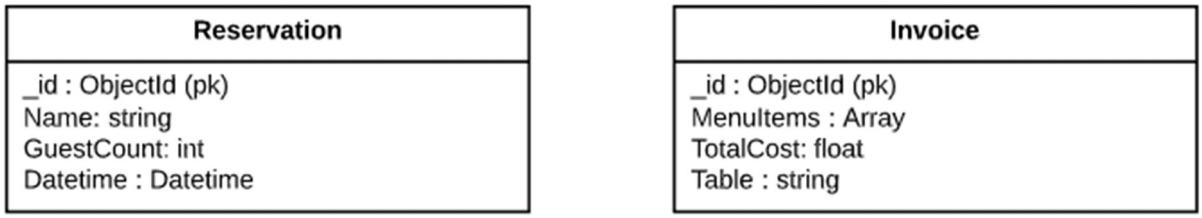


Figure 5. Database Model

1.3. View Model design

The View Model design development in assignment 2 great to develop off and only required a few small tweaks for the implementation. Please see assignment 2 appended to this assignment for the previous UML model.

The main issues I found was the over engineering within the Event dispatcher pattern. Subscribers such as ReservationSubscriber and OrderMutatedSubscriber added a layer of complexity that was not required. Therefore, these classes were removed from the model.

The addition of a 'Context' class within the State pattern created a great center for data within the app. A list of reservations and tables make an easy access point for this data. This class also contains the currently selected Table in the app.

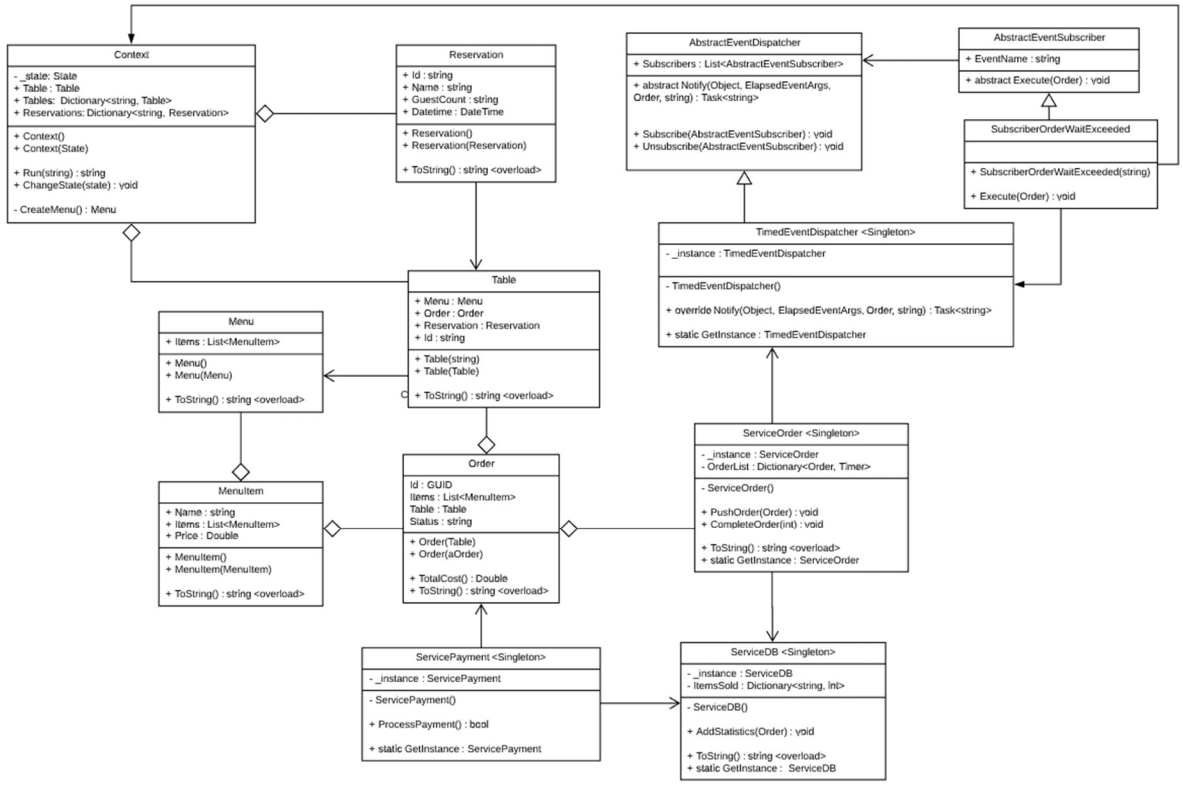


Figure 6. View Model UML Model

2. Responsibilities Changes

Many of the responsibilities have stayed the same but in some cases the responsibilities have changed. Instead of the TimedEventDispatcher holding the orders and timers to dispatch the OrderWaitExceeded event, the OrderService holds this information and calls the Dispatcher. The dispatcher then informs all its subscribers.

The Context class was added to the project and plays the role of being the 'Cafe'. Its role is to hold the Tables data and Reservation data for the Cafe which can then be accessed by the other objects.

In terms of the bootstrap process, the Main of the program creates the Context and the Context acts as the 'Main' in the old bootstrap process.

3. Changes to dynamic aspects

Adding the UI brings dynamics to the projects, therefore there is the need to incorporate additional patterns to bring this to life. There are two patterns used to create this functionality. A Command line processor to act as the main user interface with the ViewModel layer and the State Pattern to portray the differing users and commands that each user has access to.

The State Factory and State Pattern created a different layer of dynamic interaction that was not present in the old design and this changes the dynamic bootstrap process of the app. States and commands that the state allows will create or destroy varying objects.

4. Discussion of Assignment 2 Design

4.1. Good Aspects

There are many good aspects of the design for this project. Using a MVVM to be the base of our design leads to a well-structured project.

The use of singletons within the design has allowed data to be easily transferred between objects that require it. This is important within a real-time, data heavy project and has led to a simple and solid implementation.

The use of a Event Dispatcher Pattern implemented seamlessly for tracking timed events such as the Order Wait Exceeded event. This is the best pattern to implement this functionality well.

4.2. Missing from original design

The only class missing from this design was a 'Context' or 'Cafe' class. This class holds tables and reservations needed for the app. The implementation of this has led to good object interaction and data transfer that would require API calls to do otherwise.

The UI was missing from the original design, or more specifically "Interaction" methods/ classes that a UI would use. I have implemented a Command Processor within a CLI to act as my User Interface. This Command Processor could easily be moved to have UI based triggers but due to time constraints the development has been within a CLI.

4.3. Flawed Aspects

No aspects of the previous design were flawed. Although, in some areas it was better to implement a different way then the designed way. In the design there is an Event Dispatcher for Reservations which creates an event whenever a reservation is meant to start. This design isn't flawed as it works and could be implemented.

The simpler way to execute this functionality is to display future reservations within the FOH state. The user can then visually see when a reservation is meant to occur. This is the best way as it is unnecessary to have events or notifications when a reservation begins.

5. Lessons learnt

Lessons learnt over this project is around over engineering and structural design patterns. The incorporation of the MVVM pattern has demonstrated a clean approach to build data heavy applications for both the front and back end.

The Event Dispatcher pattern is an incredibly useful pattern for triggering functionality on events. The implementation is simple and clean. The lesson is that not every problem can be developed using this pattern.

6. Demonstrations

6.1. Home Screen

```
----------------------------------------
---- Welcome to the cafe ordering app ----
Please type 'help' for a list of commands

----------------------------------------
:> help
Welcome to the help pannel, see below the commands available in this state.

------------------------
       view    :: Generic View command. Syntax is ( view < menu | order > )
       add     :: Adds item to container. Syntax: add < item > < number >
       remove  :: Removes specified item from a container. Syntax: remove < container > < index >
       confirm         :: Runs child process depending on the command.
       state   :: This command will transition the users perspective from one view to another. eg Move from Customer State to Kitchen State.
       select  :: Select a table that you wish to take the view of.
       exit    :: This command will exit the program.
------------------------


:>
```

6.2. Exit Screen

```
:> exit
Exit
Thanks for using the app
Press any key to continue . . .
```

6.3. View Screens

Menu:

```
:> view menu

1. Cheese Burger : $ 15
2. Pizza : $ 17
3. Avocado on Toast : $ 13
4. Coke : $ 4
5. Nachos : $ 13.5

:>
```

Tables:

```
:> view tables

Table No. 1 :: Current reservation: None :: Order Status: Creating Order
Table No. 2 :: Current reservation: None :: Order Status: Creating Order
Table No. 3 :: Current reservation: None :: Order Status: Creating Order
Table No. 4 :: Current reservation: None :: Order Status: Creating Order
Table No. 5 :: Current reservation: None :: Order Status: Creating Order
Table No. 6 :: Current reservation: None :: Order Status: Creating Order
Table No. 7 :: Current reservation: None :: Order Status: Creating Order
Table No. 8 :: Current reservation: None :: Order Status: Creating Order
Table No. 9 :: Current reservation: None :: Order Status: Creating Order

:>
```

Reservations:

```
Reservations ----
1 :: Name. Luke Mcwha :: Guests No. 4 :: Time: 23/06/2020 12:30:00 PM

:>
```

6.4. Scenarios

6.4.1. Customer creating a reservation

Correct Data:

```
Welcome to this cafe ordering app
Please type 'help' for a list of commands

:> add reservation
Name: Luke Mcwha
Number of Guests: 3
Reservation Date:
Day (eg 7): 7
Month (eg May = 5): 5
Year (eg 2020): 2020
Hour (eg 5pm = 17): 17
Minute (eg 5:30pm = 30): 30

Reservation Details: 1 :: Name. Luke Mcwha :: Guests No. 3 :: Time: 7/05/2020 5:30:00 PM

:>
```

Incorrect Data:

```
:> add reservation
Name:
Number of Guests:
Reservation Date:
Day (eg 7):
Month (eg May = 5):
Year (eg 2020):
Hour (eg 5pm = 17):
Minute (eg 5:30pm = 30):

Invalid Command Input.

:>
```

6.4.2. Kitchen Viewing and Completing an Order

```
:> view orders

---- Kitchen Orders
f920bf42-a988-4cb4-8845-e0750033e502 -- Table No: 1 -- Current Wait Time:  seconds


:> complete 1

Order is complete and needs to be taken to the customer.

:>
```

6.4.3. Kitchen Declaring a “Wait Time Exceeded” Event

After an allocated amount of time, it is declared that the wait time has exceeded.

```
:> confirm order

Your order has been confirmed and is being sent to the kitchen.

:> Order d18a0775-7ff2-42fc-98e7-077008251914 has exceeded the wait time limit at 0 and must be compensated.
```

6.4.4. Customer Adding/ Removing Items from their order and confirming order to the Kitchen

```
:> view menu

1. Cheese Burger : $ 15
2. Pizza : $ 17
3. Avocado on Toast : $ 13
4. Coke : $ 4
5. Nachos : $ 13.5

:> add item 1

Order Item 1: Pizza has been added to the order.

:> add item 2

Order Item 2: Avocado on Toast has been added to the order.

:> view order

----- ORDER
1. Pizza : $ 17
2. Avocado on Toast : $ 13

Total: $30
-----

:> remove item 2

Order Item 2 has been removed from the order.

:> view order

----- ORDER
1. Pizza : $ 17

Total: $17
-----

:> confirm order

Your order has been confirmed and is being sent to the kitchen.

:>
```

6.4.5. Customer paying for an order

Valid data:

```
:> complete 1

Table 5 Order comes to 30.
Confirm (y/n):> y
Paying Invoice...
Order has been paid for in full

:>
```

Invalid data:

```
:> complete 1

Table 5 Order comes to 0.
Confirm (y/n):> [
Order has not been paid for.

:>
```

6.4.6. FOH Viewing Wait time exceeded orders and complete orders

```
:> view exceeded

3519b73b-769f-4d5e-9007-ad4678cac022 -- Table No: 5


:> view complete

f920bf42-a988-4cb4-8845-e0750033e502 -- Table No: 1


:>
```

6.4.7. FOH View Statistics

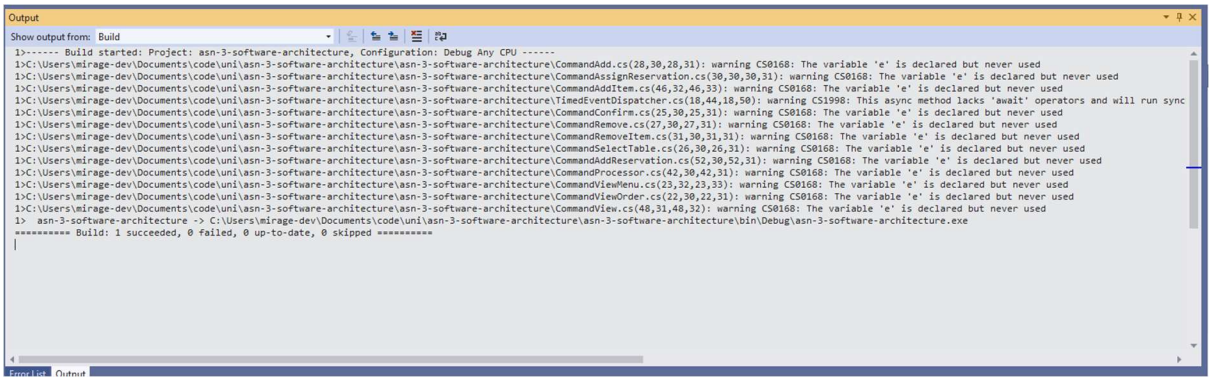
```
:> view stats

----- Statistics
Pizza :: 1 sold.
Avocado on Toast :: 1 sold.


:>
```

6.5. Development Environment:

- C# .NET Framwork 4.7.2
- Visual Studio Community 2019
- Windows 10 x64

6.6. Explicit Evidence of Compilation

Screenshot of build output.



Previous Assignment

1. Executive Summary

When expanding a business, improving the processes of that business is a must to make it cost effective and sustainable into the future. In expanding the number of customers the client is wanting to cater to means they can expand their business into the future. Tasks such as waiting tables, passing orders to the kitchen and having customers pay for their order need to be efficient to get the most out of the business and staff.

The solution is to have a piece of software that will automate many of these tasks without the need for staff intervention. Through this software, time will be saved, the number of orders will increase and order/ transaction mistakes are reduced. In the long run, the business will save more money on food by making better informed decisions when ordering stock through the statistics gathered; save on staff wages as the software can take orders and create invoices that would otherwise need human intervention to complete.

The software is being designed in a modular fashion. The use of an Event Dispatcher creates a simple and well defined interface for internal communication throughout the app. Within the current scope this is extremely useful in notifying the FOH staff when an order has taken too long to be complete. This architecture allows for the software to be extended with new features in the future due to the generic nature of the pattern.

The belief is that this solution presented will greatly improve the workflows of this cafe and help encourage growth into the future.

2. Problem Analysis

The Software Requirements Specification outlined various points, views and requirements for the project in both function and quality that must be adhered to in the planning and development phase. The key functional points described within that document encapsulate the end goal and use cases for the project.

These key functional points are outlined below:

- Allow a customer to place a reservation online
- Allow a customer to order
- Display the menu online for customers to view
- Notify and give details of a customer's order to the kitchen
- Notify front of house staff whether a customer has waited too long for their order
- Create an invoice for a table’s order and receipt for payment
- Handle monetary transactions between a customer and the cafe
- Logging order and payment statistics and data to a database

2.1. Assumptions

The following are assumptions made when designing the project:

A1 All servers / equipment required is available.

A2 3rd party software used for the transaction process is integrated with the project.

A3 3rd party accounting software can process split payments.

A4 System will not have more than 150 concurrent users.

A5 All menu items will always be available.

A6 Admin users will create accounts.

A7 Frontend (UI) has interaction functions to modify the ViewModel.

A8 Orders can not be made online.

A9 Orders will have a timer allocated to them on creation.

A10 Payment Service will deal with an external payment entity.

A11 Database service will interact with a sole database.

A12 When an object with a dispatcher attached, the dispatcher will notify all subscribers of an event.

A13 There will only be one instance of the Menu.

A14 Menu items can be changed.

A15 Reservations can be made by a customer online or in person.

2.2. Simplifications

Based on assumptions made in Section 2.2,

- The external payment services are assumed to be plug and play, we assume no matter the actual product to process the payments, it will interface the same way.
- The database is assumed to be plug and play, no matter the database type used, it will interface the same way.
- The system is loosely built as a Model View Viewmodel architecture. Our design does not include classes for the View as this model will work regardless of the frontend framework used.

2.3. Design Justification

It is assumed that the software will contain a database and it is specified that varying services will interface with that database. In the same way that the external banking software exists and will interface with the project through varying services.

Below outlines the various patterns and classes that were considered but we decided not to incorporate into the solution. The final classes are outlined and justified further in the document. The design is created to make a flexible environment for future growth and simply the objects we use.

2.4. Patterns Considered

2.5.1. Command Pattern

The Command pattern is a behavioural design pattern where command objects will modify the behaviour of other objects. For example, if an object or the user through a user interface invokes the ‘Create Reservation’ command. This command would trigger processes by the required objects to create a reservation.

Instead of this design we opted for an Event Dispatch pattern to handle internal communication between objects upon an event and we assume the User Interface will interface with objects themselves to invoke their behaviour.

2.5. Discarded Classes

2.5.1. Application Session

The application session class is designed to be a central point for data regarding the overall session of the app. This could contain user data, order data, table data for the app.

We removed this from our design as it was a God class if implemented. The usage of singletons for major container classes erased the need for a single access point.

2.5.2. Menu Components

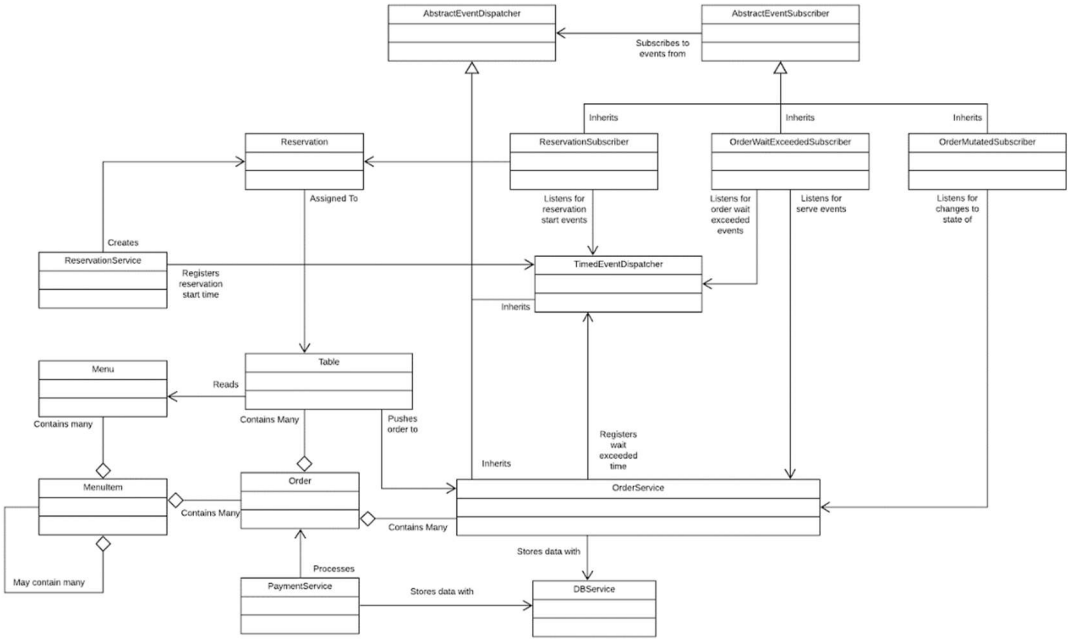
Initially, due to different instances of the application requiring different usage permissions in the menu (Web portal displaying the menu, In-restaurant devices displaying and allowing orders, admin portal allowing editing) we considered adding 'permission' components. However we decided that this was an unnecessary complication and could be implemented in other ways.

3. Candidate Classes

3.1. Class List

- Menu
- MenuItem
- ApplicationSession
- Table
- Order
- PaymentService
- DatabaseService
- OrderService
- AbstractEventSubscriber
- AbstractEventDispatcher
- TimedEventDispatcher
- OrderMutatedSubscriber
- OrderWaitExceededSubscriber
- ReservationSubscriber
- Reservation

3.2. Class Diagram



3.3. CRC Cards

3.3.1. Menu

The singleton menu provides access to a single unified list of menu items for viewing, ordering, creating, and editing.

Component: Menu	
Sub-Classes: n/a	**Super Classes: n/a**
Responsibilities	**Collaborators**
Record items within the menu.	MenuItem
Display items from within the menu.	MenuItem
Provide interface for editing items within the menu.	MenuItem

3.3.2. MenuItem

Encapsulates the data associated with a singular item orderable within the restaurant.

Component: MenuItem	
Sub-Classes: n/a	**Super Classes: n/a**
Responsibilities	**Collaborators**
Records name, ingredients, price, availability etc. of a food, drink or other orderable item.	n/a

3.3.3. Table

Encapsulates the details associated with a physical table in the restaurant.

Component: Table	
Sub-Classes: n/a	**Super Classes: n/a**
Responsibilities	**Collaborators**
Create anew, or change existing associated orders of items from the menu.	Menu, MenuItem, Order, OrderService
Record any associated reservations.	Reservation
Record any created orders.	Order
Determine aggregated payment details of all recorded orders.	Order

3.3.4. Order

An Order object encapsulates a single order of menu items made by customers at a table.

Component: Order	
Sub-Classes: n/a	**Super Classes: n/a**
Responsibilities	**Collaborators**
Record a list of ordered MenuItems.	MenuItem
Determine aggregated payment details of recorded MenuItems.	MenuItem

3.3.5. PaymentService

Provides an interface for handling payment operations, involving; preparing invoices and receipts, using external payment services, and recording all payment interactions persistently.

Component:	
Sub-Classes: n/a	**Super Classes: n/a**
Responsibilities	**Collaborators**
Create an invoice from a table's orders' payment details.	Table, Order
Process payment for a table's orders through an external service.	Table, Order
Create a receipt for a completed payment.	Order
Log completed payment details to persistent database.	DatabaseService

3.3.6. DatabaseService

Provides an interface for maintaining a connection to the application's database, and the associated CRUD operations.

Component: DatabaseService	
Sub-Classes: n/a	**Super Classes: n/a**
Responsibilities	**Collaborators**
Manage connection instance to application database.	n/a
Provide interface for storage and retrieval of data.	n/a

3.3.7. OrderService

The singleton OrderService provides a unified point of truth regarding operations upon pending orders. This not only includes a managed CRUD interface, but the firing of related events to the relevant parts of the application.

Component: OrderService	
Sub-Classes: n/a	**Super Classes: AbstractEventDispatcher**
Responsibilities	**Collaborators**
Hold a weak queue of orders currently being processed by the system	Order
Provide an interface for the creation, mutation, and finalisation of orders.	n/a
Notify subscribers of the creation and mutation of orders.	OrderMutatedSubscriber
Notify subscribers of the finalisation (serving) of orders.	OrderWaitExceededSubscriber
Register timed schedules (waiting limits) upon creation of new orders.	TimedEventDispatcher
Log statistics of orders passed through the queue in the application database	DatabaseService

3.3.8. AbstractEventSubscriber

Denotes an object that listens for notification of the firing of a certain type of event, and reacts in part.

Major base component of Event Listener pattern.

In the context of this application, instances of subclasses of this class logically create user roles, as each application instance will listen for different system events.

Component: AbstractEventSubscriber

Sub-Classes: OrderMutatedSubscriber, OrderWaitExceededSubscriber, ReservationSubscriber	Super Classes: n/a
Responsibilities	**Collaborators**
Subscribe to be notified upon the triggering of a certain type of event.	AbstractEventDispatcher
Listen for, and respond to notifications of subscribed events.	AbstractEventDispatcher

3.3.9. AbstractEventDispatcher

Denotes an object that notifies a number of subscriber objects of the triggering of an event, either in response to external stimulus or from an internal process.

Major base component of Event Listener pattern.

Component: AbstractEventDispatcher	
Sub-Classes: OrderService, TimedEventDispatcher	**Super Classes: n/a**
Responsibilities	**Collaborators**
Keep record of subscribers	AbstractEventSubscriber
Notify subscribers of event fire.	AbstractEventSubscriber

3.3.10. OrderMutatedSubscriber

Notifies the kitchen that a new order has been created, or an existing order has been changed.

Component: OrderMutatedSubscriber	
Sub-Classes: n/a	**Super Classes: AbstractEventSubscriber**
Responsibilities	**Collaborators**
Listen for creation or mutation events for orders currently in the order queue.	OrderService
Update GUI with new order state.	Order

3.3.11. OrderWaitExceededSubscriber

Notifies FOH staff in the case that a table has made an order, and too long a time has elapsed without the order being finalised (served). Allows FOH staff to adequately compensate the table.

Component: OrderWaitExceededSubscriber	
Sub-Classes: n/a	**Super Classes:**
Responsibilities	**Collaborators**
Listen for order finalisation (serving) events.	QueueService

Listen for scheduled time elapsed events.	TimedEventDispatcher
Update GUI/Notify users of wait exceeded.	Order, Table

3.3.12. ReservationService

Provides an entry point for reservations into the application from an external source. Creates reservations and registers their times to schedule a notification to front of house staff.

Component: ReservationService	
Sub-Classes: n/a	**Super Classes: n/a**
Responsibilities	**Collaborators**
Provide an external interface for reservation creation inside the application.	n/a
Create reservations.	Reservation, Table
Schedule created reservation times to notify FOH staff.	TimedEventDispatcher
Record Reservations	Reservation

3.3.13. ReservationSubscriber

Notifies front of house staff that customers with a reservation are due to arrive, and to prepare the associated table.

Component: ReservationSubscriber	
Sub-Classes: n/a	**Super Classes: AbstractEventSubscriber**
Responsibilities	**Collaborators**
Update GUI/notify user upon a reservation being due to arrive.	Reservation, Table
Listen for reservation time arrival events.	TimedEventDispatcher

3.3.14. Reservation

Encapsulates the data of a singular reservation within a given timeslot assigned to a table.

Component: Reservation	
Sub-Classes: n/a	**Super Classes: n/a**
Responsibilities	**Collaborators**
Record data associated with a reservation (ie. date & time, number of people, special dietary requirements, notes, assigned table)	Table

3.3.15. TimedEventDispatcher

Acts as an alarm clock, allowing for scheduling the triggering of events at given future times, then notifying any listeners at those times.

Component: TimedEventDispatcher	
Sub-Classes: n/a	**Super Classes: AbstractEventDispatcher**
Responsibilities	**Collaborators**
Allow interface for registering scheduled events.	n/a
Notify subscribers of scheduled time elapsing or arriving.	ReservationSubscriber, OrderWaitExceededSubscriber

4. Design Quality

4.1. Design Heuristics

H1 No 'God' classes should exist.

H2 All classes have one key goal.

H3 Abstract class must be base classes

H4 Base classes should be abstract classes

H5 Abstract classes are created as generic as possible.

H6 Generic methods are created high in the inheritance chain

H7 Concrete classes provide specific functionality based on its abstract parent class.

H8 Service classes modify only one specific aspect of the data.

H9 Only one service class of each type can exist.

H10 Each external entity must have a Service class.

H11 Code employs a consistent syntax/ structure for classes and architecture.

4.2. Design Patterns

4.2.1. Creational Patterns

4.2.1.1. Singleton Pattern

Implementing a class as a singleton ensures only one instance of that class is instantiated, and provides a universal reference to that instance.

It is implemented within classes that should only exist once globally, including:

- ReservationService
- OrderService

- Menu
- PaymentService
- DatabaseService

4.2.2. Structural Patterns

4.2.2.1. Model View ViewModel (MVVM)

MVVM splits an application into three logical structures:

Views, entailing classes related to the graphical representation of the application. These have been omitted from our design.

Models, entail classes related to the structuring and storage of data. These include:

- Reservation
- Order
- Table
- MenuItem

Finally ViewModels, entailing classes related to the interfacing and binding of models and views. These include:

- OrderService
- ReservationService

This approach allows a layered approach for modifiability, and allows a design specification which omits technology-specific graphical classes.

4.2.3. Behavioural Patterns

4.2.3.1. Observer Pattern

The observer pattern allows an arbitrary number of listeners to subscribe to an event publisher to receive notification of the triggering of events.

Publisher classes include:

- AbstractEventDispatcher
- OrderService
- TimedEventDispatcher

Subscriber services include:

- OrderMutatedSubscriber
- ReservationSubscriber
- OrderWaitExceededSubscriber

This pattern has been employed here to logically create user roles - that is, each instance of the application running on a different device can be subscribed to different system events, depending on what the user of that device needs to know about and respond to.

5. Bootstrap Process

1. *Main* instantiates **Menu**
2. **Menu** instanciates **MenuItem**s
3. *Main* instantiates *services* (**OrderService**, **DatabaseService**, **PaymentService**, **ReservationService**)
4. *Main* instantiates **Table**s
5. *Main* instantiates **TimedEventDispatcher**
6. *Main* instantiates *subscribers*, based on user roles:

a. *Kitchen*: **OrderMutatedSubscriber**

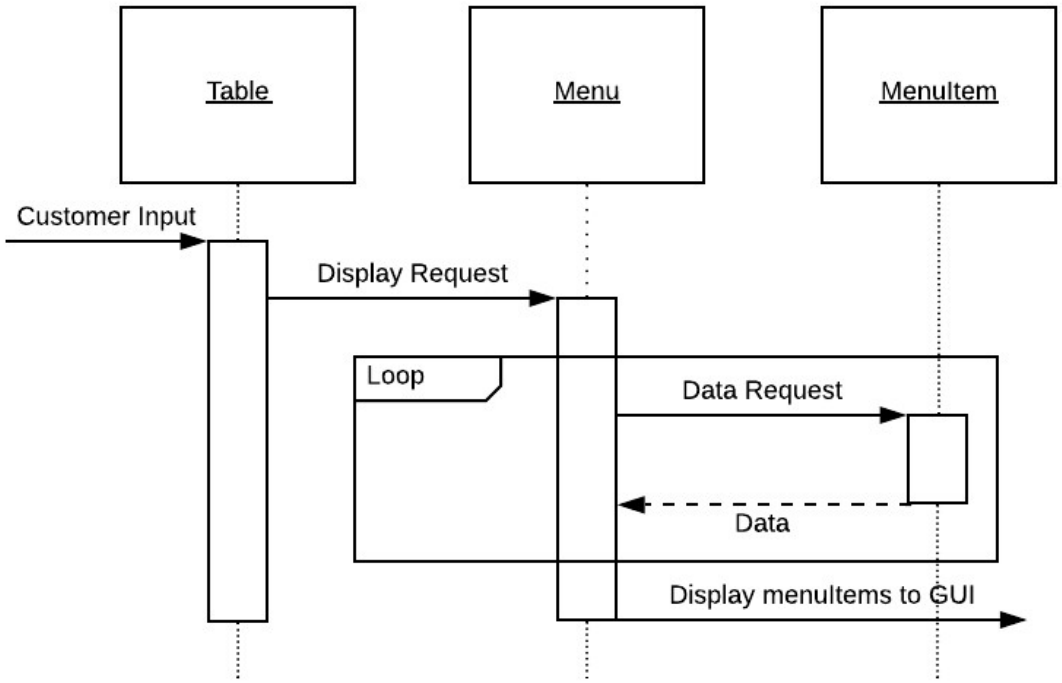
b. *FOH*: **ReservationSubscriber**, **OrderWaitExceededSubscriber**

7. Upon customer input, **Table** instantiates order and pushes to **OrderService**.
8. Upon external API input, **ReservationService** instantiates **Reservation**.

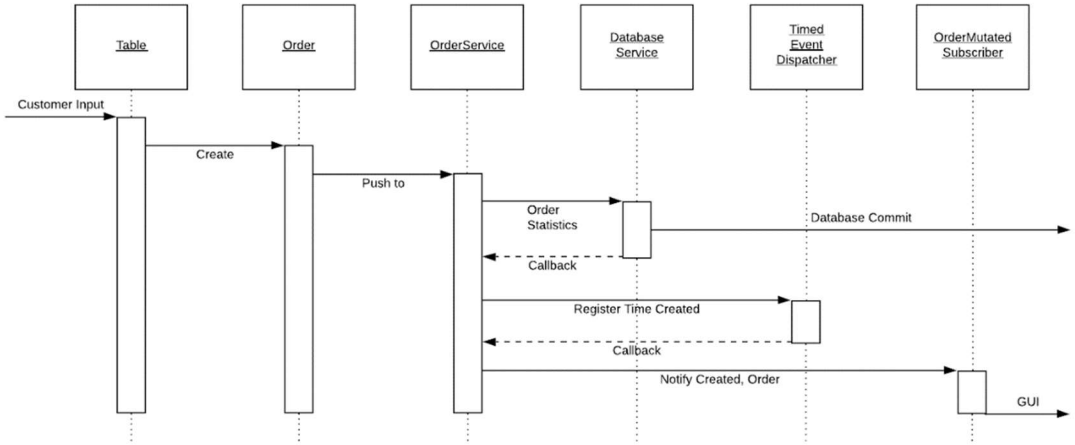
6. Verification

Below will present verification of this design through use case examples.

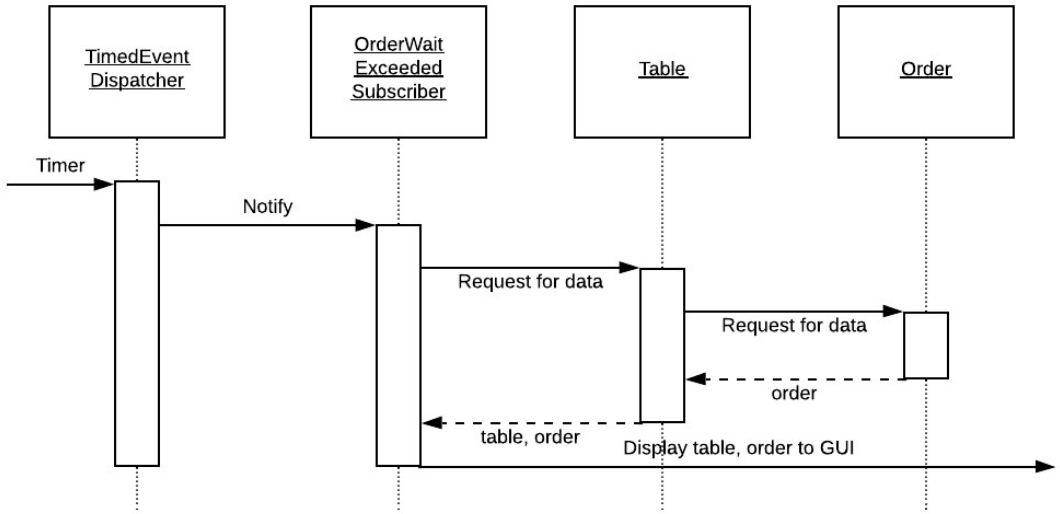
Example 1.1 - Customer views menu



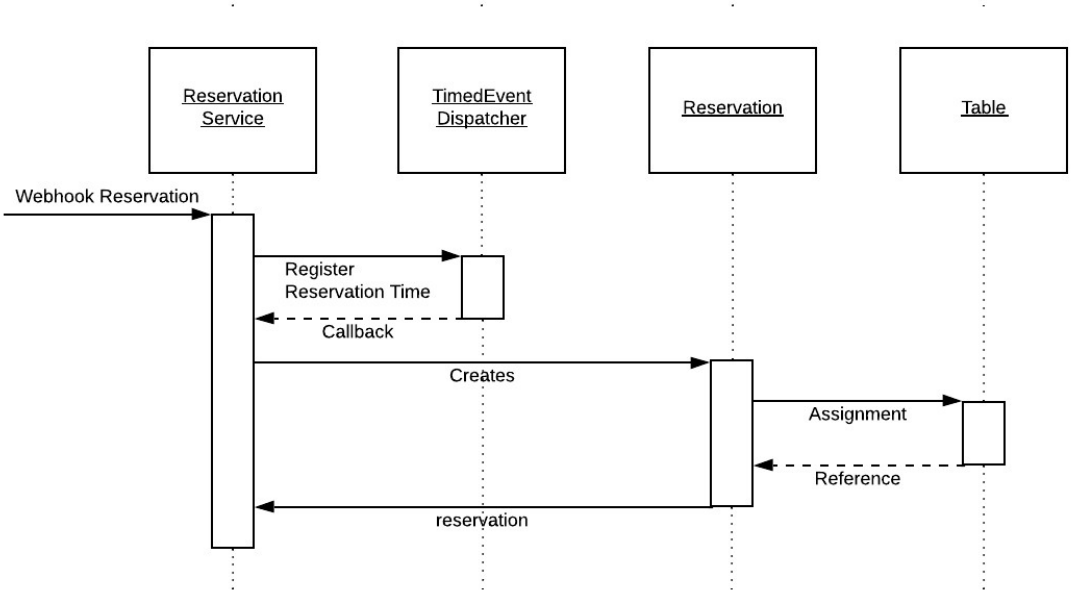
Example 1.2 - Customer makes order



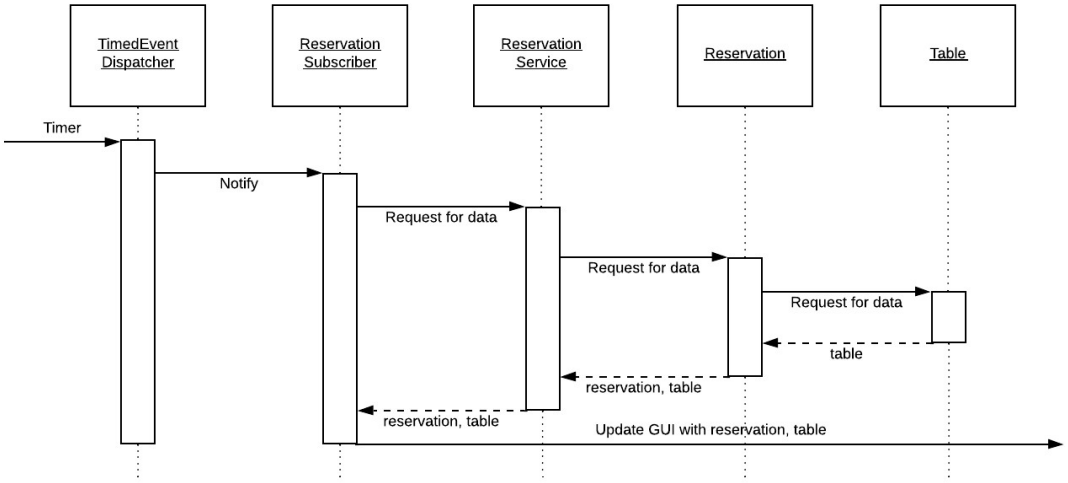
Example 1.3 - Order takes too long to be served



Example 2.1 - Customer makes reservation remotely



Example 2.2 - Reservation time arrives



Example 3 - Customer pays for order